Table des matières

Liste des figures

Liste des tableaux

Chapitre 1

Contexte et problématiques

« *The most exciting phrase to hear in science, the one that heralds the most discoveries, is not "Eureka!" (I found it!) but "That's funny..."* »

Isaac Asimov

Sommaire

1.1 Introduction

Depuis quelques années, le véhicule autonome est devenu un sujet très porteur, que ce soit au niveau de l'impact médiatique sur le grand public ou l'investissement des industriels. Pourtant le sujet n'est pas si nouveau, dès 1980 et 1990 un grand nombre de démonstrations autour du concept de la voiture autonome existait déjà. Par exemple, en octobre 1994, les véhicules VaMP et VITA-2 ont effectué un trajet en conduite autonome sur l'autoroute A1 à la vitesse de 130 km/h. Depuis les années 2010, un grand nombre d'industriels ont annoncé un intérêt dans le véhicule autonome. Cela concerne bien évidemment l'ensemble des constructeurs automobiles qui ont communiqué à ce sujet, mais également de nouveaux acteurs, jusque-là inconnu dans le monde de l'automobile, tel que Google ou Apple. Même le milieu du sport automobile s'intéresse de près à ce sujet, la FIA a annoncé en 2015 le lancement d'un championnat de véhicule autonome : Robotrace.

À l'heure actuelle, plusieurs limitations techniques et législatives font que le véhicule autonome n'est pas encore commercialisé ni accessible au grand public. Cependant, les constructeurs automobiles proposent de plus en plus de systèmes permettant d'assister le conducteur, voire d'automatiser certaines tâches : il s'agit des systèmes d'aide à la conduite. Finalement, l'industrie automobile est aujourd'hui dans un tournant : un véhicule, autrefois un engin purement mécanique, est en train de se transformer en un système complexe comprenant de plus en plus de capteurs et avec une composante logicielle de plus en plus présente. Il est donc tout à fait logique que des géants du logiciel tels que Google ou Apple s'intéressent de près au véhicule intelligent. Cette transformation amène de nouvelles problématiques, notamment un besoin de plus en plus grand en termes de puissance de calcul embarqué dans le véhicule.

Parmi les capteurs utilisés par les constructeurs automobiles, beaucoup de systèmes d'aides à la conduite utilisent des caméras et des algorithmes de traitement d'images pour percevoir l'environnement autour du véhicule. Or, ces algorithmes représentent une charge de calcul intensive à cause d'une très grande quantité de données à traiter. En effet, une caméra transmet généralement plusieurs dizaines d'images par secondes, chaque image étant composée d'environ 1 million de pixels. Pour répondre à ce besoin croissant de puissance de calcul, les fabricants de semi-conducteurs proposent depuis quelques années des systèmes embarqués à hardware mixte, ou à architectures hétérogènes. Ces systèmes intègrent sur une même puce plusieurs processeurs différents, permettant de proposer une grande flexibilité et une forte puissance de calcul. Cependant, à cause du caractère hétérogène, l'implantation d'un algorithme sur ce type de système est loin d'être triviale. De plus, du fait de la grande variété des calculateurs embarqués pour l'automobile, le choix d'une architecture de calcul pour embarquer une application donnée reste une opération très complexe pour le constructeur automobile.

1.1.1 Objectifs, contexte et contributions

Cette thèse s'inscrit dans le cadre d'une convention CIFRE et a été financée par l'entreprise Renault. L'ensemble des travaux ont été effectués au sein de l'équipe en charge des systèmes d'aides à la conduite de Renault, située au Technocentre Renault à Guyancourt. Cette thèse s'est déroulée en partenariat avec l'équipe de recherche *Méthodes et Outils pour les Signaux et Systèmes* (MOSS) du laboratoire des *Systèmes et Applications des Technologies de l'Information et de l'Énergie* (SATIE), affiliée à l'Université Paris-Sud et à l'ENS Paris-Saclay.

L'état de l'art montre une grande diversité et hétérogénéité des systèmes embarqués avec des difficultés importantes liées à leur caractérisation fiable ainsi que l'absence de maturité dans l'analyse générique et automatique de l'adéquation algorithme - architecture de calcul. L'objectif initial et principal de cette thèse était l'étude et la caractérisation générique des architectures matérielles par rapport aux demandes croissantes des algorithmes de traitement d'images intensif dans les applications des systèmes d'aide à la conduite. La thèse doit permettre la définition d'une nouvelle méthodologie d'analyse et d'évaluation de la complexité des algorithmes et des mécanismes d'évaluation robustes des performances pour une palette très large d'architectures numériques qui sont en permanente évolution. Ces enjeux sont liés étroitement aux notions de « scalabilité » des systèmes embarqués, de robustesse et de gestion dynamique des ressources pour aboutir à une démarche générique, pratique et rapide de caractérisation des architectures matérielles et des algorithmes associés destinés aux applications automobiles.

Finalement, au cours de cette thèse, nous avons défini une méthodologie globale permettant l'analyse de l'embarquabilité d'algorithmes de traitement d'images pour une exécution temps-réel. Cette méthodologie est basée sur trois contributions majeures :

— Tout d'abord, nous proposons une nouvelle approche pour modéliser un algorithme de traitement d'images à embarquer et les contraintes temps-réel associées. Cette modélisation permet, en s'appuyant sur différentes heuristiques, de déterminer la stratégie optimale pour le portage de l'algorithme sur une architecture hétérogène donnée.

— Pour estimer les capacités des différentes architectures hétérogènes du marché, nous proposons une méthodologie de caractérisation d'architectures de calcul, basée sur un ensemble de vecteurs de test. Cette approche permet l'extraction semi-automatique de caractéristiques clés d'une architecture de calcul et de son écosystème.

— En utilisant ces informations, nous proposons une méthodologie de prédiction de performances permettant d'estimer le temps d'exécution d'un bloc d'algorithme sur un ensemble de processeurs de différents types.

1.2 Systèmes d'aide à la conduite

Les systèmes d'aide à la conduite ou *Advanced Driver Assistance Systems* (ADAS) participent à la réduction de l'accidentologie et à améliorer l'expérience de conduite. Il s'agit en fait d'assister le conducteur par l'automatisation de certaines tâches. Il peut s'agir de systèmes passifs (aucune action sur le véhicule, le conducteur est alerté par un signal sonore/lumineux d'un danger probable), de systèmes actifs (action sur le véhicule, par exemple freinage automatique). Ces systèmes proposent une automatisation de plus en plus poussée de certaines actions de la conduite et ont pour avantage de ne pas souffrir de limitations propres aux êtres humains (inattention, fatigue, temps de réaction lent, etc.). Ainsi, des études ont montré que l'automatisation complète de la conduite réduirait de manière considérable la mortalité sur les routes [LITMAN, 2014].

1.2.1 Accidentologie et sécurité routière

D'après le rapport annuel de l'observatoire national interministériel de la sécurité routière (ONISR), l'année 2015 comptait 70 802 personnes blessées et 3461 personnes tuées sur les routes de France métropolitaine [ONISR, 2015]. Ces accidents représentent

un coût estimer de 38,8 milliards d'euros, soit 1,5% du PIB. Les causes des ces accidents mortels en France métropolitaine sont données dans le tableau 1.1. Certains accidents ont plusieurs causes, ce qui explique le total de 122%.

TABLEAU 1.1 – Causes identifiées des accidents mortels en France métropolitaine. Certains accidents ont plusieurs causes. Source : [ONISR, 2015].

Causes identifiées dans un accident mortel	Taux
Vitesse	32%
Alcool	21%
Priorité	13%
Autre cause	12%
Stupéfiants	9%
Cause indéterminée	9%
Inattention	7%
Dépassement dangereux	4%
Malaise	3%
Somnolence / Fatigue	3%
Contresens	2%
Changement de file	2%
Obstacle	2%
Facteurs liés au véhicule	1%
Téléphone	1%
Non respect distance de sécurité	1%
Total	122%

À la lecture de ces statistiques, on remarque que la grande majorité des accidents mortels sont causés par le comportement du conducteur. Pour améliorer la sécurité des véhicules, les constructeurs automobiles proposent de plus en des systèmes d'aides à la conduite, ou ADAS.

1.2.2 Fonctionnalités

À l'heure actuelle, les ADAS implantés dans les véhicules en circulation ne permettent pas une conduite totalement autonome. Cette section a pour but d'identifier et de présenter les différents besoins et applications rencontrés aujourd'hui dans le monde des ADAS. Les fonctionnalités ADAS peuvent être classifiées selon 4 branches :

— manœuvre,

— évitement de collisions,

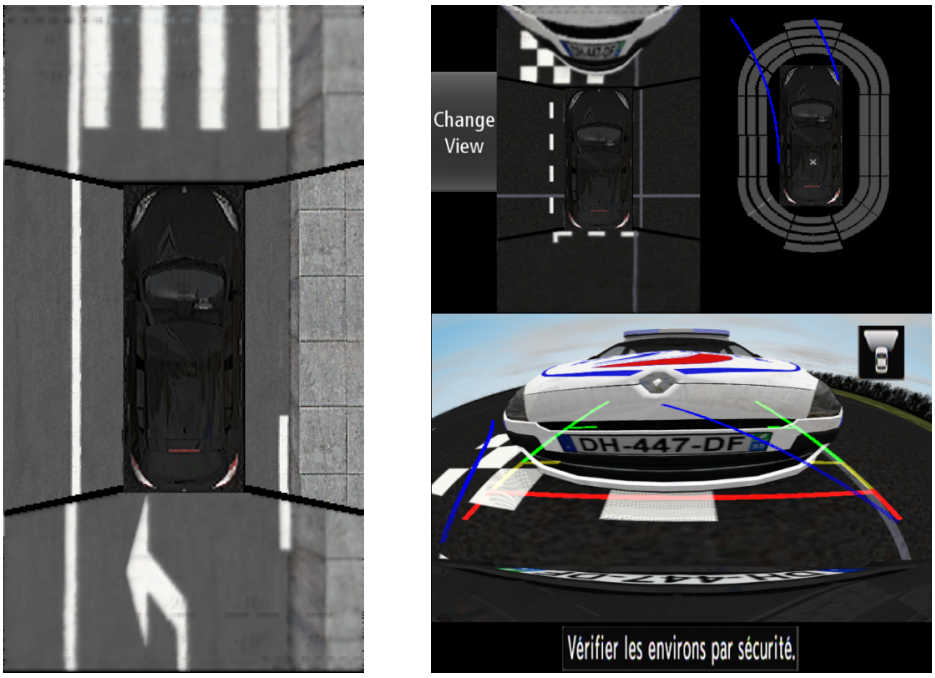
— assistance latérale et longitudinale,

— réalité augmentée et surveillance du conducteur.

Manœuvre

Cette branche regroupe tous les systèmes permettant une aide au parking voire l'automatisation complète de la manœuvre, il s'agit d'ADAS de confort. La fonctionnalité la plus basique consiste à avertir le conducteur par un signal sonore et lumineux la proximité de potentiels obstacles lorsque celui-ci cherche à se garer. On peut citer par exemple le système dit de « *radar de recul* » ou encore la « *caméra de recul* ».

L'AVM est une fonctionnalité un peu plus complexe, il s'agit de construire une vue de dessus du véhicule (ou *bird eye view*) en utilisant plusieurs caméras (le plus souvent quatre) disposées autour de celui-ci. Un tel rendu permet au conducteur d'avoir un aperçu direct de l'environnement proche du véhicule, et donc de faciliter les manœuvres à basse vitesse. Une illustration de l'AVM et de la *bird eye view* est donné en figure 1.1



(a) Exemple de *bird eye view* (b) Interface d'AVM

FIGURE 1.1 – Exemple d'AVM.

L'automatisation de la phase de manœuvre se fait sur plusieurs niveaux. Ainsi le *park assist* ou *Hand Free Parking* (HFP) propose un contrôle automatisé de trajectoire (contrôle du volant), mais laisse le conducteur gérer l'accélération et le frein. Le système *auto-park* propose quant à lui un contrôle entièrement automatisé (latéral et longitudinal) de la manœuvre de parking. Ces systèmes sont basés sur des capteurs ultrasons pour percevoir l'environnement du véhicules et les différents obstacles.

Évitement de collisions

L'évitement de collisions concerne l'ensemble des ADAS qui agissent sur le véhicule pour éviter un accident potentiel. On peut notamment citer l'exemple de l'*Advanced Emergency Braking* (AEB) qui agit automatiquement sur le frein du véhicule en cas de danger immédiat. Ce système peut être utilisé pour éviter des collisions avec d'autres véhicules, avec des piétons, etc. Le célèbre organisme Euro NCAP définit plusieurs tests pour noter les fonctionnalités d'évitement de collisions. Ainsi, depuis 2014 la notation inclue des tests d'AEB pour l'évitement de collisions avec d'autres véhicules, ces tests proposeront également en 2020 une caractérisation de l'AEB piéton et cycliste.

Très souvent, ces systèmes se basent sur des capteurs radars et caméras pour détecter les obstacles dangereux. Remarquons que ce système active un freinage d'urgence, il est donc crucial que le taux de faux positifs soit le plus faible possible pour ne pas entacher l'expérience de conduite de l'utilisateur et potentiellement générer des situations dangereuses.

Assistance latérale et longitudinale

Il s'agit là de systèmes permettant d'améliorer l'expérience de conduite et qui déchargent en partie le conducteur de certaines tâches. Par exemple l'*Adaptative Cruise Control* (ACC) permet, lorsque l'utilisateur utilise le régulateur de vitesse, de maintenir automatiquement une vitesse au plus proche de la consigne tout en garantissant le respect des distances de sécurité avec les autres véhicules. Ce système s'appuie très souvent sur un radar pour déterminer la position et la vitesse des autres véhicules.

Concernant l'assistance au contrôle latéral, il existe plusieurs degrés d'automatisation. Tout d'abord, le *Lane Departure Warning* (LDW) permet d'avertir le conducteur par un signal sonore ou lumineux du franchissement de ligne, par exemple en cas de somnolence ou d'inattention. Le *Lane Keeping Assist* (LKA) est quant à lui un système actif, il agit directement sur la trajectoire du véhicule si un franchissement de ligne est détecté. Enfin le *Lance Centering Assist* (LCA) permet un contrôle totalement automatique de la commande latérale, ce système se charge de garder le véhicule au centre de la voie.

Réalité augmentée et surveillance du conducteur

Contrairement aux autres branches, celle-ci concerne l'intérieur de l'habitacle et plus particulièrement le conducteur. L'idée ici est de pouvoir prévenir un comportement à risque du conducteur en analysant son comportement. Ainsi, le système peut détecter la somnolence de l'utilisateur et le prévenir par une alerte sonore pour éviter un accident. Ces systèmes sont en générale basés sur des caméras et des algorithmes de traitement d'images pour comprendre l'état du conducteur.

Vers la voiture autonome

Finalement, l'ensemble de ces systèmes permettent de converger vers une conduite de plus en plus automatisée et sûre. Aujourd'hui les différents constructeurs automobiles montrent de plus en plus de démonstrateurs, indiquant une forte volonté de converger vers la voiture entièrement autonome. Cependant si les technologies actuelles permettent une conduite autonome dans certaines conditions (par exemple sur autoroute avec l'ACC et le LCA), assez peu de constructeurs se risquent à vendre ce genre de service au grand public. Ainsi, le LCA est très souvent accompagné d'un système s'assurant de l'activité et de la vigilance du conducteur. À l'heure actuelle, seules quelques véhicules haut de gamme, notamment ceux du constructeur Tesla Motors, proposent des ADAS très avancés s'approchant d'une conduite autonome.

1.2.3 Capteurs

Les capteurs représentent la source d'informations pour les ADAS, ce sont eux qui approvisionnent l'algorithme décisionnel. On peut distinguer deux types de capteurs, les actifs et les passifs. Les capteurs passifs récupèrent directement une information sans agir sur leur environnement. A contrario les capteurs actifs « éclairent » l'environnement pour effectuer des mesures, comme le radar ou le flash d'un appareil photo.

Capteurs actifs

Lidar Le Lidar ou *LIght Detection And Ranging* est une technologie qui mesure une distance en illuminant une cible avec un laser et en analysant la lumière réfléchie. Ainsi en connaissant la vitesse de propagation de la lumière du laser, on est capable de déterminer

la distance à laquelle se trouve la cible. Ce type de capteur est très efficace pour la détection d'obstacles, puisqu'il renvoie directement une information de distance. Il s'agit en fait d'un nuage de points 3D. L'intérêt que peut avoir le lidar pour l'automobile et plus particulièrement la voiture autonome est discuté par RASSHOFER et GRESSER [2005]. Ainsi, le lidar peut être utilisé pour de la mesure de distance, pour des applications d'ACC en complément d'un capteur type radar. Cependant, il peut être utilisé pour d'autres applications encore, par exemple OGAWA et TAKAGI [2006] utilisent les informations du lidar pour de la détection de lignes. Il est aussi possible de s'en servir pour la détection de piéton en complément d'une caméra comme discuté par PREMEBIDA et collab. [2007]. Le lidar est capable (en fonction des modèles) de voir jusqu'à une centaine de mètres, il a cependant l'inconvénient de ne pas fonctionner par temps de pluie ou chute de neige. En effet la longueur d'onde utilisée par le lidar étant proche du visible (faible longueur d'onde), les gouttes d'eau affectent directement les ondes émises par le lidar. Notons que le prix du lidar est un énorme frein à son utilisation sur des voitures grand public.

Radar Le radar utilise le même principe que le lidar pour la mesure de distance, mais avec des longueurs d'ondes plus grandes : de l'ordre du millimètre contre plusieurs centaines de nanomètres voire du micromètre pour le lidar. De ce fait, le radar est beaucoup moins sensible en condition de brouillard et de pluie fine. Il a aussi pour avantage de pouvoir mesurer directement la vitesse d'objets en mouvement en utilisant l'effet Doppler. Dans le domaine de l'automobile, il existe deux versions : les radars 24 GHz et 77 GHz, dont les bandes de fréquences sont imposées par les réglementations en vigueur dans chaque pays. Les radars 24 GHz sont généralement utilisés pour des détections sur le côté du véhicule, par exemple pour la vérification des angles morts ou *Blind Spot Warning* (BSW) lors du changement de voie. Les radars 77 GHz sont généralement utilisés pour de la détection frontale, par exemple l'ACC. Le principal inconvénient du radar réside dans le fait qu'il possède une faible résolution angulaire (environ 3° autour du maximum de rayonnement du radar).

Ultrason Contrairement aux lidars et radars, les capteurs ultrasons mesurent des distances à l'aide d'ondes acoustiques (de 30 kHz à 5 MHz). Ils utilisent cependant le même principe de fonctionnement, c'est-à-dire le temps de parcours de l'onde pour déterminer la distance d'un obstacle. Cependant il est à noter que cette technique fonctionne sur des distances beaucoup plus petites, de l'ordre du mètre. Dû à cette petite distance de détection, l'application principale reste l'aide au parking. Une discussion sur la mise en place de capteurs ultrasons est abordée par PARVIS et CARULLO [2001]. L'étude porte sur les problématiques de précision de mesure dues à la variation de la vitesse du son avec les changements de température et d'humidité. Il en résulte qu'avec une correction adaptée, il est possible d'obtenir des résultats suffisamment précis pour un très faible coût. Le coût est l'avantage principal de ce type de capteur. On trouve d'ailleurs aujourd'hui, sur le marché, un grand nombre de véhicules équipés de « radar de recul » (utilisant les capteurs à ultrasons) pour l'aide au parking.

Capteurs passifs

Caméra visible Une caméra visible est composée d'un ou plusieurs capteurs photoélectriques (CMOS ou CCD) qui convertissent des photons (de longueur d'onde visible par exemple) en signaux électriques analogiques. Ils sont généralement constitués d'une matrice de plusieurs pixels, chacun possédant un capteur (photodiode par exemple) et un

étage de pré-traitement (gain, numérisation, etc.). Le signal de sortie peut être analogique ou numérique (sur 8, 12, 16 bits) et représente une image monochrome ou couleur. Par définition, la caméra visible est sensible au même spectre que l'œil humain. Son angle de vue varie en fonction de différents paramètres (focale, taille du capteur, etc.). Notons qu'il existe des optiques permettant d'obtenir un très grand angle de vue (les objectifs *fisheye* peuvent proposer des angles de vue de 180°).

Capteur intelligent (Mobileye) Mobileye est une société israélienne commercialisant des caméras intelligentes pour les ADAS. Les capteurs intelligents Mobileye représentent actuellement 80% du marché des caméras pour les ADAS. Il s'agit en fait d'un capteur intégrant une puce STMicroelectronics, voir illustration 1.2, renvoyant des données sur le bus de communication du véhicule. Le capteur agit comme une boîte noire, il renvoie des informations du type détection d'obstacles, détection de lignes et permet une prise de contrôle du véhicule (freinage d'urgence etc...). Du point de vue du constructeur, ces capteurs intelligents sont extrêmement pratiques, puisqu'ils intègrent les algorithmes de traitement d'images et renvoient uniquement les informations utiles sur l'environnement du véhicule. Cependant, d'un point de vue économique le monopole actuel de Mobileye sur les caméras intelligentes est assez problématique pour les constructeurs automobiles.

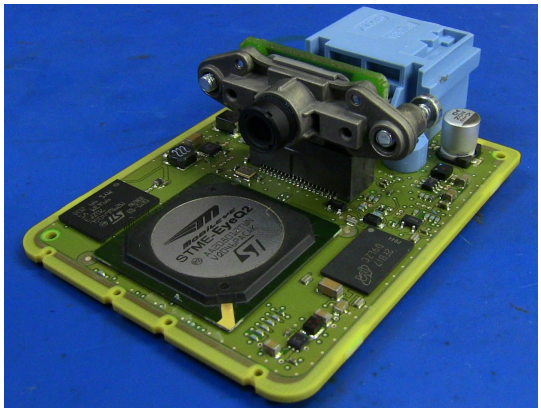


FIGURE 1.2 – Capteur EyeQ2 de Mobileye

Caméra infrarouge Ces caméras sont sensibles à des longueurs d'ondes plus grandes que celles du spectre visible qui s'étend de 400 à 700 nm. D'après le modèle du corps noir, chaque objet émet une lumière dont la longueur d'onde dépend de la température de celui-ci. Si l'on se penche sur le cas des êtres humains, naturellement chauds, on peut observer que ceux-ci émettent un spectre dont le maximum d'énergie se situe entre 8 et 14 µm. Cette gamme de longueurs d'ondes correspond au *Far Infra Red* (FIR) utilisée dans les caméras thermiques. Une autre gamme de longueurs d'ondes est utilisée dans les systèmes d'aide à la conduite, plus proche du visible, les *Near Infra Red* (NIR) allant de 0,75 à 1,4 µm. Même si elle ne détecte pas directement la lumière émise par des êtres humains, cette gamme de longueurs d'ondes a pour avantage d'être directement vue par des capteurs traditionnels type CCD ou CMOS, contrairement au FIR qui nécessite des capteurs spécifiques et plus chers. Les caméras NIR sont notamment utilisées pour des applications de surveillance du conducteur car l'éclairage infrarouge de l'habitacle (invisible pour le conducteur) facilite sa surveillance même en cas de très faible luminosité ambiante (par exemple de nuit).

GNSS Contrairement aux capteurs présentés jusqu'à présent, un capteur *Global Navigation Satellite System* (GNSS) ne sonde pas l'environnement mais effectue une mesure sur le propre comportement du véhicule. Le GNSS est un système permettant de retourner une position 3D dans l'espace en se basant sur les temps de parcours de signaux électromagnétiques émis par au moins 4 satellites différents en orbite autour de la Terre. En connaissant de manière suffisamment précise (~10 mètres) la position du véhicule, il est possible de recouper cette information avec une cartographie et ainsi de connaître la route sur laquelle se trouve l'utilisateur et donc la limitation de vitesse. Le GNSS peut aussi, en utilisant l'information temporelle, calculer la vitesse instantanée du véhicule, et donc prévenir l'utilisateur si celui ci est en excès de vitesse. Le GNSS est aujourd'hui très démocratisé, il équipe un très grand nombre de véhicules et est surtout utilisé pour une fonction de guidage. Il permet aussi d'obtenir le temps universel avec une très grande précision. Les principaux inconvénients de ce capteur sont sa faible précision spatiale et le fait que l'information ne soit pas disponible à tout instant (circulation en ville, sous un tunnel).

Mesure de la dynamique du véhicule D'autres capteurs présents dans le véhicule permettent de déterminer la dynamique du véhicule. On peut citer l'exemple des capteurs de vitesse des roues, de l'angle au volant, de l'accélération latérale et longitudinale, la vitesse angulaire, l'odométrie du véhicule, etc. Ces informations sont notamment utilisées pour des systèmes de sécurité comme le système anti-blocage des roues (également connu sous l'abréviation ABS), le correcteur électronique de trajectoire (en anglais *Electronic Stability Program*, ESP), le déclenchement des airbags, etc. Ces capteurs sont également utilisés pour des applications ADAS, par exemple la caméra intelligente Mobileye se sert de l'odomètre du véhicule pour déterminer le déplacement effectué entre deux images successives. De plus, pour des applications d'AEB, les mesures d'accélération sont utilisées pour réguler la décélération appliquée lors d'un freinage d'urgence.

1.3 Architectures embarquées pour les ADAS

Les fonctions ADAS citées précédemment nécessitent de la puissance de calcul pour pouvoir traiter les données issues des différents capteurs, prendre des décisions et agir en conséquence (pilotage d'actionneur, alarme, etc.), dans le milieu automobile on parle alors d'*Electronic Control Units* (ECUs). Les conditions et normes automobiles imposent que ces systèmes de traitement soient de faible coût, peu consommateurs et sûrs d'un point de vue fonctionnel (norme ISO 26262). Ainsi il existe sur le marché un grand nombre d'architectures utilisées pour embarquer des applications automobiles, on parle également de *Systems-on-Chip* (SoCs) qui intègrent, sur une même puce, l'ensemble des éléments nécessaires à l'ECU (processeurs, mémoire, périphériques d'interface, etc.).

1.3.1 Architecture type

De manière générale, une architecture de calcul embarquée est composée de différents éléments que l'on peut classer suivant deux catégories : les unités d'exécution et les unités de mémoire.

Unités d'exécution

Une unité d'exécution permet d'exécuter un fil d'instructions (calculs et/ou accès mémoire) également appelé *thread*. Elle est composée de plusieurs unités élémentaires qui sont activées d'une manière sélective suite au décodage de différents types d'instructions. On distingue ainsi les unités élémentaires suivantes :

- **Unité arithmétique et logique** délivrant les opérations d'additions, ou/et logique, etc. (nombre entier et virgule fixe).
- **Unité de multiplication** se chargeant d'appliquer les multiplications ou les multiplications - accumulations.
- **Unité de calcul en virgule flottante** permettant d'effectuer des opérations sur les variables à virgule flottante (additions et multiplications).
- **Unités de calcul spécifique** : unités optionnelles qui encapsulent une implémentation hardware spécifique d'une ou plusieurs instructions (par exemple *cos*, *sin*, etc.).
- **Unité de branchement** gérant les instructions du type condition (comme *if*) ou boucle (comme *for*). Elles sont souvent liées à des unités de prédictions qui peuvent mettre en place une exécution spéculative du fil d'instruction.
- **Unité d'adresse** permettant le calcul d'adresses, par exemple accès à la n^{e} valeur d'un vecteur par l'opérateur $[n]$.
- **Unité *Load & Store*** se chargeant de faire l'interface entre les registres du processeur et la mémoire du système, en utilisant les unités d'adresses.

Unités de mémoire

Les unités de mémoire permettent de stocker de l'information. Généralement, on caractérise la mémoire par sa capacité de stockage (quantité d'information), sa bande passante (vitesse d'échange de l'information) et sa latence (délai pour obtenir une information). Il existe plusieurs niveaux de mémoires, allant du voisinage le plus proche des unités d'exécution (mémoires très rapides mais de très petite taille) au stockage de masse, beaucoup plus lents (comme les mémoires non-volatiles).

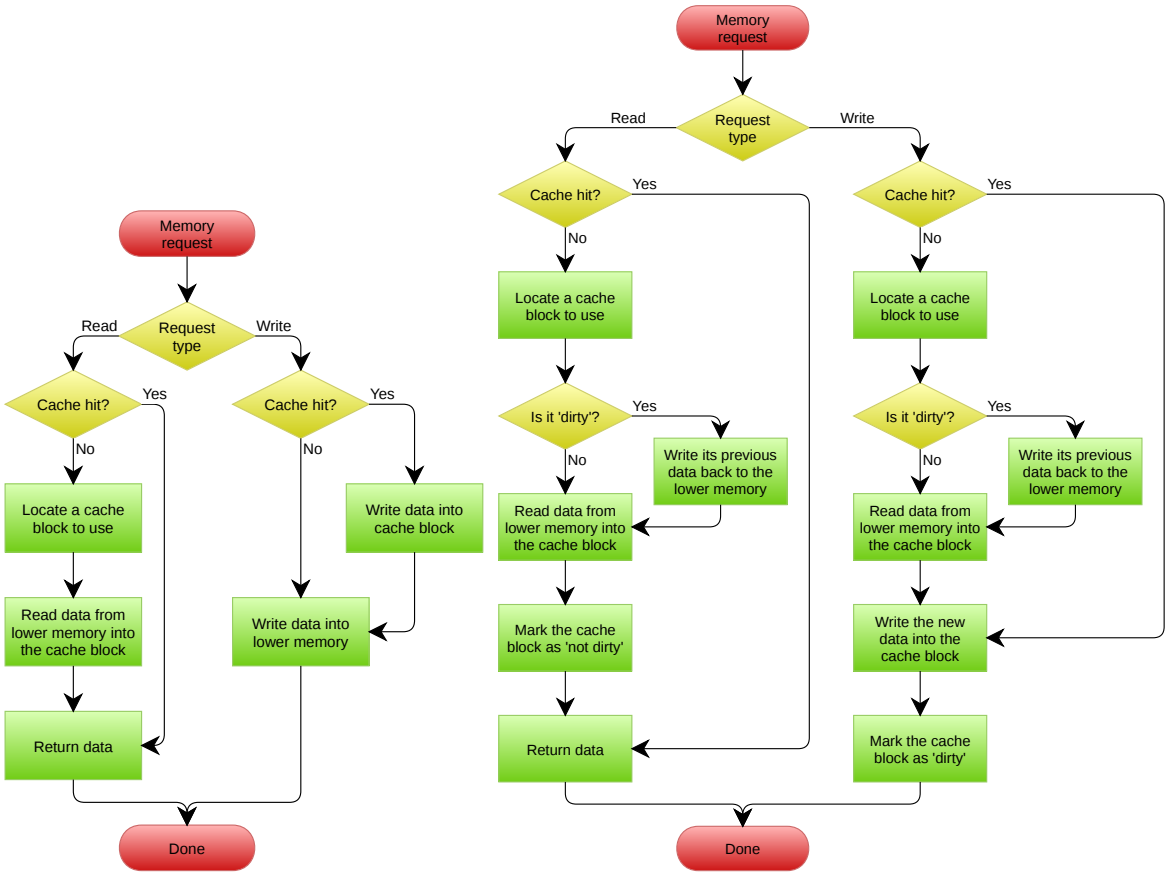
Registre Les registres constituent la mémoire la plus rapide d'un processeur. Les résultats des instructions sont généralement stockés sur des registres avant d'être sauvegardés en mémoire (par le biais d'instruction *Load & Store*). Le nombre et la taille des registres (souvent 32 ou 64 bits) varient d'un processeur à l'autre.

Mémoire cache La mémoire cache correspond à la mémoire tampon entre le processeur et la RAM. Pour des raisons d'efficacité, elle est structurée en plusieurs niveaux de caches : L1, L2 et parfois L3, qui peuvent être uniques ou partagées entre plusieurs unités d'exécution ou cœurs d'un même processeur. Il s'agit de mémoires ayant de faibles latences d'accès mais de faibles capacités de stockage (en général pas plus de quelques Mo). Le cache agit comme une mémoire temporaire permettant de stocker des données préalablement utilisées, pour diminuer le temps d'un accès ultérieur. Ainsi, lorsque l'unité d'exécution cherche à accéder à une donnée qui se trouve déjà dans le cache, on parle alors de *cache-hit*, l'accès sera très rapide. Dans le cas contraire, on parle de *cache-miss*, la latence d'accès sera beaucoup plus lente. Remarquons qu'il existe plusieurs politiques et stratégies pour gérer la mémoire cache. Dans le cas de l'écriture, on distingue deux politiques :

— l'écriture immédiate (*writhe-through*) qui fait en sorte que les données sont écrites à la fois dans le cache et dans la mémoire centrale pour garantir une cohérence entre les différents niveaux de mémoire,

— l'écriture différée (*write-back*) qui évite des écritures répétées dans la mémoire centrale en se permettant de garder une différence entres les données dans le cache et celles présentes dans la mémoire. Les lignes du cache non cohérentes avec la mémoire centrale sont marquées d'un *dirty bit*, dont nous expliquons le fonctionnent en figure 1.3.

Remarquons que la majorité des processeurs actuels proposent par défaut une politique *write-back*, ce qui permet d'avoir des délais en écriture beaucoup plus rapides.



(a) *Write-through* (b) *Write-back*

FIGURE 1.3 – Schéma de principe décrivant le fonctionnement du *write-through* et *write-back*. La politique de *write-back* permet d'éviter d'aller écrire l'ensemble des données dans la mémoire centrale et se sert du *dirty bit* pour marquer les blocs du cache qui ne sont pas cohérents avec la mémoire centrale. Ainsi, si l'on cherche à écraser un bloc du cache marqué *dirty*, alors le bloc sera préalablement réécrit sur la mémoire du niveau inférieur pour garantir la cohérence des données.

RAM ou mémoire vive Il s'agit de la mémoire principale du système, elle permet de stocker de grande quantité de données (quelques Go). Cependant accéder de manière répéter à la RAM est souvent coûteux en termes de temps d'exécution à cause de sa forte latence. Notons que la majorité des mémoires RAM d'aujourd'hui sont qualifiées de *double data rate* (DDR). Ainsi, elles sont capables de transférer des données à la fois sur le front montant et le front descendant de l'horloge de la mémoire, soit deux transactions par coup d'horloge.

Mémoire non-volatile Cette mémoire conserve les données en l'absence d'alimentation électrique (utilisée notamment pour le stockage de l'OS), elle a souvent plusieurs centaines de Go de mémoire. On peut citer les disques durs, mémoires flashs (clés USB, cartes SD, etc.). Elle est cependant assez peu utilisée lors de l'exécution d'un programme temps-réel à cause de ses performances catastrophiques (sauf dans le cas d'applications spécifiques comme la gestion de base de données).

Processeurs

Le terme de processeur désigne un système électronique capable d'effectuer des opérations et ne se limite pas uniquement au *Central Processing Unit* (CPU). Un processeur est composé d'une ou plusieurs unités d'exécution et de certaines unités mémoire comme les mémoires cache. Un processeur peut intégrer plusieurs cœurs (on parle alors de multicœur), il est alors capable d'exécuter plusieurs *threads* simultanément et donc dispose de plusieurs unités d'exécution. Remarquons que le nombre de cœurs n'est pas toujours égal au nombre d'unités d'exécution, par exemple les processeurs Intel sont capables d'exécuter simultanément 2 *threads* par coeur (soit 2 unités d'exécution par cœur) grâce à l'*Hyper-Threading*.

SoCs

Finalement, un SoC consiste à intégrer l'ensemble (ou une partie) de ces composants sur une même puce. Remarquons que plusieurs processeurs peuvent être intégrés sur une même puce et qu'un SoC peut également intégrer une mémoire non-volatile. Ainsi, nous illustrons en figure 1.4 l'exemple d'un SoC composé de deux processeurs partageant une même mémoire RAM. La figure décrit également le découpage en unités d'exécution (dans l'exemple une unité d'exécution est équivalent à un cœur du processeur) et en unités élémentaires.

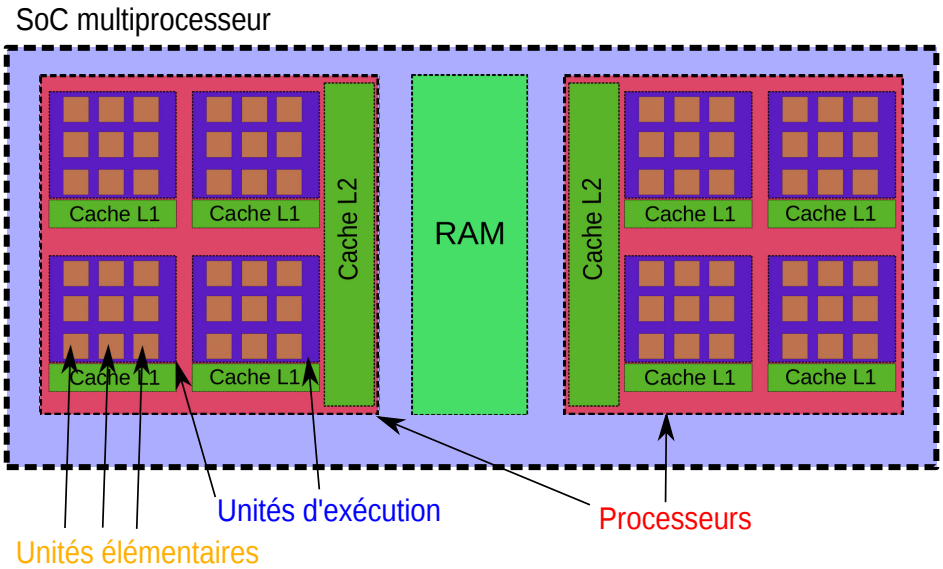


FIGURE 1.4 – Exemple d'un SoC composé de deux processeurs. Dans cet exemple la RAM est utilisée par les deux processeurs. Un processeur est composé de quatre unités d'exécution (en violet), elles même composées de plusieurs unités élémentaires (en orange). Chaque unité d'exécution dispose de son propre cache L1, alors que le L2 est partagé par l'ensemble des unités d'exécution de ce processeur.

1.3.2 Le calcul parallèle

Certaines architectures ont la particularité d'être conçues pour procéder aux différents calculs de manière parallèle. Au vu des différents types de parallélisme, il est possible de définir plusieurs classes auxquelles peuvent se rattacher les différentes architectures de calcul. Ainsi, la taxonomie de FLYNN [1972] définit quatre classes :

— **SISD** (*Single Instruction Single Data*) : le calcul s'effectue de manière séquentielle, il n'y a aucune forme de parallélisme. Cela concerne les architectures n'étant pas capables d'effectuer plusieurs calculs de manière simultanée, effectuant les opérations les unes à la suite des autres (au maximum une instruction par cycle d'horloge). Par exemple un CPU monocœur, monoscalaire pipeliné appartient à cette classe.

— **MISD** (*Multiple Instruction Single Data*) : plusieurs opérations sont appliquées en même temps à une donnée unique. Très peu d'architectures se rattachent à cette classe.

— **MIMD** (*Multiple Instruction Multiple Data*) : capable d'effectuer plusieurs opérations sur différentes données de manière simultanée. Cela concerne les systèmes présentant plusieurs unités d'exécution, ces architectures sont donc capable d'effectuer des calculs concurrentiels sur plusieurs données.

— **SIMD** (*Single Instruction Multiple Data*) : une opération est appliquée à plusieurs données en même temps. Il s'agit de ce que l'on appelle des opérations ou architectures vectorielles.

Parallélisme au niveau architecture

Du point de vue fonctionnel, il existe plusieurs stratégies pour procéder à du calcul parallèle sur une architecture. Nous présentons ici une définition des différents niveaux de parallélisme, en allant du niveau registre au sein d'une même unité d'exécution, jusqu'au niveau multiprocesseur hétérogène.

Parallélisme de registre Le parallélisme de registre consiste à appliquer une même opération à des zones régulières d'un même registre. Cela consiste souvent à un registre de grande taille (128 ou 256 bits), qui est divisé en un vecteur de plusieurs variables (c'est le cas des technologies NEON, SSE, AVX, etc.). De cette manière il est possible d'appliquer une même opération à plusieurs variables en simultanée. On parle alors d'opérateurs vectoriels, ou SIMD.

Parallélisme fonctionnel Pour exécuter une instruction, plusieurs étapes sont nécessaires. Les étapes dépendent bien évidemment de l'architecture, cependant de manière générale chaque étape fait appel à une unité différente. Ainsi, il est possible de séquencer un traitement périodique en plusieurs étapes spécifiques qui seront exécutées en parallèle par des unités différentes. C'est ce que l'on appelle le pipeline fonctionnel, un exemple est donné dans la figure 1.5. Avec une répétition suffisamment grande, le temps d'exécution du traitement tend alors vers 1 cycle d'horloge. De plus, sur les architectures actuelles plusieurs unités élémentaires sont présentes (comme présenté dans la partie 1.3.1), chacune permettant d'exécuter un type d'instruction. Ainsi il devient possible d'exécuter plusieurs instructions en même temps si elles utilisent des unités élémentaires différentes, par exemple des opérations faisant intervenir une unité arithmétique et logique en même temps que des opérations faisant intervenir l'unité de calcul en

virgule flottante. Une telle approche nécessite des stations de réservations et des *buffers* de réordonnancent. On qualifie de superscalaire les architectures capables d'utiliser cette technique.

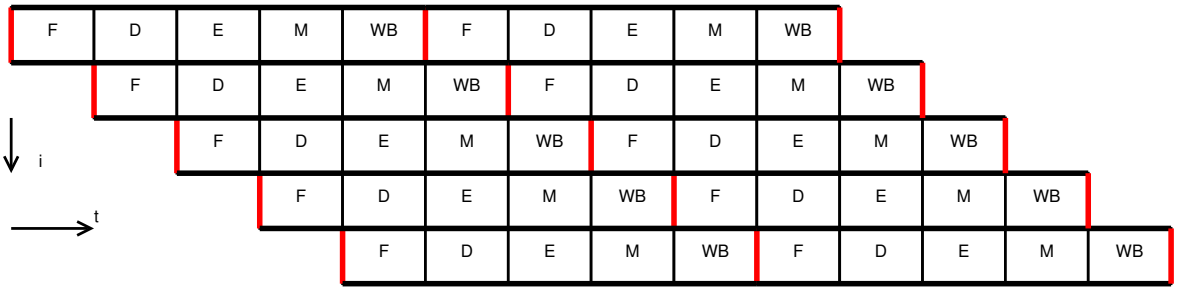


FIGURE 1.5 – Séquençage d'opérations dans un processeur doté d'un pipeline à 5 étages. Les 5 étages sont *Fetch*, *Decode*, *Execute*, *Memory* et *WriteBack*. Il faut 9 cycles pour exécuter 5 opérations et 1004 cycles pour exécuter 1000 opérations. Dans cet exemple, à partir de t = 5, tous les étages du pipeline sont sollicités, et les 5 opérations ont lieu en même temps.

Parallélisme multicœur Certaines architectures présentent plusieurs unités d'exécution parfois appelées cœurs, c'est le cas de la majorité des CPUs modernes qui peuvent avoir 4 voire 8 cœurs. Il est donc possible de profiter des différents cœurs pour effectuer plusieurs calculs simultanément. Certaines APIs permettent d'ailleurs de faciliter le portage d'algorithme sur ce type de cible, c'est notamment le cas de l'API d'OpenMP qui utilise des directives du compilateur pour la parallélisation.

Parallélisme hybride Le parallélisme hybride (ou à hardware mixte) consiste à répartir l'exécution d'un algorithme sur différents processeurs de natures différentes et présents sur un même SoC. Ainsi, sur un calculateur embarqué comportant un CPU multicœur et un GPU, il est possible d'effectuer une partie des calculs sur un des processeurs, tout en effectuant une autre partie sur l'autre processeur. Chaque processeur ayant ses propres avantages et inconvénients, l'efficacité dépendra fortement de l'adéquation entre le processeur et la partie de l'algorithme à exécuter. Il existe deux modèles pour les calculateurs hybrides :

— Chaque processeur dispose de sa propre mémoire, ils communiquent alors par le biais d'un bus de données avec un mécanisme plus ou moins explicite de transfert de données.

— Les processeurs sont organisés autour d'une même mémoire globale, les données sont donc directement partagées et sont protégées contre les lectures et écritures concurrentes par un mécanisme hardware ou software. Le système de cache local permet d'accélérer les différents accès.

Classes de parallélisme au niveau algorithme

Les architectures de calcul proposent donc différents niveaux de parallélisme et permettent donc d'accélérer grandement le temps d'exécution d'un algorithme. Cependant, tous les algorithmes ne sont pas égaux face au parallélisme, par exemple un algorithme effectuant un ensemble d'opérations successives sur une unique donnée d'entrée ne profitera nullement du parallélisme. Ce problème est d'ailleurs très bien illustré par la célèbre loi d'AMDAHL [1967] qui exprime l'accélération du temps d'exécution d'un algorithme en

fonction du nombre d'unités d'exécution utilisées. Cette loi montre que l'accélération atteint un seuil dépendant de la proportion de l'algorithme pouvant être parallélisée.

Parallélisme simple – P_0 Il s'agit des algorithmes dont la parallélisation est simple à mettre en place, il n'y a pas de dépendance entre les données, chaque calcul peut s'effectuer indépendamment des autres et donc de manière parallèle. Dans le cas du traitement d'images, il peut s'agir d'un algorithme qui effectue des opérations pixels à pixels, comme une conversion couleur vers des niveaux de gris ou la binarisation simple d'une image.

Parallélisme impliquant des opérations atomiques – P_1 Cette catégorie concerne les algorithmes pour lesquels il y a une possibilité que des opérations concurrentes accèdent à la même donnée en même temps. Le problème réside dans le fait que l'ordre d'exécution des opérations concurrentes n'est pas déterministe. Un exemple typique est une écriture concurrente à plusieurs opérations de lecture : dans ce cas le résultat n'est pas bien déterminé car il dépend du moment exact de l'opération (si la lecture à lieu avant ou après l'écriture). Il y a d'autres cas comme les écritures concurrentes ou d'autres opérations plus complexes (incrémentations, comparaisons, etc.). Il existe plusieurs stratégies pour se prémunir de ce genre de problèmes. On peut utiliser un mécanisme logiciel (grâce aux mutex ou sémaphores) qui interdit l'accès à une variable lorsque qu'un *thread* y effectue une écriture. Cependant cela implique souvent un ralentissement non négligeable en termes de temps de calcul. Une autre stratégie consiste à utiliser une éventuelle protection hardware qui est beaucoup plus efficace.

Réduction parallèle – P_2 Cette catégorie concerne les algorithmes qui calculent un unique résultat à partir d'un ensemble de n données. Il peut s'agir par exemple de déterminer le maximum ou la somme des n données d'entrée. L'exécution peut se faire classiquement séquentiellement mais on peut la paralléliser en utilisant un arbre dont les branches se regroupent. Lorsque l'on est capable d'exécuter au moins $n/2$ opérations par cycle, le temps d'exécution se ramène en $\lceil \log_2(n) \rceil$. Le principe est illustré en figure 1.6.

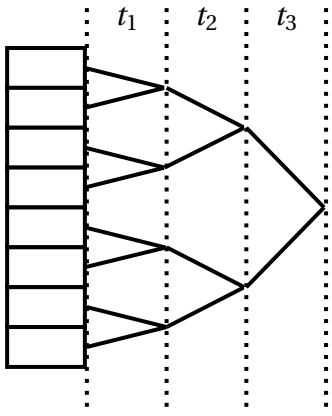


FIGURE 1.6 – Exemple de réduction parallèle pour 8 données d'entrée, dans le cas où l'on est capable d'effectuer au moins 4 opérations par cycle. Chaque intersection correspond à un calcul. L'échelle de temps est donnée en cycles d'horloge. Le nombre de données traitées est divisé par 2 à chaque cycle.

Parallélisme pour les algorithmes itératifs – P_3 Il s'agit des algorithmes pour lesquels chaque itération dépend du résultat précédent, par exemple si $u_i = f(u_{i-1})$. À cause de cette dépendance, l'algorithme est impossible à paralléliser. Cependant, le traitement effectué à chaque itération, $f(u)$ dans le cas de l'exemple, peut potentiellement être parallélisé, et donc appartenir à l'une des catégories P_0, P_1 ou P_2.

1.3.3 Types de processeur

Après avoir présenté les généralités autour des composants d'un processeur, intéressons-nous à présent aux types d'architectures de calcul pouvant être utilisées dans le cadre d'applications ADAS. Il s'agit là de présenter de manière générale les différents types de processeurs, en évoquant leurs principaux avantages et inconvénients.

Microcontrôleur

Un microcontrôleur (µc) est un circuit intégré rassemblant tous les éléments essentiels pour effectuer des calculs. Ainsi, il intègre sur la même puce une ou plusieurs unités d'exécution, la mémoire RAM et non-volatile, ainsi qu'un ensemble d'entrées/sorties (notamment bus de communications). Son haut niveau d'intégration (faible consommation, faible coût, etc.) en a fait un incontournable dans le monde de l'embarqué et dans l'automobile (les contrôleurs des moteurs par exemple).

Les µc sont cadencés à de faibles fréquences (en général une centaine de MHz) et possèdent une faible quantité de mémoire pour les données (quelques MB). De plus, pour obtenir la consommation et le coût les plus faibles possibles, les architectures sont en général simplifiées au maximum. Ainsi, sur un µc *PIC* il faut en général 4 coups d'horloge pour exécuter une instruction. Jusqu'à maintenant, ce type de processeur convenait parfaitement au besoin automobile. Cependant avec des ADAS demandant de plus en plus de calculs, les acteurs automobiles se tournent vers d'autres architectures, proposant des capacités beaucoup plus puissantes.

DSP

Le *Digital Signal Processor* (DSP) est, comme son nom l'indique, un processeur utilisé pour des applications de traitement du signal. Ainsi, il est conçu dans le but d'être optimisé pour des calculs impliquant des multiplications-accumulations, très utilisées dans des opérations de filtrage. Ils disposent également d'instructions spécifiques permettant d'accélérer grandement les boucles, très utilisées pour le parcours de tableaux. Pour accélérer certaines applications de traitement du signal, par exemple le calcul de FFT, ils proposent des modes d'adressages spécifiques, comme l'adressage en bit-reverse. Remarquons que la plupart des DSPs ne gèrent pas les opérations en virgule flottante, proposant ainsi des coûts plus faible et une consommation énergétique réduite.

Sur l'aspect parallélisme, les DSPs récents intègrent plusieurs cœurs de calcul (c'est-à-dire plusieurs unités d'exécution par processeur) et la gestion d'instructions SIMD. Les DSPs sont considérés comme des architectures à *Very Long Instruction Word* (VLIW). Cette technologie permet de regrouper plusieurs instructions indépendantes utilisant des unités élémentaires différentes en une seule longue instruction. Par opposition aux architectures superscalaires qui répartissent les différentes instructions utilisant différentes unités élémentaires lors de l'exécution, la gestion du VLIW se déroule pendant la phase de compilation.

ARM (RISC)

La société Britannique ARM est basée sur un modèle économique assez particulier. Contrairement à ses concurrents dans le domaine des semi-conducteurs, elle ne vend pas directement des processeurs. Ainsi, elle fonctionne sur la propriété industrielle des architectures qu'elle conçoit, en vendant à ses clients le droit d'utiliser les architectures ARM. Aujourd'hui, les processeurs ARM sont surtout connus pour leur utilisation dans l'embarqué grand public tel que les smartphones et tablettes

Les processeurs ARM présentent un jeu d'instructions réduit, en anglais *Reduced Instruction Set Computer* (RISC). Cela se traduit par des programmes plus volumineux, mais des instructions exécutées plus rapidement et une consommation réduite par rapport aux architectures à jeu d'instructions étendu, ou *Complex Instruction Set Computer* (CISC). Ils sont donc plus adaptés aux contraintes des systèmes embarqués.

La gamme des processeurs ARM est assez large. Dans la génération des ARMv7, on distingue ainsi les processeurs généraux de la famille des Cortex-A, très utilisés dans le domaine des appareils nomades du type smartphone ou tablette. La même génération propose également une gamme de µc basée sur la même architecture : la famille des Cortex-M, et également une famille orientée pour les applications temps-réel : les Cortex-R. Avec l'arrivée récente de la nouvelle génération, ARMv8, la société a lancé des processeurs avec des registres de 64 bits pour s'attaquer, entre autres, aux marchés des serveurs informatiques et des supercalculateurs.

Pour entrer un peu plus dans les détails de la famille de Cortex-A, les processeurs proposés sont généralement composés de plusieurs cœurs de calcul (jusqu'à 4 cœurs), chaque cœur étant capable d'exécuter sont propre fil d'instruction. Dans le cas des processeurs ARM, on considérera donc qu'un cœur de calcul est équivalent à une unité d'exécution. Chaque cœur dispose d'un ensemble d'extension lui permettant d'accélérer grandement l'exécution de certains programmes. Par exemple, les processeurs ARM proposent des instructions DSP, comme la multiplication-accumulation. Chaque cœur propose également un coprocesseur gérant les opérations en virgules flottante et un coprocesseur lui permettant d'effectuer des opérations SIMD : l'extension NEON.

La société propose également des SoCs composés de plusieurs processeurs ARM, basés sur la technologie big.LITTLE. Il s'agit d'associer deux processeurs différents : un processeur ayant une faible puissance et peu consommateur en énergie (le LITTLE) et un processeur pour le calcul intensif mais plus gourmand en énergie (le big). Ces SoCs ont pour avantage de pouvoir s'adapter en fonction des besoins de l'utilisateur et offrent ainsi un très bon compromis entre puissance de calcul et autonomie grâce à une faible consommation en énergie.

Intel x86 (CISC)

La société Intel a récemment confirmé sont intérêt pour le marché automobile avec le rachat de la société Israélienne Mobileye pour 15 milliards de dollars. Rappelons que Mobileye est le leader mondial dans le domaine des caméras intelligentes pour les ADAS.

Les CPUs Intel x86 équipent aujourd'hui la plupart des ordinateurs grand public. Par opposition à la société ARM, Intel a choisi de privilégier la performance pure dans la conception de ses processeurs par rapport à l'efficacité énergétique. Remarquons que si les processeurs x86 étaient auparavant de conception CISC, les nouvelles générations sont maintenant conçues en tant que RISC, les instructions complexes étant transformées par le processeur en instructions plus élémentaires.

Du point de vue parallélisme les processeurs x86 proposent des instructions SIMD, on

peut par exemple citer les extensions MMX, SSE, SSSE3, SSE4, etc. Ces processeurs disposent également de plusieurs cœurs de calcul. Notons qu'avec la sortie du Pentium 4 Northwood, Intel a lancé la technologie de l'*Hyper-Threading* permettant d'exécuter simultanément deux *threads* sur un même cœur de calcul. Ainsi, grâce à cette technique un cœur d'un processeur x86 est équivalent à deux unités d'exécution selon notre modélisation discutée dans la partie 1.3.1.

GPU

Les processeurs graphiques ou *Graphics Processing Units* (GPUs) ont été initialement conçus pour le calcul lié à l'affichage et au rendu 3D. Ces processeurs se sont énormément démocratisés grâce au marché du jeu vidéo, demandant un réalisme des rendus 3D de plus en plus poussé. Depuis quelques années, ils sont également utilisés pour du *General-Purpose Computing on Graphics Processing Units* (GPGPU), c'est-à-dire faire du calcul générique sur un GPU afin de profiter de leur architecture massivement parallèle. Ainsi, la société Nvidia a sorti l'API CUDA en 2007, permettant d'exécuter un programme écris en C/C++ sur un GPU Nvidia à des fins autres que de l'affichage ou du rendus 3D. Remarquons qu'il existe d'autres APIs permettant la programmation GPGPU, comme OpenCL.

Le GPU est un processeur massivement parallèle, le parallélisme se base sur le modèle d'exécution *Single Instruction Multiple Threads* (SIMT). Dans ce modèle, le processeur exécute une même série d'instructions sur un grand nombre de *threads*. Un GPU est généralement composé d'un très grand nombre de cœurs de calcul, lui permettant d'exécuter un grand nombre de *threads* de manière simultanée. Cependant du fait de son modèle d'exécution SIMT, il n'exécute qu'un seul fil d'instructions pour l'ensemble des *threads*, un GPU est donc équivalent à une seule unité d'exécution selon notre modélisation présentée dans la partie 1.3.1.

L'API CUDA permet d'exécuter un fil d'instructions, implémenté en C/C++, par un ensemble de *threads*. Les *threads* sont organisés autour d'une grille de trois dimensions, elle-même composée de plusieurs blocs. Ainsi, l'ensemble des *threads* appartenant à un même bloc partagent un certain nombre de ressources, comme la *shared memory* une mémoire très rapide analogue à un cache de niveau L1. Lors de l'exécution, les instructions sont décodées, puis exécutées par des groupes insécables de 32 *threads* appelés *warp*s. Plus de détails à propos du modèle de programmation CUDA sont présentés dans [NVIDIA, 2015].

FPGA

Le *Field-Programmable Gate Array* (FPGA) est très particulier comparé aux autres architectures de calculs présentées, jusqu'à maintenant il s'agissait d'exécuter du code machine. Le FPGA est différent dans le sens où la « programmation » consiste en fait à un design hardware de circuits logiques, LUTs, bascules, *buffers*, etc. Donc pas de code machine, mais un design de circuits électroniques.

Le design hardware étant complètement maîtrisé par l'utilisateur, l'architecture sera forcément adaptée à l'application souhaitée. Dans le cadre du traitement de l'image, il est possible de concevoir le calcul sur plusieurs pixels en parallèle. Le nombre de circuits logiques est cependant limité, il peut aller jusqu'à plusieurs millions pour certains FPGAs.

Le FPGA reste cependant assez peu performant pour certaines applications. Ainsi, effectuer des calculs en virgule flottante demandera un très grand nombre de ressources, ce qui limitera le parallélisme pour ce type de calcul sur des applications du type traitement

d'images. De plus, le design d'un FPGA est autrement plus complexe que l'implémentation d'un algorithme sur une cible hardware ordinaire. On a cependant tendance à voir émerger de plus en plus d'outils facilitant le design du FPGA.

Architectures hétérogènes

Les SoCs hétérogènes sont des architectures composées des plusieurs processeurs. Très souvent ils intègrent un ou plusieurs processeurs génériques, permettant d'effectuer des calculs simples et ayant une grande flexibilité, et un ou plusieurs processeurs massivement parallèles, permettant d'accélérer les algorithmes traitant un grand nombre de données.

Ce type de SoC présente un très bon compromis entre flexibilité et puissance de calcul, il permet d'exécuter un code générique sans trop d'effort de portage et d'accélérer de lourds calculs grâce à son/ses processeur(s) massivement parallèle(s). Comme discuté dans une de nos contributions [SAUSSARD et collab., 2015], les architectures hétérogènes représentent une bonne solution pour embarquer des algorithmes intensifs en calcul, du type traitement d'images.

D'un point de vue mémoire, deux types d'architectures existent :

— Chaque processeur a sa propre mémoire, la communication se fait par le biais d'un bus de données avec un mécanisme plus ou moins explicite de transfert de données.

— Les processeurs sont organisés autour d'une même mémoire centrale, les données sont donc directement partagées et sont protégées contre des collisions par des mécanismes software ou hardware. Chacun des processeurs dispose de ses propres niveaux de cache, un mécanisme est également en place pour garantir la cohérence des données entre les caches des différents processeurs.

1.4 Analyse de l'embarquabilité

Pour un SoC donné, un algorithme est dit embarquable si et seulement si cet algorithme peut être implémenté et exécuté par ce SoC tout en respectant les contraintes temps-réel imposées par le cahier des charges. Étudier l'embarquabilité d'un algorithme consiste donc à déterminer si son temps d'exécution respecte les contraintes temps-réel et la manière dont celui-ci doit être implémenté. La problématique d'embarquabilité d'un algorithme est un enjeu de taille pour l'industrie automobile, il s'agit en fait de déterminer quel hardware utiliser pour que l'algorithme en question soit embarquable.

1.4.1 Temps-réel

Dans le domaine de l'automobile et plus particulièrement des ADAS ou de la voiture autonome, le respect des contraintes temps-réel est critique. Par exemple dans le cas d'un AEB, il est impératif que le système ait un temps de réponse inférieur à une valeur maximale sans quoi le système pourrait ne pas freiner à temps et donc ne pas éviter l'accident.

Systèmes temps-réel

Un système pouvant garantir le respect de ce genre de contraintes est dit temps-réel. Dans le domaine de l'informatique, il s'agit de prendre en compte l'aspect temporel de

chaque tâche de l'algorithme : le temps de réponse d'une tâche est tout aussi important que son résultat, la réponse doit se faire dans un temps imparti. Il existe deux types d'approches pour la gestion du temps-réel : les systèmes temps-réel dur et les systèmes temps-réel mou.

Temps réel dur Cette politique agit de manière stricte, chaque tâche de l'algorithme doit respecter une échéance sous peine de conséquences critiques, voire catastrophiques. L'exemple typique est la conduite autonome, le système doit garantir un certain temps de réponse face à certaines situations. Du point de vue de l'OS, cela implique la mise en place d'un certain nombre de mécanismes, notamment la prise en compte de l'échéance de chaque tâche, leurs périodes d'appel, etc.

Temps réel mou Contrairement à la politique de temps-réel dur, on s'autorise ici des non-respects occasionnels des contraintes temps-réel pour certaines tâches. Il s'agit généralement d'applications non critiques, où une tâche se terminant après son échéance n'a que peu d'impact sur l'utilisateur. On peut citer l'exemple d'un système multimédia, où une surcharge exceptionnelle du calculateur peu, par exemple, faire chuter momentanément le nombre d'images par seconde affiché pour un film ou un jeu vidéo.

Contraintes temps-réel

Finalement, que ce soit pour un système temps-réel dur ou mou, l'exécution d'un algorithme en temps-réel est possible seulement si son temps d'exécution respecte certaines contraintes. On peut ainsi distinguer deux types de contraintes : les contraintes de rythme et de latence. Dans le cas contraire, quelle que soit la politique du système ou l'OS utilisé, l'exécution temps-réel n'est tout simplement pas possible.

Contrainte de rythme De manière général, un capteur envoie des données régulièrement, à une fréquence fixe (par exemple une caméra fonctionnant à 30 images par seconde). La contrainte de rythme impose au système de traiter toutes les données d'entrées, par exemple dans le cas d'une caméra l'algorithme ne doit pas « rater » des images. Cela implique donc que l'algorithme complet doit être appelé à une période fixe. Remarquons qu'une application ADAS complète utilise plusieurs capteurs fonctionnant à des fréquences différentes, la période d'appel à l'algorithme doit donc être judicieusement choisie en fonction de ces conditions et de manière à pouvoir assurer que l'intégralité des données d'entrée puisse être traitée.

Contrainte de latence La latence correspond au délai entre l'envoi d'une nouvelle donnée par un capteur et la sortie ou prise de décision de l'algorithme. Dans le cas d'une détection de piéton, il peut s'agir du délai entre l'acquisition d'une nouvelle image et la prise de décision sur l'action ou non du frein. Concrètement il s'agit du temps de réponse complet de l'algorithme.

Embarquabilité

Finalement, dans le cadre de cette thèse où nous traitons de l'embarquabilité des algorithmes, l'aspect temps-réel dur ou mou importe peu. Ainsi, étudier l'embarquabilité d'un algorithme sur une architecture de calcul donnée, revient à estimer si le temps d'exécution de l'algorithme sur cette architecture peut respecter l'ensemble des contraintes

temps-réel : rythme et latence. Dans le cadre de cette thèse nous nous focaliserons donc uniquement sur cette problématique.

Notons cependant que le respect théorique de ces contraintes est une condition nécessaire mais non suffisante pour garantir une exécution temps-réel. Cette garantie peut cependant s'obtenir à l'aide d'un système temps-réel dur correctement configuré et dimensionné, permettant de prendre en compte et de garantir le respect de l'ensemble des contraintes quelles que soient les conditions.

1.4.2 Besoin industriel

La définition de l'embarquabilité d'un algorithme étant maintenant posée, intéressons-nous à présent au besoin industriel autour de cette problématique. Tout d'abord, pour comprendre ce besoin, il convient de présenter le processus de conception d'un véhicule. Comme illustré dans la figure 1.7, ce processus se déroule en quatre phases : amont, développement, industrialisation et commercialisation.

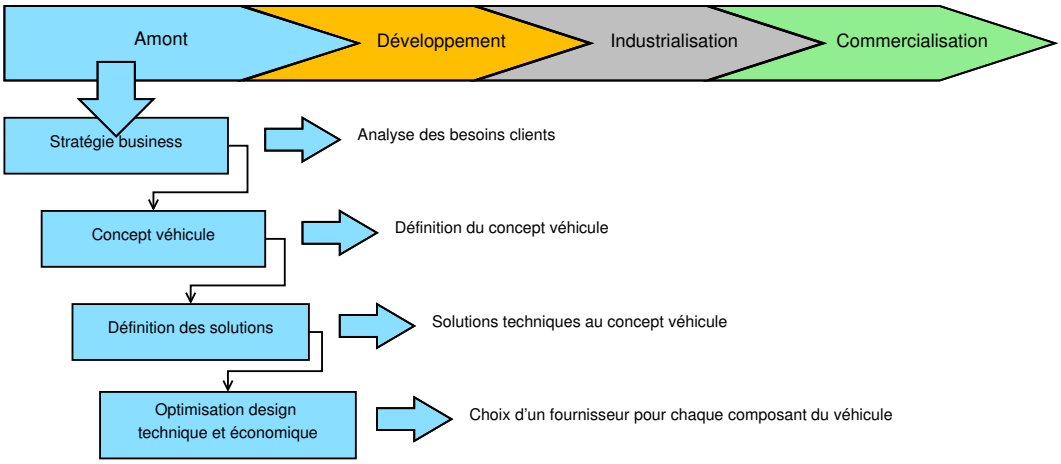


FIGURE 1.7 – Phases d'un projet véhicule allant de l'amont à la commercialisation. La phase amont peut se décomposer en quatre sous-phases : l'analyse des besoins clients, la définition du concept, le choix des solutions techniques et enfin l'optimisation technico-économique pour obtenir des réponses techniques qui soient économiquement les plus intéressantes possibles.

Fonctionnement actuel

Pour comprendre la problématique de l'embarquabilité des applications ADAS, concentrons-nous sur la phase amont et plus particulièrement les sous-phases de définition des solutions et d'optimisation. Ces deux étapes se déroulent conjointement avec un ensemble de fournisseurs automobiles. Le principe est alors le suivant, dans un premier temps on cherche d'abord à déterminer quelles solutions techniques peuvent être envisagées pour répondre au concept véhicule par le biais d'une *Request For Information* (RFI). Puis, dans un second temps, on détermine quel fournisseur propose la meilleure solution sur des critères technico-économiques par le biais d'une *Request For Quotation* (RFQ). Le processus complet est détaillé en figure 1.8.

Dans le cadre du choix d'un ECU pour la gestion des ADAS, il existe à l'heure actuelle deux problématiques. Tout d'abord il est très complexe de déterminer les caractéristiques techniques que doit avoir un ECU pour pouvoir embarquer un ensemble d'applications ADAS, notamment ses capacités de calcul pour que l'ensemble des contraintes

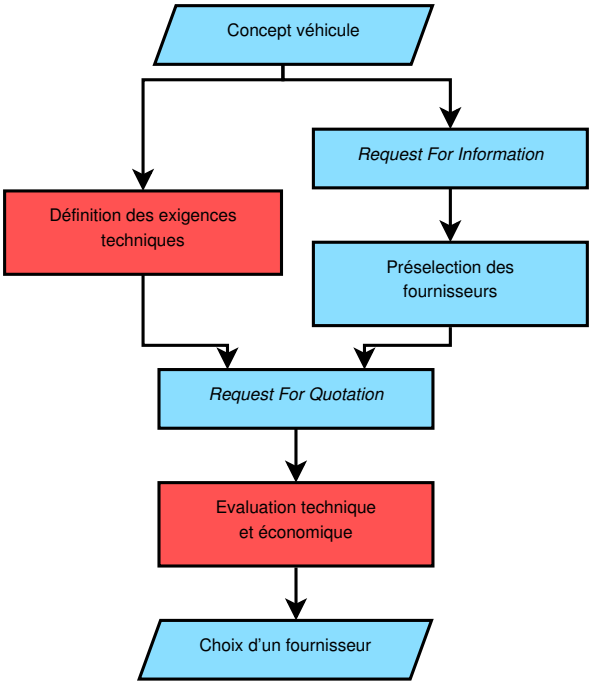


FIGURE 1.8 – Processus de définition des solutions techniques et d'optimisation technico-économique de la phase amont d'un projet véhicule. Les cases en rouge impliquent des problèmes techniques dans le cadre du choix d'un ECU pour des fonctionnalités ADAS.

temps-réel puissent être respectées. Enfin, déterminer le meilleur compromis technico-économique sur l'ensemble des propositions des fournisseurs répondant à la RFQ est également extrêmement complexe. Ainsi, il s'agit de déterminer le fournisseur proposant l'ECU le moins cher tout en s'assurant que cet ECU garantisse l'embarquabilité de l'ensemble des applications ADAS pour ce projet véhicule.

Cette problématique du dimensionnement de l'ECU devient de plus en plus critique avec l'augmentation de la complexité des applications ADAS et du besoin en calcul en résultant. Ainsi, jusqu'à aujourd'hui la plupart des traitements sont déportés dans les capteurs proposés par les différents fournisseurs (par exemple la caméra intelligente de Mobileye qui intègre toute la partie traitement d'images), la partie de l'application ADAS développée par le constructeur représente donc un très faible poids d'un point de vue calculatoire et est donc très facilement embarquable. Il s'agit généralement d'une simple loi de commande automobile permettant de piloter les différents actionneurs du véhicule en fonctions des informations renvoyées par les capteurs intelligents. Dans ce cas-là le choix de l'ECU reste simpliste et un surdimensionnent implique une hausse de coût restant très limitée et donc non problématique.

Cependant, les applications ADAS tendent à se complexifier, les constructeurs automobiles commencent à utiliser de plus en plus de capteurs, ce qui implique une part logicielle de plus en plus complexe, notamment dans le traitement des données issues des différents capteurs intelligents (par exemple fusion de données). Sur le court terme les acteurs automobiles ont donc besoin d'outils permettant de répondre à une nécessité d'un dimensionnement de l'ECU au plus juste pour garantir un coût minimal.

Vision à long terme et positionnement

Sur le plus long terme, le nombre de capteurs devrait encore augmenter pour pouvoir répondre à un besoin croissant à propos de la compréhension de l'environnement

du véhicule. Pour diminuer la redondance de la puissance de calcul et les coûts associés le choix des acteurs automobiles va s'orienter vers l'utilisation d'un calculateur unique et des capteurs fournissant des données brutes, non traitées (par exemple une caméra renvoyant des images par opposition aux capteurs intelligents Mobileye actuels). Dans ces conditions le dimensionnement de l'ECU devient extrêmement complexe et demande la capacité d'étudier l'embarquabilité d'applications comprenant le traitement d'un grand volume de données d'entrée, tel que les algorithmes de traitement d'images.

Dans ces conditions, nous estimons que la majeure partie des besoins en calcul est à imputer aux algorithmes de traitement d'images. Cette vision est d'ailleurs confirmée par une récente étude proposée par VELEZ et OTAEGUI [2016], concluant qu'il est essentiel de développer de nouvelles méthodologies et outils pour embarquer les algorithmes de traitement d'images. Dans le cadre de cette thèse, nous choisissons donc de nous positionner sur le long terme et de considérer le cas le plus complexe en proposant une méthodologie d'analyse de l'embarquabilité appliquée aux algorithmes de traitement d'images. Cependant, il pourra être envisagé par la suite de considérer l'application de la méthodologie pour d'autres types d'algorithme que le traitement d'images.

Cas des architectures hétérogènes

D'après notre discussion sur les différents types d'architectures de calcul existant présenté dans la partie 1.3.3, il n'est pas possible de trouver un type d'architecture pouvant proposer des capacités de calcul suffisantes pour exécuter un algorithme de traitement d'image complexe et étant suffisamment générique pour pouvoir exécuter d'autres algorithmes comme de la fusion de données ou des lois de commande automobile. Finalement, la solution envisagée par l'ensemble des acteurs automobiles est de proposer des architectures à hardware mixte (ou hétérogènes).

Ces architectures hétérogènes ont pour avantage de pouvoir proposer à la fois une grande flexibilité (processeurs génériques) et une grande puissance de calcul (processeurs spécialisés). Cependant l'hétérogénéité amène une nouvelle problématique : la répartition des charges de calcul entre les différents processeurs d'un même SoC. Ainsi, cette répartition à un impact direct sur le temps d'exécution d'un algorithme, il devient donc très complexe de caractériser l'embarquabilité d'un algorithme sur une architecture hétérogène. Pour un algorithme donné, déterminer la répartition permettant d'obtenir le temps d'exécution le plus bas est loin d'être trivial. Ainsi il faut prendre en compte les temps d'exécution sur chacun des processeurs, les temps de transfert, etc. Cette problématique est désignée par le *mapping* des tâches d'un algorithme sur une architecture hétérogène.

1.5 Démarche proposée

Dans le cadre de cette thèse, nous proposons donc la définition d'une méthodologie permettant l'analyse de l'embarquabilité d'algorithmes de traitement d'images sur des architectures hétérogènes. Cette méthodologie permet de répondre sur l'aspect du respect des contraintes temps-réel et de la problématique de *mapping* des charges de calcul sur un SoC hétérogène.

La suite du manuscrit est organisée de la manière suivante, nous proposons dans un premier temps une présentation générale autour des algorithmes de traitement d'images et des architectures hétérogènes dans le chapitre 2. Cette présentation sera suivie d'une discussion autour de l'évolution et des limitations des processeurs.

Une fois les bases nécessaires à la compréhension du manuscrit présentées, la méthodologie globale d'analyse de l'embarquabilité sera abordée dans le chapitre 3. Nous présenterons dans ce chapitre notre modélisation d'un algorithme et de ses contraintes temps-réel. Enfin, trois heuristiques seront présentées pour étudier l'embarquabilité des algorithmes et proposer une solution à la problématique de *mapping* dans le cas de SoCs hétérogènes.

Notre approche d'analyse de l'embarquabilité se construit autour de deux pierres angulaires correspondant à une méthode de caractérisation d'architecture de calcul et une méthodologie de prédictions de performance multi-architecture. Au cours du chapitre 4, nous présenterons notre méthodologie de caractérisation d'architectures hétérogènes. Cette méthodologie est axée autour d'un ensemble de vecteurs test catégorisés suivant trois niveaux : *low-level*, *mid-level* et *high-level*. Ces vecteurs de test permettent l'extraction semi-automatique de caractéristiques d'architectures hétérogènes. Ces caractéristiques peuvent alors être utilisées dans notre approche de prédiction de performances, présentée dans le chapitre 5.

Enfin, pour illustrer le fonctionnement de notre méthodologie ainsi que ces avantages, nous présenterons dans le chapitre 6 deux exemples correspondant à des applications automobiles. Il s'agit d'un algorithme d'odométrie visuelle utilisé pour des problématiques de localisation et d'un algorithme de détection de piétons pour une application d'AEB.

Finalement, une discussion autour de l'applicabilité et des ouvertures possibles de la méthodologie sera abordée dans le chapitre 7. Bien évidemment, nous insisterons sur les applications industrielles pouvant être mises en place au sein de Renault.

1.6 Références

AMDAHL, G. M. 1967, «Validity of the single processor approach to achieving large scale computing capabilities», dans *Proc. of the Spring Joint Computer Conference*, ACM, p. 483–485. 14

FLYNN, M. J. 1972, «Some computer organizations and their effectiveness», *IEEE transactions on computers*, vol. 100, nº 9, p. 948–960. 13

LITMAN, T. 2014, «Autonomous vehicle implementation predictions», *Victoria Transport Policy Institute*, vol. 28. 3

NVIDIA. 2015, «Cuda C Programming Guide», . 18

OGAWA, T. et K. TAKAGI. 2006, «Lane recognition using on-vehicle lidar», dans *2006 IEEE Intelligent Vehicles Symposium*, IEEE, p. 540–545. 7

ONISR. 2015, «Bilan de la sécurité routière 2015», cahier de recherche, ONISR. 3, 4

PARVIS, M. et A. CARULLO. 2001, «An ultrasonic sensor for distance measurement in automotive applications», *IEEE Sensors Journal*, vol. 1, nº 2, p. 143–147. 7

PREMEBIDA, C., G. MONTEIRO, U. NUNES et P. PEIXOTO. 2007, «A lidar and vision-based approach for pedestrian and vehicle detection and tracking», dans *2007 IEEE Intelligent Transportation Systems Conference (ITSC)*, IEEE, p. 1044–1049. 7

RASSHOFER, R. et K. GRESSER. 2005, «Automotive radar and lidar systems for next generation driver assistance functions», *Advances in Radio Science*, vol. 3, nº B. 4, p. 205–209. 7

SAUSSARD, R., B. BOUZID, M. VASILIU et R. REYNAUD. 2015, «The embeddability of lane detection algorithms on heterogeneous architectures», dans *2015 IEEE International Conference on Image Processing (ICIP)*, IEEE, p. 4694–4697. 19

VELEZ, G. et O. OTAEGUI. 2016, «Embedding vision-based advanced driver assistance systems : a survey», *IET Intelligent Transport Systems*. 23

Chapitre 2

Traitement d'images et architectures hétérogènes

« *If the auto industry advanced as rapidly as the semiconductor industry, a Rolls Royce would get half a million miles per gallon, and it would be cheaper to throw it away than to park it.* »

Gordon Earle Moore

Sommaire

2.1 Introduction

Avant d'aborder le cœur de la thèse, il nous parait indispensable de présenter au cours de ce chapitre les connaissances nécessaires à la compréhension du manuscrit. Ainsi, la thèse traite des algorithmes des traitements d'images pour les applications ADAS. Nous avons présenté au cours du chapitre précédent ce que sont les ADAS, leurs intérêts, applications et nous avons également identifié que certains de ces systèmes ont besoin du traitement des images pour fonctionner, notamment ceux basés sur une ou plusieurs caméras. La thèse se focalise sur l'embarquabilité de ces algorithmes sur des architectures à hardware mixte, également dénommées architectures hétérogènes. Après avoir présenté au cours du chapitre précédent l'état de l'art des technologies des processeurs et leur fonctionnement, nous proposons de nous focaliser, au cours de ce chapitre, sur les architectures hétérogènes pour les ADAS, leurs spécificités et les outils logiciels associés. Enfin, il nous parait indispensable d'introduire une notion clé pour la compréhension de la suite du manuscrit : l'intensité arithmétique (I_A), que nous présenterons au travers d'une analyse appuyée sur les limitations des processeurs au cours du temps et des stratégies de contournement.

2.2 Traitement d'images pour les ADAS

Le domaine du traitement d'images est très vaste, il serait très prétentieux de notre part de vouloir proposer une présentation exhaustive des algorithmes et des applications. Nous allons donc nous contenter de présenter spécifiquement les algorithmes de traitements d'images pouvant être rencontrés dans le monde des ADAS, en portant une attention toute particulière aux algorithmes abordés dans les chapitres suivants de ce manuscrit. Plus d'informations à propos du traitement d'images peuvent être trouvées dans [SZELISKI, 2011].

2.2.1 Calibration

En traitement d'images, la calibration est un processus permettant d'estimer les paramètres qui caractérisent une caméra réelle. Le but de cette opération est de pouvoir transformer les images issues de la caméra réelle en images qui seraient obtenues par une caméra idéale, conforme au modèle *pinhole*. Ce modèle décrit la relation mathématique entre un point 3D de l'espace et sa projection 2D sur le plan image. Le modèle ne prend pas en compte les artefacts causés par la lentille comme la distorsion géométrique. La phase de rectification permet justement de corriger les défauts causés par la lentille, on parle de distorsion radiale. Pour cette section, la principale source d'information est [HARTLEY et ZISSERMAN, 2000].

Modèle pinhole

Commençons tout d'abord par définir les repères utilisés, tels qu'illustrés dans la figure 2.1 :

— $(W, \vec{x_w}, \vec{y_w}, \vec{z_w})$: le repère du Monde

— $(C, \vec{x_c}, \vec{y_c}, \vec{z_c})$: le repère de la caméra avec C le centre optique.

— $(I, \vec{u}, \vec{v})$: le repère de l'image

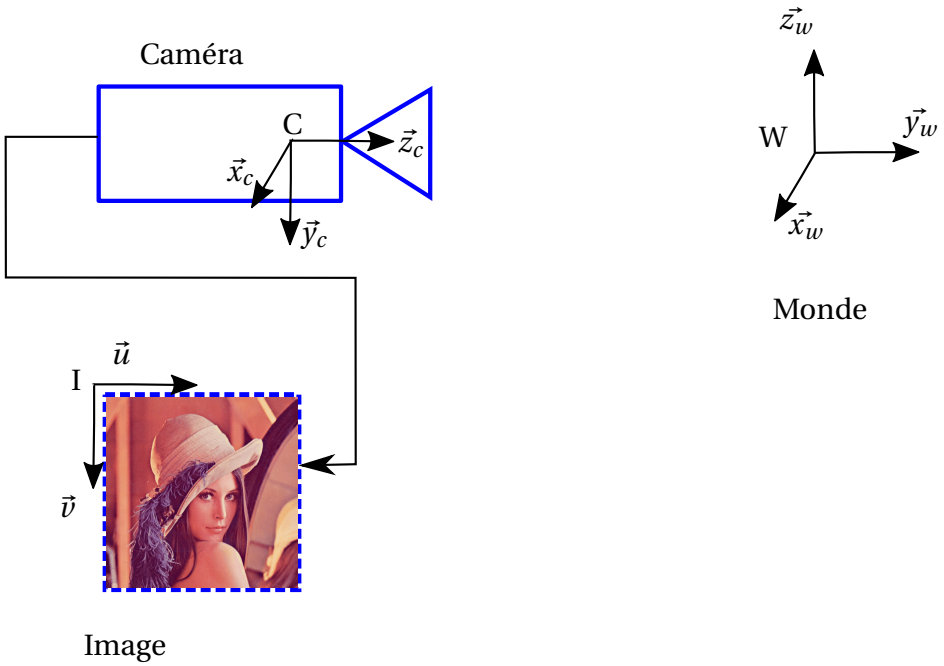


FIGURE 2.1 – Repères utilisés et une image de la très célèbre Lenna [MUNSON, 1996].

Notons aussi f la distance qui sépare le plan image du centre optique de la caméra, aussi appelé la distance focale de la caméra.

Soient un point $M_W(x, y, z, 1)$ dans W et $M_I(su, sv, s)$ sa projection sur l'image en coordonnées homogènes. Le modèle *pinhole* nous donne la relation suivante :

$$\begin{pmatrix} su \\ sv \\ s \end{pmatrix}_I = \begin{bmatrix} f_u & 0 & c_u & 0 \\ 0 & f_v & c_v & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} & R_{3\times3} & & t_x \\ & & & t_y \\ & & & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}_W \tag{2.1}$$

Les paramètres utilisés dans ce modèle sont divisés en deux catégories : les paramètres intrinsèques, qui sont internes à la caméra, et les paramètres extrinsèques, qui correspondent à la matrice de passage entre le repère du Monde et celui de la caméra. On appelle les paramètres intrinsèques de la caméra :

— f_u et f_v : la distance focale exprimée en largeur et hauteur de pixels (pas forcément carrés).

— c_u et c_v : les coordonnées du centre optique de la caméra sur le plan image, en pixel.

Remarquons que de manière générale les paramètres intrinsèques d'une caméra sont représentés par une matrice notée K et définie tel que :

$$K = \begin{bmatrix} f_u & 0 & c_u \\ 0 & f_v & c_v \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

Les paramètres extrinsèques sont :

— $R_{3\times3}$: la matrice de rotation permettant le passage du repère du Monde au repère caméra.

— t_x, t_y, t_z : la translation permettant le passage du repère du Monde au repère caméra, avec le vecteur $t = -R \times C_W$.

Lorsque l'on ne travaille qu'avec une seule caméra, il est possible de définir le repère du Monde comme étant confondu avec celui de la caméra. De cette manière les paramètres extrinsèques sont la matrice identité pour la rotation et un vecteur nul pour la translation. Calibrer des caméras revient à estimer les paramètres intrinsèques et extrinsèques d'une ou d'un réseau de caméras.

Distorsion géométrique

La distorsion est une altération géométrique causée par les lentilles qui a pour effet de ne pas conserver les propriétés géométriques des objets. De manière générale, on distingue deux types d'altérations dues à la lentille :

radiale : causée par l'asymétrie des lentilles.

tangentielle : causée par le mauvais alignement entre la lentille et le plan image.

Les altérations de type radiale sont très visibles sur les caméras à faible focale, à grand angle ou les caméras *fisheyes*, comme montré dans la figure 2.2.

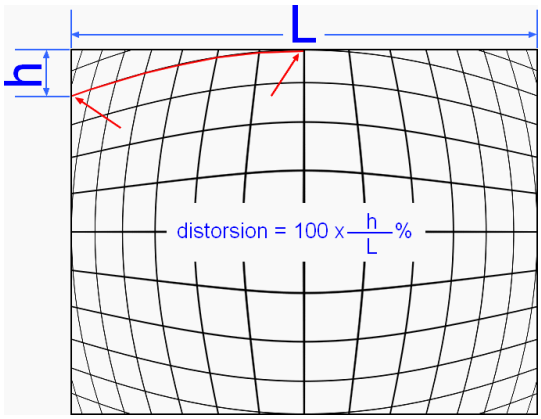


FIGURE 2.2 – Un quadrillage sous l'effet de la distorsion radiale, les lignes droites sont déformées et apparaissent comme courbes sur l'image.

Une caméra fisheye est très avantageuse puisqu'elle permet de couvrir un très grand angle de vue, pour être placée dans les angles morts d'une voiture par exemple. Il paraît cependant très discutable de renvoyer une image déformée à l'utilisateur, il est préférable de corriger cette altération en amont. Pour ce faire, il suffit de modéliser la distorsion radiale puis d'appliquer l'effet inverse. De manière générale, elle peut se modéliser par :

$$\begin{pmatrix} x_d \\ y_d \end{pmatrix} = f(\tilde{r}) \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} \tag{2.3}$$

avec :

— $(\tilde{x}, \tilde{y})$: coordonnée de l'image idéale qui obéit à la projection linéaire,

— (x_d, y_d) : coordonnée réelle de l'image, soumise à la distorsion,

— $\tilde{r}$: la distance radiale $\sqrt{(\tilde{x}, \tilde{y})}$ depuis le centre de la distorsion,

— $f(\tilde{r})$: le facteur de distorsion, qui est une fonction de $\tilde{r}$.

D'autres modèles sont également référencés par HUGHES et collab. [2009].

La correction de la distorsion radiale se fait donc en appliquant l'inverse du modèle, ainsi :

$$\hat{x} = x_c + L(r^2) \times (x - x_c) \quad \hat{y} = y_c + L(r^2) \times (y - y_c) \tag{2.4}$$

Avec (x, y) les coordonnées mesurées, $(\hat{x}, \hat{y})$ les coordonnées corrigées, (x_c, y_c) le centre de la distorsion et $r^2 = (x - x_c)^2 + (y - y_c)^2$. Pour simplifier la calibration, il est d'usage d'approximer la fonction en série de Taylor $L(r^2) = 1 + k_1 r^2 + k_2 r^4 + k_3 r^6 + \ldots$. Les coefficients $\{k_1, k_2, k_3, \ldots, x_c, y_c\}$ sont considérés comme des paramètres de calibration pour la caméra. Le centre de la caméra est souvent considéré comme étant aussi le centre de la distorsion radiale.

Estimation des paramètres intrinsèques et de la distorsion radiale

Les paramètres intrinsèques et de distorsion radiale ne dépendent que de la caméra et de son système optique associé. Une fois fixée, une seule calibration suffit. Pour être capable d'estimer ces paramètres, il suffit de projeter sur le plan image un motif connu. De manière générale, c'est un damier noir et blanc qui est utilisé, comme illustré en figure 2.3. Le damier a pour avantage d'avoir une forme très simple (un ensemble de carrés) et il peut être assez facilement détecté sur une image, comme le montre par exemple RUFLI et collab. [2008]. L'idée est donc d'observer le résultat de la projection sur le plan image du damier en différentes positions. Puis en connaissant les propriétés de celui-ci (dimension des carrés, nombre, etc.), de calculer les paramètres qui décrivent aux mieux la projection. Le calcul se fait par la minimisation de la distance entre le pixel réel observé et la projection théorique calculée. Pour plus de détails, une méthode est donnée par ZHANG [2000].

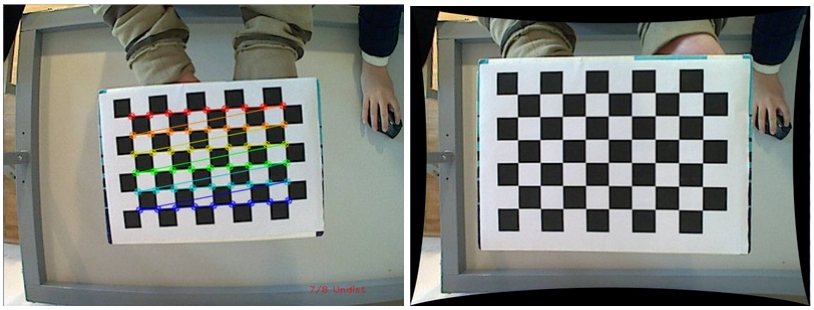


FIGURE 2.3 – Images d'un damier de calibration avant et après correction de la distorsion.

Ainsi, si l'on prend comme repère du Monde le damier de calibration, N images et M points du damier, l'estimation des paramètres de la caméra consiste à minimiser la fonction suivante :

$$\sum_{i=0}^{N-1} \sum_{j=0}^{M-1} ||m_{ij} - \hat{m}(K, k_1, k_2, R_i, t_i, M_j)||^2 \tag{2.5}$$

où $\hat{m}(K, k_1, k_2, R_i, t_i, M_j)$ désigne la projection théorique du point M_j sur l'image i et m_{ij} sa projection réelle. Les paramètres intrinsèques de la caméra étant constants au cours du temps (pas de modification de l'optique), la calibration ne se fait qu'une seule fois.

Pour résumer en quelques lignes, la procédure de calibration se déroule de la manière suivante :

1. Imprimer un damier de calibration et le fixer sur une surface plane.
2. Prendre plusieurs images du damier sous différentes orientations en insistant sur les coins de l'image (là où la distorsion radiale est la plus forte).
3. Détecter les coins du damier pour chaque image.
4. Déterminer les paramètres intrinsèques de la caméra :
   (a) Estimation des matrices K, R_i et t_i.

(b) Estimation des paramètres de la distorsion radiale k_1, k_2 en se basant sur l'estimation précédente.

(c) Affinement du résultat en minimisant l'équation 2.5.

Correction de la distortion

À partir des paramètres intrinsèques de la caméra obtenus à l'aide de la procédure calibration, on est capable de prédire la position idéale d'un pixel (x_u, y_u) en fonction des données obtenues avec l'image sans la correction de la distorsion $I_d(x_d, y_d)$:

$$\begin{cases} r_u = r_d \times (1 + k_1 r_d^2 + k_2 r_d^4) \\ x_u = x_c + (x_d - x_c) \times (1 + k_1 r_d^2 + k_2 r_d^4) \\ y_u = y_c + (y_d - y_c) \times (1 + k_1 r_d^2 + k_2 r_d^4) \\ r^2 = (x - x_c)^2 + (y - y_c)^2 \end{cases} \tag{2.6}$$

Ainsi, on aura $I_u(x_u, y_u) = I_d(x_d, y_d)$, avec I_u l'image corrigée.

Le problème réside dans le fait que la correction renvoie une coordonnée (x_u, y_u) qui est composée de valeurs non entières. Plusieurs possibilités s'offrent à nous pour obtenir l'image finale $I_u(i, j)$, soit arrondir chacun des (x_u, y_u) ou bien procéder à une interpolation pour déterminer la valeur de chaque pixel de l'image finale. Arrondir le résultat à l'entier le plus proche est très peu coûteux, mais sera aussi très imprécis sur le résultat.

Si l'on prend le problème de manière inverse, c'est-à-dire que l'on cherche g tel que $I_d(x_d, y_d) = g(i, j)$, (i, j) étant les coordonnées de l'image corrigée. Comme précédemment, les coordonnées retournées par la fonction sont aussi non entières, mais comme il s'agit de l'image déformée (image source) les valeurs des pixels voisins sont connues. Ainsi, comme illustré en figure 2.4, on peut donc calculer la valeur du pixel en coordonnée non entière par une interpolation bilinéaire :

$$\begin{aligned} &I_d(x_d, y_d) = \\ &(1 - F(x_d)) \times (1 - F(y_d)) \times I_d(E(x_d), E(y_d)) + \\ &F(x_d) \times (1 - F(y_d)) \times I_d(E(x_d) + 1, E(y_d)) + \\ &(1 - F(x_d)) \times F(y_d) \times I_d(E(x_d), E(y_d + 1)) + \\ &F(x_d) \times F(x_d) \times I_d(E(x_d + 1), E(y_d + 1)) \end{aligned} \tag{2.7}$$

avec $E(x)$ la partie entière de x et $F(x) = x - E(x)$. En ayant $I(i, j) = I_d(x_d, y_d)$, on est alors capable de déterminer la valeur de chacun des pixels de l'image finale par interpolation sur l'image déformée. Pour optimiser les calculs, il est bénéfique de garder en mémoire $E(x)$ comme un entier et $F(x)$ comme un nombre en virgule fixe sur un octet (variant entre 0 et $\frac{255}{256}$).

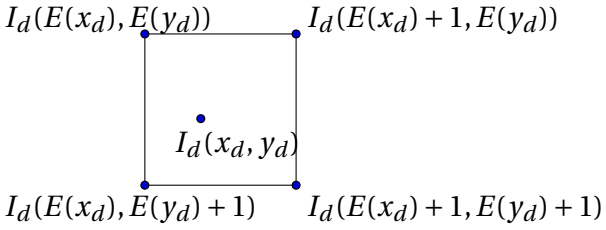


FIGURE 2.4 – Coordonnée non entière de l'image et ses pixels adjacents

Un problème reste cependant à soulever, s'il est aisé de trouver une coordonnée sur l'image rectifiée à partir des coordonnées de l'image déformée (donnée par le modèle

de la distorsion), l'inverse n'est pas trivial. Ainsi les paramètres donnant la relation $r_u = r_d \times (1 + k_1 r_d^2 + k_2 r_d^4)$ sont connus, mais pas ceux permettant d'exprimer r_d en fonction de r_u. Pour pallier ce problème, ROMERO et GOMEZ [2007] utilisent une méthode itérative pour trouver le r_d permettant de respecter l'égalité du modèle avec un r_u^2 donné. Cela revient donc à résoudre l'équation suivante par dichotomie :

$$r_u - r_d \times (1 + k_1 r_d^2 + k_2 r_d^4) = 0 \tag{2.8}$$

Une autre méthode consiste à estimer r_d de manière itérative :

$$r_d = \frac{r_u}{(1 + k_1 r_d^2 + k_2 r_d^4)} \tag{2.9}$$

En général, 10 itérations permettent de converger vers un résultat proche de la réalité.

On obtient alors le r_d correspondant au r_u donné. Il est alors aisé de calculer le couple (x_d, y_d) :

$$\begin{cases} x_d = x_c + \dfrac{x_u - x_c}{1 + k_1 r_d^2 + k_2 r_d^4} = x_c + \dfrac{r_d}{r_u} \times (x_u - x_c) \text{ si } r_u \neq 0 \\ y_d = y_c + \dfrac{y_u - y_c}{1 + k_1 r_d^2 + k_2 r_d^4} = y_c + \dfrac{r_d}{r_u} \times (y_u - y_c) \text{ si } r_u \neq 0 \end{cases} \tag{2.10}$$

Pour résumer en quelques lignes, l'algorithme de correction est défini de la manière suivante, pour chaque point (i, j) de l'image corrigée :

1. Calculer $r_u = \sqrt{(i - x_c)^2 + (j - y_c)^2}$, puis déterminer le r_d par l'une des deux méthodes. Remarquons qu'il est inutile de répéter l'opération sur toute l'image, la fonction carré étant symétrique, seul un quart de l'image suffira.
2. Déterminer la valeur du pixel $I(i, j)$ en utilisant l'interpolation bilinéaire donnée en 2.7. Si la correspondance de (i, j) fait intervenir des pixels en dehors de l'image, $I(i, j) = 0$.

Le résultat de la correction est donné en figure 2.5.

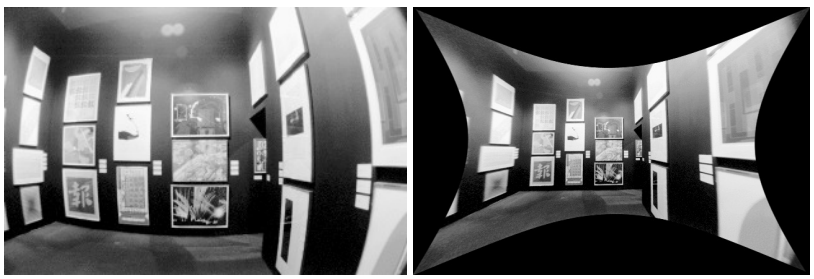


FIGURE 2.5 – Correction de la distortion sur une image issue d'une caméra de type fisheye. On remarque que sur l'image corrigée, les pixels dont la valeur est inconnue sont remplacées par des pixels noirs, c'est-à-dire $I = 0$.

2.2.2 Algorithmes de type pixelique (dense)

Nous avons donc vu comment se modélise la formation d'une image par une caméra à l'aide du modèle *pinhole* et comment obtenir les différents paramètres la caractérisant (intrinsèques et extrinsèques). Par définition, un algorithme de traitement d'images prend en entrée une ou plusieurs images mais peut avoir plusieurs types de soties :

— l'algorithme retourne une image correspondant à une transformation de l'image d'entrée, par exemple la compression ;

— l'algorithme retourne une information ne correspondant pas à une image, comme la détection d'un objet, un ensemble de coordonnées, etc.

Par ailleurs, une image consiste en un ensemble de pixels, pour procéder au traitement deux stratégies sont possibles :

— on utilise l'ensemble des pixels de l'image complète ou d'une région de celle-ci pour effectuer le traitement, il s'agit alors d'un algorithme dit de type *dense* ;

— on utilise seulement un ensemble de points d'intérêts correspondant à des pixels particuliers de l'image (par exemple les coins détectés sur l'image), il s'agit alors d'un algorithme dit de type *sparse*

Au cours de cette thèse, nous n'avons pas effectué d'expérimentation sur des algorithmes de types *sparse*. Dans le cadre de ce manuscrit, nous avons donc choisi de présenter uniquement des algorithmes de type *dense*. Remarquons que ces derniers nécessitent plus de puissance de calcul puisqu'ils ont plus de données à traiter, ils nous paraissent donc plus intéressants dans une démarche de l'analyse de l'embarquabilité.

Convolution

En traitement du signal, un produit de convolution consiste en un produit effectué dans l'espace de Fourrier. De manière plus concrète, il s'agit de la somme du produit du signal d'entrée par une fonction de convolution pour tous les échantillons du signal. Le produit de convolution est symbolisé par « $*$ ». Pour le traitement de l'image, on appelle masque la matrice par laquelle on souhaite convoluer l'image. Ainsi, soient un masque M de taille 3×3, I l'image à filtrer et O l'image résultat. La convolution s'effectue de la manière suivante :

1. Placer le centre du masque sur un pixel (i, j) de l'image, avec le reste du masque recouvrant les pixels voisins.
2. Multiplier chaque coefficient du masque par le pixel opposé par rapport au centre et sommer les résultats.
3. Placer le résultat sur le pixel (i, j) de l'image résultat.
4. Répéter le résultat pour chaque pixel de l'image.

Ainsi, si :

$$M = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \tag{2.11}$$

on aura alors pour le pixel (i, j) :

$$O(i, j) = I(i-1, j-1) \times a_9 + \cdots + I(i, j) \times a_5 + \cdots + I(i+1, j+1) \times a_1 \tag{2.12}$$

L'opération est ainsi répétée pour chaque pixel (i, j) de l'image.

Une attention particulière doit être apportée pour les pixels en bordure de l'image. Ainsi, l'opération nécessite des valeurs qui n'existent pas. Plusieurs solutions sont possibles :

— Étendre les valeurs des bordures, pour ne pas altérer la dynamique de l'image
$\rightarrow I(-1, j) = I(0, j)$.

— Utiliser les valeurs à l'autre extrémité de l'image
$\rightarrow I(-1, j) = I(I.width - 1, j)$.

— Considérer qu'en dehors de l'image, toutes les valeurs sont nulles
→ $I(-1, j) = 0$.

L'effet provoqué par le masque dépendra de sa taille et de ses composantes. On peut cependant énumérer un certain nombre de filtres :

Filtre moyenneur : les valeurs du masques sont égales, la valeur du pixel sera donc égale à une moyenne avec ses pixels voisins. Cela aura pour effet de flouter l'image.

$$M_{moyenne} = 1/9 \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.13}$$

Filtre Gaussien : les coefficients sont donnés par l'équation d'une gaussienne 2D :

$$g(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) \tag{2.14}$$

Ce filtre est utilisé pour lisser l'image, l'effet dépendra de la taille et du σ. On parle aussi de flou gaussien.

Filtre Sobel : utilisé pour la détection de contour, les coefficients sont calculés à partir de l'opération de gradient. Il existe un masque pour le gradient en x et un autre en y.

$$M_{Sobel_x} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad M_{Sobel_y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{2.15}$$

Rehaussement de contour :

$$M_{Sharpen} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{2.16}$$

Une illustration de l'effet des différents filtres est donnée en figure 2.6.

La convolution étant associative et commutative, il est parfois possible de séparer un masque en deux, par exemple pour le filtre de Sobel :

$$M_{Sobel_x} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \tag{2.17}$$

Cette opération n'est pas applicable pour tous les filtres, on parlera de filtre séparable ou non séparable. Pour accélérer les calculs, il est préférable, si possible, de séparer un masque $N \times M$ en deux masques de taille $N \times 1$ et $1 \times M$. En effet, dans le premier cas, on effectuera environ $N \times M \times Nb_{pixels}$ opérations contre $(N + M) \times Nb_{pixels}$ avec une séparation, soit un gain de $N \times M/(N + M)$.

Certains algorithmes utilisent plusieurs opérations avec plusieurs masques pour effectuer un traitement. C'est le cas du filtre de CANNY [1986]. Ce filtre est utilisé pour la détection de contour de manière à respecter trois critères :

— Faible taux d'erreur dans la signalisation des contours.

— Distance minimum entre l'emplacement les contours détectés et la réalité.

— Une seule réponse par contour et peu de faux positifs.

L'algorithme procède en plusieurs étapes :

FIGURE 2.6 – Différents filtres de convolution. De gauche à droite puis de haut en bas : l'image original, filtre Gaussien, rehaussement de contour, filtre de Sobel en x.

1. Réduction du bruit de l'image en utilisant un filtre Gaussien, par exemple à l'aide d'un filtre de taille 5 :

$$M = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} \tag{2.18}$$

2. Appliquer un filtre de Sobel dans les deux directions (x et y) de manière à obtenir G_x et G_y. Puis calculer la norme du gradient de l'image : $G(i,j) = \sqrt{G_x^2(i,j) + G_y^2(i,j)}$ ainsi que la direction du gradient $\theta(i,j) = \arctan\left(\frac{G_y(i,j)}{G_x(i,j)}\right)$, arrondi à 45° près.
3. Suppression des non-maximas : pour chaque pixel, on compare $G(i,j)$ avec les deux pixels voisins dans la direction du gradient ; si la valeur du gradient est supérieure aux deux autres alors la valeur est préservée, sinon elle est supprimée.
4. Déclenchement par hysteresis, chaque valeur est comparée à deux seuils δ_{min} et δ_{max} :
   - Si $G(i,j) > \delta_{max}$ alors le pixel est considéré comme appartenant au contour.
   - Si $G(i,j) < \delta_{min}$ alors le pixel est éliminé (valeur à 0).
   - Si $\delta_{min} \leq G(i,j) \leq \delta_{max}$, le pixel est gardé seulement s'il est connecté à un pixel supérieur à δ_{max}.

De manière plus simple, il existe aussi l'opération appelée *Difference of Gaussian* (DoG). Il s'agit tout simplement d'appliquer deux filtres gaussiens sur l'image, σ_1 et σ_2, puis de faire la différence des deux images résultantes (soustraction pixel à pixel). On peut donc en choisissant deux σ adaptés garder une bande de fréquence particulière sur l'image résultat, comme illustré en figure 2.7. Remarquons que le filtre DoG produit une image signée, c'est-à-dire comprenant des pixels avec une valeur négative.

FIGURE 2.7 – Résultat d'un filtre DoG $\sigma_1 = 1$ et $\sigma_2 = 3$.

Algorithmes morphologiques

Les opérateurs morphologiques ont été définis à l'origine pour agir sur des images binaires, après thesholding. Les opérateurs agissent par le biais d'éléments structurants. Il s'agit d'une forme utilisée pour interagir avec l'image donnée, dans l'idée d'observer les correspondances entre cet élément structurant et l'image. L'élément peut être de forme carré, octogonal, disque, ligne...

De manière général, les algorithmes morphologiques sont définis de la manière suivante :

1. Placer le centre l'élément structurant sur le pixel (i, j) de l'image.
2. Effectuer l'opération binaire défini par l'opérateur pour tous les pixels inclus dans l'élément structurant.
3. Placer le résultat dans pixel (i, j) de l'image de sortie.
4. Répéter l'opération pour tous les pixels de l'image.

Il existe quatre opérateurs majeurs, définis avec un élément structurant B et I l'image d'entrée :

Dilatation : notée $\delta_B(I)$ ou $I \oplus B$. Il s'agit d'appliquer l'opérateur binaire *OR*. Ainsi le pixel (i, j) sera égal à 1 si ou moins un pixel inclus dans l'élément structurant est égal à 1.

Érosion : notée $\epsilon_B(I)$ ou $I \ominus B$. Il s'agit d'appliquer l'opérateur binaire *AND*. Ainsi le pixel (i, j) sera égal à 1 si et seulement si tout les pixels inclus dans l'élément structurant sont égaux à 1.

Ouverture : notée $\gamma_B(I)$, correspond à l'application successive d'une érosion puis une dilatation sur l'image.

$$\gamma_B(I) = \delta_B(\epsilon_B(I)) \tag{2.19}$$

Fermeture : notée $\phi_B(I)$, correspond à l'application successive d'une dilatation, puis une érosion sur l'image.

$$\phi_B(I) = \epsilon_B(\delta_B(I)) \tag{2.20}$$

Une illustration des différents opérateurs sur une image binaire est donnée en figure 2.8.

Pour des images en niveaux de gris, l'opérateur *AND* est remplacé par un *max* sur l'ensemble appartenant à l'élément structurant. Le *OR* est remplacé par un *min*. La figure

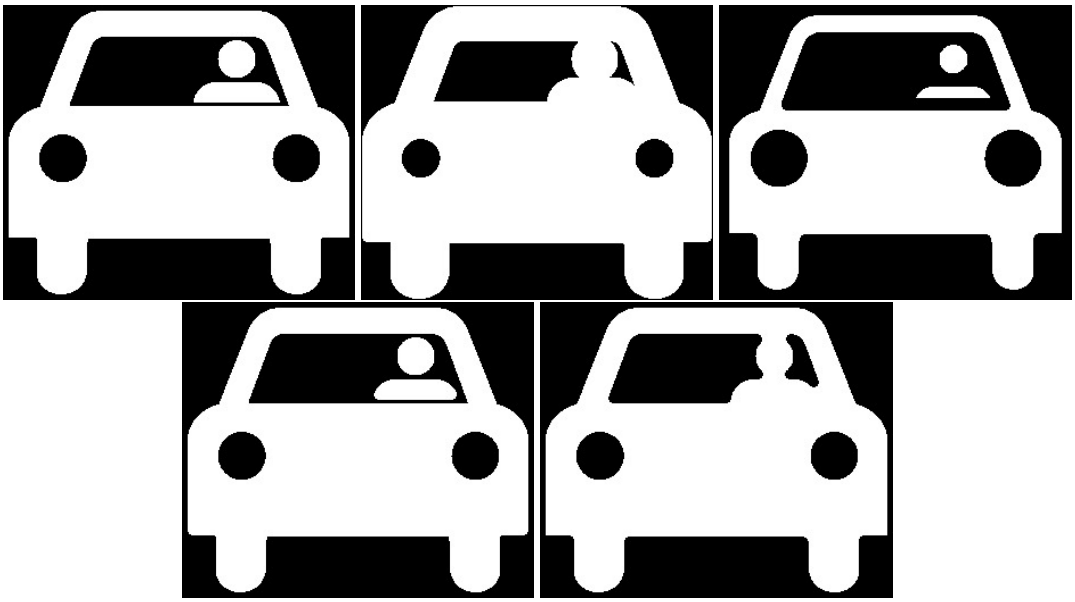


FIGURE 2.8 – Opérations Morphologiques de base pour des images binaires. De gauche à droite puis de haut en bas : image original, dilatation, érosion, ouverture, fermeture. Les opérations ont été faites avec un élément structurant de forme disque de rayon 3.5 pixels.

2.9 illustre l'effet des opérateurs morphologiques de bases sur une image en niveaux de gris.

FIGURE 2.9 – Opérations morphologiques de base pour des images en niveaux de gris. De gauche à droite puis de haut en bas : image original, dilatation, érosion, ouverture, fermeture. Les opérations ont été faites avec un élément structurant de forme disque de rayon 3.5 pixels.

À partir de ces opérateurs de base, il devient possible de définir des filtres un peu plus complexes comme le lissage de l'image ou une détection de contour. Un liste non exhaustive est donnée en figure 2.10. Le filtre *Top Hat* consiste à renvoyer les détails de l'image à la fois plus petits que l'élément structurant et plus lumineux que leurs voisins. Le *Bottom Hat* agit de même avec les pixels plus sombres. Pour plus de détails et de filtres utilisant les opérateurs morphologiques, voir [VINCENT, 1993].

Si l'on applique de manière naïve l'opérateur de morphologie, $Nb_{points\ du\ masque}$ comparaisons seront nécessaires pour chaque pixel. Il existe cependant des méthodes permettant une optimisation de l'algorithme et donc de diminuer le nombre d'opérations par pixel. URBACH et WILKINSON [2008] proposent un algorithme effectuant seulement

Nom du filtre	**Formule mathématique**	**Résultat**
Edge In	$I - \epsilon_B(I)$	
Edge Out	$\delta_B(I) - I$	
Edge	$\delta_B(I) - \epsilon_B(I)$	
Lissage	$\phi_B(\gamma_B(I))$	
Top Hat	$I - \gamma_B(I)$	
Bottom Hat	$\phi_B(I) - I$	

FIGURE 2.10 – Illustration de différents filtres morphologiques. Les filtres de type *Edge* consiste à une détection de contour, le lissage floute l'image. Le filtre *Top Hat* renvoie les détails de l'image à la fois plus petits que l'élément structurant et plus lumineux que leurs voisins. Le *Bottom Hat* agit de même avec les pixels plus sombres.

$Hauteur\ du\ masque + Largeur\ du\ masque$ opérations par pixel pour des masques de formes quelconques, mais avec une consommation mémoire plus importante.

Homographie

D'un point de vue mathématique, une homographie est une transformation linéaire entre deux plans projectifs. Ainsi, comme le montre le modèle *pinhole*, l'ensemble des points P_i appartenant à un plan π de l'espace seront projetés sur le plan image de la caméra C_a à la coordonnée p_i^a tel que :

$$p_i^a = K_a \cdot P_i^a \tag{2.21}$$

où P_i^a est la coordonnée du point P_i dans le repère de la caméra C_a. Prenons maintenant une autre caméra, C_b tel que la transformation de C_a à C_b soit donnée par $[R_{ba}, t_{ba}]$ comme illustré dans la figure 2.11. Le point P_i du plan π aura alors comme projection sur le plan image de la caméra C_b le point p_i^b défini tel que :

$$p_i^b = K_b \cdot P_i^b \tag{2.22}$$

où P_i^b est la coordonnée du point P_i dans le repère de la caméra C_b.

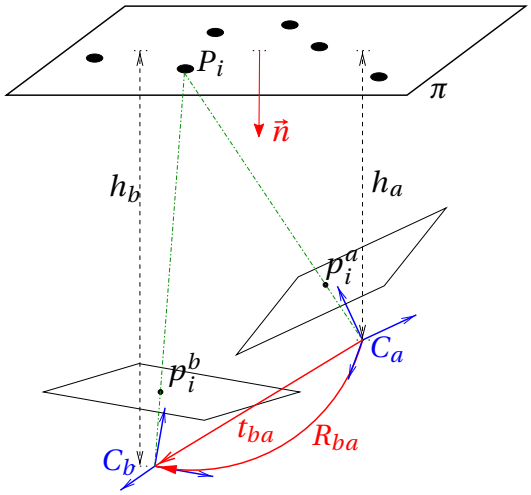


FIGURE 2.11 – Homographie

La relation liant p_i^a à p_i^b est alors donnée par la transformation linéaire suivante :

$$p_i^b = H_{ba} \cdot p_i^a \tag{2.23}$$

où H_{ba} correspond à la matrice d'homographie. Les coordonnées P_i^a et P_i^b suivant les repère des caméras C_a et C_b sont liées par la relation suivante :

$$P_i^b = R_{ba} \cdot P_i^a + t_{ba} \tag{2.24}$$

Comme P_i appartient au plan π défini par sa normale $\vec{n}$ et la distance h_a par rapport à la caméra C_a, alors :

$$P_i^b = \left(R_{ba} + \frac{t_{ba} \cdot \vec{n}^T}{h_a} \right) \cdot P_i^a \tag{2.25}$$

puis en reprenant les équations 2.21 et 2.22 :

$$p_i^b = K_b \cdot \left(R_{ba} + \frac{t_{ba} \cdot \vec{n}^T}{h_a} \right) \cdot K_a^{-1} \cdot p_i^a \tag{2.26}$$

On obtient ainsi la transformation permettant le passage entre les coordonnées de deux pixels correspondant au même point P_i sur les deux images tel que défini dans l'équation 2.23, avec :

$$H_{ba} = K_b \cdot \left(R_{ba} + \frac{t_{ba} \cdot \vec{n}^T}{h_a} \right) \cdot K_a^{-1} \tag{2.27}$$

L'homographie peut être rencontrée dans une application ADAS notamment dans un but de rendu à l'utilisateur. Ainsi l'exemple type est la construction d'une *bird eye view*, où l'on cherche à transformer l'image courante de manière à obtenir une image comme si elle avait été acquise par une caméra située en haut du véhicule et dont l'axe optique est perpendiculaire au plan de la route. Dans ce cas-là, on cherche tout simplement à appliquer une homographie H permettant cette transformation. Soit I_{in} l'image d'entrée et I_t l'image transformée, la valeur de chaque pixel à la coordonnée p_t, $I_t(p_t)$ sera donné par :

$$I_t(i, j) = I_{in}\left(H^{-1} \cdot p_t\right) \tag{2.28}$$

Remarquons que le résultat de $H^{-1} \cdot p_t$ sera très probablement des coordonnées non entières, il sera donc nécessaire de procéder à une interpolation pour obtenir la valeur de ces pixels.

Il existe d'autres applications se basant sur le principe de l'homographie. L'une d'elle est plus appliquée à la voiture autonome et la robotique : on obtient deux images d'une même caméra ayant subie un déplacement, on est capable de calculer l'homographie entre ces deux images suivant un plan connu présent dans la scène, on cherche alors à estimer le déplacement de la caméra. Des solutions permettant de décomposer une matrice d'homographie ont déjà été proposées dans la littérature, par exemple par FAUGERAS et LUSTMAN [1988] ou plus récemment par MALIS et VARGAS [2007].

2.2.3 Base de données, classificateur et apprentissage

Dans le cadre d'un application ADAS, on peut être amené à effectuer de la reconnaissance ou de la détection d'objets dans l'image. L'exemple typique est la détection de piéton pour l'AEB. Pour effectuer ce genre d'opérations, l'approche classique est de construire un descripteur, qui va transformer l'image (c'est-à-dire un ensemble de pixels) en un ensemble de caractéristiques dont le type dépend du descripteur utilisé. On utilise ensuite un classificateur qui va déterminer en fonction des caractéristiques obtenues par le descripteur la présence ou non de l'objet que l'on souhaite détecter dans l'image. Très souvent le classificateur se base sur un apprentissage pour identifier les valeurs des caractéristiques pour la reconnaissance de l'objet en question. Dans le cas de la détection de piéton, DALAL et TRIGGS [2005] proposent une approche basée sur le descripteur *Histogram of Oriented Gradient* (HOG) et le classificateur *Support Vector Machine* (SVM).

HOG

Le descripteur HOG se base sur le principe que la forme et l'apparence d'un objet dans une image peuvent être décrites par l'orientation et la valeur du gradient au niveau des contours de l'objet. En fait, l'algorithme propose de diviser l'image en un ensemble de cellules, puis pour chaque cellule de construire un histogramme de l'orientation du gradient. Ainsi, chaque pixel appartenant à la cellule vote pour une orientation, son vote étant pondéré par la norme du gradient. Les auteurs proposent plusieurs tailles de cellule, allant de 4×4 à 12×12 pixels. Pour éviter des disparités causées par des variations de l'illumination sur l'image, la dernière étape de la construction du descripteur consiste

à normaliser les histogrammes ainsi obtenus. Plusieurs cellules sont regroupées en bloc (un bloc peut aller de 1×1 à 4×4 cellules), pour chaque bloc on calcule le facteur de normalisation prenant en compte toutes les cellules du bloc. Remarquons que les blocs se chevauchent, une cellule sera donc rattachée à plusieurs blocs. Finalement, le descripteur correspond, pour l'ensemble des blocs, à l'ensemble des histogrammes normalisés.

Ainsi, pour chaque pixel de l'image, on applique les opérations suivantes :

— Calcul du gradient horizontal et vertical, par exemple avec les filtres de convolution suivants : $\begin{bmatrix}-1 & 0 & 1\end{bmatrix}$ et $\begin{bmatrix}-1 & 0 & 1\end{bmatrix}^T$. Remarquons qu'il est important de conserver le signe du gradient, les résultats doivent donc impérativement être stockés sur des entiers signées (par exemple *short*).

— Calcul de la norme du gradient, par exemple la norme L2 ou L1. Remarquons que la norme L2 est beaucoup plus coûteuse puisqu'elle fait intervenir une racine carré.

— Calcul de l'orientation du gradient : une approche naïve consisterait à appliquer un arc tangente pour déterminer la valeur exacte de l'orientation du gradient, ce qui est très coûteux en temps de calcul. Or, ce qui nous intéresse pour la suite est juste de savoir dans quel intervalle se situe l'orientation, il est donc beaucoup plus efficace de calculer cette orientation en appliquant des tests successifs.

À partir de la norme et de l'orientation du gradient pour chaque pixel, il est alors possible de construire l'histogramme pour chaque cellule de l'image. Une représentation des histogrammes de l'orientation du gradient est donnée en figure 2.12.

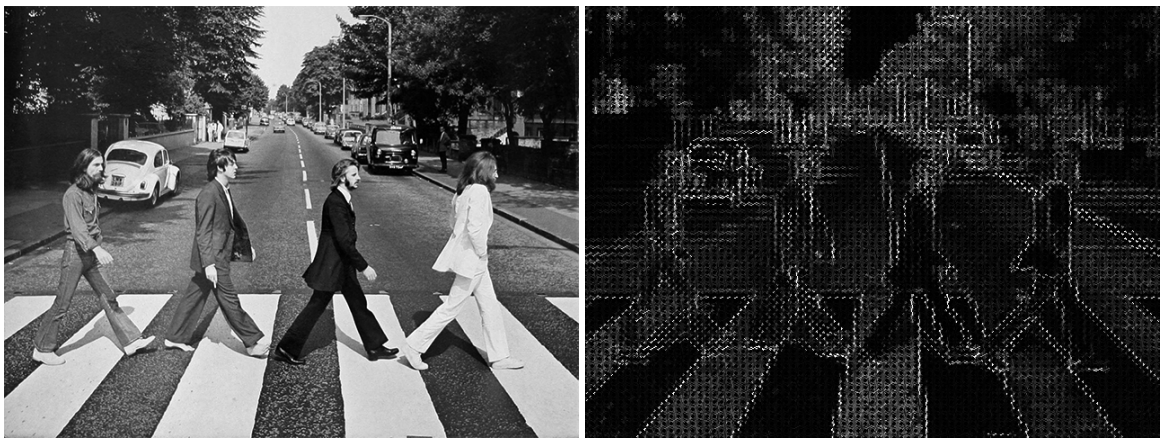


FIGURE 2.12 – Descripteur HOG : à gauche l'image d'entrée et niveaux de gris, à droite les histogrammes de l'orientation du gradient sans normalisation.

SVM

Le SVM est un classificateur permettant, à partir d'un apprentissage assisté, de déterminer l'appartenance à une classe en prenant un descripteur en entrée. Celui-ci sépare deux classes par un hyperplan déterminé lors de l'apprentissage. Soient $Y = \{-1, 1\}$ l'ensemble des deux classes et $X \in \mathbb{R}^n$ l'ensemble des vecteurs d'entrée (correspondant aux données issues du descripteur). L'idée est d'associer un vecteur d'entrée x à une des deux classes. Pour se faire, on construit un classificateur linéaire, avec $x = (x_1, \ldots, x_n)^T$ le vecteur d'entrée et un vecteur de poids $w = (w_1, \ldots, w_n)^T$, on définit :

$$h(x) = w^T x + w_0 \tag{2.29}$$

Ainsi, si $h(x) > 0$ alors l'objet ayant les caractéristiques x sera de classe 1, dans le cas contraire il sera de classe -1. La séparation entre les deux classes se fait par l'hyperplan d'équation $h(x) = 0$. Le but de l'algorithme d'apprentissage est de déterminer cet hyperplan, de manière à être capable par la suite d'associer une classe à un vecteur d'entrée non présent dans la base d'apprentissage.

Soit un ensemble d'apprentissage D :

$$D = \{(x_i, y_i) | x_i \in \mathbb{R}^n, y_i \in Y\} \tag{2.30}$$

le but de l'apprentissage est de trouver l'hyperplan optimal comme illustrer en figure 2.13. De manière mathématique, cela se traduit par la maximisation de la distance entre les échantillons d'entrée et l'hyperplan séparateur en respectant la condition d'appartenance à la classe y_i. Ainsi la distance d_i entre un point x_i et l'hyperplan est donné par :

$$d_i = \frac{y_i \cdot (w^T x_i + w_0)}{||w||} \tag{2.31}$$

La maximisation de la distance se traduit alors par :

$$\arg\max_{w, w_0} \left(\min_i (d_i) \right) \tag{2.32}$$

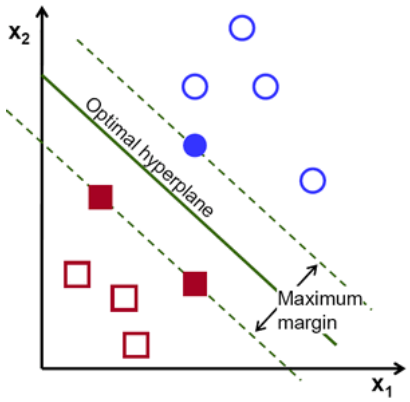


FIGURE 2.13 – Hyperplan optimal pour l'apprentissage SVM

Pour simplifier la détermination des poids w, on peut normaliser w et w_0 de telle manière que les échantillons les plus proches de l'hyperplan (x^+ pour les échantillons proches appartenant à la classe positive et x^- pour échantillons proches appartenant à la classe négative) satisfassent :

$$\begin{cases} w^T x^+ + w_0 = 1 \\ w^T x^- + w_0 = -1 \end{cases} \tag{2.33}$$

Ce qui se traduira par $y_i \cdot (w^T x_i + w_0) \geq 1$ pour tous les échantillons et $|w^T x_i + w_0| = 1$ pour les échantillons les plus proches de l'hyperplan. La distance avec l'hyperplan pour ces points sera donc de $\frac{1}{||w||}$, il faut donc maximiser $||w||^{-1}$. Le problème se ramène donc à la minimisation de $\frac{1}{2}||w||^2$ sous les contraintes $y_i \cdot (w^T x_i + w_0) \geq 1$.

Remarquons qu'une fois l'apprentissage effectué, la valeur des poids reste constante et donc déterminer la classe d'appartenance d'un objet se fait en utilisant uniquement l'équation 2.29. Nous avons présenté uniquement la forme linéaire du SVM, or il est possible que pour des ensembles échantillons-classes il n'existe pas de séparateur linéaire. Il existe cependant d'autres approchent permettant de déterminer une séparation entre deux classes [SCHÖLKOPF et SMOLA, 2002].

2.3 Architectures hétérogènes pour les ADAS

Pour répondre au besoin croissant de calcul pour des applications ADAS et/ou la voiture autonome, les fournisseurs de calculateurs proposent depuis peu des architectures hétérogènes. Par définition, un SoC hétérogène est composé de plusieurs processeurs ayant des particularités différentes. Il s'agit généralement d'une association d'un ou plusieurs processeurs génériques permettant d'exécuter des applications simples avec une grande flexibilité et d'un ou plusieurs processeurs spécialisés massivement parallèles permettant de traiter une grande quantité de données mais étant beaucoup moins flexibles et plus complexes à programmer. Plusieurs fournisseurs se sont positionnés sur le marché des SoCs hétérogènes pour les ADAS :

— Nvidia,

— Renesas,

— Texas Instrument.

Ces SoCs sont souvent composés de processeurs ARM (pour le système d'exploitation et le calcul général) et de processeurs spécifiques à chaque fondeur pour accélérer les calculs intensifs comme le traitement d'images. Nous proposons dans cette section une présentation des architectures proposées par chaque fondeur, puis une présentation globale de l'environnement logiciel associé à ces architectures.

2.3.1 Nvidia

Historiquement, la société américaine Nvidia est un fournisseur mondial de processeurs graphiques pour PC et consoles de jeu (par exemple Xbox et PlayStation 3). Cette société créée en 1993 est surtout connue dans le milieu du jeu vidéo pour ses cartes graphiques hautes performances et innovations en termes de rendu graphique. Avec la démocratisation du GPGPU, Nvidia a lancé en 2007 l'API CUDA [NVIDIA, 2015] qui permet de programmer et d'exécuter du code générique sur des GPUs et donc de pouvoir profiter de leur formidable potentiel de parallélisation. Ils sont d'ailleurs très utilisés pour l'accélération d'algorithmes de traitement d'images [Castaño-Díez et collab., 2008]. Depuis 2009, Nvidia organise chaque année une conférence, la *GPU Technology Conference* (GTC), réunissant des académiques et industriels utilisant la programmation GPGPU pour différentes applications (traitement d'images, calcul scientifique, jeu vidéo, finance, etc.). Cette conférence souligne le succès de l'API CUDA, la session de 2015 a réuni plus de 4000 participants. De plus, Nvidia équipe également des supercalculateurs, certains étant parmi les plus puissants du monde en termes de puissance de calcul [Meuer et collab., 2016].

Avec la série des processeurs Tegra, la société s'est intéressée au domaine de l'embarqué et plus particulièrement les SoCs pour les appareils nomades tels que les smartphones, tablettes, etc. Le premier à être lancé est le Tegra APX 2500, il est composé d'un ARM11 cadencé à 600 MHz et d'un processeur graphique intégré pour l'affichage, il a été annoncé en 2008. Il s'en est suivi une série de plusieurs SoCs : Tegra APX 2600, Tegra 600 et Tegra 650. La seconde génération, le Tegra 2, annoncé en 2010, embarque un processeur double cœur proposant un nouveau jeu d'instruction, l'ARMv7 et un GPU basse consommation. Le Tegra 3 propose quant à lui un processeur quadricœur, on peut noter que ce SoC a été choisi par Audi pour gérer le multimédia dans ses véhicules.

Le Tegra K1, annoncé en 2014, est le premier SoC à embarquer un GPU programmable par l'API cuda. Avec sa plateforme Jetson, intégrant un SoC Tegra K1, Nvidia se place sur

le marché des calculateurs embarqués pour les ADAS. Cette stratégie est renouvelée par la suite avec la sortie du Tegra X1 et des plateformes DrivePX, puis DrivePX 2 vendues comme étant des calculateurs pour la voiture autonome. Cette dernière est la première plateforme à embarquer un CPU conçu par Nvidia, le processeur ARM Denver. Il s'agit en fait d'un processeur 64 bits utilisant le jeu d'instruction ARMv8. Il amène également un certain nombre d'innovations et de nouveautés du fait de sa conception, comme l'optimisation dynamique de code [BOGGS et collab., 2015].

Plus récemment, l'équipementier automobile Bosch a signé un partenariat avec Nvidia pour le développement de systèmes de conduite autonome. De plus, le constructeur automobile Tesla Motors équipe aujourd'hui ses véhicules avec des puces Nvidia pour des fonctionnalités ADAS et de conduite autonome plus ou moins avancées.

Jetson Tegra K1

La plateforme automobile Jetson TK1 Pro est proposée par Nvidia comme une solution pour embarquer les applications ADAS dans le véhicule et notamment les applications utilisant des algorithmes de traitement d'images. Elle embarque un SoC Tegra K1 VCM et un ensemble d'entrées/sorties spécifiques au milieu automobile : bus CAN, interfaces CSI, etc. La version de base propose comme OS Vibrante Linux. Le SoC embarqué de cette plateforme est composé de :

— 1 processeur ARM Cortex A15 quadricœur proposant un jeu d'instruction ARMv7 ;

— 1 GPU Kepler composé d'un *Streaming Multiprocessor* (SMX) de 192 cœurs, programmable par l'API CUDA ;

— 2 *Image Signal Processors* (ISPs).

Le Cortex A15 [LANIER, 2011] dispose d'un pipeline d'exécution ayant une profondeur allant de 15 à 24 étapes, suivant la complexité des instructions à exécuter. Comme illustré en figure 2.14, ce processeur est superscalaire, c'est-à-dire qu'il peut exécuter plusieurs instructions simultanément si celles-ci utilisent des unités élémentaires différentes (soit des branches du pipeline différentes).

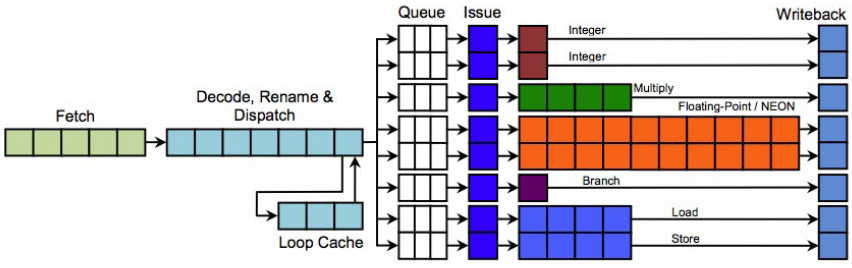


FIGURE 2.14 – Pipeline d'exécution du processeur ARM Cortex A15 : il a une profondeur allant de 15 à 24 étapes. L'A15 est superscalaire et peut exécuter jusqu'à trois instructions en *out-of-order*.

Ce processeur a également la propriété d'être *out-of-order*, c'est-à-dire qu'il peut choisir d'exécuter des instructions indépendantes dans le désordre pour profiter au mieux de sa capacité superscalaire. Enfin, il permet également une exécution spéculative pour réduire les temps d'instruction de branchement. L'A15 dispose bien évidemment d'unités pour le calcul en virgule flottante, il s'agit d'un coprocesseur, associé à chaque cœur, appelé VFP (Vector Floating Point) et qui dispose de ses propres registres. Chaque cœur propose également des extensions SIMD : NEON. En fait, NEON utilise les mêmes registres

que VFP, il s'agit de 16 registres de 128 bits pouvant être démultipliés en 32 registres de 64 bits.

Un SMX de l'architecture Kepler est, comme illustré en figure 2.15, composé de 192 cœurs de calcul CUDA simple précision, 64 unités permettant d'effectuer des calculs à double précision et 32 unités spécifiques permettant de calculer rapidement le résultat d'opérateur complexe (comme des fonctions trigonométriques, exponentielles, etc.).

FIGURE 2.15 – Un SMX de l'architecture GPU Kepler. Il dispose de 192 cœurs de calcul CUDA simple précision, 64 unités doubles précisions (DP Unit), 32 unités spécifiques (SFU).

Cette architecture dispose de 16 unités de textures qui lui permettent notamment :

— de calculer rapidement la valeur d'un pixel d'une image à une coordonnée non entière, par interpolation (par exemple la valeur du pixel $I[u+0.5][v+0.5]$) ;

— de gérer les bords de l'image et d'obtenir une valeur lorsque l'on adresse des pixels qui sont en dehors de l'image (par exemple $I[-1][-1]$).

Le GPU Kepler dispose également 64 kB de mémoire locale pouvant être utilisée comme mémoire cache L1 ou *shared memory*. Rappelons que la *shared memory* est une mémoire locale très rapide, partagée entre tous les *threads* d'un même bloc.

Les ISPs sont en fait des unités de traitement hardware permettant d'appliquer rapidement un filtre sur le flux vidéo issue de la caméra. Il s'agit de traitements bas niveau tel que la correction de la distorsion, réduction du bruit sur l'image, etc. Étant donné que les traitements sont implémentés de manière hardware, le calcul s'effectue très rapidement, cependant la flexibilité est beaucoup moins importante que sur des unités programmables

classiques. Ainsi, l'utilisateur ne peut contrôler qu'un ensemble limité de paramètres et l'ordre d'exécution des différents traitements est restreint.

Finalement, comme le montre la figure 2.16, le Tegra K1 propose d'organiser ses différents processeurs autour d'une même mémoire centralisée. L'OS réserve alors à chaque processeur une zone mémoire, de manière à éviter des accès conflictuels de différents processeurs. Remarquons que l'API CUDA propose d'allouer une zone mémoire spécifique, appelé *managed memory*, qui est à la fois accessible par le GPU et le processeur ARM. L'OS se charge alors de garantir la cohérence des données entre le cache de l'ARM et cette zone mémoire en effectuant une copie du cache vers la mémoire à chaque lancement de *kernels* sur le GPU.

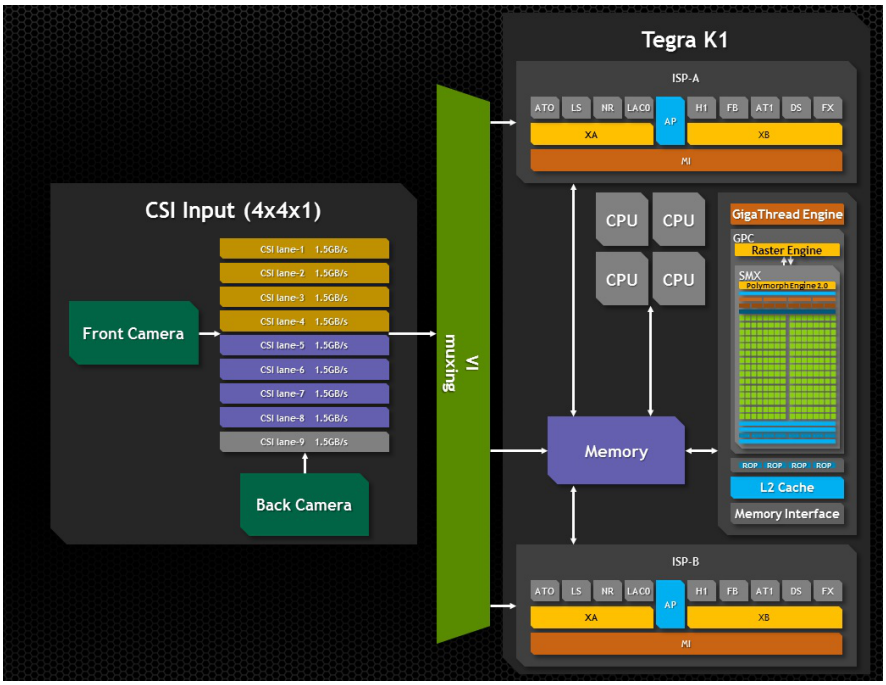


FIGURE 2.16 – Architecture du Nvidia Tegra K1

DrivePX Tegra X1

La plateforme DrivePX est vendue par Nvidia comme le calculateur de la voiture autonome. Elle est composée d'un cluster de deux Tegra X1, reliés par Ethernet, et propose 12 entrées caméras CSI, comme illustré en figure 2.17. Le SoC Tegra X1 est la seconde génération d'architecture hétérogène embarquée intégrant un GPU programmable de Nvidia. Il est composé de :

— 1 processeur ARM Cortex A57 quadricœur et 1 processeur ARM Cortex A53 utilisant la technologie big.LITTLE, tous deux 64 bits et proposant un jeu d'instruction ARMv8 ;

— 1 GPU Maxwell composé de deux SMXs de 128 cœurs, programmable par l'API CUDA ;

— 2 ISPs.

Les architectures ARMv8 [GOODACRE, 2011] proposent pour la première fois des processeurs 64 bits. Cependant son apport ne s'arrête à la taille de ses registres, ainsi un processeur ARMv8 compte 30 registres généraux, contre 15 pour l'architecture ARMv7. Même chose du côté de l'unité VFP/NEON, le nombre de registres passe de 16 à 32 registres de

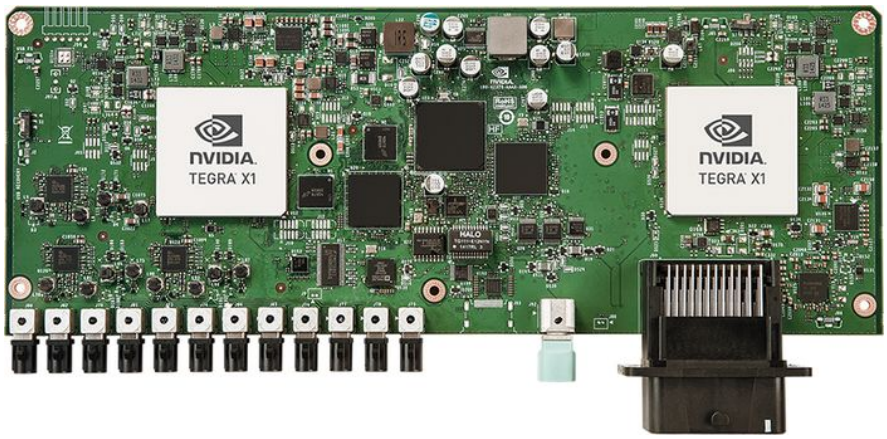


FIGURE 2.17 – DrivePX, le calculateur de la voiture autonome par Nvidia. La plateforme embarque deux Tegra X1 et propose 12 entrées caméras.

128 bits. De plus, le support de NEON devient obligatoire pour l'ensemble des processeurs ayant l'architecture ARMv8. Le processeur ARM Cortex A57 propose donc un jeu d'instruction ARMv8 et intègre un pipeline d'exécution superscalaire et *out-of-order*.

Concernant le GPU, l'architecture Maxwell est assez similaire à son prédécesseur, Kepler. En fait Nvidia a choisi de concentrer ses efforts sur la consommation électrique du GPU. Ainsi, le nombre de cœurs CUDA par SMX est passé de 192 à 128. Cette architecture propose également le support d'opérations sur des nombres en virgule flottante 16 bits, FP16, avec une capacité de traitement deux fois plus grande que celle obtenue sur des nombres à simple précision (FP32).

2.3.2 Renesas

Renesas est une société japonaise, fabricant de semi-conducteurs pour des domaines telles que les appareils mobiles (smartphones, etc.) ou l'automobile. Que ce soit pour le multimédia du véhicule, ou pour les ADAS, Renesas a sorti une série de SoCs pour répondre au besoin croissant de puissance de calcul des acteurs automobiles : les SoCs R-Car. Cette série peut se diviser en 3 générations, comme illustré en figure 2.18, chaque génération proposant différentes gammes et ciblant un besoin en particulier.

Le R-Car V2H est le premier SoC conçu pour les ADAS, notamment pour les applications de *bird eye view* et de détection de piéton. Il intègre :

— 1 processeur ARM Cortex A15 bicœur pour effectuer le calcul générique ;

— 6 IMR-LSX, qui sont des unités hardwares propres à Renesas permettant d'appliquer une correction de la distorsion et une homographie sur une image directement au moment de l'acquisition ;

— 1 unité hardware propre à Renesas conçue pour la reconnaissance de formes (par exemple la détection de piétons) : l'IMP-X4.

Les SoCs R-Car H2 et R-Car H3 constituent le haut de gamme des générations 2 et 3. Ces deux plateformes embarquent :

— des processeurs quadricœur ARM suivant la technologie big.LITTLE (4 Cortex A15 et 4 Cortex A7 pour le H2, et 4 Cortex A57 et 4 Cortex 53 pour le H3) ;

— 4 unités IMR-LSX ;

— 1 unité IMP (IMP-X4 pour le H2 et la nouvelle génération IMP-X5 pour le H3).

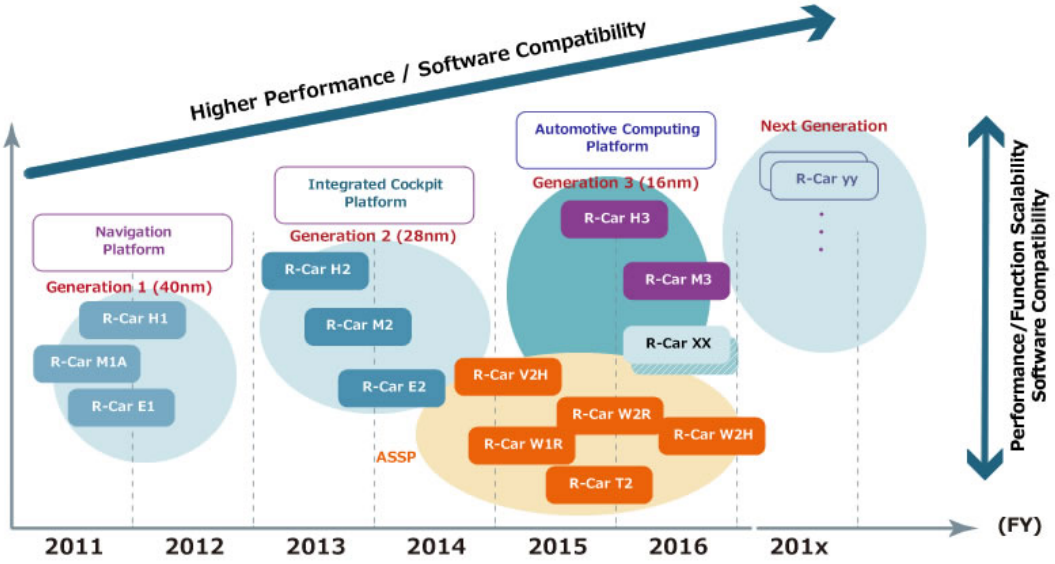


FIGURE 2.18 – Feuille de route Renesas pour les SoCs R-Car adressés au marché automobile. Chaque génération propose plusieurs gammes et cible un besoin particulier.

Une grande partie des informations concernant les unités IRM et IMP ne sont pas publiques, il nous est donc impossible de détailler avec précision leur fonctionnement dans ce manuscrit. Le R-Car H2 est conçu pour des applications de multimédia dans le véhicule, son architecture est détaillée en figure 2.19.

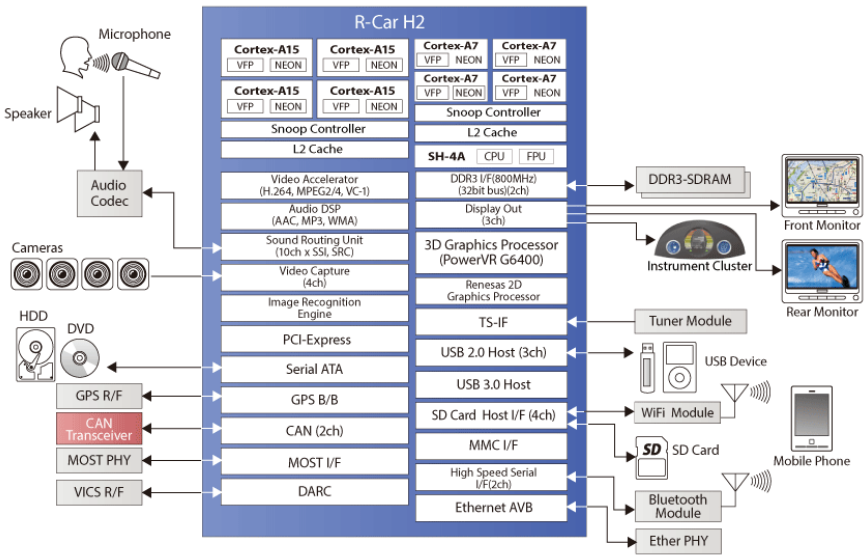


FIGURE 2.19 – Architecture du SoC R-Car H2 de Renesas.

2.3.3 Texas Instruments

TI est un fabricant de semi-conducteurs américain. Cette société tire environ 85% de ces revenues de puces analogiques (chargeurs de batteries, convertisseurs analogiques, etc.) et de processeurs embarqués. La société est très connue pour ses microcontrôleurs et ses DSPs, notamment la séries C6000. Comme Nvidia, TI se place tout d'abord sur le

marché du multimédia avec ses SoCs OMAP (*Open Multimedia Applications Platform*), qui sont des architectures hétérogènes composées de processeurs ARM et DSP. La dernière version, le OMAP 5, est sortie en 2013.

Pour l'automobile et plus particulièrement les applications ADAS, TI a sorti trois SoCs hétérogènes, correspondant à trois gammes différentes : TDA3x, TDA2Eco et TDA2x. Dans le cadre de ce manuscrit, nous nous contenterons de décrire l'architecture TDA2x, correspondant à la plus puissante des trois. Le SoC TDA2x [SANKARAN et ZORAN, 2014] est composé de quatre processeurs différents. Pour le calcul générique, il embarque un processeur ARM Cortex A15 cadencé à 750 MHz. Pour l'intégration de loi de commande automobile et la gestion des entrées/sorties, il propose 4 ARM Cortex M4. Ces processeurs ne proposent qu'un faible apport en termes de puissance de calcul. Pour répondre au besoin de traitement du signal/traitement d'images, la plateforme embarque également deux DSP C66x de TI. Il s'agit en fait d'un couple d'un DSP en virgule flottante et un autre en virgule fixe. Chacun de ces DSP peut effectuer jusqu'à 32 opérations de multiplication-accumulation par cycle. De plus, le SoC intégègre 4 *Embedded Vision Engine*s (EVEs), il s'agit en fait d'un processeur vectoriel conçu et développé par TI spécialement pour le traitement d'images. En fait, chaque EVE est composé d'un processeur RISC et d'un coprocesseur vectoriel de taille 512 bits. D'un point de vue mémoire, chaque processeur dispose d'une mémoire locale (pouvant être configurée comme mémoire cache), d'un cache L3 partagé et d'une mémoire DDR3 externe au SoC. La communication et le transfert de données entre processeur se fait par le biais d'outils propres à TI, détaillés dans la section suivante.

Finalement, les autres variantes de la série TDA sont répertoriées dans le tableau 2.1 :

TDA3x Orienté pour des applications de traitements du signal/d'images, ce SoC embarque un ISP, deux ARM M4 pour les gestions des entrées/sorties et deux DSP et un EVE pour la puissance de calcul.

TDAEco Conçu pour la construction et l'affichage de la *bird eye view*, cette plateforme embarque un GPU pour la gestion de l'affichage, un DSP pour le traitement d'images et un A15 + un M4 pour le calcul générique et la gestion des entrées/sorties.

TDA2x Il s'agit du SoC le plus puissant de TI, ses applications possibles sont le traitement du signal/d'images et la fusion de données bas niveau, travaillant à partir des données brutes issues des différents capteurs.

2.3.4 Écosystème logiciel

Pour exploiter au mieux les ressources matérielles des différents SoCs, les fournisseurs proposent un écosystème. Il s'agit en général d'un ensemble d'outils logiciels mais également un support technique pour l'aide à la prise en main. Les outils logiciels peuvent être *open source* et standardisés, c'est-à-dire maintenus par une communauté et généralement soutenus par le fondeur, ou alors propriétaire, c'est-à-dire que seul le fournisseur gère ses outils et qu'ils sont très souvent propres à un unique fondeur. Nous proposons de présenter ces différents outils dans cette section.

Systèmes d'exploitation

L'OS est la base logicielle de l'architecture de calcul, c'est lui qui nous permet d'exploiter au travers de nos programme les ressources hardwares, il est donc d'une importance cruciale. Historiquement, deux standards sont utilisés dans le monde de l'automobile : AUTOSAR pour les applications critiques et GENIVI pour le multimédia.

TABLEAU 2.1 – Architectures hétérogènes Texas Instruments pour les ADAS.

	TDA3x	**TDA2Eco**	**TDA2x**
Applications	traitement d'images *bird eye view* radars/détection *driver monitoring*	*bird eye view*	traitement d'images radars/détection fusion de données
Nombre de caméras	jusqu'à 8	jusqu'à 8	jusqu'à 10
Sorties vidéo	1	3	3
Codec H.264 matériel	non	oui	oui
GPU 3D	non	1 cœur	2 cœurs
ISP	oui	non	non
Processeurs	2 ARM Cortex M4 2 DSPs C66x 1 EVE	1 ARM Cortex A15 4 ARM Cortex M4 1 DSP C66x	2 ARM Cortex A15 4 ARM Cortex M4 2 DSPs C66x 4 EVEs

AUTOSAR (*AUTomotive Open System ARchitecture*) est un standard ouvert définissant une architecture software pour les ECUs automobiles. Il permet notamment de répondre en termes de sûreté de fonctionnement et il est compatible avec la classification *ASIL D*, c'est-à-dire qu'il est suffisamment fiable pour être utilisé dans le cadre d'applications pouvant causer des blessures à l'utilisateur en cas de mauvais fonctionnement. Cependant, AUTOSAR a été conçu principalement pour exécuter des applications assez basiques, du type loi de commande, il souffre donc de plusieurs limites :

— il ne supporte ni les périphériques vidéo ni les périphériques audio,

— il ne dispose pas de support pour la connectivité réseau (hors bus CAN),

— il ne permet pas d'exécuter des algorithmes complexes devant traiter un grand volume de données du type traitement d'images.

GENIVI, est un consortium à but non lucratif qui a défini un standard de système d'exploitation basé sur Linux pour le multimédia et la navigation dans le véhicule. Le but principal est d'encourager l'utilisation des solutions *open source* basées sur Linux par le biais d'un ensemble de spécifications. Pour aller plus loin, la fondation Linux a lancé un projet de distribution pour le multimédia et la navigation dans le véhicule : *Automotive Grade Linux* (AGL). Le projet est soutenu par un très grand nombre d'acteurs automobiles comme Toyota, Renesas, Nissan, etc.

Concernant les SoCs hétérogènes présentés dans ce manuscrit, deux stratégies différentes ont été utilisées. TI propose une solution propriétaire, StarterWare et Vision SDK. StarterWare est en fait un ensemble d'outils logiciels distribués gratuitement permettant de simplifier le portage d'applications sur des cibles embarquées sans l'utilisation de systèmes d'exploitation. Il peut être utilisé sur des cibles ARM ou DSP de TI. Renesas et Nvidia se sont appuyés sur des OS *open source* basés sur Linux. Nvidia fournit un OS appelé Vibrante Linux, qui est en fait une distribution Ubuntu avec des ajouts propriétaires pour le support des unités ISPs et des entrées caméras. Il existe également une version grand public, *Linux For Tegra* (L4T), qui n'intègre pas le support de ISPs. Renesas propose un

OS basé sur Yocto. Remarquons que les SoCs proposés par ces deux fondeurs supportent l'OS QNX. Cet OS, basé sur UNIX, est compatible POSIX (comme Linux). Il a été spécialement conçu pour les systèmes embarqués critiques et il équipe aujourd'hui environ 50 millions de véhicules pour la gestion de l'*infotainment* dans le véhicule. Remarquons que QNX fournit une solution d'OS complet pour les ADAS avec une certification *ASIL D*.

Processeur ARM

Compilateur La solution *open source* du compilateur pour les processeurs ARM est fournie par l'association à but non lucratif Linaro. Elle travaille notamment sur le portage d'outils *open source* tel que le noyau Linux et le très célèbre compilateur GCC sur des architectures ARM. Nvidia et Renesas ont choisi l'option *open source* et utilisent le compilateur GCC de Linaro. TI propose son propre compilateur, mais supporte également GCC. Remarquons qu'il existe d'autres compilateurs propriétaires, par exemple ceux fournis par la société ARM : *ARM Compiler 5* et *ARM Compiler 6*.

Multicœur Comme nous l'avons vu, la plupart des processeurs ARM est proposée avec plusieurs cœurs de calcul. Pour profiter de ces ressources, il peut être intéressant dans certains cas de séparer les données à traiter et de procéder au traitement en parallèle sur différents cœurs. Pour un OS POSIX, il est possible d'utiliser les *threads* POSIX, souvent appelés *pthreads*. L'interface de programmation permet la création, la gestion et un contrôle fin de différents *threads*. Il est ainsi possible de créer plusieurs tâches de traitement, de les affecter sur différents cœurs et de leur attacher une partie des données à traiter, de gérer la synchronisation entre ces tâches, etc. Cette API a cependant un gros défaut, elle nécessite de repenser et modifier profondément un code source initial pour un passage vers l'utilisation du multicœur. De plus elle n'est compatible qu'avec les OS POSIX. L'API OpenMP [CHANDRA, 2001; DAGUM et MENON, 1998] propose d'outrepasser ces limitations en définissant une nouvelle interface de programmation, beaucoup plus simple et intuitive. Ainsi, le passage d'un code source mono-threadé à un code multi-threadé se fait par l'utilisation de *pragmas* (c'est-à-dire des directives au compilateur) sur la partie du code que l'on souhaite paralléliser (par exemple une boucle *for*). Dans le cas d'un OS Linux, OpenMP s'appuie en fait sur *pthread* pour créer les différentes tâches concurrentes.

Vision SDK

Vision SDK [CHITNIS et collab., 2014] est un environnement logiciel propriétaire de TI. Ce *framework* permet à l'utilisateur de créer des applications ADAS pour le SoC TDA2x et permet d'adresser les différents processeurs de cette architecture. Concrètement, Vision SDK consiste en un ensemble d'outils et composants nécessaires au développement et au portage d'applications complètes, comme illustré en figure 2.20. Les outils inclus sont :

- le générateur de codes pour EVE, qui permet de transformer un code C générique en un code compilable pour ce processeur,
- BIOS qui inclut un ensemble d'interfaces logicielles pour accéder à des fonctions matérielles (par exemple l'horloge, les interruptions, etc.),
- la communication inter-processeur (IPC),
- StarterWare,
- la gestion de la couche réseau,

— les codecs audios et vidéos,

— des *kernels* correspondant à des algorithmes implémentés par le fondeur.

Le *framework* s'appuie également sur Link API qui permet de définir le pipeline d'exécution de l'application, c'est-à-dire la manière dont s'enchaînent les différents *kernels* et dont ils s'échangent les données (*buffers*).

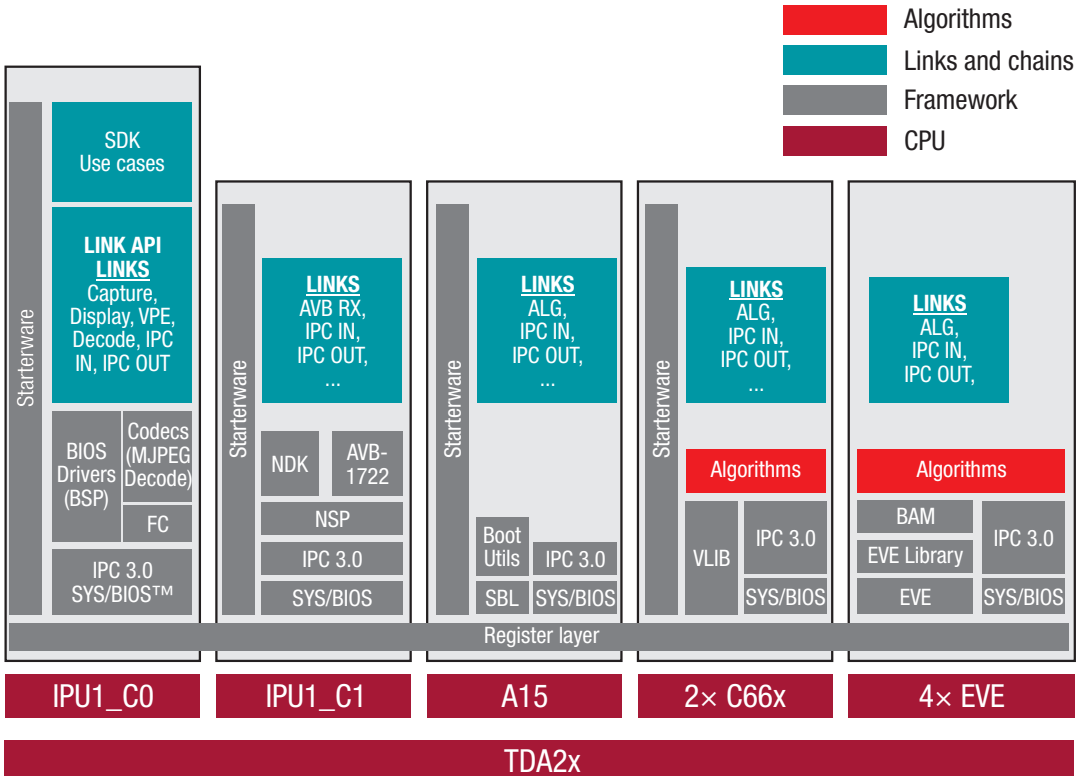


FIGURE 2.20 – Organisation du *framework* Vision SDK de TI.

OpenVX

Comme nous l'avons vu jusqu'à maintenant, les outils, les environnements de développement et parfois même les langages de programmation peuvent être propres et uniques pour chaque fondeur et SoC. Ainsi, il est très complexe d'effectuer un portage du même algorithme sur différentes cibles : un portage sur une plateforme Nvidia demandera une expertise en Linux, ARM et CUDA, alors qu'un portage sur TDA2x demandera une expertise sur les outils de TI. En fait nous avons nous même été confrontés à cette difficulté, en nous basant sur notre expérience passée de CUDA et Linux, nous avons pu très rapidement prendre en main le développement d'algorithmes sur Tegra K1, puis Tegra X1. Or, sur TDA2x il nous aurait fallu plusieurs mois à plein temps pour une prise en main complète des outils, puisque notre connaissance et notre expérience initiales étaient très limitées.

C'est pour dépasser cette difficulté et faciliter le portage d'algorithmes de traitement d'images sur des SoCs hétérogènes qu'a été conçu l'API OpenVX, implémentée en C [RAINEY et collab., 2014]. Il s'agit d'un standard ouvert et libre maintenu par le groupe Khronos. Ce groupe est un consortium industriel dont le but est de créer des APIs aux spécifications ouvertes et libres d'utilisation, il compte notamment Nvidia et TI comme membres

dirigeants. Le groupe est très célèbre pour le développement d'APIs tel que OpenGL, OpenCL, Vukan, etc. Ainsi, OpenVX propose aux développeurs d'implémenter un algorithme de traitement d'images sous la forme d'un graphe, où chaque nœud du graphe correspond à un traitement élémentaire et est représenté par un *kernel* OpenVX, comme illustré en figure 2.21. L'implémentation ainsi définie est générique et permet alors d'exécuter le graphe ainsi défini sur n'importe quelle architecture compatible avec le standard OpenVX. En fait, lorsque l'on compilera le graphe OpenVX sur un SoC donné, l'API utilisera automatiquement les *kernels* implémentés par le fondeur du SoC. De cette manière, le développeur peut profiter des accélérations matérielles du SoC sans avoir besoin de prendre en main l'environnement logiciel complet et peut porter l'algorithme sans aucune modification du graphe sur une autre architecture. L'API permet également à un utilisateur de définir et d'implémenter ses propres *kernels* pour une architecture donnée.

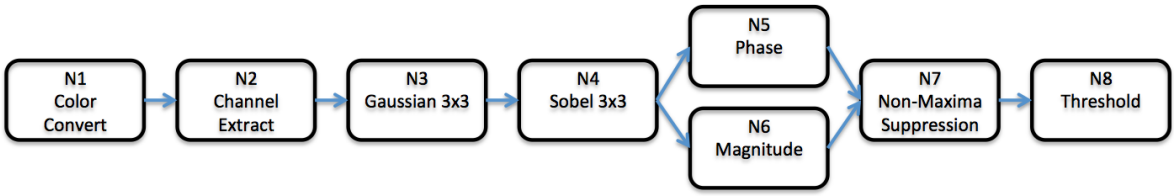


FIGURE 2.21 – Exemple de Graphe OpenVX : l'algorithme de Canny. Chaque nœud correspond à un traitement élémentaire et fait appel à un *kernel* implémenté par le fondeur du SoC lors de l'exécution.

2.3.5 Discussion

Finalement, les stratégies adoptées par les fondeurs pour répondre au besoin de puissance de calcul des acteurs automobile ont toutes en commun une solution de SoCs hétérogènes composée d'un ou plusieurs processeurs génériques et d'un ou plusieurs processeurs massivement parallèles bénéficiant d'accélérations matérielles. Si pour les processeurs génériques l'ensemble des fondeurs ont choisi une solution ARM, pour les accélérateurs matériels chaque fabricant propose une solution unique suivant son savoir-faire. Nous souhaitons mettre l'accent sur le fait que la puissance de calcul ne fait pas tout, une plateforme de développement est en général livrée avec un environnement logiciel et un ensemble d'outils permettant le portage d'applications. Si les outils fournis sont simples et intuitifs, la prise en main d'un utilisateur débutant sera rapide et efficace, il sera donc possible d'exploiter très rapidement la puissance de calcul du SoC. Dans le cas contraire, si la prise en main et la maîtrise des outils demande un coût et des efforts non négligeables avant de pouvoir obtenir des résultats convaincants, un industriel pourra être réticent à investir dans ce genre de solutions.

Nvidia illustre très bien cela, en fait la société propose depuis 2007 l'API CUDA qui permet de programmer le système hétérogène qu'est le PC : CPU et GPU. En s'appuyant sur cette expérience, ils ont su proposer, avec le Tegra K1, un environnement logiciel dans la continuité des outils pour PC, avec un système basé sur l'OS Linux et l'API CUDA. Ainsi, un développeur ayant l'expérience de CUDA sur PC pourra très facilement s'adapter et passer au développement d'applications sur les SoCs hétérogènes de Nvidia. En effet, l'architecture d'un PC est assez similaire à celle du Tegra K1 ou Tegra X1, hormis l'aspect mémoire puisque le GPU dispose d'une mémoire dédiée sur PC alors qu'elle est partagée sur les SoCs Tegra. Remarquons que c'est justement la gestion de la mémoire partagée qui

souffrait le plus de *bugs* lors des premières versions des drivers fournis par Nvidia pour le Tegra K1.

TI s'appuie sur son expérience avec les SoCs hétérogènes OMAP, composés de processeurs ARM et de DSP TI. Ces plateformes proposent un support de l'OS Linux et la gestion du DSP via des outils tel que DSP Bridge ou DSP Link, aujourd'hui considérés comme obsolètes. Par la suite, TI a sorti deux autres outils pour la communication inter-processeur SysLink, puis finalement IPC 3.x qui est aujourd'hui en développement actif et utilisé pour la TDA2x. Finalement, il y a quatre solutions différentes pour la gestion de la communication inter-processeurs, ce qui est un peu déroutant. La TDA2x amène un nouveau type de processeur, EVE et un nouvel outil : VisionSDK. Cependant, les premières versions sont assez déroutantes et peu intuitives à l'utilisation, implémenter un algorithme simple devient assez vite très complexe. En fait, contrairement à Nvidia, TI a choisi la stratégie d'une interface bas niveau pour le contrôle de son SoC, cela permet donc une configuration très fine de la part du développeur, une très grande stabilité en termes de temps de calcul (très peu de perturbations de la part de l'OS). Cependant, cela demande un temps d'apprentissage et de prise en main beaucoup plus important : plusieurs mois à plein temps, ce qui n'était pas possible dans le cadre de cette thèse.

Pour Renesas, nous avons seulement eu l'occasion de tester la distribution Linux associée (accessible librement) et non l'accélérateur IMP-X4. D'après Renesas, l'IMP-X4 supporte et peut exécuter des fonctions de la bibliothèque OpenCV, ce qui est extrêmement positif en termes de facilité de portage. Ainsi, un développeur ne connaissant pas la plateforme peut, en théorie, très simplement porter et exécuter un algorithme basé sur OpenCV. Remarquons que la bibliothèque OpenCV intègre également des fonctions implémentées en CUDA et pouvant donc être exécutées sur GPU. Cependant, vu que le processeur IMP-X4 est assez récent on est en droit de penser que la maturité de ses outils n'est pas aussi bonne que ce que peut proposer Nvidia avec ses 10 ans d'expérience sur l'API CUDA.

Globalement, Nvidia a su tirer parti de son expérience en termes de programmation GPGPU à l'aide de son API CUDA pour proposer un environnement logiciel simple et rapide à prendre en main. De notre point de vu, les SoCs proposés par Renesas et TI sont prometteurs, mais ils nécessitent un environnement logiciel plus complet pour se développer. D'ailleurs, ces trois fondeurs participent à la définition des spécifications de l'API OpenVX, mais seul Nvidia a proposé une implémentation à l'heure actuelle : l'API VisionWorks. Il s'agit en fait d'une implémentation de l'API OpenVX avec une surcouche propre à Nvidia. Conformément aux spécifications, elle permet de compiler et d'exécuter le même code source sur différentes architectures hétérogènes :

— PC Linux et PC Windows,

— Tegra K1,

— Tegra X1.

En pratique, l'ensemble des *kernels* OpenVX sont implémentés en CUDA pour être exécuté sur GPU et ne tire donc pas profit de la puissance de calcul que peut offrir le CPU. Les performances sont d'ailleurs assez décevantes par rapport à ce que l'on peut obtenir sur des algorithmes qui aurait été implémentés sans l'API.

À propos de l'API OpenVX, l'initiative est plutôt la bienvenue, elle permet de répondre à un besoin réel concernant le prototypage rapide d'algorithmes de traitement d'images sur des SoCs hétérogènes. Comme nous l'avons vu, l'API permet, en théorie, de s'affranchir d'un apprentissage de l'utilisation des outils propres à une architecture donnée. Dans la pratique le résultat est en fait fortement dépendant de ce que propose le fondeur en

termes d'implémentations pour les *kernels* OpenVX. De plus, le nombre de *kernels* actuellement proposé dans les spécifications OpenVX est assez limité (43 *kernels* sont disponible dans la version 1.1), cela restreint donc les possibilités pour le développeur. L'API laisse quand même la possibilité à l'utilisateur d'implémenter ses propres *kernels*, on peut donc espérer que ce nombre grandisse au cours du temps si une communauté suffisamment active s'implique dans le projet.

2.4 Limitations et évolutions

Pour répondre au besoin de puissance de calcul de plus en plus grand, les fabricants de semi-conducteurs s'imposent un rythme extrêmement soutenu sur l'avancée technologique des processeurs. En fait, pour augmenter les capacités de calcul de leurs processeurs, les fondeurs disposent de plusieurs leviers : le nombre de transistors embarqués dans le processeur, la fréquence d'horloge du processeur, le parallélisme qui peut se mettre en place suivant différents niveaux (comme nous l'avons présenté dans la section 1.3.2)

2.4.1 Vers le parallélisme

Ainsi, les premières décennies des micro-processeurs correspondent à une évolution exponentielle du nombre de transistors et de la fréquence d'horloge. L'un des cofondateurs de la société Intel, Gordon Earle Moore, énonça alors en 1965 que le nombre de transistors sera doublé tous les ans. En 1975, il se ravisa et énonça la très célèbre loi de MOORE et collab. [1975], qui prédit que le nombre de transistors embarqués dans un processeur double tous les deux ans. Comme illustré sur la figure 2.22, cette prédiction est, jusqu'à aujourd'hui, vérifiée. En fait, plus qu'une prédiction, cette loi est devenue un objectif et une feuille de route pour les fabricants de semi-conducteurs. En plus du nombre de transistors, la figure 2.22 illustre également l'évolution de la fréquence d'horloge des processeurs Intel au cours du temps. On peut remarquer une cassure sur l'évolution de la fréquence à partir de 2005, ce qui correspond également à un changement de stratégie et l'arrivée des processeurs multicœur chez Intel.

Pour comprendre ce choix, il faut s'intéresser à la puissance dissipée par un processeur. Il est aujourd'hui communément admis que cette puissance P est donnée par la relation suivante :

$$P = c \cdot V^2 \cdot f + P_s \tag{2.34}$$

où c est une constante liée à la capacité (en Farads), V est la tension d'alimentation, f la fréquence d'horloge et P_s correspond à la puissance consommée en veille (lorsque la fréquence est nulle). Une première analyse pourrait considérer la puissance dissipée comme proportionnelle à la fréquence, mais la fréquence d'horloge est elle-même dépendante de la tension d'alimentation du processeur. Cette relation dépend de nombreux paramètres (technologie, finesse de gravure, etc.), cependant nous admettrons l'hypothèse plutôt optimiste que la fréquence est proportionnel à la racine carré de la tension :

$$f \propto \sqrt{V} \tag{2.35}$$

En combinant les deux équations précédentes, on s'aperçoit alors que la puissance dissipée est liée au carré de la fréquence d'horloge :

$$P \propto f^2 \tag{2.36}$$

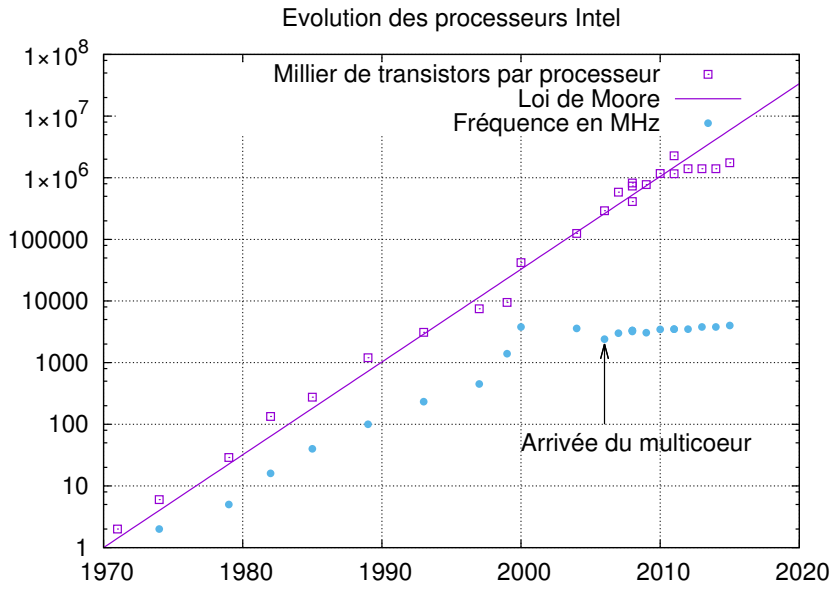


FIGURE 2.22 – Loi de Moore de 1970 à 2020, en violet la prédiction de la loi de Moore (nombre de transistors doublé tous les 2 ans) et les données réelles du nombre de transistors dans les processeurs Intel au cours du temps. Les points bleus représentent l'évolution de la fréquence d'horloge de processeurs Intel au cours du temps.

Ainsi, une augmentation exponentielle de la fréquence d'horloge implique une augmentation encore plus rapide de la puissance dissipée par les processeurs. Ils ont alors besoin d'une plus grande puissance d'alimentation et de plus d'espace pour dissiper la chaleur produite, or cela va totalement à l'encontre des besoins du marché. Notamment dans le domaine de l'embarqué ou des appareils nomades tels que les smartphones. On comprend donc mieux la stratégie d'Intel d'arrêter cette course à la fréquence d'horloge et de s'orienter vers le multicœur pour augmenter la capacité de calcul de ses processeurs.

Si la loi de Moore est encore vraie aujourd'hui, on peut se poser la question de sa pérennité. Ainsi, doubler le nombre de transistors implique que la taille de ces transistors soit également réduite, il n'est bien évidemment pas imaginable de doubler la taille des processeurs tous les deux ans. La figure 2.23 illustre l'évolution de la taille de la grille des transistors utilisée dans les processeurs Intel au cours du temps. On s'aperçoit que la finesse de gravure décroît également de manière exponentielle, jusqu'à 10 nm pour les processeurs actuels. Cependant, il existe une limite physique à la taille de la grille des transistors, en fait 10 nm correspond à une dimension inférieure à 50 atomes de silicium. La loi de Moore se heurte donc indéniablement à une limite physique causée par la miniaturisation des processeurs. Plusieurs solutions sont étudiées pour surmonter cette limite, notamment la gravure des puces en 3D.

2.4.2 Le goulot d'étranglement mémoire

Comme nous l'avons vu, que ce soit grâce au nombre de transistors, à la fréquence d'horloge ou au parallélisme, la puissance de calcul des processeurs augmente de manière exponentielle au cours du temps. Or, comme discuté précédemment, un programme nécessite à la fois des ressources de calcul et des accès à la mémoire. Les performances (en termes de temps d'exécution) sont donc liées aux capacités arithmétiques et de mémoire de l'architecture. Comme illustré dans la figure 2.24, la performance des mémoires (en

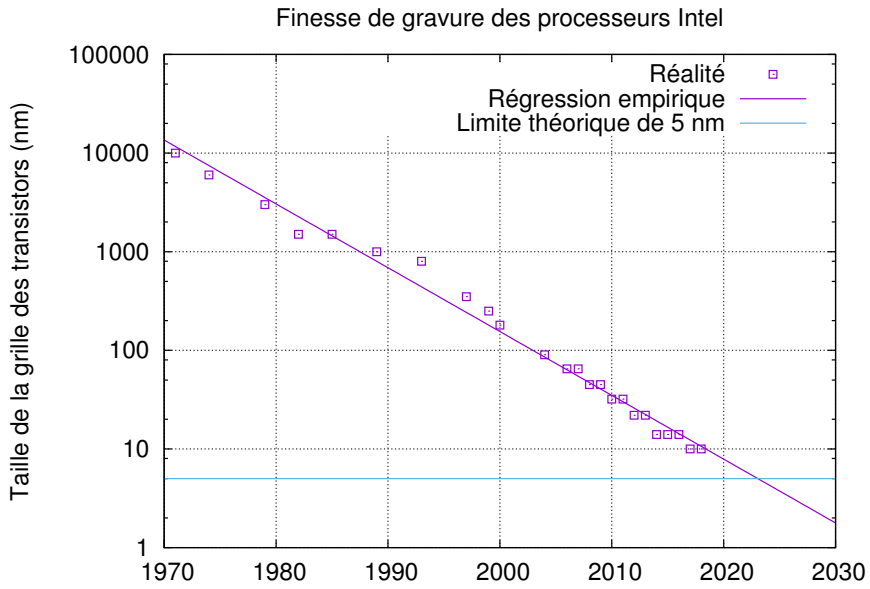


FIGURE 2.23 – Évolution de la finesse de gravure des processeurs Intel au cours du temps. En violet les données réelles et la régression empirique, en bleu une limite arbitraire de 5 nm.

termes de vitesse d'accès) évoluent de manière beaucoup plus lente que la performance des processeurs. Ce gap étant de plus en plus important au fil des ans, les accès mémoire sont devenus un goulot d'étranglement majeur.

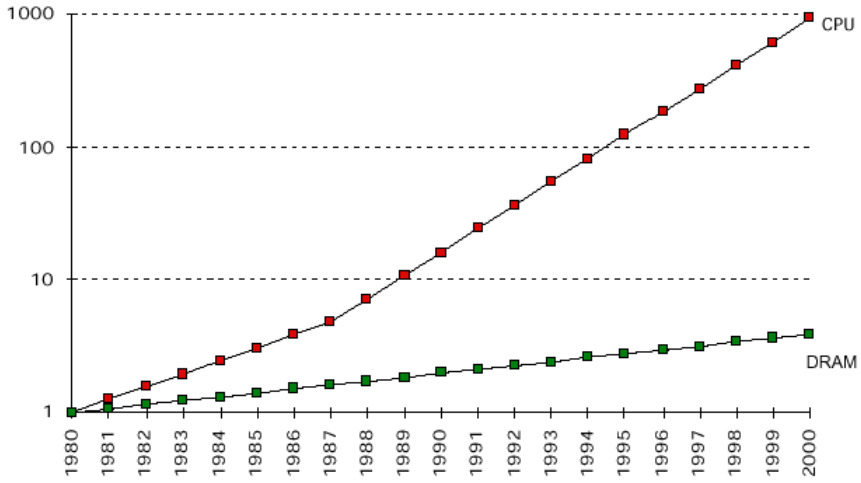


FIGURE 2.24 – Évolution du gap de performance entre mémoire et processeur au cours du temps [CARVALHO, 2002].

2.4.3 Intensité arithmétique

Comme vue dans la section 1.3, un processeur est composée de plusieurs unités appartenant à deux classes distinctes : des unités de mémoire et des unités élémentaires de calcul. De manière générale, une tâche qui s'exécute procède à des opérations de mémoire pour charger des données, des opérations arithmétiques pour traiter ces données et des opérations de sauvegarde pour écrire les résultats en mémoire. Un programme utilise donc les ressources de la mémoire et arithmétiques d'une architecture.

L'I_A est une métrique qui permet caractériser le taux d'utilisation des ressources de calcul par rapport aux besoins d'accès mémoire. Pour une tâche donnée, il s'agit tout simplement du nombre d'opérations arithmétiques divisé par le nombre d'accès mémoire :

$$I_A = \frac{Nb_{arithmétique}}{Nb_{mémoire}} \tag{2.37}$$

Ainsi une faible I_A indique que la tâche utilise fortement les ressources de la mémoire, alors qu'une forte I_A indique que la tâche utilise fortement les ressources de calcul. L'I_A a pour unité des opérations par octet (op/B).

2.4.4 Le modèle Roofline

En se basant sur le constat précédent et en utilisant la métrique de l'I_A, WILLIAMS et collab. [2009] ont défini le modèle Roofline. Ce modèle propose de représenter les performances d'une tâche, en milliard d'opérations en virgule flottante par seconde (Gflop/s), en fonction de son I_A, en nombre d'opérations en virgule flottante par octet (flop/B). Ainsi, on distingue deux cas : faible I_A et forte I_A. Dans le premier cas, lors de l'exécution la tâche effectuera beaucoup d'accès mémoire relativement au nombre d'opérations arithmétiques. Le processeur sera donc inactif une grande partie du temps, en attente que les données soient chargées depuis la mémoire. Les performances sont donc limitées par les temps d'accès mémoire, c'est-à-dire la bande passante et la latence de la mémoire. En cas de forte I_A, le processeur aura un grand nombre de calculs à effectuer pour chaque donnée chargée depuis la mémoire. Les accès mémoires s'effectuent alors que le processeur est toujours en état d'activité. Les délais dus à la latence et à la bande passante mémoire sont donc cachés par le nombre d'opérations arithmétiques à effectuer par le processeur, les performances sont alors limitées par les capacités de calcul du processeur. Un exemple de représentation Roofline est donné en figure 2.25.

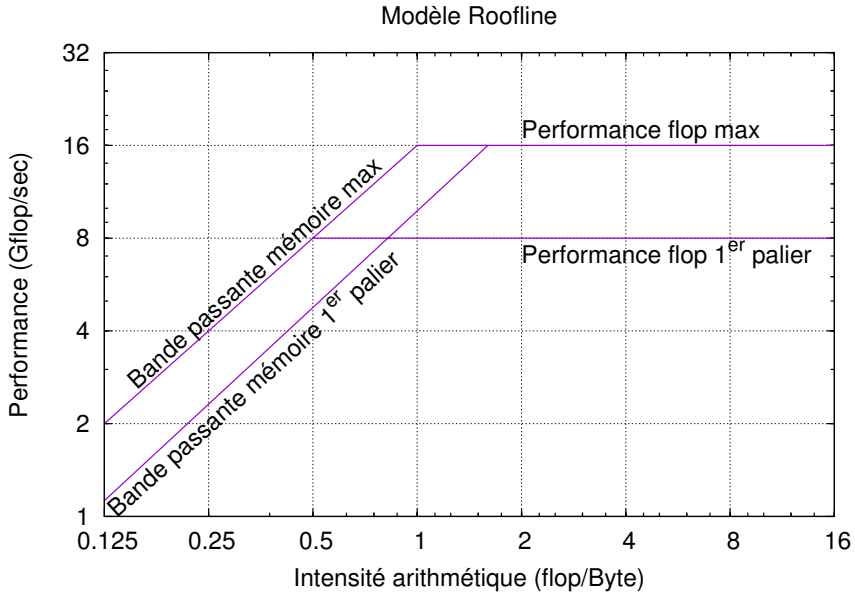


FIGURE 2.25 – Exemple de représentation du modèle Roofline donné avec un niveau de pallier. Pour la partie calcul, le premier palier peut correspondre à l'utilisation d'un seul cœur et la performance maximale pour l'utilisation de deux cœurs. Concernant la partie mémoire, il peut s'agir d'accès non-contigus pour le premier palier et d'accès contigus pour la bande passante maximale.

Dans une représentation Roofline, la partie dite de « faible I_A » est représentée par une fonction linéaire dont le coefficient directeur est égale à la bande passante de la mémoire, en milliards d'octets par seconde (GB/s), et la partie dite de « forte I_A » est représentée par une fonction constante dont la valeur est égale à la performance du processeur, en milliards d'opérations en vigule flottante par seconde (Gflop/s). En fait, le modèle est organisé par paliers ainsi lorsque l'on n'utilise qu'un seul cœur du processeur, la performance de calcul est plus faible que lorsque l'on utilise plusieurs cœurs. Il en va de même avec l'utilisation ou non d'accélérations spécifiques liées au processeur (par exemple l'utilisation d'instructions SIMD). De la même manière il y a une apparition de plusieurs paliers liés à l'accès mémoire, par exemple une bande passante plus faible en cas d'accès non-contigus. La valeur de la bande passante mémoire peut être obtenue par benchmark, par exemple STREAM [McCalpin, 1995].

Ce modèle est utilisé pour l'identification des goulots d'étranglement et comme une aide à l'optimisation. Ainsi dans le cas d'une tâche ayant une faible I_A pour lequel la performance est en dessous de la bande passante mémoire maximale, le modèle nous suggère d'optimiser les accès mémoire (en utilisant par exemple du *prefetching* ou en réorganisant les accès de manière à ce que ceux-ci soient contigus) plutôt que d'accélérer les calculs. Si la bande passante mémoire maximale est atteinte, le seul moyen d'améliorer la performance est alors d'augmenter l'I_A de la tâche en diminuant le nombre d'accès à la mémoire centrale (par exemple en utilisant une mémoire locale, plus rapide). Dans le cas d'une forte I_A, le modèle nous suggère d'optimiser l'aspect calcul en utilisant par exemple un cœur de calcul supplémentaire, des accélérations matérielles spécifiques (par exemple des instructions SIMD), etc.

2.4.5 Le modèle Boat-Hull

Une amélioration du modèle Roofline a été proposée par Nugteren et Corporaal [2012], il s'agit du modèle Boat-Hull. Contrairement à la représentation Roofline, ce modèle propose de représenter le temps d'exécution d'une tâche en fonction de sa complexité. La complexité correspond au nombre d'opérations effectuées par données d'entrée (quelle que soit la taille des données), par opposition à l'I_A qui mesure le nombre d'opérations effectuées par octet. Le but de cette représentation n'est pas de proposer un outil d'analyse de performances mais une méthode de prédiction de performances. Ainsi, en connaissant la complexité d'une tâche et la représentation Boat-Hull d'un processeur, on est alors capable de prédire facilement et rapidement un temps d'exécution de la tâche sur le processeur en question.

Le modèle distingue également la limite de performance liée à la bande passante mémoire dans le cas de faible I_A d'une part et la limite de performance liée à la capacité de calcul dans le cas de forte I_A d'autre part. Ainsi, on cherche à prédire le temps d'exécution d'une tâche appliquant un opérateur f de complexité f_{comp} à l'ensemble des données d'entrée. Le modèle définit deux fonctions :

— $c_0(f_{comp})$: la fonction de temps de calcul,

— $m_0(f_{comp})$: la fonction de délai d'accès mémoire.

Bien évidemment ces deux fonctions dépendent de nombreux paramètres :

— w : le degré de parallélisme, il s'agit en fait du nombre de données d'entrée pouvant théoriquement être traitées en parallèle. Dans le cas d'algorithmes appartenant à la classe de parallélisme simple ce nombre est égal à la quantité de données à traiter.

— α : un coefficient correspondant au nombre d'appels à l'opérateur f effectués par *work-unit*, par exemple dans le cas de l'utilisation du voisinage dans le calcul.

— o : l'offset, correspondant par exemple à la latence de lancement d'un *kernel* sur GPU.

— d : la taille de données à lire/écrire, cela comprend les données d'entrées et de sorties.

— γ : la taille des accès mémoire contigus.

— u : la taille des accès mémoire non-contigus.

Les équations sont alors définies comme :

$$c_0 = \frac{w \cdot (f_{comp} \cdot \alpha + o/w)}{P_{compute}} \tag{2.38}$$

$$m_0 = \frac{\gamma}{P_{contiguë}} + \frac{u}{P_{non-contiguë}} \tag{2.39}$$

où $P_{compute}$, $P_{contiguë}$ et $P_{non-contiguë}$ représentent respectivement la capacité de calcul maximale (en Gflop/s), la bande passante pour des accès contigus et des accès non contigus (en GB/s). Ces paramètres sont constants pour chaque architecture.

Ce modèle propose également une approche par paliers. Ainsi, les fonctions c_0 et m_0 précédemment définies correspondent aux performances maximums atteignables. Par la suite, les fonctions c_1 et m_1 représentent un pallier de performances plus faible, c'est-à-dire un temps d'exécution plus élevé. La fonction c_1 se calcule en appliquant un ratio sur c_0

$$c_1 = N \cdot c_0 \tag{2.40}$$

où N est un paramètre lié à l'architecture, par exemple $N = 2$ pour le GPU. Pour le CPU il est possible de définir plus de deux paliers, par exemple l'utilisation ou non d'opérateurs SIMD (dans ce cas là N est égal à la taille des registres SIMD), ou l'utilisation ou non du multicœur (N est alors égal au nombre de cœurs). La fonction m_1 se calcule en supposant que tous les accès mémoire se font de manière non-contiguë :

$$m_1 = \frac{d}{P_{non-contiguë}} \tag{2.41}$$

Les auteurs proposent une classification des types de traitement avec la valeur des paramètres associés. Par exemple dans le cas d'une tâche effectuant des opérations éléments par éléments, indépendamment du voisinage, sur une image de taille $A \times B$, les auteurs suggèrent d'utiliser les paramètres suivants :

$$\begin{cases} w = & A \cdot B \\ \alpha = & 1 \\ o/w = & 16 \\ d = & 2 \cdot A \cdot B \\ \gamma = & d \\ u = & 0 \end{cases} \tag{2.42}$$

Ainsi, en connaissant la classe de la tâche, donc les paramètres associés, et les caractéristiques de l'architecture, la méthodologie est capable de prédire différents paliers de temps de calcul en fonction de la complexité de l'opérateur utilisé f_{comp} pour chaque donnée d'entrée. Un exemple de représentation Boat-Hull est donné en figure 2.26.

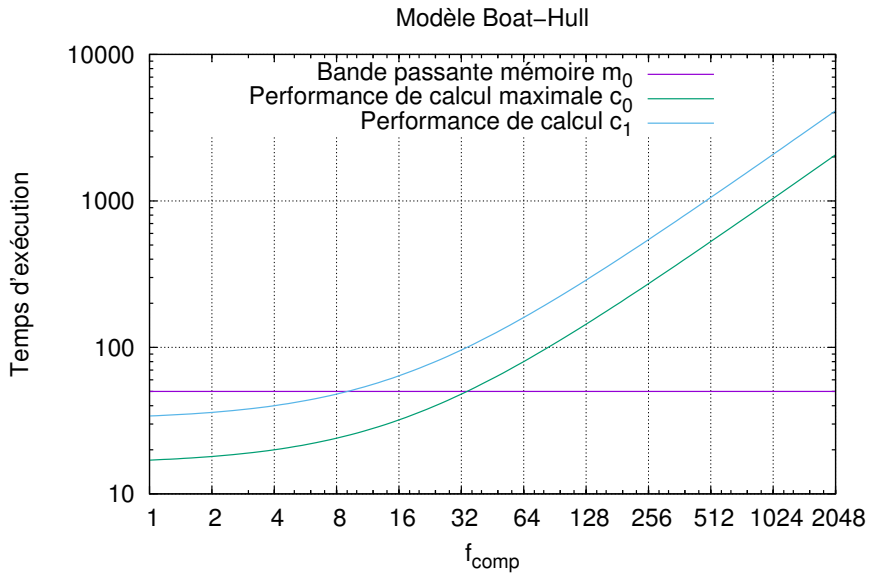


FIGURE 2.26 – Exemple de représentation en modèle Boat-Hull.

2.5 Références

BOGGS, D., G. BROWN, N. TUCK et K. VENKATRAMAN. 2015, «Denver : Nvidia's first 64-bit ARM processor», *IEEE Micro*, vol. 35, nº 2, p. 46–55. 45

CANNY, J. 1986, «A computational approach to edge detection», *IEEE Transactions on Pattern Analysis and Machine Intelligence*, , nº 6, p. 679–698. 35

CARVALHO, C. 2002, «The gap between processor and memory speeds», dans *Proceedings of IEEE International Conference on Control and Automation*. 58

CASTAÑO-DÍEZ, D., D. MOSER, A. SCHOENEGGER, S. PRUGGNALLER et A. S. FRANGAKIS. 2008, «Performance evaluation of image processing algorithms on the GPU», *Journal of Structural Biology*, vol. 164, nº 1, p. 153–160. 44

CHANDRA, R. 2001, *Parallel programming in OpenMP*, Morgan Kaufmann. 52

CHITNIS, K., R. STASZEWSKI et G. AGARWAL. 2014, «Ti vision sdk, optimized vision libraries for adas systems», *White Paper : SPRY260, Texas Instrument Inc.* 52

DAGUM, L. et R. MENON. 1998, «Openmp : an industry standard api for shared-memory programming», *IEEE computational science and engineering*, vol. 5, nº 1, p. 46–55. 52

DALAL, N. et B. TRIGGS. 2005, «Histograms of oriented gradients for human detection», dans *IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2005. CVPR 2005.*, vol. 1, IEEE, p. 886–893. 41

FAUGERAS, O. D. et F. LUSTMAN. 1988, «Motion and structure from motion in a piecewise planar environment», *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 2, nº 03, p. 485–508. 41

GOODACRE, J. 2011, «Technology preview : The armv8 architecture», *White Paper, Nov.* 47

HARTLEY, R. et A. ZISSERMAN. 2000, *Multiple view geometry in computer vision*, vol. 2, Cambridge Univ Press. 28

HUGHES, C., M. GLAVIN, E. JONES et P. DENNY. 2009, «Wide-angle camera technology for automotive applications : a review», *IET Intelligent Transport Systems*, vol. 3, nº 1, p. 19–31. 30

LANIER, T. 2011, «Exploring the design of the cortex-a15 processor», Web ressource. http://www.arm.com/files/pdf/AT-Exploring_the_Design_of_the_Cortex-A15.pdf. 45

MALIS, E. et M. VARGAS. 2007, *Deeper understanding of the homography decomposition for vision-based control*, thèse de doctorat, INRIA. 41

MCCALPIN, J. D. 1995, «Memory bandwidth and machine balance in current high performance computers», *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, p. 19–25. 60

MEUER, H., E. STROHMAIER, J. DONGARRA, H. SIMON et M. MEUER. 2016, «Top500 supercomputer site», Web ressource. http://www.top500.org. 44

MOORE, G. E. et collab.. 1975, «Progress in digital integrated electronics», dans *Electron Devices Meeting*, vol. 21, p. 11–13. 56

MUNSON, D. C. 1996, «The Lenna Story», Web ressource. http://www.lenna.org. 29

NUGTEREN, C. et H. CORPORAAL. 2012, «The boat hull model : enabling performance prediction for parallel computing prior to code development», dans *Proceedings of the 9th conference on Computing Frontiers*, ACM, p. 203–212. 60

NVIDIA. 2015, «Cuda C Programming Guide», . 44

RAINEY, E., J. VILLARREAL, G. DEDEOGLU, K. PULLI, T. LEPLEY et F. BRILL. 2014, «Addressing system-level optimization with openvx graphs», dans *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, p. 644–649. 53

ROMERO, L. et C. GOMEZ. 2007, *Correcting radial distortion of cameras with wide angle lens using point correspondences*, INTECH Open Access Publisher. 33

RUFLI, M., D. SCARAMUZZA et R. SIEGWART. 2008, «Automatic detection of checkerboards on blurred and distorted images», dans *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, p. 3121–3126. 31

SANKARAN, J. et N. ZORAN. 2014, «TDA2X, a SoC optimized for advanced driver assistance systems», dans *2014 IEEE Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, p. 2204–2208. 50

SCHÖLKOPF, B. et A. J. SMOLA. 2002, *Learning with kernels : support vector machines, regularization, optimization, and beyond*, MIT press. 43

SZELISKI, R. 2011, *Computer vision : algorithms and applications*, Springer. 28

URBACH, E. R. et M. H. WILKINSON. 2008, «Efficient 2-d grayscale morphological transformations with arbitrary flat structuring elements», *IEEE Transactions on Image Processing*, vol. 17, nº 1, p. 1–8. 38

VINCENT, L. 1993, «Morphological grayscale reconstruction in image analysis : Applications and efficient algorithms», *IEEE Transactions on Image Processing*, vol. 2, nº 2, p. 176–201. 38

WILLIAMS, S., A. WATERMAN et D. PATTERSON. 2009, «Roofline : an insightful visual performance model for multicore architectures», *Communications of the ACM*, vol. 52, nº 4, p. 65–76. 59

ZHANG, Z. 2000, «A flexible new technique for camera calibration», *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, nº 11, p. 1330–1334. 31

Chapitre 3

Méthodologie d'analyse d'embarquabilité

« *Young man, in mathematics you don't understand things. You just get used to them.* »

John von Neumann

Sommaire

3.1 Introduction

Nous définissons l'analyse de l'embarquabilité d'un algorithme comme étant le processus permettant de répondre à la question suivante : « Est-ce que mon algorithme peut s'exécuter sur ce SoC tout en respectant les contraintes temps-réel ? ». Aujourd'hui la problématique d'embarquabilité des algorithmes représente un enjeu de taille pour les acteurs de l'industrie automobile. En fait, jusqu'à aujourd'hui, la validation d'un ECU s'effectuait par le portage à la main de l'algorithme. Il s'agit en général de µc mono-cœur, où le portage n'est pas si complexe. Cependant, la plupart du temps les systèmes choisis sont sur-dimensionnés et on peut se poser la question de la pérennité de cette approche avec l'utilisation de processeurs de plus en plus complexes notamment les SoCs hétérogènes. Ainsi, valider le portage d'un algorithme sur plusieurs types d'architectures impose d'avoir une expertise sur ces architectures, de connaître les différents environnements logiciels, etc. Les architectures hétérogènes amènent également une autre problématique : la répartition des charges de calcul entre les différents processeurs d'un même SoC, appelée également la problématique de *mapping*. De plus, les acteurs automobiles souhaiteraient prévenir l'utilisation de systèmes surdimensionnés pour limiter les coûts de tels systèmes.

3.1.1 Modélisation du problème

Dans le cadre de cette thèse, nous avons choisi de nous focaliser sur des architectures hétérogènes. Ainsi, celles-ci présentent des capacités de calcul beaucoup plus intéressantes dans un contexte de systèmes embarqués mais elles sont beaucoup plus complexes à utiliser que des processeurs embarqués classiques à cause de cette problématique de *mapping*. Cette difficulté a été identifiée très tôt, nous avons donc dès le début de cette thèse cherché et proposé une modélisation du problème pour identifier de possibles solutions, comme discuté dans [SAUSSARD et collab., 2015a,b,c]. Nous décrivons ici la première approche telle qu'elle a été définie au début de cette thèse. Cependant cette approche initiale souffre d'un certain nombre de lacunes, c'est pour cela que nous décrirons comment raffiner cette modélisation au cours de ce chapitre.

Ainsi, nous cherchons donc à embarquer un algorithme sur une architecture embarquée composée de m processeurs. Tout d'abord l'algorithme en question peut être séparé en un ensemble de n blocs, chaque bloc correspond à une partie des traitements à appliquer à l'image d'entrée. Remarquons que dans cette représentation, l'algorithme s'exécute en un temps fini et correspond à l'ensemble des traitements qui sont appliqués à une image d'entrée. Les blocs ainsi décrits ont des dépendances entres eux bien définies, déterminées par la dépendance de données. Pour la suite, nous choisissons de représenter l'ensemble des processeurs disponibles par le vecteur $\mathbf{p}$ de taille m et l'ensemble des blocs de l'algorithme par le vecteur $\mathbf{k}$ de taille n :

$$\mathbf{p} = \begin{pmatrix} p_1 & p_2 & \dots & p_m \end{pmatrix}^T \qquad \mathbf{k} = \begin{pmatrix} k_1 & k_2 & \dots & k_n \end{pmatrix}^T \tag{3.1}$$

Le *mapping* des blocs $\mathbf{k}$ de l'algorithme sur les processeurs $\mathbf{p}$ du SoC hétérogène peut être représenté par une simple multiplication de matrice :

$$\mathbf{p} = \mathbf{M} \cdot \mathbf{k} \tag{3.2}$$

où $\mathbf{M}$ est ce que l'on appelle la matrice de *mapping*, de taille $m \times n$. Cette matrice est binaire, c'est-à-dire que ses coefficients ne peuvent valoir que 0 ou 1, où $\mathbf{M}_{i,j} = 1$ implique

que le bloc k_j est exécuté par (ou mappé sur) le processeur p_i. La dépendance des données peut imposer que les différents blocs s'exécutent dans un ordre spécifique. De manière évidente, un bloc k_i ne peut pas s'exécuter avant le bloc k_j s'il a besoin des résultats de celui-ci. Ainsi, nous définissons la matrice de dépendance φ, binaire, de taille $n \times n$ et définie telle que $\varphi_{i,j} = 1$ implique que le bloc k_i dépend du bloc k_j et donc que k_j doit s'exécuter avant k_i. Ensuite, nous définissons les fonctions suivantes :

— $\tau(\mathbf{M})$ la fonction d'exécution retournant un vecteur de taille $n \times 1$ correspondant au temps d'exécution de chaque bloc en fonction de $\mathbf{M}$;

— $\delta(\mathbf{M})$ la fonction de transfert retournant un vecteur de taille $n \times 1$ correspondant au délai de transfert mémoire d'un processeur à un autre nécessaire avant l'exécution de chaque bloc en fonction de $\mathbf{M}$;

— $\eta(\mathbf{M})$ la fonction d'occupation retournant un vecteur de taille $m \times 1$ correspondant au taux d'occupation de chacun des processeurs en fonction de $\mathbf{M}$;

— $f\left(\mathbf{p}, \mathbf{k}, \varphi, \tau(\mathbf{M}), \delta(\mathbf{M}), \eta(\mathbf{M})\right)$ la fonction de coût retournant le temps d'exécution global de l'algorithme sur le SoC hétérogène en fonction de $\mathbf{M}$.

Nous cherchons alors le *mapping* $\mathbf{M_0}$ qui minimise le temps d'exécution global, il s'agit donc de trouver la matrice $\mathbf{M}$ qui minimise la fonction de coût f :

$$\mathbf{M_0} = \arg\min_{\mathbf{M}} f\left(\mathbf{p}, \mathbf{k}, \varphi, \tau(\mathbf{M}), \delta(\mathbf{M}), \eta(\mathbf{M})\right) \tag{3.3}$$

Avec cette modélisation, si l'on connaît les différentes variables de f pour différentes valeurs de $\mathbf{M}$, on est alors capable de trouver le *mapping* minimisant la fonction de coût. De plus l'exploration des différents *mappings* peut se faire de manière automatique et rapide par opposition à approche manuelle qui nécessite un portage de l'ensemble des blocs d'algorithme et une mesure de leurs temps d'exécution sur l'ensemble des processeurs. En pratique, exprimer un temps d'exécution en fonction de ces différents paramètres est loin d'être évident. Ainsi, comment gérer les cas où plusieurs blocs partagent une même ressource au moment de leurs exécutions ? Normalement le temps d'exécution de chaque bloc doit en être impacté, or ce n'est pas le cas avec cette modélisation. De plus cette approche ne permet de pas d'intégrer un certain nombre de points cruciaux, par exemple si la fonction $\eta(\mathbf{M})$ permet de représenter la contrainte temps-réel de rythme (c'est-à-dire la fréquence d'arrivée des données), la contrainte de latence n'entre pas du tout en compte. La matrice de mapping φ nous permet de représenter des dépendances entre plusieurs blocs, mais elle ne permet pas de représenter les dépendances temporelles, c'est-à-dire une dépendance par rapport à un résultat antérieur (par exemple dans le cas d'un algorithme de tracking, un bloc peut avoir besoin de la position de la cible à l'instant précédent).

On s'aperçoit donc assez vite de la limite de cette approche initiale. Ainsi, nous avons décidé de revoir notre stratégie et d'explorer une modélisation basée sur une représentation en graphe (duale de la représentation matricielle) qui permet de modéliser l'ensemble des contraintes temps-réel, les dépendances temporelles et l'impact des ressources partagées.

3.1.2 Vers la prédiction de performances

Pour répondre à la problématique d'embarquabilité, nous avons besoin de définir une méthodologie permettant de déterminer rapidement si un algorithme donné peut être exécuté en temps-réel ou non sur un ensemble d'architectures. De cette manière il est

possible de choisir sereinement et facilement un ECU correctement dimensionné et permettant de répondre au cahier des charges de l'application. L'approche qui nous paraît la plus pertinente est d'utiliser la prédiction de performances. Ainsi, si l'on est capable de prédire la performance qu'aura un algorithme sur une architecture quelconque de manière plus ou moins précise, alors la réponse sur le respect ou non des contraintes temps-réel est immédiate et donc le caractère embarquable de l'algorithme sur l'architecture en question devient trivial. La méthodologie d'analyse de l'embarquabilité telle que définie dans cette thèse a été conçue pour répondre à ce besoin industriel.

3.2 Algorithmes et contraintes temps-réel

Pour parler d'embarquabilité d'algorithme, il est tout d'abord nécessaire d'affiner sa composition et donc les besoins en ressources qui en découlent. Ainsi, comme illustrée dans la figure 3.1, une application ADAS typique peut être décomposée en différentes parties :

Bloc perception Il s'agit de traiter les données brutes issues des différents capteurs (caméra et traitement d'image par exemple). De notre point de vue, puisque c'est le bloc qui traitent le plus grand nombre de données (l'ensemble des pixels dans le cas du traitement d'images), il s'agit de la partie la plus exigeante en termes de complexité de calcul et de besoins mémoire.

Bloc fusion de données Ce bloc prend en entrée un ensemble de cibles potentielles issues des différents capteurs. Il fusionne ces données pour prendre une décision sur la compréhension de l'environnement du véhicule, c'est-à-dire la localisation et le types des cibles présentes autour du véhicule.

Bloc loi de commande À l'aide des informations issues de la fusion de données, la loi de commande (Ldc) agit pour avertir le conducteur (par exemple une alerte sonore) ou agit directement sur le véhicule (freinage d'urgence, contrôle automatique de la direction, etc.). Une loi de commande représente une charge de calcul minime, elle est en général embarquée sur des µc de faible puissance mais garantissant une très grande sûreté de fonctionnement et une faible latence.

Ainsi, pour une application ADAS typique à base d'une ou plusieurs caméras, nous estimons que les algorithmes de traitement d'images représentent environ 90% de la charge de calcul et de l'impact mémoire, c'est d'ailleurs pour cela qu'ils sont si difficiles à embarquer et à exécuter en temps-réel. Dans le cadre de ce chapitre et de la thèse en général, nous avons donc choisi de nous focaliser sur les algorithmes de traitements d'images. Remarquons tout de même que notre méthodologie est tout à fait applicable sur n'importe quel bloc. Nous avons d'ailleurs, au cours de cette thèse, présenté plusieurs preuves de concept autour de l'embarquabilité de lois de commande.

3.2.1 Modélisation d'algorithmes

Nous choisissons donc de nous focaliser sur les algorithmes de traitement d'images, reste maintenant à définir une modélisation fine des algorithmes. Ainsi, pour estimer les ressources nécessaires, il est essentiel de définir une représentation qui puisse par la suite être utilisée en entrée de notre méthodologie d'analyse de l'embarquabilité. Nous avons donc besoin d'une représentation mettant en œuvre un découpage de l'algorithme en

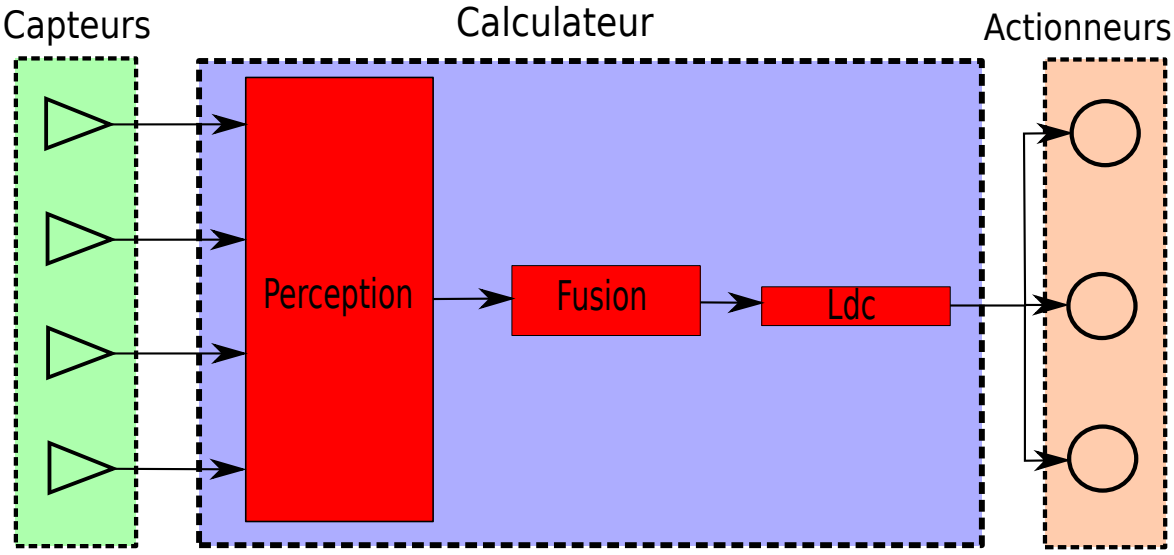


FIGURE 3.1 – Schéma d'une application ADAS typique. Les capteurs retournent des informations brutes au calculateur. Le calculateur procède dans un premier temps au traitement particulier des données issues de chaque capteur. Ensuite, il fusionne les données issues de la perception pour obtenir une information sur l'environnement. Enfin, il applique la loi de commande, qui agissent sur les actionneurs du véhicule en fonction de la présence ou non d'obstacles.

un ensemble de plusieurs blocs avec des dépendances bien définies. Il est également essentiel que la modélisation puisse mettre en œuvre les différentes contraintes temps-réel définies par le cahier des charges de l'application, pour y apporter ensuite des réponses.

Dans la littérature, il existe plusieurs modèles pour représenter un algorithme, le plus souvent sous la forme d'un *dataflow*, particulièrement bien adapté à la représentation des algorithmes de traitement du signal. Les réseaux de KAHN [1974], connus également sous le nom de *Kahn Process Networks* (KPNs), décomposent l'algorithme en un ensemble de processus indépendants et un ensemble de canaux de communications entre ces processus. Un exemple de KPN est donné en figure 3.2, on représente en général un processus par un cercle et les canaux de communications par des flèches directionnelles. Le modèle considère la communication entre processus par le biais de files de messages de tailles non bornées. Ainsi, l'écriture par un processus dans une file est considérée comme non bloquante, contrairement à la lecture sur une file vide qui est bloquante.

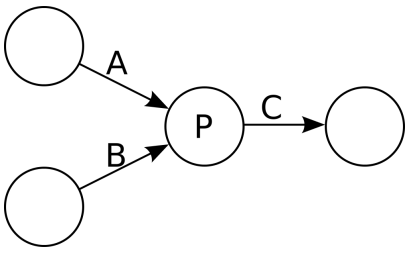


FIGURE 3.2 – Exemple de réseau de Kahn : les cercles représentent les processus et les flèches A, B et C les canaux de communications inter-processus.

La représentation KPN est basée sur le modèle d'exécution suivant : chaque tâche s'exécute de manière indépendante et simultanée et est inactive tant que le flux d'entrée est vide (la lecture sur une file vide est bloquante). Or si plusieurs tâches s'exécutent de manière simultanée sur un même processeur, cela va forcément avoir un impact sur leurs temps d'exécutions respectifs, ne vaut-il pas mieux exécuter les tâches les unes après les

autres ? De plus, dans la réalité, chaque file de communication correspond à un *buffer* circulaire de taille fixe stockant les données en attente de lecture. Si la tâche consommatrice (en aval) est particulièrement lente, on peut assister à un remplissage complet du *buffer* (*buffer overflow*), ce qui implique des pertes de données dans le processus de traitement et donc potentiellement des erreurs dans l'algorithme.

La modélisation par *Synchronous DataFlow*s (SDFs) a été proposée initialement par LEE et MESSERSCHMITT [1987]. Il s'agit en fait d'une représentation en graphe où chaque nœud correspond à une opération, également appelé acteur, et où chaque arc correspond à un *buffer* FIFO. À son exécution, chaque acteur consomme et produit des données, que l'on appelle jetons. Notons que les jetons peuvent représenter n'importe quel type de données, comme des entiers, des réels ou même des structures (matrices ou images par exemple). Les SDFs suivent les règles suivantes :

— Chaque acteur a un nombre bien défini de jetons à consommer et à produire à son exécution.

— Un acteur peut s'exécuter si le nombre de jetons disponibles dans son *buffer* d'entrée est supérieur ou égal au nombre de jetons consommés par cet acteur et l'espace disponible dans son *buffer* de sortie est supérieur ou égale au nombre de jetons qu'il produit (dans le cas contraire on assiste à des pertes de données, ce qui n'est prévu par cette modélisation).

Un exemple de SDF est donné en figure 3.3.

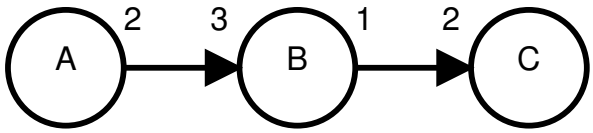


FIGURE 3.3 – Exemple de SDF : l'acteur A produit chaque fois 2 jetons, l'acteur B consomme chaque fois 3 jetons et en produit 1 et l'acteur C en consomme 2 à chaque exécution.

Le modèle *Computer Vision Synchronous DataFlow* (CV-SDF), proposé par STICHLING et KLEINJOHANN [2002], permet une représentation plus adaptée aux algorithmes de traitement d'images. Le modèle se distingue de la représentation SDF classique par deux points :

— Le fractionnement des images en blocs : au lieu de considérer l'image complète comme un jeton, chaque image est divisée suivant une partition fixe (par exemple une ligne ou un bloc de 8×8 pixels). Cela permet notamment de réduire l'impact mémoire des *buffers*.

— Accès aux blocs voisins : il est possible de définir comme entrée les blocs voisins du bloc courant et/ou les instants précédents du bloc courant.

Ainsi, au lieu d'être notée par un simple nombre de jetons, la consommation d'un acteur dans le modèle CV-SDF est représentée par le triplet $\{c, I_s, I_f\}$, où c est le nombre de blocs consommés par l'acteur, I_s est la dépendance avec le voisinage du bloc et I_f est la dépendance avec les images précédentes.

Globalement la représentation en SDF est en fait une restriction des KPN où chaque nœud consomme et produit une nombre fixe de jeton à chaque exécution. Ces deux modèles impliquent la création d'un *scheduler* (ou ordonnanceur en français) qui se charge de déclencher les exécutions des différents nœuds lorsque les conditions de déclenchement sont valides. Or, dans notre cas, nous considérons des systèmes équipé d'un OS disposant déjà de son propre *scheduler*, ces modélisations paraissent donc non adaptées

à notre utilisation. De plus, nous pensons qu'elles ne sont pas assez complètes pour intégrer correctement les contraintes temps-réel telles que nous les définissons.

3.2.2 Topologie en *kernels*

Notre modélisation choisit de s'inspirer des modélisations précédemment définies. En fait, nous reprenons la représentation en graphe qui permet d'illustrer clairement les dépendances entres les différents blocs d'algorithme, c'est-à-dire une représentation du type *dataflow*. Cependant nous choisissons de pas voir chaque nœud du graphe comme une tâche ou un acteur indépendant mais comme un traitement correspondant à un bloc de l'algorithme. Ainsi, l'algorithme à embarquer est divisé en blocs ayant des entrées et sorties bien définies (en nombre, en type et taille de données). Par définition un algorithme consiste en une description mathématique, il en va donc de même pour les blocs le composant. Il existe plusieurs possibilités pour transcrire un bloc en code source, pseudo-code ou code C. Cette transcription est une implémentation du bloc que nous appelons *kernel*. Finalement, un algorithme peut être décrit par un ensemble de *kernels* $K = \{k_i\}$ et un graphe de dépendances inspiré des traitements de type *dataflow*. Cette description peut avoir naturellement plusieurs solutions pour le même algorithme qui n'ont pas forcément les mêmes performances. Pour ces raisons pratiques, notre méthodologie va prendre en entrée un seul graphe de traitement à la fois, associé à l'algorithme à embarquer. Nous verrons ensuite que l'étude des autres versions de graphe seront nécessaires si l'étude du premier mène à un échec.

Nous proposons de modéliser un graphe de traitement avec les propriétés suivantes :

— *Orienté* : le graphe est composé d'arcs qui autorisent le passage d'un nœud à un autre que dans un seul sens, indiqué par une flèche.

— *Connexe* : il n'existe pas de nœud non connecté.

— *Sans boucle* : il n'existe pas de chemin dans le graphe orienté permettant de relier un nœud à lui-même.

— *À arcs pondérés* : chaque arc du graphe est associé à une valeur correspondant à son poids : $\delta_{i,j}$.

— *À nœuds pondérés* : chaque nœud du graphe est associé à une valeur correspondant à son poids : τ_i.

Le graphe ainsi défini est noté $G = (K, \Delta)$, où K est l'ensemble des nœuds du graphe représentant les *kernels* et Δ est l'ensemble des arcs pondérés de G représentant la dépendance de données entre deux *kernels* adjacents. La valeur du poids $\delta_{i,j}$ de l'arc allant du *kernel* k_i au *kernel* k_j correspond au temps de transfert des données nécessaire entre ces deux *kernels*. Ainsi, on a :

$$\begin{array}{rcl} K & = & \{k_i\} \\ \Delta & = & \{\delta_{i,j}\} \end{array} \tag{3.4}$$

Remarquons que $\delta_{i,j}$ existe si et seulement si k_j dépend des résultats de k_i. De plus, chaque *kernel* a son propre temps d'exécution noté τ_i. Il s'agit en fait du délai entre le début de l'exécution de k_i, lorsque toutes les données d'entrée de k_i sont disponibles, et la fin de l'exécution de k_i, lorsque les données de sorties de k_i sont disponibles.

Pour la suite, il convient dans un premier temps de définir trois sous-ensembles de K :

— $KI = \{k_j \in K \mid \forall i \in [1, card(K)], \nexists\, \delta_{i,j}\}$: les *kernels* sans aucun arc incident appartiennent au sous-ensemble des *kernels* d'entrée.

— $KO = \{k_i \in K \mid \forall j \in [1, card(K)], \nexists\, \delta_{i,j}\}$: les *kernels* sans aucun arc émergeant appartiennent au sous-ensemble des *kernels* de sortie.

— $KP = K \backslash (KI \cup KO)$: le reste des *kernels*

Nous définissons un chemin de traitement $\mathscr{P}_{i,j} = (K_{\mathscr{P}_{i,j}}, \Delta_{\mathscr{P}_{i,j}})$ comme un sous-graphe de G représentant un chemin continu, élémentaire (ne passant pas plusieurs fois par le même nœud) et simple (ne passant pas plusieurs fois par le même arc) entre un *kernel* d'entrée k_i et un *kernel* de sortie k_j. Le sous-graphe ainsi défini est composé de h_{max} nœuds et chacun de ses nœuds a au plus un arc incident et un arc émergent. Pour plus de lisibilité et une meilleure compréhension, chaque *kernel* de $\mathscr{P}_{i,j}$ est renommé kp et est indexé de 1 à h_{max}, avec $kp_1 \equiv k_i \in KI$ et $kp_{h_{max}} \equiv k_j \in KO$. On obtient alors :

$$\begin{aligned} K_{\mathscr{P}_{i,j}} &= \{kp_h, h \in [1, h_{max}] \mid kp_1 \equiv k_i, kp_{h_{max}} \equiv k_j\} \\ \Delta_{\mathscr{P}_{i,j}} &= \{\delta_{h,h+1}, h \in [1, h_{max} - 1]\} \end{aligned} \tag{3.5}$$

Le chemin $\mathscr{P}_{i,j}$ peut ne pas être le seul chemin possible entre k_i et k_j. Ainsi, soit $\mathbb{P}_{i,j}$ l'ensemble des chemins possibles entre k_i et k_j :

$$\mathbb{P}_{i,j} = \{\mathscr{P}_{i,j}\} \tag{3.6}$$

Chaque chemin $\mathscr{P}_{i,j}$ représente une chaîne correspondant à une contrainte temporelle et imposant un ordre d'exécution pour les *kernels* le composant à cause de la dépendance de données. Ainsi, un *kernel* en amont d'un sous-graphe doit s'exécuter avant un *kernel* en aval de ce même sous-graphe. À l'inverse, deux *kernels* k_a et k_b sont considérés comme indépendants si et seulement s'il n'existe pas de chemin qui inclut à la fois k_a et k_b :

$$k_a, k_b \in K, \nexists \mathscr{P}_{i,j} \mid k_a, k_b \in K_{\mathscr{P}_{i,j}} \tag{3.7}$$

Dans ce cas, les deux *kernels* peuvent s'exécuter de manière concurrente sur un même ou différents processeurs. Un exemple de graphe de traitement est donné en figure 3.4.

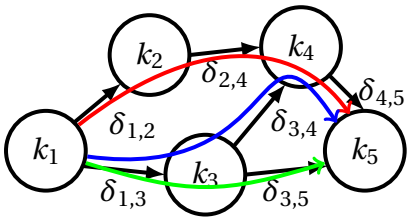


FIGURE 3.4 – Exemple de graphe de traitement composé de cinq *kernels*. Chaque arc implique un temps de transfert de données $\delta_{i,j}$. Dans ce graphe, il y a trois chemins possibles $\mathscr{P}_{1,5}$ allant de k_1 à k_5 : $(\{k_1, k_2, k_4, k_5\}, \{\delta_{1,2}, \delta_{2,4}, \delta_{4,5}\})$ en rouge, $(\{k_1, k_3, k_5\}, \{\delta_{1,3}, \delta_{3,5}\})$ en vert et $(\{k_1, k_3, k_4, k_5\}, \{\delta_{1,3}, \delta_{3,4}, \delta_{4,5}\})$ en bleu. Les *kernels* k_2 et k_3 sont indépendants et peuvent être exécutés en parallèle.

3.2.3 Contrainte temps-réel de latence

Pour une application ADAS typique, les entrées de G sont alimentées par des flux vidéos depuis une ou plusieurs caméras avec un rythme fixe (par exemple une caméra fonctionnant à 30 images par seconde). Ainsi, chaque nouvelle image envoyée par la caméra déclenchera une nouvelle exécution du graphe G. Pour une application donnée, le cahier

des charges fixe des contraintes de latence : il s'agit en fait de limiter le délai entre l'arrivée d'une nouvelle donnée d'entrée depuis un capteur et la sortie associée à cette donnée. Par exemple, dans le cas d'une application de détection de piétons, il peut s'agir du délai entre l'instant où une image avec un piéton est envoyée par la caméra et la prise de décision d'agir ou non sur le frein du véhicule. Du point de vue du graphe, la contrainte de latence se traduit par le délai entre le début de l'exécution d'un *kernel* d'entrée $k_i \in KI$ et la fin de l'exécution d'un *kernel* de sortie $k_j \in KO$. Ce délai doit donc être inférieur à celui spécifié dans le cahier des charges de l'application, noté $dt_{spec}(i,j)$.

Soit $t_{\mathscr{P}}(i,j)$ la latence du chemin $\mathscr{P}_{i,j}$. Cette latence est égale à la somme des temps d'exécution des *kernels* appartenant au chemin plus la somme des délais de transfert de données :

$$t_{\mathscr{P}}(i,j) = \sum_{kp_h \in K_{\mathscr{P}_{i,j}}} \tau_h + \sum_{\delta \in \Delta_{\mathscr{P}_{i,j}}} \delta_{h,h+1} \tag{3.8}$$

La latence entre une entrée k_i et une sortie k_j dépend en fait de tous les chemins possibles $\mathscr{P}_{i,j} \in \mathbb{P}_{i,j}$. Cette latence, $t_p(i,j)$, est égale au maximum des temps d'exécution sur l'ensemble des chemins possibles :

$$\begin{aligned} t_p(i,j) &= \max_{\mathscr{P}_{i,j} \in \mathbb{P}_{i,j}} \left(t_{\mathscr{P}}(i,j) \right) \\ &= \max_{\mathscr{P}_{i,j} \in \mathbb{P}_{i,j}} \left(\sum_{kp_h \in K_{\mathscr{P}_{i,j}}} \tau_h + \sum_{\delta \in \Delta_{\mathscr{P}_{i,j}}} \delta_{h,h+1} \right) \end{aligned} \tag{3.9}$$

Respecter les contraintes de latences implique que tous les chemins $\mathscr{P}_{i,j} \in \mathbb{P}_{i,j}$ aient un temps d'exécution plus petit que la latence maximale spécifiée dans le cahier de charges de l'application, $dt_{spec}(i,j)$:

$$t_p(i,j) < dt_{spec}(i,j) \tag{3.10}$$

Comme chaque image d'entrée suit le même graphe de traitement, chaque nouvelle image n (arrivant après une période d'acquisition T_c) invoquera une nouvelle instance du graphe de traitement noté $G^{(n)} = (K^{(n)}, \Delta^{(n)})$. Cette nouvelle instance est en fait composée des mêmes *kernels* et des mêmes dépendances que l'instance précédente $G^{(n-1)} = (K^{(n-1)}, \Delta^{(n-1)})$. Ainsi, on a :

$$\begin{aligned} K^{(n)} &= \{k_i^{(n)}\} \\ K^{(n-1)} &= \{k_i^{(n-1)}\} \end{aligned} \tag{3.11}$$

De manière plus générale, nous définissons $G^{(n-u)} = (K^{(n-u)}, \Delta^{(n-u)})$ comme le graphe de la u^e instance précédente. Certains *kernels* de $G^{(n)}$ peuvent dépendre de résultats provenant de *kernels* de l'instance précédente $G^{(n-1)}$. On parle alors de dépendances spatio-temporelles, utilisées par exemple dans les algorithmes de tracking.

Nous proposons de modéliser ce type de dépendance en définissant un graphe global $\mathbb{G}$ comme une union infinie de toutes les instances $G^{(n)}$ et les dépendances spatio-temporelles $\Delta^{(*)} = \{\delta_{i,j}^{(*)}\}$ représentant l'ensemble des arcs allant de $G^{(n-1)}$ à $G^{(n)}$. Remarquons que $\mathbb{G}$ est, comme G, sans boucle. Ainsi, on a :

$$\mathbb{G} = \left(\bigcup_{n \in \mathbb{N}} K^{(n)}, \Delta^{(*)} \cup \bigcup_{n \in \mathbb{N}} \Delta^{(n)} \right) \tag{3.12}$$

Pour de très simples graphes de traitement, il est possible d'avoir un $\Delta^{(*)}$ vide. Remarquons que certains *kernels* peuvent dépendre d'une instance plus ancienne que la précédente, par exemple de l'instance $n-2$. Notre représentation par graphe multi-instances

permet de modéliser ce type de dépendance implicitement par récurrence. En effet, si un *kernel* $k_a^{(n)}$ dépend des *kernels* $k_b^{(n-1)}$ et $k_c^{(n-2)}$, alors cela est modélisé par une dépendance spatio-temporelle allant de $k_b^{(n-1)}$ à $k_a^{(n)}$ et une dépendance spatio-temporelle allant de $k_c^{(n-1)}$ à $k_b^{(n)}$. Les dépendances seront alors propagées par récurrence, il est donc suffisant de définir uniquement les dépendances spatio-temporelles entre $G^{(n-1)}$ et $G^{(n)}$. Un exemple de dépendances spatio-temporelles est donné en figure 3.5.

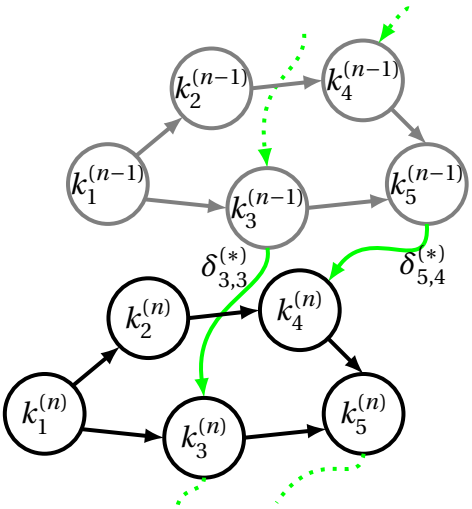


FIGURE 3.5 – Exemple de deux dépendances spatio-temporelles allant de $k_3^{(n-1)}$ à $k_3^{(n)}$: $\delta_{3,3}^{(*)}$ et de $k_5^{(n-1)}$ à $k_4^{(n)}$: $\delta_{5,4}^{(*)}$.

Pour prendre en compte ces dépendances spatio-temporelles, c'est-à-dire dans le cas d'algorithmes fonctionnant sur plusieurs images, il est nécessaire d'adapter le calcul de la latence. Ainsi, nous proposons de définir un chemin de traitement $\mathscr{P}_{i,j,u} = (K_{\mathscr{P}_{i,j,u}}, \Delta_{\mathscr{P}_{i,j,u}})$ comme un sous-graphe de $\mathbb{G}$ représentant un chemin continu, élémentaire (ne passant pas plusieurs fois par le même nœud) et simple (ne passant pas plusieurs fois par le même arc) entre un *kernel* d'entrée $k_i^{(n-u)} \in KI^{(n-u)}$ et un *kernel* de sortie $k_j^{(n)} \in KO^{(n)}$. De la même manière que précédemment, le délai $t_p(i,j,u)$ entre $k_i^{(n-u)}$ et $k_j^{(n)}$ est égal au maximum des temps d'exécution de l'ensemble des chemins possibles :

$$t_p(i,j,u) = \max_{\mathscr{P}_{i,j,u} \in \mathbb{P}_{i,j,u}} \left(\sum_{kp_h \in K_{\mathscr{P}_{i,j,u}}} \tau_h + \sum_{\delta \in \Delta_{\mathscr{P}_{i,j,u}}} \delta_{h,h+1} \right) \tag{3.13}$$

Le délai ainsi calculé doit être inférieur à la valeur spécifiée $dt_{spec}(i,j)$ entre $k_i^{(n)}$ et $k_j^{(n)}$ plus le délai écoulé entre u images (ou entre u instances du graphe), c'est-à-dire $u \cdot T_c$. On obtient alors :

$$t_p(i,j,u) < dt_{spec}(i,j) + u \cdot T_c \tag{3.14}$$

Remarquons que l'équation 3.10 devient juste un cas particulier de la contrainte précédente où $u = 0$. Dans le cas typique, l'analyse temps-réel considère seulement de faibles valeurs de u. Ainsi, lorsque u devient grand, la contrainte $dt_{spec}(i,j)$ devient négligeable devant $u \cdot T_c$ et l'équation 3.14 devient une contrainte imposée par la période du capteur T_c. Les contraintes temps-réel impliquant T_c seront étudiées dans la sous-section suivante.

3.2.4 Contrainte temps-réel de rythme

Par définition, une architecture hétérogène est composée d'un ensemble d'unités d'exécution que l'on notera $PU = \{pu_i\}$. Le *mapping* consiste à déterminer comment répartir la charge de calcul, c'est-à-dire quelle unité d'exécution exécutera quels *kernels*. Soit M la fonction de *mapping*, qui à un *kernel* associe une unité d'exécution sur laquelle il doit s'exécuter :

$$M : K \mapsto PU \tag{3.15}$$

Le temps de calcul d'un *kernel* τ_i et les délais de transfert $\delta_{i,j}$ sont dépendant du *mapping* utilisé. Pour une meilleure lisibilité, dans la suite du document nous utiliserons les noations suivantes :

— $K|_x = \{k_i \in K \mid M(k_i) = pu_x\}$: l'ensemble des *kernels* associés à l'unité d'exécution pu_x.

— $\tau_{i|x}$: le temps de calcul du *kernel* $k_i \in K|_x$.

— $\delta_{i,j|x,y}$: le délai de transfert entre k_i exécuté sur l'unité d'exécution pu_x et k_j exécuté sur l'unité d'exécution pu_y.

Le nombre de *mappings* possibles grandit de manière exponentielle avec la taille de l'ensemble PU et de l'ensemble K. Ainsi, soit $\mathbb{M}$ l'ensemble des fonctions de *mapping* possibles. Le cardinal de $\mathbb{M}$ est donné par :

$$card(\mathbb{M}) = card(PU)^{card(K)} \tag{3.16}$$

Remarquons qu'un *kernel* peut être implémenté pour utiliser plusieurs cœurs de calcul, par exemple en utilisant l'API OpenMP ou *pthread*, et il doit donc être mappé avec plusieurs unités d'exécution, par exemple les cœurs d'un unique CPU. Dans ce cas, la fonction de *mapping* devient multivaluée et renvoie toutes les unités d'exécution utilisées par le *kernel*. De plus, certains *kernels* peuvent ne s'exécuter que sur un type d'unité d'exécution, par exemple les lois de commande automobile ne peuvent pas s'exécuter sur GPU. Tous ces cas particuliers tendent à réduire le nombre de *mappings* possibles donné par l'équation 3.16.

Notre méthodologie permet d'explorer automatiquement l'ensemble des fonctions de *mapping* de $\mathbb{M}$, cela est donc beaucoup plus efficace et rapide qu'une approche humaine classique qui consiste à tester à la main les différents *mappings*. Une fonction de *mapping* M permet de définir comment répartir l'exécution des différents *kernels* sur les unités d'exécution d'un SoC hétérogène, cependant elle ne fixe pas l'ordre dans lequel doivent s'exécuter les *kernels*. Ainsi, pour un *mapping* M, un graphe $\mathbb{G}$ et une période de caméra T_c donnés, il est possible de définir plusieurs pipelines d'exécution. L'exploration des pipelines d'exécution possibles sera abordée ultérieurement dans cette thèse.

Pour respecter les contraintes temps-réel et ne pas « rater » de nouvelles images, une nouvelle instance du graphe $G^{(n)}$ doit être instanciée à chaque nouvelle image acquise depuis la caméra, c'est-à-dire avec une période T_c. Au cours de cette thèse nous avons choisi de nous focaliser uniquement sur un *mapping* statique, donc pour un M donné, $k_i^{(n)}$ est exécuté sur la même unité d'exécution pu_x pour toutes les instances $n \in \mathbb{N}$. Ainsi, respecter les contraintes temps-réel implique que tous les *kernels* d'une même instance associés à une même unité d'exécution doivent s'exécuter en un temps t_{pu_x} inférieur à T_c. On a donc :

$$\forall pu_x \in PU, \quad \sum_{k_i \in K|_x} \tau_{i|x} = t_{pu_x} < T_c \tag{3.17}$$

Remarquons que nous ne considérons pas les délais de transfert δ dans le calcul des temps d'exécution de chaque unité d'exécution pu_x. En fait, nous partons du principe que chaque transfert mémoire est géré par *Direct Memory Access* (DMA) donc de manière asynchrone et utilisant seulement les ressources des unités DMA. Dans le cas où un *kernel* doit effectuer des opérations mémoire synchrones, les délais sont inclus dans le temps de calcul du *kernel* $\tau_{i|x}$. En ayant le temps de calcul de chaque unité d'exécution, il est alors possible de définir et calculer le taux d'occupation de ces unités d'exécution. Ainsi ce taux d'occupation η_{pu_x} est donné par :

$$\eta_{pu_x} = \frac{1}{T_c} \sum_{k_i \in K|_x} \tau_{i|x} = \frac{t_{pu_x}}{T_c} \tag{3.18}$$

3.3 Optimisation du mapping

Comme discuté précédemment, dans le cas de SoCs hétérogènes le temps d'exécution de chaque *kernel* et donc de l'algorithme dépend fortement du *mapping*, c'est-à-dire de la manière dont sont répartis les *kernels* entre les différentes unités d'exécution. Le but de l'optimisation du *mapping* est de trouver une ou plusieurs fonctions de *mapping M* permettant de respecter l'ensemble des contraintes temps-réel telles que définies dans les équations 3.10, 3.14 et 3.17.

Dans le cas où les valeurs de $\tau_{i|x}$ et $\delta_{i,j|x,y}$ sont connues avec une précision infinie, alors les équations précédentes deviennent de simples conditions booléennes et la solution à la problématique du *mapping* est immédiate.

3.3.1 Approche stochastique

Dans la réalité, les valeurs de temps d'exécution ou délais de transfert ne sont jamais connues avec une précision infinie. Généralement, ces valeurs τ et δ sont représentées par des variables aléatoires positives définies par une densité de probabilité (PDF), nulle pour des valeurs négatives (la probabilité qu'un temps d'exécution soit négatif est nulle).

Dans le cas où deux ou plusieurs *kernels* sont exécutés séquentiellement, ou mappés sur des unités d'exécution différentes, nous supposerons que leurs temps d'exécution sont indépendants. Donc les variables aléatoires représentant leurs temps d'exécution sont indépendantes. De plus, comme les transferts mémoire utilisent uniquement les ressources des unités DMA, nous supposerons également que les temps d'exécution τ et les délais de transfert mémoire δ sont indépendants. Pour la suite de la section, les PDFs de τ et δ seront respectivement notées f_τ et f_δ.

Contrainte de latence

Ainsi, en connaissant les PDFs de τ et δ il devient possible de calculer la PDF de la latence $t_p(i,j,u)$ du chemin $\mathscr{P}_{i,j,u}$. Celle-ci est égale à la somme de l'ensemble des variables aléatoires τ et δ appartenant au chemin $\mathscr{P}_{i,j,u}$. Comme nous supposons ces variables aléatoires indépendantes, la PDF de la latence est alors donnée par la convolution de l'ensemble des PDFs des variables aléatoires τ et δ :

$$f_{t_{\mathscr{P}}(i,j,u)}(x) = (f_{\tau_0} * f_{\delta_{0,1}} * f_{\tau_1} * f_{\delta_{1,2}} * \ldots)(x) \tag{3.19}$$

où « ∗ » représente l'opérateur de convolution. La probabilité Pr que le chemin $\mathscr{P}_{i,j,u}$ respecte la contrainte de latence est alors :

$$\Pr\left[t_{\mathscr{P}}(i,j,u) < dt_{spec}(i,j) + u \cdot T_c\right] = \int_0^{dt_{spec}(i,j)+u\cdot T_c} f_{t_{\mathscr{P}}(i,j,u)}(x)dx \tag{3.20}$$

Comme discuté précédemment, il peut exister plusieurs chemins entre un *kernel* d'entrée $k_i^{(n-u)}$ et un *kernel* de sortie $k_j^{(n)}$. Pour que la contrainte de latence soit respectée, il faut que l'ensemble des chemins possibles $\mathscr{P}_{i,j,u} \in \mathbb{P}_{i,j,u}$ respectent cette contrainte, ce qui se traduit en termes de probabilité par :

$$\Pr\left[t_p(i,j,u)) < dt_{spec}(i,j) + u \cdot T_c\right] = \Pr\left[\bigcap_{\mathscr{P}_{i,j,u}\in\mathbb{P}_{i,j,u}} (t_{\mathscr{P}}(i,j,u) < dt_{spec}(i,j) + u \cdot T_c)\right] \tag{3.21}$$

où « ⋂ » est l'opérateur Booléen « ET » itéré sur les contraintes individuelles de l'ensemble des chemins possibles. Or, les temps d'exécution des chemins ne sont pas indépendants, car les différents chemins $\mathscr{P}_{i,j,u} \in \mathbb{P}_{i,j,u}$ ont forcément plusieurs *kernels* en commun (en fait ils en ont au moins deux : le *kernel* d'entrée k_i et le *kernel* de sortie k_j). L'hypothèse d'indépendance n'étant pas vérifiée, il devient très complexe d'évaluer la probabilité de respecter la contrainte de latence pour n'importe quelle f_τ et f_δ.

Contrainte de rythme

Pour une instance n, le temps d'activité de l'unité d'exécution pu_i est donné par la somme des temps d'exécution de l'ensemble des *kernels* mappés sur pu_i. Comme ces *kernels* s'exécuteront séquentiellement sur pu_i, nous supposons que les variables aléatoires représentant leurs temps d'exécution sont indépendants. Ainsi la PDF du temps d'exécution de l'unité d'exécution pu_i sera donc donnée par la convolution de l'ensemble des PDF représentants les temps de calcul des *kernels* mappés sur pu_i. La probabilité de respecter la contrainte de rythme sur l'unité d'exécution pu_i est donc donnée par :

$$\Pr[t_{pu_i} < T_c] = \int_0^{T_c} f_{t_{pu_i}}(x)dx = \int_0^{T_c} (f_{\tau_{1|i}} * f_{\tau_{2|i}} * \ldots)(x)dx \tag{3.22}$$

Si l'on suppose que les temps d'exécution sur différentes unités d'exécution sont indépendants, alors la probabilité de respecter la contrainte de rythme pour l'ensemble des unités d'exécution est donnée par :

$$\prod_{i=1}^{card(PU)} \Pr[t_{pu_i} < T_c] = \prod_{i=1}^{card(PU)} \int_0^{T_c} f_{t_{pu_i}}(x)dx \tag{3.23}$$

Évaluation

Toutes ces probabilités sont difficiles à évaluer pour n'importe quelle PDF f_τ et f_δ. Ainsi, déterminer la PDF de chaque variable aléatoire avec précision, que ce soit par le biais de mesures ou par une méthode de prédiction, est loin d'être triviale. De plus l'évaluation des différentes convolutions et des lois de probabilité jointes est également très complexe.

Pour simplifier ces évaluations, nous proposons d'utiliser soit une distribution Gaussienne (définie par un mode et un écart type) soit une distribution uniforme (définie par

deux valeurs extrêmes) pour chaque f_τ et f_δ. Ainsi, les équations précédemment définies peuvent être évaluées de manière analytique en utilisant une PDF multidimensionnelle d'un vecteur aléatoire composé de l'ensemble des τ and δ du graphe de traitement. Les PDFs de $t_{\mathscr{P}}(i,j,u)$ et t_{pu_x} peuvent alors être évaluées comme une loi de probabilité marginale du vecteur ainsi défini [EVERITT, 1998]. Cette évaluation reste tout de même très complexe à calculer à cause de la non-indépendance de certaines variables aléatoires. Étant donné que le calcul doit s'effectuer pour l'ensemble des *mappings* possibles $\mathbb{M}$, cette approche est plutôt adaptée à des graphes de traitement de petite taille.

3.3.2 Approche par l'arithmétique des intervalles

Un manière plus grossière de caractériser un temps d'exécution ou un délai de transfert consiste à utiliser un intervalle de temps. Ainsi, chaque τ et δ peut être représenté par un intervalle ayant pour valeurs extrêmes des temps maximaux ou minimaux mesurés ou prédits. En utilisant l'arithmétique des intervalles [DAWOOD, 2011], l'évaluation des différentes équations devient triviale. De plus, dans le cas où la valeur maximale d'un intervalle respecte l'inégalité représentant une contrainte temps-réel, il est alors possible de conclure avec une certitude de 100% que la dite contrainte est respectée.

Cependant, cette approche à un gros inconvénient : plus le nombre d'opérations est grand, plus la largeur de l'intervalle résultat est grande et donc plus il est difficile de conclure. En effet, si à la fois les deux bornes de l'intervalle respectent les comparaisons données dans les équations 3.10, 3.14 et 3.17 alors les contraintes temps-réel sont respectées, dans le cas contraire il est impossible de conclure et une autre approche doit être utilisée. Ainsi nous utilisons cette stratégie comme un test initial permettant une réponse rapide mais parfois non concluante.

3.3.3 Approche par fonction de coût

Une autre approche consiste à représenter chaque τ et δ par un scalaire, cela peut être une valeur moyenne ou la médiane d'un intervalle. Ainsi les conditions Booléennes représentant les contraintes temps-réel doivent être remplacées par des fonctions de coût continues. Ces fonctions doivent tendre vers zéro lorsque le temps d'exécution s'éloigne de la contrainte et doivent croître de manière monotone en tendant vers l'infini lorsque le temps d'exécution se rapproche de la contrainte, comme illustré en figure 3.6.

Ainsi, soit $\mathscr{F}$ la fonction de coût définie tel que décrit précédemment. Cette fonction dépend de plusieurs paramètres :

- la fonction de mapping M, qui est en fait la variable de la fonction de coût ;
- le graphe global $\mathbb{G}$ correspondant à l'algorithme à embarquer ;
- l'ensemble des unités d'exécution disponible PU ;
- l'ensemble des contraintes de latence à respecter $\{dt_{spec}\}$ définies par le cahier de charges de l'application ;
- la période d'acquisition T_c définie par le cahier des charges de l'application.

Elle peut se décomposer en deux parties : l'une appliquée aux contraintes de latences (liée à $\{dt_{spec}\}$) et l'autre à la contrainte de rythme. Ainsi, soient :

- $f_l(M,\mathbb{G},dt_{spec}(i,j),T_c)$: la fonction de coût liée à la contrainte de latence $dt_{spec}(i,j)$;
- $f_r(M,pu_x,\mathbb{G},T_c)$: la fonction de coût liée à la contrainte de rythme pour l'unité d'exécution pu_x.

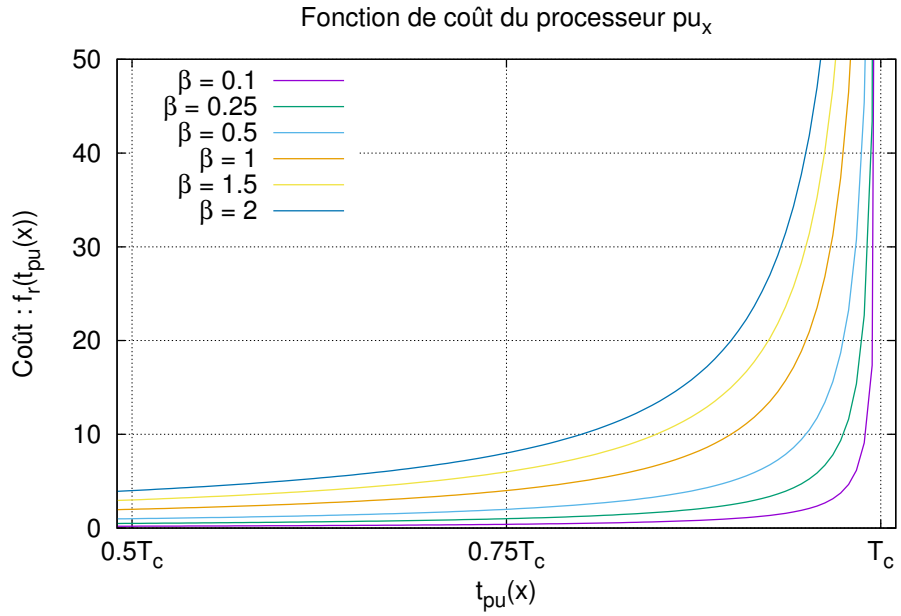


FIGURE 3.6 – Exemple d'une fonction de coût sur la contrainte de rythme donnée pour différentes valeurs de β. Lorsque le temps d'activité du processeur $t_{pu}(x)$ devient proche de la contrainte T_c, le coût $f_r(t_{pu}(x))$ tend vers l'infini.

La fonction de coût globale $\mathscr{F}$ est alors donnée par :

$$\mathscr{F}\left(M,\mathbb{G},PU,\{dt_{spec}\},T_c\right) = \sum_{\{dt_{spec}\}} f_l(M,\mathbb{G},dt_{spec}(i,j),T_c) + \sum_{pu_x \in PU} f_r(M,pu_x,\mathbb{G},T_c) \quad (3.24)$$

À titre d'exemple, nous proposons d'exprimer f_l et f_r comme :

$$f_l(i,j) = \begin{cases} \infty & \text{si } t_p(i,j,u) \geqslant dt_{spec}(i,j) + u \cdot T_c \\ \beta_l \cdot \dfrac{dt_{spec}(i,j) + u \cdot T_c}{dt_{spec}(i,j) + u \cdot T_c - t_p(i,j,u)} & \text{sinon} \end{cases} \quad (3.25)$$

$$f_r(pu_x) = \begin{cases} \infty & \text{si } \eta_{pu_x} \geqslant 1 \\ \dfrac{\beta_r}{1-\eta_{pu_x}} & \text{sinon} \end{cases} \quad (3.26)$$

De cette manière chaque coefficient β permet à l'utilisateur d'ajuster le lissage de la fonction de coût associée et d'associer plus d'importance aux contraintes de latences ou à la contrainte de rythme. La figure 3.6 donne un exemple de plusieurs fonctions de coût avec plusieurs valeurs de β. Ainsi le but est donc de déterminer la fonction de *mapping* M qui minimise la fonction de coût $\mathscr{F}$:

$$\underset{M \in \mathbb{M}}{\operatorname{argmin}}\ \mathscr{F}\left(M,\mathbb{G},PU,\{dt_{spec}\},T_c\right) \quad (3.27)$$

Une approche naïve pour résoudre cette optimisation serait d'implémenter chaque *kernel*, puis de mesurer chaque τ et δ pour différents *mappings*, mais cela nécessiterait un travail énorme et un temps très important. Comme discuté dans [SAUSSARD et collab., 2015b,c], nous proposons de résoudre ce problème de minimisation sans aucune implémentation ni profiling sur l'architecture cible. En effet, nous explorons de manière automatique l'ensemble des *mappings* appartenant à $\mathbb{M}$ en utilisant une méthodologie de prédiction de performances qui sera abordé dans les chapitres suivants.

3.3.4 Contributions sur les travaux existants

Nous avons montré au cours de cette section que le fait d'embarquer un algorithme sur une architecture hétérogène est loin d'être trivial car il n'est pas évident de trouver comment répartir les charges de calcul entre les différents processeurs du même SoC. Notre approche permet de modéliser mathématiquement cette problématique tout en prenant en compte les contraintes temps-réel. Nous avons également proposé différentes approches pour trouver une solution à partir des temps d'exécution et de délais de transfert non exacts (sous forme de PDFs, d'intervalles ou de valeurs moyennes avec une fonction de coût).

En fait, la problématique de *mapping* est un thème bien connu et a été étudiée par bon nombre de contributions scientifiques, comme le montrent SINGH et collab. [2013] dans leur étude sur les techniques de *mapping* existantes. Dans cette étude, les auteurs montrent que les différents travaux à ce sujet peuvent se diviser en deux catégories :

Design-time La résolution du *mapping* se fait lors du portage de l'algorithme sur la cible embarquée et reste constante. Cette stratégie nécessite de connaître le comportement de l'algorithme et son exécution en détail, ce qui n'est pas toujours possible (par exemple en cas de boucle de taille variable).

Run-time La résolution se fait dynamiquement au cours de l'exécution de l'algorithme. Ainsi si une tâche nécessite une puissance de calcul plus grande pour un court instant, il est possible de la mapper de manière temporaire sur un processeur répondant à ce critère. Cependant, cette stratégie implique un surcoût en termes de temps de calcul.

En plus de ces deux catégories, certaines contributions ne se focalisent que sur des architectures homogènes, c'est-à-dire composées de plusieurs processeurs identiques. Nous cherchons ici à comparer nos travaux avec l'état de l'art, nous nous limiterons donc à l'étude des stratégies de *mapping* du type *design-time* visant des architectures hétérogènes.

Il est important de préciser que toutes les contributions à ce sujet ne cherchent pas à obtenir la même chose. Ainsi, dans notre cas, nous cherchons un *mapping* permettant un respect de l'ensemble des contraintes temps-réel, or certains travaux cherchent à optimiser sur d'autres critères. Par exemple, certains se focalisent sur l'optimisation de la consommation électrique, le but étant de cibler des calculateurs nomades (du type smartphone, tablette, etc.) dont la durée de vie de la batterie est un critère très important. On peut donc citer des contributions telle que [MURALI et collab., 2006], [RHEE et collab., 2004], [CHEN et collab., 2008] ou encore [WU et collab., 2003], chacune de ces approches permet une réduction de la consommation d'au moins 50% sur les benchmarks proposés. Étant donné que nous avons choisi de ne pas étudier l'aspect consommation des systèmes embarqués, nous n'entrerons pas plus dans les détails de ces approches.

Concernant l'optimisation du temps de calcul, la plupart des approches existantes cherchent à optimiser le *mapping* d'un algorithme décrit par KPN ou SDF dont le comportement n'est pas déterministe et varie en fonction de l'entrée. Ainsi la plupart des approches, telles que [CASTRILLON et collab., 2013] ou [MICHALSKA et collab., 2016], construisent une trace de l'exécution du KPN ou SDF d'entrée et optimisent le temps de calcul global de l'algorithme suivant différents critères :

— le *mapping* des acteurs,

— l'ordre d'exécution des acteurs,

— la taille des *buffers* entre acteurs.

Cependant, très peu s'intéressent à une optimisation suivant le respect des contraintes temps-réel.

La méthode proposée par MANOLACHE et collab. [2008] cherche à optimiser la répartition de tâches en maximisant la probabilité de respecter les contraintes temps-réel. Il s'agit en fait d'une approche stochastique, où le temps d'exécution de chaque tâche est modélisé par une PDF. Cependant, les auteurs considèrent l'ensemble des variables aléatoires comme indépendantes.

Notre approche est beaucoup plus pragmatique que les stratégies abordées dans la littérature. Ainsi, vu que nous nous adressons au contexte automobile avec une forte contrainte de sûreté de fonctionnement et d'exécution temps-réel, nous prenons seulement en compte des algorithmes dont le temps d'exécution ne varie pas en fonction de la valeur des données d'entrées. Nous n'avons donc pas besoin de construire une trace de l'algorithme. De plus, notre approche est basée sur une modélisation différente des KPN et SDF et nous ne cherchons pas à dimensionner les tailles de *buffers*. En résumé notre approche ne s'intéresse qu'à une partie du problème global tel qu'étudié dans la littérature, mais simplifie énormément la recherche de solution et donc le temps de convergence. En fait, en effectuant une optimisation de type force brute, c'est-à-dire en étudiant toutes les possibilités, la recherche converge en seulement quelques secondes.

3.4 Méthodologie globale

Au cours de ce chapitre, nous avons donc proposé une nouvelle approche pour modéliser un algorithme et les contraintes temps-réel qu'il doit respecter. Cette modélisation définit un découpage en *kernels* de l'algorithme et est représentée par un graphe $\mathbb{G}$ composé d'une infinité d'instances du graphe de traitement G et des dépendances temporelles entre ces différentes instances. Nous avons également proposé une approche pragmatique permettant de modéliser et résoudre rapidement le problème de *mapping*. Ainsi, notre approche se base sur trois stratégies différentes :

Stochastique Les temps d'exécution et les délais de transfert sont représentés par des variables aléatoires. Cette approche est la plus précise, cependant à cause de la non-indépendance de certaines variables aléatoires, les calculs deviennent très complexes dans la majorité des cas.

Intervalle Les temps d'exécution et les délais de transfert sont représentés par des intervalles. Même si cette approche ne permet pas de conclure dans tout les cas, sa mise en œuvre est simple et rapide.

Fonction de coût Les temps d'exécution et les délais de transfert sont représentés par des valeurs moyennes. Une fonction de coût est utilisée pour trouver le *mapping* permettant d'avoir le plus de marge possible par rapport aux contraintes temps-réel.

Finalement, après avoir présenté notre vision de la modélisation d'algorithmes et de la résolution de la problématique de *mapping*, nous définissons notre méthodologie globale d'analyse de l'embarquabilité telle qu'illustrée en figure 3.7. La méthodologie prend en entrée un algorithme à embarquer, un ensemble de SoCs candidats et les contraintes temps-réel définies dans le cahier des charges de l'application ADAS par le constructeur. Dans un premier temps, il faut procéder au partitionnement de l'algorithme en blocs, puis implémenter chacun de ces blocs en *kernels* et ainsi définir le graphe de traitement. Pour chaque SoC, s'il n'a pas déjà été modélisé, la procédure de modélisations est appliquée

(basée sur un ensemble de vecteurs de test), sinon le modèle préexistant peut être réutilisé sans aucun besoin de le modifier. À partir de l'ensemble des modèles des SoCs candidats et du graphe de traitement, notre méthodologie est capable de prédire si le graphe de traitement peut s'exécuter en temps-réel et prédire également le taux d'occupation (en pourcentage d'utilisation des capacités de l'architecture) pour l'ensemble des SoCs candidats. Si le graphe de traitement tel que défini par l'utilisateur ne peut pas s'exécuter avec le respect des contraintes temps-réel sur aucun des SoCs candidats, notre méthodologie est capable de mettre en évidence les goulots d'étranglement et de proposer des solutions pour optimiser l'implémentation de l'algorithme. Ainsi, l'utilisateur peut profiter d'accélérateurs matériels en réimplémentant les *kernels* les plus gourmands en termes de temps de calcul (par exemple en utilisant des ISPs, des instructions SIMD, etc.). Si même après ces optimisations le graphe de traitement n'est toujours pas embarquable, le partitionnement de l'algorithme doit être repensé de manière à obtenir un meilleur degré de parallélisme et donc potentiellement mieux profiter des accélérateurs matériels. Notre méthodologie est capable de prédire l'accélération obtenue par des optimisations matérielles ou logicielles. Pour une application donnée, il devient alors beaucoup plus facile de choisir un SoC en prenant en compte également la puissance électrique consommée et le coût global du système.

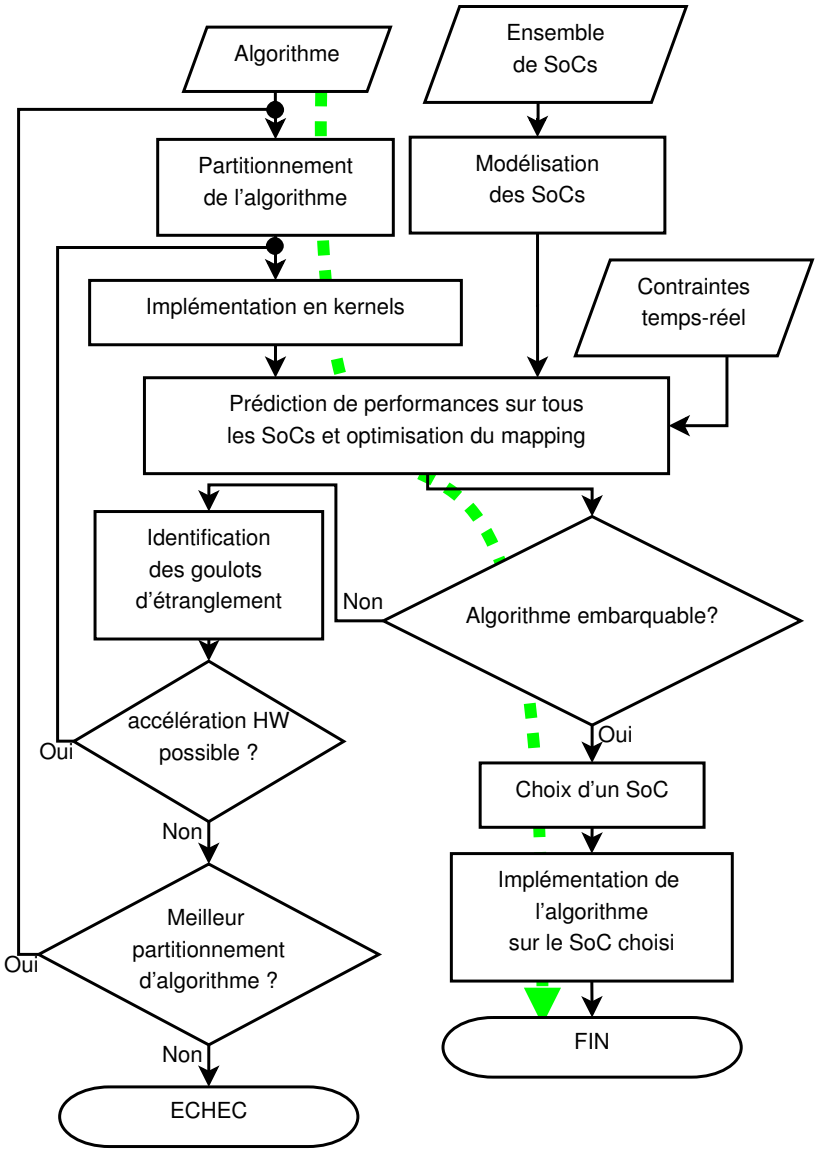


FIGURE 3.7 – Méthodologie globale de l'analyse de l'embarquabilité : elle prend en entrée un algorithme à embarquer et une liste de SoCs candidats. Elle retourne le SoC le plus adapté et des pistes pour le portage de l'algorithme en se basant sur notre modèle de prédiction de performances. La ligne verte représente le chemin typique qui est effectué pour chaque algorithme.

3.5 Références

CASTRILLON, J., R. LEUPERS et G. ASCHEID. 2013, «Maps : Mapping concurrent dataflow applications to heterogeneous MPSoCs», *IEEE Transactions on Industrial Informatics*, vol. 9, nº 1, p. 527–545. 80

CHEN, G., F. LI, S. W. SON et M. KANDEMIR. 2008, «Application mapping for chip multiprocessors», dans *Proceedings of the 45th annual design automation conference (DAC)*, ACM, p. 620–625. 80

DAWOOD, H. 2011, *Theories of interval arithmetic : mathematical foundations and applications*, LAP Lambert Academic Publishing. 78

EVERITT, B. 1998, *Cambridge dictionary of statistics*, Cambridge University Press. 78

KAHN, G. 1974, «The semantics of a simple language for parallel programming», *In Information Processing*, vol. 74, p. 471–475. 69

LEE, E. A. et D. G. MESSERSCHMITT. 1987, «Synchronous data flow», *Proceedings of the IEEE*, vol. 75, nº 9, p. 1235–1245. 70

MANOLACHE, S., P. ELES et Z. PENG. 2008, «Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints», *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, nº 2, p. 19. 81

MICHALSKA, M., N. ZUFFEREY, E. BEZATI et M. MARCO. 2016, «Design space exploration problem formulation for dataflow programs on heterogeneous architectures», dans *IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-16)*, IEEE, p. 217–224. 80

MURALI, S., M. COENEN, A. RADULESCU, K. GOOSSENS et G. DE MICHELI. 2006, «A methodology for mapping multiple use-cases onto networks on chips», dans *Proceedings of the conference on Design, automation and test in Europe (DATE)*, European Design and Automation Association, p. 118–123. 80

RHEE, C.-E., H.-Y. JEONG et S. HA. 2004, «Many-to-many core-switch mapping in 2-d mesh noc architectures», dans *2004 IEEE International Conference on Computer Design (ICCD 2004)*, IEEE, p. 438–443. 80

SAUSSARD, R., B. BOUZID, M. VASILIU et R. REYNAUD. 2015a, «The embeddability of lane detection algorithms on heterogeneous architectures», dans *2015 IEEE International Conference on Image Processing (ICIP)*, IEEE, p. 4694–4697. 66

SAUSSARD, R., B. BOUZID, M. VASILIU et R. REYNAUD. 2015b, «Optimal performance prediction of ADAS algorithms on embedded parallel architectures», dans *2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC)*, IEEE, p. 213–218. 66, 79

SAUSSARD, R., B. BOUZID, M. VASILIU et R. REYNAUD. 2015c, «Towards an automatic prediction of image processing algorithms performances on embedded heterogeneous architectures», dans *2015 44th International Conference on Parallel Processing Workshops (ICPPW)*, IEEE, p. 27–36. 66, 79

SINGH, A. K., M. SHAFIQUE, A. KUMAR et J. HENKEL. 2013, «Mapping on multi/many-core systems : survey of current and emerging trends», dans *Proc. of the 50th Annual Design Automation Conference*, ACM, p. 1–10. 80

STICHLING, D. et B. KLEINJOHANN. 2002, «Cv-sdf-a model for real-time computer vision applications», dans *Proceedings of the Sixth IEEE Workshop on Applications of Computer Vision, 2002.(WACV 2002).*, IEEE, p. 325–329. 70

WU, D., B. M. AL-HASHIMI et P. ELES. 2003, «Scheduling and mapping of conditional task graph for the synthesis of low power embedded systems», *IEE Proceedings-Computers and Digital Techniques*, vol. 150, n° 5, p. 262–73. 80

Chapitre 4

Caractérisation d'architectures de calcul

« *If a machine is expected to be infallible, it cannot also be intelligent.* »

Alan Turing

Sommaire

4.1 Introduction

Comme discuté dans le chapitre précédent, notre méthodologie d'analyse de l'embarquabilité nécessite un modèle d'architecture. Ce modèle doit être générique et réutilisable, c'est-à-dire qu'une fois construit, il doit pouvoir être réutilisé pour tester l'embarquabilité de n'importe quel algorithme. Les données issues de cette étape de caractérisation d'architectures sont ensuite utilisées dans notre méthode de prédiction de performance.

Lors d'une première approche, nous souhaitions construire ce modèle uniquement à partir d'informations disponibles dans les documentations et *datasheets* des fondeurs. Ainsi, une analyse de l'embarquabilité basée uniquement sur ces informations permettrait de prédire le respect des contraintes temps-réels uniquement à partir de la *datasheet* d'un SoC, sans aucune implémentation ni mesure. Nos premières contributions [Saussard et collab., 2015a,b] ont montré qu'il est possible de prédire un temps d'exécution sur différents processeurs uniquement à partir d'informations issues de *datasheets*. Cependant cette approche ne fonctionne que dans le cas de forte I_A (c'est-à-dire lorsque les performances sont limitées par la capacité de calcul) et elle ne permet pas de prendre en compte des informations non communiquées par le fondeur, comme le comportement d'exécutions concurrentes, les effets du compilateur, etc.

Il est très complexe de proposer une prédiction de performances précise dans le cas d'une faible I_A (c'est-à-dire lorsque les performances sont limitées par les capacités de la mémoire) uniquement en utilisant la bande passante théorique de la mémoire et la taille du cache. En effet, le délai d'accès à des données est très variable en fonction de la localisation de ces données, suivant qu'elles se trouvent dans le cache L1, L2 ou dans la mémoire centrale. Dans notre raisonnement nous sommes arrivés à la conclusion qu'il est indispensable de passer par une étape de caractérisation d'architecture en exécutant un programme spécifique permettant d'extraire des caractéristiques non présentes dans la *datasheet* mais indispensables pour une prédiction précise. De plus, dans un contexte industriel, il est tout à fait envisageable de demander à un fournisseur de SoC d'exécuter un programme spécifique permettant la caractérisation de ses architectures. Cependant, cela implique que ce programme spécifique soit le plus générique possible de manière à pouvoir être exécuté sur un très grand nombre d'architectures différentes.

Finalement, notre approche de prédiction de performance se place sur deux niveaux de précision, comme illustré sur la figure 4.1. Dans le premier cas seulement les informations issues des fondeurs (*datasheet* ou autres documents) sont utilisées pour construire le modèle de l'architecture. Cette approche a pour avantage de pouvoir fonctionner sans la nécessité d'avoir le SoC à modéliser sous la main. Cependant, elle ne permet pas de prendre en compte les informations non communiquées par le fondeur du type effet de cache, latence mémoire, comportement des accès mémoire concurrents ou même les effets de compilation.

Ainsi nous proposons un seconde approche qui consiste à extraire des caractéristiques d'une architecture en utilisant un ensemble de vecteurs de test et un processus d'analyse de données bruts. Le but est de pouvoir modéliser :

— Les capacités de calcul pour tout type d'opération (par exemple addition, multiplication, division, etc.) et pour tout type de variable (*int32*, *int8*, *float32*, etc.).

— Les délais d'accès mémoire pour tout type d'opération (lecture, écriture, copie) et les délais de transfert mémoire entre les unités d'exécution du même SoC.

— Les influences de l'environnement logiciel sur les capacités de calcul et mémoire, c'est-à-dire l'influence du compilateur, de l'OS, etc.

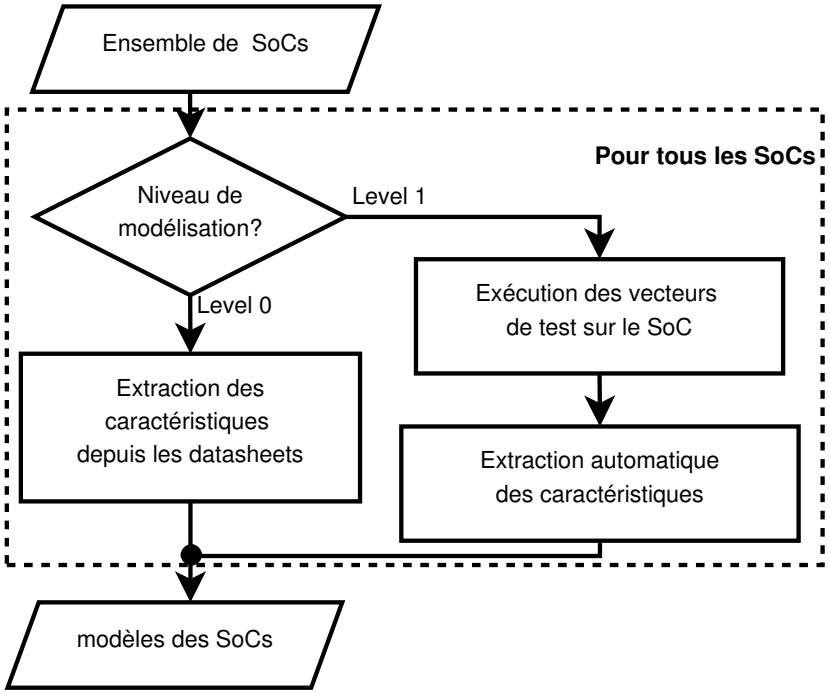


FIGURE 4.1 – Processus de caractérisation des architectures. Il s'agit du détail du bloc « Modélisation des SoCs » de la figure 3.7.

— L'impact d'exécutions concurrentes sur les capacités de calcul et mémoire.
— L'impact de l'utilisation de ressources matérielles ou logicielles spécifiques.

4.2 État de l'art

Dans un premier temps, nous proposons une discussion autour des méthodologies existantes à propos de la caractérisation d'architectures. Lorsque l'on cherche à comparer deux performances il est primordial de définir les critères et les métriques à utiliser pour déterminer la meilleure des deux. Ainsi, dans le milieu du sport l'épreuve du 100 mètres définit le test suivant : l'athlète qui parcours 100 mètres en moins de temps que ses concurrents est le meilleur, on cherche donc ici à qualifier la vitesse de sprint de chaque athlète. Or un champion de 100 mètres ne sera probablement pas aussi performant sur une autre épreuve tel que le marathon ou le saut à la perche. Il est donc très complexe de déterminer quel est meilleur sportif sur tous les plans. Le même raisonnement peut s'appliquer à la comparaison des puissances de calcul entre deux architectures.

4.2.1 Métriques usuelles

Flops

Lorsque l'on parle de puissance de calcul d'un système, on retrouve très souvent les mêmes métriques. Ainsi, le classement des 500 ordinateurs les plus puissants [MEUER et collab., 2016] est donné en *FLoating point Operations Per Second* (FLOPS), ou opérations en virgule flottante par seconde. À l'heure actuelle le calculateur le plus puissant, utilisé au *National Supercomputing Center in Wuxi* (Chine), a une performance mesurée autour de 90 peta FLOPS. Concernant les calculateurs embarqués, la DrivePX de Nvidia fournit une puissance de calcul théorique d'environ 1 tera FLOPS. Or cette métrique ne

prend pas du tout en compte les performances sur les nombres entiers. D'ailleurs elle est souvent mise en avant par Nvidia pour démontrer la performance de ses GPU, comme illustré en figure 4.2. En effet, ceux-ci sont plus performants sur des opérations en virgule flottante qu'avec des opérations sur des entiers, contrairement au processeur classique du type CPU Intel ou ARM. De plus, il n'est jamais précisé le type d'opérations qui est utilisé, à savoir addition, multiplication, division, etc. Si dire qu'une unité de calcul en virgule flottante est aussi performante sur des additions et des multiplications paraît logique et cohérent, cela paraît toutefois moins convaincant de dire qu'il faut autant de cycles pour effectuer une addition qu'une division. Il suffit d'ailleurs de regarder n'importe quelle *datasheet* pour se rendre compte qu'une division prend beaucoup plus de cycles qu'une addition ou une multiplication.

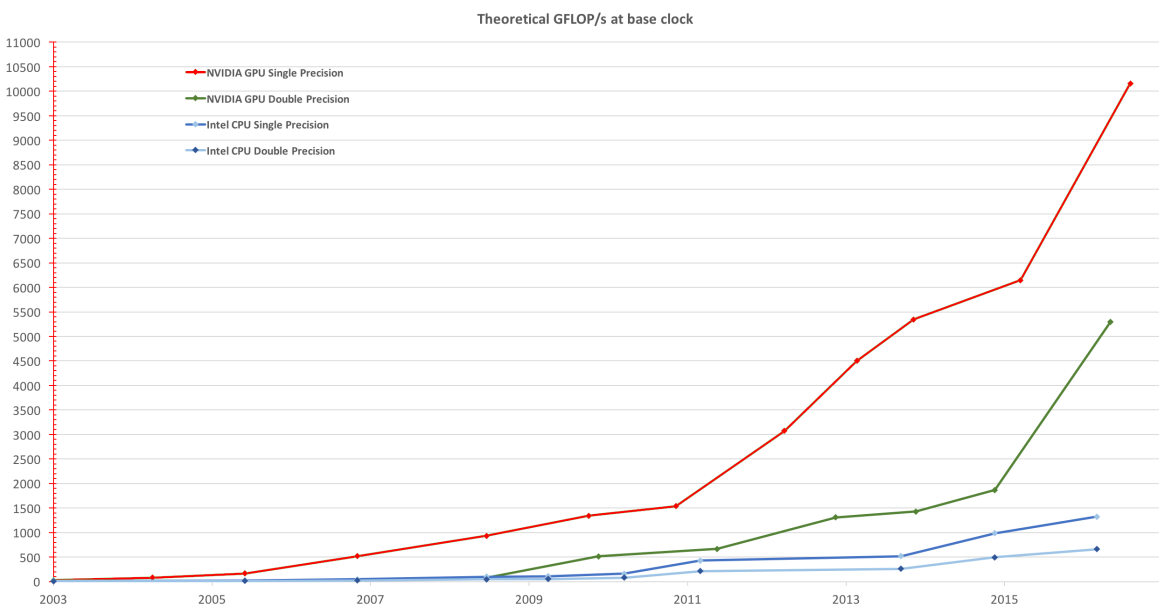


FIGURE 4.2 – Évolution des FLOPS des GPUs Nvidia au cours du temps [NVIDIA, 2015].

Mips

Le *Millions of Instruction Per Second* (MIPS) est une métrique très connue, même si moins utilisée que le FLOPS par le grand public. Elle consiste à mesurer le nombre d'instructions exécutées en une seconde par le processeur. En théorie, la métrique est censée prendre en compte tous types d'instructions : arithmétiques (entières et en virgule fixe), déplacements de registre, opérations en virgule flottante, etc. Une mesure en MIPS s'effectue tout simplement en mesurant le temps d'exécution d'une charge de calcul (également appelée *workload*) dont le nombre d'instructions est connu. Bien évidemment la performance en MIPS d'un processeur dépend fortement du *workload* utilisé, par exemple si celui-ci utilise beaucoup d'opérations en virgule flottante, s'il y a beaucoup de branchements, etc. En plus de la grande dépendance au *workload*, la métrique MIPS a un énorme défaut : il n'est pas possible de comparer deux architectures ayant des jeux d'instructions différents. En effet, le même *workload* écrit en C générera un nombre d'instructions différents pour des architectures n'ayant pas le même jeu d'instructions (par exemple ARMv7 et x86).

D-Mips

Le *benchmark* Drystone, de WEICKER [1984], a été conçu dans le but d'être représentatif de l'utilisation standard d'un CPU. Il ne traite que des opérations entières et donc pas d'opérations en virgule flottante. La sortie du *benchmark* correspond alors au nombre d'itérations de la boucle principale que le processeur est capable d'exécuter en une seconde, cette valeur est notée en *Drystone MIPS* (D-MIPS). Ce *benchmark* a été conçu en 1984 et il est aujourd'hui complètement dépassé par les processeurs et compilateurs actuels. En effet, comme précisé par WEISS [2002], un bon compilateur optimisera énormément le *benchmark* lors de la compilation et donc donnera un score en D-MIPS plus élevé pour le même processeur. L'auteur précise également que la petite taille du code du *benchmark* peut tenir dans le cache d'instruction des processeurs actuels. En plus du fait qu'il n'utilise pas d'opérations en virgule flottante, le *benchmark* Drystone n'est donc finalement pas si représentatif de l'utilisation standard que l'on peut faire d'un CPU.

4.2.2 Autres *benchmarks*

D'une manière générale un *benchmark* consiste à mesurer le temps d'exécution d'un ou plusieurs *workload* dans le but de caractériser une architecture de calcul. Le problème est que cette caractérisation est très fortement dépendante du type de *workloads* utilisé, à savoir les types d'opérations, le degré de parallélisme, la concurrence éventuelle, etc.

Standardisation

Dans le but de proposer des *benchmarks* utilisables par tous pour la comparaison de puissance de calcul des différents processeurs, certains consortiums cherchent à définir des standards.

La *Standard Performance Evaluation Corporation* (SPEC) [SPEC, 1995] est une organisation américaine à but non lucratif, fondée en 1988. Elle a pour but de concevoir et maintenir un ensemble de *benchmarks* standardisés pour la caractérisation de la performance des ordinateurs. Le SPEC-2006 [HENNING, 2006] est un de leur *benchmark* composé de 49 *workloads*, incluant des tests sur des opérations arithmétiques et des tests sur des opérations en virgule flottante se basant sur des applications réelles (par exemple compression de données, compilation, etc.). Cependant il ne caractérise que des architectures du type CPU et n'utilise pas de parallélisme.

De la même manière, le *Embedded Microprocessor Benchmark Consortium* (EEMBC) [EEMBC, 2009] est une organisation à but non lucratif, fondée en 1997, dont le but est de développer des *benchmarks* pour les architectures embarquées. Cette organisation a également pour but de faire du *benchmark* CoreMark [GAL-ON et LEVY, 2009] un standard de l'industrie pour l'évaluation des performances des systèmes embarqués et des environnements logiciels associés. Ce *benchmark* est une évolution du Drystone et retourne un scalaire caractérisant la capacité globale de calcul d'une architecture de calcul. Il est basé sur un ensemble de *workloads* mettant en œuvre des opérations mémoire non-contigues, des opérations matricielles (pour illustrer les accélérations matérielles tel que les instructions SIMD) et des tests du type machine d'état (pour illustrer les performances sur des opérations du type branchement).

Calcul parallèle

Certains *benchmarks* ont été conçus pour caractériser des architectures distribuées massivement parallèles et les caractéristiques qui en découlent. Par exemple le *benchmark* SPLASH-2, de WOO et collab. [1995], propose une méthodologie pour caractériser les performances d'architectures massivement parallèles. Il utilise différents *workloads* du type calcul de FFT ou factorisation de Cholesky. Par la suite, les auteurs montrent que ces *workloads* permettent de caractériser différentes architectures suivant plusieurs critères par opposition aux *benchmarks* précédemment présentés. Les critères mesurés sont :

— accélération du temps de calcul suivant le nombres de processeur utilisés,

— le taux de *cache-miss* en fonction de la taille du cache,

— le ratio du temps de communication entres les différents processeurs par rapport au temps de calcul,

— le trafic d'échange de données en fonction du nombre de processeurs.

Le très célèbre *benchmark* Rodinia, de CHE et collab. [2009], adresse à la fois des architectures du type CPU multi-cœurs et GPU, en utilisant OpenMP et CUDA. Les *workloads* sont inspirés des travaux de ASANOVIC et collab. [2006], "*Berkeley's dwarf taxonomy*". Ceux-ci adressent des domaines très variés, par exemple du *data-mining* avec un *workload* de K-means ou l'analyse de graphe avec un *workload* basé sur l'algorithme de parcours en largeur. Ce *benchmark* est connu pour couvrir un très large panel d'applications et pour avoir des *workloads* faisant grandement varier la puissance électrique consommée. Le *benchmark* Parboil, de STRATTON et collab. [2012] est assez similaire, il utilise également des *workloads* associés à des domaines très variés. Il s'intéresse en plus à la caractérisation des performances apportées par le compilateur.

Scalable Heterogeneous computing (SHOC), de DANALIS et collab. [2010], propose une approche à deux niveaux pour caractériser des systèmes hétérogènes du type CPU et GPU, en utilisant les outils OpenCL et CUDA. Chaque niveau est défini de la manière suivante :

— *Level zero* : il s'agit en fait de mesurer des caractéristiques bas niveau de l'architecture, comme la performance en FLOPS, la bande passante de la mémoire, la vitesse d'échange en CPU et GPU, etc.

— *Level one* : ce niveau caractérise des *workloads* classiques du type calcul de FFT, réduction parallèle, classification, etc.

Caractérisation de la mémoire

Certains *benchmarks* se concentrent particulièrement sur la caractérisation des performances de la mémoire (en ignorant les capacités de calcul). Par exemple le *benchmark* STREAM, proposé par MCCALPIN [1995], mesure la performance de la mémoire avec quatre *workloads* différents :

— Copie : il s'agit tout simplement de copier les données d'une zone mémoire à une autre $a[i] \leftarrow b[i]$

— Copie avec facteur d'échelle : une copie en appliquant un coefficient constant à l'ensemble des données $a[i] \leftarrow q \times b[i]$.

— Somme : $a[i] \leftarrow b[i] + c[i]$.

— Somme et facteur d'échelle : $a[i] \leftarrow b[i] + q \times c[i]$.

Ce *benchmark* est utilisé par WILLIAMS et collab. [2009] pour mesurer la bande passante mémoire réelle des architectures dans la modélisation Roofline.

Le *Low Level Architectural Characterization Benchmark Suite* (LLCBench), de MUCCI [2009], caractérise une architecture en utilisant trois *benchmarks* :

— MPBench pour la mesure de performances sur l'échange de messages basé sur MPI,

— BLASBench pour la mesure de performances sur l'utilisation de routines BLAS (algèbre linéaire) en relation avec l'utilisation de la mémoire,

— CacheBench pour la mesure de performances mémoire.

Cette suite de *benchmarks* retourne un ensemble de résultats bruts, devant être interprétés par l'utilisateur. Un exemple issu de CacheBench est donné en figure 4.3.

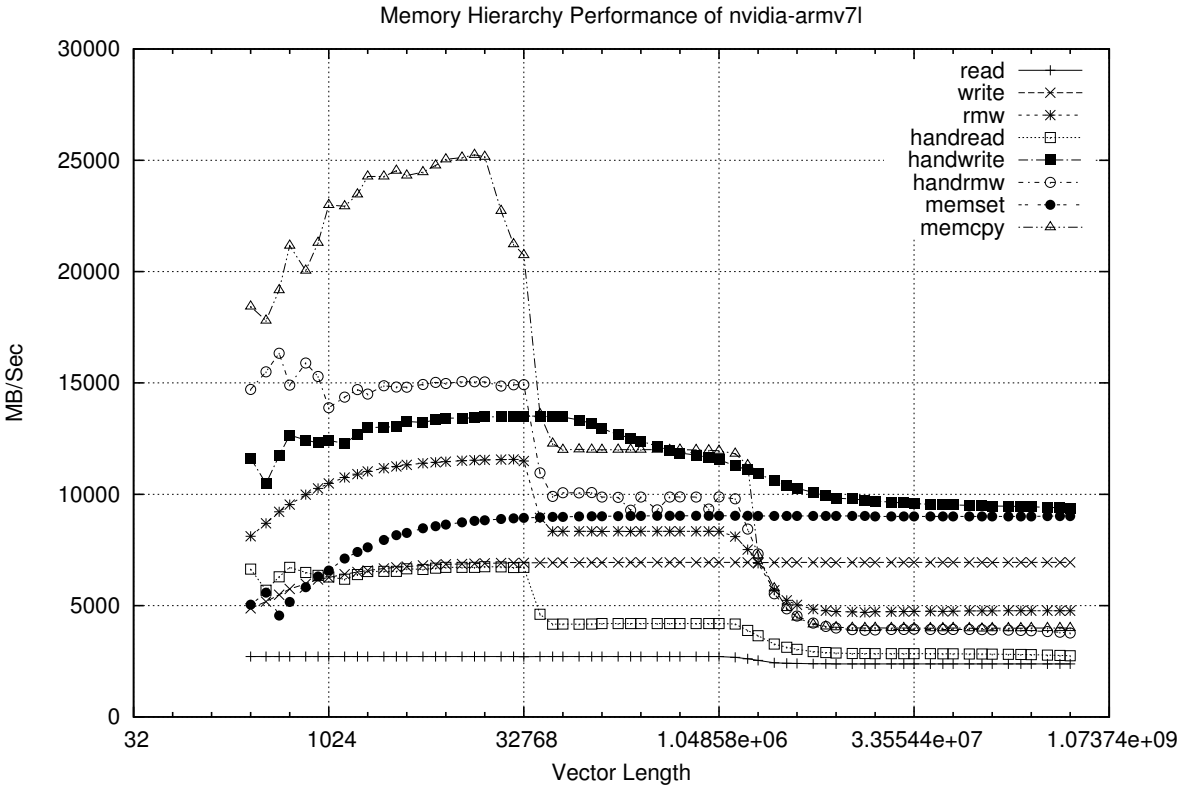


FIGURE 4.3 – Résultats de CacheBench, issu de LLCBench, sur l'ARM A15 du Tegra K1. Les cassures des courbes sont directement corrélées à la tailles de caches L1 et L2 du processeur (respectivement 32 kB et 2048 kB).

4.2.3 Discussion

La plupart des *benchmarks* de l'état de l'art proposent de se focaliser sur des applications spécifiques et ne mettent pas en valeur l'ensemble des capacités d'un processeur (par exemple uniquement l'unité arithmétique et logique ou uniquement l'unité d'opérations en virgule flottante). On remarque quand même que certains proposent une très grande variété de *workloads* permettant de toucher un grand panel de domaines. Pour en revenir à notre besoin, nous cherchons un moyen d'obtenir les caractéristiques d'une architecture pour en construire un modèle. Or, la plupart des *benchmarks* présentés ici proposent juste des ensembles de *workloads* mais sans plus d'analyse sur les résultats. Donc ces *benchmarks* sont utiles si l'on souhaite obtenir une comparaison rapide de plusieurs

architectures sur des applications spécifiques, mais ne répondent pas à notre besoin de caractérisation fine.

On peut retenir les approches de SHOC [DANALIS et collab., 2010], STREAM [MCCALPIN, 1995] et LLCBench [MUCCI, 2009] qui proposent une caractérisation bas niveau des architectures. Cependant cela se limite très souvent à une analyse très sommaire, restant en surface (par exemple quantifier la puissance de calcul par des FLOPS, etc). De plus, ils sont très souvent, plus focalisés sur les performances de la mémoire que sur les capacités de calcul.

Notre constat est donc le suivant, nous avons besoin d'aller plus loin que le *benchmarking*. Cette nouvelle approche doit pouvoir caractériser l'ensemble des unités élémentaires d'une architecture de calcul, les comportements des accès mémoires et aller plus loin que les informations disponibles dans une *datasheet*. Puisque nous travaillons avec des architectures massivement parallèles, nous avons également besoin de caractériser les effets de concurrence, c'est-à-dire le comportement lorsque plusieurs tâches utilisent la même ressource simultanément. Nous avons pour cela développé un ensemble de vecteurs de test, qui ne sont pas des *benchmarks* à proprement parler, mais qui permettent une extraction des caractéristiques d'une architecture. Notre approche est générique dans le sens où elle peut cibler n'importe quel type d'architecture programmable (au sens logiciel), y compris PC. Pour être plus précis, les résultats obtenus à partir de nos vecteurs de test caractérisent le triplet : matériel, système d'exploitation et environnement logiciel (comprenant le compilateur et l'ensemble des outils livrés par le fondeur). Il est important de noter que ces outils sont utilisés tels quels, sans aucune modification, ils intègrent donc potentiellement des bugs et ne sont pas forcément entièrement optimisés. Donc nos caractérisations vont refléter l'état à un instant précis d'un écosystème matériel et logiciel. Si une nouvelle version de l'OS, du compilateur ou du matériel sont sur le marché, alors notre méthodologie permet sa caractérisation rapide et met en évidence les apports (ou les régressions) des mises à jour vis-à-vis des performances de la même architecture.

L'ensemble de nos vecteurs de test permettant de répondre au besoin de caractérisation d'architectures sont divisés en trois catégories :

— des vecteurs de test *low-level* étudiant les caractéristiques bas niveau des architectures,

— des vecteurs de test *mid-level* caractérisant le comportement d'exécutions concurrentes,

— des vecteurs de test *high-level* pour la caractérisation d'application ou de fonctionnalité complète.

Remarquons que l'ensemble de nos vecteurs de test sont implémentés en C++ et utilisent des extensions spécifiques du type OpenMP, SIMD ou CUDA permettant d'utiliser le maximum des capacités d'une architecture. Cette méthodologie de caractérisation est également présentée dans [SAUSSARD et collab., 2016].

4.3 Configuration et compilation

Avant d'introduire le principe de fonctionnement de nos vecteurs de test, il paraît opportun de présenter notre choix pour la mesure du temps d'exécution et notre stratégie de compilation. Ainsi, notre approche est basée sur la mesure du temps d'exécution de tests spécifiquement conçus pour mettre en avant les caractéristiques de l'architecture. Il faut cependant rester conscient que le temps d'exécution d'un *kernel* ne dépend pas

uniquement des capacités de calcul et de la mémoire de l'architecture mais également d'autres facteurs (par exemple le compilateur utilisé et les options de compilation). De plus la technique utilisée pour la mesure du temps peut également avoir un impact sur les résultats des tests.

4.3.1 Mesure du temps

Très souvent, les processeurs proposent des compteurs hardware permettant de mesurer un délai de temps en nombre de cycles. Cette approche est généralement peu coûteuse, puisqu'il s'agit seulement d'aller lire des registres spécifiques, et est très précise. Cependant, ces registres sont propres à chaque processeur, il n'est donc pas possible de définir une procédure de mesure de temps générique en utilisant cette approche. Remarquons également que pour les processeurs ARM, ces registres sont uniquement accessibles en *kernel mode* par défaut, l'accès à ces registres en *user mode* implique d'écrire dans un registre particulier qui est uniquement accessible en *kernel mode*. Puisque nos vecteurs de test ont pour but d'adresser plusieurs types d'architecture, nous avons donc décider d'opter pour une mesure du temps qui soit générique et puisse fonctionner sur un maximum de processeurs. Dans la section 2.3.4 concernant l'environnement logiciel des architectures hétérogènes automobiles, nous avons vu que certains fabricants ont fait le choix d'un OS POSIX, basé sur Linux. Dans les cas où cela est possible, nous avons donc choisi d'utiliser les APIs fournies par l'OS Linux, permettant de s'abstraire des spécificités hardware de chaque processeur. Dans le cas de système hétérogènes, l'OS Linux est généralement hébergé par le CPU. Lors de la caractérisation d'autres processeurs que le CPU (par exemple GPU ou DSP), nous choisissons d'effectuer la mesure indirectement par le CPU hébergeant l'OS, même si ces autres processeurs disposent de leurs propres compteurs hardware. De cette manière, cela nous permet de rester générique et d'être plus conforme à la réalité, puisque nous mesurons le temps écoulé du point de vue de l'OS ce qui inclut les latences potentielles causées par les délais de communications inter-processeurs. Il faut cependant rester conscient que ce choix de généricité implique une perte de précision sur la mesure. Ainsi, si lire des valeurs stockés dans des registres (correspondant aux compteurs hardwares) est très rapide (seulement quelques cycles), l'appel à des fonctions système peut être particulièrement long. Dans le cas où l'on souhaite mesurer des délais particulièrement courts, il est donc préférable de sacrifier la généricité des fonctions système et d'utiliser les compteurs hardwares spécifiques à chaque architecture. Mis à part ce cas spécifique, on peut considérer que les délais mesurés sont suffisamment grands pour utiliser les fonctions système.

Il existe plusieurs fonctions de mesure du temps compatibles avec la norme POSIX. La plus connue des développeurs est probablement la fonction *gettimeofday*, elle retourne la date du système exprimée à l'échelle de la microseconde. Par la suite, la mesure d'un délai d'exécution se fait en mesurant la date avant, puis après l'exécution, et en effectuant la différence entre ces deux dates. Il existe cependant un risque avec cette approche : si l'heure du système est changée entre la mesure de départ et la mesure de fin, la valeur du délai sera faussée. Cela peut notamment être le cas sur des systèmes qui utilisent des mécanismes de synchronisation d'horloge par le réseau (par exemple NTP). Une autre possibilité consiste à utiliser la fonction *clock_gettime*, dont la mesure est exprimée en nanosecondes (à une telle échelle le temps d'appel à la fonction a un impact sur la mesure). De plus, elle dispose de la possibilité d'effectuer la mesure suivant ces différentes configurations :

— **CLOCK_REALTIME :** on utilise l'horloge globale du système, comme pour *getti-*

meofday ;

— **CLOCK_MONOTONIC :** on utilise une horloge monotonique (elle est insensible à un éventuel changement d'heure) ;

— **CLOCK_PROCESS_CPUTIME_ID :** l'horloge tourne uniquement lorsque le processus est actif, il s'agit du temps d'exécution total de tous les *threads* du processus ;

— **CLOCK_THREAD_CPUTIME_ID :** l'horloge tourne uniquement lorsque le *thread* appelant est actif.

Dans notre cas, nous avons choisi d'utiliser la fonction *clock_gettime* dans la configuration **CLOCK_MONOTONIC**. Avec cette configuration, nos tests ont montrés que la latence causée par la mesure du temps est d'environ 400 nanosecondes.

Après avoir discuté autour de l'outil de mesure à utiliser, intéressons-nous maintenant à la méthode de mesure. Ainsi est-il préférable de mesurer le temps d'exécution d'un test une seule fois ou plusieurs fois, est-il préférable de représenter la mesure par une valeur moyenne, un intervalle, une distribution ou un histogramme, etc. Tout d'abord il faut être conscient que sur un OS complet tel que Linux, de nombreuses tâches sont exécutées en plus des algorithmes de l'utilisateur (par exemple les drivers). Il est donc tout à fait probable que deux mesures de temps d'exécution d'un même programme soient légèrement différentes. Nous donnons en figure 4.4 un histogramme des mesures du temps d'exécution d'un test de lecture sur GPU pour différentes tailles de données, mesuré sur un processeur ARM du Tegra K1. Pour chaque taille, la mesure est effectuée 256 fois. On s'aperçoit alors que la mesure du temps d'exécution présente une distribution gaussienne avec quelques valeurs extrêmes. Nous pensons que ces valeurs extrêmes sont causées par d'autres processus en cours d'exécution sur le CPU qui effectue la mesure.

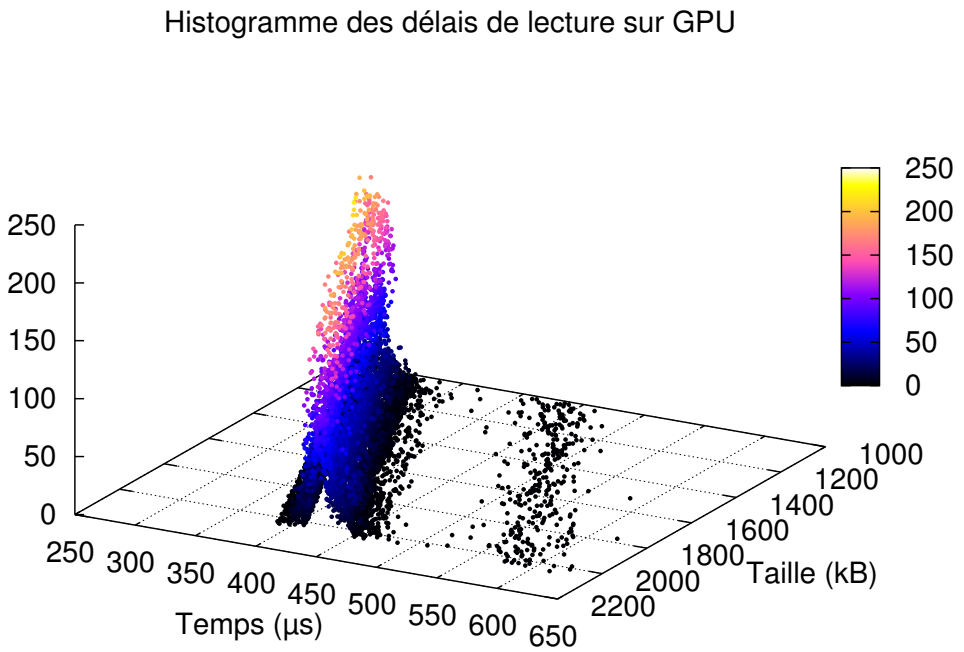


FIGURE 4.4 – Histogramme de la mesure d'un temps de lecture sur le GPU du Tegra K1 pour différentes tailles de données. Pour chaque taille, la mesure est effectuée 256 fois.

La question de savoir si ces valeurs extrêmes doivent être considérées comme aberrantes ou comme pertinentes fait débat. En fait, dans la théorie, un temps d'exécution est

très souvent représenté par une gaussienne, avec une valeur moyenne et un écart type. Dans ce modèle théorique, les valeurs extrêmes doivent donc être considérée comme aberrantes. Or, nous souhaitons caractériser ici le comportement réel de l'architecture, nous devons donc considérer ces valeurs comme pertinentes. Récemment, une nouvelle représentation a été proposée par WORMS et TOUATI [2016]. Les auteurs considèrent un temps d'exécution comme un mélange de gaussiennes et proposent un *framework* complet permettant la manipulation des mesures.

4.3.2 Compilation et optimisation

Un compilateur est un outil qui permet de transformer un code source, tel qu'écrit par le développeur, dans un code binaire spécifique au processeur sur lequel il doit s'exécuter. Il est donc tout à fait logique que la compilation ait un impact non négligeable sur le temps d'exécution d'une tâche. Un compilateur peut agir de différentes manières :

— Transformer le code source en un code binaire assez proche de ce que le développeur a défini, ce qui permet notamment de simplifier le débogage. Sur la plupart des environnements de développement il s'agit de la configuration dite *Debug*.

— Transformer le code source en un code binaire optimisé sur un critère particulier (par exemple le temps d'exécution ou la taille de l'exécutable). Il s'agit très souvent de la configuration dite *Release*.

Les connaisseurs du compilateur GCC connaissent bien évidemment la fameuse option de compilation *-O* qui permet de définir différents niveaux d'optimisations. Nous donnons dans le tableau 4.1 les différentes options d'optimisation et leurs impacts.

On s'aperçoit alors que le temps d'exécution est très fortement dépendant de l'option d'optimisation utilisée lors de la compilation. Dans le cadre de l'embarqué, si l'on essaye d'avoir un regard pragmatique sur la compilation alors deux stratégies peuvent être abordées :

— Si le code à embarquer exige peu de ressources de calcul et qu'il est embarqué sur un processeur faible coût possédant peu de mémoire, on cherchera alors à réduire la taille de l'exécutable (avec l'option *-Os*) ou l'impact mémoire (option *-O0*). Cependant ce genre de cas ne rentre pas dans le contexte de la thèse tel que nous l'avons défini, puisque seule la taille de la mémoire représente le facteur limitant de l'embarquabilité.

— Si le code à embarquer exige une forte capacité de calcul et qu'il est embarqué sur un processeur à haute performance avec une grande quantité de mémoire, on cherchera alors à optimiser au maximum le temps de calcul. Il peut y avoir débat entre les options *O3* et *Ofast*, en fait cette dernière option propose de s'affranchir de certaines règles des standards IEEE et ANSI sur le calcul en virgule flottante pour optimiser le temps de calcul. Il est important de remarquer qu'en cas de mauvaise utilisation, cette option peut s'avérer désastreuse sur le comportement et la stabilité de certains algorithmes.

Le compilateur GCC propose également des options qui sont propres à une architecture, indiquées par le préfixe *-m*. L'exemple typique est l'utilisation de *-march* qui permet de préciser au compilateur le type d'architecture de la cible et donc de proposer une optimisation spécifique à ce type de processeur (par exemple en utilisant des spécificités du jeu d'instructions). Dans le cas de l'ARM A15, où l'unité NEON n'est pas automatiquement présente, il est également possible de préciser au compilateur qu'il peut générer des instructions utilisant cette unité à l'aide de l'option *-mfpu=neon-vfp4*.

TABLEAU 4.1 – Impact des options d'optimisations avec le compilateur GCC sur le temps d'exécution, la taille de l'exécutable, l'utilisation de la mémoire et le temps de la compilation. Un « + » signifie une légère augmentation, « ++ » une augmentation modérée et « +++ » une forte augmentation. De la même manière le nombre de « − » indique une diminution plus ou moins forte.

Option	Optimisation	Temps d'exécution	Taille de l'exécutable	Impact mémoire	Temps de compilation
-O0	optimisation du temps de compilation (par défaut)	+	+	−	−
-O1	optimisation de la taille et du temps d'exécution	−	−	+	+
-O2	optimisation de la taille et du temps d'exécution	−−		+	++
-O3	optimisation de la taille et du temps d'exécution	− − −		+	+ + +
-Os	optimisation de la taille		−−	+	++
-Ofast	*O3* avec une réduction de la précision sur les opérations mathématiques	− − −		+	+ + +

Lors de la phase de caractérisation d'une architecture, les vecteurs de test sont compilés par un compilateur et des options de compilations données, avant d'être exécutés sur cette architecture. Il est important de noter que la caractérisation de cette architecture est alors uniquement valable pour le compilateur et les options de compilation utilisées.

4.4 Vecteurs de test *low-level*

Les vecteurs de test *low-level* ont pour but d'extraire les caractéristiques de bas niveau des architectures, c'est-à-dire d'extraire les capacités de calcul et mémoire brutes pour un environnement logiciel donné. Logiquement nous proposons de diviser ces tests en deux catégories : les tests mémoire qui caractérisent la latence, la bande passante mémoire et les tests de calcul qui caractérisent les capacités opérationnelles.

4.4.1 Calcul

La caractérisation des capacités de calcul consiste à modéliser les temps de calcul en fonction des types d'opération et des types de variables à traiter. Nous commencerons par expliquer le principe général de ce vecteur de test, puis nous exposerons les résultats obtenus sur différentes architectures hétérogènes. Nous donnons dans le tableau 4.2 les configurations hardwares des SoCs utilisées pour présenter nos tests de calcul.

TABLEAU 4.2 – Configurations hardwares des SoCs pour les tests de calcul. La fréquence de chaque processeur a été mesurée et reste fixe au cours des tests.

SoC	Processeur	Cœurs/SMX	Fréquence (MHz)
Tegra K1	ARM A15	4	1800
	GPU Kepler	1	780
Tegra X1	ARM A57	4	1632
	GPU Maxwell	2	806
R-Car H2	ARM A15	4	1400

Principe de fonctionnement

Nous cherchons ici à caractériser les capacités de calcul (somme, multiplication, opérations bit à bit, etc.) d'une architecture. Comme discuté dans la section 1.3, chaque unité d'exécution est composée de plusieurs unités élémentaires permettant à chacune de ses unités d'effectuer des opérations bien définies. Ainsi, la cadence d'une opération dépendra de la capacité de traitement de l'unité élémentaire associée, on s'attend donc à obtenir des performances différentes pour différents types d'opérations. Ce comportement est d'ailleurs bien-connu et il est toujours mentionné par les fondeurs dans les *datasheets*. En plus des différences, nous supposons que les performances sont également liées aux types de données à traiter (par exemple *int32*, *int16*, *int8*, *float64*, *float32*, etc.).

Ainsi pour une opération à caractériser, le test consiste à mesurer le temps d'exécution pour différentes valeurs d'I_A et pour l'ensemble des types de variables. Pour une taille de données fixée, l'I_A est tout simplement accentuée en augmentant le nombre d'opérations. On s'attend alors à obtenir un temps d'exécution en fonction de l'I_A respectant le Boat Hull Model de NUGTEREN et CORPORAAL [2012], c'est-à-dire un temps d'exécution :

— constant lorsque l'I_A est faible : la performance est alors limitée par la bande passante et la latence de la mémoire,

— proportionnel à l'I_A lorsque celle-ci est suffisamment forte : la latence mémoire est alors cachée par les temps de calculs. La pente de cette partie linéaire représente la capacité de traitement du processeur pour l'opération et le type de variable testés.

Le fonctionnement du test est détaillé dans l'algorithme 1. Le paramètre OP correspond au nombre d'opérations par octet lu.

Algorithme 1 : Caractérisation de l'opérateur F sur le type de variable $Type$ pour OP opérations par octet lu.

Result : t : temps d'exécution de la première opération mémoire

```
1  Type* pIn, pOut;
2  Alloc(pIn, SIZE);
3  Init(pIn, SIZE);
4  Alloc(pOut, SIZE);
5  Init(pOut, SIZE);
6  StartChrono();
7  for j ∈ [1 : SIZE/sizeof(Type)] do
8      a = pIn[j];
9      b = 1;
10     for i ∈ [1 : OP × sizeof(Type)] do
11         c = F(a, b);
12         a = b;
13         b = c;
14     pOut[j] = c;
15 StopChrono();
16 t= Chrono();
17 Free(pIn);
18 Free(pOut);
```

Remarquons que la limite entre une I_A faible et une I_A forte est spécifique pour chaque architecture, elle dépend en fait des performances de la mémoire et des capacités de calcul. De plus, ces tests sont basés sur plusieurs boucles dont la taille est connue au moment de la compilation, il est donc possible de mettre en valeur la capacité du compilateur à optimiser le temps de calcul, par exemple en utilisant l'auto-vectorisation [NAISHLOS, 2004] ou le déroulage de boucles. Ces tests sont conçus pour avoir un très haut degré de parallélisme et donc pour exploiter au maximum les processeurs massivement parallèles comme le GPU ou les CPU multicœurs avec des instructions SIMD écrites à la main. Pour caractériser des instructions séquentielles, nous exécutons également ces tests sans utiliser d'accélération hardware, par exemple en utilisant un seul cœur et sans instructions SIMD écrites à la main pour le CPU.

Configurations

Pour l'ensemble des résultats obtenus, nous utilisons les configurations données dans le tableau 4.3. Pour la mesure du temps, nous utilisons la configuration **CLOCK_MONOTONIC**, comme nous l'avons discuté précédemment. L'option *-funroll-loops* permet au compilateur de dérouler les boucles dont la taille est connue à la compilation. L'option *-fopenmp* active l'utilisation de l'outil OpenMP. Concernant la taille des données à traiter, nous travaillons avec des vecteurs de taille 1 Mo, quel que soit le type de donnée.

TABLEAU 4.3 – Configurations de compilation des vecteurs de test calcul pour les architectures testées

SoC	GCC	Options de compilation	CUDA
K1	4.8	*-O3 -mtune=cortex-a15 -mfpu=neon-vfpv4 -funroll-loops -fopenmp*	7.0
X1	4.9	*-O3 -mtune=cortex-a57 -funroll-loops -fopenmp*	7.0
R-Car	4.8	*-O3 -mtune=cortex-a15 -mfpu=neon-vfpv4 -funroll-loops -fopenmp*	—

Résultats sur *int32*

Dans un premier temps, nous donnons les résultats obtenus avec l'addition et la multiplication sur des *int32* dans la figure 4.5, pour le Tegra K1, Tegra X1 et le R-Car H2. Remarquons que les résultats sont donnés pour plusieurs configurations :

— ARM utilisant un seul cœur,

— ARM utilisant un seul cœur et des instructions SIMD implémentées à la main, à l'aide des intrinsèques NEON,

— ARM utilisant quatre cœurs à l'aide d'OpenMP,

— ARM utilisant quatre cœurs et des intrinsèques NEON,

— GPU si disponible.

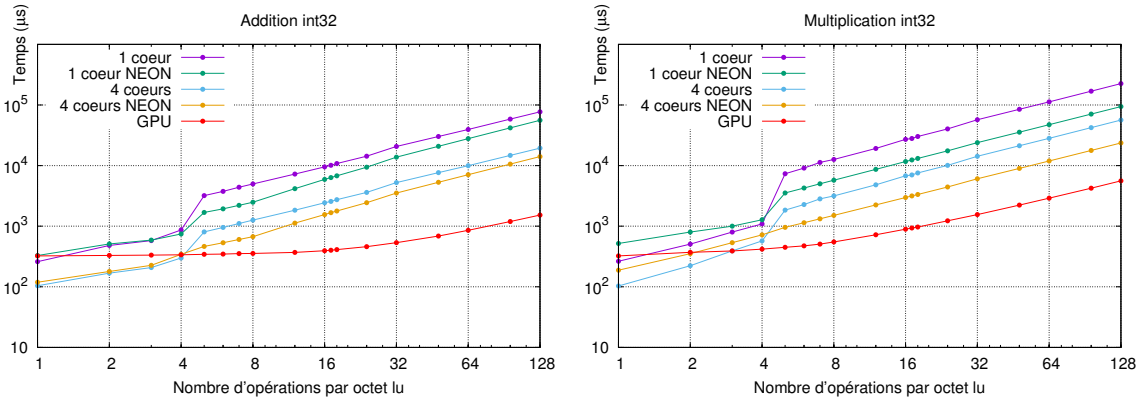
Pour plus de lisibilité, nous choisissons de représenter le temps d'exécution en fonction du nombre d'opérations par octet lu, ce qui correspond en fait à la variable *OP* dans l'algorithme 1. De cette manière, il est beaucoup plus simple de voir l'impact de la taille de la boucle sur le temps d'exécution des tests, la taille de la boucle étant égale au nombre d'opérations par octet lu fois la taille du type de variable testé. Remarquons que comme nous effectuons autant d'opérations de lecture que d'écriture dans ces tests, l'I_A est en fait égale à la moitié du nombre d'opérations par octet :

$$I_a = \frac{OP}{2} \tag{4.1}$$

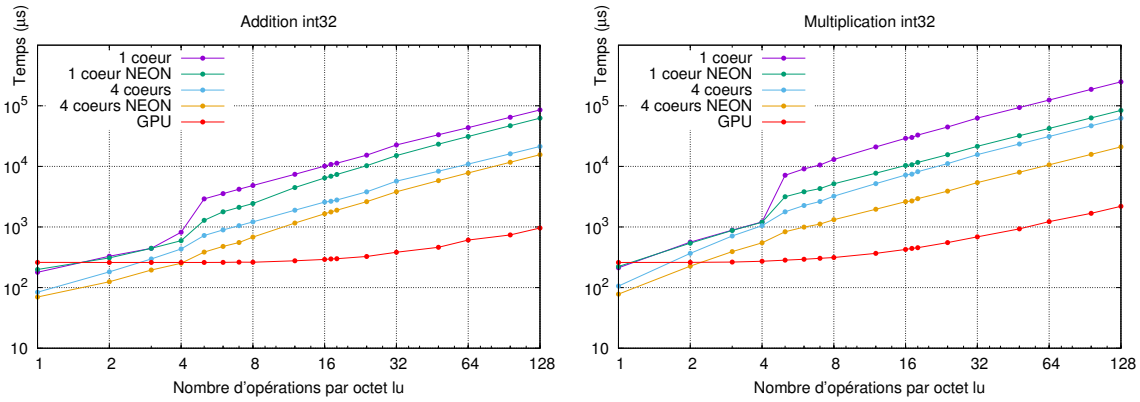
On peut tout d'abord remarquer que d'un point de vue qualitatif les temps d'exécution sont conformes à la prédiction du Boat Hull Model, c'est-à-dire une partie constante lorsque l'I_A est faible et une partie linéaire lorsque celle-ci est forte. En fait, sur cette partie linéaire, la pente de la courbe correspond directement à la capacité de calcul de l'unité élémentaire associée pour le type de variable testé. Comme nous l'attendions également, les temps de calcul sur des multiplications sont plus importants que ceux obtenus avec l'addition sur l'ensemble des architectures. Remarquons que le GPU nécessite une I_A beaucoup plus forte pour être dans la partie linéaire, ce qui nous montre que pour un grand nombre d'applications, les performances des GPUs sont très souvent limitées par la bande passante mémoire.

Concernant la zone où l'I_A est faible, on s'aperçoit que l'utilisation de plusieurs cœurs permet de diminuer les délais d'accès mémoire. La vectorisation semble avoir un surcoût sur les processeurs ARM A15 (Tegra K1 et R-Car H2) car les délais d'accès mémoire sont plus importants avec l'utilisation de NEON. On remarque également, que ce temps est encore plus important sur les opérations de multiplication. Sur l'ARM A57, on peut se rendre compte des apports du nouveau jeu d'instructions ARMv8 ; en fait, les registres SIMD ont été repensés et les temps de chargement sont beaucoup plus rapides que sur le

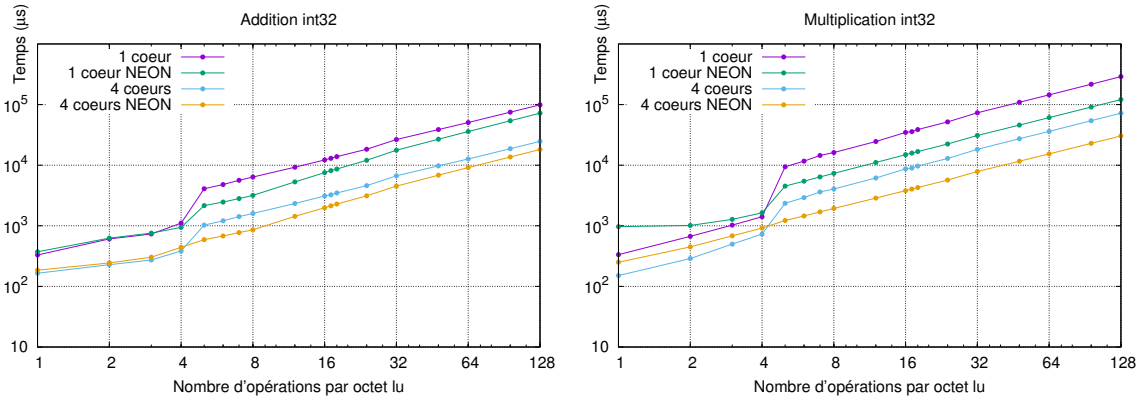
jeu d'instruction précédent (ARMv7). Ainsi, sur Tegra X1 l'utilisation d'instructions NEON n'influence que très peu les délais d'accès mémoire.



(a) K1



(b) X1



(c) R-Car H2

FIGURE 4.5 – Résultats des tests pour l'addition et la multiplication sur des *int32*

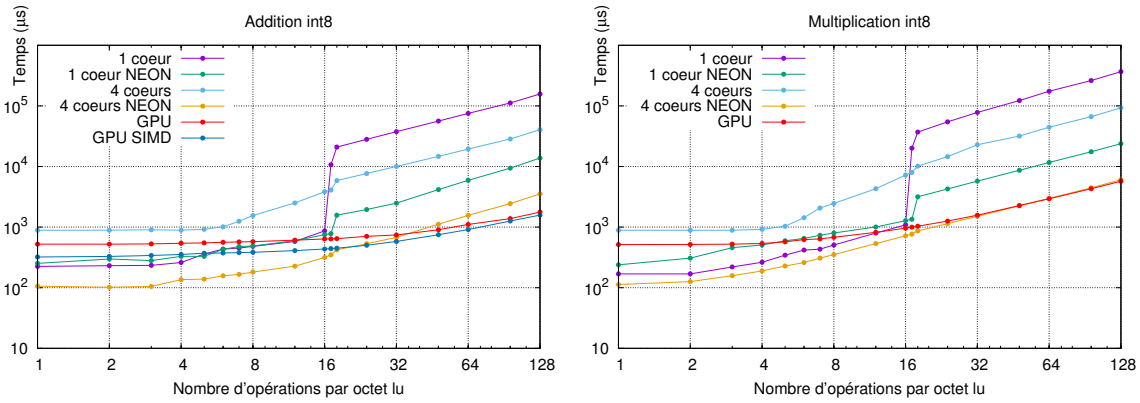
Résultats sur *int8*

Sur la figure 4.6, nous donnons les résultats obtenus pour l'addition et la multiplication des *int8*. Pour l'addition sur GPU, nous donnons également les temps d'exécutions obtenus dans la configuration SIMD. Il s'agit d'instructions CUDA permettant de charger 4 *int8* ou 2 *int16* dans un registre de 32 bits, puis d'appliquer une même opération sur l'ensemble des variables chargées dans ce registre. Par rapport aux résultats obtenus sur des *int32*, plusieurs remarques peuvent être faites. Tout d'abord, dans les configurations

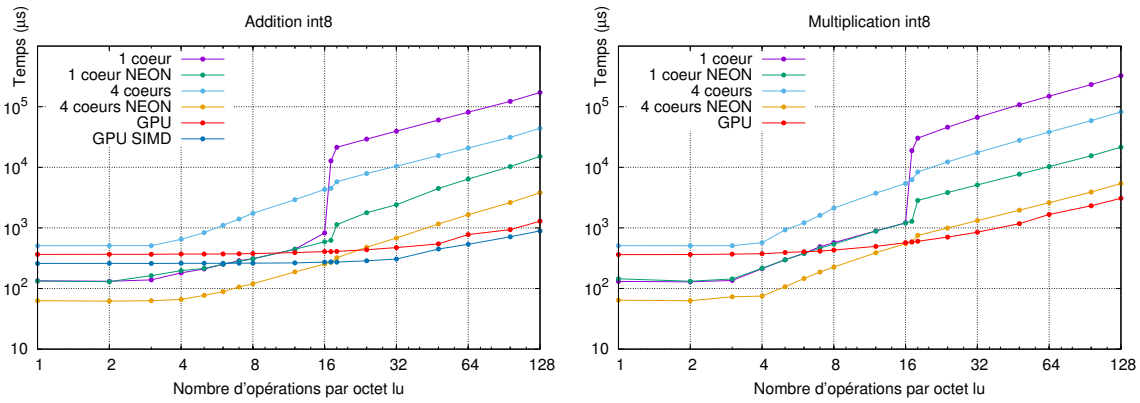
où les instructions SIMD ne sont pas utilisées, les temps de calcul semblent plus longs sur des *int8* que sur des *int32*. Cela valide donc notre hypothèse que les capacités de calcul d'une unité élémentaire varient en fonction du type de variable utilisé. On remarque également que l'accélération obtenue sur les CPU avec NEON est beaucoup plus importante sur des *int8* que sur des *int32*, ce qui est tout à fait logique puisqu'un registre NEON de 128 bits peut contenir 16 *int8* contre seulement 4 *int32*.

Sur la partie à temps d'exécution constante (faible I_A), on peut remarquer que les délais d'accès mémoire sont beaucoup plus importants lorsque l'on utilise 4 cœurs sans NEON que dans les autres configurations. Nous nous sommes aperçus que la parallélisation de la boucle principale à l'aide d'OpenMP empêche le compilateur GCC d'optimiser cette boucle à l'aide de l'auto-vectorisation. Une autre particularité est la cassure sur la configuration 1 seul cœur (courbe violette) qui apparaît lorsque le nombre d'opérations par octet lu est égal à 16. Il s'agit là encore d'une particularité du compilateur, pour une taille de boucle inférieure à 16 le compilateur vectorise de manière automatique (les deux configurations 1 cœur et 1 cœur avec NEON ont des temps d'exécution similaires jusqu'à 16), puis le compilateur change complètement de stratégie et arrête l'auto-vectorisation, d'où cette cassure. On peut remarquer d'ailleurs une cassure identique pour des *int32* qui apparaît pour un nombre d'opérations par octet égal à 4 (soit une boucle de taille 16 pour des *int32*). On a donc identifié ici des particularités propres au compilateur et à l'environnement software qui ne peuvent pas être détaillées dans la documentation hardware fournie par le fondeur. Cela souligne une des forces de notre approche, car nous sommes capables d'identifier et mettre en avant de genre de comportement. Par la suite, ces phénomènes sont directement intégrés dans le modèle de l'architecture (par exemple les délais d'accès mémoire avec OpenMP).

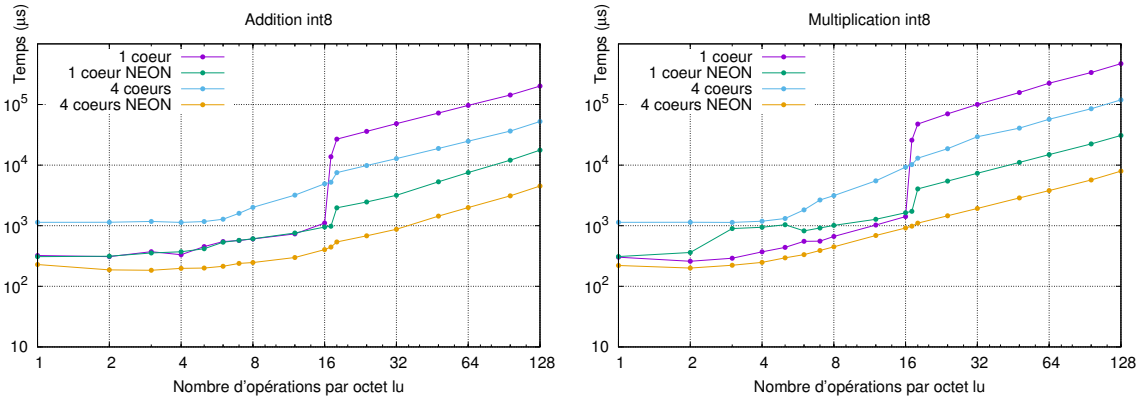
Concernant le GPU, on peut remarquer que les délais d'accès à la mémoire sont plus lents que pour les résultats obtenus sur les *int32* alors que l'on traite la même quantité de données (1 Mo). En fait, cela s'explique tout simplement par le fait que le GPU doit effectuer plus d'accès à la mémoire pour charger $1024 \times 1024/1$ *int8* contre $1024 \times 1024/4$ *int32*. D'ailleurs, dans le cas de l'utilisation des instructions SIMD sur GPU, 4 *int8* sont chargés en même temps dans un registre 32 bits, on retrouve alors le même comportement qu'avec des *int32*. Il s'agit en fait d'un phénomène très connu par les experts en traitement d'image sur GPU, en général on préfère charger les pixels par paquet de 4 (soit $4 \times 8 = 32$ bits) pour optimiser au maximum les accès à la mémoire qui sont très souvent le goulot d'étranglement des applications sur GPU.



(a) K1



(b) X1



(c) R-Car H2

FIGURE 4.6 – Résultats des tests pour l'addition et la multiplication sur des *int8*

Résultats sur les opérations en virgule flottante

Sur la figure 4.7, nous donnons les résultats obtenus avec des multiplications en virgule flottante sur des variables à simple et double précision. Concernant les opérations sur des nombres à double précision, nous n'effectuons pas de mesure sur des instructions SIMD, car les intrinsèques NEON sur des *float64* n'existent pas en ARMv7. Concernant les résultats, nous pouvons remarquer dans un premier temps que les temps de calcul sont plus importants que ceux obtenus avec les nombres entiers, sauf pour le GPU qui à l'air plus performant sur des opérations à simple précision. Sur les processeurs ARM, les temps de calcul semblent similaires entre les opérations à simple précision et celles à double précision. Par contre, cela n'est pas du tout le cas sur GPU qui est beaucoup moins performant sur des nombres à double précision.

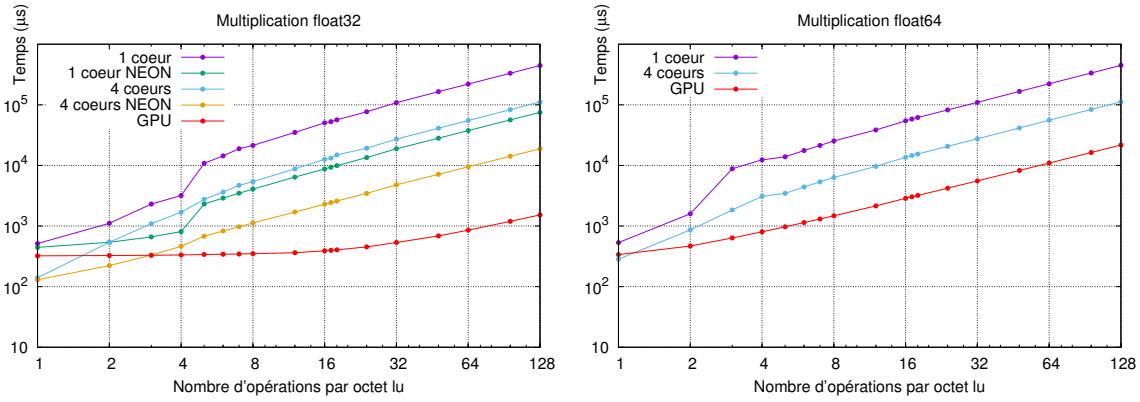
Dans la zone à faible I_A, on peut remarquer sur l'ARM A15 (K1 et R-Car H2) que les délais mémoire obtenus dans la configuration à 1 cœur semblent identiques sur des données à simple et double précision, or les délais sont différents dans le cas de l'utilisation de 4 cœurs. En théorie, puisque la même quantité de données est accédée dans les deux configurations (1 Mo), on devrait obtenir des délais similaires. Le fait que les accès à 4 cœurs soient plus rapide que les accès à 1 cœur peut s'expliquer par la bande passante du cache de chaque cœur qui fait office de goulot d'étranglement. D'après ces résultats, il semble donc que les accès depuis la mémoire à des *float64* soient plus lents que les accès à des *float32*.

Une remarque peut être faite concernant l'apport des intrinsèques NEON sur les variables de type *float32* : si l'on compare l'accélération obtenue avec l'utilisation de NEON sur la multiplication des *int32* et celle obtenue sur la multiplication sur des *float32*, on s'aperçoit que l'utilisation du NEON est plus profitable sur les nombres en virgule flottante. Lorsque l'on souhaite utiliser les instructions SIMD et donc l'unité NEON, il est nécessaire de charger les données dans les registres NEON (de taille 128 bits). Ainsi, il faut donc un certain délai pour charger 4 *int32* dans un registre NEON de 128 bits avant de pouvoir effectuer une opération SIMD, on obtient donc une accélération d'un facteur légèrement inférieur à 4. Or, sur les processeurs ARM ce sont les même registres qui sont utilisés pour les opérations en virgule flottante et les instructions SIMD, il n'y a donc aucun délai nécessaire pour convertir 4 *float32* en un vecteur de 128 bits. On obtient donc logiquement une accélération d'un facteur environ égal à 4 avec l'utilisation des intrinsèques NEON sur des *float32*, soit une accélération plus importante qu'avec des *int32*.

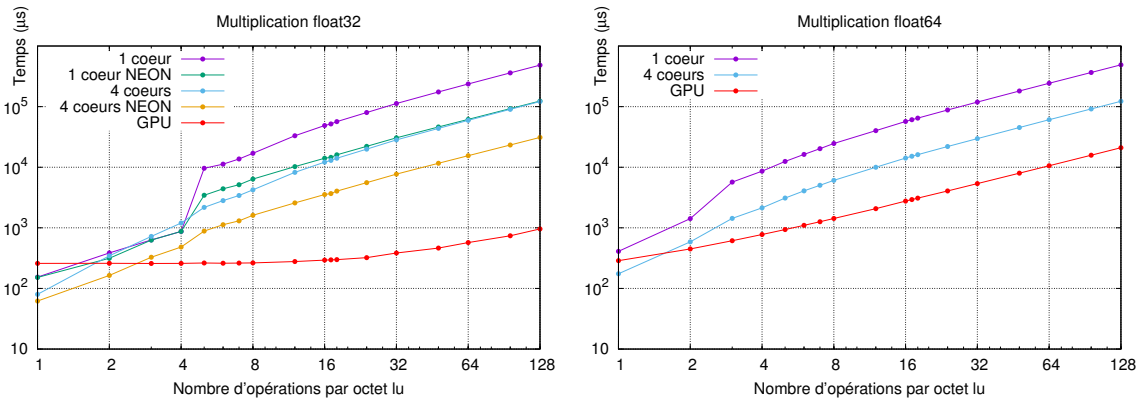
Exploitation des résultats

Comme nous l'avons montré avec les résultats présentés, nos vecteurs de test *low-level* sont capables de mettre en avant des phénomènes qui ne sont pas décrits ni mentionnés dans une documentation technique. Ils nous ont également permis d'identifier des bugs ou comportements atypiques de logiciels utilisés dans l'environnement de ces architectures, comme les compilateurs. Il faut savoir qu'il n'est pas toujours possible de trouver des explications complètes permettant d'interpréter un résultat atypique ou inattendu, d'ailleurs ceci n'est pas le but recherché par notre approche. Ce que nous cherchons à obtenir ici c'est une modélisation précise des comportements de l'architecture de calcul telle quelle est fournie par le fondeur, donc chaque résultat atypique est intégré dans le modèle et est utilisé lors de l'étape de prédiction de performances.

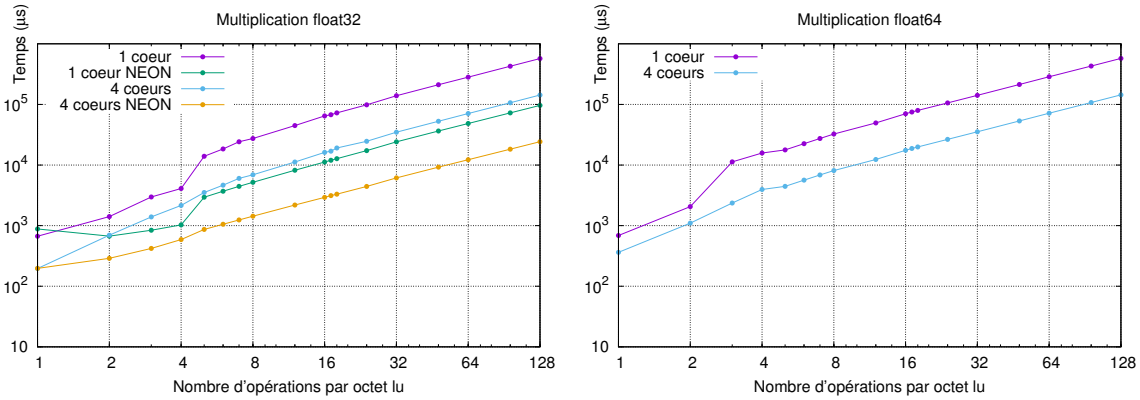
Pour le moment, nous avons juste présenté les résultats sous la forme d'un ensemble de courbes. Comme nous l'avons montré, la pente de la partie linéaire du temps d'exécution peut être exploitée pour obtenir la capacité de calcul de l'architecture pour l'opé-



(a) K1



(b) X1



(c) R-Car H2

FIGURE 4.7 – Résultats des tests pour la multiplication en virgule flottante sur des variables à simple et double précision

ration et le type de données testés. Pour le GPU, cette opération d'exploitation est assez simple, il suffit d'appliquer une régression linéaire puis d'extraire le coefficient directeur de la droite, l'ordonnée à l'origine correspondant à la latence de lancement d'un kernel. Pour les processeurs ARM, l'opération est un peu plus complexe. Ainsi pour des opérations du type ALU, il est nécessaire de prendre en compte les opérations effectuées par les boucles, de plus il faut prendre en compte le fait que le processeur A15 et A57 disposent chacun de deux ALU et deux FPU. Ainsi, dans le cas d'opérations en virgule flottante (donc utilisant les unités FPU), nos tests mettent en œuvre une suite d'opérations dépendantes les unes des autres, le processeur utilisera donc une seule FPU, soit la moitié de la capacité réelle du processeur.

Finalement, nous donnons dans le tableau 4.4 les capacités de calcul obtenues avec l'addition et la multiplication sur différents types de variables pour l'ARM A15 et les GPUs du Tegra K1 et du Tegra X1. Les valeurs mesurées sont comparées avec celles obtenues depuis la documentation technique correspondante. Tout d'abord, il est possible de remarquer que toutes les capacités ne sont pas disponible dans les *datasheets* (marqués *n.a.* dans le tableau ou « ∗ » pour préciser que l'opération nécessite plusieurs instructions). Notre approche permet alors de pallier ce manque d'information. Ensuite, on peut remarquer le léger écart entre les valeurs mesurées et les valeurs théoriques. En fait cet écart peut s'expliquer par la conception de notre vecteur de test et un choix de notre part : après chaque calcul, nous effectuons des opérations de transfert de données entre les registres (instruction *mov*) et donc nous n'exploitons pas au maximum le pipeline d'exécution. Nous avons fait ce choix de conception pour être plus proches des applications réelles.

TABLEAU 4.4 – Capacités de calcul obtenues via les vecteurs de test comparées aux valeurs issues des *datasheets*. Les capacités de calcul sont données en nombre d'opérations par cycle pour le processeur ARM A15 (1 cœur) et les GPUs du Tegra K1 et Tegra X1. Le « ∗ » est défini dans la *datasheet* comme impliquant plusieurs instructions. Les valeurs pour les variables *float* sur l'ARM A15 sont données pour une seule FPU sur les deux disponibles.

Opération	**A15** *datasheet*	**A15** mesuré	**GPU-K1** *datasheet*	**GPU-K1** mesuré	**GPU-X1** *datasheet*	**GPU-X1** mesuré
int8 add	n.a.	1	n.a.	124	n.a.	200
int16 add	n.a.	1	n.a.	124	n.a.	200
int32 add	2	2	160	128	256	225
float32 add	0.25	0.22	192	128	256	225
float64 add	0.25	0.22	8	8	8	8
int8 mult	n.a.	0.22	n.a.	30	n.a.	60
int16 mult	n.a.	0.22	n.a.	30	n.a.	60
int32 mult	0.5	0.4	32	32	∗	84
float32 mult	0.2	0.18	192	128	256	225
float64 mult	0.2	0.18	8	8	8	8

Remarquons que les résultats obtenus ici sont complémentaires des résultats obtenus dans les tests mémoire. Ainsi, nous avons montré par le biais de résultats pratiques, que les délais d'accès mémoire sont très sensibles à un ensemble de paramètres comme le type de variable, l'utilisation d'OpenMP, le nombre de cœurs utilisés, l'utilisation de SIMD, etc. La palette des vecteurs de test est toujours ouverte, elle pourrait, par exemple, être enrichie en rajoutant une dimension : effectuer les mesures pour différentes tailles de données. On obtiendrait alors un temps d'exécution dépendant de 4 paramètres : l'I_A, la configuration (nombre de cœurs, SIMD, etc.), le type de données et la taille des données.

4.4.2 Mémoire

La caractérisation de la mémoire consiste à modéliser les délais d'accès en fonction des différents types d'accès, de la taille et du type des données utilisées. Ainsi, un *int8* est, par définition, quatre fois plus petit qu'un *int32*. On peut donc logiquement penser qu'une lecture de 1024 kB d'*int8* implique donc plus d'accès à la mémoire que la lecture de 1024 kB d'*int32*. De la même manière, un *float32* étant de la même taille qu'un *int32*, on peut s'attendre à des comportements de lecture/écriture assez similaire pour ces deux types de données. Remarquons que pour les processeurs ARM, les variables *float32* sont chargées dans des registres différents que les *int32*, ce qui peut causer un délai supplémentaire. Cependant, en cas d'utilisation des registres SIMD NEON (par exemple avec de l'auto-vectorisation), les variables *int8*, *int16* ou *int32* sont chargées dans les mêmes registres que les variables *float32*. Dans le cadre d'applications ADAS, il est assez rare que l'on utilise des nombres en virgule flottante double précision (*float64*). Ainsi, ce type de variable a un impact mémoire beaucoup plus important Les ECU ne sont pas tous capables de gérer ce type de donnée et les *float32* sont, dans la grande majorité des cas, suffisants en termes de précision. Pour limiter le temps de mesure des vecteurs de test liés à la mémoire, nous choisissions donc de cantonner nos mesures aux variables *int8*, *int16* et *int32*. Les tests restent cependant tout à fait compatibles avec des variables *float32* et *float64*.

Nous commencerons par expliquer le principe général de ces vecteurs de test, puis nous exposerons des résultats obtenus sur différentes architectures hétérogènes. Nous donnons dans le tableau 4.5 les caractéristiques mémoire mesurées des différents SoCs testés. Pour les trois architectures, la taille du cache L2 du processeur ARM est de 2 MB partagé pour tous les cœurs et celle du cache L1 est 32 kB pour chaque cœur. Le détail des tailles de cache des différents processeurs est donné dans le tableau 4.6. Remarquons que pour le GPU du Tegra K1 (architecture Kepler), le cache L1 de 64 kB peut être utilisé comme *shared memory*. Trois configurations sont alors possibles : 16 kB L1 et 48 kB *shared memory*, 32 kB L1 et 32 kB *shared memory* ou 48 kB L1 et 16 kB *shared memory*. Pour le GPU du Tegra X1 (architecture Maxwell), le cache L1 est maintenant unifié avec le *texture cache*, d'une taille de 24 kB. Le GPU dispose d'une *shared memory* de 64 kB par SMX.

TABLEAU 4.5 – Configurations hardwares des SoCs pour les tests de mémoire. La fréquence des différentes mémoires a été mesurée et reste fixe au cours des tests. Pour la plateforme R-Car, nous n'avons pas été en mesure de mesurer la fréquence mémoire, nous donnons donc ses caractéristiques telles que présentées dans la *datasheet*.

SoC	Génération	Fréquence (MHz)	Canaux	Bande passante (GB/s)
Tegra K1	DDR3	792	2 × 32 bits	12.7
Tegra X1	DDR4	1600	4 × 16 bits	25.6
R-Car H2	DDR3	800*	2 × 32 bits	2 × 6.4*

Principe de fonctionnement

Comme nous l'avons vu dans la section 1.3.1, les architectures mémoire deviennent de plus en plus complexes. Ainsi, elles sont composées de différents niveaux de caches qui peuvent être partagés entre différents cœurs ou non. Or, les comportements des accès à la mémoire peuvent être complètement différents en fonction de l'endroit où se trouvent les données et des conditions dans lesquels elles sont lues ou écrites (par exemple un *cache-miss*, la gestion d'un *dirty bit*, etc.) Tout d'abord, nous définissons les actions sur la

TABLEAU 4.6 – Taille des caches pour les SoCs testés pour la mémoire. Les données sont issues des *datasheets* fournies par les fondeurs.

SoC	Processeur	L1 / *shared memory* (kB)	L2 (kB)
Tegra K1	ARM A15 GPU Kepler	32 par cœur 64 par SMX	2048 partagé 128
Tegra X1	ARM A57 GPU Maxwell	32 par cœur 24/64 par SMX	2048 partagé 256 partagé
R-Car H2	ARM A15	32 par cœur	2048 partagé

mémoire que nous cherchons à tester pour l'ensemble des processeurs d'un SoC à caractériser :

— **Lecture** : il s'agit tout simplement de lire une quantité fixe de données en mémoire.

— **Écriture** : il s'agit tout simplement d'écrire une quantité fixe de données en mémoire.

— **Transfert** : il s'agit en fait de transférer des données de la zone mémoire d'un processeur vers celle d'un autre processeur du même SoC. Cette opération met en œuvre le ou les unités DMA du SoC.

— **Copie** : on applique la lecture depuis une zone mémoire pour aller écrire sur une autre zone mémoire du même processeur.

La lecture et l'écriture correspondent aux opérations mémoire basiques (très souvent représentées par les instructions *load* et *store*). Nous cherchons donc à caractériser ces deux opérations dans trois conditions différentes :

— Accès à des données avec un *cache-miss*, c'est-à-dire lorsque les données ne sont pas présentes dans le cache.

— Accès à des données avec un *cache-hit*, c'est-à-dire lorsque les données sont présentes dans le cache.

— Accès à des données avec plusieurs blocs du caches ayant un *dirty bit* levé.

Ainsi, nous avons défini une approche qui met en œuvre l'ensemble des types d'accès à la mémoire pour différentes conditions. En fait après l'allocation des données et avant d'appliquer la mesure, nous proposons d'initialiser la zone mémoire de manière à se placer dans l'une des trois conditions précédemment décrites.

Les tests consistent tout simplement à mesurer le temps nécessaire pour exécuter l'une des quatre opérations mémoire listées précédemment pour différentes tailles de données. Pour l'ensemble des tests, nous proposons de fixer la taille d'allocation en mémoire à une valeur supérieure à la dimension du cache de plus bas niveaux de manière à éviter des phénomènes parasites dans les mesures. Cette valeur correspond à un paramètre des tests mémoire, pour nos mesures nous avons choisi une taille de 4 Mo. De plus, nous avons choisi de nous focaliser sur des accès mémoire contigus, de manière à obtenir la bande passante la plus élevée et être le plus représentatif du domaine du traitement d'images. Il est cependant tout à fait possible d'appliquer la même méthodologie avec des accès non-contigus ou avec des accès aléatoires.

Ainsi, l'approche de caractérisation d'une action sur la mémoire est détaillée dans l'algorithme 2. L'initialisation correspond à appliquer une action sur la mémoire pour se placer dans l'une des conditions définies précédemment, il peut s'agir de lire les données pour les charger dans le cache, d'écrire sur les données pour lever le *dirty bit*, etc. Remarquons que pour éviter d'obtenir des valeurs aberrantes, pour chaque taille nous effec-

tuons plusieurs mesures puis calculons la moyenne sur l'ensemble des mesures. Conformément à notre discussion de la section 4.3, nous choisissons également de conserver la valeur minimale et la valeur maximale des mesures pour chaque taille.

Algorithme 2 : Caractérisation de la mémoire avec une allocation de taille fixe.

Data : pa : pointeur vers les données
Result : t_1 : temps d'exécution de l'opération mémoire

```
1 for size ∈ [1 : MAX] do
2     Alloc(pa, MAX);
3     Initialisation(pa, size);
4     StartChrono();
5     MemoryAction(pa, size);
6     StopChrono();
7     t= Chrono();
8     Free(pa);
```

En plus des conditions de cache, nous pensons également que le type de données utilisé a un impact sur les délais d'accès à la mémoire. Ainsi, nous appliquerons donc ces différents tests sur trois types de données : *int8*, *int16* et *int32*. Remarquons que pour ces types la taille d'allocation reste toujours à 4 Mo. Finalement, ces tests retournent pour chaque type d'opération mémoire, pour les différentes configurations et pour différents types de données un délai en fonction de la taille des données à traiter. Pour une taille donnée, le résultat correspond à la valeur moyenne de l'ensemble des mesures, la valeur minimale et la valeur maximale.

Configurations

Pour l'ensemble des résultats obtenus, nous utilisons les configurations données dans le tableau 4.7. Pour la mesure du temps, nous utilisons la configuration **CLOCK_MONOTONIC**, comme nous l'avons discuté précédemment. Concernant l'évolution de la taille de données (paramètre *size*), nous choisissons de l'augmenter linéairement avec un pas de 4 kB, qui correspond en fait à la taille d'une page mémoire.

TABLEAU 4.7 – Configurations de compilation des vecteurs de test mémoire pour les architectures testées

SoC	GCC	Options de compilation	CUDA
K1	4.8	*-O3 -mtune=cortex-a15 -mfpu=neon-vfpv4*	7.0
X1	4.9	*-O3 -mtune=cortex-a57*	7.0
R-Car	4.8	*-O3 -mtune=cortex-a15 -mfpu=neon-vfpv4*	/

Résultats des tests de lecture

Nous commençons donc par donner les résultats pour les opérations de lectures. Pour confirmer notre approche initiale, nous proposons dans un premier temps d'étudier les comportements de lecture en fonction des opérations précédentes qui ont été faites sur la même zone mémoire, c'est-à-dire les lectures dans les conditions suivantes :

— Test **Read** : il s'agit de la première lecture après l'allocation de données, celles-ci ne sont pas présentes dans le cache, elles sont lues depuis la mémoire, on devrait observer un comportement linéaire. En théorie il devrait correspondre au pire cas, c'est-à-dire la configuration où les délais de lecture sont les plus longs. Ce test simule le comportement d'un *cache-miss*.

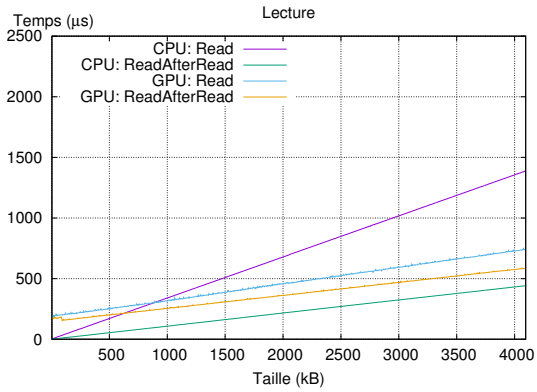
— Test **ReadAfterRead** : il s'agit d'aller lire des données qui ont déjà été lues précédemment, les données sont donc très probablement présentes dans le cache, on devrait observer un temps de lecture inférieur à la première lecture. En théorie il s'agit du cas le plus favorable, là où la latence des accès est la plus faible. Ce test simule le comportement d'un *cache-hit*.

— Test **ReadAfterWrite** : il s'agit d'effectuer une lecture sur des données qui ont été écrites précédemment, les données sont donc très probablement présentes dans le cache mais on est également en présence d'un grand nombre de blocs du cache avec le *dirty bit* levé, on devrait donc observer un délai supplémentaire dû à la mise en cohérence de la mémoire.

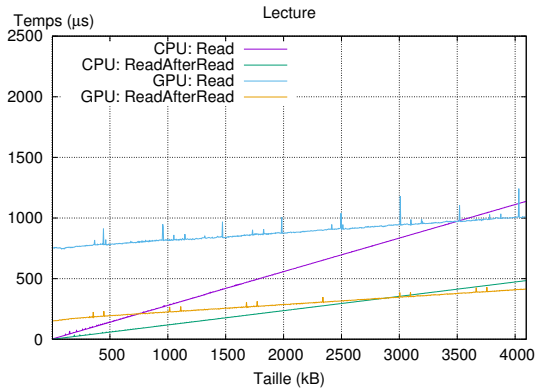
Les résultats sont donnés en figure 4.8, pour des lectures sur des *int32* sur ARM et GPU. On peut remarquer tout d'abord que sur l'ensemble des processeurs ARM, la lecture se comporte comme nous l'avons prévu c'est-à-dire un délai très rapide dans le cas d'un *cache-hit* (test ReadAfterRead) et un délai de lecture beaucoup plus important si la lecture est effectuée sur des données non présentes dans le cache (test Read). De plus, les délais de lecture sont parfaitement linéaires dans les deux cas. Les résultats sur le test de ReadAfterWrite pour le processeur ARM seront présentés ultérieurement dans le document. Sur GPU, le phénomène n'est pas présent, on observe le même comportement pour les tests ReadAfterRead et ReadAfterWrite (pour plus de lisibilité sur les courbes, nous avons choisi d'afficher uniquement les résultats du test ReadAfterRead). Remarquons que pour le GPU, l'ordonnée à l'origine n'est pas nulle, elle est en fait égale à la latence de lancement du test. En effet, lorsque l'on commande l'exécution d'un kernel sur le GPU avec l'API CUDA, il faut un certain temps avant que celui s'exécute réellement, d'où cet offset sur la mesure. On remarque d'ailleurs que la latence de lancement du kernel de test Read est plus élevée que les deux autres. La latence de la première exécution d'un kernel est toujours plus grande que les latences de ses exécutions suivantes, c'est pour cela que l'on observe une latence différente entre le test Read et les deux autres tests. De manière assez surprenante, la latence du test Read sur Tegra X1 est très largement supérieure à la latence de lancement des autres tests (environ quatre fois plus grande). Finalement, on peut remarquer que pour les plateformes Tegra K1 et Tegra X1 la bande passante mémoire (qui est inversement proportionnelles à la pente des différentes courbes) semble plus rapide pour le GPU que pour le processeur ARM, dans le cas du test Read. Sur Tegra X1 on remarque également que même le test ReadAfterRead a une bande passante plus rapide sur GPU que sur ARM.

En plus de l'impact du cache, nous supposons également que le type de données doit avoir un impact sur les délais d'accès en lecture. Ainsi nous avons identifié dans les tests de calcul présentés précédemment que le type de variable (par exemple *int8*, *int16* ou *int32*) a un très fort impact sur les délais mémoire du GPU. Pour illustrer ce phénomène, nous donnons en figure 4.9 les temps de lecture sur GPU pour différents types de variable.

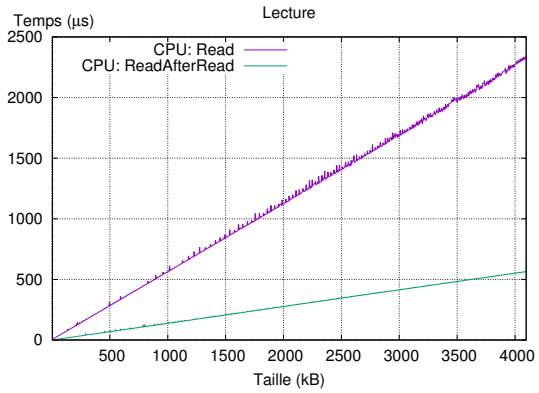
On s'aperçoit alors, comme nous nous y attendions, que le type de variable a un très fort impact sur le délai de lecture. Au niveau de la pente (c'est-à-dire en faisant abstraction de la latence de lancement des kernels de test), on observe logiquement que les lectures sur des *int32* sont quatre fois plus rapides que sur des *int8* et deux fois plus rapides que sur des *int16*. Cela s'explique tout simplement par le nombre d'accès qui est



(a) K1

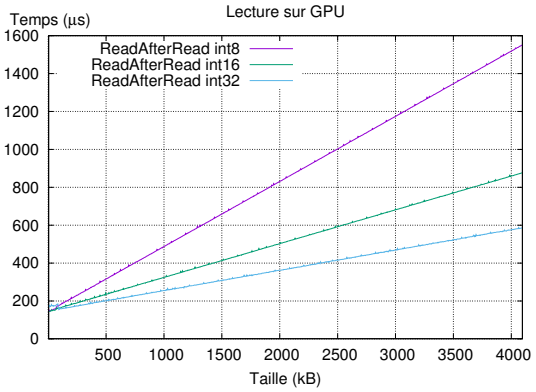


(b) X1

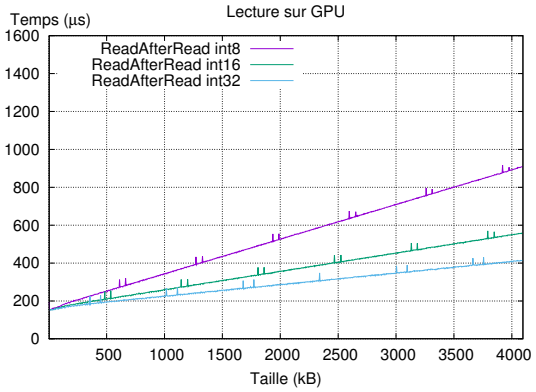


(c) R-Car H2

FIGURE 4.8 – Résultats des tests de lecture sur des *int32*



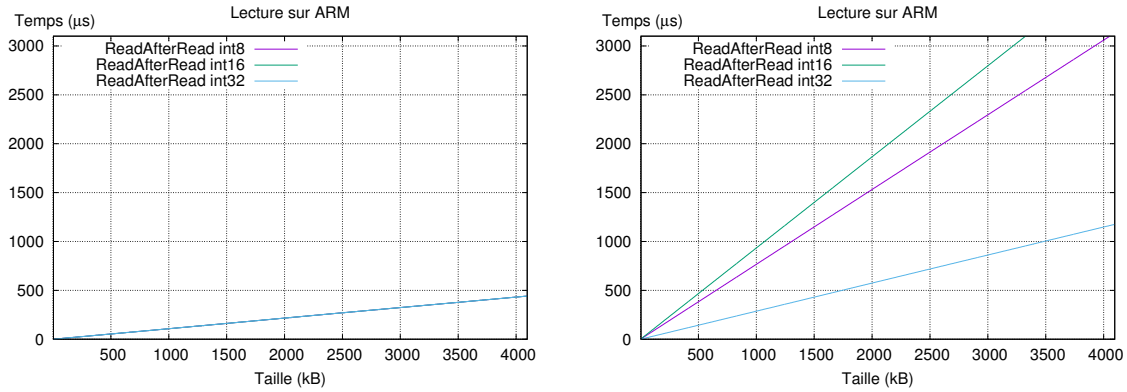
(a) K1



(b) X1

FIGURE 4.9 – Résultats du test ReadAfterRead sur GPU pour différents types de données

plus important lorsque l'on accède à des données plus petites. Sur le processeur ARM, nous avons vu dans les résultats des vecteurs de test de calcul que le type de variable n'a que très peu d'impact sur les délais d'accès à la mémoire. Cela s'explique par le fait que les données sont vectorisées et donc adressées en mémoire par paquet de 128 bits. Pour s'en convaincre, nous donnons en figure 4.10 les délais de lecture sur un processeur A15 lorsque l'auto-vectorisation est désactivée (c'est-à-dire sans l'option *-mfpu=neon-vfpv4*). Ce phénomène a d'ailleurs été discuté précédemment à propos des résultats sur les tests de calcul avec le fameux bug OpenMP qui empêche la vectorisation automatique du compilateur et réduit donc considérablement les délais d'accès à la mémoire sur des *int8*.



(a) Avec vectorisation (les 3 courbes sont confondues)

(b) Sans vectorisation

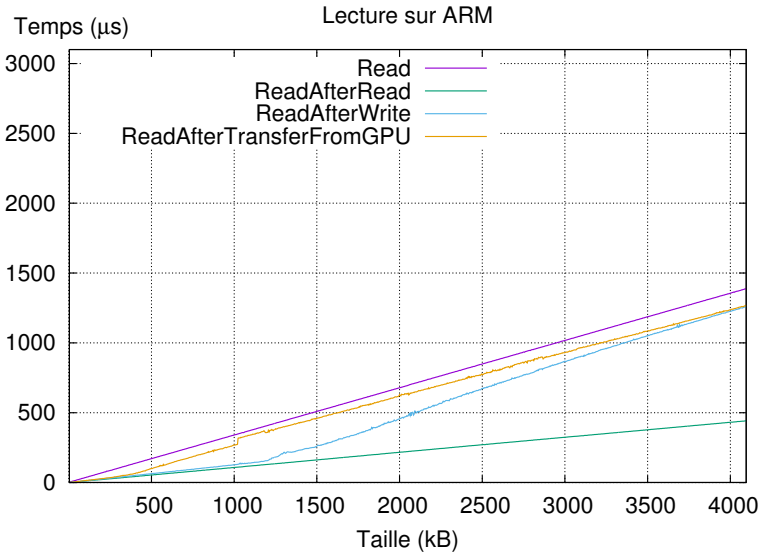
FIGURE 4.10 – Résultats du test ReadAfterRead sur l'ARM du Tegra K1 pour différents types de données

Ensuite, nous choisissons de nous intéresser plus en profondeur au processeur ARM de ces différents SoCs. Comme nous l'avons vu jusqu'à maintenant, les délais des accès mémoire par le processeur ARM sont plus lents que les accès depuis le GPU, ils sont très sensibles au contexte du cache, mais ils ne sont pas sensibles au type de données grâce à la vectorisation effectuée par le compilateur.

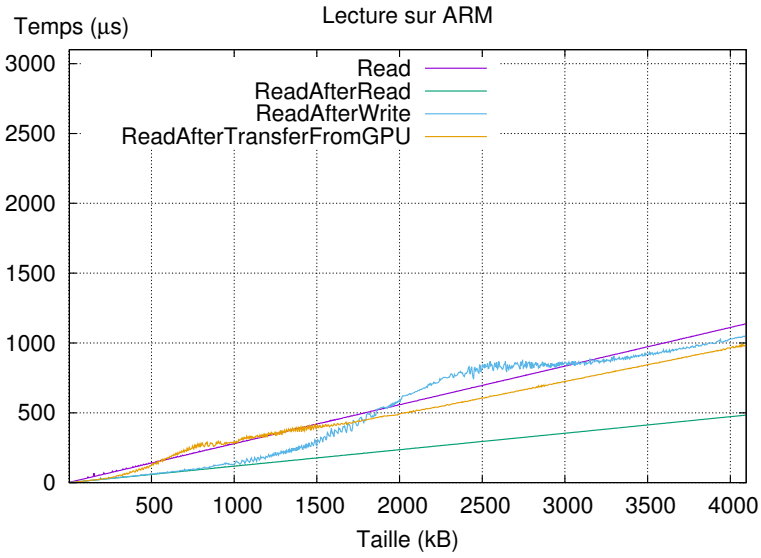
Ainsi, nous donnons les résultats des tests de lecture obtenus sur Tegra K1, Tegra X1 et R-Car H2 en figure 4.11. En plus des tests présentés précédemment, nous pensons également qu'il est intéressant d'étudier le comportement d'une lecture sur des données qui viennent d'être écrites par un transfert DMA. Pour le Tegra K1 et Tegra X1, nous notons ce test **ReadAfterTransferFromGPU**. Étant donné que nous n'avons pas accès à l'accélérateur du RCar H2, nous ne sommes pas en mesure de donner le résultat de ce dernier test pour ce SoC.

Tout d'abord, si l'on s'intéresse à la pente des délais de lecture depuis la mémoire (test Read), on s'aperçoit que les bandes passantes théoriques de la mémoire de chaque SoCs sont respectées : la bande passante mémoire du Tegra X1 est la plus rapide, suivie du Tegra K1 et enfin le R-Car H2 qui a la mémoire la moins rapide. Cependant, si l'on regarde les bandes passantes des lectures depuis le cache (test ReadAfterRead), on s'aperçoit qu'elles sont quasiment identiques pour les trois SoCs alors que leurs bandes passantes théoriques sont différentes. Il semblerait donc que les bandes passantes des caches de ces trois SoCs soient très proches.

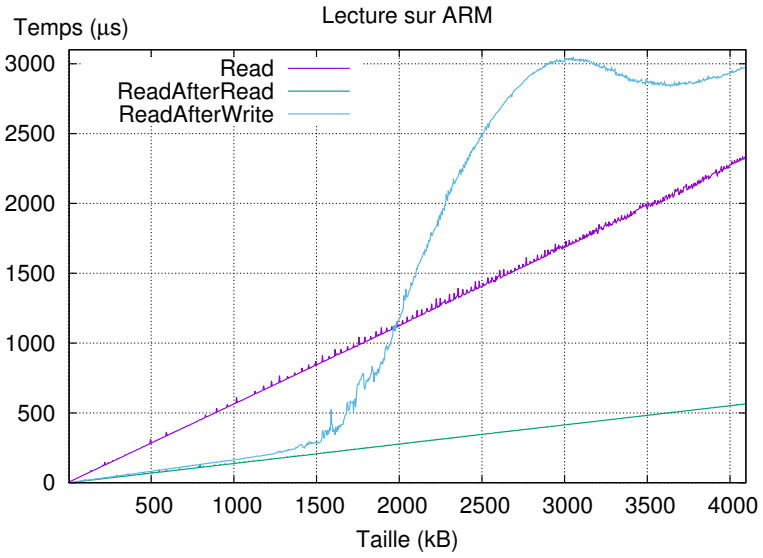
On observe un comportement non linéaire pour les trois plateformes sur les lectures avec un *dirty bit* levé (test ReadAfterWrite) et les lectures sur des données écrites par un DMA. Concernant le test ReadAfterWrite, on observe pour les trois processeurs des délais



(a) K1



(b) X1



(c) R-Car

FIGURE 4.11 – Résultats des tests de lecture pour le processeur ARM

d'accès à la mémoire assez similaires pour des tailles inférieures à 1024 kB, soit la moitié de la taille du cache L2. Ces délais sont assez proches de ce que l'on obtient pour des lectures avec un *cache-hit* (test ReadAfterRead). Or, pour des tailles plus grandes, on observe des comportements différents :

— Sur K1, le temps d'accès en lecture pour des données supérieures à 1024 kB semble linéaire, la pente est toutefois plus importante que celle obtenue pour les accès inférieurs à 1024 kB. Dans tous les cas, le test ReadAfterWrite semble plus rapide que des délais de lectures sur des données non présentes dans le cache (test Read).

— Sur X1, on observe une pente très importante pour des accès dont la taille varie entre 1024 et 2048 kB (soit la taille du cache L2), ce phénomène est très probablement causé par le cache L2.

— Sur R-Car H2 le résultat du temps de lecture après écriture est totalement surprenant et ne correspond pas du tout à ce que nous attendions. Au delà de 1024 kB, on observe une très forte pente jusqu'à un maximum local à 3000 kB. Ce comportement est probablement causé par un effet de cache, mais il n'est pas explicable avec le peu d'information dont nous disposons à partir de la *datasheet* de l'architecture.

À propos du test de lecture sur des données écrites par un DMA (test ReadAfterTransferFromGPU), on observe dans les deux cas (Tegra K1 et X1) un comportement assez proche d'une lecture sur des données non présentes dans le cache (test Read). Cela parait tout à fait logique puisque par définition les données écrites pas DMA ne sont pas écrites sur le cache du processeur ARM. Sur Tegra K1 on remarque tout même une particularité : une cassure pour une taille égale à 1024 kB.

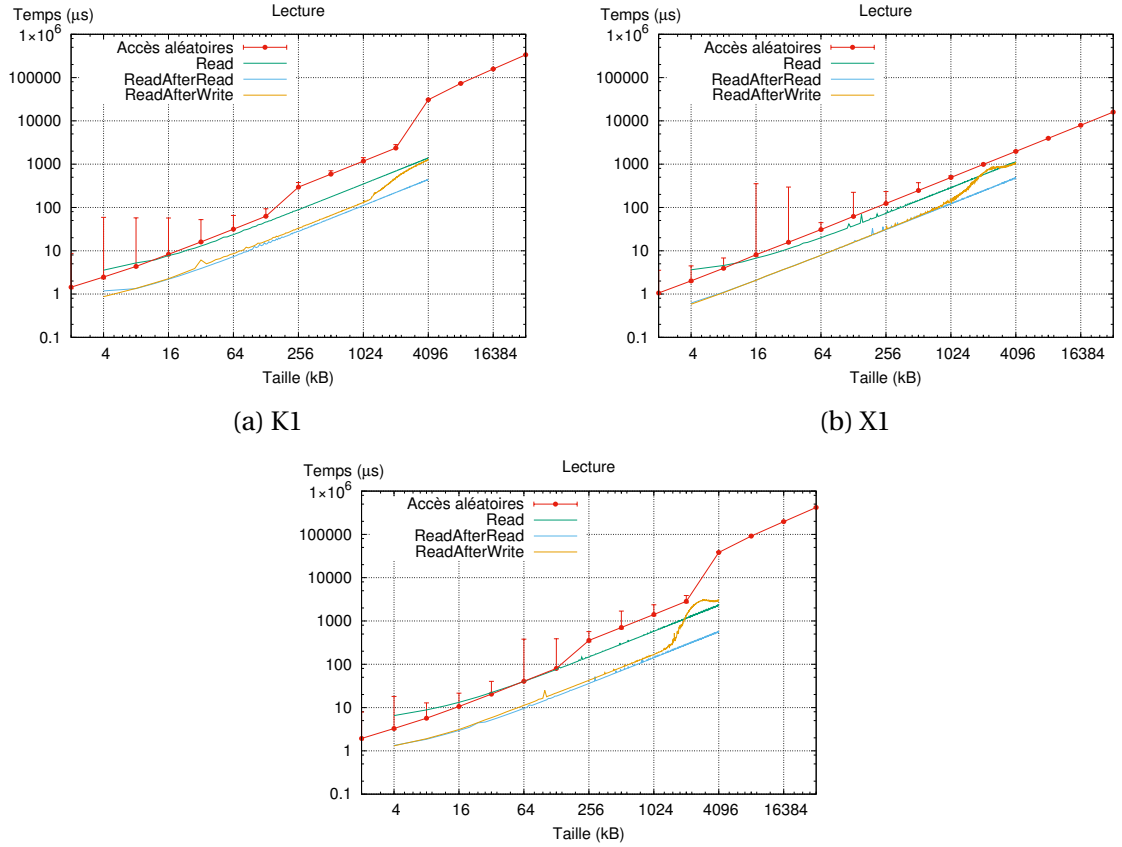
Nous observons donc des comportements qui ne sont pas exactement tels que nous les attendions et qui ne sont pas décrits dans les *datasheets* correspondantes. Il s'agit là encore d'un nouveau point fort de notre approche, nous sommes capables d'identifier des phénomènes non décrits par la documentation et de les modéliser. Ainsi, les délais d'accès à la mémoire peuvent être modélisés par des tables de correspondance pour lesquels on est capable de déterminer un délai mémoire pour une configuration et une taille données. Cependant, il n'est pas forcément optimal de stocker l'ensemble des délais pour l'ensemble des tailles d'accès, de plus la table de correspondance ne permet pas de déterminer un délai pour une taille qui n'est pas dans la table (par exemple si la taille est supérieure à la taille maximale du test). Pour pallier ces problèmes, on se propose, quand cela est possible, d'effectuer une interpolation automatique de la table de correspondance par des fonctions polynomiales par morceaux dont le degré dépend des cas. De cette manière, en connaissant le nombre de lectures d'un *kernel*, on est capable de prédire les délais d'accès en lecture tout simplement en utilisant la modélisation ainsi obtenue. L'approche est discutée plus en détail dans le chapitre suivant.

Lecture non-contiguë

En plus des tests de lecture traditionnels, nous proposons d'aller un peu plus loin dans la caractérisation de ces architectures. Ainsi, nous avons étudié jusqu'à maintenant des accès contigus en lecture, ce choix est justifié par le fait que nous cherchons à obtenir la bande passante la plus élevée et à être représentatif du domaine du traitement d'images. Nous proposons ici une extension du test de lecture pour la caractérisation de l'accès à des données de manière non-contiguë.

Le test fonctionne de la même manière que le test de lecture standard, mais accède aux données en utilisant l'adressage par *bit-reversed*. Ainsi, au lieu d'incrémenter normalement le pointeur, l'adresse de la donnée à lire est calculée en l'incrémentant par une

opération d'addition à propagation inversée de la retenue. Si la plage totale d'adresses est de 2^n alors l'incrément est 2^{n-1}. Par exemple pour un compteur sur un octet (c'est-à-dire 256 adresses), les données seront adressées dans l'ordre suivant : 0 (00000000_b), 128 (10000000_b), 64 (01000000_b), 192 (11000000_b), etc. Remarquons que c'est ce type d'adressage qui est utilisé dans les algorithmes de calcul de FFT [GOLD et collab., 1969]. De plus, on se propose de travailler sur une grande quantité de données (soit au moins dix fois la taille du cache L2) ; dans notre cas nous avons choisi 32 MB. Remarquons que pour la mesure des délais, nous effectuons une moyenne sur 1024 mesures successives.



(a) K1

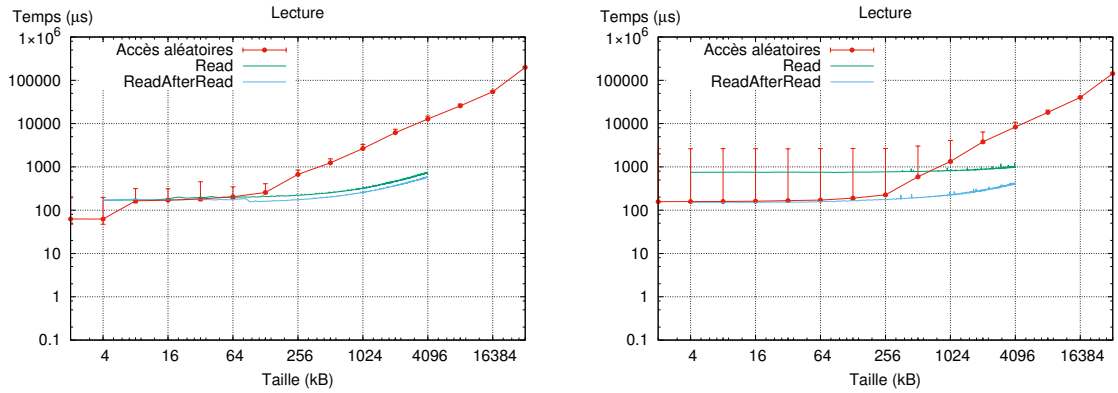
(b) X1

(c) R-Car H2

FIGURE 4.12 – Résultats du test de lecture non contiguë sur des *int32* pour le CPU ARM

Ce test met le cache à rude épreuve : on accède à chaque fois à une nouvelle page mémoire, on doit donc se trouver avec très fort taux de *cache-miss*. On devrait également voir apparaître des cassures lorsque la taille des données est supérieure à la taille du cache L1 (soit 32 kB) et du cache L2 (soit 2 MB). Les résultats sont donnés en figure 4.12 pour le processeur ARM du K1, X1 et R-Car H2. À titre de comparaison, nous affichons également les résultats des tests de lecture Read, ReadAfterRead et ReadAfterWrite. Remarquons que les graphiques utilisent une échelle logarithmique. Tout d'abord, on observe bien deux cassures pour le K1 et R-Car H2 (ARM A15 pour les deux SoCs). La première cassure se situe à 128 kB, alors que l'on s'attendait plus à l'observer autour de 32 kB. Par contre, la seconde cassure se situe à 2048 kB, soit la taille du cache L2. Pour la Tegra X1, contrairement à nos attentes, on observe un comportement linéaire pour toutes les tailles testées. De plus, les temps d'accès sont environ 20 fois plus rapides que sur K1 dans la zone supérieure à 2 MB. Il faut savoir que la bande passante mémoire de ce SoC est plus rapide que les deux autres, et que la mémoire est d'une génération plus récente (DDR4 contre DDR3). Ainsi, ces ré-

sultats nous montrent que la mémoire DDR4 du X1 est beaucoup plus performante que la mémoire DDR3 du K1 ou du R-Car H2 pour des accès aléatoires, non contigus. Nous supposons que ce comportement linéaire est dû au fait que les performances sont limitées par les capacités de calcul et non par la mémoire, on se situe dans la limite entre la zone de forte I_A et de faible I_A. En effet, ce test implique d'effectuer quelques opérations supplémentaires pour calculer l'adresse de la donnée à charger.



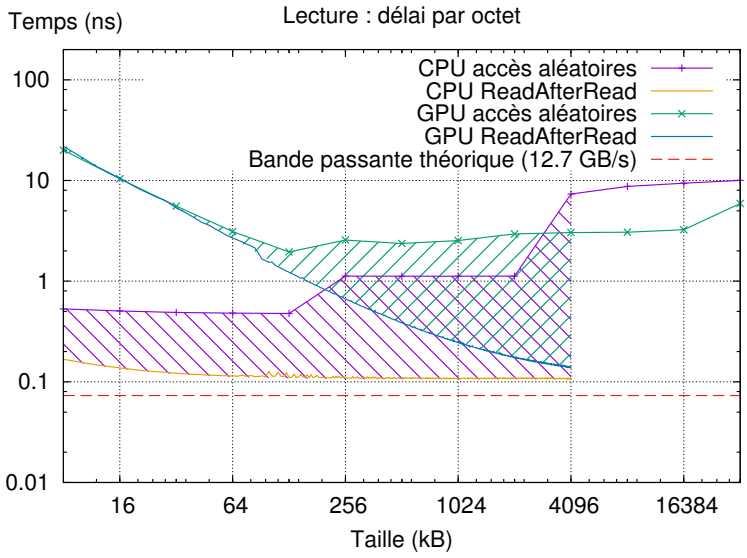
(a) K1 (b) X1

FIGURE 4.13 – Résultats du test de lecture non contiguë sur des *int32* pour le GPU

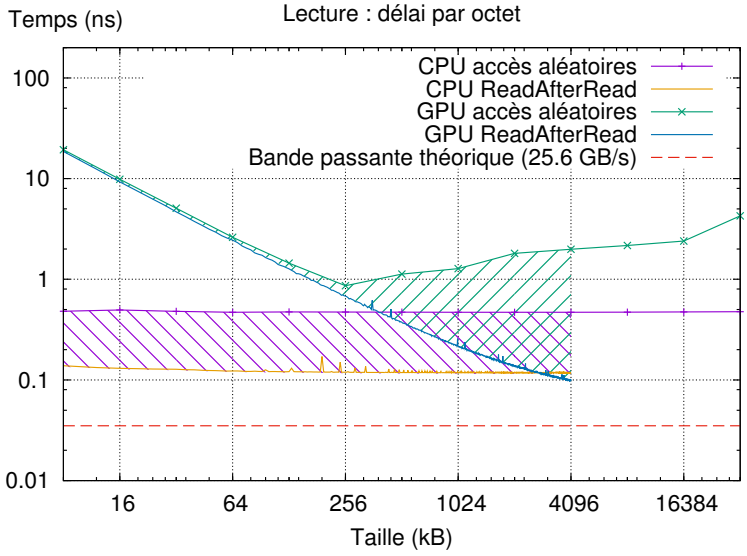
Nous donnons les résultats de ce test obtenus sur le GPU du Tegra K1 et Tegra X1 en figure 4.13, en comparant avec les résultats du test ReadAfterRead. On peut observer dans un premier temps une cassure au niveau du cache L2 de chaque GPU, 128 kB pour K1 et 256 kB pour X1. Pour des lectures inférieures à la taille du cache L2, on remarque que le cache L2 joue parfaitement sont rôle : les délais sont équivalents à une lecture effectuée de manière contiguë (ReadAfterRead). Après cette cassure, on peut alors se rendre compte de l'énorme coût que représente les accès non contigus à la mémoire pour le GPU, quelque soit l'architecture (Kepler ou Maxwell). Ainsi, pour le Tegra K1, on observe que l'accès à 1024 kB de données est environ 10 fois plus long si elles sont lues de manière non contiguë. Ce test souligne donc l'importance de la lecture des données de manière contiguë sur GPU et nous permet d'estimer l'impact d'accès à des données de manière non contiguë. Il s'agit là du pire cas possible pour des délais de lecture sur GPU.

À partir de ces résultats, nous sommes capables d'estimer le délai de lecture par octet lu dans le cas le plus favorable (ReadAfterRead) et dans le pire cas (lecture non contiguë). Ainsi, dans le figure 4.14, nous affichons les délais de lecture par octet lu en fonction de la taille des données pour le meilleur et le pire cas. Les résultats nous permettent alors d'affirmer que le délai de lecture par octet sera forcément dans la zone hachurée du processeur correspondant quel que soit le contexte du cache, le taux de lecture non contiguë, etc.

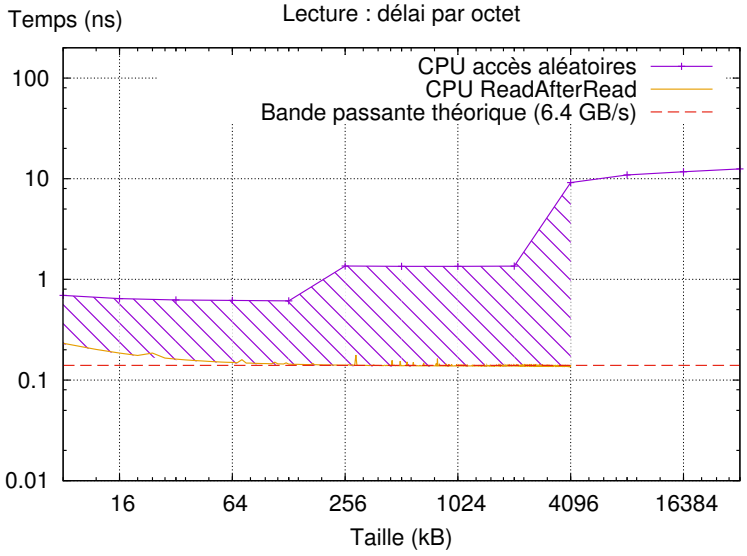
Les délais de lecture par octet sont également confrontés à la bande passante théorique de chaque SoC, correspondant à la limite théorique de la vitesse d'accès à la mémoire. On remarque alors que la lecture sur Tegra X1 est assez éloignée de la limite théorique, probablement limitée par la bande passante du cache L1. On constate d'ailleurs que les bornes inférieures des délais de lectures sont assez similaires sur les trois plateformes. Sur R-Car, on atteint la limite théorique de 6.4 GB/s.



(a) K1



(b) X1



(c) R-Car H2

FIGURE 4.14 – Temps de lecture par octet sur des *int32*

Résultats des tests d'écriture

Intéressons-nous à présent aux résultats obtenus sur les tests d'écriture. Nous proposons trois tests permettant de caractériser les accès en écriture dans trois conditions différentes :

— Test **Write** : il s'agit de la première écriture après l'allocation de données, celles-ci ne sont pas présentes dans le cache, elles sont donc écrites pour la première fois, on devrait observer un comportement linéaire. En théorie il devrait correspondre au pire cas, c'est-à-dire la configuration où les délais d'écriture sont les plus longs. Ce test simule le comportement d'un *cache-miss*.

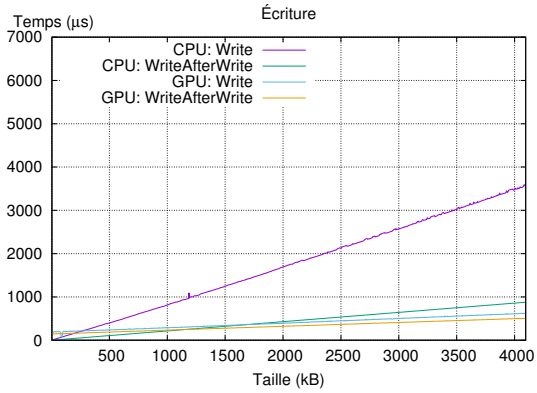
— Test **WriteAfterWrite** : il s'agit d'aller écrire sur des données qui ont déjà été écrites précédemment, les données sont donc très probablement présentes dans le cache, on devrait observer un temps d'écriture inférieur à la première écriture. En théorie il s'agit du cas le plus favorable, là où la latence des accès est la plus faible. Ce test simule le comportement d'un *cache-hit*.

— Test **WriteAfterRead** : il s'agit d'effectuer une écriture sur des données qui ont été lues précédemment, les données sont donc très probablement présentes dans le cache, on devrait donc observer un comportement assez similaire au test WriteAfterWrite.

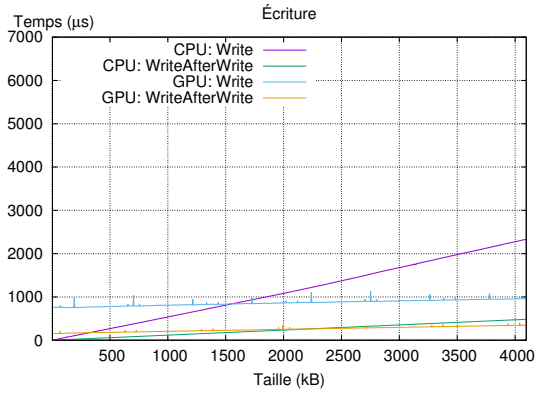
Les résultats des tests d'écriture sur des *int32* sont donnés en figure 4.15 pour les différents SoCs. Comme nous l'avions prévu, on peut observer sur le processeur ARM que l'écriture en cas de *cache-hit* (test WriteAfterWrite) est beaucoup plus rapide qu'en cas de *cache-miss* (test Write). On peut remarquer également que ces tests mettent en avant des comportements parfaitement linéaires, excepté pour le R-Car H2 où l'on observe des petites cassures pour des tailles de 2400 kB et 3700 kB. Les détails à propos des tests d'écriture sur le processeur ARM seront discutés ultérieurement. Sur GPU, on obtient des comportements similaires aux tests de lecture, c'est-à-dire une latence de lancement plus forte sur le premier test d'écriture que sur le second. Cette différence est d'ailleurs tout aussi marquée sur le Tegra X1. Globalement, comme pour la lecture, on remarque que la bande passante du GPU en écriture est plus rapide que celle du CPU sur Tegra K1 et Tegra X1.

Intéressons-nous maintenant à l'impact du type de données sur les délais de lecture. Conformément aux résultats obtenus sur les tests de lecture on pensait obtenir des comportements différents pour les différents types de données sur GPU et des comportements identiques sur le processeur ARM. Les résultats sur GPU sont données dans la figure 4.16. Contrairement à nos attentes, les écritures sur des *int16* et des *int32* ont exactement le même comportement sur les GPUs de la Tegra K1 et X1. De plus, le rapport entre la pente des écritures sur *int8* et la pente des écritures sur des *int16* est d'environ 1,7, alors que l'on s'attendait plus à un rapport égal à 2.

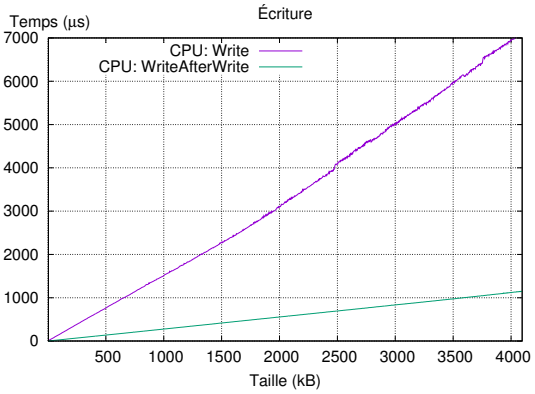
Sur le processeur ARM, nous avons vu que le type de données n'a pas d'impact sur les délais de lecture grâce à la vectorisation du compilateur, on peut donc s'attendre à un comportement similaire pour l'écriture. Nous donnons dans la figure 4.17 les résultats du test WriteAfterWrite pour différents types de données obtenus pour le processeur ARM A15 (Tegra K1) et A57 (Tegra X1). Sur l'ARM A15, comme nous nous y attendions, les accès sur les différents types de données sont parfaitement identiques, les trois droites sont confondues. Or, sur l'ARM A57 (Tegra X1) on remarque que l'écriture sur des *int32* se fait plus rapidement, alors que les écritures sur des *int16* et *int8* ont le même comportement. Il faut savoir que, sur Tegra X1, les délais d'écriture obtenus sur le test Write sont identiques pour les différents types de données testés. Le phénomène observé sur



(a) K1

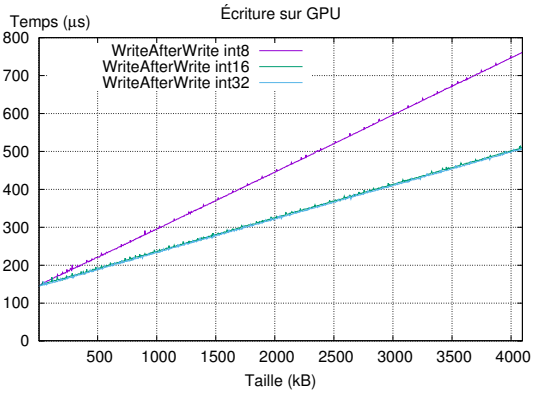


(b) X1

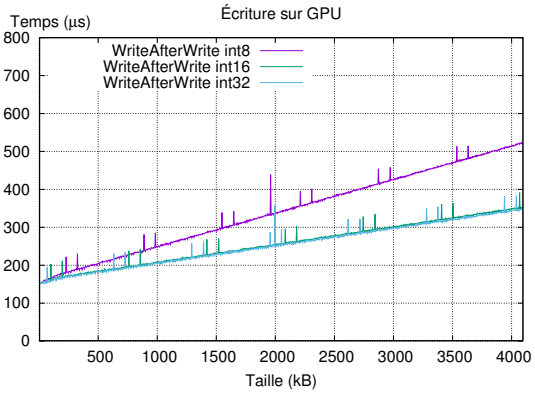


(c) R-Car H2

FIGURE 4.15 – Résultats des tests d'écriture sur des *int32*



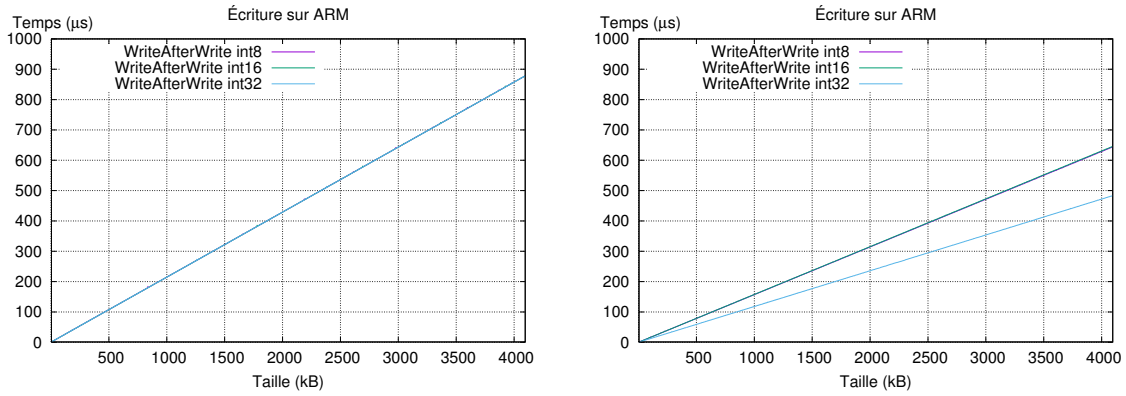
(a) K1



(b) X1

FIGURE 4.16 – Résultats du test WriteAfterWrite sur GPU pour différents types de données

le test WriteAfterWrite est donc très probablement causé par le cache de l'A57. Dans la figure 4.18, nous montrons les délais obtenus sur le CPU sur Tegra X1 sur le test WriteAfterRead pour différents types de données. Pour ce test on s'aperçoit alors que les écritures sur des *int32* et des *int16* ont le même comportement, mais que l'écriture sur des *int8* est plus longue, alors que pour le test WriteAfterWrite les comportements sur *int16* et *int8* sont identiques. Nous supposons que ce phénomène est également causé par la mémoire cache.



(a) K1 (b) X1

FIGURE 4.17 – Résultats du test WriteAfterWrite sur CPU ARM du Tegra K1 et X1 pour différents types de données

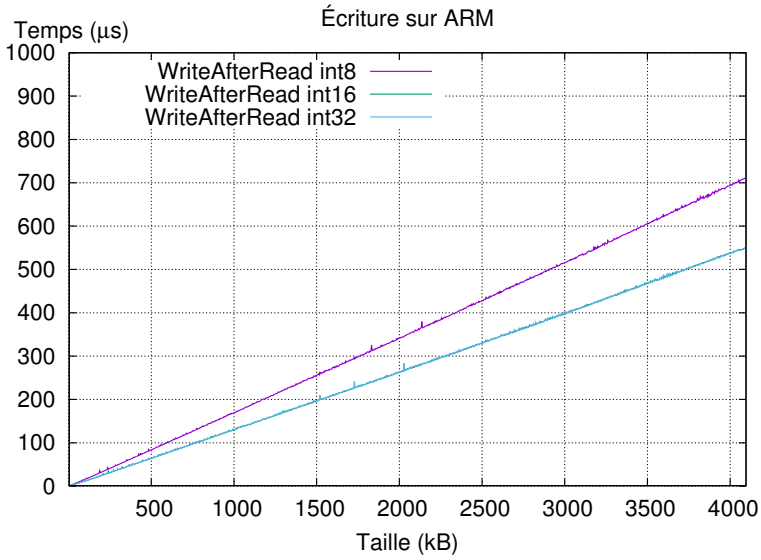
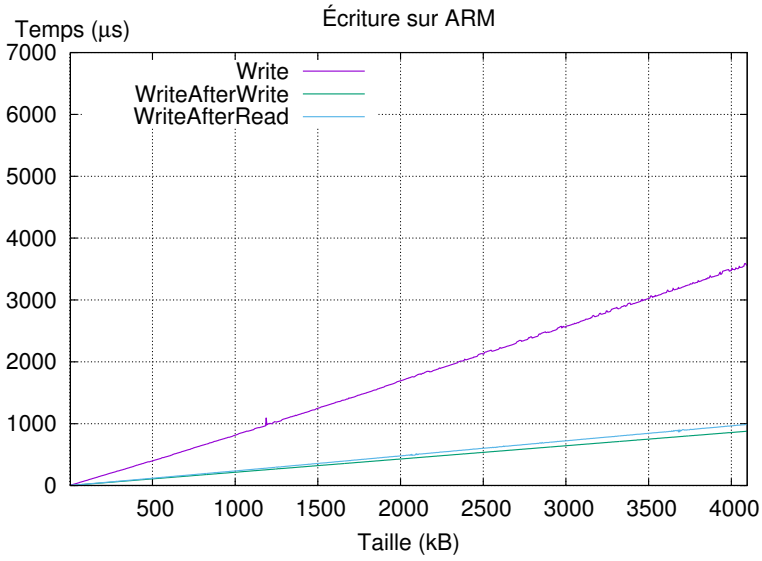
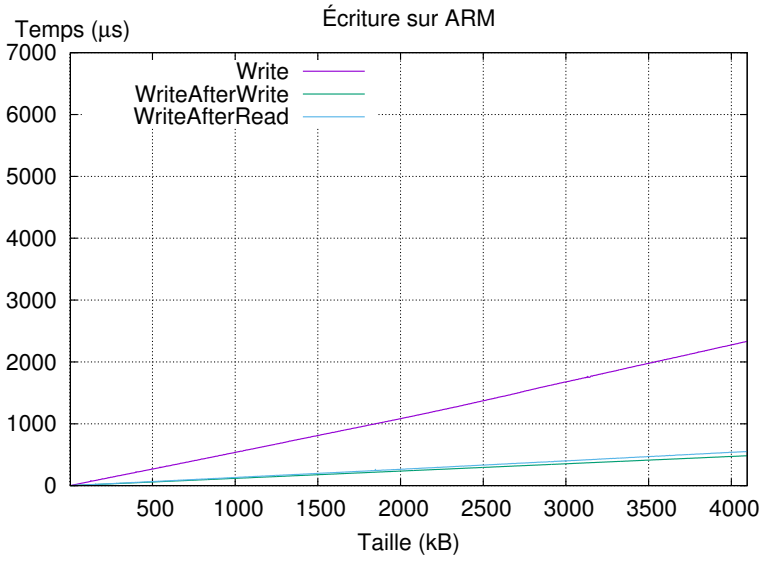


FIGURE 4.18 – Résultats du test WriteAfterRead sur le CPU ARM du Tegra X1 pour différents types de données

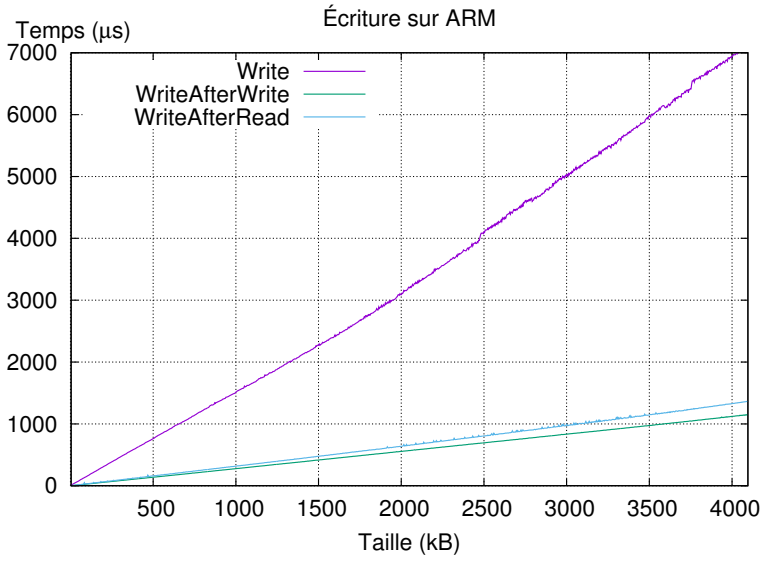
Enfin, nous proposons d'étudier les comportements des écritures sur les processeurs ARM. Comme nous l'avons vu jusqu'à maintenant, les délais des écritures par le processeur ARM sont plus lents que les écritures effectuées par le GPU, ils sont très sensibles au contexte du cache, mais ils ne sont pas sensibles au type de données grâce à la vectorisation (excepté pour l'ARM A57 du Tegra X1). Ainsi, les résultats des tests d'écriture sur des *int32* obtenus sur les CPU des SoCs Tegra K1, Tegra X1 et R-Car H2 sont donnés en figure 4.19. De manière générale, on peut remarquer que la pente des accès en écriture est plus



(a) K1



(b) X1



(c) R-Car

FIGURE 4.19 – Résultats des tests d'écriture sur des *int32* pour le processeur ARM

élevée sur R-Car H2 par rapport à celle obtenue sur K1 et X1, et que les délais d'écriture sur K1 sont plus élevés sur X1. Ce qui est tout à fait en accord avec les bandes passantes mémoires théoriques de ces architectures.

Résultats des tests de copie

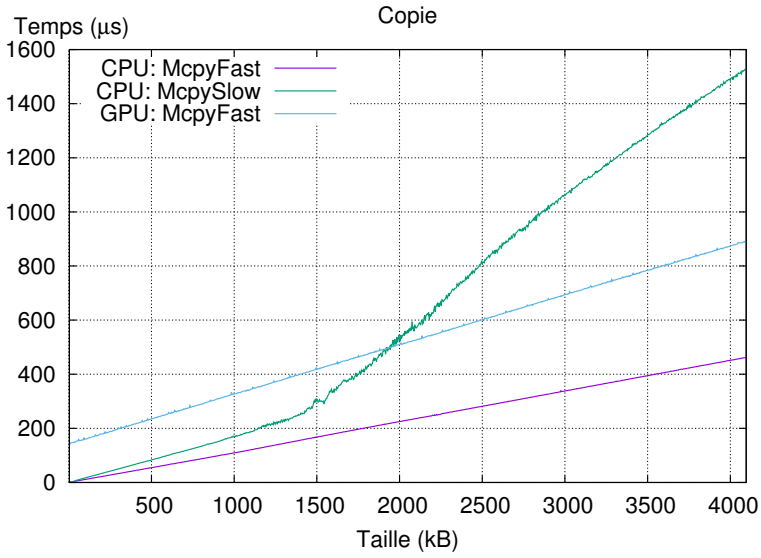
Maintenant que nous avons étudié les comportements des lectures et des écritures, nous proposons de nous intéresser à un test mêlant ces deux opérations, c'est-à-dire une copie entre deux zones mémoires. Remarquons que nous n'adressons pas ici les copies de données impliquant des unités DMAs dédiées, ce type de copie étant abordé dans les tests de transfert. Un programmeur confirmé sait qu'il existe une fonction C de bibliothèque standard permettant d'effectuer cette tâche, cependant nous avons choisi d'écrire nous même notre propre fonction de copie et de ne pas utiliser *memcpy*. De cette manière nous avons une plus grande maîtrise du test, nous pouvons être plus proche d'une application réelle, et cela évite d'introduire des effets parasites pouvant provenir de la manière dont la fonction *memcpy* a été implémentée. Remarquons qu'un test sur la fonction *memcpy* peut être tout aussi pertinent, mais il ne sera pas abordé ici. Nous proposons de présenter les résultats en deux temps :

— Test **McpyFast** : nous caractérisons ici le meilleur cas, c'est-à-dire un accès aux données qui sont probablement dans le cache. Pour reproduire ce cas, nous effectuons tout simplement deux copies consécutives et mesurons le temps d'exécution de la seconde copie. Dans ces conditions, les temps de copie doivent être linéaires.

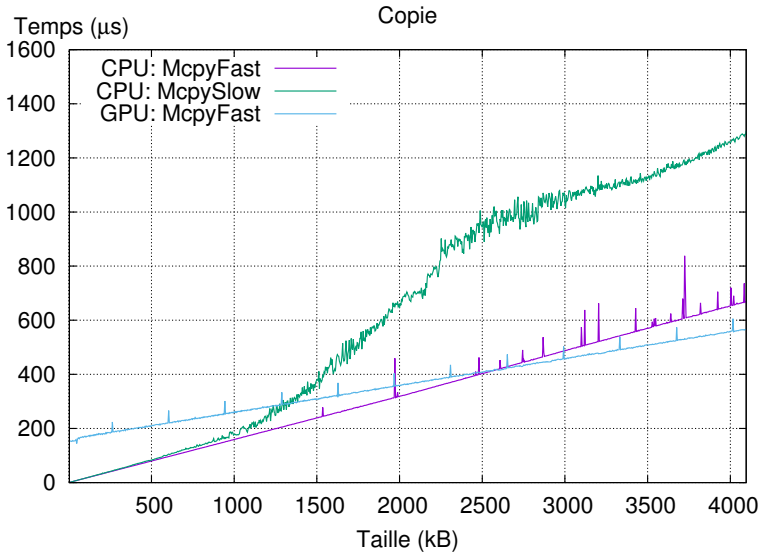
— Test **McpySlow** : nous essayons ici de nous rapprocher des pires cas. Ainsi les données d'entrée et de sortie sont préalablement écrites pour provoquer la levée d'un *dirty bit*. On doit donc logiquement s'attendre à des délais de copie beaucoup plus importants que dans le cas précédent, proches des comportements obtenus pour le test ReadAfterWrite.

Nous donnons les résultats obtenus sur ces deux tests en figure 4.20 pour Tegra K1, Tegra X1 et R-Car H2. Remarquons que les résultats sur R-Car H2 sont données à une échelle différente, à cause de délais bien plus grands que sur K1 et X1. À la lumière des tests précédents de lecture et d'écriture sur GPU, nous pensions obtenir des comportements identiques entres les deux tests McpyFast et McpySlow. Cette supposition est entièrement vérifiée par les résultats obtenus : il n'y a aucune différence visible entre les deux courbes, donc nous présentons dans la figure 4.20 la courbe commune. La vitesse de copie sur le GPU du Tegra X1 est plus rapide que sur celui du K1, ce qui en accord avec les bandes passantes mémoire théorique des deux SoCs. De plus, les copies effectuées par le GPU sont plus rapide que celles effectuées par le CPU ARM sur K1 et X1. À propos de l'ARM, on remarque que le test McpyFast est plus rapide sur Tegra K1 que sur Tegra X1 alors que la bande passante de la mémoire centrale de ce dernier est théoriquement plus rapide. Ce phénomène est très probablement causé par un management plus efficace du cache sur K1. Sur R-Car H2 on observe un délai de copie plus important que sur les deux autres plateformes, conformément aux bandes passantes théoriques. Les performances du test McpySlow sur ARM sont assez similaires aux performances des tests ReadAfterWrite. Cela est tout à fait logique, puisque, comme nous l'avons présenté, le test McpySlow consiste à copier des données qui sont déjà écrites (*dirty bit* levé).

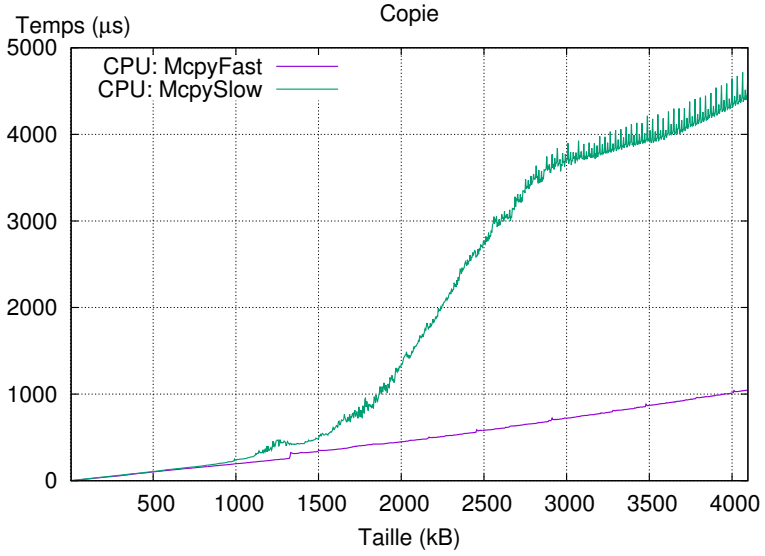
Ainsi, nous donnons dans la figure 4.21 les résultats des tests de copie et les délais mesurés pour les tests WriteAfterWrite, ReadAfterRead et ReadAfterWrite sur le processeur ARM. Remarquons que là aussi, les résultats sur R-Car H2 ont une échelle différente. Comme discuté précédemment, on observe des comportements très proches entre les



(a) K1

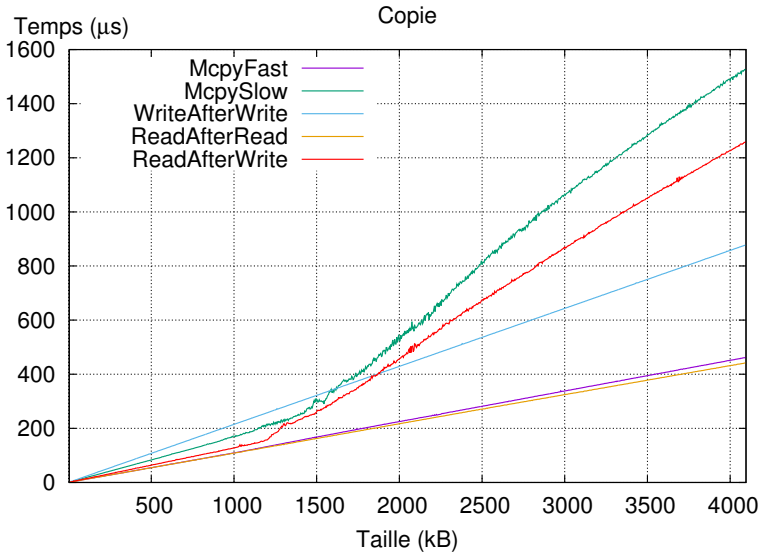


(b) X1

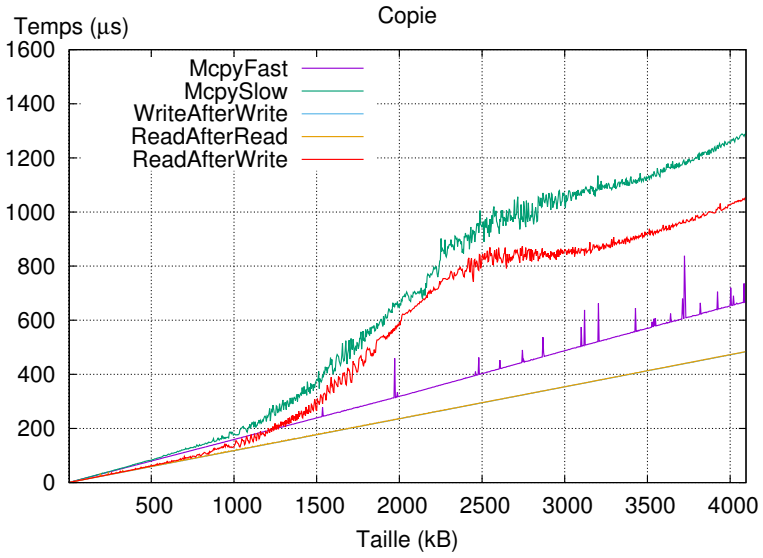


(c) R-Car H2

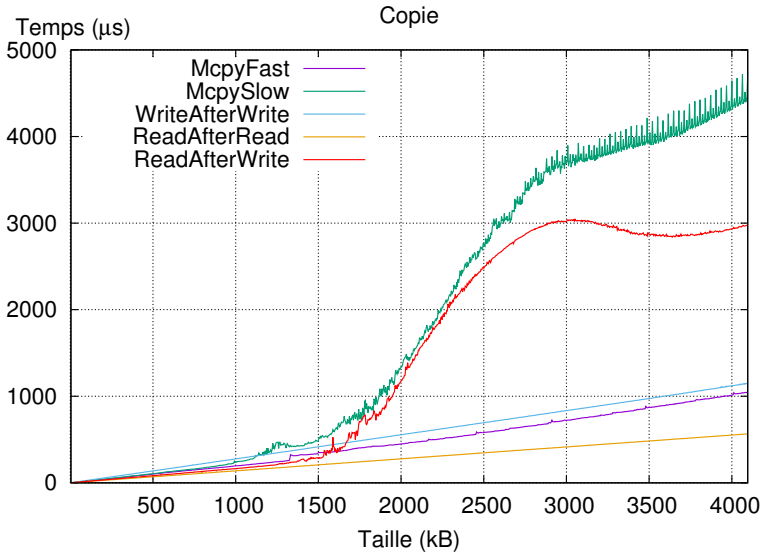
FIGURE 4.20 – Résultats du test Memcopy



(a) K1



(b) X1 (les courbes WriteAfterWrite et ReadAfterRead sont confondues)



(c) R-Car H2

FIGURE 4.21 – Résultats du test Memcopy

tests McpySlow et ReadAfterWrite surtout, sur Tegra K1 et Tegra X1 où les écarts entre les deux courbes sont très faibles. Sur R-Car H2, les comportements entre les deux tests diffèrent pour des tailles supérieures à 2500 kB. Concernant les résultats sur le test McpySlow, on remarque que les délais mesurés sont très proches des résultats obtenus sur le test ReadAfterRead. Ainsi, sur Tegra K1 l'écart entre ces deux tests est de seulement quelques dizaines de microsecondes et le délai de copie est bien inférieur à celui de l'écriture (test WriteAfterWrite). Sur Tegra X1, on peut remarquer tout d'abord que les courbes illustrant les délais les tests ReadAfterRead et WriteAfterWrite sont confondues, ce qui peut s'expliquer par le fait que dans les deux cas les données se trouvent dans le cache. Ensuite, on peut observer logiquement un temps de copie McpyFast supérieur à la lecture (ReadAfterRead) et à l'écriture (WriteAfterWrite). De plus, le délai de copie est inférieur à la somme des temps de lecture et d'écriture. Cela s'explique tout simplement par le fait que la lecture et l'écriture sur ARM sont gérées par deux unités différentes et donc que ces opérations peuvent, en théorie, se faire en parallèle.

Il était déjà difficile de trouver des explications sur les comportements des opérations de lecture et écriture à partir des *datasheets* des fondeurs, mais cela devient encore plus complexe avec la copie de données qui met en œuvre ces deux opérations. Ainsi, cela souligne encore une fois l'intérêt de notre approche : sans connaître tous les détails d'une architecture, nos vecteurs de test sont capables de déterminer les comportements des accès mémoire et d'identifier des phénomènes non décrits par les fondeurs.

Résultats du test de transfert

Pour une architecture hétérogène, le transfert consiste à transférer des données d'un processeur à un autre (par exemple CPU vers GPU, CPU vers DSP, etc.). Dans ce contexte, nous ne considérons pas l'échange de données entre deux unités d'exécution d'un même processeur qui partagent un même niveau de cache, dans ce cas les données passent par ce niveau de cache et non par des unités DMA. En fonction du type d'architecture, on peut distinguer trois types de transfert :

— Lorsque la mémoire est commune à l'ensemble des processeurs d'un même SoC, il est possible d'allouer une zone mémoire qui soit accessible par plusieurs processeurs. En général les architectures proposant ce genre de fonctionnalité ont également un mécanisme de protection pour éviter des conflits liés à des accès concurrents entre différents processeurs. Ainsi, l'OS laisse la main à l'un des processeurs, le transfert consiste alors à s'assurer de la cohérence des données entre la zone mémoire et le cache du processeur actif, puis à autoriser l'accès depuis un autre processeur. Dans ce cas, le temps de transfert correspond au temps de vidage de cache du processeur actif.

— Dans le cas où la mémoire est commune à l'ensemble des processeurs, l'OS peut définir une zone mémoire qui soit propre à chaque processeur, permettant ainsi d'éviter des conflits. Le transfert au sein d'une même mémoire physique consiste alors à copier les données d'une zone mémoire à une autre. C'est le comportement par défaut que l'on a sur Tegra K1 ou Tegra X1 où la mémoire est commune au CPU, GPU et ISP. Remarquons qu'avant d'effectuer le transfert entre les deux zones mémoires, l'OS s'assure de la cohérence des données entre le cache du processeur et la zone mémoire à transférer.

— Si chaque processeur dispose de sa propre mémoire, le transfert de données s'effectue alors d'une mémoire à une autre en utilisant un bus de données. Par exemple dans le cas d'un PC le transfert de données entre la mémoire du CPU vers celle

du GPU passe par le bus PCIe. On peut également citer l'exemple du SoC TDA2x. Il s'agit théoriquement du cas où les transferts sont les plus longs. Remarquons que cela s'applique également aux clusters liés par exemple par un bus Ethernet, comme la DrivePX qui est composée de deux Tegra X1 liés par Ethernet.

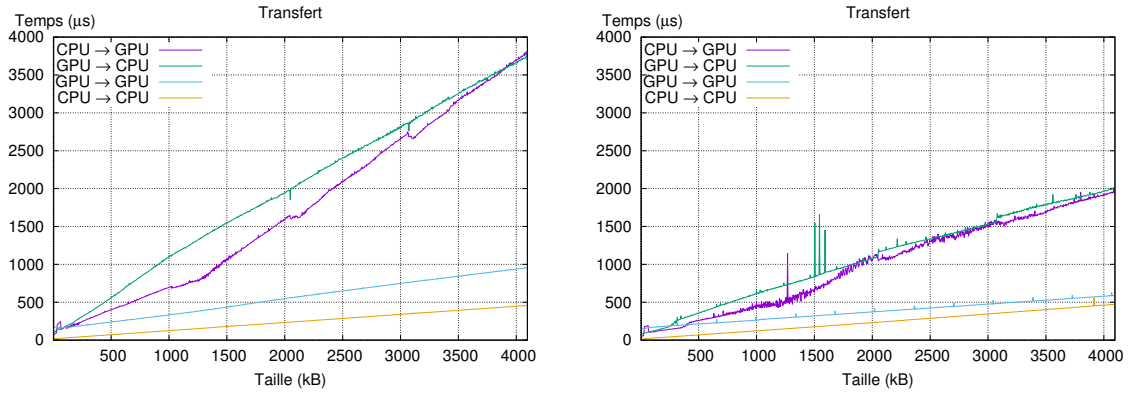
Dans le cas de l'API CUDA, il est possible de définir des zones mémoire spécifiques qui peuvent simplifier ou accélérer les transferts entre CPU et GPU. Ainsi, il existe depuis la version 4.0 l'adressage virtuel unifié, ou *Unified Virtual Addressing* (UVA), qui définit un adressage mémoire virtuel comprenant la mémoire du CPU et du GPU. Cela permet donc de passer simplement des pointeurs de données du CPU au GPU et inversement, quel que soit l'endroit où se trouve les données. L'UVA permet l'utilisation des accès « *Zero-Copy* », c'est-à-dire que le GPU accède directement aux données présentent sur la mémoire du CPU, en passant par le bus PCIe. Pour que les accès *Zero-Copy* fonctionnent, il faut cependant veiller à allouer la zone mémoire en question sur le CPU en mode *pinned memory*, par opposition à la mémoire paginée.

Depuis la version 6.0, l'API CUDA met également à disposition la mémoire managée (*managed memory*), qui permet à l'utilisateur de simplifier la gestion de transfert car il n'est plus obligé d'appeler explicitement des APIs de transfert CUDA comme *cudaMemcpy*, *cudaMemcpy2DAsync*, etc. L'implantation de cette *managed memory* est laissée entièrement à la charge des drivers. Ainsi, ils peuvent choisir la meilleure solution pour un système donné en utilisant de la mémoire paginée, de la *pinned memory*, ou en utilisant la même zone mémoire, avec ou sans UVA. La synchronisation est provoquée par l'utilisateur d'une manière implicite au moment du lancement d'un *kernel* GPU. Cette approche peut se révéler parfois plus coûteuse qu'une synchronisation explicite. Dans le cas du Tegra K1 et Tegra X1, vu que la mémoire est commune au CPU et le GPU, la *managed memory* utilise une zone mémoire où les différents processeurs peuvent accéder en lecture et écriture, ce qui permet donc de s'affranchir de la copie de données d'une zone mémoire à une autre. Remarquons que pour garantir la cohérence des données avec le cache du CPU dans cette configuration, l'intégralité du cache CPU est réécrit dans la mémoire avant le lancement d'un kernel sur GPU. On observe donc une augmentation significative de la latence d'exécution des kernels sur GPU.

Dans le cas du SoC TDA2x, chaque processeur dispose de sa propre mémoire locale. TI fournit une interface hardware permettant à l'utilisateur le transfert de données entre les différents processeurs : l'*Enhanced Direct Memory Access* (EDMA). Cette interface hardware permet un contrôle particulièrement fin sur la gestion des transferts de données, elle permet notamment de gérer des transferts de données multidimensionnels (par exemple tableau 2D), des modes d'adressages différents entre la source et la destination, la gestion d'interruptions, le découpage d'un gros transfert en un ensemble de petits transferts, etc. Cependant elle est également assez complexe à utiliser, car elle demande à l'utilisateur la programmation explicites des différents registres de configurations de chaque unité EDMA. L'écosystème oblige l'utilisateur à programmer explicitement cette interface, sans aucune autre alternative comme pour le cas de l'API CUDA.

Enfin, intéressons-nous aux tests concernant le transfert de données entre différents processeurs. Dans une application réelle, le transfert de données d'un processeur p_a à un processeur p_b n'a de sens que si les données à transférer ont été préalablement écrites par p_a. Donc, dans le cadre de ce test, nous effectuons une écriture des données par le processeur concerné avant l'opération de transfert. En plus de permettre des transferts de données entre le CPU et le GPU via la ou les unités DMA, l'API CUDA permet d'effectuer des copies de données entre deux zones mémoires attachées au même processeur (par exemple transfert CPU vers CPU ou GPU vers GPU). Les résultats des délais de transfert

entre le CPU et le GPU sont donnés en figure 4.22 pour le Tegra K1 et le Tegra X1.



(a) K1 (b) X1

FIGURE 4.22 – Résultats du test de transfert

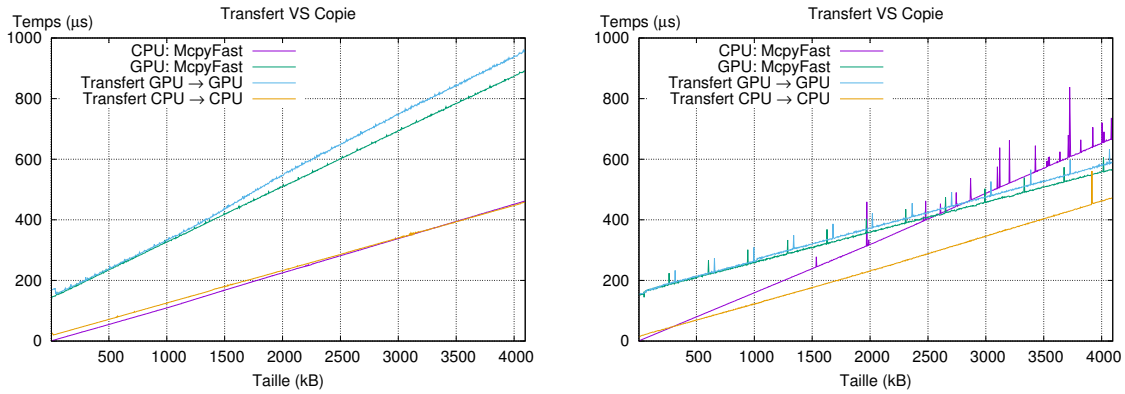
Tout d'abord, on observe que les transferts sont beaucoup plus rapides sur X1 que sur K1, conformément aux bandes passantes mémoire théoriques des deux SoCs. Pour les deux plateformes, les temps de transfert semblent beaucoup plus grands que les délais de copie, alors que le transfert correspond juste à une copie des données d'une zone mémoire à une autre, vu que le CPU et le GPU partagent physiquement la même mémoire. Tout d'abord, il est important de noter que les transferts sont gérés par les unités DMA et non par les processeurs. De plus, lors d'un transfert de données, l'OS doit s'assurer de la cohérence des données entre le cache du processeur source et la zone mémoire à transférer, cela peut expliquer ce surcoût. Sur X1, on observe d'ailleurs des comportements assez similaires entre le transfert CPU vers GPU et GPU vers CPU pour des tailles supérieures au cache L2 de l'ARM (2 MB).

Dans la figure 4.23, nous comparons les délais de transfert CPU vers CPU et GPU vers GPU par rapport aux délais obtenus pour le test McpyFast. L'utilisation des fonctionnalités de transfert entre deux zones mémoires du même processeur peut être avantageuse puisque, contrairement à une copie traditionnelle, les transferts utilisent les unités DMA et n'impliquent pas de charge de calcul pour le processeur en question. Cependant, comme nous l'avons montré dans le chapitre précédent, l'utilisation de transfert par DMA implique une latence plus grande ce qui est en contradiction par rapport au gain sur la charge du processeur. Dans le cas d'une vraie application temps-réel, il s'agit donc de trouver le bon compromis entre la contrainte de rythme et la contrainte de latence. Sur Tegra K1 les résultats montrent un comportement assez similaire entre le transfert et la copie. On remarque cependant un transfert légèrement plus lent sur GPU pour une taille de données supérieures à 1500 kB. Sur Tegra X1, la copie et le transfert GPU vers GPU sont similaires, cependant les résultats montrent que le transfert CPU vers CPU est beaucoup plus rapide qu'une copie entre deux zones mémoires du CPU.

4.4.3 Gestion de conflits

Principe de fonctionnement

Comme discuté dans la section 1.3.2, il existe différents types de parallélisme. Dans le cas du parallélisme simple, chaque *thread* peut s'exécuter indépendamment des autres *threads*, il n'y a pas de dépendance de données. Cependant, dans le cas où plusieurs



(a) K1 (b) X1

FIGURE 4.23 – Résultats du test de transfert vs copie

threads cherchent à écrire sur la même donnée il risque d'y avoir un conflit. Il existe différents mécanismes pour se prémunir de ces conflits, cependant ils impliquent un surcoût en termes de temps d'exécution. Le but de ces tests est de caractériser et modéliser ce surcoût pour être ensuite capable de prédire un temps d'exécution en fonction du nombre de conflits à gérer.

Dans le cas du CPU multicœur, il est généralement assez simple de s'affranchir de ce genre de problématique. Ainsi, le nombre de cœurs/*threads* étant assez faible par rapport à la quantité de données à traiter (notamment dans le cas du traitement d'images), il est possible de séparer les données entre les différents *threads*. Par exemple dans le cas d'une réduction parallèle sur un CPU 4 cœurs, les données sont séparées en 4, chacun des cœurs opérant sur 1/4 des données, puis on termine en réduisant le résultat de chacun des 4 cœurs pour obtenir le résultat final (en utilisant par exemple *InterlockedAdd* ou *__sync_fetch_and_add* ou tout simplement en effectuant la réduction sur un seul cœur). Dans ce cas, le surcoût de gestion d'un éventuel conflit est complètement négligeable par rapport au temps total de calcul. Cependant, dans le cas du GPU il est beaucoup plus difficile d'appliquer cette approche, puisque chaque *thread* traite seulement quelques pixels de l'image, soit un très grand nombres de *threads* et donc de conflits potentiels. L'API CUDA propose un mécanisme pour la gestion de conflit : les instructions atomiques, comme la fonction *AtomicAdd*. Cette fonction permet d'incrémenter une variable tout en s'assurant qu'elle n'est pas modifiée entre le moment de la lecture de sa valeur et de l'écriture du résultat via un mécanisme de protection hardware. Nous nous intéressons donc au GPU et plus particulièrement aux instructions atomiques dans le cadre de ce test.

Un des exemples le plus parlant d'algorithme impliquant des conflits est le calcul d'un histogramme : il s'agit en fait d'incrémenter une valeur en mémoire en fonction de la valeur de la donnée en cours de traitement. Puisque plusieurs *threads* peuvent chercher à incrémenter la même valeur au même moment, il est indispensable de mettre en place une gestion des conflits en cas d'exécution sur le GPU. Nous avons donc choisi de concevoir un test basé sur le calcul d'un histogramme, le principe consiste à étudier le temps d'exécution en fonction du nombre de conflits à gérer. Pour faire varier le nombre de conflits, nous proposons de faire varier l'entropie de SHANNON [2001] H du signal d'entrée (l'image à traiter par exemple). Ainsi, le signal d'entrée est généré aléatoirement sous la contrainte d'une entropie H fixe. De cette manière, un signal d'entrée ayant une faible entropie implique un nombre de conflits important, alors qu'un signal ayant une forte entropie implique un nombre de conflits plus faible. Le détail du fonctionnement du test

est donné dans l'algorithme 3.

Algorithme 3 : Caractérisation d'une gestion de conflit pour un signal d'entropie H et de taille $SIZE$.

Result : t : temps d'exécution

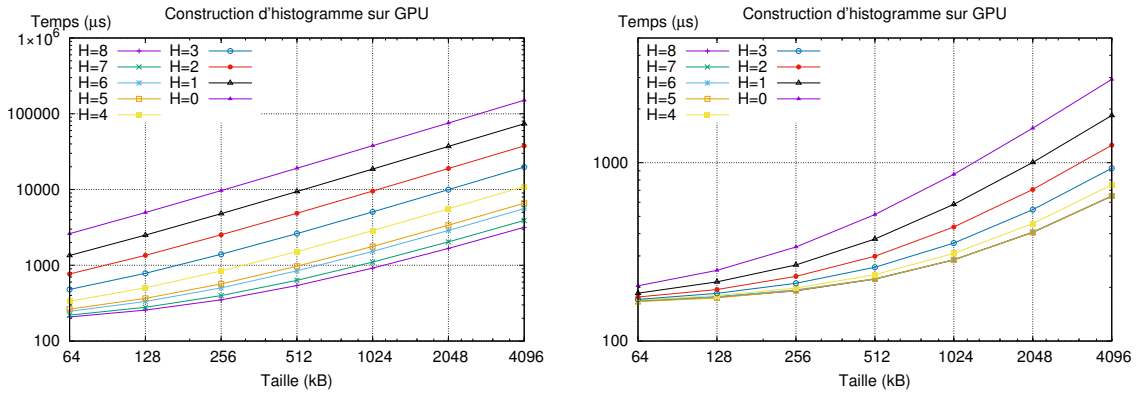
```
1 Alloc(Data, SIZE);
2 MOD = 2^H;
3 for j ∈ [1 : SIZE] do
4     Data[i] = rand()%MOD + (256 − MOD)/2;
5 StartChrono();
6 ComputeHistogram(Data, BIN_NUMBER);
7 StopChrono();
8 t= Chrono();
9 Free(Data);
```

Résultats

Pour les détails de l'implémentation du test, nous utilisons l'instruction *AtomicAdd* de l'API CUDA et les valeurs de l'histogramme sont stockées dans la *shared memory* du GPU. Nous choisissons de travailler avec des *int8* (la valeur de chaque élément peut donc varier entre 0 et 255) et un histogramme ayant 256 classes (une classe pour chaque valeur possible). Ainsi, un signal d'entrée avec une entropie nulle impliquera que tous les *threads* chercheront à incrémenter la même classe de l'histogramme et donc un conflit à chaque incrémentation. Dans cette configuration, on se place alors dans un cas similaire à de la réduction parallèle où l'on cherche par exemple à additionner l'ensemble des valeurs d'un vecteur d'entrée. Un signal d'entrée ayant une entropie égale à 8 impliquera une grande variété des valeurs, on aura donc peu de conflits à gérer.

Les résultats obtenus sur Tegra K1 et Tegra X1 sont donnés en figure 4.24. Nous donnons les délais d'exécutions pour différentes valeurs d'entropie H. Sur K1, on observe directement l'impact du nombre de conflits sur le temps d'exécution, une diminution de l'entropie implique dans tous les cas une augmentation du temps de calcul. Pour de grosse quantités de données, on remarque même qu'il y a un facteur 50 entre le temps de calcul de l'histogramme pour un signal d'entropie $H = 8$ et un signal d'entropie $H = 0$, où toutes les valeurs sont identiques. Sur Tegra X1, les résultats montrent que les gestions de conflits sont beaucoup plus performantes que sur K1. Ainsi, on observe que les temps d'exécution obtenus avec des signaux d'entropie allant de $H = 8$ à $H = 5$ sont identiques, les courbes sont confondues. De plus, on observe que pour une taille de 4 MB, l'écart entre le délai d'exécution d'un signal d'entropie $H = 8$ et un signal d'entropie $H = 0$ est seulement d'un facteur 5.

À partir de ces résultats, il devient facile en connaissant la taille et l'entropie d'un signal d'entrée de prédire l'impact de l'utilisation de l'instruction *AtomicAdd*. Remarquons qu'un algorithme qui effectue une réduction parallèle aura le même nombre de conflits que la construction d'un histogramme sur un signal ayant une entropie $H = 0$. Dans la réalité, les calculs se font sur des flux vidéo où chaque image est unique, il devient donc assez difficile de savoir quelle sera l'entropie de l'image d'entrée et donc de prédire le temps d'exécution d'un *kernel* utilisant le mécanisme d'instruction atomique pour la construction d'histogramme. Cependant, dans le cadre d'une application automobile, il est possible d'estimer a priori dans quel intervalle variera l'entropie d'une image d'entrée.



(a) K1 (b) X1

FIGURE 4.24 – Résultats de la gestion de conflit sur GPU

Ainsi, dans le cas d'une caméra à l'avant du véhicule, on peut supposer qu'une partie de l'image correspondra au ciel aura donc une valeur quasi uniforme ce qui fera donc diminuer la valeur de l'entropie. À titre d'exemple, nous avons mesuré une entropie moyenne de 7.34 avec un écart type 0.30 sur une séquence issue du très célèbre dataset KITTI [GEIGER et collab., 2013].

4.5 Vecteurs de test *mid-level*

4.5.1 Principe et mise en œuvre

La fonction de *mapping* définie dans l'équation 3.15 ne permet pas de définir un pipeline d'exécution, c'est-à-dire l'ordre dans lequel doivent s'exécuter les différents *kernels* indépendants. Ainsi, que faire lorsque deux *kernels* indépendants sont mappés sur la même unité d'exécution ? Doit-on les exécuter séquentiellement, c'est-à-dire l'un après l'autre ? Ou bien vaut-il mieux les exécuter de manière concurrente, c'est-à-dire en même temps ? Par exemple, le graphe de traitement de la figure 4.25, nous indique que k_2 et k_3 sont indépendants. Alors, s'ils sont mappés sur la même unité d'exécution, il est possible de définir plusieurs pipelines d'exécution. Sur le même exemple, la figure 4.26a montre le pipeline d'exécution obtenu si k_2 et k_3 sont mappés sur deux unités d'exécution différentes : dans ce cas il n'y a pas d'ambiguïté, un seul pipeline est possible. Or dans le cas où k_2 et k_3 sont mappés sur la même unité d'exécution il est possible de définir deux pipelines :

— k_2 et k_3 s'exécutent séquentiellement, comme illustré en figure 4.26b ;

— k_2 et k_3 s'exécutent de manière concurente, comme illustré en figure 4.26c.

Remarquons que chacune de ces solutions implique des valeurs de taux d'occupation et de latence différentes. Pour déterminer le meilleur pipeline à un *mapping* donné, il est donc nécessaire de connaître l'impact des exécutions concurrentes sur une unité d'exécution.

Les vecteurs de test *mid-level* peuvent répondre à toutes ces questions car ils caractérisent, entre autres, les exécutions concurrentes sur la même unité d'exécution, c'est-à-dire l'impact sur les performances lorsque l'on exécute plusieurs tâches indépendantes sur la même unité d'exécution en même temps. De manière plus concrète, il s'agit d'utiliser les vecteurs de test *low-level* précédemment définis, d'exécuter plusieurs de ces tests

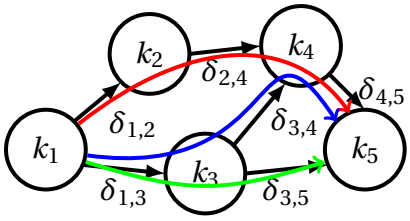
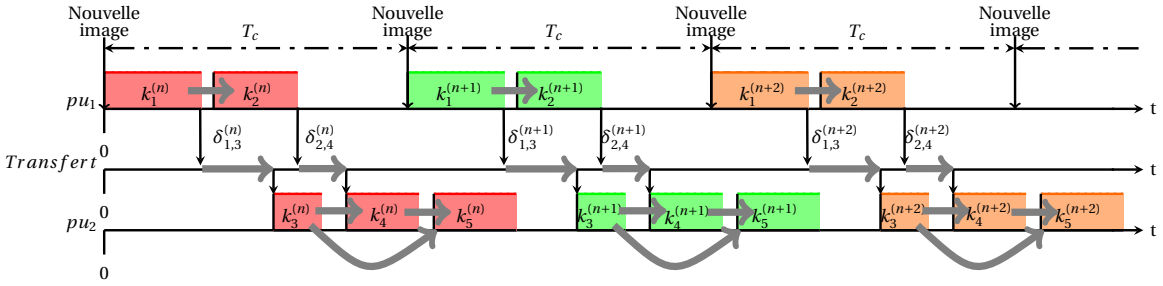
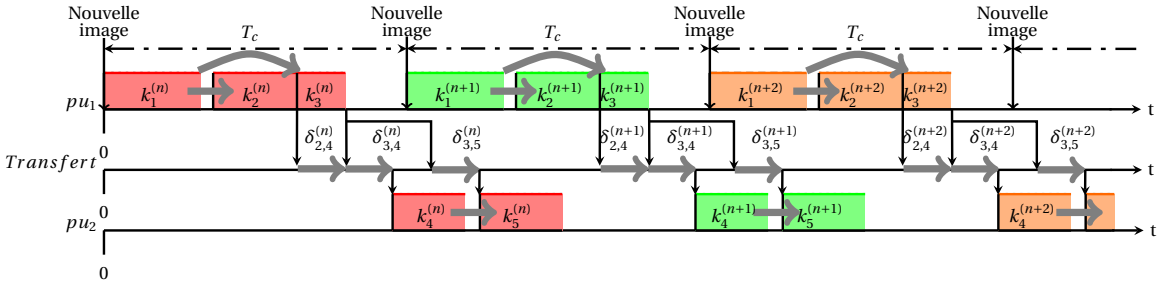


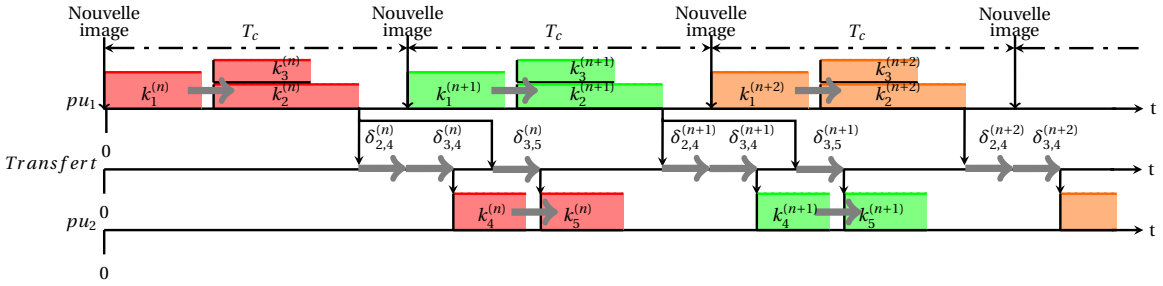
FIGURE 4.25 – Exemple de graphe de traitement composé de cinq *kernels*. Les *kernels* k_2 et k_3 sont indépendants, s'ils sont mappés sur la même unité d'exécution alors ils peuvent soit être exécutés séquentiellement, soit de manière concurrente.



(a) k_2 et k_3 sont mappés sur deux unités d'exécution différentes et exécutés en même temps.



(b) k_2 et k_3 sont mappés sur la même unité d'exécution et exécutés séquentiellement.



(c) k_2 et k_3 sont mappés sur la même unité d'exécution et de manière concurrente.

FIGURE 4.26 – Exemples de pipelines d'exécution pour le graphe de traitement donné en figure 4.25. Pour la même itération, $k_2^{(n)}$ et $k_3^{(n)}$ sont indépendants, ils peuvent donc s'exécuter de manière concurrente. Remarquons que le temps de traitement d'une image est plus grand que la période d'acquisition T_c, cependant la contrainte temps-réel de rythme est respectée grâce à la pipelinisation des *kernels* d'instances successives.

de manière concurrente et de comparer les résultats ainsi obtenus avec ceux obtenus dans des conditions non concurrentes. Nous cherchons à analyser deux configurations différentes :

— Des exécutions parallèles sur des unités d'exécution différentes appartenant à un même processeur : dans ce cas les différents tests partageront uniquement certains niveaux de cache donc seuls les délais d'accès à la mémoire devraient être impactés.

— Des exécutions concurrentes sur la même unité d'exécution : dans ce cas les différents tests partageront toutes les ressources (tous les niveaux de cache et les ressources de calcul), donc les temps d'exécution seront fortement impactés.

Dans le cas d'un *kernel* qui utilise une technique de parallélisation (implicite comme OpenMP ou explicite comme *pthread*), il va occuper les N unités d'exécution d'un même processeur. Si l'on accepte la décomposition de ce *kernel* en N sous-*kernel*s de poids similaire alors on retombe dans la configuration d'exécutions parallèles sur des unités d'exécution différentes appartenant au même processeur. Ceci nous permet alors d'évaluer aisément l'impact d'une parallélisation en utilisant les vecteurs de test *mid-level*.

Dans le cas du GPU, l'API CUDA ne permet pas de configurer manuellement la séparation des ressources d'exécution, il est seulement possible de demander l'exécution concurrente de plusieurs *kernel*s sur un GPU et de laisser l'API CUDA gérer cette exécution. À la vue de cette limitation, nous choisissons de représenter un GPU comme étant composé d'une unique unité d'exécution de calcul non séparable (même si dans la réalité un GPU est constitué d'un grand nombre de cœurs physiques).

Ainsi, à partir des résultats obtenus, il est possible de déterminer la politique la plus adaptée pour un processeur donné et ses unités d'exécution le composant. En cas d'exécutions concurrentes, il est également possible de prédire l'impact sur les temps d'exécution.

D'un point de vue pratique, pour lancer plusieurs instances du même test au même moment deux approches différentes ont été étudiées :

— La création de plusieurs *threads* à l'aide de l'API *pthread*.

— La création de plusieurs processus à l'aide des APIs Linux.

Un *thread* est différent d'un processus, dans le sens où plusieurs *threads* issus d'un même processus peuvent accéder à la même zone mémoire, alors que deux processus distincts ont chacun leurs propres zones mémoire. En ce qui nous concerne, nos tests n'ont pas besoin de communiquer lors de leur exécution et nous avons observé de manière empirique que les deux approches présentent des résultats identiques. Pour s'assurer de la concurrence, nous avons mis en place un système de barrière utilisant l'API *pthread* permettant de synchroniser les différents *threads* pour chaque point de mesure.

Dans le cas où plusieurs *threads* sont exécutés sur le même cœur physique, c'est l'ordonnanceur (*scheduler*) de l'OS qui se charge de déterminer la manière dont les *threads* doivent s'exécuter. Sur la majorité des OS, il existe des mécanismes de préemption permettant au *scheduler* d'interrompre l'exécution d'un *thread* pour en exécuter un autre.

En plus du mécanisme de préemption, il existe des OS dits temps-réel qui proposent un ensemble d'APIs permettant de définir des priorités entres les différents *threads* à exécuter. Ainsi, au moment de choisir un *thread* à exécuter, l'ordonnanceur prend en compte la priorité de chaque *thread* dans sa sélection. Remarquons qu'il existe des systèmes qualifiés de temps-réel durs. Ces systèmes permettent de garantir des échéances pour l'ensemble des *threads* à ordonnancer, l'ordonnanceur utilise alors une politique qui choisit le *thread* à exécuter en se basant sur la priorité et l'échéance de chaque *thread* en attente. Par opposition, il est également possible d'utiliser des politiques d'ordonnancement dites

non-temps-réel, où chaque *thread* dispose de la même priorité, l'ordonnanceur se base alors sur le temps d'attente de chaque *thread* pour choisir celui à exécuter. Dans ce cas, on obtient un partage quasi équitable des ressources de calcul entre les différents *threads*.

La période de temps élémentaire de contrôle de l'ordonnanceur est appelé quantum et elle est paramétrable. Un faible quantum permet d'avoir un contrôle très fin sur l'ordonnancement des différents *threads* et une latence plus faible, mais représente un surcoût de calcul. Un quantum élevé permet de réduire l'impact de l'ordonnanceur sur l'utilisation du CPU mais implique une moins bonne réactivité. Par exemple si le quantum est de 25 ms et qu'un processus met 20 ms à s'exécuter, alors il faut attendre la fin du premier processus avant qu'un autre processus commence à s'exécuter. Dans le cas où le quantum est plus petit que 20 ms, alors l'ordonnanceur peut préempter l'exécution du premier processus. L'ordonnanceur de l'OS Linux utilise l'algorithme *Completely Fair Scheduler* (CFS) [PABLA, 2009]. Avec ce type d'ordonnanceur le quantum n'est plus fixe mais varie en fonction du nombre de *threads* à exécuter et de leurs priorités. De plus il est propre à chaque *thread*. Le détail du calcul est donné dans [KOBUS et SZKLARSKI, 2009]. La valeur du quantum peut évoluer entre deux valeurs extrêmes, correspondant à des paramètres du CFS :

— $sched\_latency\_ns$: la valeur maximale,

— $sched\_min\_granularity\_ns$: la valeur minimale.

Dans notre cas, les différents SoCs testés proposent un noyau Linux préemptif, avec un ordonnanceur configuré avec une politique non-temps-réel (tous les *threads* ont la même priorité) ayant les configurations du CFS suivantes :

$$\begin{cases} sched\_latency\_ns = & 18ms \\ sched\_min\_granularity\_ns = & 2.25ms \end{cases} \tag{4.2}$$

Concernant la mesure des délais d'exécutions, nous choisissons de laisser à chacun des *threads* le soin de mesurer leurs temps d'exécution respectifs. Nous utilisons également l'option **CLOCK_MONOTONIC** de manière à prendre en compte le temps où le *thread* actif est en attente de la disponibilité d'une ressource. On peut donc s'attendre à des mesures différentes pour les différents *threads*. Par exemple si deux *threads* concurrents partagent une même ressource, le premier *thread* accédant à la ressource s'exécutera rapidement, alors que le second devra attendre que l'ordonnanceur lui attache la ressource et aura donc un temps d'exécution plus grand. Ce phénomène risque d'être d'autant plus marqué si le temps d'exécution de chaque *thread* est proche de son quantum.

4.5.2 Calcul

Les temps d'exécutions sont exprimés par un facteur correspondant au temps d'exécution relatif à ceux obtenus dans le test *low-level* correspondant. Ainsi, un temps de « ×1 » signifie que le temps d'exécution est égal à celui mesuré lorsque le test s'exécute sans concurrence (*low-level*) et un temps de « ×2 » signifie qu'il met deux fois plus de temps à s'exécuter. Pour les résultats obtenus sur CPU, nous présentons les temps d'exécutions mesurés lorsque deux tests s'exécutent simultanément sur le même cœur de calcul et sur deux cœurs différents. Notons que pour les tests de calcul, nous effectuons 128 mesures par valeur d'I_A.

Résultats sur *int8*

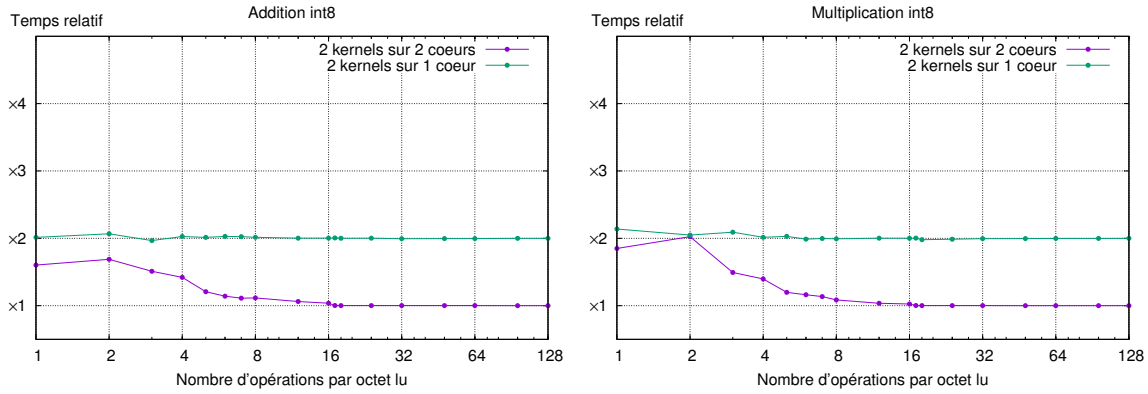
Dans un premier temps, nous donnons en figure 4.27 les résultats de nos vecteurs de test *mid-level* obtenus sur des *int8* sur Tegra K1, Tegra X1 et R-Car H2 pour l'addition et la multiplication. Dans tous les cas, on s'aperçoit que lorsque le nombre d'opérations par octet lu est supérieur à 16 le comportement est identique pour tous les SoCs testés. Ainsi, dans le cas où les tests sont exécutés sur des cœurs différents le temps de calcul est identique à une exécution sans concurrence, tel qu'obtenu lors des tests *low level*, alors que le temps de calcul est doublé lorsque les tests sont exécutés sur le même cœur. Comme nous l'avons vu précédemment, lorsque le nombre d'opérations par octet lu est supérieur à 16 pour des *int8*, le compilateur cesse de vectoriser, on se trouve dans la zone de forte I_A et donc limité par les capacités de calcul. Il parait donc tout à fait logique que seule la concurrence de deux tests sur le même cœur ait un impact sur le temps de calcul. De plus, dans cette zone le temps de calcul multiplié par le nombre de mesures (128) est suffisamment grand par rapport au quantum de l'ordonnanceur pour que son impact ne soit pas visible.

Dans le cas où le nombre d'opérations par octet lu est inférieur à 16, on se trouve dans la zone où le compilateur vectorise les opérations et il est donc potentiellement limité par les capacités de la mémoire. Ainsi, si les tests sont exécutés sur des cœurs différents, seul le cache L2 sera partagé entre les deux cœurs et peut potentiellement être une source de conflits. Or, si les tests sont exécutés sur le même cœur, ils partageront le cache L1. De plus, les temps de calcul dans cette zone sont assez faibles comparés au quantum de l'ordonnanceur. On observe donc une différence de comportements entre les deux *threads* pour certains points. Sur la figure 4.27, nous avons cependant choisi d'afficher uniquement le temps d'un seul *thread*, l'impact de l'ordonnanceur peut se voir par la grande variabilité du temps mesuré (par exemple un temps proche de ×1 sur Tegra X1 ou proche de ×3 pour R-Car). Notons que pour un nombre de mesures plus important (par exemple 1024 ou 2048 mesures par point au lieu de 128), la charge de calcul de chaque *thread* est plus importante et ce phénomène n'est plus visible. Cependant cela implique également un temps de test beaucoup trop long.

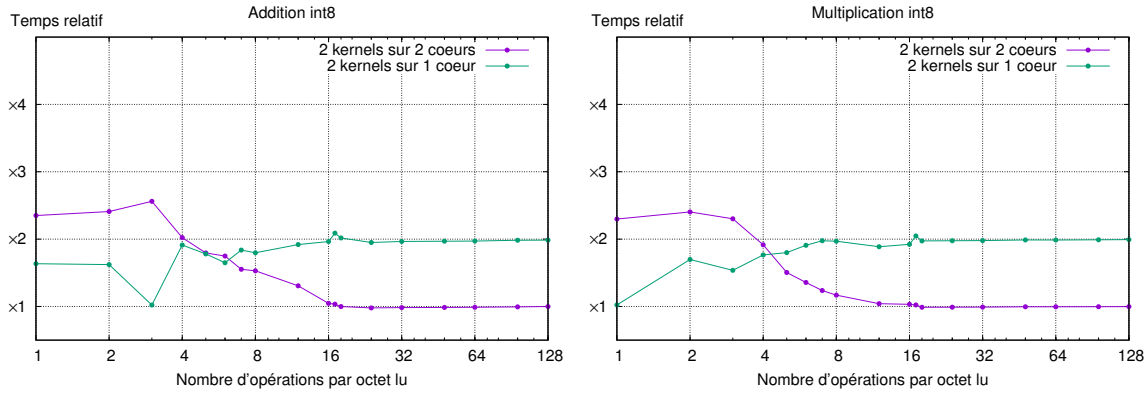
Pour le Tegra K1, on observe que le cache L2 est assez peu sensible aux accès concurrents (du moins pour cette taille de données) : les délais mesurés lorsque deux tests s'exécutent sur deux cœurs différents sont assez proches du délai d'un test sans concurrence pour l'addition et la multiplication. Pour Tegra X1, même si sa bande passante mémoire théorique est plus rapide que celle du K1, on observe un impact plus important lorsque que l'on se trouve dans la zone de faible I_A. Cela provient très probablement du cache L2. Cela est encore plus marqué sur le R-Car H2 avec des temps d'exécution quatre fois plus grands à faible I_A.

Résultats sur *int32*

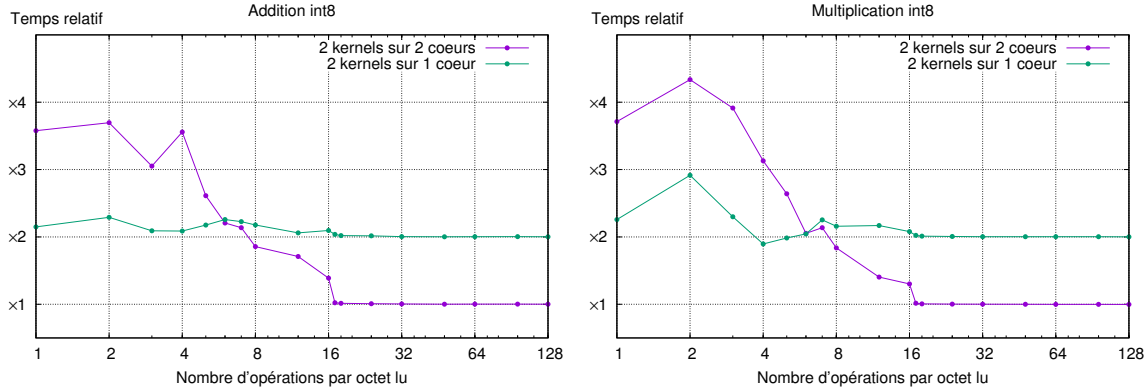
Ensuite, nous donnons en figure 4.28 les résultats de nos vecteurs de test *mid-level* obtenus sur des *int32* sur Tegra K1, Tegra X1 et R-Car H2 pour l'addition et la multiplication. Dans tous les cas, on s'aperçoit que lorsque le nombre d'opérations par octet lu est supérieur à 4, le comportement est identique pour tous les SoCs testés. Ainsi, dans le cas où les tests sont exécutés sur des cœurs différents le temps de calcul est identique à une exécution non concurrente, alors que le temps de calcul est doublé lorsque les tests sont exécutés sur le même cœur. Comme nous l'avons vu précédemment, lorsque le nombre d'opérations par octet lu est supérieur à 4 pour des *int32*, le compilateur cesse de vectoriser, on se trouve donc dans la zone de forte I_A et donc limité par les capacités de calcul.



(a) K1



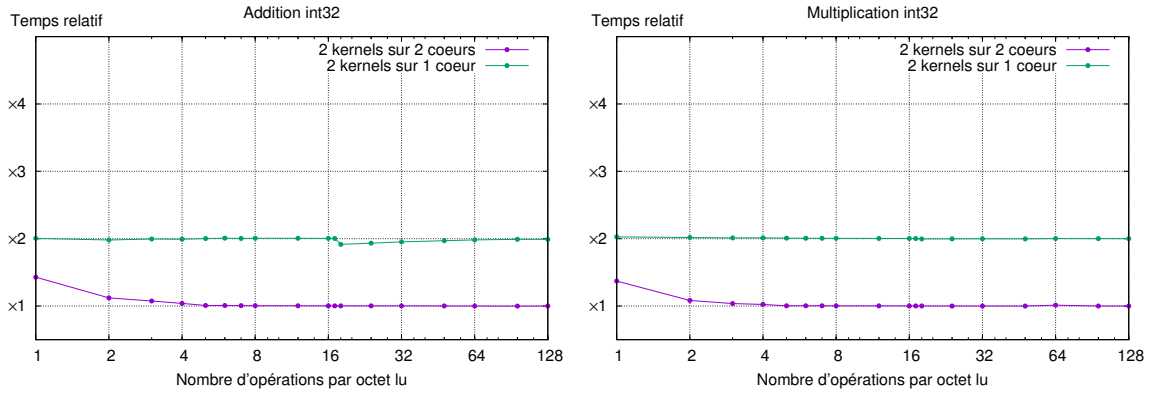
(b) X1



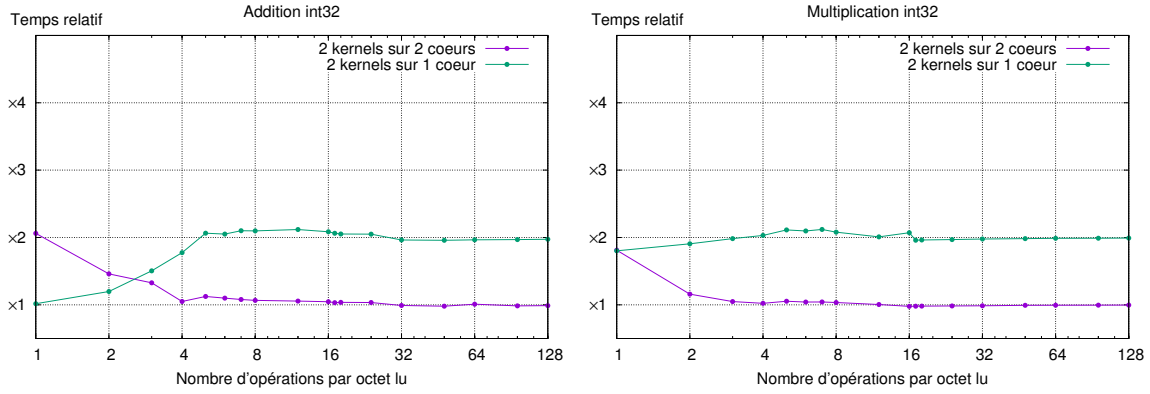
(c) R-Car H2

FIGURE 4.27 – Résultats des tests *mid-level* de calcul sur des *int8*

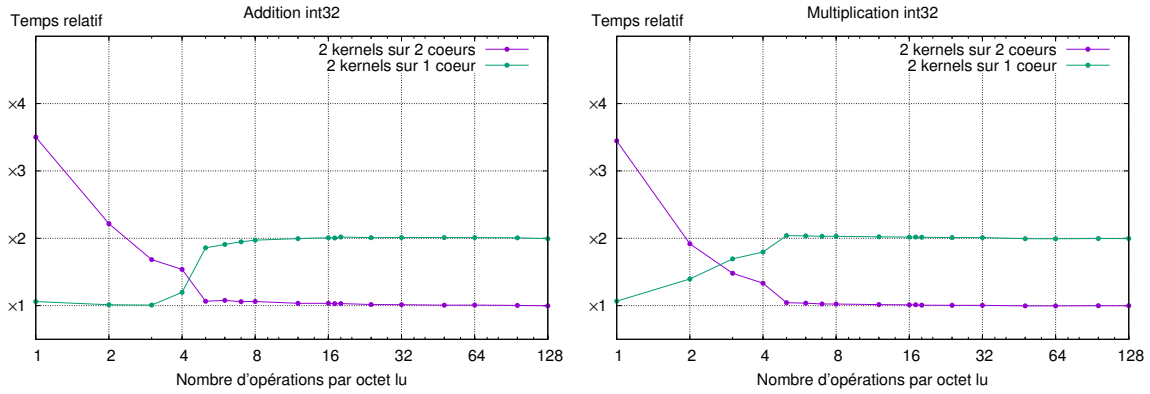
Il parait donc tout à fait logique que seule la concurrence de deux tests sur le même cœur ait un impact sur le temps de calcul.



(a) K1



(b) X1



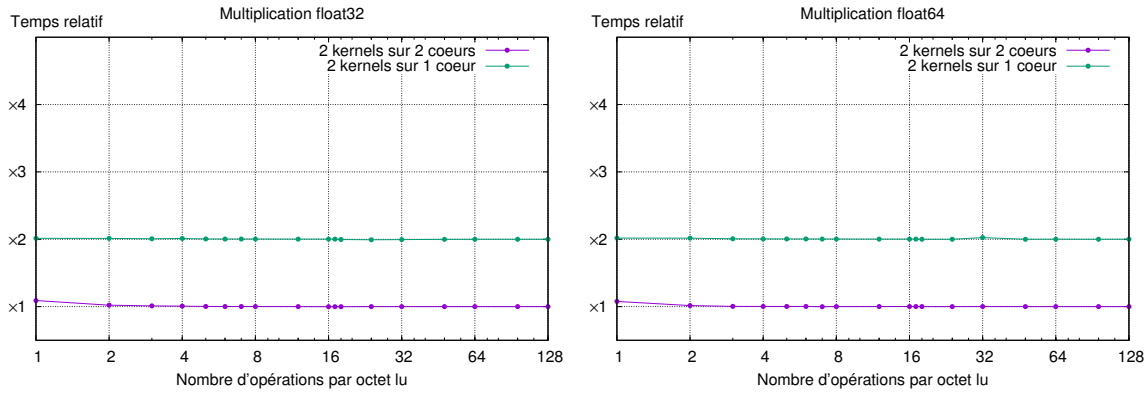
(c) R-Car H2

FIGURE 4.28 – Résultats des tests *mid-level* de calcul sur des *int32*

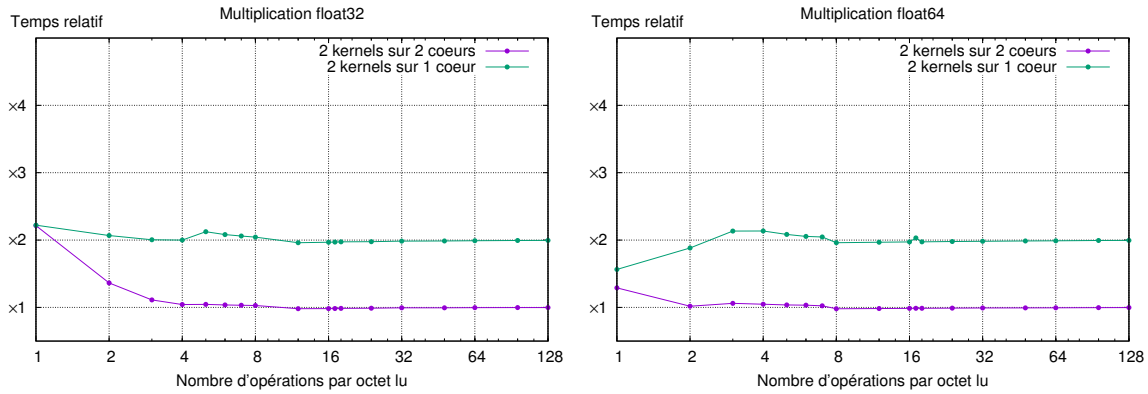
Dans le cas où les tests sont exécutés sur des cœurs différents, on obtient des résultats qui confirment les observations faites pour les *int8*. Ainsi, sur Tegra K1 le temps d'exécution de chaque test est assez peu impacté par la parallélisation, on obtient un temps seulement 1,5 fois plus grand pour 1 opération par octet lu. Sur Tegra X1, l'impact de la parallélisation est un peu plus marqué, on observe un temps environ 2 fois plus grand pour 1 opération par octet lu. Finalement, conformément aux résultats sur *int8*, c'est sur R-Car H2 que l'impact de la parallélisation est le plus marqué : on obtient un temps d'exécution 3,5 fois plus grand pour 1 opération par octet lu lorsque deux tests s'exécutent en parallèle sur deux cœurs différents.

Résultats sur *float*

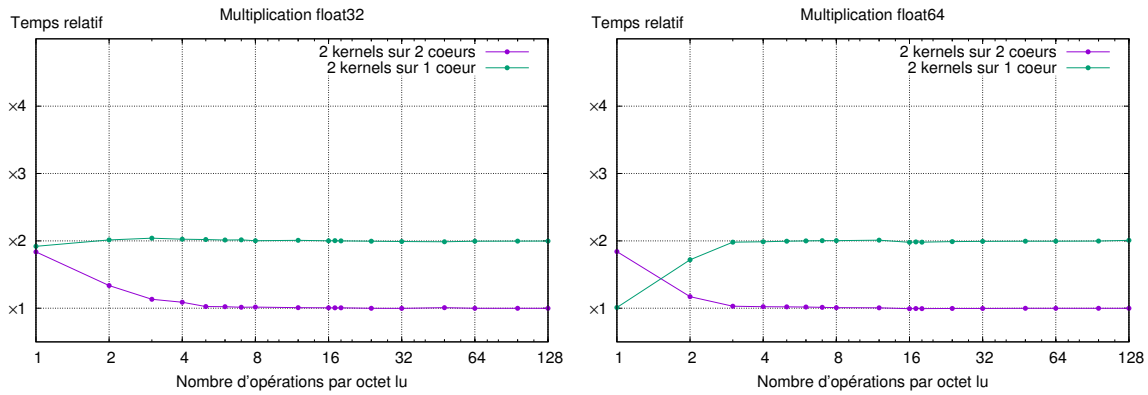
Nous donnons en figure 4.29 les résultats de nos vecteurs de test *mid-level* obtenus sur des *float32* et des *float64* sur Tegra K1, Tegra X1 et R-Car H2 pour la multiplication. On observe finalement des comportements assez similaires à ceux obtenus pour des *int32*. On remarque toutefois sur R-Car H2 que l'impact de la concurrence parait plus faible sur des *float* que sur des *int8* ou *int32*. Nous pensons que cela est causé par le chargement des données dans les registres de l'unité de calcul en virgule flottante, qui sont les mêmes registres utilisés pour le calcul SIMD par NEON.



(a) K1



(b) X1

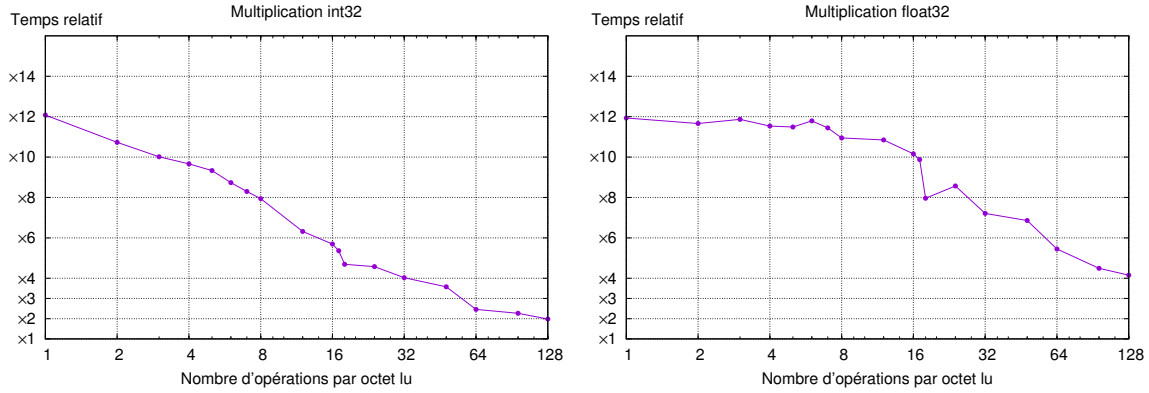


(c) R-Car H2

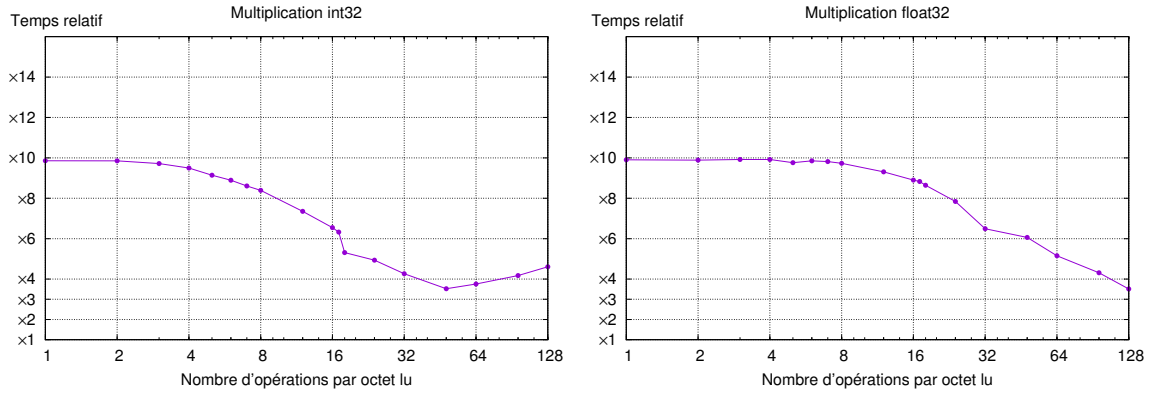
FIGURE 4.29 – Résultats des tests *mid-level* de calcul sur des *float*

Résultats sur GPU

Nous donnons en figure 4.30 les résultats de nos vecteurs de test *mid-level* obtenus sur des multiplications *int32* et *float32* sur le GPU du Tegra K1 et du Tegra X1. Dans tous les cas, on s'aperçoit que les temps d'exécution mesurés explosent, surtout lorsque le nombre d'opérations par octet lu est faible et donc que l'on utilise fortement les ressources de la mémoire. Il faut savoir que ces GPU disposent de très peu de SMX (seulement un pour le K1 et deux pour le X1), dans ces conditions les kernels concurrents sont donc obligés de s'exécuter les uns après les autres. Après investigations, nous nous sommes aperçus que ce surcoût est causé par un très grand temps d'inactivité entre le lancement des deux kernels CUDA. Dans les conditions d'un GPU disposant d'un grand nombres de SMX, sur PC, nous avons observé que les exécutions des différents kernels concurrents parviennent à se superposer et permet donc d'obtenir des comportements d'exécutions concurrentes beaucoup plus intéressants, comme illustré en figure 4.31.



(a) K1



(b) X1

FIGURE 4.30 – Résultats des tests *mid-level* de calcul sur GPU

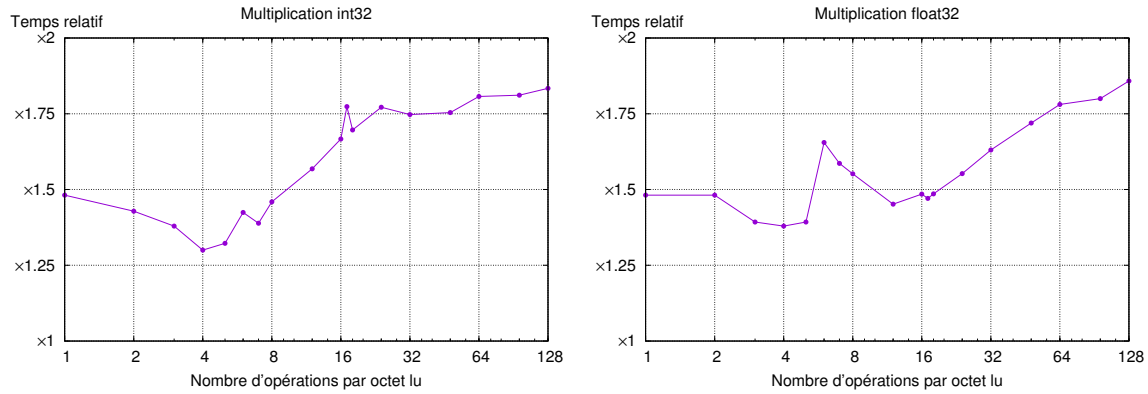


FIGURE 4.31 – Résultats des tests *mid-level* de calcul sur un GPU GTX 980M pour PC

Discussion

Finalement, nos vecteurs de test *mid-level* ont pu montrer que, sur ARM, l'exécution concurrente sur un même cœur de deux threads utilisant les même types d'opérations ne faisait que multiplier par deux leurs temps d'exécution respectifs dans le cas où ceux-ci ont une forte I_A. Ainsi, dans ce cas, exécuter deux *kernels* indépendants en même temps sur le même cœur ou séquentiellement revient finalement au même. Cela n'est plus valable lorsque le temps d'exécution de chaque *thread* est proche du quantum de l'ordonnanceur : dans ce cas l'exécution concurrente peut être légèrement plus coûteux qu'une exécution séquentielle. Notons que l'API Linux propose différentes politiques d'ordonnancement [GALLMEISTER, 1995], il pourrait être intéressant d'effectuer la même approche avec des politiques différentes.

Remarquons que nous avons testé ici le pire cas, c'est-à-dire lorsque les *kernels* concurrents cherchent à effectuer les mêmes types d'opération et donc utilisent la même unité élémentaire. Il pourrait être intéressant de mesurer l'impact d'une exécution concurrente de deux *kernels* effectuant des opérations différentes (par exemple un *kernel* effectuant des opérations entières et l'autre des opérations en virgule flottante). Concernant l'exécution concurrente de deux *kernels* à forte I_A sur deux cœurs différents, nos vecteurs de test ont montré que le délai d'exécution est équivalent à une exécution sans concurrence, même s'ils partagent le même cache L2.

Cependant sur GPU, nous avons vu par le biais des résultats issus de nos vecteurs de test que l'impact d'une exécution concurrente de plusieurs *kernels* est catastrophique. Ainsi, dans le cas où deux *kernels* indépendants sont à exécuter par le même GPU au même moment, il est très fortement préférable de considérer de les exécuter séquentiellement, plutôt que de manière concurrente. Néanmoins, dans le cas où un utilisateur souhaiterait tout de même exécuter deux *kernels* indépendants simultanément, les résultats issues de nos vecteurs de test nous permettent, en connaissant l'I_A de chacun des *kernels*, d'estimer l'impact de la concurrence sur le temps d'exécution de chaque *kernel*.

4.5.3 Mémoire

Pour les tests *mid-level* de la mémoire, nous choisissons de nous focaliser sur le test de copie dans la configuration McpyFast et McpySlow. Ainsi, cela permet d'illustrer directement l'impact de la concurrence sur le temps de lecture et d'écriture dans le cas où les données sont probablement dans le cache (McpyFast) et dans le cas où le *dirty bit* est levé (McpySlow). De plus, nous effectuons uniquement les mesures lorsque les tests sont exé-

cutés de manière concurrente sur des cœurs différents. En effet, les mesures effectuées dans la configuration où les tests sont en concurrence sur le même cœur sont particulièrement bruités et donc difficilement interprétables.

Les temps d'exécutions sont donnés par un facteur correspondant au temps d'exécution relatif par rapport à celui obtenu dans le test *low-level* correspondant. Ainsi, un temps de « ×1 » signifie que le temps d'exécution est égal à celui mesuré lorsque le test s'exécute sans concurrence (*low-level*), et un temps de « ×2 » signifie qu'il met deux fois plus de temps à s'exécuter.

McpyFast

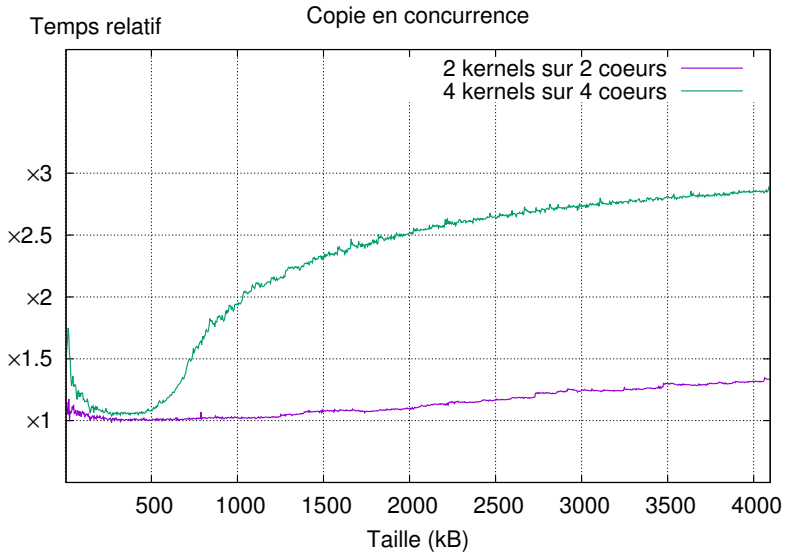
Dans la figure 4.32, nous donnons les résultats des tests de copie obtenus sur Tegra K1, Tegra X1 et R-Car H2 en utilisant 2 et 4 cœurs du CPU dans la configuration McpyFast. Remarquons que la quantité totale de données à copier est liée aux nombres de tests exécutés simultanément, ainsi si quatre tests effectuent simultanément une copie d'une taille de 1 Mo, il y aura un total de 4 Mo de données copiées simultanément.

Commençons dans un premier temps par étudier les résultats obtenus sur Tegra X1, là où la bande passante mémoire théorique est la plus grande. On constate qu'effectuer deux copies de manière concurrente n'a pratiquement aucun effet sur les délais d'accès mémoire, on obtient un facteur de ×1. Le même phénomène est également visible lorsque nous effectuons quatre copies simultanées, le temps d'exécution est seulement augmenté d'un facteur inférieur à ×1.25 alors que nous effectuons quatre fois plus de copies de données. Nous pensons que ce comportement est causé par la bande passante des caches L1 (unique à chaque cœur) qui est plus faible que la bande passante de la mémoire. Ainsi, lorsque l'on travaille avec des *kernels* ayant une faible I_A, il y a tout intérêt à profiter du multicœur pour minimiser les délais d'accès mémoire, soit en exécutant plusieurs *kernels* simultanément sur des cœurs différents soit en parallélisant les *kernels* de manière à ce que ceux-ci utilisent l'ensemble des cœurs disponibles. Les mêmes conclusions sont valables sur le CPU du Tegra K1, mais avec une accélération plus faible. On observe ainsi que le temps de deux copies concurrentes est augmenté d'un facteur inférieur à ×1.5 et inférieur à ×3 lorsque quatre copies sont en concurrence.

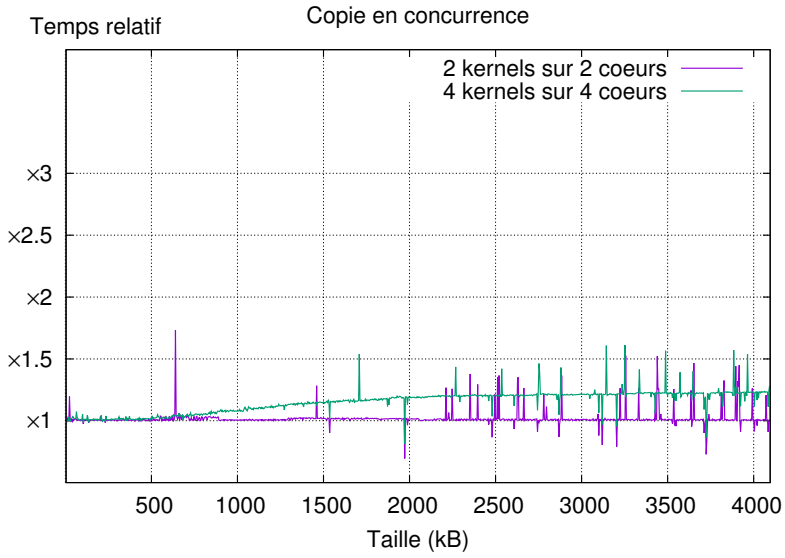
La bande passante théorique du R-Car H2 est de 6,4 Go/s. Comme nous avons montré dans la figure 4.12, les délais de lecture sont à la limite de la bande passante théorique, on peut donc s'attendre à un impact de la concurrence beaucoup plus marqué que ce que l'on obtient sur Tegra K1 ou Tegra X1. Comme attendu, on observe que le temps d'exécution de deux copies concurrentes est augmenté d'un facteur supérieur à ×2 et d'un facteur supérieur à ×4 pour quatre copies concurrentes. On peut donc conclure que l'utilisation d'un seul cœur suffit à saturer la bande passante mémoire sur R-Car H2, il faut donc éviter autant que possible les accès mémoires concurrents sur cette plateforme.

McpySlow

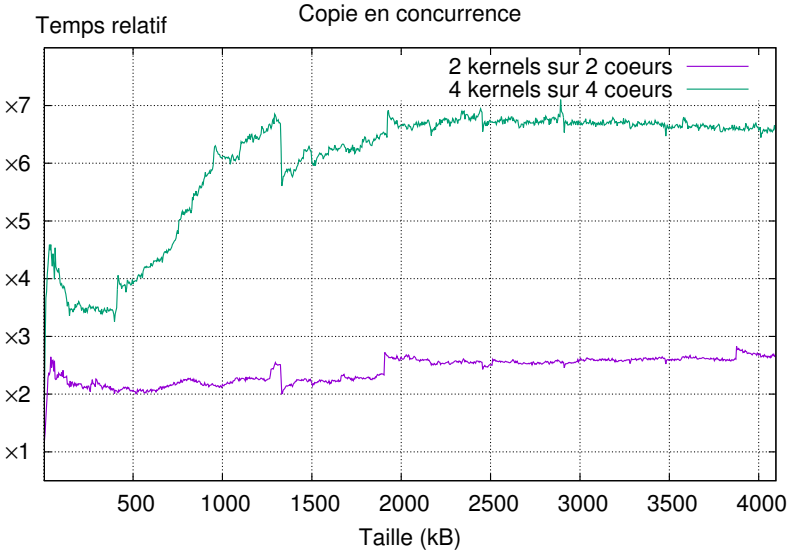
Dans la figure 4.33, nous donnons les résultats des tests de copie obtenus sur Tegra K1, Tegra X1 et R-Car H2 en utilisant 2, 3 et 4 cœurs du CPU dans la configuration McpySlow. Nous nous attendons donc à des délais plus importants que ceux observés précédemment à cause des effets de caches qui sont particulièrement mis en évidence par les accès concurrents. Sur l'ensemble des SoCs, on s'aperçoit alors de l'impact de la concurrence sur le cache L2 lorsque le *dirty bit* est levé. Ainsi, on observe un temps de copie beaucoup plus important lorsque la taille des données à copier est inférieure à la taille du cache L2 (2 MB).



(a) K1

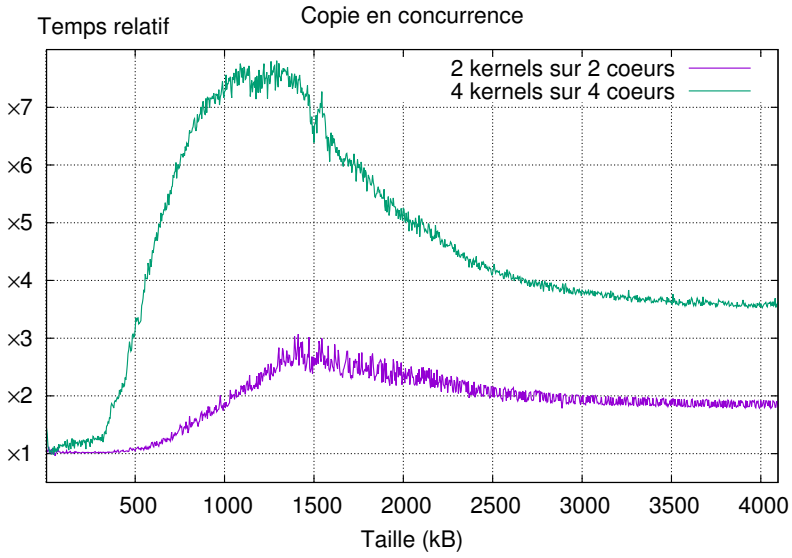


(b) X1

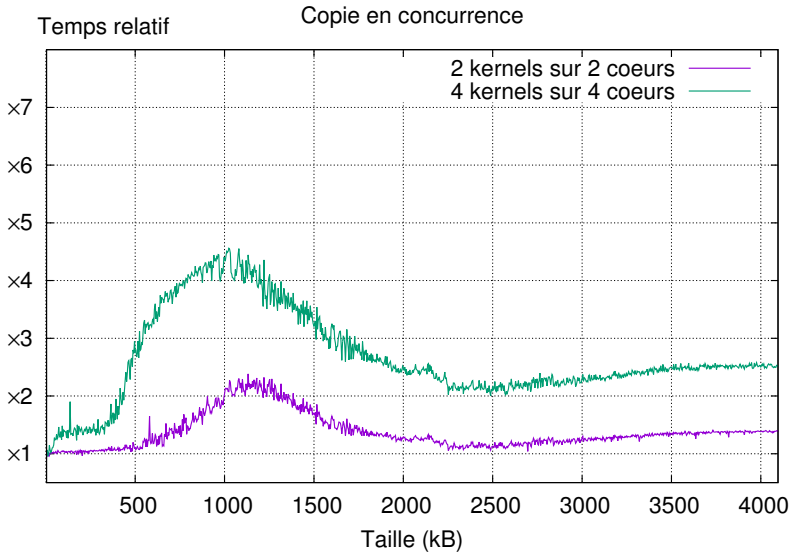


(c) R-Car H2

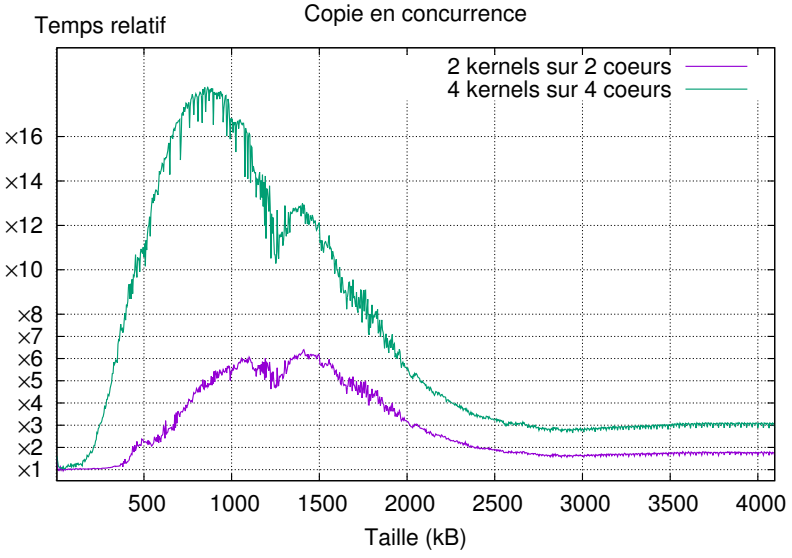
FIGURE 4.32 – Résultats du test Memcopy *mid-level*



(a) K1



(b) X1



(c) R-Car H2

FIGURE 4.33 – Résultats du test Memcopy *mid-level*

Intéressons-nous tout d'abord aux résultats obtenus sur Tegra X1. Dans cette configuration le délai de copie dépasse le facteur 2 lorsque deux *threads* copient chacun 1024 kB (soit la moitié du cache L2). Ce délai dépasse le facteur 4 lorsque quatre *threads* effectuent la copie de manière concurrente. Remarquons que cela est cohérent avec les résultats obtenus sur les tests *mid-level* de calcul lorsque le nombre de calculs par octet lu est égal à 1. Sur Tegra K1, l'impact de la concurrence dans cette configuration est un peu plus marqué, on mesure des délais de copie jusqu'à 7.5 fois plus grand lorsque quatre *threads* sont exécutés en même temps. Dans la configuration où deux *threads* copient chacun 1500 kB, le délai est environ 3 fois plus grands qu'une copie sans concurence de la même taille.

C'est sur R-Car que les effets des accès concurrents au cache L2 sont les plus visibles. Ainsi, on observe un temps de copie jusqu'à 15 fois plus grand lorsque quatre *threads* copient chacun 800 kB. Le temps de copie est environ 6 fois plus grand lorsque deux *threads* copient chacun 1500 kB.

Finalement, on remarque que sur l'ensemble des plateformes l'impact de la concurrence est beaucoup moins marqué lorsque la taille des données à copier devient supérieure à la taille du cache L2.

Lecture non contiguë

Enfin, nous proposons d'étudier l'impact de la concurrence sur des lectures effectuées de manière non contiguë. Tout d'abord, nous donnons dans la figure 4.34 les délais de lecture non contiguë mesurés pour Tegra K1, Tegra X1 et R-Car H2 dans les configurations suivantes :

— 1 cœur : accès à des données de manière non contiguë sans concurrence, comme nous l'avions présenté en figure 4.12 ;

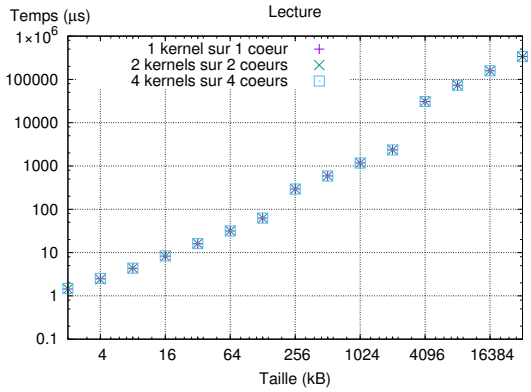
— 2 cœurs : exécution du test de lecture non contiguë sur deux cœurs différents de manière simultanée ;

— 4 cœurs : exécution du test de lecture non contiguë sur quatre cœurs différents de manière simultanée.

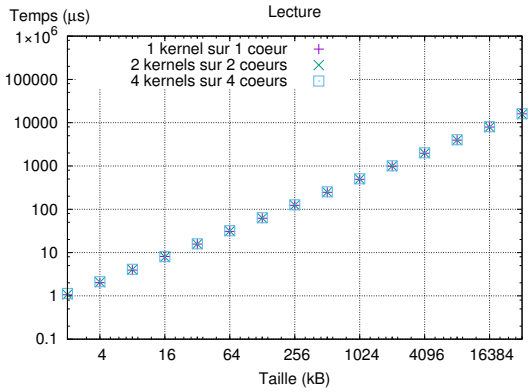
De manière assez surprenante on observe que la concurrence n'a aucun effet sur les délais de lecture, quelques soient la plateforme et la taille des données à lire. Ainsi, dans la configuration 4 cœurs, on lit quatre fois plus de données que dans la configuration 1 cœur et le temps d'exécution reste constant. Comme nous l'avons montré dans la figure 4.14, les temps de lecture non contiguë sont assez éloignés des temps de lecture contiguë et de la bande passante théorique.

Pour souligner les effets du cache dans cette configuration, nous proposons d'effectuer le même test mais en effectuant après l'allocation une écriture sur l'ensemble des données, de manière à lever le *dirty bit* des blocs du cache. Les résultats pour Tegra K1 et Tegra X1 sont donnés en figure 4.35.

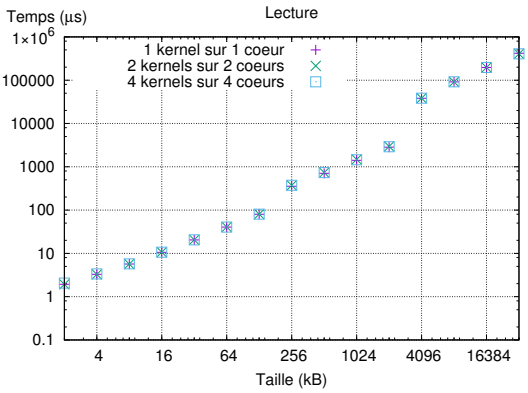
Conformément aux résultats obtenus précédemment on observe sur K1 un temps de lecture plus grand lors de la lecture concurrente pour des données dont la taille est inférieure à celle du cache L2. L'impact est le plus visible lorsque chaque *thread* cherche à lire chacun 1024 kB. Pour Tegra X1, nous n'observons aucune différence pour rapport aux temps de lecture sur des données non initialisées. Cela confirme donc notre hypothèse initiale que, pour ce test, les performances ne sont pas limités par les délais mémoires mais seulement par le calcul d'adresse à la donnée courante.



(a) K1

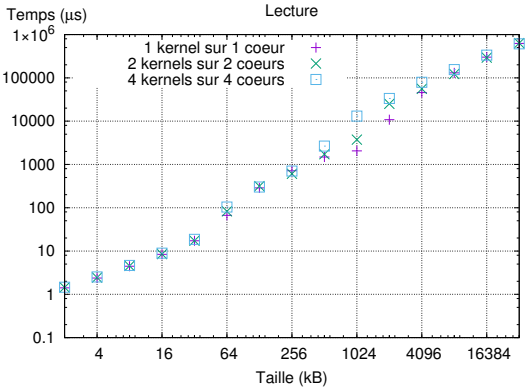


(b) X1

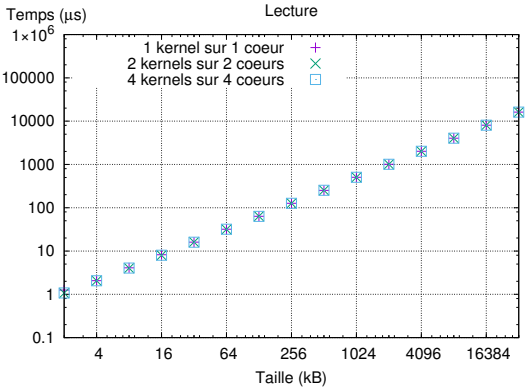


(c) R-Car H2

FIGURE 4.34 – Temps de lecture sur des *int32* pour des accès non contigus en concurrences



(a) K1



(b) X1

FIGURE 4.35 – Temps de lecture sur des *int32* pour des accès non contigus en concurrences après initialisation

4.6 Vecteurs de test *high-level*

Les vecteurs de test test *high-level*, contrairement aux vecteurs précédents, s'intéressent aux performances d'exécutions d'applications complètes. Ils peuvent être divisés en deux catégories : la caractérisation d'applications dont le code source est connu et la caractérisation de codes binaires (bibliothèques par exemple) considérés comme des « boîtes noires ». Dans les deux cas le but est de caractériser les performances de ces applications tests en faisant varier divers paramètres (par exemple la taille de l'image, le nombre d'itérations, etc.) pour pouvoir prédire les futures performances dans des conditions différentes (nouvelle architecture, taille différente d'images, etc.).

4.6.1 Applications au code source connu

À partir d'une application complète dont le code source est entièrement connu, il est possible d'appliquer notre méthodologie d'analyse de l'embarquabilité tel que nous l'avons définie. Ainsi, nous sommes capables de prédire les performances de cette application et le respect ou non des contraintes temps-réel pour plusieurs SoCs, et pour différents paramètres d'entrée (par exemple la taille de l'image). Or notre approche n'est pas infaillible et dépend fortement du modèle utilisé pour l'architecture. Ainsi, il peut arriver que la prédiction de performance se trompe par rapport à une mesure réelle tout simplement à cause d'un modèle d'architecture erroné, par exemple à cause d'une capacité de calcul ou une bande passante fausse, ou alors à cause d'une fonctionnalité hardware qui n'a pas été intégrée dans le modèle.

Dans ce cas, le but des vecteurs de test *high-level* est donc d'appliquer une procédure de vérification et validation d'un modèle d'architecture précédemment construit à l'aide des vecteurs de test *low-level* et *mid-level*. Ainsi, si pour un modèle d'architecture donné, l'ensemble des performances prédites sont cohérentes avec les temps d'exécutions mesurés, on peut alors en déduire que le modèle en question est consistant pour le type d'application testée. Dans le cas contraire, il est nécessaire d'investiguer, de déterminer la source de l'erreur et de corriger le modèle en conséquence. Le processus complet est illustré dans la figure 4.36.

4.6.2 Applications au code source fermé

Une application ADAS complète peut demander l'utilisation de bibliothèques externes dont le code source n'est pas toujours ouvert ni disponible. Dans ce cas, la caractérisation de l'embarquabilité de l'application complète est plus difficile sans connaître les performances des bibliothèques en question en termes de temps de calcul. Les vecteurs de test *high-level* ont été conçus pour répondre à cette problématique. Il s'agit de caractériser et de modéliser les performances de « boîtes noires » logicielles pour différentes architectures, et ainsi d'être capable de prédire les performances pour différents paramètres d'entrée.

Caractérisation de kernels

L'exemple typique est la caractérisation des applications basées sur OpenVX. Comme discuté dans la section 2.3.4, ce standard émergeant permet d'exécuter le même code source sur différentes architectures hétérogènes, en utilisant des *kernels* implémentés par les fondeurs. Nvidia a proposé son implémentation de OpenVX, il s'agit de VisionWorks. Ce framework permet de compiler et d'exécuter le même code source sur un PC équipé

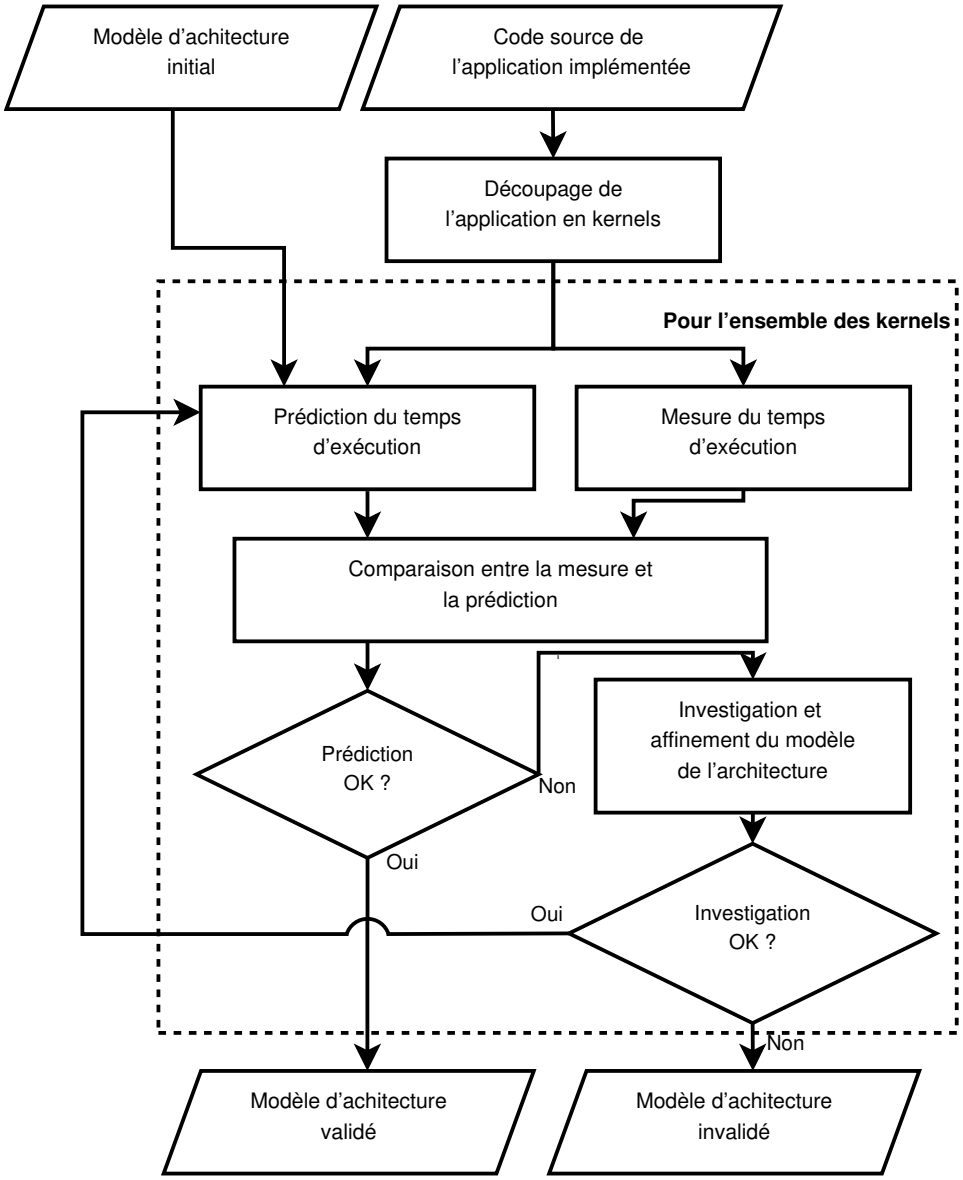


FIGURE 4.36 – Schéma de principe d'un test *high-level* pour un code source connu. Pour une architecture et une application données, il s'agit de valider ou corriger le modèle existant de l'architecture.

d'un GPU Nvidia et sur les plateformes embarquées Jetson et DrivePX. Ainsi, nous cherchons donc à caractériser les différents *kernels* de ce framework sans en connaître réellement le code source. Notre but est de modéliser et de pouvoir prédire les performances de ces applications en fonction de plusieurs paramètre (par exemple les dimensions de l'image).

Un exemple typique est donné dans la figure 4.37, où nous présentons le temps d'exécution du *kernel* OpenVX du filtre Sobel sur Tegra X1. Le code source nous est totalement inconnu, cependant grâce à ce test nous sommes en mesure d'affirmer que le temps d'exécution de ce *kernel* sur Tegra X1 est proportionnel à la hauteur de l'image, mais ne dépend pas de sa largeur (excepté une cassure pour une largeur de 1024 pixels). Par régression linéaire, on peut alors en déduire l'équation du temps d'exécution en fonction de la taille de l'image :

$$t(u,v) = \begin{cases} [8.4,8.6] \times 10^{-5}\ v + [7.1,7.3] \times 10^{-3} & \text{if } u \leqslant 1024 \\ [0.9,1.1] \times 10^{-4}\ v + [8.9,9.1] \times 10^{-3} & \text{else} \end{cases} \tag{4.3}$$

où u correspond à la largeur de l'image et v correspond à la hauteur de l'image.

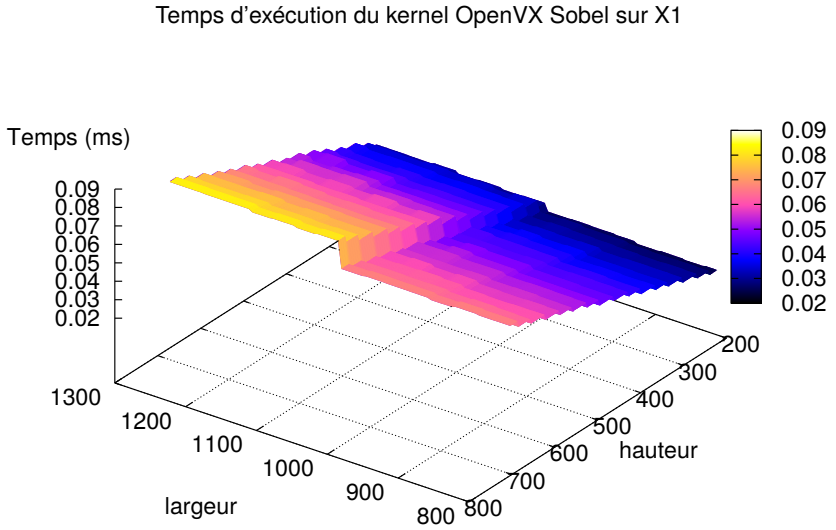


FIGURE 4.37 – Caractérisation du kernel OpenVX Sobel sur X1. Le temps d'exécution varie linéairement par rapport à la hauteur de l'image.

Cette approche est très intéressante dans le sens où elle permet de modéliser le temps de calcul d'un *kernel* au code source inconnu pour une architecture donnée. Ainsi, le test est capable de mettre en avant des spécificités en termes de temps de calcul liées à la manière dont le *kernel* a été implémenté, sans aucun accès à son code source. Remarquons tout de même que cette approche à plusieurs inconvénients. Tout d'abord la liste des paramètres à faire varier doit être choisie judicieusement. Par exemple dans le cas du *kernel* donné en figure 4.37, si l'on avait uniquement fait varier le nombre de pixels à traiter (sans distinction entre la largeur et la hauteur de l'image), la modélisation du temps d'exécution aurait été totalement différente et probablement erronée. De plus l'approche est assez longue et laborieuse, elle nécessite d'exécuter le *kernel* pour différents paramètres d'entrée et la modélisation obtenue semble valable uniquement valable pour l'architecture donnée.

Prédiction de performances

Pour contourner cette dernière difficulté, nous proposons d'étendre cette approche de caractérisation à de la prédiction de performances de *kernels* au code source fermé pour différentes architectures. En effet, si l'on suppose que les implémentations d'un même *kernel* pour deux architectures différentes sont suffisamment similaires, alors il est possible en connaissant la performance de ce *kernel* sur une architecture de prédire sa performance sur une autre architecture. Ainsi, en s'appuyant sur les caractéristiques extraites par le biais des vecteurs de test *low-level*, et en connaissant la performance d'un *kernel* pour une architecture donnée, nous visons la prédiction de la performance de ce même *kernel* pour une autre architecture, sans que son code source soit connu. Par exemple, il peut s'agir de mesurer le temps d'exécution d'un *kernel* sur PC et de prédire son temps d'exécution sur un SoC Tegra K1 ou Tegra X1.

Pour valider cette approche, nous avons choisi de prendre l'exemple de deux *kernels* OpenVX au code source inconnu :

— Conversion couleur vers niveau de gris : *COLOR_CONVERT*,

— Détection de HARRIS et STEPHENS [1988] : *HARRIS_CORNERS*.

Les caractérisations de ces deux *kernels* pour les architecture K1, X1 et PC sont données en figure 4.38.

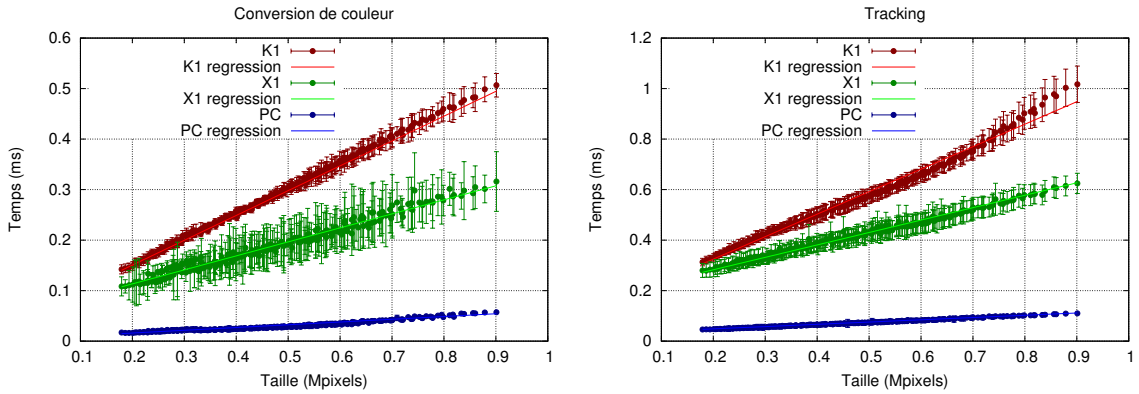


FIGURE 4.38 – Caractérisation des kernels OpenVX *COLOR_CONVERT* et *HARRIS_CORNERS*, données pour K1, X1 et PC. Les temps d'exécution sont exprimés en fonction du nombre de pixels à traiter.

D'après ces résultats, les temps d'exécutions paraissent proportionnelles par rapport au nombre de pixels à traiter, pour chaque *kernel* et pour chaque architecture nous pouvons donc obtenir par régression linéaire une équation du temps d'exécution. Par ailleurs, nous savons également que les *kernels* sont exécutés sur le GPU de chaque architecture. Or, comme nous l'avons montré dans les résultats obtenus par les vecteurs de test *low-level*, les *kernels* exécutés sur GPUs sont très souvent limités par la bande passante mémoire. Ainsi, pour l'architecture a et le *kernel* k, le temps d'exécution est donné par :

$$t_{k,a} = \alpha_a \cdot \gamma_{k,a} \cdot x + \beta_{k,a} \tag{4.4}$$

où $\beta_{k,a}$ correspond à la latence de lancement du *kernel*, α_a la bande passante mémoire obtenue à partir des vecteurs de test *low-level*, x le nombre de pixels à traiter, et $\gamma_{k,a}$ le nombre d'accès mémoire effectués pour chaque pixel. En dérivant l'équation précédente suivant x, on obtient alors :

$$\gamma_{k,a} = \frac{1}{\alpha_a} \cdot \frac{dt_{k,a}}{dx} \tag{4.5}$$

Pour la suite nous allons supposer que l'implémentation de ces deux *kernels* est identique pour les trois architectures. Cette hypothèse est plutôt raisonnable dans le sens où l'implémentation est proposée par le même fondeur (Nvidia) et les *kernels* s'exécutent sur le GPU de chaque architecture. Ainsi, si cette hypothèse est vérifiée, alors pour un *kernel* donné le nombre d'accès mémoire par pixel $\gamma_{k,a}$ doit être identique pour chacune des trois architectures :

$$\gamma_{k,K1} = \gamma_{k,X1} = \gamma_{k,PC} \tag{4.6}$$

À partir des valeurs de $\frac{dt_{k,a}}{dx}$ mesurées, et des bandes passantes α_a obtenues à partir des vecteurs de test *low-level*, nous obtenons les valeurs présentées dans le tableau 4.8.

TABLEAU 4.8 – Valeurs des $\gamma_{k,a}$ mesurées pour les architectures PC, X1 et K1.

kernel	$\gamma_{k,PC}$	$\gamma_{k,X1}$	$\gamma_{k,K1}$
Conversion	3,2	8,6	8,7
Détection	5,3	15,3	15,9

Les valeurs obtenues pour K1 et X1 sont suffisamment proches pour considérer que notre hypothèse de base concernant l'implémentation identique des *kernels* sur les deux architectures comme étant vérifiée. Cela n'est pas le cas avec les résultats obtenus sur PC, les valeurs de $\gamma_{k,PC}$ mesurées sont différentes de celle obtenus sur K1 et X1, les implémentations sont donc probablement différentes sur cette architecture. Cependant on peut s'apercevoir que la valeur de $\gamma_{k,PC}$ semble approximativement proportionnelle aux valeurs obtenues pour K1 et X1, d'un facteur donné par l'intervalle $[2.7, 3]$. D'autres mesures sur d'autres *kernels* pourrait augmenter le degré de fiabilité de cette hypothèse

Si cette approche est vérifiée, il devient alors possible en utilisant l'équation 4.5 de prédire la pente du temps d'exécution pour plusieurs architectures, en utilisant uniquement les bandes passantes α_a et les valeurs $\gamma_{k,a}$. Par la suite, le temps d'exécution du *kernel* peut être déterminé en ajoutant la latence de lancement d'un *kernel* $\beta_{k,a}$. Cette approche de prédiction de performance nous permettrait donc d'évaluer les temps de calcul d'un bibliothèque binaire considérée comme une « boîte noire » sur plusieurs architectures, en effectuant uniquement une caractérisation sur une seule architecture. Cette approche reste encore à valider. De plus, il faudrait également pouvoir adresser des cas où le *kernel* n'est pas limité par la bande passante mémoire, mais par les capacités de calculs de l'architecture. La caractérisation doit alors être un peu plus poussée, il faut notamment analyser les types d'unités élémentaires utilisées par le *kernel* pour pouvoir prédire son temps d'exécution sur d'autres architectures.

4.7 Vers la prédiction de performances

Nous avons donc présenté une méthodologie complète et innovante pour la caractérisation et la modélisation d'une architecture de calcul. Contrairement aux techniques existantes qui se focalisent soit sur un aspect particulier soit sur des applications de haut niveau, notre approche permet une caractérisation complète et approfondie. Celle-ci est basée sur des vecteurs de test à trois niveaux :

low-level : extraction des caractéristiques bas niveaux d'une architecture, par exemple la bande passante mémoire, les capacités de calcul, etc.

mid-level : étude des comportements et des effets d'exécutions concurrentes,

high-level : validation et affinement d'un modèle d'architecture en utilisant des algorithmes complexes, et caractérisation d'applications « boîtes noires ».

Nous avons montré par le biais de résultats que les données extraites sont cohérentes avec les descriptions issues des *datasheets* des fabricants, mais également que nos vecteurs tests permettent d'aller beaucoup plus loin dans l'analyse, la compréhension et la modélisation d'une architecture que ce que peut nous apprendre une *datasheet*. Ainsi, nous sommes capables de caractériser un triplet architecture de calcul, système d'exploitation et environnement logiciel tel que fournis par le fondeur. Remarquons qu'il n'est pas toujours possible d'expliquer ni de justifier des comportements en termes de performances, il est en effet très difficile de maîtriser l'ensemble des caractéristiques de ce triplets. Cependant le but de notre approche pragmatique est de mettre en avant des comportements non décrits dans les *datasheets* et de les intégrer dans le modèle de l'architecture pour servir de base à notre méthode de prédiction de performance.

Grâce à l'analyse approfondie des comportements d'une architecture lors d'exécutions concurrentes à partir de nos vecteurs de test *mid-level*, nous sommes en mesure de répondre à la question initialement posée : est-il préférable d'exécuter deux *kernels* indépendants simultanément ou séquentiellement ? Ainsi, en connaissant les caractéristiques de ces *kernels* (I_A, type d'opérations utilisés, etc.) et les résultats de vecteurs de test pour une architecture donnée, nous sommes capable de prédire l'impact qu'aura une exécution concurrente de ces deux *kernels*. La classe de vecteurs de test *mid-level* pourra bien évidement être élargie pour étudier l'impact d'autres phénomènes comme les mécanismes de synchronisation, l'impact de l'ordonnanceur, etc.

Par souci de fiabilité, nous avons également proposé une procédure de validation des modèles d'architectures, intégrée dans les vecteurs de test *high-level*. Cette procédure permet une confrontation des prédictions issues du modèle et des mesures réelles effectuées sur une ou plusieurs applications tests. Nos vecteurs de test *high-level* permettent également de caractériser des *kernels* ou des applications complètes au code source fermé et devant être considérés comme des « boites noires ». Nous avons également présenté une première approche de prédiction de performances sur des *kernels* au code source fermé pour différentes architectures. Nous pensons qu'il peut être intéressant de continuer et approfondir cette approche pour converger vers une méthode de prédiction de performances appliquée à des *kernels* « boites noires ». Cette étape de caractérisation et validation par des vecteurs de type *high-level* est extrêmement importante dans un cadre industriel qui demande des résultats concrets sur des applications complexes et réelles, malgré le fait que l'étape éventuelle d'investigation et d'affinement du modèle demande un travail d'expert qualifié et qu'il est difficilement automatisable.

Finalement, nos vecteurs de test nous permettent de construire un modèle d'architecture hétérogène pour être utilisé dans notre méthodologie d'analyse de l'embarquabilité telle qu'illustrée dans la figure 3.7. Dans notre cas, nous utilisons ces caractéristiques pour notre méthodologie de prédiction de performances. Cependant il est tout à fait envisageable d'utiliser les vecteurs de test ainsi définis pour d'autres applications, comme la comparaison objective de plusieurs architectures de calcul selon différents critères (par exemple quel processeur est le plus performant sur des calculs en virgule flottante) ou bien l'utilisation d'une autre méthode de prédiction de performances qui nécessiterait des caractéristiques très précises d'une architecture (par exemple la bande passante mémoire, le nombre de FLOPS, etc.).

4.8 Références

ASANOVIC, K., R. BODIK, B. C. CATANZARO, J. J. GEBIS, P. HUSBANDS, K. KEUTZER, D. A. PATTERSON, W. L. PLISHKER, J. SHALF, S. W. WILLIAMS et collab.. 2006, «The landscape of parallel computing research : A view from Berkeley», cahier de recherche, UCB/EECS-2006-183, EECS Department, University of California, Berkeley. 92

CHE, S., M. BOYER, J. MENG, D. TARJAN, J. W. SHEAFFER, S.-H. LEE et K. SKADRON. 2009, «Rodinia : A benchmark suite for heterogeneous computing», dans *2009 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, p. 44–54. 92

DANALIS, A., G. MARIN, C. MCCURDY, J. S. MEREDITH, P. C. ROTH, K. SPAFFORD, V. TIPPARAJU et J. S. VETTER. 2010, «The scalable heterogeneous computing (shoc) benchmark suite», dans *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ACM, p. 63–74. 92, 94

EEMBC. 2009, «EEMBC benchmark suite», Web ressource. http://www.eembc.org. 91

GAL-ON, S. et M. LEVY. 2009, «Exploring coremark™ – a benchmark maximizing simplicity and efficacy», Web ressource. http://www.eembc.org/techlit/coremark-whitepaper.pdf. 91

GALLMEISTER, B. O. 1995, «Programming for the real world, posix. 4», *OReilly & Associates, Inc.* 140

GEIGER, A., P. LENZ, C. STILLER et R. URTASUN. 2013, «Vision meets robotics : The kitti dataset», *International Journal of Robotics Research (IJRR)*. 131

GOLD, B., T. G. STOCKHAM, A. V. OPPENHEIM et C. M. RADER. 1969, *Digital processing of signals*, McGraw-Hill. 116

HARRIS, C. et M. STEPHENS. 1988, «A combined corner and edge detector.», dans *Alvey vision conference*, vol. 15, Citeseer, p. 50. 149

HENNING, J. L. 2006, «Spec cpu2006 benchmark descriptions», *ACM SIGARCH Computer Architecture News*, vol. 34, nº 4, p. 1–17. 91

KOBUS, J. et R. SZKLARSKI. 2009, «Completely fair scheduler and its tuning», . 134

MCCALPIN, J. D. 1995, «Memory bandwidth and machine balance in current high performance computers», *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, p. 19–25. 92, 94

MEUER, H., E. STROHMAIER, J. DONGARRA, H. SIMON et M. MEUER. 2016, «Top500 supercomputer site», Web ressource ; http://www.top500.org. 89

MUCCI, P. 2009, «Llcbench-low level architectural characterization benchmark suite», Web ressource. http://icl.cs.utk.edu/projects/llcbench. 93, 94

NAISHLOS, D. 2004, «Autovectorization in GCC», dans *Proceedings of the 2004 GCC Developers Summit*, p. 105–118. 100

NUGTEREN, C. et H. CORPORAAL. 2012, «The boat hull model : adapting the roofline model to enable performance prediction for parallel computing», *ACM Sigplan Notices*, vol. 47, nº 8, p. 291–292. 99

NVIDIA. 2015, «Cuda C Programming Guide», . 90

PABLA, C. S. 2009, «Completely fair scheduler», *Linux Journal*, vol. 2009, nº 184, p. 4. 134

SAUSSARD, R., B. BOUZID, M. VASILIU et R. REYNAUD. 2015a, «Optimal performance prediction of ADAS algorithms on embedded parallel architectures», dans *2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC)*, IEEE, p. 213–218. 88

SAUSSARD, R., B. BOUZID, M. VASILIU et R. REYNAUD. 2015b, «Towards an automatic prediction of image processing algorithms performances on embedded heterogeneous architectures», dans *2015 44th International Conference on Parallel Processing Workshops (ICPPW)*, IEEE, p. 27–36. 88

SAUSSARD, R., B. BOUZID, M. VASILIU et R. REYNAUD. 2016, «A robust methodology for performance analysis on hybrid embedded multicore architectures», dans *IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-16) (IEEE MCSoC-16)*, IEEE. 94

SHANNON, C. E. 2001, «A mathematical theory of communication», *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, nº 1, p. 3–55. 129

SPEC. 1995, «The Standard Performance Evaluation Corporation (SPEC)», Web ressource. http://www.spec.org. 91

STRATTON, J. A., C. RODRIGUES, I.-J. SUNG, N. OBEID, L.-W. CHANG, N. ANSSARI, G. D. LIU et W.-M. W. HWU. 2012, «Parboil : A revised benchmark suite for scientific and commercial throughput computing», *Center for Reliable and High-Performance Computing*. 92

WEICKER, R. P. 1984, «Dhrystone : a synthetic systems programming benchmark», *Communications of the ACM*, vol. 27, nº 10, p. 1013–1030. 91

WEISS, A. R. 2002, «Dhrystone benchmark : History, analysis, scores and recommendations», . 91

WILLIAMS, S., A. WATERMAN et D. PATTERSON. 2009, «Roofline : an insightful visual performance model for multicore architectures», *Communications of the ACM*, vol. 52, nº 4, p. 65–76. 93

WOO, S. C., M. OHARA, E. TORRIE, J. P. SINGH et A. GUPTA. 1995, «The splash-2 programs : Characterization and methodological considerations», dans *ACM SIGARCH Computer Architecture News*, vol. 23, ACM, p. 24–36. 92

WORMS, J. et S. TOUATI. 2016, «Going beyond mean and median programs performances», dans *IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-16)*, IEEE, p. 93–100. 97

Chapitre 5

Prédiction de performances

« *As I have said so many times, God doesn't play dice with the world.* »

Albert Einstein

Sommaire

5.1 Introduction

Nous avons présenté notre méthodologie d'analyse de l'embarquabilité dans sa globalité ainsi que notre approche pour la caractérisation d'une architecture de calcul hétérogène. Notre méthodologie d'analyse de l'embarquabilité des algorithmes décrite dans la section 3.2 repose sur la connaissance plus ou moins précise des temps d'exécution de chaque *kernel* $\tau_{i|x}$ et des temps de transfert $\delta_{i,j|x,y}$ entre les unités d'exécution. Ces valeurs peuvent être représentées sous forme d'intervalle, de variable aléatoire, ou alors sous la forme d'une valeur moyenne. Pour déterminer ces valeurs nous avons choisi une approche basée sur de la prédiction de performances. Ainsi, avec cette approche, il devient beaucoup plus facile et rapide de tester l'embarquabilité et de trouver un *mapping* satisfaisant plutôt qu'une approche « à la main » qui nécessite le portage de l'ensemble des *kernels* sur l'ensemble des unités d'exécution disponibles. Pour être applicable, cette méthodologie de prédiction de performances doit :

— être capable de prédire des temps de calcul pour différents types d'architectures tels que décrits dans la section 1.3 (par exemple CPU, GPU, DSP, etc.),

— être capable de prendre en entrée des *kernels* quelconques, et non de se limiter à certains types de *kernels*,

— être rapide et facile à mettre en place.

Pour fonctionner, la prédiction de performances nécessite les caractéristiques d'une architecture, fournies par le fabricant ou obtenues par le biais de *benchmarks*. Dans le deuxième cas, nous pouvons nous appuyer sur les données issues de nos vecteurs de test pour caractériser une architecture de calcul, comme discuté dans le chapitre précédent. Cependant il n'est pas toujours possible de lancer la procédure de caractérisation pour toutes les architectures (par exemple en cas de non-disponibilité), il faut donc être capable d'utiliser la prédiction de performances en se basant uniquement sur des informations issues de *datasheets*, même si la prédiction risque d'être moins précise.

En plus du modèle de l'architecture, la prédiction de performances nécessite une description de l'algorithme à embarquer. Dans notre approche il s'agit des caractéristiques des *kernels* de l'algorithme. Il peut s'agir d'une description haut niveau, d'une analyse de code source ou d'une analyse d'un fichier binaire. En fait la difficulté réside dans le compromis entre la complexité du modèle de prédiction de performances et la précision des prédictions. Ainsi, si la méthode nécessite un code source entièrement optimisé pour une unité d'exécution en particulier, dans le but de prédire le temps de calcul sur cette unité d'exécution, alors la méthode en question n'apporte pas un grand intérêt. Dans ce cas là autant mesurer directement les performances réelles. Cependant, si la méthode nécessite seulement une description haut niveau d'un *kernel* pour prédire sa performance sur différents types d'unités d'exécution, alors il devient possible de trouver rapidement quelle unité d'exécution sera la plus intéressante pour embarquer ce *kernel*.

Dès le début de la thèse, nous avons identifié la problématique de prédiction de performances comme étant l'un des objectifs principaux pour notre méthodologie de l'analyse de l'embarquabilité. C'est d'ailleurs sur ce sujet que nous avons proposé nos premières contributions. En fait, comme présenté dans notre poster [SAUSSARD et collab., 2015a], nous souhaitions initialement être capables de prédire le temps d'exécution d'un *kernel* uniquement à partir de données issues des *datasheets*. Notre approche initiale était basée sur une classification des types d'opérations suivant les différentes unités élémentaires présentes sur une unité d'exécution. Les premiers résultats se sont montrés assez encourageants, mais souvent avec une très grande marge d'erreur. De plus nous étions incapables de prédire les temps d'accès mémoire et les temps de transfert. C'est donc pour

palier ces problèmes que nous avons travaillé sur la caractérisation d'architecture à l'aide des vecteurs de test et amélioré notre méthode initiale de prédiction de performances.

5.2 Travaux existant sur la prédictions de performances

Avec l'utilisation de plus en plus fréquente d'accélérateurs matériels massivement parallèles comme des GPUs, l'implémentation d'algorithmes devient de plus en plus complexe. En effet, comme discuté par LEE et collab. [2010] et VUDUC et collab. [2010], l'utilisation d'accélérateurs GPUs n'est pas toujours bénéfique comparée à une version optimisée s'exécutant sur CPU. De plus, d'après SHEN et collab. [2014] les temps de transfert entre CPU et GPU doivent impérativement être pris en compte, puisqu'il s'agit très souvent d'un goulot d'étranglement. Ainsi, pour pallier cette difficulté, la communauté scientifique se tourne de plus en plus vers des approches de prédiction de performances, de manière à déterminer s'il peut être bénéfique d'utiliser un accélérateur hardware et comment répartir les charges de calcul entre un CPU et un ou plusieurs accélérateurs hardwares de manière à être le plus performant possible.

LOPEZ-NOVOA et collab. [2015] proposent une étude des modèles de prédictions existant. D'après les auteurs, un modèle de prédiction de performances peut être vu comme un système renvoyant des données de sorties en fonction de caractéristiques de l'architecture et de caractéristiques de l'application. Les données de sortie peuvent être différentes d'un système à l'autre et dépendent de ce pour quoi le modèle à été conçu. Ainsi, les auteurs distinguent quatre types d'informations différentes pouvant être issues de ces modèles :

— la prédiction du temps d'exécution de l'application d'entrée sur l'architecture cible,

— l'identification des goulots d'étranglement et la proposition de modifications permettant de passer outre ces limitations,

— une estimation de la puissance électrique consommée,

— une description détaillée, pas à pas, de l'utilisation de ressources de l'architecture cible par l'application d'entrée, basée sur de la simulation.

Dans le cadre de cette thèse, nous nous intéressons à l'embarquabilité d'un algorithme en termes de temps de calcul. Ainsi, nous avons choisi de ne pas adresser les problématiques de puissance électrique consommée, nous ne détaillerons donc pas les modèles de prédiction de performances de ce type.

5.2.1 Simulation

Un simulateur est capable de reproduire le comportement détaillé, pas à pas, d'une architecture de calcul. La précision d'un simulateur dépend de nombreux facteurs, notamment la qualité de la modélisation de l'architecture et de l'application. Certains simulateurs sont conçus pour cibler des GPUs, c'est le cas notamment de GPGPU-sim [BAKHODA et collab., 2009] et de Barra [COLLANGE et collab., 2010]. Ces deux simulateurs sont capables de simuler l'exécution de code CUDA et génèrent des informations détaillées à propos des compteurs de performances, le nombres d'accès mémoire, l'utilisation des différentes ressources. GPGPU-sim affiche une précision de 97,3% sur le nombre d'instructions par cycle prédit sur un GPU Fermi. Quant à Barra, les auteurs rapportent une erreur allant de 0 à 23 pourcents entre les prédictions et les mesures, suivant les *workloads* utilisés.

D'autres simulateurs ciblent des architectures hétérogènes, le plus souvent composées d'un CPU et d'un GPU avec une mémoire propre à chaque processeur ou avec une mémoire partagée. On peut citer par exemple le très célèbre Multi2Sim, de UBAL et collab. [2012], mais également MacSim [KIM et collab., 2012] ou encore FusionSim [ZAKHARENKO et collab., 2013]. La dernière version de Multi2Sim permet de cibler des CPUs x86, MIPS, ARM, et des GPUs ARM et Nvidia à partir de code OpenCL ou CUDA. Les auteurs rapportent une erreur allant de 7 à 30 pourcents sur plusieurs kernels OpenCL.

Les différentes performances évaluées par des simulateurs sont en général proches de la réalité. Cependant cette approche a plusieurs défauts majeurs. Tout d'abord, les temps de simulation sont plus importants que les temps d'exécution réels sur l'architecture cible, ce qui limite donc assez fortement le nombre de simulations que l'on peut faire en un temps limité. De plus, chacun de ces simulateurs nécessite un code source complet et déjà optimisé, on peut donc se poser la question de l'utilité d'un tel simulateur par rapport à une mesure directe sur l'architecture cible si celle-ci est disponible. Enfin, le nombre et les types d'architectures cibles restent limités, par exemple Multi2Sim ne peut cibler que les GPUs Fermi pour Nvidia, et ne supportent pas le 64 bits.

5.2.2 Temps d'exécution

Pour prédire un temps d'exécution, plusieurs approches peuvent être étudiées : la prédiction basée sur des techniques d'apprentissage ou la prédiction basée sur un modèle analytique de l'architecture et de l'algorithme.

Apprentissage

Les techniques d'apprentissages permettent de prédire un temps d'exécution en se basant sur un ensemble de *workloads* d'apprentissage et des caractéristiques hardware de l'architecture de calcul. KERR et collab. [2010] proposent une méthode pour prédire des performances sur des architectures hétérogènes composées d'un CPU et d'un GPU. L'approche est basée sur le *framework* Ocelot [DIAMOS et collab., 2010], qui permet de transformer un fichier de type *Parallel Thread Execution* (PTX) en un exécutable pour CPU. Ainsi, la méthode propose de prendre en entrée un PTX. L'apprentissage se déroule de la manière suivante :

1. Analyse en composantes principales du couple PTX du *workload* et architecture de calcul pour identifier des caractéristiques clés.
2. Partitionnement des données (*data clustering*) pour déterminer différents ensembles d'applications.
3. Régression pour construire un modèle de prédiction à partir des composantes principales.

Les auteurs montrent que leur approche permet de prédire le temps de calcul d'un *workload* sur un GPU inconnu avec une erreur inférieure à 16%.

HOSTE et collab. [2006] proposent une approche similaire, où chaque application est décrite par 47 caractéristiques indépendantes de l'architecture de calcul. Ensuite, la prédiction se fait en mesurant la similarité de l'application avec l'ensemble des *workloads* utilisés dans la phase d'apprentissage d'une architecture, en utilisant ces 47 caractéristiques. Les auteurs montrent des résultats sur différents CPUs, mais ils n'adressent pas d'autres types d'architectures comme le GPU.

BenchFriend, de CHE et SKADRON [2014], propose de prédire les temps d'exécution d'applications sur GPU en se basant sur une corrélation avec des *workloads* d'apprentissage. Cependant, le modèle ne se montre pas si précis que le précédent, les auteurs donnent une erreur de prédiction comprises entre 15,8% et 27,3%.

L'approche discutée par SATO et collab. [2011] propose une prédiction du temps de calcul au moment de l'exécution pour une répartition automatique de *kernels* OpenCL sur les différents processeurs d'une même architecture hétérogène. La méthode se base sur l'analyse des historiques des temps d'exécution d'un kernel sur différents processeurs pour prédire son temps d'exécution. Elle permet également d'identifier l'impact des paramètres d'entrée sur le temps d'exécution. Les auteurs rapportent une très bonne précision des prédictions comparée aux autres approches de l'état de l'art.

Ces différentes approches proposent des résultats intéressants, mais elles ont cependant plusieurs défauts qui nous ont poussés à ne pas les utiliser. Tout d'abord, chacune propose d'utiliser un framework en particulier, ce qui implique un travail supplémentaire pour traduire un *kernel* décrit en pseudo code dans le langage du framework en question. De plus les résultats de ces approches sont très fortement dépendants de la base d'apprentissage. Ainsi, la prédiction du temps d'exécution d'un *kernel* pour lequel il n'existe pas de *workload* similaire dans la base d'apprentissage risquera d'être plutôt mauvaise. L'approche par analyse de l'historique permet une prédiction très précise, mais elle est beaucoup trop contraignante. En effet, elle nécessite d'exécuter le *kernel* sur les différentes unités d'exécution pour modéliser son temps d'exécution. Pour ces différentes raisons, nous avons choisi de nous focaliser sur un modèle analytique pour la prédiction de performances.

Modèle analytique pour GPU

Un modèle analytique consiste à un ensemble d'équations représentant les caractéristiques de l'application et de l'architecture de calcul et permet de cette manière de prédire le temps d'exécution. Remarquons que la plupart des modèles analytique n'adressent qu'un seul type d'architecture. Par exemple le très célèbre modèle de HONG et KIM [2009] cible uniquement la prédiction de performances pour GPU. Le modèle se base sur deux métriques :

— MWP pour *Memory Warp Parallelism*, qui mesure le nombre maximum de *warps* par SMX pouvant accéder à la mémoire de manière simultanée. Cette métrique caractérise le degré de parallélisme mémoire de l'architecture, elle dépend de la bande passante mémoire, du nombre de *warps* par SMX, etc.

— CWP pour *Computation Warp Parallelism*, qui mesure le nombre de *warps* pouvant être exécutés pendant qu'un autre *warp* est en attente de données depuis la mémoire, plus 1. Cette métrique est surtout liée aux caractéristiques de l'application, un peu comme l'I_A mais de manière inverse : un grand CWP signifie peu de calculs par accès mémoire.

Les auteurs proposent plusieurs formules permettant d'exprimer ces deux métriques et d'en déduire un temps d'exécution. En fait l'idée générale du modèle est que lorsque $MWP \leqslant CWP$ la performance est limitée par le temps d'accès à la mémoire, et lorsque $MWP > CWP$ les délais d'accès à la mémoire sont cachés par le temps de calcul. Les auteurs rapportent une erreur moyenne de 13,3% entre les prédictions et les temps d'exécutions mesurés. Il s'agit un peu de la même approche que le modèle Roofline de WILLIAMS et collab. [2009] avec l'I_A.

Grâce à sa simplicité et la fidélité avec laquelle elle décrit le fonctionnement du GPU, cette approche a été reprise et améliorée par de nombreux travaux. Par exemple, GANESTAM et DOGGETT [2012] proposent d'utiliser cette méthode de prédiction de performances pour déterminer quels paramètres de leur application permet d'atteindre le *frame rate* souhaité. GROPHECY, proposé par MENG et collab. [2011], permet de prédire l'accélération que peut apporter une exécution sur GPU à partir d'une description haut niveau du type pseudo code. Ce *framework* est basé sur les métriques MWP-CWP et l'approche de HONG et KIM [2009] pour proposer une prédiction et une aide au portage sur GPU, en déterminant l'implémentation permettant d'atteindre l'accélération la plus intéressante.

Même si ces approches présentent des résultats intéressants et sont assez simples à mettre en œuvre, elles ne sont applicables que sur GPU. Elles sont donc non utilisables dans le cadre d'une architecture hétérogène incluant des CPUs et accélérateurs hardwares tels que des GPUs. La seule solution est d'envisager un modèle de prédiction analytique qui puisse adresser différents types de processeurs et prédire les temps de transfert mémoire entre ceux-ci.

5.2.3 Prédiction pour des architectures hétérogènes

Lorsque l'on parle d'accélération utilisant du parallélisme, les connaisseurs abordent immédiatement la très célèbre loi d'AMDAHL [1967]. Cette loi propose de lier l'accélération d'une tâche en termes de temps de calcul en fonction du nombre de cœurs alloués à cette tâche. Ainsi, la loi définit la relation suivante :

$$S(p) = \frac{1}{\alpha + \frac{1-\alpha}{p}} \tag{5.1}$$

où $S(p)$ correspond à l'accélération en fonction du nombre de cœurs alloués p, α correspond au pourcentage du temps d'exécution de la tâche devant s'exécuter séquentiellement et donc ne pouvant pas bénéficier d'une parallélisation. L'approche proposée par BENOIT et LOUISE [2015] utilise cette loi pour déterminer l'accélération suivant plusieurs paramètres :

— le nombre de processeurs génériques, du type CPU,

— le nombre de processeurs spécialisés, du type GPU,

— le nombre de partitions sur les données d'entrée.

Cette approche définit un modèle d'exécution différent du nôtre. Les données d'entrée sont divisées en un certain nombre de partitions, un *thread* est alors créé pour chaque partition et suit le processus d'exécution de l'application défini par un ensemble de tâches. Chaque tâche est définie comme étant soit générique, et donc exécutée sur un processeur générique, ou spécialisée, et donc exécutée sur un processeur spécialisé. Pour prédire le temps d'exécution, chaque tâche est taguée avec deux valeurs :

— un coût correspondant à son temps d'exécution sans parallélisation, obtenue par mesure ou simulation,

— le ratio de parallélisme de la tâche, correspondant à la valeur de $1 - \alpha$ dans la description de la loi d'AMDAHL [1967].

Les auteurs montrent une très bonne prédiction de l'accélération en fonction du nombre de partitions sur un système hétorogène composé d'un CPU et d'un GPU : 3,8% d'erreur. On regrettera cependant que la méthodologie utilisée ne soit pas décrite plus en détail, empêchant toute reproduction des travaux. De plus, les informations nécessaires

sur chaque tâche nécessitent des mesures assez poussées, ou l'utilisation d'un simulateur, l'approche paraît donc assez complexe et lourde à mettre en œuvre. Concernant le modèle d'exécution, nous avons choisi une autre représentation qui nous parait plus adaptée pour les raisons discutées dans la section 3.2.

Une méthodologie permettant de déterminer la répartition optimale de la quantité de données à traiter entre un CPU et un GPU a été proposée par SHEN et collab. [2014]. Cette approche est assez originale dans le sens où les auteurs prennent en compte et insistent fortement sur les délais de transfert entre le CPU et le GPU. Le modèle prend en entrée un *kernel* OpenCL, s'exécutant à la fois sur CPU et sur GPU, et détermine le pourcentage des données β devant être traitées par le GPU pour obtenir un temps d'exécution optimal. Soient $T_C(\beta)$ le temps d'exécution du kernel sur CPU, $T_G(\beta)$ le temps de calcul sur GPU et $T_D(\beta)$ le délai de transfert, la méthodologie cherche alors la valeur de β permettant d'obtenir le temps de calcul optimal, c'est-à-dire :

$$T_C(\beta) = T_G(\beta) + T_D(\beta) \tag{5.2}$$

Par la suite, en définissant :

- P_C la capacité de calcul du CPU,
- P_G la capacité de calcul du GPU,
- n la taille des données d'entrée à traiter,
- w le nombre d'opérations à effectuer pour chaque donnée,
- $O(\beta)$ la quantité de données à transférer,
- Q la bande passante du bus entre le CPU et le GPU.

On obtient :

$$T_G = \frac{w \cdot n \cdot \beta}{P_G} \qquad T_C = \frac{w \cdot n \cdot (1-\beta)}{P_C} \qquad T_D = \frac{O(\beta)}{Q} \tag{5.3}$$

En combinant 5.2 et 5.3 on en déduit :

$$\frac{\beta}{1-\beta} = \frac{P_G}{P_C} \times \frac{1}{1 + \frac{O}{w \cdot n \cdot \beta} \times \frac{P_G}{Q}} \tag{5.4}$$

Pour résoudre cette équation, les auteurs proposent d'estimer les valeurs de $\frac{P_G}{P_C}$ et $\frac{P_G}{Q}$ à partir de mesures sur les temps d'exécution du kernel sur CPU puis sur GPU. Ils montrent que ces valeurs dépendent de l'application, il est donc nécessaire d'effectuer ces mesures pour tous les *kernels*. Cette approche est très intéressante, mais le fait de devoir mesurer les temps d'exécution pour l'appliquer la rend trop contraignante pour être utilisée dans notre cas. De plus, elle s'applique à des algorithmes dont les données sont facilement séparables, ce qui n'est pas le cas de tous les algorithmes de traitement d'image. De plus, cela n'est pas compatible avec notre modèle d'exécution où nous avons choisi de définir un *kernel* comme étant insécable et exécuté par un et un seul processeur.

Le modèle Boat-Hull, de NUGTEREN et CORPORAAL [2012], est une adaptation du très célèbre modèle Roofline [WILLIAMS et collab., 2009]. Il permet de prédire le temps d'exécution d'un tâche sur CPU et GPU en se basant sur la métrique de l'I_A. La méthode de prédiction est détaillée dans la section 2.4.5. Les auteurs proposent également une prédiction des temps de transfert entre CPU et GPU en se basant sur les bandes passantes théoriques et la quantité de données à transférer. La méthode donne des résultats très intéressants, entre 3% et 8% d'erreur pour deux applications différentes. Cependant l'approche se base sur un ensemble de classes, où chaque classe est associée une équation permettant de déterminer le temps de calcul. Ainsi, la méthode nécessite un *kernel* pouvant être casé dans une des classes telles que définies dans [NUGTEREN et CORPORAAL, 2011] et ne peut pas prendre en compte du code arbitraire.

5.2.4 Discussion

En étudiant les travaux existant dans la littérature à propos de la prédiction de performances, on s'aperçoit qu'il existe très peu d'approches pouvant répondre à notre besoin. Ainsi, la majorité des approches ne cible qu'un seul type d'architecture ou exige d'effectuer une mesure pour chaque prédiction de performances. En fait, à notre connaissance, seul le modèle Boat-Hull permet de répondre à nos besoins, mais avec la contrainte de la classification qui ne lui permet pas de gérer n'importe quel type de *kernels*.

De plus, nous avons pu voir au cours du chapitre précédent que les performances d'un *kernel* dépendent fortement des types d'opérations et des types de données à traiter. Or cette dimension n'est pas du tout explorée dans les méthodes de prédiction de performances existantes dans la littérature. Nous souhaitons donc contribuer à la recherche sur la prédiction de performances en proposant un modèle prenant en compte les différents types d'opérations et de données à traiter, et qui puisse gérer du code arbitraire.

5.3 Méthodologie de prédiction de performances

Notre méthodologie de prédiction de performances doit pouvoir adresser différents types d'unités d'exécution. Comme discuté dans la section 1.3.1, une unité d'exécution de l'état de l'art est composée de plusieurs unités élémentaires, chacune étant conçue pour exécuter un certain type d'instruction. Ainsi, soient sept classes d'unités élémentaires composant une unité d'exécution :

— **ALU** : unité arithmétique et logique,
— **BMU** : unité de multiplication,
— **FPU** : unité de calcul à virgule flottante,
— **SFU** : unité de calcul spécifique,
— **BU** : unité de branche,
— **AU** : unité d'adresse,
— **LSU** : unité d'instruction mémoire.

Au moment de l'exécution, chaque unité élémentaire exécute les instructions pour lesquels elle a été conçue. Remarquons que le nombre d'unités élémentaires n'est pas le même pour l'ensemble des processeurs, par exemple certains processeurs ne disposent pas de FPU, certains GPUs disposent de SFU (notamment pour le calcul trigonométrique), ou encore l'ARM A15 qui dispose de deux ALUs par cœur.

Comme discuté dans la section 2.4.3 et illustré par WILLIAMS et collab. [2009] et NUGTEREN et CORPORAAL [2012], le temps d'exécution d'un *kernel* peut être soit limité par les capacités de calcul soit par la bande passante et latence mémoire de l'architecture sur laquelle il est exécuté. Ainsi, nous choisissons donc de différencier la prédiction du temps de calcul et la prédiction des délais d'accès à la mémoire.

5.3.1 Prédiction du temps de calcul

Dans [SAUSSARD et collab., 2015b,c], nous présentons une méthodologie pour prédire le temps de calcul d'un *kernel* sur différentes unités d'exécution. Nous nous intéressons ici uniquement à l'aspect calcul, donc soit $\mathbb{U}_c$ l'ensemble des unités élémentaires impliquant des instructions de calcul (pas de LSU) :

$$\mathbb{U}_c = \{ALU, BMU, FPU, SFU, BU, AU\} \tag{5.5}$$

Comme vu dans le chapitre précédent, les capacités de calcul d'une unité élémentaire peut dépendre du type de données à traiter. Ainsi, nous définissons $\mathbb{S}$, l'ensemble des types de données :

$$\mathbb{S} = \{int8, int16, int32, int64, float32, float64, \ldots\} \tag{5.6}$$

Soit $N_c(s)$ le nombre d'instructions associées à l'unité élémentaire $c \in \mathbb{U}_c$ pour le type de données $s \in \mathbb{S}$. Soit $p_{c,a}(s)$ la capacité de calcul en nombre d'opérations par cycle de l'unité élémentaire c de l'unité d'exécution a pour le type de données s. Le temps de calcul $t_{c,a}$ de l'unité élémentaire c de l'unité d'exécution a sera donc donné par la somme sur l'ensemble des types de données du rapport nombre d'opérations sur la capacité de traitement associée :

$$t_{c,a} = \sum_{s \in \mathbb{S}} \frac{N_c(s)}{p_{c,a}(s)} \tag{5.7}$$

Un processeur est dit superscalaire s'il est capable d'utiliser simultanément plusieurs unités élémentaires d'une même unité d'exécution. Ainsi, une unité d'exécution d'un processeur de ce type pourra donc exécuter plusieurs opérations simultanément si celles-ci sont indépendantes et si elles nécessitent des unités élémentaires différentes (ou si plusieurs unités élémentaires du même type sont disponibles sur la même unité d'exécution). Donc, dans le cas idéal où tous les types d'instructions s'exécutent de manière simultanée, le temps d'exécution t_{min} (en cycles) sera donné par le maximum des $t_{c,a}$ sur l'ensemble des types d'instructions :

$$t_{min} = \max_{\{c \in \mathbb{U}_c\}} t_{c,a} \tag{5.8}$$

En pratique ce cas n'arrive pratiquement jamais, d'ailleurs les processeurs sont superscalaires à un certain degré c'est-à-dire qu'ils ne sont capable d'exécuter de manière simultanée qu'un certain nombre d'instructions (par exemple l'ARM A15 est superscalaire de degré trois, ce qui signifie qu'une unité d'exécution peut exécuter au maximum trois opérations indépendantes de manière simultanée). Dans le pire cas, c'est-à-dire lorsqu'il n'est pas possible de tirer parti de la propriété superscalaire (par exemple lorsque toutes les opérations sont dépendantes les unes des autres) le temps d'exécution t_{max} est alors égal à la somme des $t_{c,a}$ pour l'ensemble des types d'instructions :

$$t_{max} = \sum_{c \in \mathbb{U}_c} t_{c,a} \tag{5.9}$$

Dans la réalité, le temps d'exécution réel t_a sera compris entre ces deux extrêmes. On peut donc en conclure que le temps d'exécution est donné par l'intervalle suivant :

$$\begin{aligned} t_a &= [t_{min}, t_{max}] \\ &= \left[\max_{\{c \in \mathbb{U}_c\}} t_{c,a}, \quad \sum_{c \in \mathbb{U}_c} t_{c,a} \right] \end{aligned} \tag{5.10}$$

Ainsi, pour un *kernel* et une unité d'exécution donnés, notre méthodologie de prédiction de performances est capable de prédire un intervalle du temps d'exécution réel. Le modèle repose sur les capacités de calcul $p_{c,a}(s)$, informations qui peuvent être obtenues depuis une datasheet ou par le biais de nos vecteurs de test comme discuté lors du chapitre précédent. Le *kernel* est quant à lui modélisé par le nombre de calculs qu'il doit effectuer pour chaque type d'instructions et chaque type de données. Cette information peut être facilement obtenue à partir d'une implémentation générique du *kernel* du type pseudo code ou même en C sans utiliser d'instructions spécifiques à une architecture.

5.3.2 Prédiction des délais d'accès à la mémoire

D'après WILLIAMS et collab. [2009] et NUGTEREN et CORPORAAL [2012] la performance d'un *kernel* ayant une faible I_A est limitée par la bande passante et la latence mémoire de l'architecture sur laquelle il s'exécute. Le modèle de prédiction de performances doit être capable de prendre en compte ce genre de comportement. De plus, nous avons également besoin d'être capables de prédire les temps de transfert entre les différents processeurs d'un même SoC.

Comme discuté dans la section 1.3, la mémoire est organisée suivant plusieurs niveaux (registres, caches, RAM, etc.), chacun ayant des latences et bandes passantes différentes. Or lorsque l'on accède à une donnée (en lecture ou en écriture), il est très complexe de prédire depuis quel niveau de mémoire celle-ci sera chargée et encore plus complexe si l'on cherche à prédire le délai d'accès. Pour garder une approche simple et générique, nous avons choisi de baser la prédiction des délais d'accès à la mémoire directement à partir des résultats issus des vecteurs de test low-level en rapport avec la mémoire. Ainsi, à partir des données brutes issues de ces tests, nous procédons à une régression linéaire par morceaux, nous obtenons de la sorte un délai en fonction d'une taille de donnée. Un exemple de régression linéaire par morceaux est donné en figure 5.1. Remarquons que cette interpolation linéaire n'est pas toujours possible, par exemple sur les résultats des tests ReadAfterWrite. Dans ce cas, nous considérerons des interpolations par des polynômes dont le degré varie en fonction des cas.

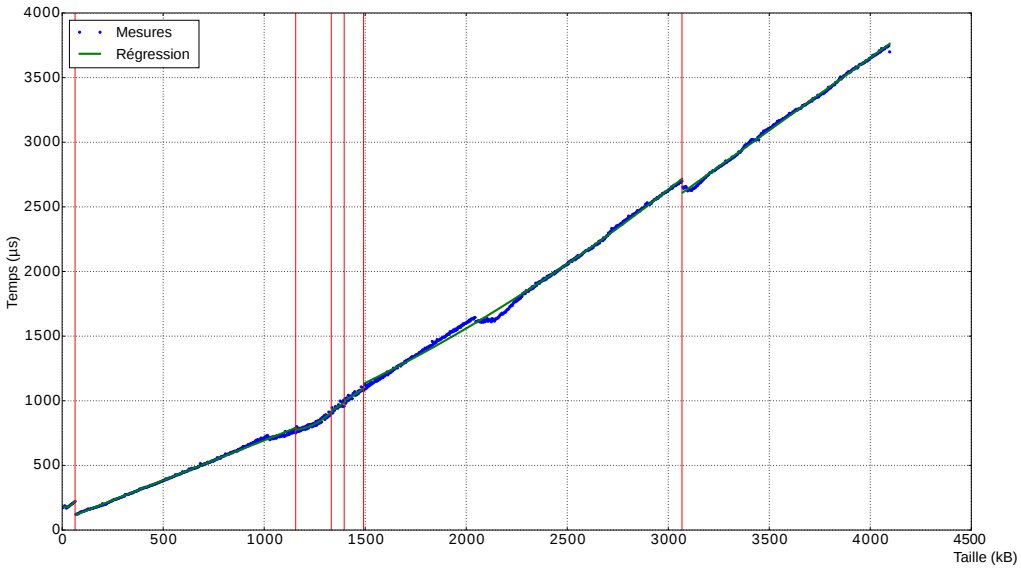


FIGURE 5.1 – Régression linéaire par morceaux sur des données issues du test de transfert sur Tegra K1. Les lignes verticales rouges indiquent les changements de morceaux. On remarque que l'interpolation (en vert) est très proche des données réelles (en bleu), excepté autour de 2048 kB où la cassure n'est pas détectée.

Nous sommes donc capables de prédire un délai en connaissant les types et le nombre d'accès à la mémoire simplement en appliquant les fonctions linéaires par morceaux définies précédemment. Cependant, déterminer le type et le nombre d'accès n'est pas toujours évident, il parait important d'ajouter quelques précisions sur le dénombrement et la classification de ces accès. De plus notre méthodologie nécessite un intervalle et non la valeur moyenne d'une mesure. En fait, seuls les accès à la mémoire globale doivent être

pris en compte, et non les accès locaux (variables locales ou temporaires, ou encore l'utilisation de mémoires locales, comme la *shared memory* pour le GPU), on s'intéresse donc uniquement aux entrées et sorties du *kernel*. Ainsi, soient :

— R_i : le nombre total de lectures sur les données d'entrée, correspondant à un temps t_{r_i} ;

— R_p : le nombre total de lectures sur des paramètres, correspondant à un temps t_{r_p} ;

— W_o : le nombre total d'écritures sur la sortie, correspondant à un temps t_{w_o}.

L'ensemble des délais mémoire est directement obtenu à partir des résultats issus des vecteurs de test *low-level*. Pour la prédiction, nous allons nous baser sur l'hypothèse suivante : la lecture et l'écriture peuvent se superposer. Concrètement, cela signifie que la mémoire peut être accédée simultanément en lecture et en écriture sur deux zones mémoire distinctes. Dans le cas du GPU, cela se manifeste par le fait que certains *threads* peuvent charger des données depuis la mémoire, tandis que d'autres sont en train d'écrire sur une autre zone mémoire. Sur les processeurs ARM, et les CPUs RISC de manière générale, les unités *load* et *store* sont très souvent distinctes et peuvent également s'utiliser de manière simultanée. Ainsi, le délai des accès à la mémoire du *kernel* sera donc compris entre le temps de lecture et le temps de lecture plus le temps d'écriture :

$$[t_{r_i} + t_{r_p}, \quad t_{r_i} + t_{r_p} + t_{w_o}] \tag{5.11}$$

Remarquons que dans le cas du GPU, il faut également prendre en compte la latence de lancement du *kernel*, et il ne faut la prendre en compte qu'une seule fois. Or, pour l'ensemble des mesures issues des tests mémoire, nous obtenons un temps correspondant aux délais d'accès à la mémoire plus la latence de lancement du *kernel*, il est donc nécessaire de ramener l'ensemble des mesures à une ordonnée à l'origine nulle, puis de rajouter la valeur de la latence λ pour obtenir l'intervalle de prédiction :

$$[\lambda + t_{r_i} + t_{r_p}, \quad \lambda + t_{r_i} + t_{r_p} + t_{w_o}] \tag{5.12}$$

De manière naïve, on pourrait penser que la valeur de la latence λ peut être obtenue tout simplement par l'ordonnée à l'origine de n'importe quel test. Or, cette latence peut varier, il serait d'ailleurs très intéressant de compléter notre collection de vecteurs de test *low-level* par une caractérisation de la latence de lancement d'un kernel sur GPU. Malheureusement, faute de temps cette piste n'a pas été suffisamment explorée au cours de la thèse pour que nous puissions présenter des résultats dans ce manuscrit.

L'hypothèse de base concernant la superposition des accès en lecture et écriture est en fait issue de plusieurs observations, comme montré en figure 5.2. Ainsi, dans cette figure, nous donnons pour l'ensemble des processeurs :

— le temps d'exécution d'un *kernel* effectuant tout simplement une copie entre les données d'entrée et la sortie dans la configuration du test McpySlow,

— le temps de lecture tel que prédit par le test correspondant (test ReadAfterWrite),

— le temps de lecture plus le temps d'écriture tels que prédits par les tests correspondant (tests ReadAfterWrite et WriteAfterWrite).

Nous pouvons observer que pour l'ensemble des processeurs testés le temps d'exécution du *kernel* de copie se situe toujours à l'intérieur de l'intervalle prédit, excepté pour l'ARM A15 sur R-Car H2 où le délai de copie dépasse nos prédictions lorsque la taille des données à copier devient supérieure à 3250 kB. Ce phénomène n'étant pas présent sur Tegra K1 et Tegra X1, nous pensons que cela est causé par une politique de cache particulière au R-Car H2. Pour corriger nos prédictions sur cette plateforme avec des tailles de données

supérieures à 3250 kB, conformément à notre méthodologie de caractérisation *high-level* décrite dans la section , une investigation plus en profondeur et un affinement du modèle doivent être envisagés.

Comme nous l'avons montré dans le chapitre précédent, nous caractérisons les opérations de lecture et d'écriture dans trois configurations différentes, chaque configuration ayant ses propres comportements. Remarquons que pour la lecture, il existe également une quatrième configuration : la lecture de données écrites par un autre processeur. On en vient donc à se poser la question suivante : quelles configurations de lecture et d'écriture utilisées pour la détermination de t_{r_i}, t_{r_p} et t_{w_o} ? La réponse est loin d'être triviale et il est très complexe de déterminer un couple de configuration lecture/écriture qui puisse fonctionner dans tous les cas. Pour les prédictions présentées dans la figure 5.2, nous avons choisi les configurations en rapport avec la définition du test de copie : le test écrit sur les données d'entrée et de sortie avant de procéder à la copie, il convient donc d'utiliser les configurations ReadAfterWrite et WriteAfterWrite pour la prédiction. Généralement, pour obtenir des prédictions précises, le choix doit s'effectuer au cas par cas en se basant sur des hypothèses de contexte permettant de déterminer la configuration à utiliser pour chaque accès.

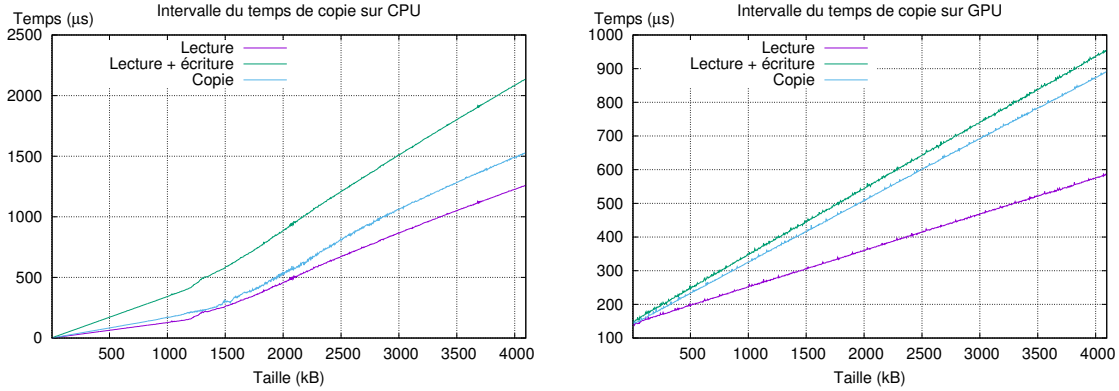
Cependant, si l'on suit la représentation en graphe de traitement telle que définie dans la section 3.2.2, alors il est possible d'effectuer les constats suivants :

— La lecture sur les données d'entrée d'un *kernel* k_i se fait forcément dans la configuration où ces données ont été précédemment écrites par un ou plusieurs *kernels* en amont de k_i. Ainsi, avoir des données d'entrée pour un *kernel* qui ne sont jamais écrites par aucun processeur parait contre intuitif et inutile. Deux cas sont alors possibles, soit les données ont été écrites par le même processeur, soit elles ont été écrites par un autre processeur. En fonction du *mapping*, nous en déduisons donc que la prédiction du t_{r_i} doit s'appuyer sur le test ReadAfterWrite pour les lectures sur des données issues d'un *kernel* s'exécutant sur un même processeur et ReadAfterTransfer pour les lectures sur des données issues d'un transfert données (*kernel* en amont exécuté sur un autre processeur).

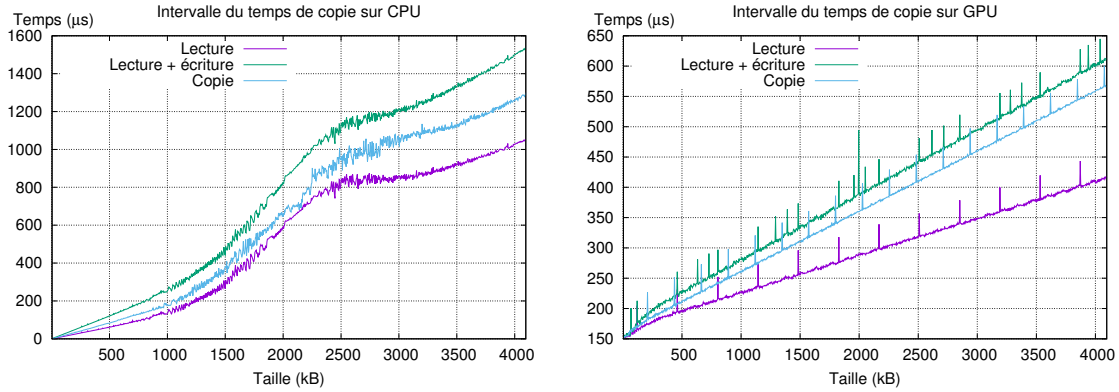
— L'écriture sur les données de sortie d'un *kernel* k_i se fait forcément dans la configuration où ces données ont été soit lues par un *kernel* en aval de k_i, soit écrites par l'instance précédente de k_i, on peut donc supposer qu'elles sont probablement dans le cache. Cela élimine donc la configuration Write, il reste donc les configurations WriteAfterWrite et WriteAfterRead. D'après les résultats présentés dans le chapitre précédent, ces deux configurations ont des comportements assez similaires, mais avec un délai légèrement plus grand pour la configuration WriteAfterRead. Généralement, nous choisissons d'utiliser la configuration WriteAfterRead pour la prédiction du t_{w_o}.

— L'accès à des paramètres se fait uniquement en lecture et on peut considérer que ceux-ci sont lus suffisamment régulièrement pour être dans le cache, cela exclut donc la configuration Read. Finalement deux stratégies peuvent être abordées : soit on considère le meilleur cas : la configuration ReadAfterRead, soit on considère le pire cas : la configuration ReadAfterWrite.

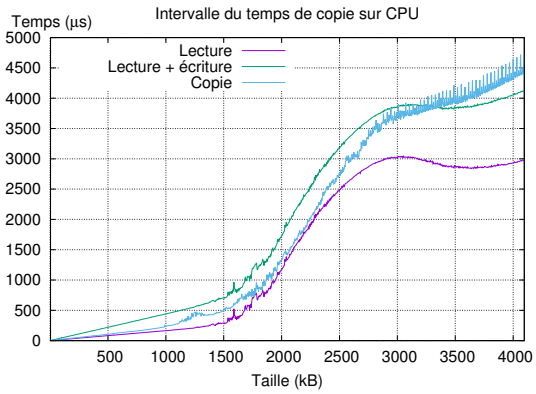
Remarquons que nous ne considérons ici que des accès contigus, pour la lecture et l'écriture. Dans le cas d'accès non contigus (par exemple des accès aléatoires), les prédictions des temps t_{r_i}, t_{r_p} et t_{w_o} doivent s'appuyer sur des tests qui correspondent aux types d'accès utilisés dans le *kernel*. Il peut s'agir par exemple des tests de lecture non contigus présentés dans les figures 4.12 et 4.13.



(a) K1



(b) X1



(c) R-Car H2

FIGURE 5.2 – Temps d'exécution d'une copie dans la configuration du test McpySlow, les prédictions (temps de lecture et écriture) sont basées sur les tests ReadAfterWrite et WriteAfterWrite. On remarque que le temps de copie se situe toujours dans l'intervalle de prédiction, excepté pour R-Car H2 où le délai de copie dépasse le temps de lecture plus écriture pour des tailles supérieures à 3250 kB.

Concernant les temps de transfert, nous utilisons tout simplement les données issues du test correspondant. Ainsi, pour une quantité de données à transférer, le temps de transfert est obtenu par l'intervalle correspondant à la mesure minimale et la mesure maximale obtenues par le test correspondant :

$$[Transfert_{min}, \quad Transfert_{max}] \tag{5.13}$$

5.3.3 Spécificités du compilateur et des processeurs

Comme nous l'avons montré dans la section 4.3.2 le compilateur a un impact non négligeable sur le temps de calcul d'un *kernel*. Par exemple, le compilateur GCC peut optimiser le temps d'exécution d'un *kernel* via l'option *-O3*. Les optimisations du type *-O* ne dépendent pas de l'architecture, contrairement aux options *-m* qui permettent d'activer des optimisations spécifiques à un processeur. Pour prédire l'impact du compilateur sur les performances, nous nous appuyons sur les résultats issus de nos vecteurs de test *low-level*. Ainsi, les vecteurs de test sont compilés puis exécutés avec un compilateur et des options donnés, ils permettent alors de caractériser les performances d'une architecture dans cette configuration.

L'un des exemples typiques est l'auto-vectorisation effectuée par le compilateur : dans certains cas, le compilateur se permet de modifier le code de l'utilisateur pour que celui-ci utilise les instructions SIMD d'un processeur et accélère donc le temps d'exécution. Dans le cas de l'ARM A15 et du compilateur GCC, cette fonctionnalité s'active à l'aide de l'option *-mfpu=neon-vfp4*. Sur l'ARM A57 elle est activée par défaut. Nous cherchons donc à prédire l'impact sur le temps d'exécution que peut avoir le compilateur lorsque l'auto-vectorisation est activée.

Pour illustrer cet impact, nous donnons en figure 5.3 le temps de calcul de l'addition et de la multiplication en fonction du nombre d'opérations par donnée d'entrée (ou complexité) lorsque l'auto-vectorisation est activée (avec l'option de compilation *-mfpu=neon-vfp4*) et lorsqu'elle est désactivée pour des *int8* et *int32*. Il s'agit tout simplement des mêmes résultats issus des vecteurs de test *low-level* de calcul, mais où le temps d'exécution est représenté en fonction de la complexité au lieu du nombre d'opérations par octet lu. Remarquons que dans le cas de variables du type *int8*, 1 opération par donnée d'entrée est équivalente à 1 opération par octet lu, il s'agit donc de la même courbe que celle présentée dans le chapitre précédent (figure 4.6). Or, pour des *int32* 1 opération par donnée d'entrée équivaut à 0,25 opérations par octet lu, on a donc un effet de facteur d'échelle par rapport à la figure 4.5, présentée dans le chapitre précédent.

On remarque alors que lorsque le nombre d'opérations par donnée d'entrée devient supérieur à 16, le temps de calcul se comporte exactement de la même manière avec ou sans auto-vectorisation pour chaque type de données. Or, pour une complexité inférieure ou égale à 16, on observe un temps de calcul plus rapide lorsque l'auto-vectorisation est activée pour chaque type de données. Les additions sont environs 8 fois plus rapides sur des *int8* avec l'auto-vectorisation et 2 fois plus rapides avec l'auto-vectorisation sur des *int32*. Nous avons identifié ici, grâce à nos vecteurs de test, le comportement du compilateur pour l'auto-vectorisation. Ainsi quel que soit le type de données :

— si le nombre d'opérations par donnée d'entrée est inférieur ou égal à 16 alors les opérations sont vectorisées,

— si le nombre d'opérations par donnée d'entrée est supérieur à 16 alors les opérations ne sont pas vectorisées.

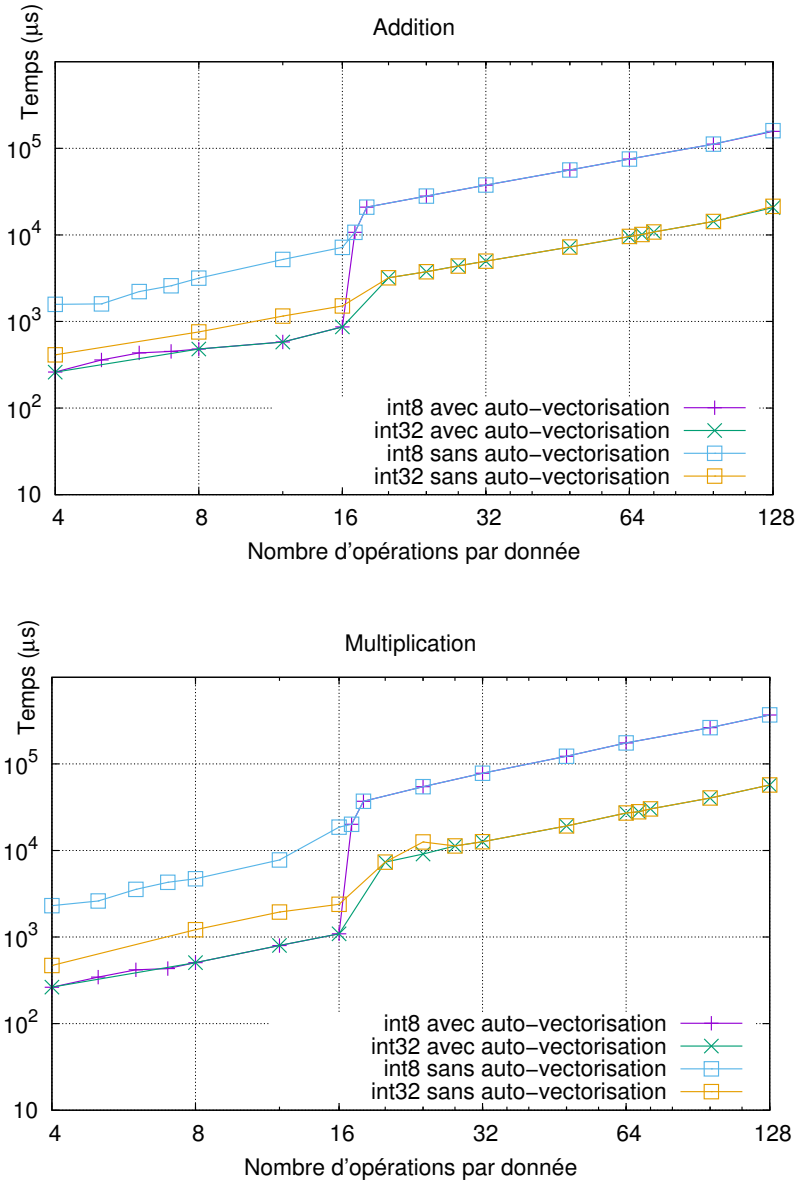


FIGURE 5.3 – Impact de l'auto-vectorisation du compilateur GCC sur le processeur A15 du Tegra K1.

Bien évidemment ce raisonnement n'est valide que si le nombre de données d'entrée est suffisamment grand, c'est-à-dire supérieur à la taille d'un registre SIMD.

Concernant la prédiction de performances impliquant des spécificités matérielles, nous proposons d'appuyer les prédictions sur des tests *low-level* spécialement conçus pour mettre en œuvre la fonctionnalité à caractériser. Par exemple, la caractérisation des opérations atomiques du GPU *AtomicAdd* a été présentée dans le chapitre précédent (figure 4.24). Ainsi, en connaissant le nombre de conflits potentiels, par exemple en utilisant l'entropie de Shannon, il est possible de prédire le délai causé par l'utilisation de cette ressource matérielle. La même approche peut se généraliser à d'autres fonctionnalités hardwares, comme les unités de texture des GPUs.

5.4 Résultats

Pour illustrer notre méthodologie de prédictions de performances et démontrer sa pertinence, nous proposons de l'appliquer sur un *kernel* typique pouvant être rencontré dans une application de traitement d'images : le calcul du gradient horizontal et vertical de l'image. Pour comparer nos résultats avec des travaux existants, nous proposons également de donner les prédictions obtenues par le modèle Boat-Hull. Nous considérerons le cas d'une image en niveau de gris, de résolution 1280×720, et à titre d'exemple l'architecture étudiée sera le Tegra K1. Les raisonnements qui vont suivre peuvent s'appliquer aussi bien dans le cas d'images de natures différentes, de tailles différentes et sur des architectures différentes (mais préalablement caractérisées).

Les *kernels* du calcul du gradient horizontal et vertical de l'image sont décrits dans les algorithmes 4 et 5. Dans les deux cas, les *kernels* prennent en entrée des *int8* non signés et retournent des *int16* signées. Au niveau algorithmique, il n'y a pas de grande subtilité, mise à part la taille des boucles *for* pour prendre en compte les effets du bord de l'image et éviter des erreurs de segmentation dues à des accès en dehors de l'image.

Algorithme 4 : Gradient horizontal de l'image.

Data : I_{in}
Result : I_{out}
1 **for** $j = 0; j < 720 \times 1280; j += 1280$ **do**
2 // Boucle j
3 **for** $i = 1; i < 1279; i++$ **do**
4 // Boucle i
5 $I_{out}[i+j] = I_{in}[i+j-1] - I_{in}[i+j+1];$

Algorithme 5 : Gradient vertical de l'image.

Data : I_{in}
Result : I_{out}
1 **for** $j = 1280; j < 719 \times 1280; j += 1280$ **do**
2 // Boucle j
3 **for** $i = 0; i < 1280; i++$ **do**
4 // Boucle i
5 $I_{out}[i+j] = I_{in}[i+j-1280] - I_{in}[i+j+1280];$

5.4.1 Analyse

Intéressons-nous tout d'abord au cas du gradient horizontal. Dans la boucle i, le *kernel* effectue 3 opérations AU, représentées en vert et 1 opération ALU sur *int8*, représentée en rouge. Remarquons que nous ne comptons pas les opérations entre « [] », celles-ci étant incluses dans les calculs des adresses. La boucle i représente 1 opération BU, correspondant au test de la boucle *for* et 2 opérations ALU sur *int32*, représentées en orange. La boucle i est parcourue $720 \times 1278 = 920160$ fois. De la même manière, la boucle j représente 1 opération BU et 2 opérations ALU sur *int32* mais elle est parcourue seulement 720 fois. Finalement, le nombre d'opérations du *kernel* de gradient horizontal pour chaque classe :

— **ALU *int32*** $\Leftarrow 2 \times 920160 + 2 \times 720 = 1\,841\,760$

— **ALU *int8*** $\Leftarrow 1 \times 920160 = 920\,160$

— **BU** $\Leftarrow 1 \times 920160 + 1 \times 720 = 920\,880$

— **AU** $\Leftarrow 3 \times 920160 = 2\,760\,480$

Le même raisonnement peut s'appliquer sur le *kernel* du gradient vertical, mais avec les différences suivantes : la boucle j est parcourue 718 fois et boucle i est parcourue $718 \times 1280 = 919\,040$. On obtient alors pour ce *kernel* :

— **ALU *int32*** $\Leftarrow 2 \times 919040 + 2 \times 718 = 1\,839\,516$

— **ALU *int8*** $\Leftarrow 1 \times 919040 = 919\,040$

— **BU** $\Leftarrow 1 \times 919040 + 1 \times 718 = 919\,758$

— **AU** $\Leftarrow 3 \times 919040 = 2\,757\,120$

Remarquons que l'on ne compte pas la multiplication de la boucle j puisque le résultat peut être connu à la compilation, le compilateur se chargera donc de remplacer cette opération par le résultat.

5.4.2 Prédiction sur CPU

Concernant le processeur ARM, chaque pixel de l'image d'entrée est lu deux fois, on a donc un total de 1800 kB de lecture et 1800 kB d'écriture. En se référant aux résultats des vecteurs de test *low-level* (ReadAfterWrite et WriteAfterRead), on obtient la prédiction suivante :

$$t_{mémoire} = [0.37, \quad 0.80]\, ms \tag{5.14}$$

Un cœur

Du point de vue du calcul, si l'on prend les capacités de calcul suivante :

— **ALU** sur *int32* : 2 opérations par cycle

— **ALU** sur *int8* : 1 opération par cycle

— **BU** : 1 opération par cycle

— **AU** : 2 opérations par cycle

— Fréquence d'horloge : 1800 MHz

on obtient alors les prédictions suivantes pour le temps de calcul :

— pour le gradient horizontal :

$$t_{calcul} = \left[\frac{1841760}{2} + 920160, \quad \frac{1841760}{2} + 920160 + 920880 + \frac{2760480}{2} \right] \times \frac{1}{1800000}$$
$$= [1.02, \quad 2.30]\, ms \tag{5.15}$$

— pour le gradient vertical :

$$t_{calcul} = \left[\frac{1839516}{2} + 919040, \quad \frac{1839516}{2} + 919040 + 919758 + \frac{2757120}{2} \right] \times \frac{1}{1800000}$$
$$= [1.02, \quad 2.30]\, ms \tag{5.16}$$

La prédiction du temps de calcul est supérieure à la prédiction de la mémoire (équation 5.4.2), on est donc en condition de forte I_A, et donc limité par les capacités de calcul.

À l'intérieur de la boucle i, on remarque que l'on effectue uniquement 3 opérations AU et 1 opération ALU sur des *int8*. La complexité à l'intérieur de cette boucle étant inférieure à 16, on se situe donc dans la condition où le compilateur GCC effectue une vectorisation si l'option de compilation *-mfpu=neon-vfp4* est activée. La boucle i sera donc vectorisée.

Un cœur SIMD

Intéressons-nous à la performance de ces *kernels* avec l'utilisation de l'unité SIMD, que ce soit l'opération du compilateur qui auto-vectorise ou une implémentation à la main à l'aide des intrinsèques NEON. Ainsi, il est possible à l'aide des intrinsèques NEON de calculer la différence entre deux vecteurs de 8 *int8* et de stocker le résultat sur un vecteur de 8 *int16*. La prédiction de calcul est très simple, on remplace la capacité de calcul de l'ALU sur *int8* par la capacité de calcul NEON sur des *int16*. D'après nos mesures, la soustraction en NEON sur un vecteur de 8 *int16* a une capacité de traitement de 6 données par cycle, soit $\frac{6}{8} = 0{,}75$ vecteurs de 8 *int16* traités par cycle. Puis on divise le nombre d'itérations dans la boucle i par 8. Pour le gradient horizontal, le nombre d'opérations devient alors :

— **ALU *int32*** $\Leftarrow 2 \times 920160/8 + 2 \times 720 = 231480$
— **ALU *int8*** $\Leftarrow 1 \times 920160/8 = 115020$
— **BU** $\Leftarrow 1 \times 920160/8 + 1 \times 720 = 115740$
— **AU** $\Leftarrow 3 \times 920160/8 = 345060$

et pour le gradient vertical :

— **ALU *int32*** $\Leftarrow 2 \times 919040/8 + 2 \times 718 = 231196$
— **ALU *int8*** $\Leftarrow 1 \times 919040/8 = 114880$
— **BU** $\Leftarrow 1 \times 919040/8 + 1 \times 718 = 115598$
— **AU** $\Leftarrow 3 \times 919040/8 = 344640$

Finalement, si l'on compile avec l'option *-mfpu=neon-vfp4* (auto-vectorisation activée) les prédictions sont :

— pour le gradient horizontal

$$t_{calcul} = \left[\frac{231480}{2} + \frac{115020}{0.75}, \quad \frac{231480}{2} + \frac{115020}{0.75} + 115740 + \frac{345060}{2} \right] \times \frac{1}{1800000}$$
$$= [0.15, \quad 0.31]\, ms \tag{5.17}$$

— pour le gradient vertical

$$t_{calcul} = \left[\frac{231480}{2} + \frac{115020}{0.75}, \quad \frac{231196}{2} + \frac{114880}{0.75} + 115598 + \frac{344640}{2}\right] \times \frac{1}{1800000}$$
$$= [0.15, \quad 0.31]\,ms \tag{5.18}$$

Dans cette configuration, on obtient une prédiction qui est inférieure au temps d'accès à la mémoire (équation 5.4.2). On se situe donc dans une configuration de faible I_A, avec une performance limitée par la capacité de la mémoire :

$$t_{mémoire} = [0.37, \quad 0.80]\,ms \tag{5.19}$$

Quatre cœurs

Intéressons-nous maintenant à une configuration utilisant 4 cœurs. On se propose de paralléliser les deux *kernels* à l'aide d'une parallélisation *parallel for* d'OpenMP sur la boucle j. OpenMP va alors séparer la boucle j en quatre sous-*kernels*, où chaque sous-*kernels* traitera un quart de l'image et sera exécuté sur un cœur du processeur. On se trouve ici dans un cas très simple de catégorie de parallélisme : il n'y a pas de conflit entre les *threads*. En théorie la prédiction est donc très aisée, il suffit de diviser le nombre de calculs par quatre. Cependant, il convient de s'appuyer sur nos vecteurs de test *mid-level* pour vérifier si l'exécution en parallèle des quatre *threads* peut avoir un impact sur leurs temps d'exécution. Dans le cas où nous n'activons pas l'accélération SIMD, nous sommes dans la zone de forte I_A, limitée par la capacité de calcul. Or, nos vecteurs de test ont montré que dans cette zone, les exécutions concurrentes sur des cœurs différents n'ont pas d'impact sur le temps d'exécution de chaque *threads*. Dans cette configuration, on a donc les prédictions suivantes :

— pour le gradient horizontal

$$t_{calcul} = [1.02, \quad 2.30] \times \frac{1}{4} = [0.26, \quad 0.58]\,ms \tag{5.20}$$

— pour le gradient vertical

$$t_{calcul} = [1.02, \quad 2.30] \times \frac{1}{4} = [0.26, \quad 0.58]\,ms \tag{5.21}$$

Quatre cœurs SIMD

Dans le cas où l'accélération SIMD est activée, l'hypothèse de forte I_A n'est plus valide. Si l'on souhaite prédire le temps de calcul des sous-*kernels* dans la configuration où l'on utilise 4 cœurs et l'accélération SIMD, il faut alors appuyer les prédictions sur les mesures d'accès mémoire concurrents obtenues par les vecteurs de test *mid-level*. Ainsi, dans cette configuration chaque sous-*kernel* effectue 1800/4 = 450 kB de lectures et 1800/4 = 450 kB d'écritures. En s'appuyant sur nos tests *mid-level* mettant en œuvre les accès mémoire ReadAfterWrite et WriteAfterRead dans la configuration 4 cœurs, on obtient la prédiction suivante :

$$t_{mémoire} = [0.10, \quad 0.22]\,ms \tag{5.22}$$

5.4.3 Prédiction sur GPU

Intéressons-nous maintenant à la prédiction des performances de ces *kernels* sur le GPU du Tegra K1. D'après les résultats de nos vecteurs de test *low-level*, nous savons que l'I_A du *kernel* s'exécutant sur GPU doit être très élevée pour que le temps d'exécution soit limité par les capacités de calcul. L'I_A de ces *kernels* est assez faible, on considérera donc que les performances sont limitées par les capacités de la mémoire. En partant de cette hypothèse et en sachant que chaque *kernel* doit charger 800 kB de données sous la forme d'*int8* et sauvegarder 1900 kB sous la forme de *int16* la prédiction devient triviale, il suffit d'utiliser les résultats des vecteurs de test mémoire et l'équation 5.3.2. En supposant que la latence de lancement $\lambda = 50\,\mu s$, on obtient alors :

$$t_{mémoire} = [0.42, \quad 0.75]\,ms \tag{5.23}$$

Dans la pratique, l'implémentation de ces *kernels* implique plusieurs mécanismes spécifiques au GPU. Ainsi les données sont préalablement chargées dans la *shared memory* avant d'être utilisées pour le calcul du gradient. Une attention toute particulière doit être apportée à la gestion des bords des blocs, puisque chaque *thread* doit pouvoir accéder au pixel voisin, or la *shared memory* n'est accessible que pour les *threads* du même bloc. Ainsi, si la prédiction parait très simple, l'implémentation est quant à elle loin d'être aisée et triviale. On comprend donc tout l'intérêt de la prédiction de performance, qui est rapide à mettre en œuvre, comparée à l'implémentation et la mesure de la performance réelle, qui est longue et fastidieuse.

Remarquons qu'une exécution sur GPU d'un des deux *kernels* peut impliquer, dans certains cas, un transfert $CPU \rightarrow GPU$ et $GPU \rightarrow CPU$ (en pratique cela dépend du *mapping* des *kernels* en amont et en aval). Pour prédire ces temps de transfert, il suffit d'appliquer les résultats du test correspondant. Dans le sens CPU vers GPU, 900 kB sont transférés, alors que dans le sens GPU vers CPU 1800 kB sont transférés. On obtient alors les prédictions suivantes :

$$t_{CPU \rightarrow GPU} = [0.59, \quad 1.02]\,ms \tag{5.24}$$

$$t_{GPU \rightarrow CPU} = [1.68, \quad 3.13]\,ms \tag{5.25}$$

Si l'on prend en compte les temps de transfert, on obtient alors une latence pour l'exécution du *kernel* de gradient horizontal sur GPU donnée par l'intervalle :

$$\begin{aligned} t_p &= t_{CPU \rightarrow GPU} + \tau_{GradHorizontal} + t_{GPU \rightarrow CPU} \\ &= [0.59, \quad 1.02] + [0.42, \quad 0.75] + [1.68, \quad 3.13] \\ &= [2.69, \quad 4.90]\,ms \end{aligned} \tag{5.26}$$

On est donc clairement au-dessus du temps d'exécution prédit pour l'ARM A15 monocœur sans SIMD, l'utilisation du GPU n'est pas toujours bénéfique.

5.4.4 Comparaison avec le modèle Boat-Hull

Avant de présenter les mesures des temps d'exécution réels de ces deux *kernels* dans les différentes configurations, on se propose de comparer nos prédictions avec l'approche de prédiction de performances proposée par le modèle Boat-Hull [NUGTEREN et CORPORAAL, 2012]. Le fonctionnement du modèle est présenté dans la section 2.4.5. Pour nos

deux *kernels* et en accord avec le modèle nous choisissons d'utiliser les paramètres suivants :

$$\begin{cases} w = & 1280 \times 720 = 921600 \\ \alpha = & 2 \\ o/w = & 64 \text{ (uniquement pour le GPU)} \\ d = & 2 \times 1280 \times 720 = 1843200 \\ \gamma = & d \\ u = & 0 \end{cases} \tag{5.27}$$

Du point de vue du CPU, le modèle nécessite la capacité de calcul théorique $P_{compute}$ (en GFLOPS), la bande passante mémoire pour des accès contigus $P_{contigu}$ (en GB/s). D'après les résultats de nos vecteurs de test, un FPU de l'ARM A15 est capable d'effectuer 0.2 opérations à virgule flottante par cycle. L'ARM A15 dispose de 4 cœurs composés de chacun de deux FPUs et il dispose également de registres SIMD d'une taille de 128 bits (soit 4 variables *float32* par registre SIMD), on a donc :

$$P_{compute} = 0.2 \times 2 \times 4 \times 4 \times 1.8 \times 10^9 = 11.52 \text{ GFLOPS} \tag{5.28}$$

Pour le GPU, l'architecture Kepler est théoriquement capable d'effectuer 192 opérations à virgule flottante par cycle, soit :

$$P_{compute} = 192 \times 0.78 \times 10^9 = 149.76 \text{ GFLOPS} \tag{5.29}$$

Pour la bande passante mémoire $P_{contiguë}$, on se propose d'utiliser un délai mémoire de 0,1 ns par octet (conformément aux résultats présentés dans la figure 4.14), soit une bande passante de 10 GB/s.

Finalement, on obtient les équations suivantes pour le CPU :

$$c_0 = \frac{1280 \times 720 \times (f_{comp} \times 2)}{11.52 \times 10^9} \tag{5.30}$$

$$c_1 = 4 \times \frac{1280 \times 720 \times (f_{comp} \times 2)}{11.52 \times 10^9} \tag{5.31}$$

$$c_2 = 4 \times 4 \times \frac{1280 \times 720 \times (f_{comp} \times 2)}{11.52 \times 10^9} \tag{5.32}$$

$$m_0 = \frac{1280 \times 720 \times 2}{10 \times 10^9} \tag{5.33}$$

et pour le GPU :

$$c_0 = \frac{1280 \times 720 \times (f_{comp} \times 2 + 64)}{149.76 \times 10^9} \tag{5.34}$$

$$m_0 = \frac{1280 \times 720 \times 2}{10 \times 10^9} \tag{5.35}$$

Les différents paliers ainsi prédits sont représentés dans la figure 5.4. Contrairement à nos prédictions, le modèle Boat-Hull nous indique que le temps de calcul est supérieur au temps d'accès mémoire quelle que soit la configuration.

Pour comparer ces prédictions aux nôtres, il convient tout d'abord de définir la complexité f_{comp} de nos *kernels*. Cette valeur peut être sujette à discussion, cependant nous choisissons une valeur de $f_{comp} = 2$. Ainsi, pour chaque donnée d'entrée le *kernel* doit effectuer une opération pour le calcul de l'adresse et une autre opération pour le calcul de la différence. La complexité maintenant fixée, le modèle nous donne les prédictions suivantes :

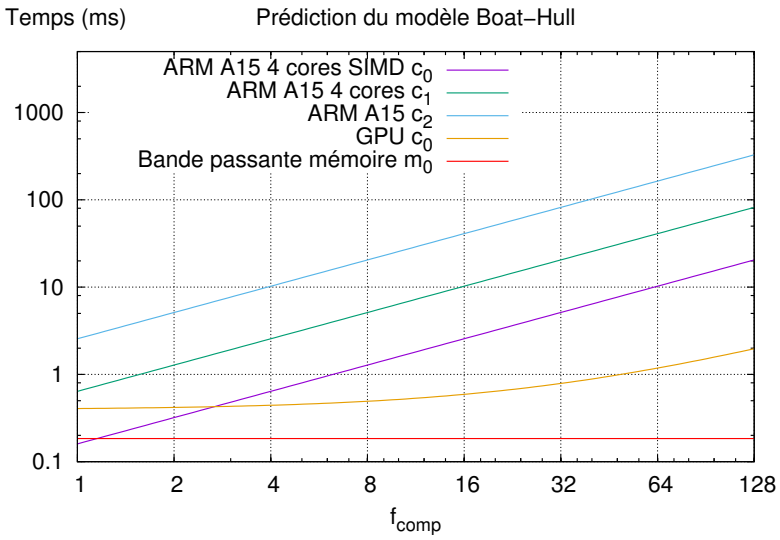


FIGURE 5.4 – Représentation Boat-Hull pour les kernels de calcul du gradient. Chaque palier représente une configuration différente. Remarquons que nous n'affichons pas la configuration ARM A15 utilisant 1 seul cœur avec SIMD, puisqu'elle est équivalente à la prédiction dans la configuration de l'ARM A15 utilisant 4 cœurs.

— ARM A15 4 cœurs SIMD : 0.32 ms
— ARM A15 4 cœurs : 1.28 ms
— ARM A15 SIMD : 1.28 ms
— ARM A15 : 5.12 ms
— GPU : 0.42 ms

5.4.5 Mesures

Après cette discussion sur les prédictions de nos deux *kernels* de calcul du gradient dans différentes configurations et après avoir présenté les prédictions du modèle Boat-Hull, nous proposons de confronter la théorie à la réalité, c'est-à-dire aux temps d'exécutions réels, mesurés sur le Tegra K1. Ainsi, le tableau 5.1 résume les différentes prédictions et le temps d'exécution mesuré pour les différentes configurations. On peut alors remarquer que pour chaque configuration, le temps d'exécution mesuré se situe bien dans l'intervalle prédit, que l'on soit dans une configuration à faible I_A (surligné en jaune) ou à forte I_A (surligné en vert).

Cependant, on observe que les prédictions du modèle Boat-Hull pour l'ARM A15 sont très largement supérieures au temps d'exécution réels, elles se situent même en dehors de notre intervalle. Ce résultat s'explique par le fait que le modèle Boat-Hull s'appuie sur les capacités de calcul à virgule flottante pour ses prédictions (FLOPS). Or nos *kernels* de gradient ne font aucune opération en virgule flottante. De plus, nous avons vu dans le chapitre précédent qu'un processeur n'a pas la même performance en fonction du type d'opération et du type de données. C'est là un des avantages majeurs de notre approche, les différences de capacités de calcul selon les types d'opérations et de variables sont prises en compte par notre modèle de prédiction de performances.

De la même manière, nous donnons dans le tableau 5.2 les prédictions et les temps de transfert mesurés. Notons que ces transferts sont nécessaires si les *kernels* s'exécutent sur GPU. Les temps de transfert réels se situent bien dans nos intervalles prédits.

TABLEAU 5.1 – Prédictions comparées aux mesures des temps d'exécution dans les différentes configurations. Les temps sont donnés en milliseconde. Remarquons que le temps mesuré correspond à une moyenne effectuée sur plusieurs mesures. Les temps surlignés en vert signifie que ce les capacités de calcul sont limitantes (forte I_A), alors que les temps surlignés en jaune signifie que les capacités de la mémoire sont limitantes (faible I_A).

Configuration	**Gradient horizontal**				**Gradient vertical**			
	t_{min}	t_{max}	Boat-Hull	$t_{mesuré}$	t_{min}	t_{max}	Boat-Hull	$t_{mesuré}$
A15	1.02	2.30	5.12	1.25	1.02	2.30	5.12	1.41
A15 SIMD	0.37	0.80	1.28	0.55	0.37	0.80	1.28	0.48
A15 4 cores	0.26	0.58	1.28	0.37	0.26	0.58	1.28	0.36
A15 4 cores SIMD	0.10	0.22	0.32	0.15	0.10	0.22	0.32	0.14
GPU	0.42	0.75	0.42	0.55	0.42	0.75	0.42	0.58

TABLEAU 5.2 – Prédictions comparées aux mesures des temps de transfert entre CPU et GPU. Remarquons que le temps mesuré correspond à une moyenne effectuée sur plusieurs mesures.

Transfert	t_{min}	t_{max}	$t_{mesuré}$
CPU→GPU	0.59	1.02	0.64
GPU→CPU	1.68	3.13	1.73

5.5 *Mapping* et pipeline d'exécution

Au cours de ce chapitre, nous avons présenté notre méthodologie de prédiction de performances. Notre approche permet d'adresser différents types de processeurs (par exemple CPU, GPU, etc.) et peut gérer du code arbitraire. La prédiction du temps d'exécution d'un *kernel* se divise en deux parties : la prédiction des délais mémoire et la prédiction du temps de calcul. Contrairement aux approches existantes, nous proposons une méthode qui permet de prendre en compte les différences de capacité de calcul d'un processeur en fonction du type d'opération et du type de variables utilisé. De plus un temps d'exécution prédit n'est pas représenté par un scalaire mais par un intervalle représentant un ensemble des temps d'exécution possibles. Les prédictions s'appuient directement sur les caractéristiques des architectures issues de nos vecteurs de test. Nous avons montré de cette manière que notre méthodologie est capable de prédire les effets du compilateur sur le temps d'exécution en fonction des options de compilation utilisées. Contrairement aux approches de l'état-de-l'art qui utilisent une bande passante fixe pour prédire les délais d'accès à la mémoire, notre approche s'appuie sur nos vecteurs de test pour déterminer un délai d'accès mémoire. Ainsi, nous proposons d'utiliser des fonctions par morceaux ou des tables de correspondances pour déterminer un temps d'accès à la mémoire en fonction de la quantité de données à accéder.

Les intervalles obtenus par notre méthodologie de prédiction de performances sont tout à fait compatibles avec l'approche discutée dans la section 3.3. Ainsi, un temps d'exécution prédit peut être représenté sous la forme d'un intervalle, ou comme la médiane de l'intervalle. Il est également possible d'utiliser l'approche stochastique et de représenter le temps d'exécution prédit comme une variable aléatoire suivant une PDF uniforme ou Gaussienne (par exemple en définissant la moyenne de la Gaussienne comme la médiane de l'intervalle et l'écart type comme un tiers de l'écart entre la médiane et les bornes de l'intervalle). Le choix de la stratégie à appliquer revient à l'utilisateur, même si certaines contraintes peuvent imposer une approche plutôt qu'une autre (par exemple la taille et la complexité du graphe de traitement).

Comme discuté dans la section 3.3.1, nous supposons que deux variables aléatoires représentant deux temps d'exécutions sont indépendantes si les deux *kernels* sont exécutés sur deux processeurs différents, ou sur le même mais de manière non concurrente. Cependant, lorsque plusieurs *kernels* sont exécutés de manière concurrente et partagent certaines ressources (par exemple s'ils sont exécutés sur la même unité d'exécution ou alors s'ils partagent le même niveau de cache) cette hypothèse n'est plus valable. Dans ce cas, la prédiction est basée sur les vecteurs de test. En effet, les vecteurs de test *mid-level* illustrent directement l'impact des exécutions concurrentes sur le temps de calcul et sur les délais d'accès à la mémoire. Les différentes approches discutées dans la section 3.3 permettent d'obtenir un *mapping* répondant au cahier des charges, mais ne renseignent pas sur le pipeline d'exécution à implémenter (c'est-à-dire l'ordre dans lequel doivent d'exécuter les *kernels*). Les vecteurs de test *mid-level* nous permettent là encore de répondre à cette problématique, à savoir s'il est préférable d'exécuter deux *kernels* indépendants séquentiellement ou alors simultanément sur la même unité d'exécution, et donc plus généralement de savoir quel est le pipeline d'exécution le mieux adapté pour une architecture donnée. En fait le processus de recherche de pipeline d'exécution peut changer légèrement les valeurs prédites, puisque deux *kernels* exécutés simultanément n'ont pas le même temps d'exécution que s'ils sont exécutés séparément. Cette étape devient de plus en plus importante avec le nombre de *kernels* indépendant mappés sur la même unité d'exécution, et donc avec le nombre de pipelines d'exécution possible.

Le processus complet de recherche de *mapping* optimal basé sur notre méthodologie de prédiction de performance et intégrant le raffinement des prédictions après la recherche du pipeline d'exécution est donné en figure 5.5. Ainsi, nous commençons à appliquer la prédiction de performance sur l'ensemble des temps d'exécutions τ et des délais de transfert δ. Nous proposons d'utiliser l'approche par arithmétique des intervalles comme une analyse préliminaire permettant une présélection rapide des *mappings* ayant un respect des contraintes temps-réel avec 100% de certitude et un rejet de ceux qui ne respectent pas au moins l'une des contraintes temps-réel. Ensuite, l'approche par fonction de coût est appliquée pour classer le reste des *mappings* et pour garder un nombre limité de *mappings* possibles. Enfin, les différents pipelines d'exécution sont explorés et, à l'aide des vecteurs de test *mid-level*, nous proposons un ajustement du coût de chaque *mapping* dans le but de trouver le meilleur *mapping* et pipeline d'exécution.

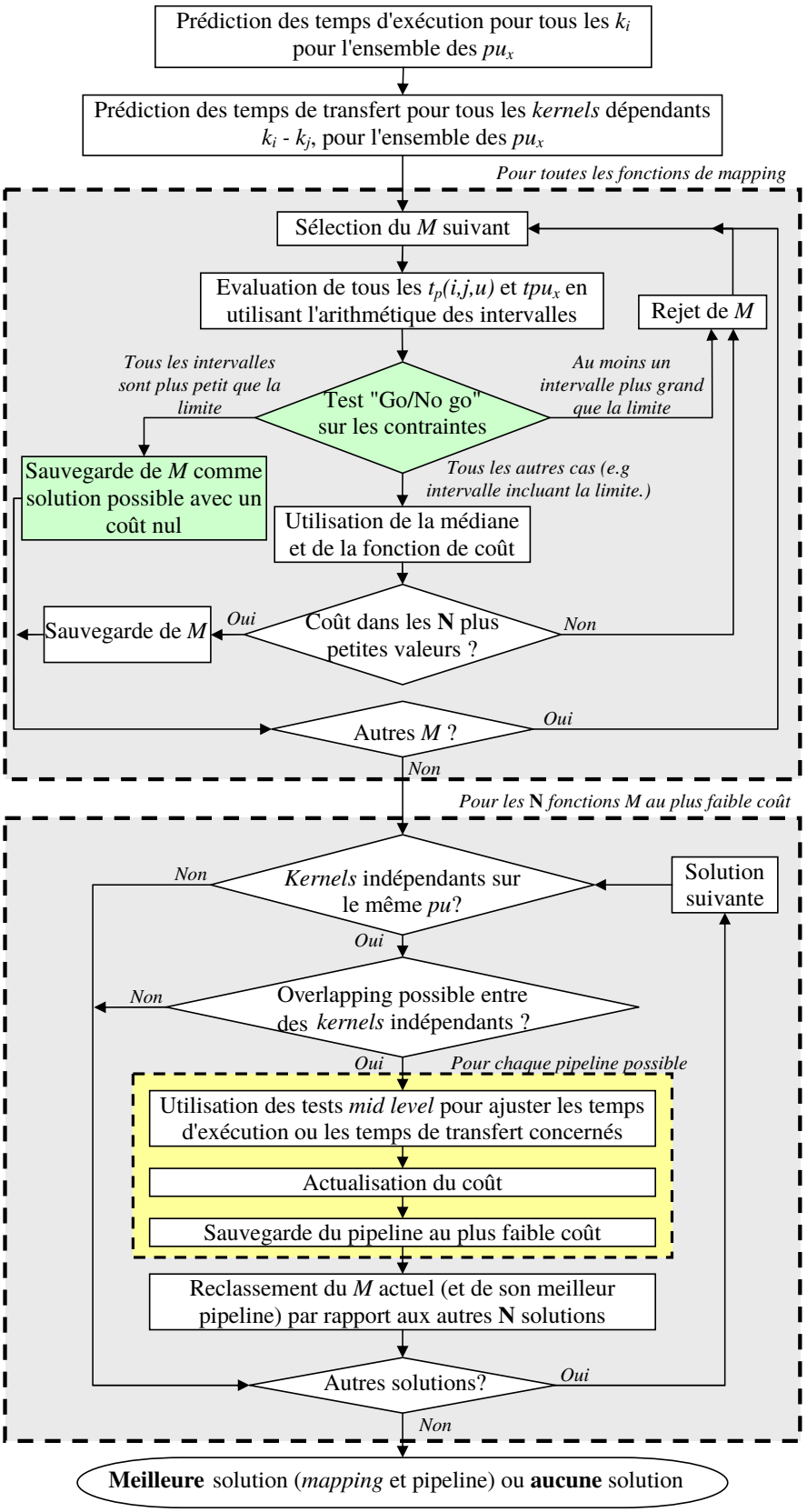


FIGURE 5.5 – Processus de choix du meilleur *mapping*. Il s'agit du détail du bloc « Prédiction de performances sur tous les SoCs et optimisation du *mapping* » de la figure 3.7.

5.6 Références

AMDAHL, G. M. 1967, «Validity of the single processor approach to achieving large scale computing capabilities», dans *Proc. of the Spring Joint Computer Conference*, ACM, p. 483–485. 160

BAKHODA, A., G. L. YUAN, W. W. FUNG, H. WONG et T. M. AAMODT. 2009, «Analyzing cuda workloads using a detailed gpu simulator», dans *2009 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS 2009.*, IEEE, p. 163–174. 157

BENOIT, N. et S. LOUISE. 2015, «A performance prediction for automatic placement of heterogeneous workloads on many-cores», dans *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, p. 159–166. 160

CHE, S. et K. SKADRON. 2014, «Benchfriend : Correlating the performance of gpu benchmarks», *International Journal of High Performance Computing Applications*, vol. 28, nº 2, p. 238–250. 159

COLLANGE, S., M. DAUMAS, D. DEFOUR et D. PARELLO. 2010, «Barra : A parallel functional simulator for gpgpu», dans *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, IEEE, p. 351–360. 157

DIAMOS, G. F., A. R. KERR, S. YALAMANCHILI et N. CLARK. 2010, «Ocelot : a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems», dans *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ACM, p. 353–364. 158

GANESTAM, P. et M. DOGGETT. 2012, «Auto-tuning interactive ray tracing using an analytical gpu architecture model», dans *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, ACM, p. 94–100. 160

HONG, S. et H. KIM. 2009, «An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness», dans *ACM SIGARCH Comp. Architecture News*, vol. 37, ACM, p. 152–163. 159, 160

HOSTE, K., A. PHANSALKAR, L. EECKHOUT, A. GEORGES, L. K. JOHN et K. DE BOSSCHERE. 2006, «Performance prediction based on inherent program similarity», dans *Proc. of the 15th International Conference on Parallel Architectures and Compilation Techniques*, ACM, p. 114–122. 158

KERR, A., G. DIAMOS et S. YALAMANCHILI. 2010, «Modeling GPU-CPU workloads and systems», dans *Proc. of the 3rd Workshop on General-Purpose Computation on GPU*, ACM, p. 31–42. 158

KIM, H., J. LEE, N. B. LAKSHMINARAYANA, J. SIM, J. LIM et T. PHO. 2012, «Macsim : A cpu-gpu heterogeneous simulation framework user guide», *Georgia Institute of Technology*. 158

LEE, V. W., C. KIM, J. CHHUGANI, M. DEISHER, D. KIM, A. D. NGUYEN, N. SATISH, M. SMELYANSKIY, S. CHENNUPATY, P. HAMMARLUND et collab.. 2010, «Debunking the 100x gpu vs. cpu myth : an evaluation of throughput computing on cpu and gpu», *ACM SIGARCH Computer Architecture News*, vol. 38, nº 3, p. 451–460. 157

LOPEZ-NOVOA, U., A. MENDIBURU et J. MIGUEL-ALONSO. 2015, «A survey of performance modeling and simulation techniques for accelerator-based computing», *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, nº 1, p. 272–281. 157

MENG, J., V. A. MOROZOV, K. KUMARAN, V. VISHWANATH et T. D. URAM. 2011, «Grophecy : Gpu performance projection from cpu code skeletons», dans *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, p. 14. 160

NUGTEREN, C. et H. CORPORAAL. 2011, «A modular and parameterisable classification of algorithms», *Eindhoven University of Technology, Tech. Rep. ESR-2011-02*. 161

NUGTEREN, C. et H. CORPORAAL. 2012, «The boat hull model : adapting the roofline model to enable performance prediction for parallel computing», *ACM Sigplan Notices*, vol. 47, nº 8, p. 291–292. 161, 162, 164, 174

SATO, K., K. KOMATSU, H. TAKIZAWA et H. KOBAYASHI. 2011, «A history-based performance prediction model with profile data classification for automatic task allocation in heterogeneous computing systems», dans *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, IEEE, p. 135–142. 159

SAUSSARD, R., B. BOUZID, R. REYNAUD et M. VASILIU. 2015a, «Predicting ADAS algorithms performances on K1 architecture», Web ressource. http://on-demand-gtc.gputechconf.com. 156

SAUSSARD, R., B. BOUZID, M. VASILIU et R. REYNAUD. 2015b, «Optimal performance prediction of ADAS algorithms on embedded parallel architectures», dans *2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC)*, IEEE, p. 213–218. 162

SAUSSARD, R., B. BOUZID, M. VASILIU et R. REYNAUD. 2015c, «Towards an automatic prediction of image processing algorithms performances on embedded heterogeneous architectures», dans *2015 44th International Conference on Parallel Processing Workshops (ICPPW)*, IEEE, p. 27–36. 162

SHEN, J., A. L. VARBANESCU et H. SIPS. 2014, «Look before you leap : using the right hardware resources to accelerate applications», dans *2014 IEEE International Conference on High Performance Computing and Communications (HPCC)*, IEEE, p. 383–391. 157, 161

UBAL, R., B. JANG, P. MISTRY, D. SCHAA et D. KAELI. 2012, «Multi2sim : a simulation framework for CPU-GPU computing», dans *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ACM, p. 335–344. 158

VUDUC, R., A. CHANDRAMOWLISHWARAN, J. CHOI, M. GUNEY et A. SHRINGARPURE. 2010, «On the limits of gpu acceleration», dans *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, USENIX Association, p. 13–13. 157

WILLIAMS, S., A. WATERMAN et D. PATTERSON. 2009, «Roofline : an insightful visual performance model for multicore architectures», *Communications of the ACM*, vol. 52, nº 4, p. 65–76. 159, 161, 162, 164

ZAKHARENKO, V., T. AAMODT et A. MOSHOVOS. 2013, «Characterizing the performance benefits of fused cpu/gpu systems using fusionsim», dans *Proceedings of the Conference on Design, Automation and Test in Europe*, EDA Consortium, p. 685–688. 158

Chapitre 6

Résultats et applications

« *So with each advance in understanding come new questions. So we need to be very humble. We shouldn't have hubris and think that we can understand everything. But history tells us that there is good reason to believe that we will continue making fantastic progress in the years ahead.* »

Max Tegmark

Sommaire

6.1 Introduction

Après avoir présenté le principe de fonctionnement de notre méthodologie d'analyse de l'embarquabilité dans les chapitres précédents, on se propose d'étudier deux applications de traitement d'image ADAS et d'appliquer le processus tel que nous l'avons défini.

Au cours de cette thèse, nous avons participé à l'élaboration et à l'implémentation d'un algorithme de traitement d'images appliqué à la problématique du véhicule autonome. Il s'agit d'un algorithme d'odométrie visuelle, pouvant déterminer le déplacement du véhicule à l'aide d'une caméra. Contrairement à la majorité des contributions dans ce domaine [FRAUNDORFER et SCARAMUZZA, 2012], cet algorithme a pour particularité d'être de type *dense*. Dans la communauté de la vision artificielle le choix *dense* vs *sparse* fait très souvent débat. Remarquons tout de même que certains auteurs, comme VALGAERTS et collab. [2012], montrent clairement que l'approche *dense* apporte de meilleurs résultats pour l'estimation de la matrice fondamentale. Une approche de type *dense* implique forcément un plus grand volume de données à traiter mais également un plus grand degré de parallélisme. Ce type d'algorithme est donc très intéressant pour notre méthodologie d'analyse de l'embarquabilité. Dans un second temps, nous nous intéresserons à un algorithme beaucoup plus classique, la détection de piéton de DALAL et TRIGGS [2005].

6.2 Odométrie visuelle

La localisation est une problématique majeure dans le cadre du véhicule autonome, il est indispensable de connaître avec précision la position du véhicule sur une carte au cours du temps. Les technologies à base de GNSS permettent une localisation d'une précision de quelques mètres, ce qui est insuffisant par exemple pour déterminer dans quelle voie se situe le véhicule à un instant donné. De plus, ces systèmes nécessitent un environnement propice à la communication avec les satellites pour pouvoir fonctionner. Ainsi, dans le cas d'une conduite sous un tunnel ou en ville à proximité de bâtiments, les capteurs GNSS deviennent beaucoup moins précis voire inutilisables.

Pour répondre à cette problématique de localisation précise, nous proposons un algorithme d'odométrie visuelle utilisant la caméra de recul du véhicule. Le principe est le suivant : on cherche à déterminer le déplacement du véhicule entre deux images successives. Par itérations il est alors possible de reconstruire la trajectoire complète du véhicule et donc de connaître la position de celui-ci par rapport à une position initiale à tout instant. Remarquons que la trajectoire calculée à partir des déplacements successifs du véhicule a tendance à diverger au cours du temps. Ainsi, le déplacement effectué entre deux images n'est pas déterminé avec une précision de 100% par rapport à la réalité. Lors du calcul de la trajectoire la composition des déplacements successifs accumule les erreurs sur chaque déplacement et fait donc diverger la trajectoire. Il convient donc d'utiliser l'odométrie visuelle avec d'autres capteurs ou informations pour effectuer un recalage de la trajectoire et éviter une divergence trop forte (comme un capteur GNSS). Par exemple, OHNO et collab. [2003] proposent d'utiliser un capteur GNSS, de l'odométrie et une fusion de données à partir d'un filtre de Kalman étendu [MAYBECK, 1982] pour déterminer la localisation d'un robot mobile en extérieur. Ainsi les auteurs obtiennent une localisation pouvant compenser les erreurs du capteur GNSS grâce à l'odométrie et qui ne diverge pas. Dans le cadre de ce manuscrit, nous nous limiterons cependant à la présentation de la partie concernant l'algorithme d'odométrie visuelle, la fusion de données avec un autre capteur ne sera pas abordée.

6.2.1 Fonctionnement de l'algorithme

Nous avons choisi de reprendre l'algorithme de LOVEGROVE et collab. [2011]. Les auteurs proposent d'utiliser l'*Efficient Second order Minimisation* (ESM), telle que définie par MALIS [2004], pour déterminer le déplacement effectué par le véhicule à partir de deux images successives. Remarquons que l'algorithme est de type dense, c'est-à-dire qu'il utilise toute l'image (ou une région d'intérêt) pour procéder, contrairement à des algorithmes dits *sparse* qui utilisent des points d'intérêts pour calculer le déplacement. Pour déterminer le déplacement du véhicule, l'algorithme se base sur l'hypothèse que la route est approximable par un plan, nous considérerons donc dans la suite de la section que la route est un plan.

Configuration

Pour commencer, il convient de définir la configuration de l'installation. Un schéma de principe est donné en figure 6.1, la caméra C est située à l'arrière du véhicule. Son repère $\mathscr{R}_C = (C, \vec{x}_c, \vec{y}_c, \vec{z}_c)$ est illustré en bleu dans le schéma. La projection du centre optique C sur le plan de la route suivant l'axe z_c est noté O. Pour la suite nous définissons le repère orthonormé $\mathscr{R}_V = (V, \vec{x}_v, \vec{y}_v, \vec{z}_v)$, tel que V soit la projection de C sur le plan de la route suivant la normale de ce plan, tel que $\vec{z}_v$ soit orthogonal au plan de la route et tel que la rotation entre $\mathscr{R}_C$ et $\mathscr{R}_V$ autour de l'axe $\vec{z}_v$ soit nulle.

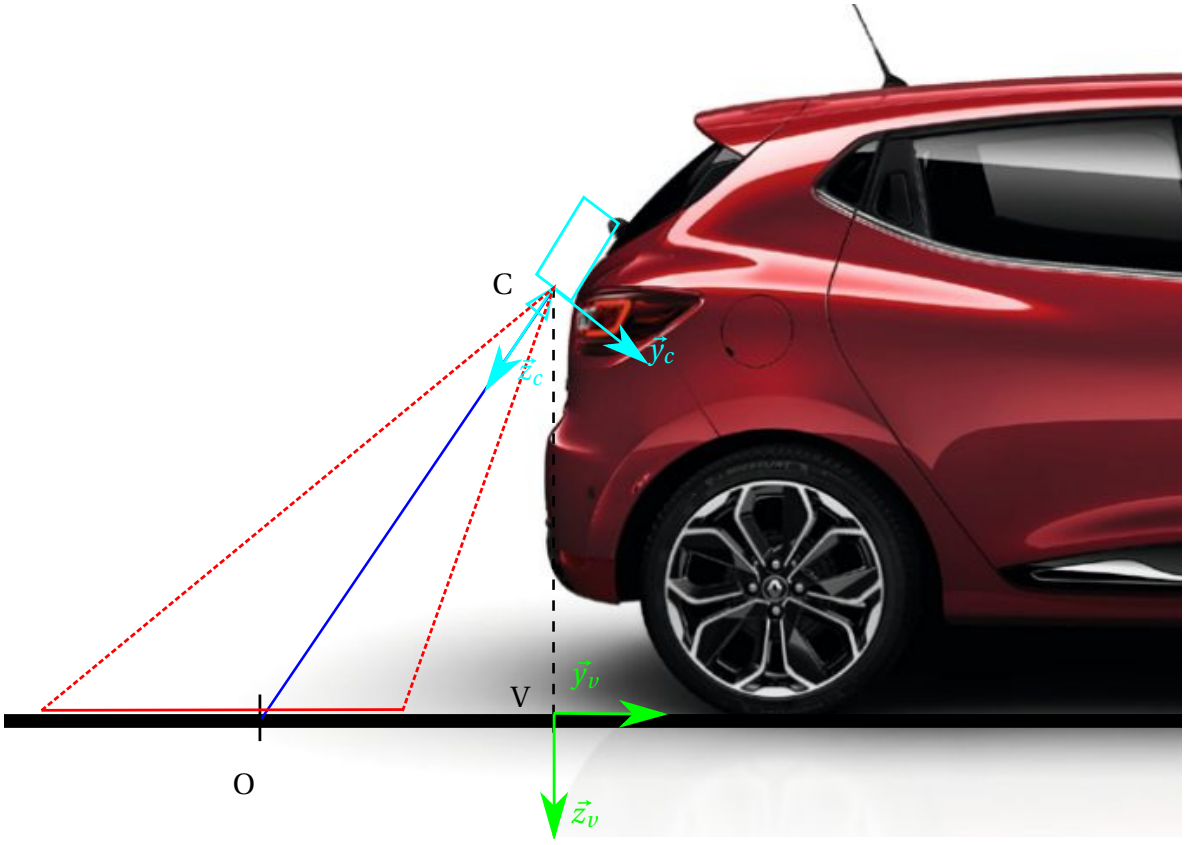


FIGURE 6.1 – Schéma de principe

Le véhicule étant en déplacement au cours du temps, nous noterons C^a, V^a la position de C, V à l'instant courant et C^b, V^b la position de C, V à l'instant précédent. De la même manière, I^a correspond à l'image courante et I^b correspond à l'image précédente. Soit

H^{ab} l'homographie permettant la transformation de I^b à I^a, définie tel que :

$$I^a(H^{ab} \cdot p_b) = I^b(p_b) \tag{6.1}$$

où p_b désigne la coordonnée en pixels de l'image I^b. Le déplacement du véhicule entre deux instants, c'est-à-dire la transformation T_V^{ab}, peut être directement déduite de la relation entre I^a et I^b, c'est-à-dire H^{ab}. Nous utiliserons les notations ainsi définies pour la suite de la section.

Fonction de coût

Dans la version initiale de l'algorithme [LOVEGROVE et collab., 2011], les auteurs proposent de déterminer directement le déplacement du véhicule entre deux images successives T_V^{ab}. Étant donné que l'axe optique de la caméra $\vec{z}_c$ n'est pas perpendiculaire au plan de la route, un déplacement du véhicule composé d'une translation sur deux dimensions $(\Delta x_v, \Delta y_v)$ et d'une rotation autour de $\vec{z}_v$ implique dans le plan image une translation sur trois dimensions $(\Delta x_c, \Delta y_c, \Delta z_c)$ et une rotation pouvant être décomposée en trois rotations autour des trois axes du repère caméra $\mathscr{R}_C$.

Pour plus de simplicité, nous avons choisi de travailler sur des images en *bird eye view*, comme illustré en figure 6.2. Pour rappel, le principe autour de l'homographie et de la construction de *bird eye views* est donné dans la section 2.2.2. Ensuite, le calcul du déplacement est décomposé en deux étapes :

1. calcul de l'homographie H^{ab} entre deux images *bird eye view* successives,
2. calcul du déplacement du véhicule T_V^{ab} entre ces deux images à partir de H^{ab}.

Dans la configuration d'image en *bird eye view*, la matrice d'homographie H^{ab} ne dépend plus que de trois paramètres :

— la translation u sur l'axe horizontal de l'image,
— la translation v sur l'axe vertical de l'image,
— la rotation θ autour de la normale du plan de l'image (qui est colinéaire à la normale du plan de la route dans cette configuration).

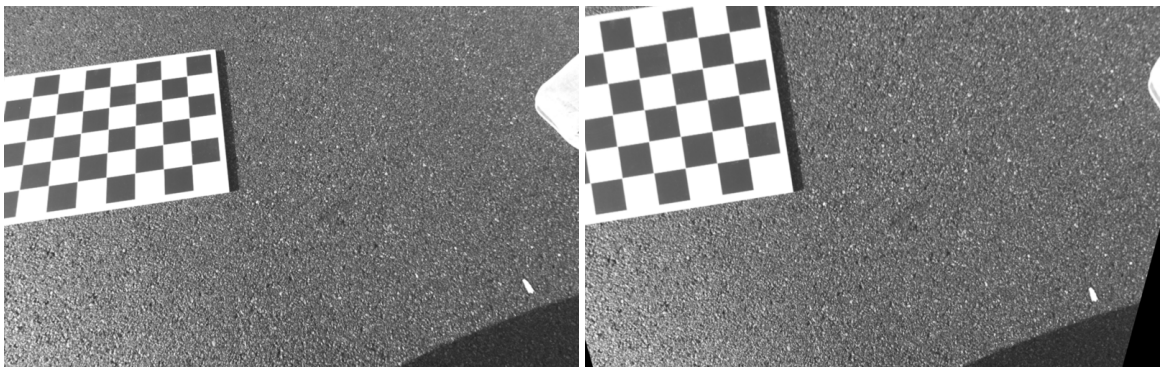


FIGURE 6.2 – Transformation en *bird eye view* : à gauche l'image d'origine et à droite l'image transformée.

Pour la suite nous noterons $\mathbf{x}$:

$$\mathbf{x} = (u, v, \theta) \tag{6.2}$$

On a alors :

$$H^{ab}(\mathbf{x}) = \begin{pmatrix} \cos\theta & -\sin\theta & u \\ \sin\theta & \cos\theta & v \\ 0 & 0 & 1 \end{pmatrix} \tag{6.3}$$

Pour estimer le déplacement du véhicule, il faut donc déterminer le vecteur $\mathbf{x}$ sous la contrainte définie dans l'équation 6.1. Ainsi, nous définissons la fonction de coût $F(\mathbf{x})$, comme étant la *Sum of Square Difference* (SSD) entre l'image I^a et l'image I^b après l'application de l'homographie $H^{ab}(\mathbf{x})$. La SSD est calculée pour chaque pixel p_b appartenant à la région d'intérêt Ω de l'image.

$$F(\mathbf{x}) = \frac{1}{2} \sum_{p_b \in \Omega} \left(f_{p_b}(\mathbf{x}) \right)^2 \tag{6.4}$$

$$f_{p_b}(\mathbf{x}) = I^a(H^{ab}(\mathbf{x}) \cdot p_b) - I^b(p_b) \tag{6.5}$$

Les différences par pixel $f_{p_b}(\mathbf{x})$ peuvent être représentées sous la forme d'un vecteur colonne $f(\mathbf{x})$ tel que :

$$\frac{1}{2} \| f(\mathbf{x}) \|^2 = \frac{1}{2} \sum_{p_b \in \Omega} \left(f_{p_b}(\mathbf{x}) \right)^2 = F(\mathbf{x}) \tag{6.6}$$

Minimisation par ESM

Nous cherchons donc à obtenir $\mathbf{x_0}$ tel que $\mathbf{x_0}$ minimise la fonction de coût $F(\mathbf{x})$:

$$\mathbf{x_0} = \arg\min_{\mathbf{x}} (F(\mathbf{x})) \tag{6.7}$$

Pour ce faire, nous utilisons la méthode ESM telle que définie par MALIS [2004] et MEI et collab. [2008]. Cette méthode permet une optimisation itérative d'une fonction de coût, suivant une approximation du second ordre mais ne nécessitant que le calcul de la dérivée première.

Soit $\nabla f(\mathbf{x})$ le vecteur ligne des dérivés partielles de f évalué en $\mathbf{x}$. Les développements en série de Taylor de $f(\mathbf{x})$ et $\nabla f(\mathbf{x})$ autour de $\mathbf{0}$ sont données par :

$$f(\mathbf{x}) = f(\mathbf{0}) + \nabla f(\mathbf{0})\mathbf{x} + \frac{1}{2}\mathbf{x}^T \nabla\nabla f(\mathbf{0})\mathbf{x} + \dots \tag{6.8}$$

$$\nabla f(\mathbf{x}) = \nabla f(\mathbf{0}) + \mathbf{x}^T \nabla\nabla f(\mathbf{0}) + \dots \tag{6.9}$$

Soient $\hat{F}(\mathbf{x})$ et $\hat{f}(\mathbf{x})$ les développements en séries de Taylor à l'ordre deux de $F(\mathbf{x})$ et $f(\mathbf{x})$. En utilisant le terme $\mathbf{x}^T \nabla\nabla f(\mathbf{0})$ dans les deux équations précédentes, et en approximant $\hat{F}(\mathbf{x}) \approx F(\mathbf{x})$ et $\hat{f}(\mathbf{x}) \approx f(\mathbf{x})$, on en déduit :

$$\hat{f}(\mathbf{x}) = f(\mathbf{0}) + \nabla f(\mathbf{0}) + \frac{1}{2} \left(\nabla f(\mathbf{0}) + \nabla f(\mathbf{x}) \right) \mathbf{x} \tag{6.10}$$

$$\hat{F}(\mathbf{x}) = \frac{1}{2} \| \hat{f}(\mathbf{x}) \|^2 = \frac{1}{2} \hat{f}(\mathbf{x})^T \hat{f}(\mathbf{x}) \tag{6.11}$$

Par définition, la présence d'un minimum local de $F(\mathbf{x})$ est détecté par $\nabla F(\mathbf{x_0}) = \mathbf{0}$. Si l'on suppose que ce minimum local existe et que l'on approxime $\nabla F(\mathbf{x})$ par :

$$\nabla F(\mathbf{x}) \approx \nabla \hat{f}(\mathbf{x})^T \hat{f}(\mathbf{x}) \tag{6.12}$$

trouver le point $\mathbf{x_0}$ tel que $\nabla F(\mathbf{x_0}) = \mathbf{0}$ revient alors à :

$$\nabla \hat{f}(\mathbf{x_0})^T \left(f(\mathbf{0}) + \frac{1}{2} \left(\nabla f(\mathbf{0}) + \nabla f(\mathbf{x_0}) \right) \mathbf{x_0} \right) = \mathbf{0} \tag{6.13}$$

En posant $J = \frac{1}{2}\left(\nabla f(\mathbf{0}) + \nabla f(\mathbf{x_0})\right)$, on cherche alors un $\mathbf{x_0}$ tel que :

$$J \cdot \mathbf{x_0} = -f(\mathbf{0}) \tag{6.14}$$

Nous proposons alors de résoudre ce système surdimensionné par la méthode des moindres carrés :

$$J^T J \cdot \mathbf{x_0} = -J^T \cdot f(\mathbf{0}) \tag{6.15}$$

$$\mathbf{x_0} = -(J^T J)^{-1} \cdot J^T \cdot f(\mathbf{0}) \tag{6.16}$$

Remarquons que $J^T J$ est une matrice symétrique, son inverse peut donc facilement être calculée en utilisant par exemple la factorisation de Cholesky [CIARLET, 1982].

Pour déterminer la solution $\mathbf{x_0}$ de l'équation précédente, nous avons besoin d'évaluer les dérivées partielles de la fonction de coût en $\mathbf{0}$ et en $\mathbf{x_0}$. Ainsi, en reprenant l'équation 6.5, pour chaque pixel on a :

$$\frac{\partial f_{p_b}(\mathbf{x})}{\partial x_i} = \frac{\partial I^a(H^{ab}(\mathbf{x}) \cdot p_b)}{\partial H^{ab}(\mathbf{x}) \cdot p_b} \cdot \frac{\partial H^{ab}(\mathbf{x}) \cdot p_b}{\partial x_i} \cdot p_b \tag{6.17}$$

où x_i correspond à la i^{e} composante du vecteur $\mathbf{x}$. Le détail du calcul de cette dérivée est donné en annexe A.1. De manière itérative, l'algorithme est alors capable de déterminer l'homographie H^{ab} permettant la transformation de I^b vers I^a.

Position du véhicule

Le résultat souhaité de l'algorithme d'odométrie visuelle est un déplacement relatif entre deux instants, et non l'homographie H^{ab}. Il convient donc de définir comment obtenir ce déplacement T^{ab} à partir de H^{ab}. Comme discuté précédemment, nous avons choisi de travailler sur des images *bird eye view*, or dans cette configuration il est possible de définir une relation entre une coordonnée en pixels et une coordonnée en millimètres valable sur toute l'image. Ainsi, un point M appartenant au plan de la route, de coordonné $(M_x, M_y, 1)^T$ dans le repère $(V, \vec{x_v}, \vec{y_v})$ aura une coordonnée en pixels m :

$$\begin{pmatrix} m_u \\ m_v \\ 1 \end{pmatrix} = K \cdot T_{OV} \cdot \begin{pmatrix} M_x \\ M_y \\ 1 \end{pmatrix} \qquad K = \begin{pmatrix} f_c/h & 0 & c_u \\ 0 & f_c/h & c_v \\ 0 & 0 & 1 \end{pmatrix} \qquad T_{OV} = \begin{pmatrix} 1 & 0 & O_x \\ 0 & 1 & O_y \\ 0 & 0 & 1 \end{pmatrix} \tag{6.18}$$

où f_c correspond à la distance focale de la caméra en pixel, (c_u, c_v) correspond à la coordonnée du centre optique de la caméra dans le repère image, (O_x, O_y) correspond à la coordonnée du point O dans le repère $(V, \vec{x_v}, \vec{y_v})$ et h correspond à la hauteur de la caméra par rapport au plan du sol, c'est-à-dire $h = \overrightarrow{CV} \cdot \vec{z_v}$. De la même manière un pixel p de coordonnée $(p_u, p_v, 1)$ aura pour coordonnée sur le plan de la route :

$$\begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix} = T_{OV}^{-1} \cdot K^{-1} \cdot \begin{pmatrix} p_u \\ p_v \\ 1 \end{pmatrix} \qquad K^{-1} = \begin{pmatrix} h/f_c & 0 & -c_u h/f_c \\ 0 & h/f_c & -c_v h/f_c \\ 0 & 0 & 1 \end{pmatrix} \qquad T_{OV}^{-1} = \begin{pmatrix} 1 & 0 & -O_x \\ 0 & 1 & -O_y \\ 0 & 0 & 1 \end{pmatrix} \tag{6.19}$$

En composant les deux équations précédentes, on peut alors en déduire :

$$T^{ab} = T_{OV}^{-1} \cdot K^{-1} \cdot H^{ab} \cdot K \cdot T_{OV} \tag{6.20}$$

Le sujet principal de la thèse étant l'analyse de l'embarquabilité, nous nous focaliserons dans ce chapitre autour des performances et de l'embarquabilité de cet algorithme. Les résultats obtenus à partir de l'algorithme sont présentés et discutés en annexe A.2.

6.2.2 Modélisation de l'algorithme

On se propose de définir un graphe de traitement composé de 4 *kernels* pour représenter l'algorithme d'odométrie visuelle :

— k_1 : acquisition de l'image

— k_2 : correction de la distorsion et construction de la *bird eye view*

— k_3 : optimisation et calcul de l'homographie entre deux images successives

— k_4 : calcul de la trajectoire

Comme le *kernel* k_3 nécessite l'image actuelle et l'image précédente pour calculer l'homographie entre ces deux images, il y a donc une dépendance spatio-temporelle entre $k_2^{(n-1)}$ et $k_3^{(n)}$, notée $\delta_{2,3}^{(*)}$. Le graphe de deux instances successives est donné en figure 6.3.

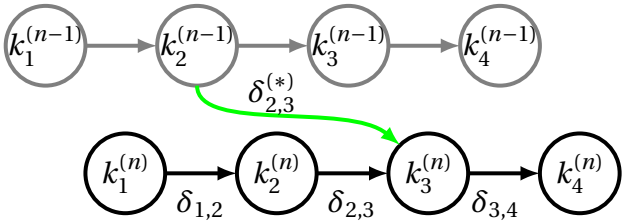


FIGURE 6.3 – Graphe de traitement pour l'algorithme d'odométrie visuelle

Nous choisissons de travailler avec une caméra haute définition : 1936×1216 pixels en niveau de gris. Concernant les contraintes temps-réel, les spécifications nous imposent une contrainte de rythme $T_c = 20$ ms (soit une caméra fonctionnant à 50 fps). Pour cette application, la contrainte de latence est beaucoup moins critique, on s'imposera donc $dt_{spec}(1,4) = 500$ ms. Pour embarquer cet algorithme, on se propose d'étudier deux SoCs : le Tegra K1 et le Tegra X1.

Kernel k_1

Le *kernel* de l'acquisition d'image, comme son nom l'indique, récupère l'image issue de la caméra pour la rendre accessible par le CPU et le GPU. Le fonctionnement de ce *kernel*, et donc son temps d'exécution, est fortement dépendant du type de caméra à utiliser. Par exemple, si l'on utilise une caméra USB 3.0 alors le CPU doit exécuter le driver permettant l'acquisition des images, l'acquisition implique donc une charge de calcul non négligeable pour le CPU. Or si l'on utilise l'une des nombreuses interfaces CSI, alors les images acquises sont directement copiées sur la mémoire par DMA depuis la caméra, ce qui ne représente donc aucune charge pour le CPU. Ce *kernel* ne peut pas s'exécuter sur GPU.

Kernel k_2

Le *kernel* k_2 permet de corriger la distorsion et d'appliquer l'homographie permettant de construire la *bird eye view*. Ce *kernel* fonctionne par le biais de deux tables de correspondances : à chaque coordonnée de l'image de sortie (u, v) on associe le couple de coordonnées correspondantes de l'image d'entrée (x, y). Ces tables de correspondances sont déterminées lors de la phase de calibration et dépendent des paramètres intrinsèques et extrinsèques de la caméra. Bien évidemment (x, y) ne sont pas des nombres entiers, pour déterminer la valeur de l'image d'entrée $I_{in}(x, y)$ il est donc nécessaire de procéder à une interpolation bilinéaire.

Du point de vue mémoire, ce *kernel* nécessite un très grand nombre d'accès à cause des tables de correspondances. Ainsi, pour une image en 1936×1216, si chaque valeur de la table est codée sur des *float32*, on obtient alors une taille de $1936 \times 1216 \times 2 \times 4 = 17.96$ MB. Concernant le calcul, la plus grosse charge est imputée à l'interpolation bilinéaire. Remarquons qu'il est possible de fortement diminuer la complexité du *kernel* en remplaçant l'interpolation par un simple arrondi de la coordonnée. Dans ce cas, l'image résultante n'est cependant plus fidèle à la réalité ce qui affecte d'une manière importante la fonction de coût et la mise en correspondance.

Nous choisissons donc un *kernel* appliquant une interpolation bilinéaire, dont le calcul est basé sur la fonction *floor()* de la librairie standard. La prédiction de performances de ce *kernel* doit donc s'appuyer sur la caractérisation de cette fonction pour l'ensemble des architectures envisagées. Cette étude a donc été effectuée sur les ARM A15 et A57 du Tegra K1 et Tegra X1. Les résultats de cette étude montrent un temps de calcul beaucoup plus important sur A15 que sur A57. Ainsi, pour l'A15 le code assembleur nous indique que le calcul de la fonction *floor()* se fait par une librairie externe, ce qui implique un délai supplémentaire lors de l'exécution. Alors que sur A57 le compilateur remplace la fonction *floor()* par une instruction permettant de convertir directement un *float32* vers un *int32*. Sur GPU l'interpolation peut être effectuée de manière hardware par les *texture units*, ce qui réduit considérablement la charge de calcul.

Kernel k_3

Le *kernel* k_3 représente le cœur de l'algorithme, il s'agit de l'optimisation permettant de déterminer l'homographie entre deux images successives. Ce *kernel* est itératif, c'est-à-dire qu'il effectue successivement (plusieurs fois) une même fonction. Un plus grand nombre d'itérations permet de converger vers une estimation plus précise de l'homographie, mais représente une charge de calcul plus grande. Pour plus de simplicité dans l'analyse de l'embarquabilté, nous choisissons de fixer le nombre d'itérations à une constante N, dont la valeur reste à déterminer. Pour diminuer le temps d'exécution de ce *kernel*, nous choisissons de limiter le calcul de l'optimisation à une région d'intérêt d'une taille de 968×608 pixels, dont le centre est identique au centre de l'image complète.

À chaque itération, le *kernel* applique une homographie sur l'image d'entrée, calcul la différence pixel à pixel par rapport à l'image précédente, calcul les matrices $J^T J$ et $J^T f(\mathbf{0})$ pour chacun des pixels de la région d'intérêt puis somme l'ensemble des $J^T J$ et $J^T f(\mathbf{0})$ pour enfin résoudre le système de l'équation 6.16. La matrice d'homographie entre les deux images est alors actualisée, puis on recommence l'optimisation jusqu'à ce qu'elle ait été appliquée N fois.

Kernel k_4

Le *kernel* k_4 permet de calculer le déplacement du véhicule entre deux images successives à partir de l'homographie précédemment calculée et de mettre à jour la trajectoire du véhicule. Dans sa version actuelle le calcul de la trajectoire se base uniquement sur l'homographie, cependant dans une future version nous envisageons de rajouter une fusion avec d'autres capteurs (GNSS par exemple). À l'heure actuelle, le *kernel* effectue très peu de calculs : 3 multiplications de matrices de petite taille. Du fait de la faible quantité de calcul nous choisissons de forcer le *mapping* de ce *kernel* sur CPU.

6.2.3 Prédiction et *mapping*

Après avoir défini les contraintes temps-réel et défini le graphe de traitement de l'algorithme, nous pouvons maintenant passer à l'étude d'embarquabilité sur Tegra K1 et Tegra X1. Remarquons que le nombre d'itérations N du *kernel* k_3 n'est pas encore déterminé. On cherchera donc à déterminer la valeur maximale de N permettant de respecter les contraintes temps-réel. Comme discuté précédemment, les *kernels* k_1 et k_4 ne peuvent pas s'exécuter sur GPU. À l'heure actuelle, le *kernel* k_4 représente une complexité de calcul très faible par rapport à k_2 ou k_3, nous choisissons donc de focaliser l'étude uniquement sur les *kernel* k_2 et k_3. Concernant les transferts, $\delta_{3,4}$ représente un délai très faible, puisqu'il implique la copie de seulement 12 octets.

Les prédictions des temps d'exécutions de k_2 et k_3 obtenues pour les processeurs du Tegra K1 et du Tegra X1 sont données dans le tableau 6.1. On remarque que les temps d'exécutions réels mesurés t_{real} sont tous conformes aux prédictions.

TABLEAU 6.1 – Prédictions et mesures des temps d'exécutions des *kernels* de l'odométrie visuelle, données en millisecondes. Pour le *kernel* k_3, les temps sont donnés en fonction du nombre d'itérations N.

	Tegra K1						**Tegra X1**					
	A15 : 1 cœur			GPU K1			A57 : 1 cœur			GPU X1		
	t_{min}	t_{max}	t_{real}	t_{min}	t_{max}	t_{real}	t_{min}	t_{max}	t_{real}	t_{min}	t_{max}	t_{real}
k_2	80.0	131.6	114	3.04	3.53	3.50	43.3	71.2	60	1.88	2.24	1.93
k_3	$100N$	$190N$	$149N$	$1.53N$	$2.64N$	$2.3N$	$79.3N$	$111N$	$83N$	$1.00N$	$1.97N$	$1.3N$

Ensuite, nous donnons dans le tableau 6.2 les temps de transfert prédits comparés aux mesures réelles. On peut remarquer que la prédiction est mauvaise pour le temps de transfert entre l'ARM A15 et le GPU du Tegra K1. Il faut savoir que dans l'implémentation réelle de l'algorithme, nous avons choisi une fonctionnalité spécifique de l'API CUDA pour ce transfert : *cudaMemcpy2D*. Cette fonctionnalité permet de gérer la copie d'un tableau 2D, comme une image, dont la taille mémoire est volontairement plus grande pour garantir un bon alignement en mémoire de chaque ligne du tableau. Le contexte est donc légèrement différent de celui de nos vecteurs de test, ce qui explique l'erreur sur la prédiction. Remarquons que les mêmes prédictions sont valides pour le Tegra X1, alors que nous utilisons la même technique de transfert.

TABLEAU 6.2 – Prédictions et mesures des temps de transferts pour l'odométrie visuelle, données en milliseconde. On peut remarquer une erreur sur la prédiction pour le transfert A15 vers le GPU Tegra K1.

	Tegra K1						**Tegra X1**					
	A15 → GPU			GPU → A15			A57 → GPU			GPU → A57		
	t_{min}	t_{max}	t_{real}	t_{min}	t_{max}	t_{real}	t_{min}	t_{max}	t_{real}	t_{min}	t_{max}	t_{real}
$\delta_{1,2}$	1.82	1.93	2.92				1.16	1.64	1.21			
$\delta_{2,3}$	1.82	1.93	2.92	2.10	3.98	2.54	1.16	1.64	1.21	1.20	2.39	1.29

À propos du *mapping*, les *kernels* k_1 et k_4 ne peuvent pas être exécutés sur GPU, ils sont donc mappés sur le CPU. La discussion se porte donc sur le *mapping* de k_2 et k_3. En l'utilisant l'arithmétique des intervalles, on remarque que le *mapping* de k_2 sur ARM ne permet pas de respecter la contrainte temps-réel de rythme :

$$\tau_{2|A15} = [80, 131.6] \not< T_c = 20 \tag{6.21}$$

$$\tau_{2|A57} = [43.3, 71.2] \not< T_c = 20 \tag{6.22}$$

Le même constat peut être fait pour le *mapping* de k_3 sur ARM, quel que soit le nombre d'itérations N :

$$\tau_{3|A15} = [100N, 190N] \not< T_c = 20 \tag{6.23}$$

$$\tau_{3|A57} = [79.3N, 111N] \not< T_c = 20 \tag{6.24}$$

Pour Tegra K1 et Tegra X1, le seul *mapping* valable est donc d'exécuter k_2 et k_3 sur GPU.

Intéressons-nous maintenant à l'impact du nombre d'itérations N sur l'embarquabilité de l'algorithme. Pour respecter la contrainte de rythme, on doit avoir :

$$\tau_{2|GPU} + \tau_{3|GPU} < T_c \tag{6.25}$$

soit pour Tegra K1 :

$$\begin{aligned} &[3.04, 3.53] + [1.53N, 2.53N] < 20 \\ \Rightarrow &N < \frac{20 - 3.53}{2.53} = 6.51 \end{aligned} \tag{6.26}$$

et pour Tegra X1 :

$$\begin{aligned} &[1.88, 2.24] + [1.00N, 1.97N] < 20 \\ \Rightarrow &N < \frac{20 - 2.24}{1.97} = 9.02 \end{aligned} \tag{6.27}$$

De la même manière, cette approche nous permet d'affirmer qu'un nombre d'itérations $N > 11.08$ pour Tegra K1 et $N > 18.12$ pour Tegra X1 ne permet pas de respecter la contrainte temps-réel de rythme. Cependant l'arithmétique des intervalles ne nous permet pas de répondre à propos du respect de cette contrainte dans le cas où $N \in [7, 11]$ pour Tegra K1 ou $N \in [10, 18]$. Dans ce cas, nous sommes obligés d'utiliser une autre approche : stochastique ou par fonction de coût.

Pour aller plus loin, on se propose d'appliquer l'approche stochastique pour déterminer la probabilité du respect de la contrainte de rythme sur GPU. Ainsi, le temps de calcul sur GPU est donné par :

$$t_{pu_{GPU}} = \tau_2 + \tau_3; \qquad f_{\tau_2+\tau_3} = f_{\tau_2} * f_{\tau_3} \tag{6.28}$$

où $f_{\tau_2+\tau_3}$, f_{τ_2} et f_{τ_3} représentent respectivement les PDFs de $t_{pu_{GPU}}$, τ_2 et τ_3. On choisit de représenter les variables aléatoires τ_2 et τ_3 par des distributions uniformes. On est alors capable de calculer la PDF $f_{\tau_2+\tau_3}$ pour Tegra K1 :

$$f_{\tau_2+\tau_3}(t) = \begin{cases} \dfrac{1}{0.5439 \times N} \times t - \dfrac{3.04 + 1.53 \times N}{0.5439 \times N} & \text{si } 3.04 + 1.53 \times N < t < 3.53 + 1.53 \times N \\ \dfrac{1}{1.11 \times N} & \text{si } 3.53 + 1.53 \times N \leqslant t \leqslant 3.04 + 2.64 \times N \\ -\dfrac{1}{0.5439 \times N} \times t + \dfrac{3.53 + 2.64 \times N}{0.5439 \times N} & \text{si } 3.04 + 2.64 \times N < t < 3.53 + 2.64 \times N \\ 0 & \text{sinon} \end{cases} \tag{6.29}$$

et pour Tegra X1 :

$$f_{\tau_2+\tau_3}(t) = \begin{cases} \frac{1}{0.3492 \times N} \times t - \frac{1.88 + 1.00 \times N}{0.3492 \times N} & \text{si } 1.88 + 1.00 \times N < t < 2.24 + 1.00 \times N \\ \frac{1}{0.97 \times N} & \text{si } 2.24 + 1.00 \times N \leqslant t \leqslant 1.88 + 1.97 \times N \\ -\frac{1}{0.3492 \times N} \times t + \frac{2.24 + 1.97 \times N}{0.3492 \times N} & \text{si } 1.88 + 1.97 \times N < t < 2.24 + 1.97 \times N \\ 0 & \text{sinon} \end{cases} \tag{6.30}$$

À titre d'exemple, la figure 6.4 illustre les PDFs pour Tegra K1 et Tegra X1 lorsque le nombre d'itérations $N = 10$.

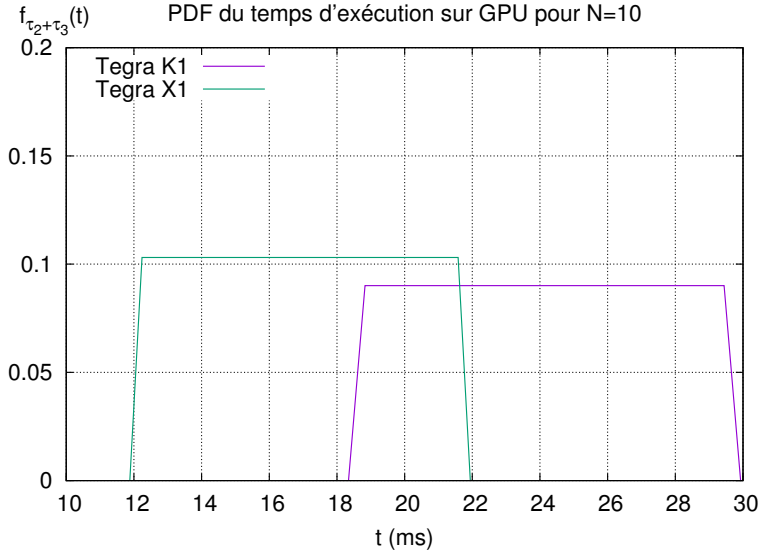


FIGURE 6.4 – Densités de probabilité du temps d'exécution pour Tegra K1 et Tegra X1

Finalement, la probabilité de respecter la contrainte de rythme est donnée par :

$$\Pr[t_{pu_{GPU}} < T_c] = \int_0^{T_c} f_{\tau_2+\tau_3}(x)dx \tag{6.31}$$

Si l'on choisit de fixer le nombre d'itérations à 10, on obtient alors une probabilité de respecter la contrainte de rythme sur Tegra K1 :

$$\Pr[t_{pu_{GPU}} < 20] = \frac{1.415}{11.1} \approx 0.127 \tag{6.32}$$

et sur Tegra X1 :

$$\Pr[t_{pu_{GPU}} < 20] = \frac{7.94}{9.7} \approx 0.819 \tag{6.33}$$

Ainsi, le SoC Tegra X1 permet d'exécuter l'algorithme d'odométrie visuelle avec $N = 10$ en respectant la contrainte de rythme avec une probabilité de 81.9%, contre seulement 12.7% pour le Tegra K1.

6.3 Détection de piétons

Pour les applications ADAS et la voiture autonome, la détection de piétons représente un enjeu majeur. Dans un futur proche (Horizon 2020), les fonctionnalités AEB intégreront ce type de détection sur la majorité des véhicules dans le but de pouvoir activer un freinage d'urgence pour éviter la collision avec un piéton. L'aspect temps réel de ce type de détection est critique, il est impératif de respecter la contrainte de latence, sans quoi le système risquerait de réagir trop tard et de ne pas pouvoir éviter la collision.

6.3.1 Description de l'algorithme

Pour illustrer notre méthodologie de l'analyse de l'embarquabilité, nous choisissons d'implémenter et d'embarquer un algorithme de détection de piétons basé sur l'implémentation de DALAL et TRIGGS [2005]. Cet algorithme utilise le descripteur HOG et un apprentissage SVM pour détecter les piétons sur une image. Nous proposons de décrire l'algorithme par le graphe de traitement donné en figure 6.5, graphe qui est composé de 8 *kernels* :

— k_1 : correction de la distorsion de l'image
— k_2 : gradient horizontal
— k_3 : gradient vertical
— k_4 : norme du gradient
— k_5 : orientation du gradient
— k_6 : construction de l'histogramme pour chaque cellule (8×8 pixels dans notre cas) et des blocs du descripteur HOG (4×4 cellules).
— k_7 : classification à l'aide d'un SVM linéaire
— k_8 : loi de commande automobile

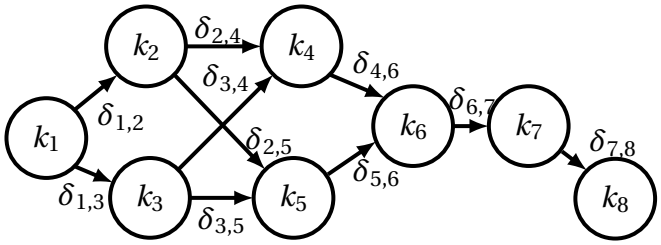


FIGURE 6.5 – Graphe de traitement de l'algorithme de détection de piétons

La caméra utilisée retourne une image de 1280×720 pixels, en niveau de gris, à une cadence de 200 images par seconde. On se place donc avec une contrainte de rythme $T_c = 5$ ms, correspondant à un cas d'usage très particulier de haute vitesse. De plus, nous choisissons d'imposer une contrainte de latence $dt_{spec}(1,8) = 15$ ms.

6.3.2 Analyse de l'embarquabilité

Pour embarquer cet algorithme de traitement d'images, nous proposons d'étudier les trois SoCs suivant : Tegra K1, Tegra X1 et R-Car H2. En fonction du nombre d'unités d'exécution proposées par chacun de ces SoCs, nous sommes capables de déterminer le nombre de *mappings* possibles :

— K1 (4 A15, 1 GPU) : $5^8 = 390\,625$ *mappings* possibles,

— X1 (4 A57, 4 A53, 2 SMX GPU) : 10^8 *mappings* possibles,

— R-Car H2 (using 4 A15 and 4 A7) : $8^8 = 16\,777\,216$ *mappings* possibles.

Cela fait donc un total de 117 167 841 à explorer, ce qui représenterait un travail colossal si l'exploration devait s'effectuer à la main. Remarquons que le *kernel* k_8 étant une loi de commande automobile, il ne peut pas s'exécuter sur GPU. De plus le *kernel* k_1 est le seul qui puisse s'exécuter sur l'unité spécialisée ISP de Tegra K1 ou de Tegra X1.

Nous choisissons de donner uniquement les résultats de l'exploration de *mappings* sur le Tegra K1, cependant la même méthodologie peut s'appliquer aux deux autres SoCs ou à n'importe quel autre SoC. Les prédictions et les mesures réelles des temps d'exécutions des différents *kernels* pour l'ARM A15 et le GPU du Tegra K1 sont données dans le tableau 6.3. Les cas où les performances sont limitées par les capacités de calcul (forte I_A) sont surlignés en vert et les cas où les performances sont limitées par la mémoire (faible I_A) sont surlignés en jaune. De la même manière, les différents délais de transfert sont donnés dans le tableau 6.4. À partir de ces différentes valeurs, nous sommes capables, en utilisant l'approche par fonction de coût, de tester de manière automatique le respect des contraintes temps-réel pour l'ensemble des *mappings* possibles. Remarquons que même si le nombre de *mappings* possibles est très grand, l'approche par fonction de coût nous permet de tester très rapidement (quelques secondes avec un script Python) l'ensemble des possibilités. Finalement, nous donnons dans la figure 6.6 le coût et le taux d'occupations de chaque unité d'exécution pour différents *mappings*. La fonction de coût utilisée est définie conformément aux équations 3.24, 3.25 et 3.26, avec $\beta_l = 1$ et $\beta_r = 1$. Pour des raisons évidentes de lisibilité nous limitons l'affichage aux 100 meilleurs *mappings*. Remarquons que seulement moins de 40 *mappings* sur les 390 625 possibles ont un coût inférieur à $+\infty$ et sont susceptibles de respecter les contraintes temps-réels.

TABLEAU 6.3 – Prédictions et mesures des temps d'exécutions des *kernels* de l'algorithme de détections de piétons. Les temps sont donnés en milliseconde, les *kernels* à forte I_A sont surlignés en verts et ceux à faible I_A en jaune.

	ISP			**A15**			**GPU**		
	t_{min}	t_{max}	t_{real}	t_{min}	t_{max}	t_{real}	t_{min}	t_{max}	t_{real}
k_1	0.71	0.79	0.75	5.13	7.68	5.84	1.54	2.55	1.86
k_2				2.05	3.59	2.23	0.36	0.81	0.67
k_3				2.05	3.59	3.11	0.36	0.81	0.69
k_4				4.09	7.16	5.34	0.48	0.83	0.71
k_5				13.82	21.50	15.42	0.48	0.83	0.71
k_6				2.05	3.07	2.30	2.02	3.42	2.92
k_7				2.88	3.46	3.01	0.31	0.42	0.35
k_8				1.2×10^{-3}	1.6×10^{-3}	1.4×10^{-3}			

TABLEAU 6.4 – Prédictions et mesures des temps de transfert pour l'algorithme de détections de piétons. Les temps sont donnés en millisecondes.

	ISP→ GPU/ARM			**ARM → GPU**			**GPU → ARM**		
	t_{min}	t_{max}	t_{real}	t_{min}	t_{max}	t_{real}	t_{min}	t_{max}	t_{real}
$\delta_{1,2}$	0.65	1.09	0.67	0.65	1.09	0.67	0.94	1.25	0.98
$\delta_{1,3}$	0.65	1.09	0.67	0.65	1.09	0.67	0.94	1.25	0.98
$\delta_{2,4}$				1.06	1.73	1.13	1.79	3.26	1.86
$\delta_{2,5}$				1.06	1.73	1.13	1.79	3.26	1.86
$\delta_{3,4}$				1.06	1.73	1.13	1.79	3.26	1.86
$\delta_{3,5}$				1.06	1.73	1.13	1.79	3.26	1.86
$\delta_{4,6}$				0.65	1.09	0.67	0.94	1.25	0.98
$\delta_{5,6}$				0.65	1.09	0.67	0.94	1.25	0.98
$\delta_{6,7}$				0.44	0.73	0.48	0.63	0.86	0.65
$\delta_{7,8}$							0.08	0.19	0.09

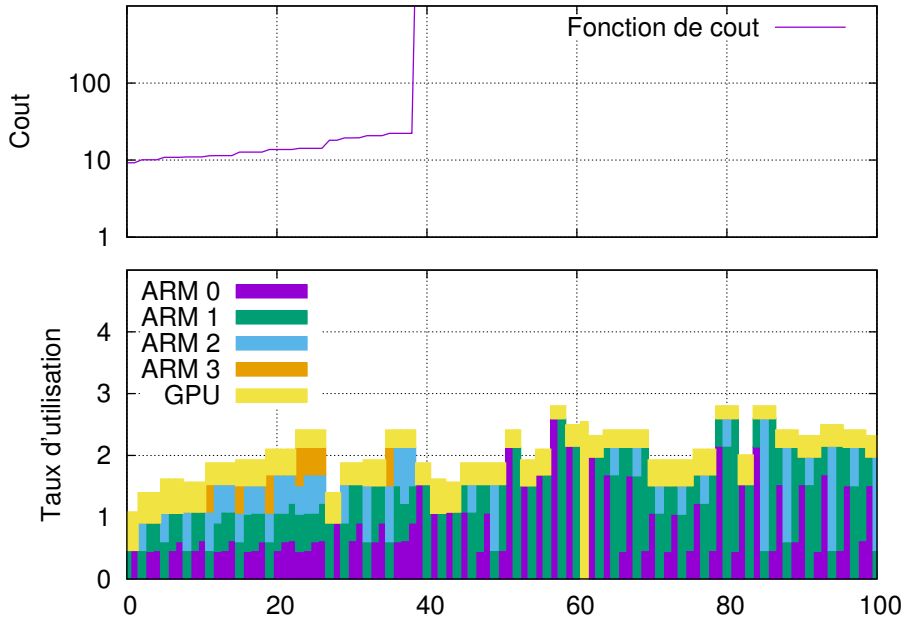


FIGURE 6.6 – Coût et taux d'occupation obtenues pour les 100 *mappings* au plus faible coût par rapport aux 390 625 possibilités pour le Tegra K1. Finalement, moins de 40 *mappings* sont susceptibles de respecter les contraintes temps-réel.

Le meilleur *mapping* résultant est alors le suivant : k_1 sur ISP, k_2, k_3, k_4, k_5 et k_7 sur GPU ($\eta_{GPU} = 0.63$), k_6 sur ARM 2 ($\eta_{ARM\,2} = 0.46$) et k_8 sur ARM 3 ($\eta_{ARM\,3} = 2.8 \times 10^{-4}$). Après la définition du pipeline d'exécution, la latence obtenue est $t_p(1,8) = 8.75$ ms. Ce *mapping* est suffisamment bon pour respecter les contraintes temps-réel, cependant notre modèle de prédiction de performance nous indique que l'ensemble des *kernels* exécutés sur GPU ont des performances limitées par la mémoire (faible I_A), la performance de chacun de ces *kernels* peut donc être augmentée en diminuant le nombre d'accès à la mémoire globale. Ainsi, une possibilité pour aller dans ce sens consiste à regrouper les quatre *kernels* k_2, k_3, k_4 et k_5 en un seul et de stocker les résultats temporaires dans une mémoire locale et rapide comme la *shared memory*.

6.4 Références

CIARLET, P.-G. 1982, *Introduction a l'analyse numerique matricielle et à l'optimisation*, Masson, Gordon & Breach Science Publishers. 188

DALAL, N. et B. TRIGGS. 2005, «Histograms of oriented gradients for human detection», dans *2005 IEEE Conference on Computer Vision and Pattern Recognition*, vol. 1, IEEE, p. 886–893. 184, 194

FRAUNDORFER, F. et D. SCARAMUZZA. 2012, «Visual odometry : Part II : Matching, robustness, optimization, and applications», *IEEE Robotics & Automation Magazine*, vol. 19, nº 2, p. 78–90. 184

LOVEGROVE, S., A. J. DAVISON et J. IBANEZ-GUZMÁN. 2011, «Accurate visual odometry from a rear parking camera», dans *Intelligent Vehicles Symposium (IV), 2011 IEEE*, IEEE, p. 788–793. 185, 186

MALIS, E. 2004, «Improving vision-based control using efficient second-order minimization techniques», dans *2004 IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04.*, vol. 2, IEEE, p. 1843–1848. 185, 187

MAYBECK, P. S. 1982, *Stochastic models, estimation, and control*, vol. 3, Academic press. 184

MEI, C., S. BENHIMANE, E. MALIS et P. RIVES. 2008, «Efficient homography-based tracking and 3-d reconstruction for single-viewpoint sensors», *IEEE Transactions on Robotics*, vol. 24, nº 6, p. 1352–1364. 187

OHNO, K., T. TSUBOUCHI, B. SHIGEMATSU, S. MAEYAMA et S. YUTA. 2003, «Outdoor navigation of a mobile robot between buildings based on dgps and odometry data fusion», dans *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, vol. 2, IEEE, p. 1978–1984. 184

VALGAERTS, L., A. BRUHN, M. MAINBERGER et J. WEICKERT. 2012, «Dense versus sparse approaches for estimating the fundamental matrix», *International Journal of Computer Vision*, vol. 96, nº 2, p. 212–234. 184

Chapitre 7

Conclusion et perspectives

« *"Forty-two !" yelled Loonquawl. "Is that all you've got to show for seven and a half million years' work ?" "I checked it very thoroughly," said the computer, "and that quite definitely is the answer. I think the problem, to be quite honest with you, is that you've never actually known what the question is."* »

Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

Sommaire

7.1 Synthèse

Avec l'augmentation des fonctionnalités ADAS et l'arrivée prochaine de la voiture autonome, l'industrie automobile a un besoin croissant de puissance de calcul à embarquer dans les véhicules. La méthodologie présentée dans cette thèse est destinée à participer au choix d'une architecture embarquée pour l'exécution d'algorithmes de traitement d'images en temps-réel. Cette thèse a mené à la définition d'une méthodologie globale d'analyse de l'embarquabilité comprenant trois contributions majeures : caractérisation d'architectures, prédiction de performances et méthode de *mapping*.

7.1.1 Caractérisation d'architectures

Une nouvelle méthodologie de caractérisation d'architecture a été proposée, il s'agit de l'extraction de caractéristiques par le biais de vecteurs de test. Ces vecteurs de test sont regroupés en trois catégories :

Low-level : extraction des caractéristiques bas niveaux tels que les comportements des accès mémoire, les capacités de calcul pour différentes opérations et différents types de variables. Ces caractéristiques sont utilisées pour la prédiction du temps d'exécution d'un *kernel* et des temps de transfert entre processeurs.

Mid-level : étude des comportements lors d'exécutions concurrentes de plusieurs *kernels*, sur l'aspect de l'accès mémoire et de la capacité de calcul. Les résultats permettent entre autres de déterminer s'il vaut mieux exécuter deux *kernels* indépendants séquentiellement ou de manière concurrente sur un même processeur.

High-level : étude et modélisation du temps d'exécution de fonctionnalités d'une bibliothèque binaire, sans que le code source soit accessible, ou d'une application avec un code source ouvert. À partir de ces résultats, il devient possible de prédire un temps d'exécution pour différentes paramètres et pour plusieurs architectures. Ces vecteurs de test sont également utilisés comme une étape essentielle de validation d'un modèle d'architecture.

Ces vecteurs de test permettent de construire une modélisation complète d'un ensemble composé d'une architecture de calcul, d'un OS et d'un compilateur. Contrairement aux benchmarks de l'état de l'art, cette méthodologie a pour avantage d'étudier l'ensemble des caractéristiques d'une architecture, c'est-à-dire à la fois les comportements des opérations mémoire et des différentes opérations de calcul dans des conditions fidèles à des applications réelles.

Cette méthodologie a donné lieu à une implémentation en C++, basée sur des templates génériques et des opérations spécifiques à chaque architecture, comme l'utilisation de NEON pour les processeurs ARM ou de CUDA pour les GPU.

7.1.2 Prédiction de performances

À partir d'un modèle d'architecture existant et d'un algorithme à embarquer, une méthode de prédictions de performances a été définie. Cette méthodologie est capable de prédire un intervalle de temps d'exécution pour un *kernel* donné et un intervalle de temps de transfert entre deux processeurs. Elle s'appuie sur le nombre et les types d'opérations de calcul, et le nombre et les types d'accès mémoire de ce *kernel*, puis la capacité de calcul et les caractéristiques de la mémoire de l'architecture obtenues via les vecteurs de test low-level ou depuis une datasheet.

Par la suite, cet intervalle peut être exprimé sous forme d'une densité de probabilité. Par ailleurs, il a été montré en utilisant la théorie de probabilités il est possible de combiner les prédictions sur les temps d'exécution de plusieurs *kernels*, de manière à obtenir une densité de probabilité du temps d'exécution complet d'un ensemble de *kernels*.

7.1.3 Mapping et embarquabilité

Une nouvelle méthodologie pour l'analyse de l'embarquabilité des algorithmes de traitement d'images sur des architectures hétérogènes a été présentée dans cette thèse. Cette méthodologie s'appuie sur la prédiction de performances pour déterminer sans effort de portage si un algorithme donné peut s'exécuter en temps-réel, c'est-à-dire en respectant les contraintes de rythme et de latence, sur un ensemble d'architectures matérielles. De plus, toujours en utilisant la prédiction de performances, une méthodologie de *mappings* sur SoCs hétérogènes a été défini. Nous avons proposé trois approches différentes :

Stochastique qui permet d'obtenir une prédiction fine et proche de la réalité mais reste cependant très complexe à calculer.

Intervalle qui permet une réponse rapide et directe sur l'embarquabilité, mais est parfois non concluante.

Fonction de coût qui permet d'obtenir une solution du meilleur *mapping* en laissant le choix à l'utilisateur de privilégier soit la contrainte temps-réel de rythme soit la containte temps-réel de latence.

Il a été montré que le très grand nombre de *mappings* possibles peut être exploré automatiquement en utilisant la méthodologie de prédiction de performances.

Cette méthodologie globale propose un processus simple permettant de donner une réponse rapide à propos de l'embarquabilité d'un algorithme et du respect des contraintes temps-réel sans aucune intervention d'un expert ayant une connaissance approfondie d'une architecture en particulier. Cependant, si pour un graphe donné la méthodologie échoue ou si les performances prédites sont trop proches des contraintes temps-réel, il est alors nécessaire d'envisager un nouveau partitionnement en *kernels* et d'utiliser des accélérations matérielles spécifiques (par exemple NEON ou un ISP) ou alors d'envisager l'utilisation d'un autre SoC. Le processus d'analyse n'est cependant pas automatisable à 100% et nécessite des utilisateurs ayant une connaissance basique des écosystèmes liés aux architectures embarquées.

7.2 Perspectives et applications

Les résultats présentés dans cette thèse amènent à plusieurs perspectives : d'une part les applications industrielles au sein de Renault, mais également les travaux de recherche futurs pouvant s'appuyer sur cette thèse.

7.2.1 Réponse au besoin industriel

La thèse a été initiée par Renault dans le but de répondre à la problématique d'embarquabilité, l'application majeure des résultats de cette thèse est donc logiquement de répondre à ce besoin industriel. Pour rappel, cela concerne le dimensionnement d'un ECU pour les ADAS et il est détaillé dans la section 1.4.2. Notre méthodologie d'analyse

de l'embarquabilité peut s'intégrer dans le processus amont tel qu'il est illustré dans la figure 7.1.

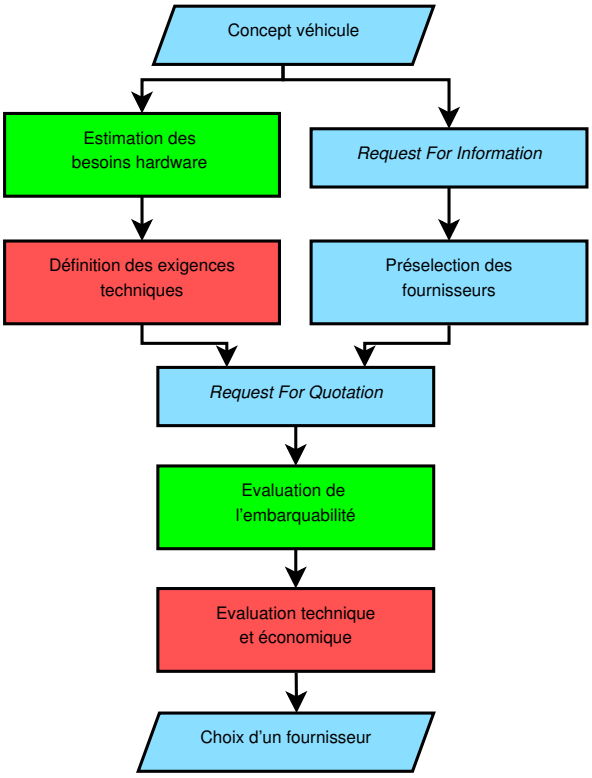


FIGURE 7.1 – Réponse au besoin industriel dans le processus amont du projet véhicule. L'estimation des besoins hardware permet de définir de manière précise les exigences techniques concernant les capacités de calcul nécessaires. L'évaluation de l'embarquabilté permet de tester l'ensemble des solutions proposées par les fournisseurs et ainsi de déterminer le meilleur choix sur des critères technico-économiques.

Ainsi, nous envisageons d'utiliser notre prédiction de performance sur l'ensemble des applications ADAS à embarquer pour déterminer les caractéristiques minimales que doit avoir un ECU pour pouvoir exécuter ces applications en respectant les contraintes temps-réel. Il peut par exemple s'agir de valeurs de capacités de traitement et de la fréquence d'horloge, de temps d'accès en lecture et écriture, etc. Cette estimation des besoins hardware permet alors de définir de manière précise l'ensemble des exigences techniques concernant les capacités de calcul de l'ECU. Remarquons que nous n'adressons pas ici les besoins en termes de quantité de mémoire nécessaire pour l'exécution des applications. Cependant les applications automobiles utilisent très peu d'allocations dynamiques, la quantité de mémoire nécessaire est donc stable et peu complexe à déterminer.

Ensuite, lors de la phase de RFQ, nous envisageons de demander à l'ensemble des fournisseurs d'exécuter une partie de nos vecteurs de test puis de nous retourner les résultats. Ainsi, à partir de ces données il est alors possible de construire un modèle d'architecture pour chaque solution proposée par les fournisseurs et donc d'appliquer notre méthodologie d'analyse de l'embarquabilité globale telle que décrite dans la section 3.4. De cette manière l'évaluation des solutions fournisseurs devient très rapide et simple sur le plan technique.

Notons que ces modifications de la procédure ne sont pas encore mises en place. Cependant, il y a une volonté claire de la part de Renault de maîtriser au mieux le dimensionnement de l'ECU et l'évaluation des solutions proposées par les fournisseurs. Ainsi, notre

approche est en cours de mise en œuvre et sera appliquée lors des prochains projets.

7.2.2 Implémentation logicielle

Dans une continuité normale de la réponse au besoin industriel, il faut envisager une industrialisation de la méthodologie. Ceci peut se traduire par l'implémentation d'un *framework* complet permettant une utilisation simple et rapide de la méthodologie incluant les différentes étapes de la figure 3.7. Tout d'abord, il est important de noter qu'à l'heure actuelle la modélisation d'un algorithme est assez lourde à mettre en place : il est nécessaire de dénombrer à la main le nombre d'opérations pour chaque *kernel* de l'algorithme. Il y a donc tout intérêt à automatiser cette opération. Nous pensons que cela peut se faire par une analyse syntaxique automatique d'une implémentation générique en C de l'algorithme, en utilisant par exemple une représentation en *Abstract Syntax Tree* (AST) [JONES, 2003]. Cette représentation permet notamment d'analyser facilement l'implémentation d'un l'algorithme et ainsi dénombrer le nombre d'opérations et les types de données mises en œuvre.

Au cours de la thèse, nous avons implémenté une série de vecteurs de test ciblant divers processeurs. Bien évidement cette bibliothèque peut être enrichie avec l'ajout de nouveaux tests, permettant de caractériser d'autres spécificités d'un processeur. Il peut être également très intéressant de créer un outil permettant de fournir une exploitation automatique des données (par exemple régression linéaire par morceau). Par la suite, les résultats obtenus peuvent alors être stockés dans une base de données de modèles d'architectures de calcul. Ceci permet de réutiliser les modèles ainsi construits, pour une utilisation future visant ces même architectures de calcul, et évite donc de répéter indéfiniment l'étape de caractérisation.

On peut également imaginer une application un peu plus éloignée, comme un outil de conception d'algorithmes à partir de graphes de flux de données suivant les contraintes définies dans cette thèse, c'est-à-dire un graphe orienté, connecté, sans boucle et à arcs pondérés. Il peut être également envisagé de fournir une fonction de génération de code à partir du graphe conçu et donc de proposer un portage sur une cible embarquée semi-automatique. De plus, en fournissant plusieurs modèles d'architectures, cet outil pourrait déterminer automatiquement le SoC le plus adapté et une solution à la problématique de *mapping*.

7.2.3 Approche stochastique

Concernant l'approche stochastique, nous avons choisi de modéliser, pour des raisons de simplicité des calculs, une prédiction par une distribution uniforme ou gaussienne. Les prédictions des délais mémoire étant directement basées sur les mesures effectuées par les vecteurs de test, il peut être intéressant d'utiliser une modélisation qui soit plus proche de la mesure. Ainsi, comme présenté par WORMS et TOUATI [2016], une modélisation basée sur des mélanges de gaussiennes permet une représentation plus fidèle des temps d'exécutions mesurés.

L'approche stochastique permet de répondre avec précision à la probabilité de respecter ou non les contraintes temps-réel. Cependant, certaines variables aléatoires n'étant pas indépendantes, les calculs deviennent assez rapidement très complexes. Au cours de cette thèse, nous avons choisi de ne pas creuser cette approche, il pourrait être intéressant de l'étudier plus en profondeur en utilisant par exemple les lois de probabilités marginales [EVERITT, 1998].

7.2.4 Autres types d'algorithmes

Au cours de cette thèse, nous avons choisi de nous focaliser sur des algorithmes de traitement d'images puisqu'ils représentent une très grande charge de calcul à cause du volume de données à traiter. Pour ce type d'algorithme, la taille des données à traiter est généralement fixe (images de taille fixe) et la quantité de calcul est également souvent fixe, ce qui facilite grandement l'étude de l'embarquabilité.

Dans le contexte automobile, et en se positionnent sur des applications à court terme, le besoin d'analyse de l'embarquabilité se situe au niveau des algorithmes de fusion de données. Ainsi ils représentent une charge de calcul beaucoup plus importante que des lois de commande automobile et peuvent donc amener à demander un ECU plus puissant. Le besoin immédiat est donc l'application de l'analyse de l'embarquabilité sur ce type d'algorithme pour s'intégrer dans le processus industriel du choix de l'ECU. Par opposition au traitement d'images, la taille des données d'entrée n'est pas fixe. Une approche par le pire cas peut être envisagée.

7.2.5 Autres architectures

Au cours de cette thèse, la plupart des implémentations, des mesures et des validations ont été faites sur des plateformes Nvidia. Ainsi, les SoC fournis par ce fondeur présentaient, au cours de la thèse, une plus grande maturité en terme d'environnement de développement. Par exemple, à l'heure actuelle seul Nvidia propose une implémentation OpenVX (le framework VisionWorks) compatible pour ses architectures hétérogènes.

Nous avons présenté d'autres SoC provenant d'autres constructeurs (Renesas, Texas Instrument), en théorie les processeurs massivement parallèles de ces fournisseurs sont compatibles avec notre méthodologie globale et la prédiction de performances. Cependant si un intérêt industriel se faisait sentir il pourrait être intéressant d'étudier de manière plus approfondie ces plateformes, par exemple le processeur vectoriel EVE de Texas Instrument [MANDAL et collab., 2014], ou l'IMP-X4 de Renesas.

Dès lors que d'autres fondeurs proposeront une implémentation OpenVX compatible avec leurs SoCs, le portage d'une architecture à une autre sera simplifié. Les vecteurs de test high-level permettront alors une comparaison simple et rapide des performances des différents accélérateurs hardware, pour des *kernels* donnés.

7.2.6 Vers le véhicule autonome

Notre processus d'embarquabilité permet la simplification du passage du prototypage d'un algorithme à son exécution temps-réel sur une plateforme embarquée. Comme discuté au début de ce manuscrit, les fonctionnalités ADAS tendent à se complexifier et à proposer des services des plus en plus aboutis jusqu'à obtenir une automatisation complète de la conduite du véhicule. Pour atteindre le Graal d'une voiture autonome et sûre, le véhicule devra être capable de comprendre son environnement de manière au moins aussi efficace qu'un être humain. Cela implique donc une utilisation croissante de capteurs et un besoin en puissance de calcul de plus en plus fort à embarquer dans le véhicule. Sur ce point, les fondeurs de semi-conducteurs se positionnent pour répondre à ces besoins. Nvidia est un très bon exemple, la société sort chaque année des architectures hétérogènes de plus en plus puissantes pour la voiture autonome. Par exemple, la nouvelle plateforme DrivePX 2 est un monstre de calcul, elle atteint les 8 téraFLOPS ! Cependant l'enveloppe thermique de ses processeurs est de 250 Watts, la plateforme est

d'ailleurs équipée d'un refroidissement liquide. Le caractère « embarqué » d'un calculateur avec une enveloppe thermique aussi forte peut poser question et peut être très problématique pour une intégration dans un véhicule. Cela n'a cependant pas empêché le constructeur Tesla Motors de choisir d'équiper l'ensemble de ses véhicules avec ce calculateur depuis octobre 2016.

À propos de l'aspect perception et compréhension de l'environnement du véhicule, la stratégie actuelle employée aujourd'hui dans la recherche est d'utiliser des approches basées sur l'apprentissage profond, ou *deep learning* en anglais [LeCun et collab., 2015]. Concernant le traitement d'images, il s'agit d'apprendre à un réseau de neurones à détecter et à identifier des obstacles sur l'image en utilisant directement l'image brute en entrée, par opposition aux approches conventionnelles qui nécessitent un descripteur d'une part et un classificateur d'autre part. L'utilisation du *deep learning* va même plus loin que la perception et peut être utilisée pour agir directement sur les actionneurs. Par exemple Bojarski et collab. [2016], de la société Nvidia, ont proposé un système basé sur une seule caméra et un réseau de neurones agissant directement sur le contrôle latéral du véhicule. Le réseau de neurones est exécuté en temps-réel sur une DrivePX. L'apprentissage du réseau de neurones s'est effectué en enregistrant l'angle au volant appliqué par un conducteur réel en fonction des images perçues par la caméra. Cette vision d'un réseau de neurones maître du véhicule est sujet à débat. En effet, la compréhension d'un réseau de neurones se limite uniquement à des cas proches de sa base d'apprentissage. Si l'on se place hors de ce cadre, alors le réseau n'est plus capable de fonctionner, mais n'est également pas en mesure de savoir s'il sort de sa base de connaissance.

Finalement, si les problèmes techniques sont en voie d'être résolus, d'autres freins existent au sujet de la voiture autonome. Ainsi, l'accident mortel de mai 2016 causé par un système d'aide à la conduite de Tesla Motors nous rappelle que ces systèmes ne sont pas infaillibles et peuvent être la source d'accidents. Cela soulève une question sur le point de vue législatif : en cas d'un accident imputé à un véhicule autonome, qui est responsable ? Le constructeur, l'éditeur du logiciel ou le propriétaire du véhicule ? Enfin, il reste encore un énorme palier à franchir avant de pouvoir démocratiser le véhicule autonome : convaincre l'utilisateur. Ainsi, un sondage effectué par TNS Sofres [2016] en mai 2016 montre que 65% des Français appréhendent l'utilisation d'une voiture autonome et que seulement 18% pensent qu'elle répond à un réel besoin. De plus, le sondage indique que 82% répondent qu'ils resteront attentifs à la route en cas de conduite autonome, cela montre clairement le manque de confiance actuel des utilisateurs à propos de ce type de véhicule. Ce dernier élément peut sembler paradoxal : une très grande majorité des conducteurs ne sont pas encore prêts à faire confiance à une voiture autonome même si celle-ci présente un risque d'accident beaucoup plus faible qu'une conduite humaine.

7.3 Références

BOJARSKI, M., D. DEL TESTA, D. DWORAKOWSKI, B. FIRNER, B. FLEPP, P. GOYAL, L. D. JACKEL, M. MONFORT, U. MULLER, J. ZHANG et collab.. 2016, «End to end learning for self-driving cars», *arXiv preprint arXiv :1604.07316*. 205

EVERITT, B. 1998, *Cambridge dictionary of statistics*, Cambridge University Press. 203

JONES, J. 2003, «Abstract syntax tree implementation idioms», dans *Proceedings of the 10th conference on pattern languages of programs (PLOP)*, p. 26. 203

LECUN, Y., Y. BENGIO et G. HINTON. 2015, «Deep learning», *Nature*, vol. 521, nº 7553, p. 436–444. 205

MANDAL, D. K., J. SANKARAN, A. GUPTA, K. CASTILLE, S. GONDKAR, S. KAMATH, P. SUNDAR et A. PHIPPS. 2014, «An embedded vision engine (EVE) for automotive vision processing», dans *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, p. 49–52. 204

TNS SOFRES. 2016, «Les français et l'automobile (mai 2016)», Web ressource. http://www.tns-sofres.com/publications/les-francais-et-lautomobile-mai-2016. 205

WORMS, J. et S. TOUATI. 2016, «Going beyond mean and median programs performances», dans *IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-16)*, IEEE, p. 93–100. 203

Annexe A

Odométrie visuelle

A.1 Calcul des dérivées partielles

On cherche à évaluer pour $\mathbf{x} = \mathbf{0}$ et $\mathbf{x} = \mathbf{x_0}$:

$$\frac{\partial f_{p_b}(\mathbf{x})}{\partial x_i} = \frac{\partial I^a(H^{ab}(\mathbf{x}) \cdot p_b)}{\partial H^{ab}(\mathbf{x}) \cdot p_b} \cdot \frac{\partial H^{ab}(\mathbf{x}) \cdot p_b}{\partial x_i} \cdot p_b \tag{A.1}$$

où x_i correspond à la i^{e} composante du vecteur $\mathbf{x}$.

A.1.1 Calcul des dérivées de H^{ab}

Par définition, on a :

$$\mathbf{x} = \begin{pmatrix} u & v & \theta \end{pmatrix} \qquad H^{ab}(\mathbf{x}) = \begin{pmatrix} \cos\theta & -\sin\theta & u \\ \sin\theta & \cos\theta & v \\ 0 & 0 & 1 \end{pmatrix} \tag{A.2}$$

donc :

$$\frac{\partial H^{ab}(\mathbf{x})}{\partial u} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad \frac{\partial H^{ab}(\mathbf{x})}{\partial v} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \qquad \frac{\partial H^{ab}(\mathbf{x})}{\partial \theta} = \begin{pmatrix} -\sin\theta & -\cos\theta & 0 \\ \cos\theta & -\sin\theta & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{A.3}$$

Pour de petit $\mathbf{x_0}$, nous considérerons que les termes $\frac{\partial H^{ab}(\mathbf{0})}{\partial x_i}$ et $\frac{\partial H^{ab}(\mathbf{x_0})}{\partial x_i}$ sont équivalents et constants pour tous les pixels à une itération donnée.

A.1.2 Calcul des dérivées de I^a

Cas $\mathbf{x} = \mathbf{0}$

On cherche alors à évaluer :

$$\frac{\partial I^a(H^{ab}(\mathbf{0}) \cdot p_b)}{\partial H^{ab}(\mathbf{0}) \cdot p_b} = \frac{\partial I^a(p_b)}{\partial p_b} = \frac{\partial I^a(p_b)}{\partial p_a} \cdot \frac{\partial p_a}{\partial p_b} \tag{A.4}$$

De manière évidente, $\frac{\partial I^a(p_b)}{\partial p_a}$ correspond au gradient de l'image I^a évalué à la coordonné p_b :

$$\frac{\partial I^a(p_b)}{\partial p_a} = \begin{pmatrix} G_x^a(p_b) & G_y^a(p_b) & 0 \end{pmatrix} \tag{A.5}$$

où G_x^a correspond au gradient horizontal de I^a et G_y^a au gradient vertical de I^a. Ensuite, en sachant $p_a = H^{ab}(\mathbf{x}) \cdot p_b$:

$$\frac{\partial I^a(p_b)}{\partial p_a} \cdot \frac{\partial p_a}{\partial p_b} = \begin{pmatrix} G_x^a(p_b) & G_y^a(p_b) & 0 \end{pmatrix} \cdot \begin{pmatrix} \cos\theta & -\sin\theta & u \\ \sin\theta & \cos\theta & v \\ 0 & 0 & 1 \end{pmatrix} \tag{A.6}$$

$$= \begin{pmatrix} G_x^a(p_b)\cos\theta + G_y^a(p_b)\sin\theta & -G_x^a(p_b)\sin\theta + G_y^a(p_b)\cos\theta & G_x^a(p_b)u + G_y^a(p_b)v \end{pmatrix}$$

Cas $\mathbf{x} = \mathbf{x_0}$

Par définition on a $I^a(H^{ab}(\mathbf{x_0})\, p_b) = I^b(p_b)$, donc :

$$\frac{\partial I^a(H^{ab}(\mathbf{x_0}) \cdot p_b)}{\partial H^{ab}(\mathbf{x_0}) \cdot p_b} = \frac{\partial I^a(p_a)}{\partial p_a} = \frac{\partial I^b(p_b)}{\partial p_b} \tag{A.7}$$

où $\frac{\partial I^b(p_b)}{\partial p_b}$ correspond au gradient de l'image I^b évalué en p_b :

$$\frac{\partial I^b(p_b)}{\partial p_b} = \begin{pmatrix} G_x^b(p_b) & G_y^b(p_b) & 0 \end{pmatrix} \tag{A.8}$$

avec G_x^b le gradient horizontal, et G_y^b le gradient vertical de l'image I^b.

A.1.3 Calcul des dérivées de f_{p_b}

Cas $\mathbf{x} = \mathbf{0}$

En reprenant les équations précédentes, et en posant :

$$p_b = \begin{pmatrix} p_x^b \\ p_y^b \\ 1 \end{pmatrix} \tag{A.9}$$

on obtient alors :

$$\frac{\partial f_{p_b}(\mathbf{0})}{\partial u} = \frac{\partial I^a(p_b)}{\partial p_a} \cdot \frac{\partial p_a}{\partial p_b} \cdot \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot p_b = G_x^a(p_b)\cos\theta + G_y^a(p_b)\sin\theta \tag{A.10}$$

$$\frac{\partial f_{p_b}(\mathbf{0})}{\partial v} = \frac{\partial I^a(p_b)}{\partial p_a} \cdot \frac{\partial p_a}{\partial p_b} \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \cdot p_b = -G_x^a(p_b)\sin\theta + G_y^a(p_b)\cos\theta \tag{A.11}$$

$$\frac{\partial f_{p_b}(\mathbf{0})}{\partial \theta} = \frac{\partial I^a(p_b)}{\partial p_a} \cdot \frac{\partial p_a}{\partial p_b} \cdot \begin{pmatrix} -\sin\theta & -\cos\theta & 0 \\ \cos\theta & -\sin\theta & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot p_b \tag{A.12}$$

Cas $\mathbf{x} = \mathbf{x_0}$

De la même manière :

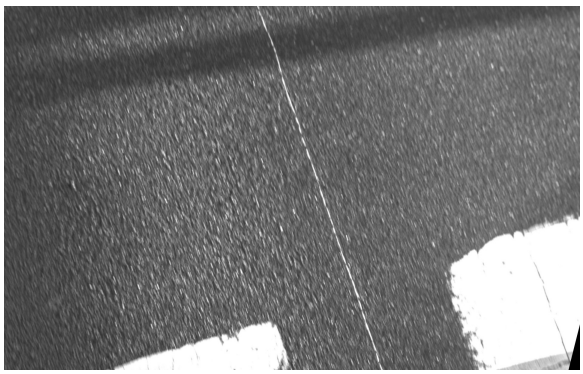
$$\frac{\partial f_{p_b}(\mathbf{x_0})}{\partial u} = \frac{\partial I^b(p_b)}{\partial p_b} \cdot \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot p_b = G_x^b(p_b) \tag{A.13}$$

$$\frac{\partial f_{p_b}(\mathbf{x_0})}{\partial v} = \frac{\partial I^b(p_b)}{\partial p_b} \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \cdot p_b = G_y^b(p_b) \tag{A.14}$$

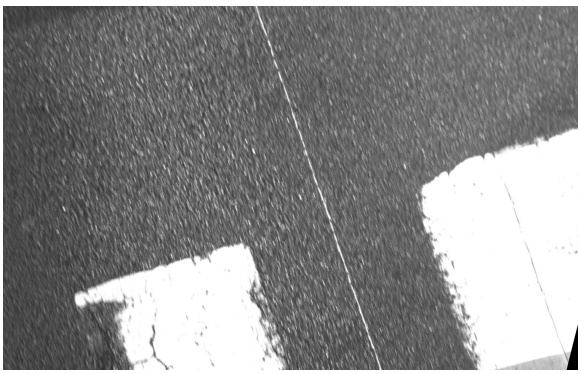
$$\frac{\partial f_{p_b}(\mathbf{x_0})}{\partial \theta} = \frac{\partial I^b(p_b)}{\partial p_b} \cdot \begin{pmatrix} -\sin\theta & -\cos\theta & 0 \\ \cos\theta & -\sin\theta & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot p_b \tag{A.15}$$

A.2 Résultats de l'algorithme

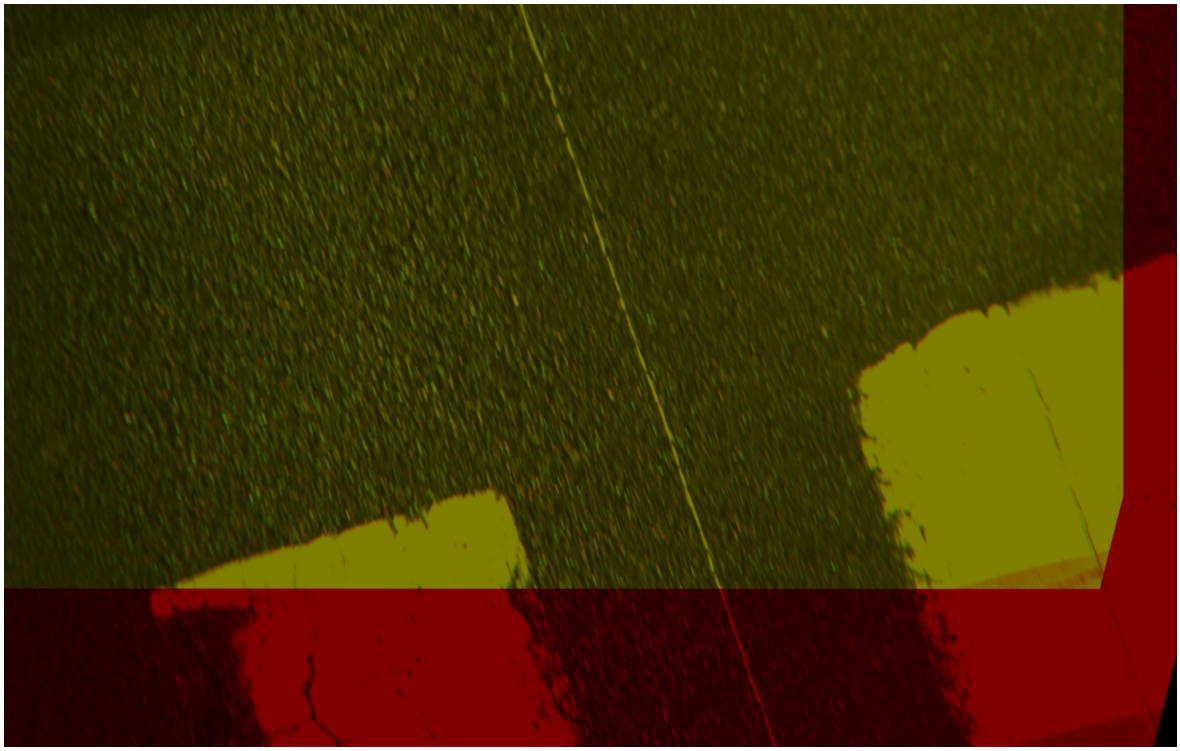
L'algorithme permet d'estimer l'homographie entre deux images consécutives. Pour illustrer sa performance, nous donnons en figure A.1 la superposition de deux images avec l'homographie calculée pour une vitesse du véhicule avoisinant les 50 km/h.



(a) Image précédente



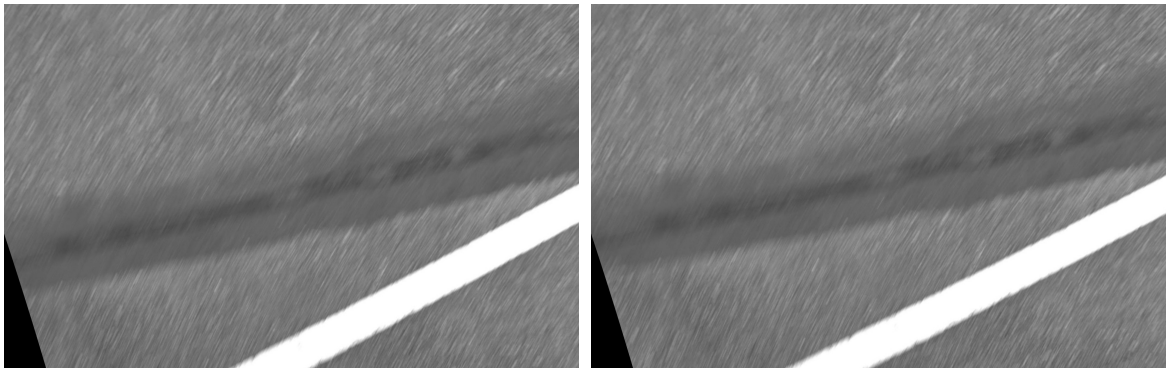
(b) Image courante



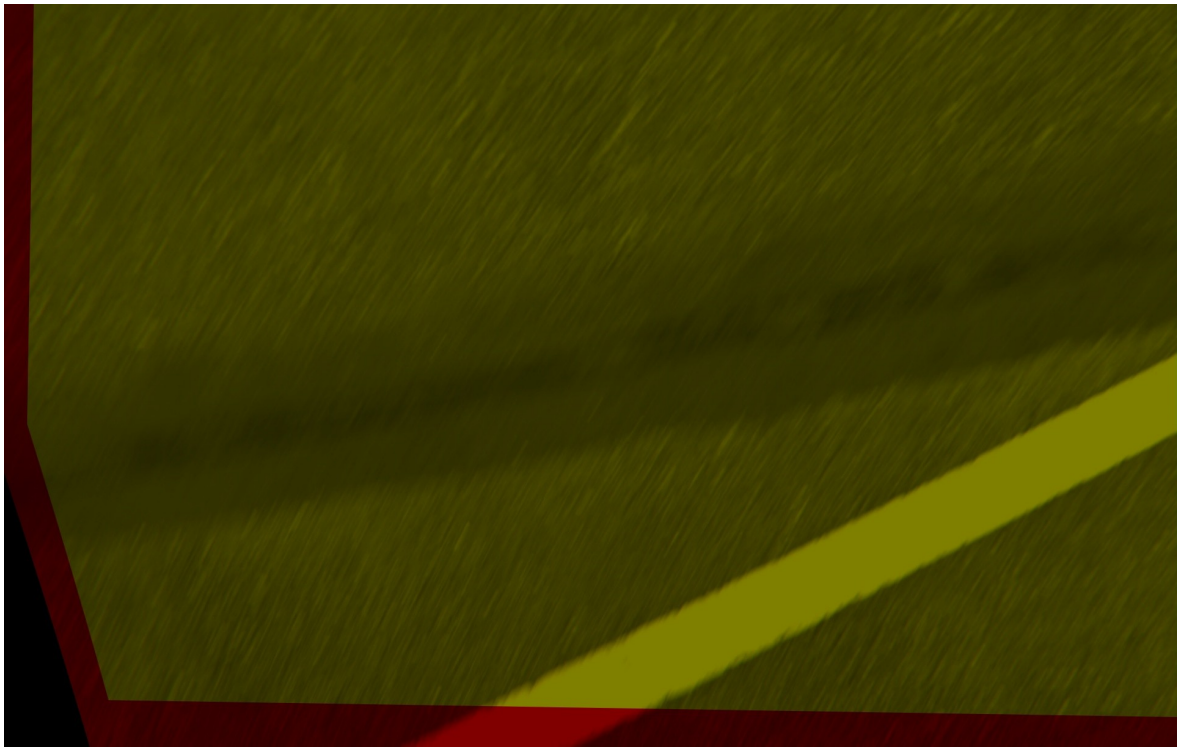
(c) Superposition

FIGURE A.1 – Superposition de deux images successives après application de l'homographie calculée par l'algorithme. L'image courante est en niveaux de rouge, l'image précédente est en niveaux de vert.

À titre d'exemple, la figure A.2 illustre l'homographie obtenue pour une autre acquisition avec un temps d'exposition de la caméra beaucoup plus grand et donc des images flous. Malgré la faible netteté des images, l'algorithme converge vers une homographie permettant une bonne superposition des deux images. Remarquons que cette acquisition correspond à de faibles vitesses du véhicule, moins de 15 km/h.



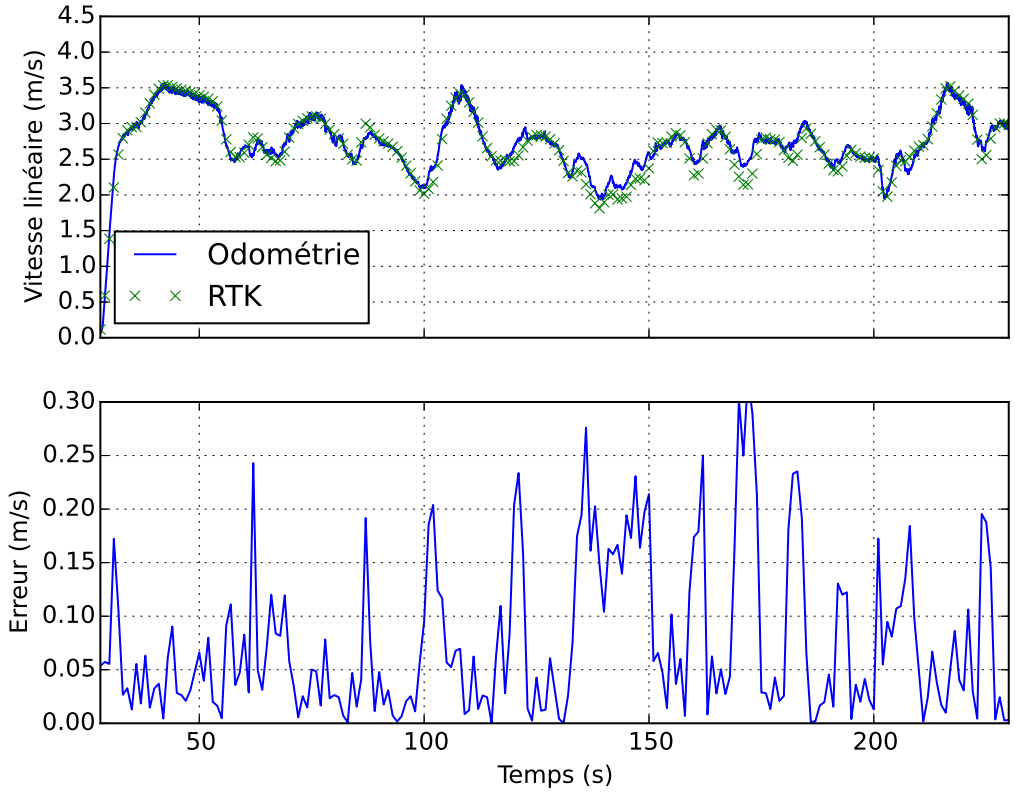
(a) Image précédente (b) Image courante



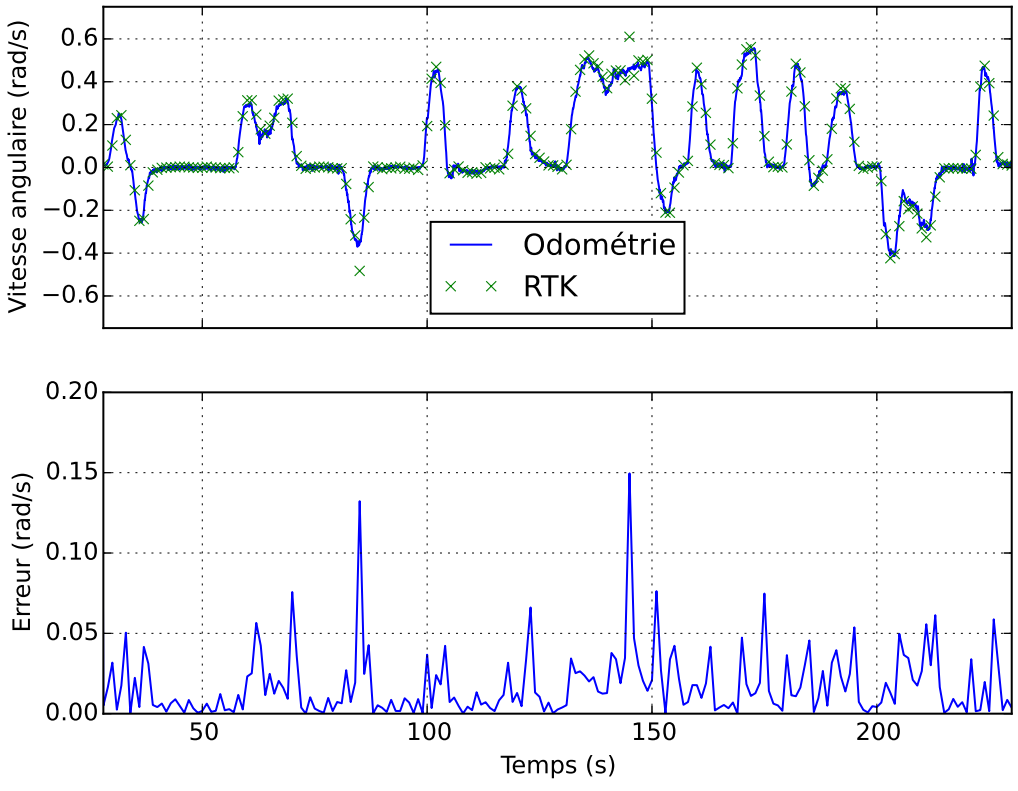
(c) Superposition

FIGURE A.2 – Superposition de deux images successives après application de l'homographie dans le cas d'une faible netteté. L'image courante est en niveaux de rouge, l'image précédente est en niveaux de vert.

À partir de l'homographie calculée entre deux images successives, il est possible de déterminer le déplacement effectué par le véhicule (rotation et translation). Ainsi, nous donnons en figure A.3 les vitesses linéaire et angulaire obtenues par l'algorithme et la vérité terrain obtenue par un capteur GPS RTK. Si les résultats de l'odométrie semblent cohérents avec la vérité terrain lorsque le véhicule se déplace en ligne droite, les courbes montrent une erreur plus grande sur l'estimation des vitesses lorsque le véhicule est en rotation. Cette erreur peut s'expliquer par le roulis du véhicule lors des virages qui n'est pas pris en compte par l'algorithme.



(a) Vitesse linéaire



(b) Vitesse angulaire

FIGURE A.3 – Vitesses linéaire et angulaire obtenues par l'odométrie visuelle comparées aux données issues d'un GPS RTK.

Finalement, nous donnons en figure A.4 la trajectoire du véhicule obtenue à partir de l'algorithme. Comme prévu, on observe une divergence de la trajectoire calculée à cause de l'erreur accumulée au cours du temps. Remarquons que la trajectoire diverge plus lorsque le véhicule effectue une rotation, conformément à nos observations sur les résultats de la vitesse angulaire et de la vitesse linéaire.

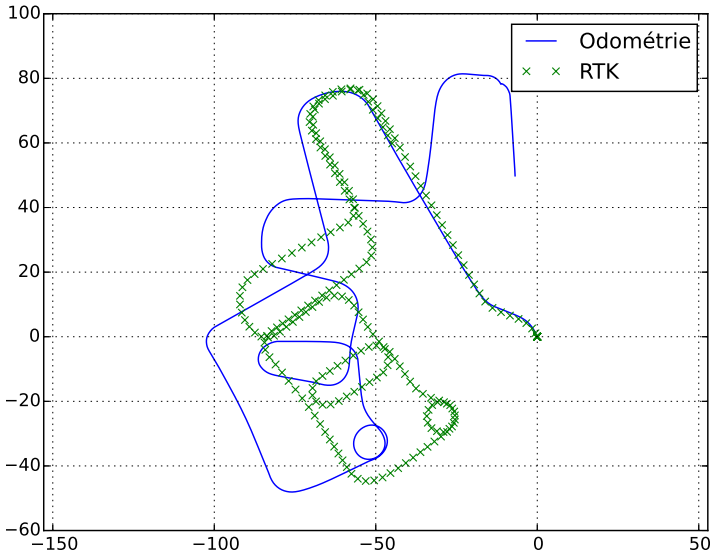


FIGURE A.4 – Trajectoire obtenue par l'odométrie visuelle comparée à la vérité terrain (GPS RTK) sur une séquence de plus 4 minutes de roulage. Les axes des abscisses et des ordonnées sont donnés en mètre.

Enfin, la figure A.5 reprend les deux trajectoires sur une vue satellite issue de Bing Maps.

FIGURE A.5 – Vue satellite représentant la trajectoire obtenue par l'algorithme (en noir) et la trajectoire obtenue par le GPS RTK (en bleu). La vue satellite est extraite de Bing Maps.

Annexe B

Liste des acronymes

I_A intensité arithmétique. 26, 57, 58, 84, 95–97, 99, 101, 103, 113, 130, 131, 136, 137, 147, 155, 157, 160, 168–170, 172, 173, 190, 191

µc microcontrôleur. 16, 17, 64, 66

ACC *Adaptative Cruise Control* : régulation de la vitesse et de la distance de suivi. 6–8

ADAS *Advanced Driver Assistance Systems*. 4–7, 9, 15, 16, 19–22, 26, 39, 42, 43, 46–48, 50, 66, 70, 79, 104, 142, 180, 189, 196, 199

AEB *Advanced Emergency Braking*. 6, 19, 39, 189

AVM *Around View Monitor*. v, 5

BSW *Blind Spot Warning* : alerte de présence d'obstacles dans les angles morts. 8

CISC *Complex Instruction Set Computer*. 16, 17

CV-SDF *Computer Vision Synchronous DataFlow*. 68

DMA *Direct Memory Access*. 73, 74, 105, 109, 111, 119, 122–124, 185

D-MIPS *Drystone MIPS*. 87

DSP *Digital Signal Processor*. 16, 17, 47–49, 53, 91

ECU *Electronic Control Unit*. 7, 21, 22, 48, 64, 66, 104

EDMA *Enhanced Direct Memory Access*. 123

EEMBC *Embedded Microprocessor Benchmark Consortium*. 87

ESM *Efficient Second order Minimisation*. 180, 183

EVE *Embedded Vision Engine*. 48, 49

FLOPS *FLoating point Operations Per Second*. 85, 86, 88, 90, 147, 171, 172

FPGA *Field-Programmable Gate Array*. 18

GNSS *Global Navigation Satellite System*. 8, 180, 186

GPGPU *General-Purpose Computing on Graphics Processing Units*. 17, 18, 42

GPU *Graphics Processing Unit*. 17, 18, 42–46, 48, 49, 52, 53, 59, 73, 86, 88, 91, 92, 95–99, 101, 103–105, 107, 109, 113, 115, 117, 119, 122–126, 129, 135, 136, 144–146, 152–158, 161, 166, 170–173, 185–188, 190, 191, 196

HFP *Hand Free Parking*. 5

HOG *Histogram of Oriented Gradient*. 39, 40, 189, 190

ISP *Image Signal Processor*. 43–45, 48, 49, 80, 122, 190, 197

KPN *Kahn Process Network*. 67, 68, 78, 79

LCA *Lane Centering Assist*. 6, 7

LDW *Lane Departure warning* : alerte de franchissement de lignes. 6

LKA *Lane Keeping Assist* : aide au maintien dans la voie. 6

LLCBench *Low Level Architectural Characterization Benchmark Suite*. 89, 90

MIPS *Millions of Instruction Per Second*. 86

PDF densité de probabilité. 74, 75, 78, 79, 173, 188

PTX *Parallel Thread Execution* langage pseudo assembleur généré lors de la compilation par CUDA. 154

RFI *Request For Information*. 21

RFQ *Request For Quotation*. 21

RISC *Reduced Instruction Set Computer*. 16, 17

SDF *Synchronous DataFlow*. 68, 78, 79

SIMT *Single Instruction Multiple Threads*. 18

SMX *Streaming Multiprocessor*. 43–46, 95, 104, 105, 135, 155, 190

SoC *Systems-on-Chip*. 7, 12, 14, 17–19, 23, 42, 43, 45–53, 64, 65, 73, 74, 77, 79, 80, 84, 85, 95, 97, 104–106, 109, 112, 113, 115, 117, 119, 122–124, 130, 131, 137, 142, 145, 175, 185, 189, 190, 197–199

SPEC *Standard Performance Evaluation Corporation*. 87

SSD *Sum of Square Difference*. 182

SVM *Support Vector Machine*. 39–41, 189, 190

TI Texas Instruments. v, 47–51, 53, 123

UVA *Unified Virtual Addressing*. 123

VLIW *Very Long Instruction Word*. 16

Annexe C

Glossaire

big.LITTLE technologie conçue par ARM associant un processeur faible consommation (LITTLE) et un processeur plus puissant (big) sur la même puce. 17, 45, 47

bird eye view vue de dessus du véhicule. vii, 5, 39, 47–49, 182, 184, 185

CUDA API C++ pour la programmation de GPU Nvidia. 18, 42–46, 51–53, 88, 90, 97, 98, 106, 107, 123, 125, 126, 129, 135, 153, 154, 187, 196

dirty bit flag permettant de statuer sur la cohérence en un bloc du cache et le bloc associé de la mémoire centrale. 11, 104, 105, 107, 109, 119, 136, 137, 140

kernel implémentation générique d'un bloc d'algorithme. 69–75, 77, 79, 80, 90, 111, 126–129, 136, 137, 142, 144–147, 152, 155, 157–162, 164, 166–174, 184–187, 190–192, 196, 197, 199

mapping répartition de l'exécution des kernels sur les différents processeurs d'une architecture hétérogène. 23, 64, 65, 72–75, 77–79, 127, 152, 162, 170, 174, 175, 186, 187, 190–192, 196–198

NEON unité de calcul SIMD des processeurs ARM. 17, 43, 45, 93, 97–99, 101, 104, 134, 168, 196, 197

unité d'exécution unité composant un processeur avec la capacité d'exécuter un *thread* (une série d'instructions). 10–14, 16–18, 72–75, 77, 84, 95, 122, 127–129, 152, 155, 158–160, 174, 190, 191

unité élémentaire unité atomique faisant partie une unité d'exécution. 10, 12, 13, 16, 43, 57, 90, 95, 97, 99, 136, 146, 152, 158, 159

warp groupement insécable de 32 *threads* propre à l'API CUDA exécutant la même instruction. 18, 155

workload charge de calcul utilisée pour caractériser la performance d'un processeur. 86–89, 154, 155

Annexe D

Liste des symboles

Δ ensemble des arcs du graphe de traitement représentant les délais de transfert de données. 69

$dt_{spec}(i,j)$ valeur maximale de la latence entre k_i et k_j définie par le cahier des charges de l'application. 71, 72, 74, 75, 77

$\mathscr{F}$ fonction de coût globale pour l'optimisation du mapping. 76, 77

f_l fonction de coût liée à la contrainte de latence. 77

f_r fonction de coût liée à la contrainte de rythme. 77

G graphe de traitement. 69–71, 79

$\mathbb{G}$ graphe global, comprenant toute les instances de G. 71–73, 77, 79

K ensemble des nœuds du graphe de traitement représentant les kernels. 69, 70, 73

$K|_x$ ensemble des kernels associés à l'unité de calcul pu_x. 73, 74

M mapping des kernels sur les unités d'exécution d'une architecture hétérogène. 72–74, 77

$\mathbb{M}$ ensemble des mappings possibles. 73, 77

η_{pu_x} taux d'occupation de l'unité d'exécution pu_x. 74, 77

$\mathscr{P}_{i,j}$ chemin de G allant du kernel k_i au kernel k_j. 70, 71

$\mathbb{P}_{i,j}$ ensemble des chemins possibles allant du kernel k_i au kernel k_j. 70, 71

$\mathscr{P}_{i,j,u}$ chemin de $\mathbb{G}$ allant du kernel $k_i^{(n-u)}$ au kernel k_j^n. 72, 74, 75

$t_{\mathscr{P}}(i,j)$ temps d'exécution du chemin $\mathscr{P}_{i,j}$. 71

$t_p(i,j)$ latence entre k_i et k_j. 71

$t_p(i,j,u)$ latence entre $k_i^{(n-u)}$ et k_j^n. 72, 74, 75, 77

T_c période d'acquisition. 71–75, 77, 128

t_{pu_x} temps d'exécution de l'unité de calcul pu_x. 73, 74

PU ensemble des unités d'exécution d'une architecture hétérogène. 72, 73, 75, 77

Annexe E

Publications et communications

Conférences nationales

Romain SAUSSARD, Boubker BOUZID, Marius VASILIU, Roger REYNAUD, « Prédiction des performances d'algorithmes ADAS sur architectures embarquées », dans *2015 CNRS Innovatives*, Paris, 2015.

Conférences internationales

Romain SAUSSARD, Boubker BOUZID, Marius VASILIU, Roger REYNAUD, « Predicting ADAS algorithms performances on K1 architecture », dans *2015 Nvidia GPU Technology Conference (GTC)*, San Jose, 2015.

Romain SAUSSARD, Boubker BOUZID, Marius VASILIU, Roger REYNAUD, « Towards an automatic prediction of image processing algorithms performances on embedded heterogeneous architectures », dans *2015 44th International Conference on Parallel Processing Workshops (ICPPW)*, Pékin, 2015.

Romain SAUSSARD, Boubker BOUZID, Marius VASILIU, Roger REYNAUD, « Optimal performance prediction of ADAS algorithms on embedded parallel architectures », dans *2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC)*, New York, 2015.

Romain SAUSSARD, Boubker BOUZID, Marius VASILIU, Roger REYNAUD, « The embeddability of lane detection algorithms on heterogeneous architectures », dans *2015 IEEE International Conference on Image Processing (ICIP)*, Québec, 2015.

Romain SAUSSARD, Boubker BOUZID, Marius VASILIU, Roger REYNAUD, « A Robust Methodology for Performance Analysis on Hybrid Embedded Multicore Architectures », dans *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, Lyon, 2016.

Brevets

Romain SAUSSARD, Boubker BOUZID, Marius VASILIU, Roger REYNAUD, « Procédé d'estimation du temps d'exécution d'une partie de code par un processeur ».

Romain SAUSSARD, Boubker BOUZID, Marius VASILIU, Roger REYNAUD, « Méthode de détermination d'une performance temporelle d'une unité de traitement électronique exécutant un algorithme ».

Romain SAUSSARD, Boubker BOUZID, Marius VASILIU, Roger REYNAUD, « Procédé de caractérisation d'une unité de traitement électronique ».

Jounal

Romain SAUSSARD, Boubker BOUZID, Marius VASILIU, Roger REYNAUD, « A Novel Global Methodology to Analyze the Embeddability of Real-Time Image Processing Algorithms », dans *Journal of Real-Time Image Processing (JRTIP)*, 2017.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Méthodologies et outils de portage d'algorithmes de traitement d'images sur cibles hardware mixte

Mots clefs : SoCs hétérogènes – Temps-réel – Traitement d'images – GPU – ADAS

Résumé : Les constructeurs automobiles proposent de plus en plus des systèmes d'aide à la conduite, en anglais Advanced Driver Assistance Systems (ADAS), utilisant des caméras et des algorithmes de traitement d'images. Pour embarquer des applications ADAS, les fondeurs proposent des architectures embarquées hétérogènes. Ces Systems-on-Chip (SoCs) intègrent sur la même puce plusieurs processeurs de différentes natures. Cependant, avec leur complexité croissante, il devient de plus en plus difficile pour un industriel automobile de choisir un SoC qui puisse exécuter une application ADAS donnée avec le respect des contraintes temps-réel. De plus le caractère hétérogène amène une nouvelle problématique : la répartition des charges de calcul entre les différents processeurs du même SoC.
Pour répondre à cette problématique, nous avons défini au cours de cette thèse une méthodologie globale de l'analyse de l'embarquabilité d'algorithmes de traitement d'images pour une exécution temps-réel. Cette méthodologie permet d'estimer l'embarquabilité d'un algorithme de traitement d'images sur plusieurs SoCs hétérogènes en explorant automatiquement les différentes répartitions de charge de calcul possibles. Elle est basée sur trois contributions majeures : la modélisation d'un algorithme et ses contraintes temps-réel, la caractérisation d'un SoC hétérogène et une méthode de prédiction de performances multi-architecture.

Title : Methodologies and tools for embedding image processing algorithms on heterogeneous architectures

Keywords : Heterogeneous SoCs – Real-time – Image processing – GPU – ADAS

Abstract : Car manufacturers increasingly provide Advanced Driver Assistance Systems (ADAS) based on cameras and image processing algorithms. To embed ADAS applications, semiconductor companies propose heterogeneous architectures. These Systems-on-Chip (SoCs) are composed of several processors with different capabilities on the same chip. However, with the increasing complexity of such systems, it becomes more and more difficult for an automotive actor to chose a SoC which can execute a given ADAS application while meeting real-time constraints. In addition, embedding algorithms on this type of hardware is not trivial: one needs to determine how to spread the computational load between the different processors, in others words the mapping of the computational load.
In response to this issue, we defined during this thesis a global methodology to study the embeddability of image processing algorithms for real-time execution. This methodology predicts the embeddability of a given image processing algorithm on several heterogeneous SoCs by automatically exploring the possible mapping. It is based on three major contributions: the modeling of an algorithm and its real-time constraints, the characterization of a heterogeneous SoC, and a performance prediction approach which can address different types of architectures.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France