Étude des temps d'accès aux diverses mémoires et de l'exploitation du parallélisme gros grain sur GPU Nvidia

Sommaire

Dans le chapitre 3, nous avons défini une méthodologie de portage d'algorithmes sur GPU. Celle-ci considère les domaines d'itérations des nids de boucles pour les adapter aux domaines d'instances des GPUs. Cette méthodologie, validée au cours du chapitre 4, est décomposée en plusieurs étapes :

— les analyses statiques et dynamiques du code source (sections 3.1 et 3.2),
— la vérification des trois critères de placement sur GPU (section 3.3),
— l'application de différentes transformations de code pour améliorer quantitativement et qualitativement le placement sur GPU(section 3.4),
— la génération de code pour GPU (sections 3.5 et 3.6),
— la validation des *kernels* au moyen des analyses dynamiques (section 3.7).

Au final, ces différentes procédures suffisent pour générer un placement valide sur GPU à partir d'un algorithme séquentiel. Certains points d'optimisation ont de plus été introduits tels que :

— pour le critère 1 de placement (section 3.3.1) :

  - la minimisation des dépendances à l'intérieur d'un *block* et
  - la maximisation du nombre de dimensions d'instances de *threads*
- pour le critère 2 de placement (section 3.3.2) :
  - la saturation des unités SM en utilisant une quantité minimale d'instances de *thread* par *block* et
  - la saturation des *warps* en utilisant un multiple de 32 *threads* pour la taille des *blocks*
- la linéarisation des accès mémoire (section 3.5.4) lors de la préparation avant génération de code,
- l'élimination des allocations et libérations d'instances mémoire redondantes sur l'accélérateur (section 3.6.2)
- et pour la génération des communications mémoire hôte/accélérateur (section 3.6.3) :
  - l'élimination des communications redondantes et
  - la réduction des transferts aux espaces de données modifiés.

Ces diverses optimisations contribuent à l'identification d'une solution de placement qualitative, réduisant le temps d'exécution de l'algorithme résultant. Mais l'optimisation de code est un sujet très vaste. Le flot de calculs, le flot d'instructions et le flot de données, abordés dans la section 1.3, sont respectivement exploités, au sein des GPUs, par des ressources dédiées telles que :

- les *CUDA cores*,
- les *instruction dispatch units* et
- les *load/store units*.

Leur saturation a alors une influence directe sur le temps d'exécution d'un *kernel*. On retrouve notamment, parmis les nombreuses sources possibles de sous-exploitation de ces ressources :

- le nombre élevé de cycles nécessaire pour l'exécution de certaines opérations,
- le phénomène de "bulles" dans les *pipelines* d'instructions ou encore
- le temps de latence et la bande passante des différentes mémoires.

Nous nous intéressons, dans notre méthodologie, à la minimisation du temps d'exécution des algorithmes placés sur GPU. Nous abordons ainsi, dans ce chapitre, deux ensembles de mesures expérimentales, préalables à de futures optimisations fines de code pour CUDA, dans le cadre de notre méthodologie, qui améliorent le placement sur les GPUs Nvidia.

Cette thèse s'intéressant en particulier aux applications de traitement d'image, nous retrouvons dans ce domaine trois facteurs dont la prise en compte présente une source potentielle d'accélération du temps d'exécution :

1. En raison du volume important de données à traiter, la bande passante mémoire constitue souvent le facteur limitant sur GPU.

2. Il arrive aussi, dans certains traitements complexes, que ce volume de données soit fragmenté en plusieurs sous-ensembles, traités dans des *kernels* distincts. La quantité de données à traiter peut alors devenir insuffisante pour exploiter les capacités du GPU.

3. Enfin, l'utilisation de sources de données multiples [1], engendre des *pipelines* d'applications de traitement d'image portant sur des ensembles distincts de données. Ces *pipelines* sont ainsi compatibles avec une exécution concurrente, lorsque les portions du programme ne présentant pas de dépendance.

1. tel que l'ensemble des capteurs du véhicule *model S* de Tesla cité dans le chapitre 1

Nous évaluons ainsi, dans la section 5.1, plusieurs performances des différents espaces mémoire sur GPU pour le premier facteur. Pour les deux derniers, nous étudions, dans la section 5.2, le comportement du GPU lors d'un placement exploitant un parallélisme à gros grain (*coarse grained parallelism*).

Les expérimentations décrites dans ce chapitre sont préliminaires à l'intégration de ces deux sujets dans notre méthodologie.

5.1 Étude des espaces mémoire sur GPU

Notre méthodologie de placement sur GPU (chapitre 3) spécifie, lors du processus de génération de code (section 3.6), le placement des données sur la *global memory*. Dans cette section, nous étendons les capacités de notre méthodologie à l'usage des autres espaces mémoires mis à disposition par CUDA.

Concrètement, trois catégories de mémoire physique sont présentes sur toutes les architectures GPU :

***Device memory* :** Physiquement située à l'extérieur du SOC, la *device memory* est partagée par l'ensemble des unités SM. Elle est accessible par l'ensemble des *blocks* de *threads* et est utilisée pour l'échange de données avec le processeur hôte.

***Shared memory* :** Présente dans chaque unité SM, la *shared memory* est exclusivement accessible aux CUDA cores de l'unité SM concernée. Elle est utilisable par l'ensemble des *threads* de chaque *block* et permet l'échange de données entre ces *threads*.

Unités registres : Comme pour la *shared memory*, les unités registres sont présentes dans chaque unité SM et sont dédiées aux CUDA cores. Chaque allocation d'une unité registre est dédiée à un *thread* et l'échange de données n'est de ce fait pas possible.

Ces trois mémoires physique sont exploitées, au travers du langage de programmation CUDA, au moyen de sept espaces mémoires. La différence entre ces espaces provient essentiellement des différentes méthodes d'accès aux données et plus particuliérement des mémoires caches dédiées. En reprenant la nomenclature issue de la documentation officielle de CUDA [123], ces espaces correspondent à :

— la *Global memory*,
— la *Local memory*,
— la *Constant memory*,
— la *Texture memory*,
— la *Surface memory*,
— la *Shared memory* et
— les registres.

Nous faisons ainsi la distinction entre les unités de mémoires physiquement présentes sur le GPU des espaces mémoires logiques fournis par CUDA. En complément, CUDA appelle "*Host memory*", l'espace mémoire du processeur hôte mis à disposition pour le contexte CUDA.

L'ensemble des solutions présentées dans l'état de l'art (chapitre 2) propose le placement des données en *global memory*. La *local memory* et les unités registres sont utilisées de manière implicite par le compilateur NVCC.

Nous considérons alors les solutions permettant d'identifier le placement de données sur les autres espaces mémoire. Parmi ces solutions, le placement de données en *shared memory* est géré de manière automatique par :

— PPCG [160, 159] (section 2.2.3),
— *C-to-CUDA* [23] (section 2.2.1),
— R-Stream [103] (section 2.2.4) et
— *Cuda-Lite* [153] (section 2.4.1).

Pour la *constant memory*, seul *C-to-CUDA* semble l'exploiter. Le placement est réalisé par ce dernier, lorsque les accès sur un tableau sont en lecture seule et sont communs pour l'ensemble des *threads* de chaque *block*.

Enfin, nous ne connaissons aucun compilateur source-à-source exploitant la *texture memory* ou la *surface memory*.

De manière générale, les informations sur les modèles de placement mémoire utilisés par les compilateurs sont assez rares et peu détaillées. Nous ne connaissons, de plus, aucune publication portant sur la détermination automatisé de l'espace mémoire le plus adapté à chaque espace de données d'un algorithme donné.

Ce constat peut s'expliquer par les modifications architecturales, spécifiques aux mémoires, apportées à chaque nouvelle génération de GPU chez Nvidia [132, 131, 130, 129, 120]. Les optimisations dans ce domaine sont donc spécifiques à chaque génération. Ces modifications permanentes soulignent le besoin actuel d'améliorer les bandes passantes d'accès aux données, abordé en introduction du chapitre 1.

Nous nous intéressons ainsi à la problématique portant sur l'identification de l'espace mémoire minimisant le temps d'accès aux données dans un algorithme. Pour cela, nous synthétisons, dans la section 5.1.1, les caractéristiques de chacun de ces espaces mémoire. Nous établissons ensuite l'objectif de cette expérimentation dans la section 5.1.2 et détaillons notre protocole de test dans la section 5.1.3. Enfin, dans la section 5.1.4, nous analysons les performances des différents espaces mémoire afin d'établir des critères de sélection.

Cette étude est réalisée avec le GPU Quadro K2000 d'Endicott et le SOC T210 de la Jetson TX1. Ces deux plateformes sont décrites dans la section 4.1.

5.1.1 Descriptions des espaces mémoire CUDA

Nous présentons, dans cette section, les spécificités des différents espaces mémoires disponibles dans CUDA. L'ensemble des informations utilisées ont été extraites des documentations officielles CUDA [122, 123].

Global memory

La *global memory* est l'espace d'échange principal entre le processeur hôte et le GPU. Il prend place sur la majeure partie de la *device memory*.

Son utilisation a déjà été décrite dans la section 3.6.2. Les espaces de données sont alloués au moyen de l'instuction *cudaMalloc*, qui garantit un alignement des données sur 256 $Bytes$. Leur libération se fait au moyen de la fonction *cudaFree*. Les données non libérées ont une persistance correspondant à la durée de vie du contexte CUDA utilisé.

Les communications de données entre l'hôte et l'accélérateur ont été abordées dans la section 3.6.3.

Depuis la génération *Fermi* (*compute capability* $\geq 2.x$), les communications de données avec la *global memory* passent par un processus de cache. Pour la génération *Fermi*, il s'agit d'un cache à double niveaux (L1 + L2). Cependant, depuis la génération *Kepler* (*compute capability* $\geq 3.x$), les données ne transitent plus que par le cache L2. Les lignes de cache L1 font 128 $Bytes$ tandis que celles du cache L2 font 32 $Bytes$.

L'utilisation de mémoire cache apporte une accélération du temps d'accès aux données de la *global memory* en cas de *cache hit*. L'alignement des données et l'augmentation du *stride* lors des accès ont donc un impact direct sur le temps d'accès global aux données.

Local memory

La *local memory* n'est pas utilisable de manière explicite au moyen de CUDA. Dans le processus de compilation de NVCC, son utilisation ne peut se faire qu'au moyen des instructions *ld.local* et *st.local* de l'ISA PTX de Nvidia [127]. Chaque *thread* dispose d'un espace privé de $512\,KB$, résidant dans la *device memory*.

De manière générale, la *local memory* est utilisée par le compilateur NVCC, pour les tableaux déclarés de manière statique dans le *kernel*. Elle permet de limiter la surcharge des unités registres, en appliquant le principe du *register spilling*. Dans le cadre de notre méthodologie, la privatisation de tableaux dans un *kernel*, invoquée dans les transformations de code de la section 3.4, a un impact sur l'utilisation de la *local memory*.

Les communications de données transitent par les mémoires cache L1 et L2, décrites pour la *global memory*. L'utilisation de ces deux niveaux de cache permet de limiter la pénalité liée à la bande passante plus faible de la *device memory*, en comparaison à celle des unités registre.

L'emploi de la *local memory*, associée à la privatisation de tableaux, sera cependant évitée autant que possible. Nous lui préférons au contraire la privatisation de scalaire, qui maintient les données dans les unités registre. Ce sujet a été abordé dans la section 4.3.4.

Texture/Surface memory

La *texture memory* est un espace mémoire accessible en lecture seule. Concrètement, elle exploite, comme la *global memory*, une partie de la *device memory*, externe au SOC. Elle se différencie cependant par un canal d'accès distinct, composé d'un cache dédié (le *texture cache*) et d'unités de calcul spécialisées (les *texture units*).

Le **texture cache**, en association avec l'exploitation des *CUDA arrays*, conçoit la localité spatiale des données selon deux dimensions. Il est ainsi optimisé pour les accès bi-dimensionnels en accélérant, dans ce cas précis, le temps d'accès aux données, comparativement aux classiques méthodes d'accès mono-dimensionnelles. De plus, la *texture memory* étant accessible en lecture seule, l'absence de vérification pour la cohérence des données, entre les différents modules de *texture cache*, se traduit par un gain supplémentaire sur le temps d'accès aux données placées en cache.

Enfin, les **texture units** sont des unités de calculs spécialisées, dont le rôle est d'effectuer :

— l'interpolation spatiale de données selon une loi linéaire et
— la réplication dynamique de données lors d'accès en dehors des bornes de définition des dimensions de l'espace mémoire alloué.

Ces unités permettent ainsi de réduire la quantité de calculs effectués par les *CUDA cores* en formattant de manière dynamique les données acheminées. En pratique, cet espace mémoire remplace judicieusement les fonctions OpenCV :

— *copyMakeBorder* avec la réplication dynamique des données et
— *resize* avec l'interpolation spatiale des données.

De nombreux exemples d'utilisation de ces fonctions sont visibles dans l'algorithme *simpleFlow* de l'annexe A.

La *texture memory* étant prévue pour un accès en lecture seule, l'accès en écriture pour cet espace de données a été introduit avec la *surface memory* (*compute capability*

$> 2.x$). Cette dernière permet ainsi de s'affranchir du transit des données par la *global memory* (accessible en écriture), lorsque l'on souhaite procéder à une modification par le GPU des données déclarées en *texture memory*. L'usage de la *global memory* a alors pour désavantage de générer des transferts de données internes au GPU avec la *texture memory*. Bien que la *surface memory* soit aussi accessible en lecture, son principal intérêt reste l'accès en écriture des données exploitées par le *texture memory*.

Shared memory

À la différence de la *device memory*, la *shared memory* est un espace mémoire physiquement intégré dans le SOC du GPU. Sa proximité avec les SMs offre ainsi une bande passante supérieure et un temps de latence plus faible. Son utilisation est exclusive à chaque SM et le processeur hôte ne peut, de ce fait, y accéder. La persistance des données instanciées est limitée à la durée d'exécution de chaque *block* de *threads* dans le SM concerné. Au maximum 1024 *threads*, peuvent ainsi s'échanger des données au moyen de la *shared memory*. La cohérence des données entre *threads* est alors préservée en utilisant l'instruction de synchronisation *___syncthreads()*.

La *shared memory* est répartie en 32 modules appelés *memory banks*. Ces derniers présentent l'avantage, pour les architectures parallèles, de pouvoir travailler de manière simultanée, permettant ainsi d'améliorer la bande passante globale d'accès aux données. En revanche, l'inconvénient de ce type de disposition mémoire est la génération de conflits lorsque des accès concurrents se font sur une même *memory bank*. La résolution de cette problématique passe alors par une sérialisation des accès, dégradant ainsi le temps d'accès aux données. Cependant, dans un *warp*, les accès concurrents à une même *memory bank* ne sont pas en conflit lorsque la requête porte sur la même donnée. Cette dernière est alors diffusée à l'ensemble des *threads* concernés. Depuis la génération *Fermi* (*compute capability* $\geq 2.x$), 32 *memory banks* sont utilisées pour une répartition cyclique en mots de $32\,bits$.

Pour chaque *block*, nous identifions les conflits sur les *memory banks* au moyen de 5.1.

$$\forall x \in [0, 1023],\ \exists ? y \in [0, 1023]\ tq \begin{cases} y \neq x \\ a(y) \neq a(x) \\ a(y) \mod 32 = a(x) \mod 32 \end{cases} \tag{5.1}$$

L'ensemble y des solutions représente alors les identifiants de *threads* dont l'accès à la *shared memory* présente un conflit de *memory bank*. La variable x représente l'identifiant d'un *thread* dans un *block* et a est une fonction d'accès en *shared memory*.

Nous considérons l'emploi de la *shared memory* lorsque les différents *threads* d'un *block* présentent une réutilisation de données. Cet espace mémoire est alors considéré comme une *scratchpad memory*, gérée manuellement dans le *kernel*. Il permet notamment de résoudre la problématique de coalescence des données lorsque les accès ne présentent pas un *stride* unitaire. En conséquence, la localité spatiale s'en retrouve améliorée, ce qui se traduit par un temps d'accès plus faible.

Constant memory

Sur un principe similaire à celui de la *texture memory*, l'usage de la *constant memory* correspond à l'exploitation d'une partie dédiée de la *device memory* du GPU. Cette mémoire dispose cependant d'une mémoire cache distincte (le *constant cache*) pour chaque SM. Comme pour le *texture cache*, le *constant cache* profite de l'accès restreint en lecture

seule pour s'affranchir du processus de vérification de cohérence des données. Cependant, pour chaque *warp*, les accès aux données au moyen de la *constant memory* ne sont pas réalisés sous forme de lignes de cache mais sont au contraire linéarisés. Le temps d'accès aux données est donc proportionel au nombre d'adresses distinctes utilisées dans chaque *warp*. Enfin, la persistance des données instanciées en *constant memory* correspond à la durée de vie du contexte CUDA correspondant. Les données allouées ne peuvent être libérées par le programme du fait qu'il n'y a aucune instruction au sein de l'ISA permettant de le faire.

Afin d'être placés en *constant memory*, les espaces de données doivent impérativement respecter les trois critères suivants :

1. Les accès devront exclusivement être en lecture à l'intérieur des kernels concernés.
2. La somme des empreintes mémoires correspondantes, pour la globalité de l'application, devra être inférieure à la capacité mémoire de la *constant memory*. Chez Nvidia, cet espace mémoire représente 65Ko de données librement disponibles pour le programmeur et 65Ko de données supplémentaires utilisables par le compilateur NVCC.
3. L'allocation de l'espace mémoire devra être statique et défini au moment de la compilation. Dans le cas contraire, le calcul d'une enveloppe convexe maximale au moyen d'une analyse de région permettra d'utiliser une approche dynamique.

Si l'ensemble de ces critères est rempli, le placement est alors considéré comme légal. Cependant, afin de maximiser la bande passante de cet espace mémoire, on ajoutera comme critère d'optimisation la maximisation des accès communs entre *warps*.

5.1.2 Description du sujet d'expérience

Dans la section 5.1.1, nous avons détaillé les spécificités des cinq espaces mémoire pour GPU mis à disposition par CUDA : la *Global memory*, la *Constant memory*, la *Texture memory*, la *Shared memory* et les registres. Nous cherchons maintenant à déterminer les paramètres favorisant la selection d'un de ces espaces mémoire en fonction des caractéristiques de l'algorithme étudié. Ce sujet est vaste et nous avons spécialisé notre étude, dans le cadre de cette thèse, pour un unique accès en lecture par *thread*. Nous nous intéressons, dans ce cadre, à la corrélation entre les différentes méthodes d'accès aux données avec le temps d'exécution d'un *kernel* type. Ces travaux initiaux seront complétés dans de futures publications.

5.1.3 Protocole expérimental

Nous avons défini un *kernel* spécifique à l'évaluation de chaque espace mémoire. Chacun de ces *kernels* est exécuté dix fois avec les même paramètres, afin d'évaluer la variabilité des temps d'exécution.

Les temps d'exécutions sont collectés au moyen de *CUDA Events*. Cette instruction renvoie au processeur hôte, sa propre date d'exécution sur l'accélérateur. En encadrant, dans le code hôte, l'appel d'un *kernel* par deux de ces instructions, nous obtenons, par la différence des deux dates résultantes, le temps d'exécution du *kernel*. L'instruction *CUDA Event* correspondant à la fin d'exécution du *kernel* est associée à l'instruction *cudaEventSynchronize* afin de maintenir une synchronisation avec le processeur hôte.

Les transferts de données entre l'hôte et l'accélérateur transitent par l'unité de *cache L2*, globale au GPU. En conséquence, la persistance des données dans cette mémoire cache vient "polluer", dans le cadre de notre expérimentation, la mesure des temps d'accès

aux différents espaces mémoires. Nous avons de ce fait ajouté, avant chaque exécution d'un *kernel*, une étape "d'empoisonement" du cache L2. N'ayant pas d'information sur la méthode de mise en cache employée, nous transférons $100\,MB$ de données vers la *global memory* afin de s'assurer de la saturation du *cache L2*. Pour rappel, la taille de cette mémoire cache est de $256\,KB$ sur les GPUs d'Endicott et de la Jetson TX1. Les tests que nous avons menés ont montré que l'empoisonnement préalable du cache L2 implique une augmentation du temps d'exécution de chaque *kernel*, justifiant ainsi son utilisation. Enfin, le *cache L1* est nativement invalidé entre chaque *kernel* exécuté, afin de garantir la cohérence des données. De ce fait, aucune action n'a été prise à son sujet.

L'ensemble des évaluations a été compilé avec NVCC 8.0.33 en utilisant l'option $-O0$. Cette précaution nous assure qu'aucune optimisation sur les accès aux données n'est effectuée par le compilateur.

Chaque *kernel* est exécuté pour 2025 *blocks*, saturés par 1024 instances de *threads*. Ces chiffres ont été choisis car ils correspondent à la quantité de données d'une image de résolution standard *full HD*. En adaptant le nombre global d'instances de *threads* à la quantité de données d'une image, nous nous plaçons ainsi dans le cas du parallélisme de données, typique au traitement d'images.

L'ensemble des *kernels* a été spécifié pour que chaque *thread* effectue un accès en lecture et un accès en écriture pour un entier codé sur $8\,bits$. L'accès en écriture sur la *global memory*, correspondant à la sortie du *kernel*, est identique pour l'ensemble des *kernels*. Notre évaluation porte donc sur l'évaluation et la comparaison des performances en lecture des différents canaux d'accès mémoire. Le flot de communication des données est le facteur limitatif pour l'exécution de ces *kernels*. Le flot d'instructions et le flot d'opérations arithmétiques sont réduits au strict minimum. Le temps d'excution des *kernels* dépend donc des temps d'accès aux données.

Chaque *kernel* présente deux variantes se distinguant par une fonction distincte d'accès en lecture :

— $\mathcal{R}_1$, définie en (5.2), distribue les accès distincts de manière cyclique [2] et
— $\mathcal{R}_2$, définie en (5.3), effectue, au contraire, une distribution par blocs regroupant les accès identiques.

$$\mathcal{R}_1 = (threadId \mod s) \times c \tag{5.2}$$

$$\mathcal{R}_2 = \lfloor \frac{threadId}{s} \rfloor \times c \tag{5.3}$$

La distinction entre $\mathcal{R}_1$ et $\mathcal{R}_2$ permet d'évaluer l'impact de la contiguité des accès communs. Pour ces deux fonctions, $threadId$ représente l'identifiant du *thread* courant, tel que $threadId \in [0; 1023]$ pour chaque *block*. Le paramètre s définit le nombre d'accès distincts, tel que $s \in [1; 1024]$. Enfin, le coefficient c modifie le *stride*, correspondant à l'écart spatial entre les accès aux données. Ce coefficient permet d'augmenter la quantité de *cache miss* ce qui engendre une augmentation des communications avec la mémoire physique, les transferts se faisant par lignes de données contiguës. À ce sujet, le bus mémoire de la TX1 est de 8 *Bytes* et celui de la K2000 d'Endicott de 16 *Bytes*. Nous avons contraint cette expérimentation aux cas d'étude $c \in \{1, 16\}$ tel que :

c = 1 correspond à un cas de parfaite coalescence des données et

c = 16 correspond à la plus grande largeur de bus mémoire pour les deux plateformes considérées.

2. Méthode de type Round-robin

Enfin, nous ne considérons pas dans les résultats de cette évaluation les temps de transfert pour l'échange de données entre le processeur hôte et le GPU.

Nous détaillons à présent les méthodes d'évaluation des différents espaces mémoire.

Évaluation des registres

Pour l'ensemble des figures, l'évaluation du temps d'accès des registres correspond à :

1. l'initialisation d'une variable scalaire dans l'espace des registres,
2. l'écriture de la valeur de cette variable, dans la *global memory* en sortie.

Chaque unité registre est dédiée à un unique *thread*. Le temps d'exécution correspond alors nécessairement à 1024 accès distincts dans le référentiel d'un *block* et à 32 accès distincts dans le référentiel d'un *warp*. De ce fait, $\mathcal{R}_1$ et $\mathcal{R}_2$ ne peuvent s'appliquer pour ce cas.

Évaluation de la *shared memory*

Lorsque les données sont utilisées au-delà du context d'un *block* (au sein d'une unité SM), la *shared memory* ne peut être employée de manière exclusive du fait que :

— le processeur hôte ne peut y accéder et
— la cohérence des données est limitée à un unique *block*.

Ainsi, notre évaluation de la *shared memory* est réalisée en faisant préalablement transiter les données par la *global memory*. Les performances de la *global memory* ont donc une influence certaine sur les résultats obtenus pour la *shared memory*. Ce binôme nous permet de plus d'observer le comportement de la *shared memory* comme l'équivalent, pour la *global memory*, d'une mémoire cache L1 gérée manuellement par le développeur.

Le *kernel* servant à l'évaluation de la *shared memory* est spécifié de la façon suivante :

1. Chaque instance du *kernel* débute en effectuant un unique accès en lecture à la *global memory* selon les fonctions $\mathcal{R}_1$ ou $\mathcal{R}_2$.
2. Un branchement conditionnel vérifie auparavant que l'accès aux données se fait de manière unique pour chaque *block* limitant ainsi la redondance de communications.
3. Pour les *threads* concernés, chaque donnée récupérée est alors transférée dans la *shared memory*.
4. Afin de nous assurer de la cohérence de ces données pour l'intégralité de chaque *block*, nous effectuons une synchronisation des *threads* au moyen de l'instruction *___syncthreads()*.
5. Chaque instance du *kernel* procède ensuite à un accès en lecture à la *shared memory* afin d'écrire, en sortie de *kernel*, la valeur récupérée dans la *global memory*.

Évaluation des autres espaces mémoire

L'évaluation des autres espaces mémoire se fait selon l'implémentation, dans chaque *kernel*, du *pattern* algorithmique commun suivant :

1. lecture des données dans l'espace mémoire évalué selon les fonctions $\mathcal{R}_1$ ou $\mathcal{R}_2$ et
2. écriture de la valeur lue dans la *global memory*.

5.1.4 Analyse et interprétation des résultats

Nous abordons dans un premier temps le formalisme employé pour la représentation des résultats. Dans un second temps, nous procédons pour chaque plateforme à l'analyse de ces résultats, représentés dans les figures 5.1 à 5.4 pour *Endicott* puis dans les figures 5.5 à 5.8 pour la *Jetson TX1*.

Représentation des résultats

Afin de pouvoir comparer les résultats des fonctions d'accès $\mathcal{R}_1$ et $\mathcal{R}_2$, nous avons défini comme métrique commune le nombre d'accès distincts dans le référentiel d'un *block* de 1024 *threads*. Cette métrique correspond à :

— la fonction $N1_{block}$ en (5.4) pour $\mathcal{R}_1$,
— la fonction $N2_{block}$ en (5.5) pour $\mathcal{R}_2$.

$$\forall s \in [1;1024],\ N1_{block}(s) = s \tag{5.4}$$

$$\forall s \in [1;1024],\ N2_{block}(s) = \lceil \frac{1024}{s} \rceil \tag{5.5}$$

La même démarche a été effectuée en utilisant pour référentiel un *warp* de 32 *threads*. Nous avons alors adapté la représentation des résultats dans ce référentiel en modifiant :

— la fonction (5.4) en (5.6) afin d'obtenir $N1_{warp}$,
— la fonction (5.5) en (5.7) pour $N2_{warp}$.

$$\forall s \in [1;32],\ N1_{warp}(s) = s \tag{5.6}$$

$$\forall s \in [1;32],\ N2_{warp}(s) = \frac{1024}{s \times 32} = \frac{32}{s} \tag{5.7}$$

Dans le cas de $N1_{warp}$, les données sont redondantes entre *warps*. Pour $N2_{warp}$ les données sont au contraires distinctes. Ces deux fonctions nous permettent d'évaluer dans une unité SM l'effet lié à la réutilisation de données entre *warps*.

La **moyenne des temps d'exécutions**, pour les dix exécutions de chaque *kernel*, est représentée par un trait hachuré pour chaque espace mémoire. Nous avons ajouté en trait continu l'**écart-type par rapport à la moyenne** pour l'ensemble de ces courbes. Plus l'écart entre ces traits est important, plus la variation des temps mesurés est forte. Un unique trait continu indique au contraire un écart-type proche de zéro. Ce dernier cas correspond à une excellente stabilité des temps d'exécution ayant pour conséquence une meilleure reproductibilité des temps d'exécution.

Nous avons volontairement figé les échelles des figure 5.1 à 5.8, afin d'en faciliter leurs comparaisons.

Enfin, le temps d'exécution du *kernel* évaluant les registres a été "extrapolé" à l'ensemble des valeurs d'accès distincts. Les unités registres offrant le temps d'accès le plus faible et en considérant ce temps comme négligeable, cette courbe permet d'apprécier :

— le temps d'exécution minimal pouvant être atteint,
— le temps d'écriture en sortie, commun à l'ensemble des *kernels* et
— le temps d'accès en lecture des différents espaces mémoire [3].

Résultats pour Endicott - Quadro K2000

Dans le **référentiel d'un *block***, les temps d'exécution pour les différents espaces mémoire sont représentés :

— dans la figure 5.1 pour la **distribution cyclique** des accès et
— dans la figure 5.2 pour la **distribution par bloc**.

Dans le **référentiel d'un *warp***, les résultats sont représentés :

— dans la figure 5.3 pour la **distribution cyclique** et
— dans la figure 5.4 pour la **distribution par bloc**.

3. en évaluant la différence entre la courbe des registres et les courbes des différents espaces mémoire.

Pour la ***constant memory***, les temps d'accès augmentent globalement avec le nombre d'accès distincts pour l'ensemble des représentations.

Pour la distribution cyclique (figure 5.1a), le temps d'accès maximal est atteint pour 32 accès distincts et reste constant au-delà, révélant ainsi un phénomène de saturation. Pour la distribution par bloc (figure 5.2a), le temps d'accès reste au contraire minimal et constant jusqu'à 32 accès distincts et augmente au-delà. Le maximum est atteint pour 1024 accès distincts. Cette différence de résultat révèle l'influence du modèle de distribution employé sur les temps d'accès aux données. Nous constatons que le point de rupture est commun dans les deux cas, soit 32 accès distincts et que la saturation a lieu :

— au-delà du point de rupture pour la distribution cyclique et
— avant le point de rupture pour la distribution par bloc.

Dans les deux cas, ce point correspond au cas où chacun des *threads* composant un *warp* procède à un accès distinct. Ces observations nous permettent de confirmer, dans ce cas de figure, que :

— l'utilisation d'accès communs entre *warps* ne permet pas d'accélérer le temps d'accès global à la constant memory,
— le temps d'accès global à la *constant memory* évolue avec le nombre d'accès distincts dans un *warp*.

Ce dernier point se vérifie dans les figures 5.3a et 5.4a où, dans le référentiel d'un *warp*, les courbes correspondant au temps d'accès à la *constant memory* sont similaires.

La pression sur les échanges de données en mémoire, exercée par le paramètre $c = 16$, implique un comportement distinct dans l'ensemble des cas. Dans le référentiel d'un *block*, nous observons dans les figures 5.1b et 5.2b le même phénomène de dégradation des temps d'exécution au-delà de 128 accès distincts. En revanche, dans le cadre d'un *warp* :

— la distribution cyclique de la figure 5.3b ne révèle pas de changement notable tandis que
— la distribution par blocs de la figure 5.4b connait un taux de croissance plus prononcé, au delà de 4 accès distincts.

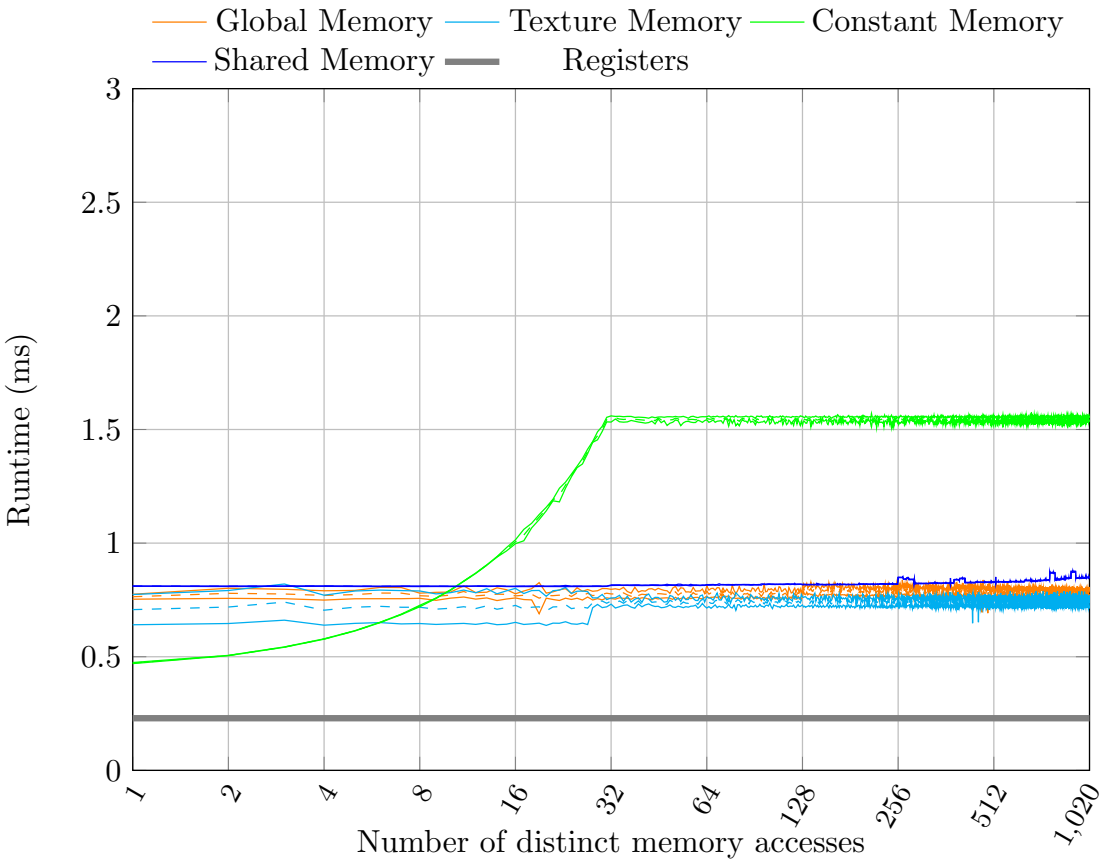
Dans le premier cas, le nombre d'accès distincts reste inférieur ou égal à 32 dans le référentiel d'un *block*. Dans le second cas, la rupture observée à partir de 4 accès distincts correpond à 128 accès distincts dans le référentiel d'un *block*. Nous en déduisons que seul le nombre d'accès distincts, global à un *block*, semble avoir un lien avec l'augmentation des temps d'accès globaux à la *constant memory*. Ces résultats nous révèlent l'implication d'une mémoire cache, pouvant potentiellement correspondre au cache L2 de la *device memory*, dans l'utilisation de la *constant memory*.

Enfin, nous constatons que la *constant memory* présente les meilleures performances d'accès en lecture (à l'exclusion des unités registre) pour moins de 8 accès distincts par *warp*. Cette valeur est cependant influençable par la valeur du *stride* des accès aux données comme le montre la figure 5.4b.

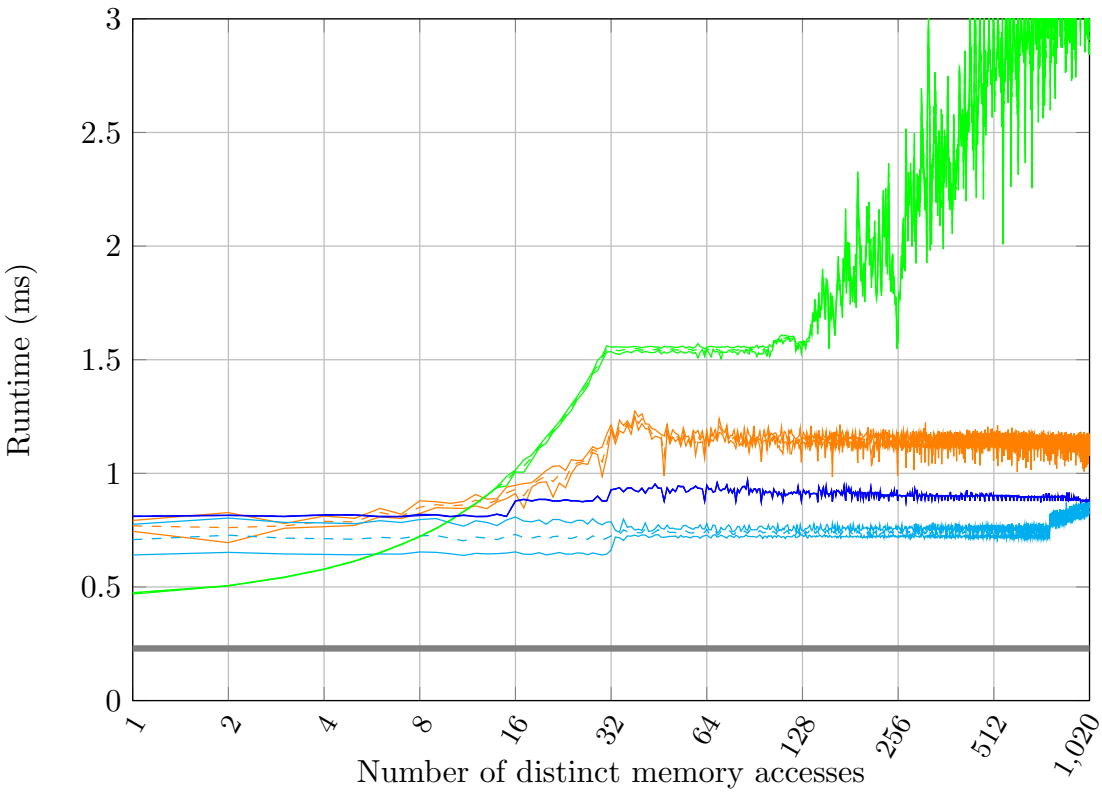
Pour la ***texture memory***, nous observons que la variance des temps d'exécution est plus importante :

— en-dessous de 32 accès distincts pour la distribution cyclique (figure 5.1a) et
— au-delà de 32 accès distincts pour la distribution par blocs (figure 5.2a).

Le nombre d'accès distincts par *warp* semble donc avoir un effet négatif sur la stabilité des temps d'accès en lecture pour la *texture memory*. Les figures 5.3a et 5.4a renforcent ce constat, avec une variance plus élevée sur l'ensemble de la courbe. Cependant, dans ce cas de figure, l'augmentation de la variance ne vient pas dégrader les temps d'accès. Les accès communs dans un *warp* permettent ici de réduire, de façon non prédictible, les temps d'accès en lecture à la *texture memory*.

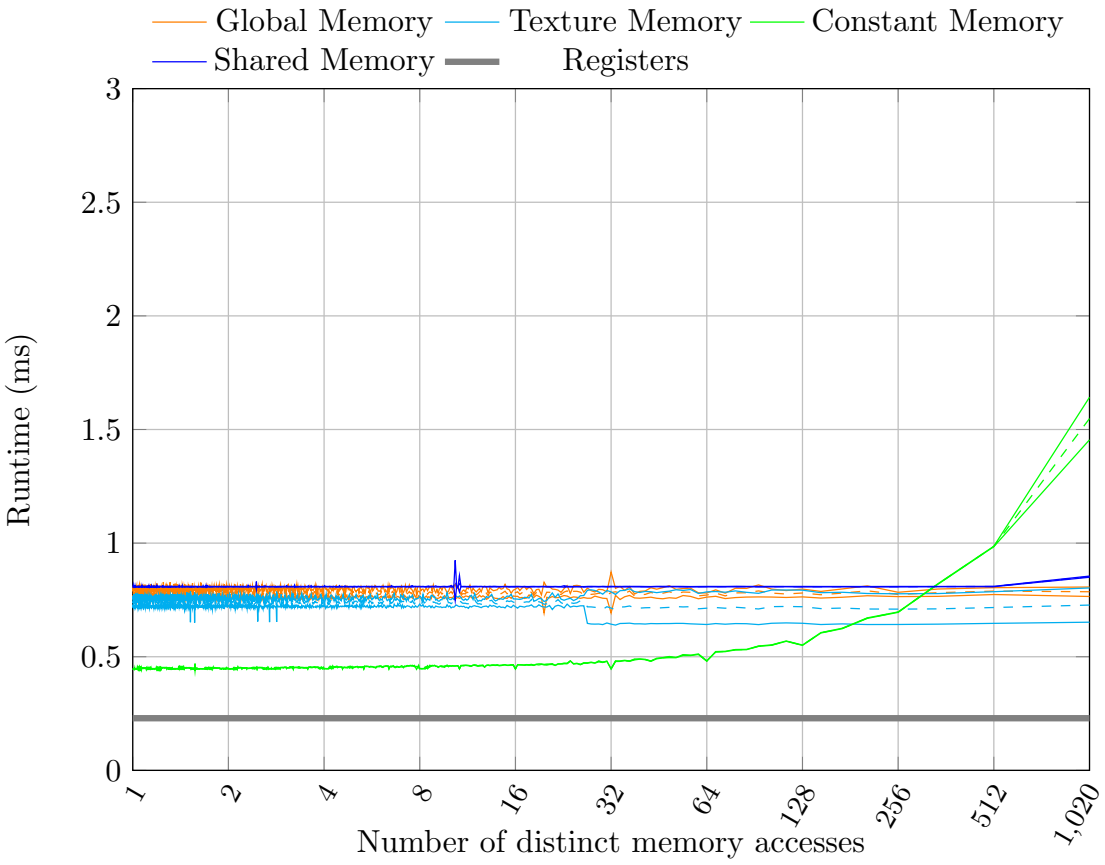


(a) Accès mémoire contigues. $c = 1$

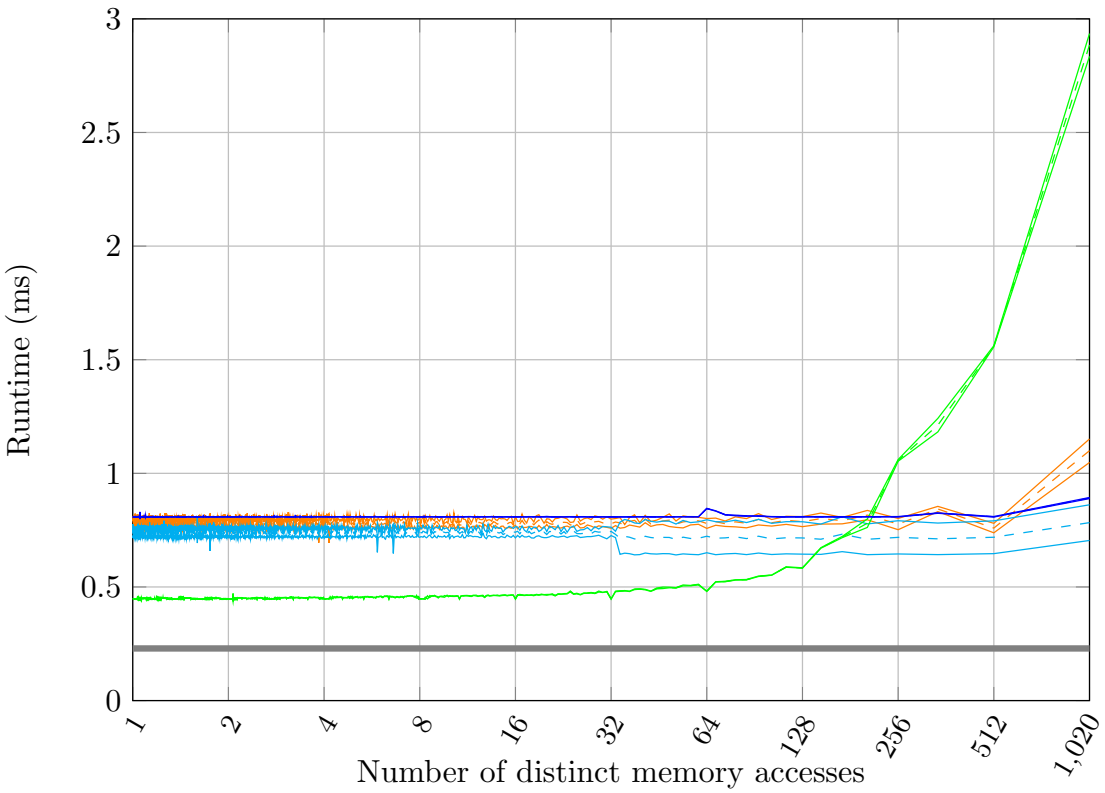


(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.1 – Temps d'accès moyen en lecture pour une distribution cyclique des accès mémoire sur Nvidia Quadro K2000. Fonction d'accès : $\mathcal{R}_1$. Référentiel : *Block*

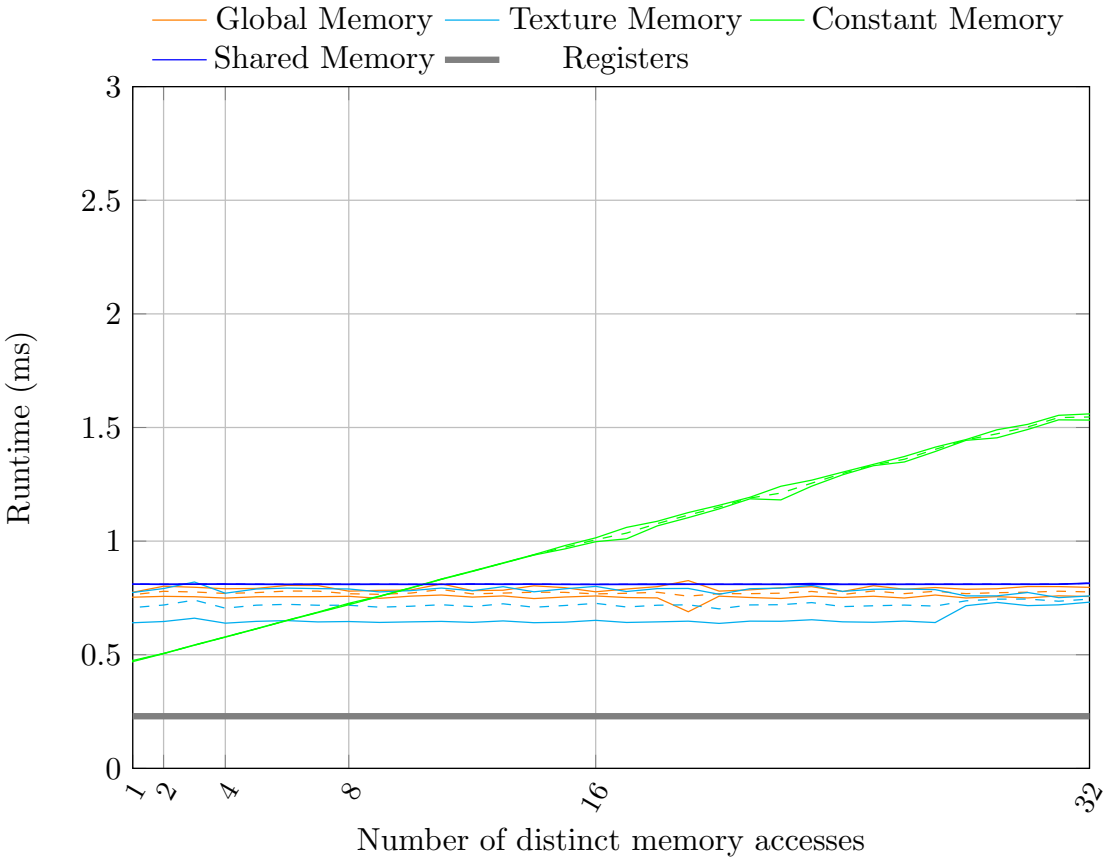


(a) Accès mémoire contigues. $c = 1$

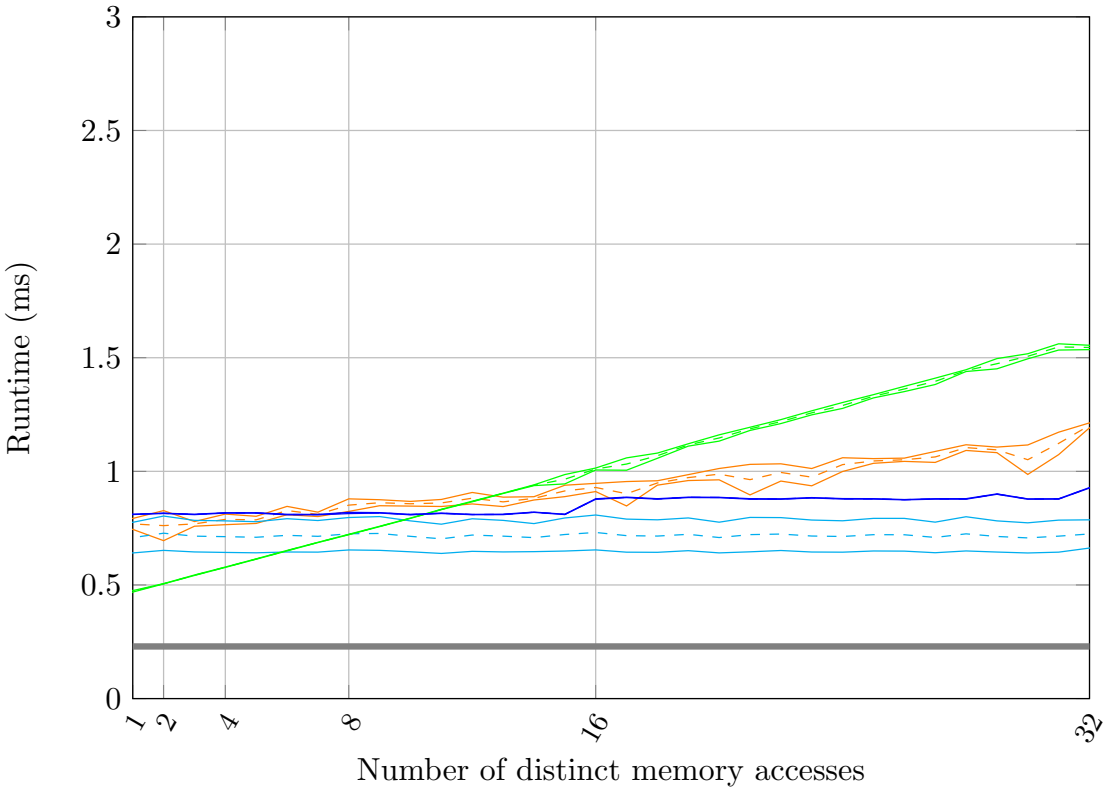


(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.2 – Temps d'accès moyen en lecture pour une distribution par blocs des accès mémoire sur Nvidia Quadro K2000. Fonction d'accès : $\mathcal{R}_2$. Référentiel : *Block*

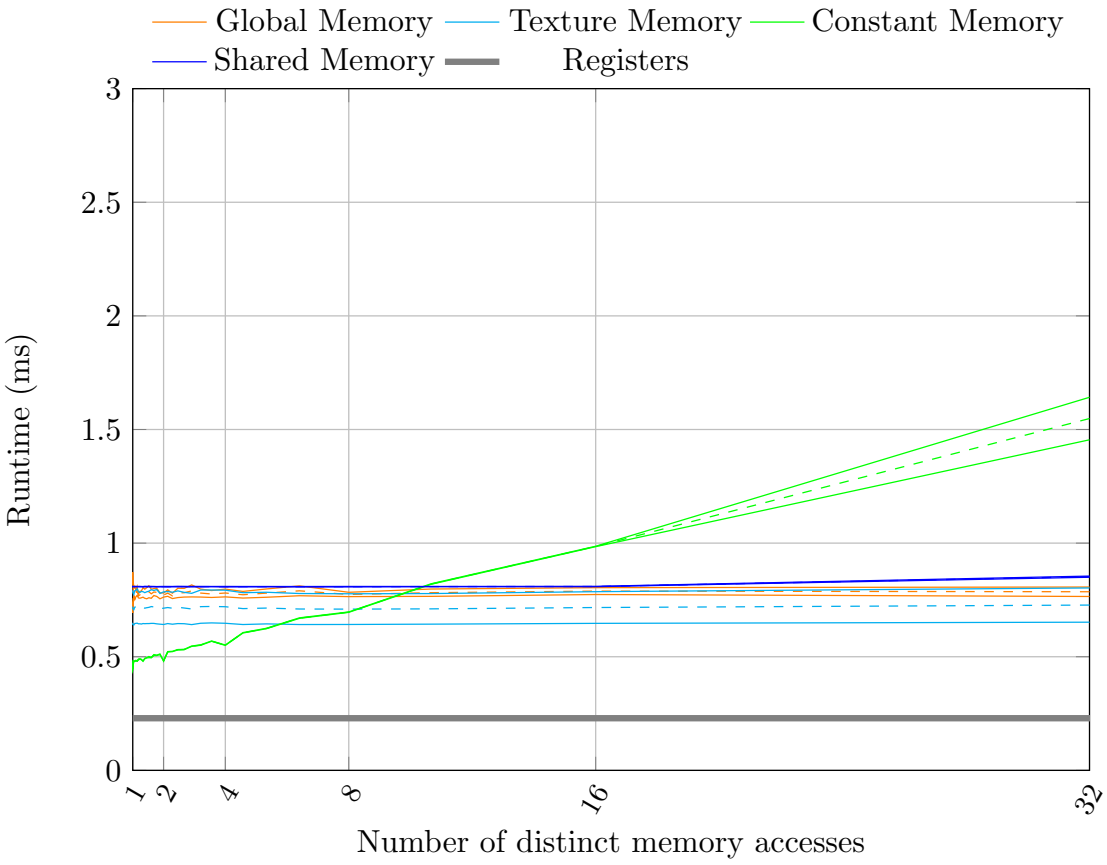


(a) Accès mémoire contigues. $c = 1$

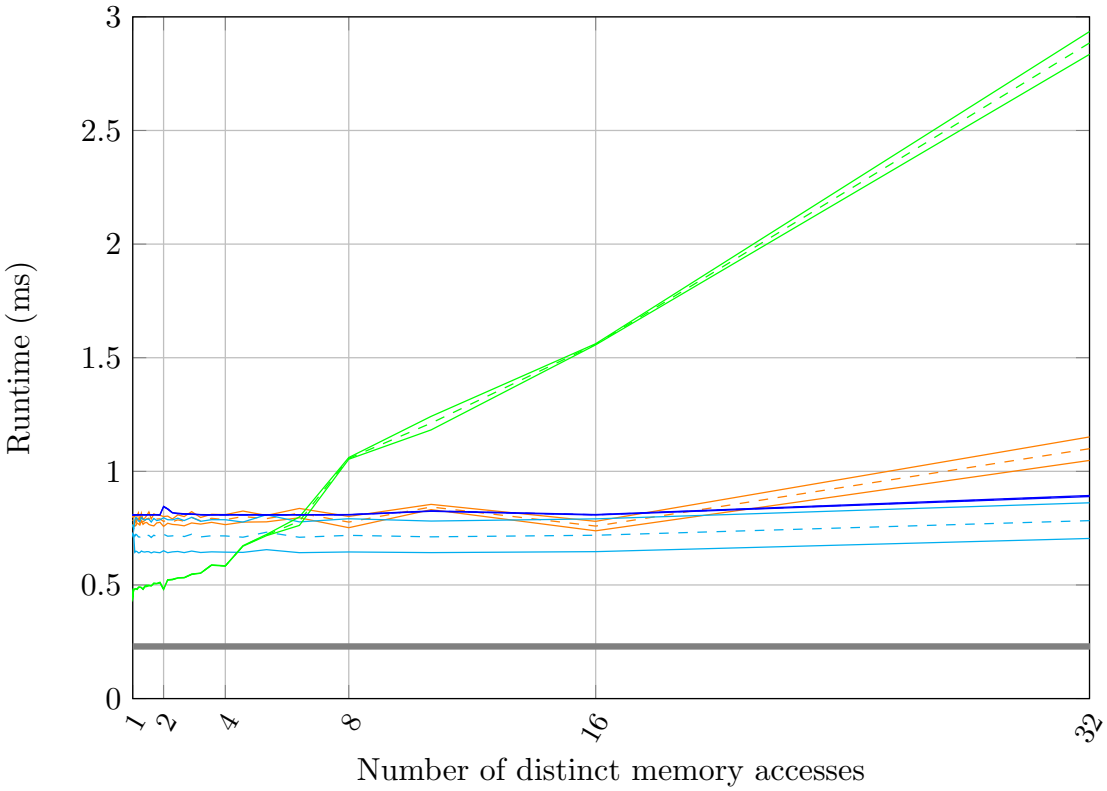


(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.3 – Temps d'accès moyen en lecture pour une distribution cyclique des accès mémoire sur Nvidia Quadro K2000. Fonction d'accès : $\mathcal{R}_1$. Référentiel : *Warp*



(a) Accès mémoire contigues. $c = 1$



(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.4 – Temps d'accès moyen en lecture pour une distribution par blocs des accès mémoire sur Nvidia Quadro K2000. Fonction d'accès : $\mathcal{R}_2$. Référentiel : *Warp*

Le *stride* pour l'accès aux données semble ne pas avoir d'effet notable sur les temps d'accès. Pour un *stride* de 16, nous ne constatons, dans les figures 5.1b et 5.2b, qu'une légère augmentation, lorsque le nombre d'accès distincts dans un *block* est proche de 1024. Le cache de la *texture memory* semble donc moins affecté par l'augmentation du stride dans les accès aux données. Ce constat reste cependant à vérifier avec l'augmentation du *stride* à des valeurs plus élevées.

Enfin, nous noterons que pour l'ensemble des cas, les temps d'accès à la *texture memory* restent inférieurs à ceux de la *global memory*.

L'évaluation de la ***global memory*** laisse apparaitre, dans le cas d'accès coalescents ($c = 1$), un temps d'accès stable et une faible variance pour les deux modèles de distribution (figures 5.1a et 5.2a). Cependant l'augmentation du *stride* laisse apparaître un impact direct sur les temps d'exécution. Dans les figures 5.1b et 5.3b, nous observons pour la distribution cyclique une augmentation linéaire des temps d'exécution jusqu'à environ 32 accès distincts. Au-delà de ce point, nous observons dans la figure 5.1b, un phénomène de saturation des temps d'accès. De plus, la figure 5.2b met en évidence l'aspect bénéfique de la localité temporelle, apportée par la distribution par blocs des accès, en repoussant l'augmentation des temps d'exécution au-delà de 128 accès distincts. Les accés commun entre *warps* ont donc un effet limité sur les temps d'accès.

Nous en concluons que :

— la coalescence des données permet d'améliorer les temps d'accès à la *global memory*,
— les accès communs dans un *warp* permettent de limiter la pénalité des accès non coalescents.

La ***shared memory*** présente pour l'ensemble des résultats une stabilité supérieure et une faible variance des temps d'exécution. Pour rappel, nous évaluons cet espace mémoire conjointement à la *global memory*. Les données, provenant de cette dernière, transitent exclusivement par le cache L2 tandis que le cache L1 est dédié à la *local memory*. Nous observons, dans les figures 5.1a, 5.2a, 5.3a et 5.4a, que la *shared memory* ne permet pas d'obtenir des temps d'accès inférieurs à ceux de la *global memory* lors d'accès coalescents et redondants. Le cache L2 permet ainsi d'atteindre un niveau de performance équivalent dans ce cas. En revanche, lors d'accès non coalescents (figures 5.1b, 5.2b, 5.3b et 5.4b), la *shared memory* permet de limiter la pénalité des *cache-miss* sur la *global memory* :

— en effectuant le préchargement des données de la *global memory* vers la *shared memory*,
— en chargeant les données sur la *shared memory* de manière à recréer une coalescence des accès en lecture, améliorant ainsi la localité spatiale.

Le gain apporté par la *shared memory* est particulièrement visible dans la figure 5.1b. la distribution cyclique ayant un impact sur la localité temporelle des données, la *shared memory* permet de limiter le phénomène de *cache-miss* par une gestion manuelle de la persistence des données.

Nous en déduisons que :

— l'usage systématique de la *shared memory* n'est pas fondé du fait qu'elle n'apporte pas un gain systématique par rapport au cache L2,
— la *shared memory* permet de limiter la dégradation des temps d'accès à la *global memory* notamment lors d'accès distincts et non coalescents dans les *warps*.

Cependant, notre utilisation de la *shared memory* évite les phénomènes de conflit sur les accès concurrents en lecture aux *memory banks* de la *shared memory*. Dans le cas contraire, les performances de cet espace mémoire auraient été dégradées.

En **conclusion** : Les résultats présentés concernent le GPU *Quadro K2000*, basé sur une architecture *Kepler*.

Pour notre cas d'évaluation, les temps d'accès en lecture [4] sont minimisés en utilisant :
— la *constant memory* lorsque le nombre d'accès distincts par *warp* reste inférieur à huit et
— la *texture memory* au-delà.

Ce point de transition est cependant influencé par :
— le nombre d'accès distincts entre *warps* et
— la taille du *stride* pour l'accès aux données.

Le domaine de supériorité de la *constant memory* est alors réduit au profit de la *texture memory*.

Au sujet de la stabilité des temps d'accès en lecture, la *shared memory* a montré une variance minimale pour l'ensemble des cas testés. Ainsi, pour de multiples exécutions, cet espace mémoire offre la meilleure récurrence des temps d'accès.

Résultats pour la Jetson TX1 - T210

Nous abordons à présents les résultats obtenus pour le GPU de la *TX1*. Notre analyse se focalise en particulier sur les différences observables, par rapport aux résultats du GPU *Quadro K2000* d'Endicott.

Dans le **référentiel d'un *block***, les temps d'exécution pour les différents espaces mémoire sont représentés :
— dans la figure 5.5 pour la **distribution cyclique** des accès et
— dans la figure 5.6 pour la **distribution par bloc**.

Dans le **référentiel d'un *warp***, les résultats sont représentés :
— dans la figure 5.7 pour la **distribution cyclique** et
— dans la figure 5.8 pour la **distribution par bloc**.

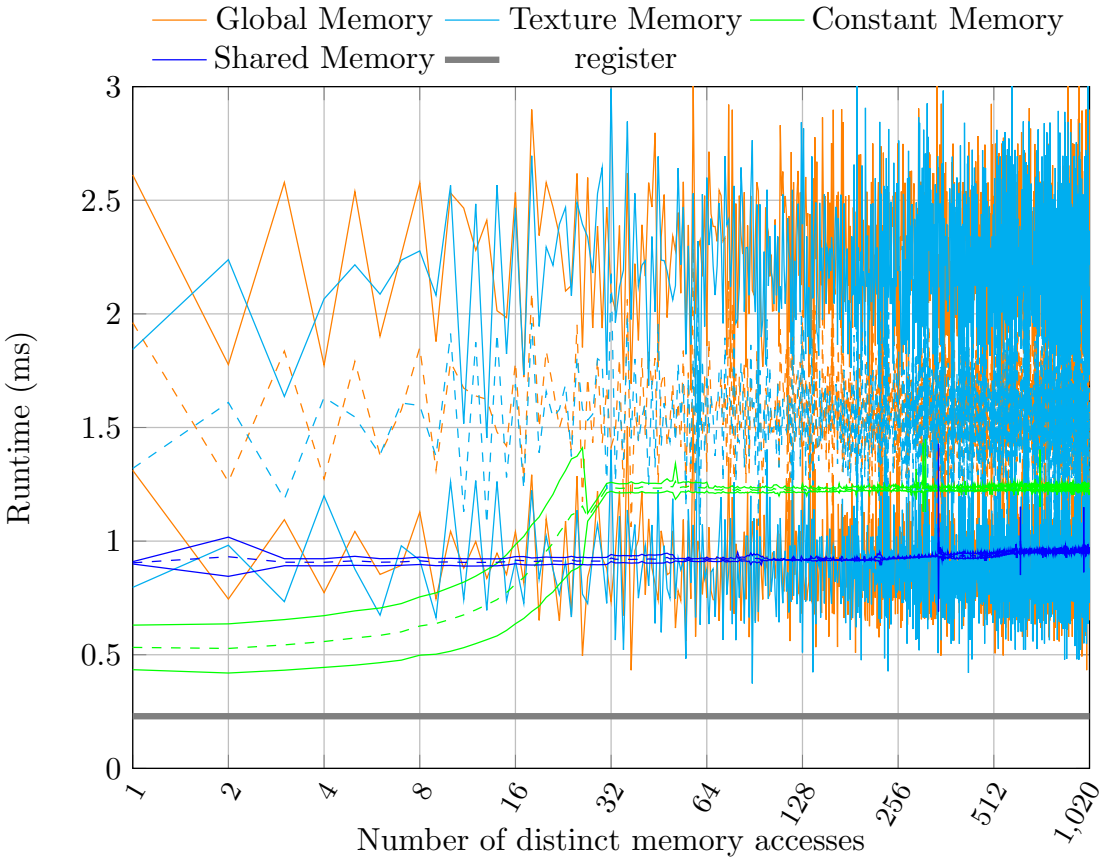
Comparée à la *Quadro K2000*, nous relevons, pour l'ensemble des représentations, une **nette augmentation de la variance** des temps d'exécution. Le format basse consommation des mémoires *LPDDR4* de la *TX1* semble être une raison légitime. Afin de vérifier ce point, il serait nécessaire, en complément de cette étude, d'effectuer la même évaluation sur une architecture comparable à la *Quadro K2000*, soit :
— de type *Maxwell* et
— employant des unités mémoire de type *GDDR5*.

Nous constatons, cependant, que la ***shared memory*** fait exception avec une variance nettement plus faible, comparable à celle de la Quadro K2000. Pour l'ensemble des cas, elle permet d'améliorer sensiblement le temps d'accès aux données de la *global memory*.

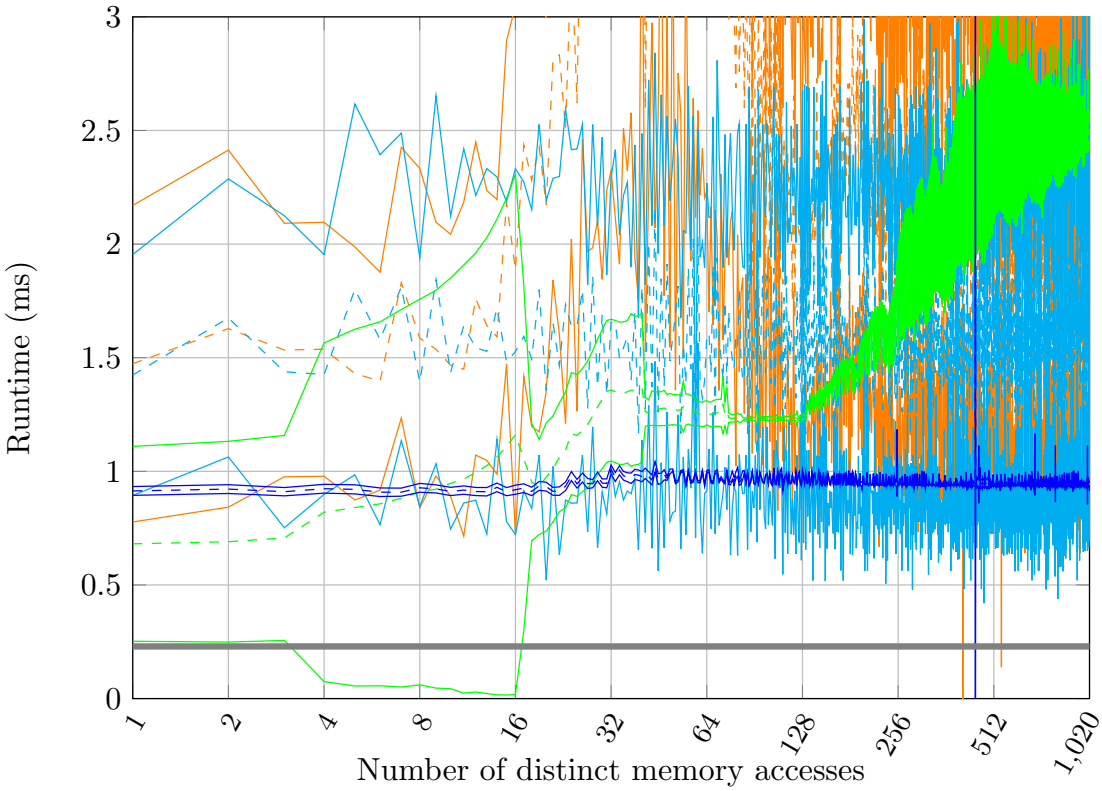
Pour la ***constant memory***, en ne considérant pas la variation plus prononcée des mesures, les résultats sont similaires à ceux d'Endicott. Nous retrouvons ainsi le même comportement, incitant à maximiser le nombre d'accès communs entre les *threads* d'un même *warp*. Au niveau de la variance, nous notons en particulier une forte augmentation, lorsqu'il existe une réutilisation des données dans un *warp*. Dans le cas contraire, la variance reste modérée au prix de l'augmentation attendue des temps d'accès. Les accès à la constant memory étant linéarisés, l'absence de réutilisation des données engendre une augmentation des *cache-misses*. Ces derniers se traduisent par une pénalité systématique des temps d'accès, caractérisée par une faible variance des résultats. La *constant memory* permet ainsi, pour cette architecture, d'améliorer les temps d'accès en échange d'une dégradation de leur stabilité.

À la différence de la quadro K2000, la ***texture memory*** et la ***global memory*** présentent des performances similaires pour leur temps d'accès moyen. Cependant, la *global*

4. à l'exception des accès aux registres

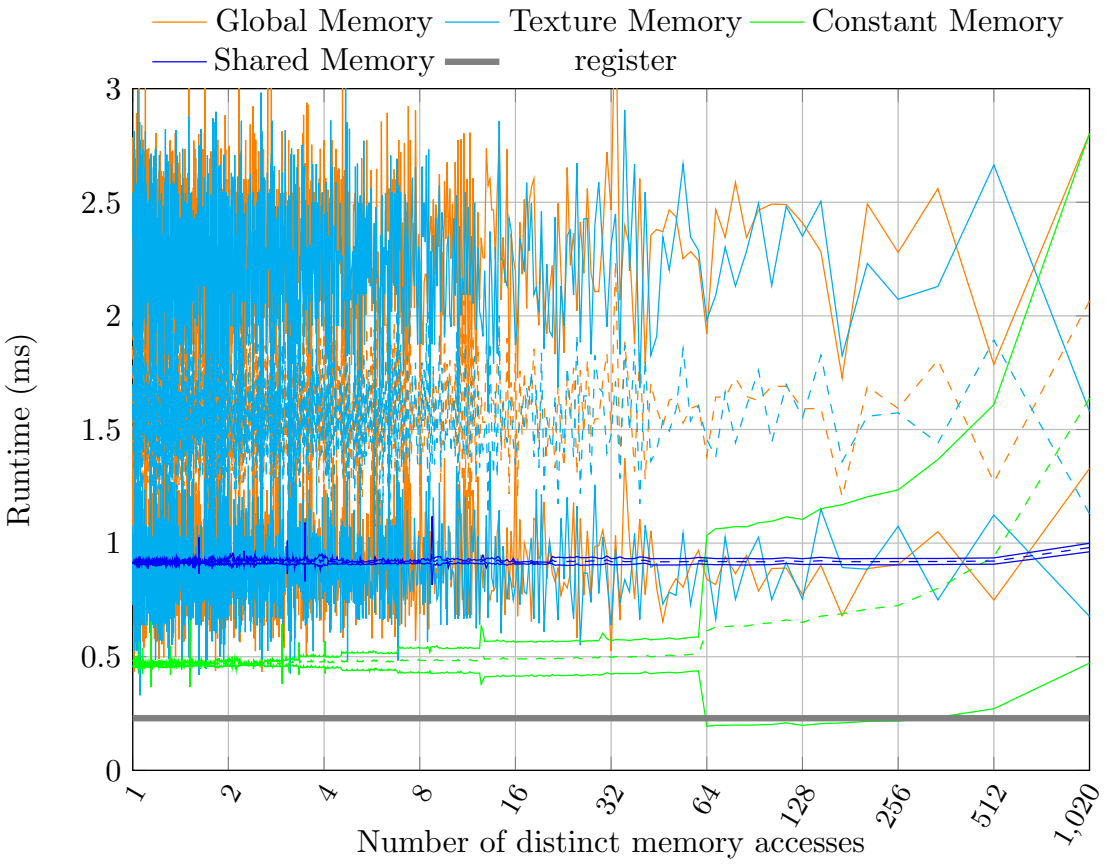


(a) Accès mémoire contigues. $c = 1$

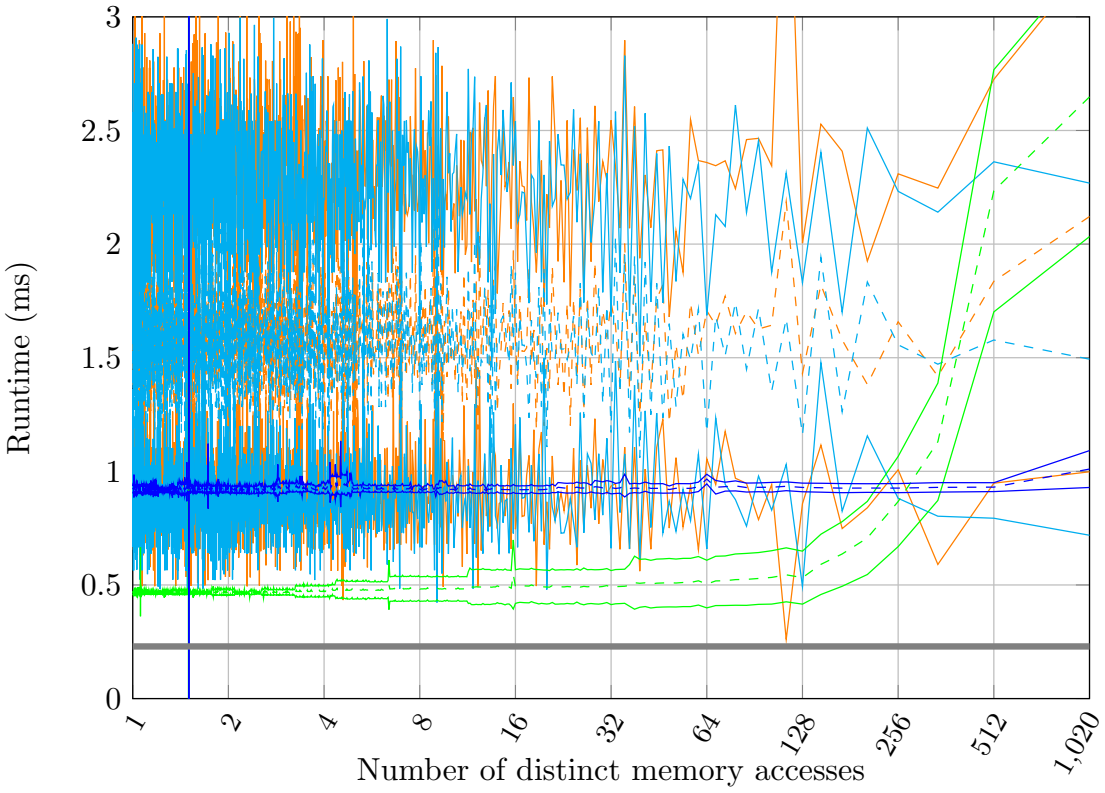


(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.5 – Temps d'accès moyen en lecture pour une distribution cyclique des accès mémoire sur Nvidia TX1. Fonction d'accès : $\mathcal{R}_1$. Référentiel : *Block*

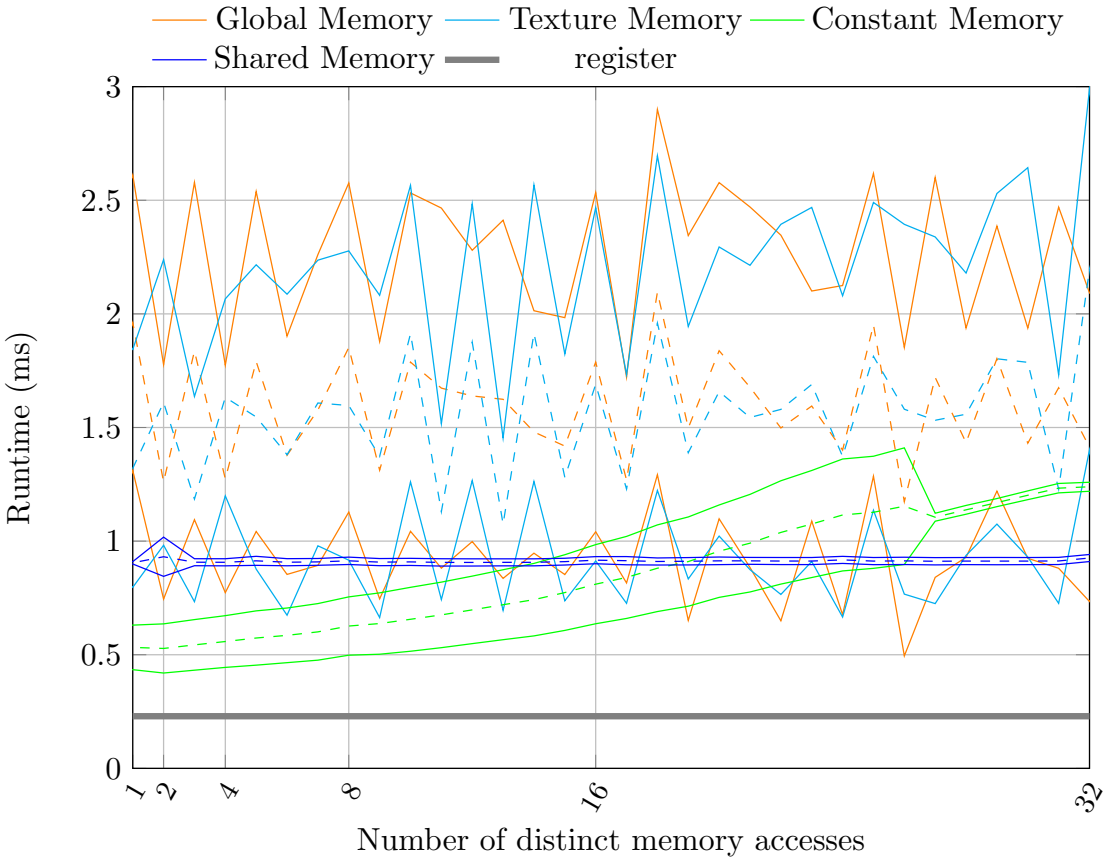


(a) Accès mémoire contigues. $c = 1$

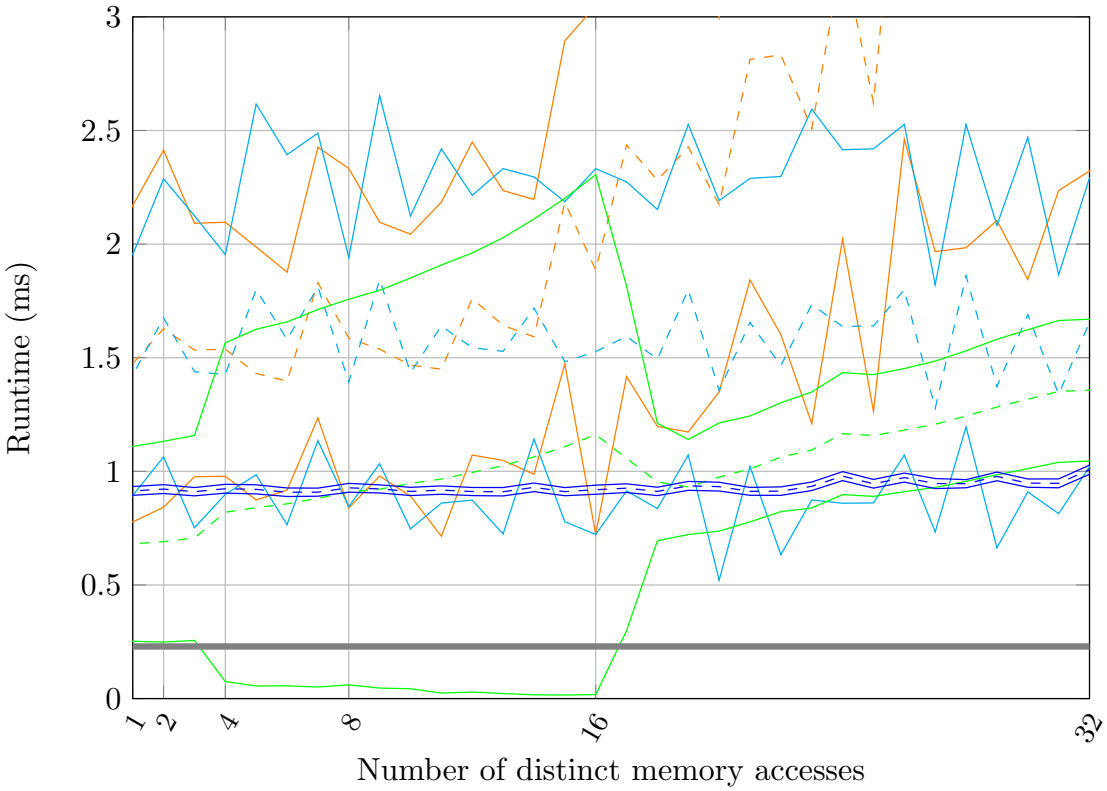


(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.6 – Temps d'accès moyen en lecture pour une distribution par blocs des accès mémoire sur Nvidia TX1. Fonction d'accès : $\mathcal{R}_2$. Référentiel : *Block*

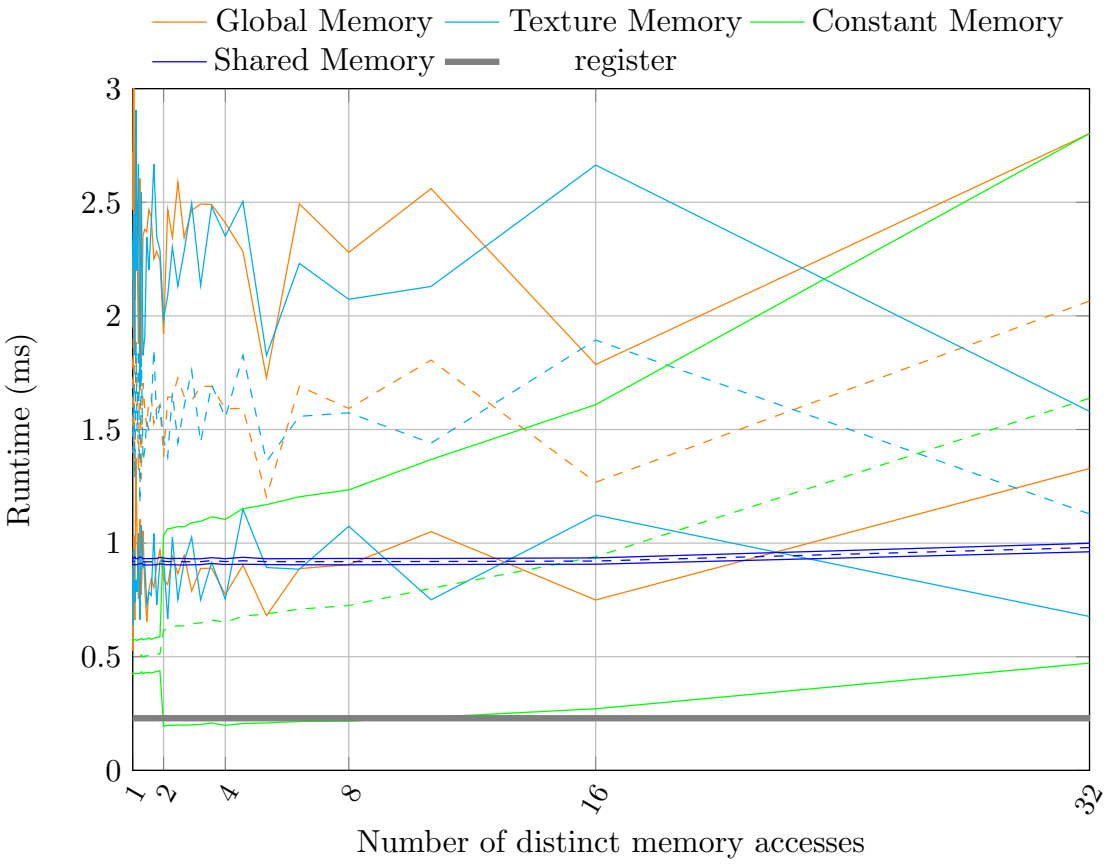


(a) Accès mémoire contigues. $c = 1$

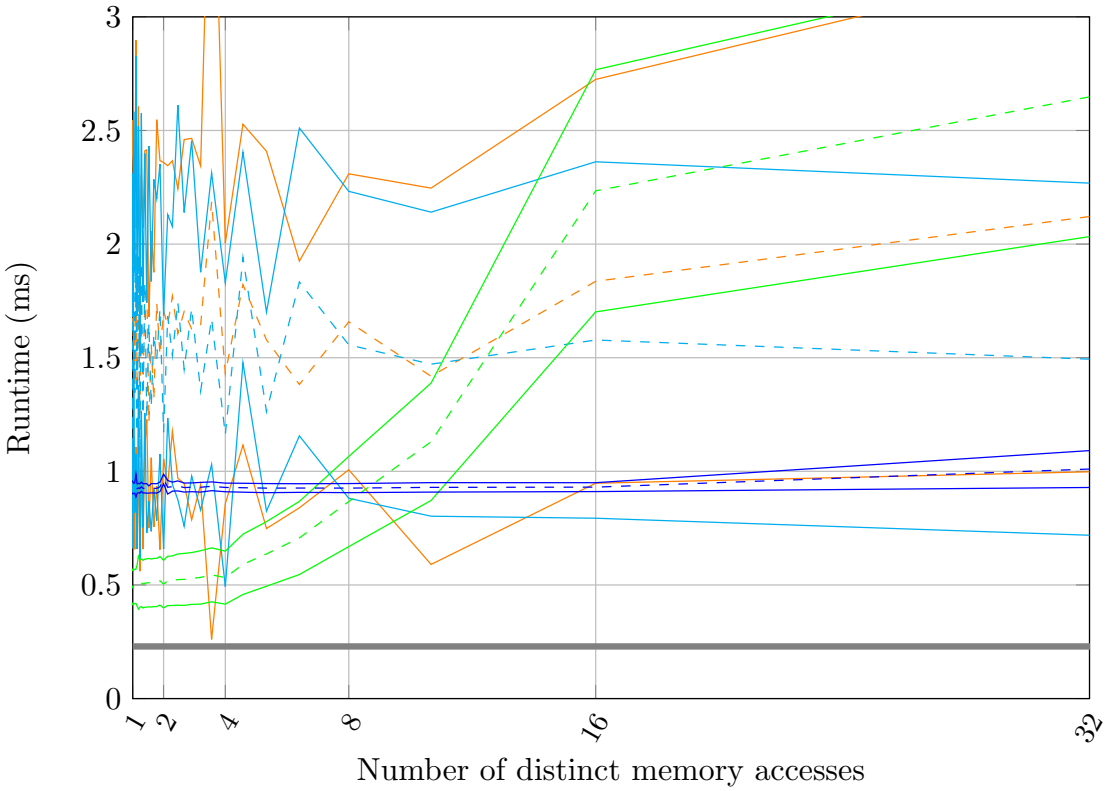


(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.7 – Temps d'accès moyen en lecture pour une distribution cyclique des accès mémoire sur Nvidia TX1. Fonction d'accès : $\mathcal{R}_1$. Référentiel : *Warp*



(a) Accès mémoire contigues. $c = 1$



(b) Amplification des communications mémoires. $c = 16$

FIGURE 5.8 – Temps d'accès moyen en lecture pour une distribution par blocs des accès mémoire sur Nvidia Quadro K2000. Fonction d'accès : $\mathcal{R}_2$. Référentiel : *Warp*

memory reste plus sensible à l'augmentation de la taille du *stride* lors des accès aux données. Ce phénomène est particulièrement observable dans la figure 5.7b, où l'augmentation du nombre d'accès distincts, affecté d'un stride de 16 $Bytes$, engendre une augmentation du temps d'accès global.

En **conclusion** : Jusqu'à 16 accès distincts par *warp*, l'usage de la *constant memory* permet de minimiser les temps d'accès, lorsque ceux-là sont contigus. Au-delà, le placement des données communes en *shared memory* donne les meilleurs résultats. Ce point de transition est cependant influencé par le *stride* d'accès aux données. Enfin, même si la *texture memory* présente des résultats similaires, comparé à la *global memory*, sa meilleure résistance au stride la rend préférable.

Pour la variance des résultats, l'augmentation relevée nous laisse supposer l'usage d'un modèle de *mapping* des mémoires cache basé sur un algorithme aléatoire de remplacement des lignes de cache. Ce type d'algorithme, du fait de sa simplicité de mise en œuvre, permet de répondre au besoin d'économie en énergie. Ce dernier point est cohérent avec les applications embarquées visées par la TX1. Ce principe de renouvellement du cache est d'ailleurs employé pour les architectures CPU basse consommation du fabricant *ARM* et notamment pour l'*A57* présent dans la *TX1*.

5.1.5 Conclusion

Nous avons étudié les performances en lecture de quatre espaces mémoire : la *global memory*, la *texture memory*, la *constant memory* et la *shared memory*. Cette analyse s'est focalisée en particulier sur l'influence des temps d'accès mémoire en fonction de trois principaux paramètres :

— le nombre d'accès distincts dans chaque *block*,
— la distribution des accès distincts et
— le *stride* des accès aux données.

Les communications mémoire constituent ici le facteur influençant directement le temps d'exécution des *kernels*.

Les expérimentations ont été menées sur deux architectures GPU distinctes : La *Quadro K2000* et la *TX1*. La première est de génération *Kepler* et dispose d'une mémoire dédiée de type *GDDR5*. La seconde, adaptée aux contraintes énergétique d'un usage embarqué, est de génération *Maxwell* et partage sa mémoire de type *LPDDR4* avec le processeur hôte.

Entre les deux architectures, nous retrouvons des similitudes sur le comportement des différents espaces mémoire. Ainsi, lorsque les accès sont coalescents, nous avons observé sur la *Quadro K2000* les temps d'accès minimaux pour :

— la *constant memory* de 1 à 8 accès communs par *warp* et
— la *texture memory* au-delà de 8 accès communs.

Pour la *TX1*, les temps d'accès sont minimaux pour :

— la *constant memory* de 1 à 16 accès communs par *warp* et
— la *shared memory* au-delà de 16 accès communs.

Ce point de transition varie cependant, au profit de la *shared memory*, en augmentant la valeur du *stride* lors des accès mémoire.

Cependant, la Jetson TX1 présente une variance des temps d'accès plus prononcée. Ce constat peut s'expliquer par :

— la mémoire basse consomation de type *LPDDR4* utilisée et
— l'utilisation d'une méthode de remplacement aléatoire des lignes de cache qui est plus économe en consommation énergétique.

Seule la *shared memoy* a présenté une faible variance sur ses temps d'accès.

Afin de compléter ces résultats, il conviendrait d'étudier en complément :

— les performances des accès en écriture,
— la variation de la taille du *stride*,
— la variation de la taille des données utilisées,
— la variation de la quantité globale de données exploitées,
— la variation du nombre d'accès communs entre *blocs*, afin d'évaluer le comportements entre les SMs,
— les cas où la bande passante des communications de données ne limite pas le temps d'exécution du *kernel* et
— le phénomène de conflit sur les banques de données de la *shared memory*.

L'idée serait alors de pouvoir déterminer selon une méthode heuristique l'espace mémoire adapté à chaque espace de données utilisé dans un algorithme. Il serait alors possible d'inclure la sélection des espaces mémoire dans notre méthodologie. Face au nombre de paramètres à prendre en compte, des résultats différents pour chaque architecture GPU et des fonctions de coût considérées (temps d'accès, consommation energétique, ...), il serait intéressant d'étudier les résultats que pourrait apporter un réseau de neuronnes, entraîné à partir de l'extension du modèle de *benchmark* abordé dans cette section.

5.2 Exploitation du parallélisme *coarse grain* sur GPUs Nvidia

Dans la précédente section, nous avons abordé l'analyse des performances des différents espaces mémoire mis à disposition par CUDA. Nous nous intéressons à présent à la capacité et aux performances du GPU pour exploiter le parallélisme à gros grain (*coarse grain*).

En programmation, le parallélisme est applicable :

— aux instructions (ILP),
— aux itérations de boucles,
— aux sections parallèles ou encore
— aux programmes informatiques.

Ce classement, par ordre croissant de granularité, est corrélé à la quantité et à la taille des tâches pouvant être réalisées de manière concurrentielle. La principale difficulté est alors de trouver le bon niveau de segmentation en fonction des capacité de traitement parallèle de l'architecture matérielle ciblée.

Le parallélisme à gros grain se distingue du parallélisme à grain fin (*fine grain*) par la mise en concurrence de tâches :

— plus importantes (souvent assimilées à un programme informatique) et
— dont le temps d'exécution est irrégulier.

L'ensemble des solutions de transformation automatique de code, listées dans la section 2.2, exploitent principalement, pour le placement sur GPU, le parallélisme à grain fin. Chaque itération d'un nid de boucles est alors transformé en un *thread*, exécuté par un *CUDA core*. Cette méthode est particulièrement adaptée à la plupart des applications de traitement d'images, présentant des *patterns* algorithmiques réguliers sur des ensembles importants de données.

Cependant, il arrive, même en traitement d'images, que les portions parallèles de code présentent :

— des quantités irrégulières d'opérations à effectuer et/ou
— un faible nombre d'instances parallèles.

Le parallélisme à gros grain est alors adapté à ces caractéristiques.

Dans le cas de l'algorithme *simpleFlow* (section 4.2.1), les fonctions *selectPointsToRecalcFlow* et *extrapolateFlow* sont deux exemples où :

- les boucles externes du nid sont séquentielles et
- les boucles internes :
  - sont parallèles,
  - présentent un faible nombre d'itérations et
  - ont des bornes dynamiques.

Dans ces deux cas, le placement des boucles internes sur GPU ne permet pas de saturer les capacités calculatoires de cette architecture massivement parallèle. De plus, leur maintien sur le processeur hôte nécessite de mettre en place des transferts mémoire entre l'hôte et l'accélérateur afin de maintenir la cohérence des données partagées. Chaque *kernel* constituant pour le GPU un programme exécutable, nous envisageons, dans ce cas de figure, l'exploitation du parallélisme *coarse grain*. Nous étudions ainsi, dans cette section, la capacité ainsi que les performances du GPU pour exploiter cette forme de parallélisme.

À ce sujet, Amini [16] a évalué deux algorithmes de détection de parallélisme pour le placement de code sur GPU. Le premier, basé sur l'algorithme de parallélisation d'Allen et Kennedy [43], maximise la distribution des boucles selon un parallélisme à granularité fine et est particulièrement adapté aux architectures vectorielles. Le second [79], utilisant les analyses de régions de tableau [37], permet une parallélisation de type *coarse grain* sur les données. Amini a conclu que l'algorithme d'Allen et Kennedy présentait des performances supérieures avec un temps d'exécution quatre fois plus faible sur architecture *Kepler*. Cependant, contrairement à notre évaluation, l'approche *coarse grain* a été utilisée par Amini :

- pour du parallélisme de donnée,
- pour un nid présentant un nombre élevé d'itérations indépendantes (les boucles parallèles i et j présentent un total de $\sum_{i=1}^{2999} \sum_{j=i}^{2999} j = 4\ 498\ 500$ itérations) et
- pour un pattern de calcul parfaitement régulier.

Nous détaillons brièvement, dans la section 5.2.1, l'état de l'art ainsi que les caractéristiques des techniques des GPUs permettant l'exécution d'opérations concurrentes. Nous décrivons ensuite, dans la section 5.2.2, l'objectif de cette expérimentation. Le protocole expérimental est défini dans la section 5.2.3. Enfin, les résultats sont analysés et interprétés dans la section 5.2.4.

5.2.1 Description du parallélisme *coarse grain* pour les GPUs

Nous abordons, dans cette expérimentation, le parallélisme de tâches. Nous étudions ainsi, selon la taxonomie de Flynn, l'aspect MIMD des GPUs.

À partir de la génération *Kepler*, les GPUs Nvidia ont la capacité d'exploiter de manière concurrentielle plusieurs *pipelines* d'instructions, alimentés par le processeur hôte. Ces *pipelines*, portant le nom de *CUDA stream*, permettent d'exploiter le parallélisme *coarse grain*, au niveau des *kernels*, sur GPU. Ils sont notamment utilisés pour :

- la concurrence entre le processeur hôte (CPU) et l'accélérateur GPU,
- la concurrence entre accélérateurs GPU,
- les transferts mémoire asynchrones entre l'hôte et l'accélérateur et
- la concurrence entre *kernels* dans un même GPU.

Les instructions *CUDA event* fournissent alors les informations permettant la synchronisation entre *CUDA streams*. Par défaut, un unique *CUDA stream*, d'identifiant 0, est utilisé. L'exploitation de *CUDA streams* supplémentaires requière une déclaration explicite de la part du développeur.

Nous abordons à présent chacune des approches permettant l'exploitation du parallélisme *coarse grain* sur GPU.

Concurrence CPU/GPU

Cette forme de concurrence exploite un unique *CUDA stream*. Elle repose sur le fonctionnement nativement asynchrone du GPU pour l'exécution des *kernels*. Le processeur hôte empile ainsi, dans le *CUDA stream* choisi, les ordres d'exécution de *kernel* sans attendre de retour. Son exploitation ne requière donc aucune action de la part du développeur. Cette solution permet :

— de masquer l'*overhead* lié au lancement des *kernels* et
— d'exécuter des tâches concurrentes sur le processeur hôte.

Les communications synchrones de données sont souvent utilisées pour assurer la synchronisation entre le processeur hôte et l'accélérateur.

La principale problématique réside dans le bon équilibrage des charges entre l'hôte et l'accélérateur. À ce sujet, un état de l'art assez complet a été réalisé par Mittal et Vetter [104].

Concurrence inter-GPUs

Dans le cadre du parallélisme de tâches, la concurrence inter-GPUs permet de distribuer l'exécution de *kernels* indépendants sur différents GPUs. Le processeur hôte communique alors avec chaque GPU au moyen d'un *CUDA stream* dédié et leur synchronisation est réalisée au moyen d'instructions *CUDA event*. La sélection d'un GPU est effectué au moyen de la fonction *cudaSetDevice*. Chaque GPU constitue un nœud de type Non Uniform Memory Access (NUMA) dans l'architecture globale.

Les problématiques liées à l'usage de plusieurs GPUs sont :

— la répartition des données sur les GPUs,
— la minimisation des données échangées et
— la minimisation des dépendances entre les *kernels* distribués.

Dans l'état de l'art, StarPU [21] et R-Stream [94] sont capables de distribuer les *kernels* et leurs données associées sur plusieurs GPU.

Concurrence des tranferts de données

En complément de la concurrence portant sur l'exécution de *kernels*, il est possible d'effectuer les transferts de données de manière asynchrone au moyen d'un *CUDA stream* dédié. Cette forme de concurrence permet de masquer le temps de transfert des données en exécutant sur le GPU un ou plusieurs *kernels* concurrents. Le transfert est alors assuré par un *copy engine*, capable d'effectuer des requêtes Direct Memory Access (DMA) au travers du bus PCIe. De ce fait, les données dans l'espace mémoire du processeur hôte doivent être déclarées comme *page locked memory* (appelée *pinned memory* par CUDA). Enfin, l'achèvement d'un transfert est signalé au moyen d'instructions *CUDA event*.

Concurrence intra-GPU

Pour ce dernier cas, plusieurs *kernels* sont exécutés en concurrence dans un unique GPU. À la différence de la concurrence inter-GPUs, les données résident dans la mémoire globale du GPU, qui est unifiée selon un modèle d'architecture SMP. Ce sujet a été abordé par Guevara *et al.* [69]. Cependant leur approche est antérieure à l'existence des *CUDA streams*. Plus récemment Cruz *et al.* [39] ont aussi abordés ce sujet.

Nous retrouvons en complément le parallélisme dynamique chez Nvidia. Celui-ci permet à une instance de *kernel* d'ordonner l'exécution d'un nouvel ensemble d'instances de *kernel*.

5.2.2 Description du sujet d'expérience

Nous nous intéressons à la concurence de *kernels* intra-GPU afin de trouver une solution aux problématiques de granularité irrégulière et de sous-utilisation des ressources calculatoires du GPU. L'enjeux est alors de réduire le taux d'activité du CPU au profit du GPU, ce qui a pour conséquence de limiter les transferts de données entre CPU et GPU.

L'objectif de cette expérimentation est double :

1. Nous cherchons à définir la quantité minimale d'instances de *threads* pour surcharger les unités SM.
2. Nous souhaitons étudier le comportement d'un GPU pour un placement parallèle de type *coarse grain*.

L'architecture des GPUs, décrite dans le chapitre 1, est basée sur un modèle hiérarchique à deux niveaux, composé de processeurs SM intégrant des Cuda *cores*. Lors de l'exécution d'un *kernel* :

— la grille des instances de *threads* est décomposée en *blocs*[5], distribués par le GPU sur les différents SMs,
— ces *blocs* sont eux-même décomposés en *warps* de 32 *threads* et placés sur les *CUDA cores* par les *warp schedulers* et les *dispatch units*.

Selon ce modèle architectural, les GPUs sont couramment cités pour leur pertinence à exploiter le parallélisme de données selon une granularité fine. L'usage courant est de sur-alimenter en *blocs* les *sm*s et en *threads* les *CUDA core*s, afin d'optimiser la bande passante des *pipelines* d'intructions. Cette sur-alimentation permet notamment de réduire le phénomène de "bulles" dans les *pipelines* selon le principe du Simultaneous multithreading (SMT). Les chiffres de 64, 128 ou encore 256 *threads* par *block* sont souvent avancés, sur les forums spécialisés, comme étant le point d'équilibre entre sous-alimentation et sur-alimentation des unités SM. Nous cherchons à déterminer la valeur exacte pour une architecture GPU donnée.

Le problème de sous-alimentation se pose lorsque le nombre défini de *blocs* et de *threads* est inférieur respectivement au nombre de SMs ou de *CUDA core*s disponibles. Ce type d'utilisation est moins performant, comparé à une exécution sur CPU, du fait de la frèquence de fonctionnement plus faible, du cache d'instructions réduit et du modèle simplifié d'exécution des instructions sur GPU. Nous souhaitons ainsi étudier, dans ce cas de figure, le comportement d'une architecture GPU selectionée, pour une exécution concurrentielle de *kernels*.

5.2.3 Protocole expérimental

Cette expérimentation a été exclusivement réalisée avec le GPU *Quadro K2000* de la plateforme *Endicott*. L'ensemble des caractéristiques de cette architecture *Kepler* sont détaillées dans la section 4.1. Selon le tableau 4.2 et la documentation CUDA [123], cette architecture GPU permet d'exécuter de manière simultanée jusque 16 *kernels* concurrents.

Notre objectif étant d'évaluer la concurrence des *kernels*, les durées présentées dans cette expérimentation correspondent uniquement au temps d'exécution des *kernels*. Les durées propres aux communications de données entre l'hôte et l'accélérateur ne sont pas prises en compte.

5. selon le paramètrage défini lors de l'appel au *kernel* par le développeur

Pour cette expérimentation, nous avons utilisé le modèle de *kernel* spécifié dans le listing 5.1. Nous avons distribué celui-ci sur 32 *CUDA streams*. Ce *kernel* effectue un traitement encapsulé dans une boucle *do/while*. Pour chaque itération de cette boucle, son entête exécute une nouvelle itération jusqu'à ce que le temps d'exécution de l'instance du *kernel* dépasse la valeur de la variable *clock_count*. Pour cette expérimentation nous avons fixé cette variable à $10\,ms$. Nous noterons que la durée d'exécution d'une instance de *kernel* ne pourra faire précisémment $10\,ms$. Nous considérons, cependant, cette différence négligeable.

Nous lançons l'exécution d'un *kernel* sur chacun des 32 *CUDA streams* et étudions la durée d'exécution globale pour l'ensemble de ces instances de *kernels*. Cette opération est réalisée pour plusieurs configurations de test, en faisant varier le nombre d'instances de *threads* et de *blocs*. Pour chaque configuration de test, le même paramètrage (*blocks/threads*) est utilisé pour l'ensemble des *kernels*. En conséquence de la combinatoire importante de paramètres (*blocks/threads*) à évaluer, l'expérimentation a été entièrement automatisée par l'utilisation d'instructions *cudaDeviceSynchronize* entre chaque configuration de test.

Les résultats ont été récupérés au moyen de *nvprof*, le *profiler* de Nvidia.

```
83 __global__ void concurrencyKernel(uint1* output, const unsigned int length, const clock_t clock_count){
84     const unsigned int id = blockIdx.x * blockDim.x + threadIdx.x;
85 
86     if(id<length){
87         const clock_t start_clock = clock();
88 
89         clock_t clock_offset;
90         do{
91             //Specialize kernel here with a distinct computation
92             //...
93             clock_offset = clock() - start_clock;
94         }while (clock_offset < clock_count);
95 
96         output[id] = (uint1) clock_offset;
97     }
```

Listing 5.1 – Modèle de *kernel* utilisé pour l'évaluation du parallélisme *coarse grain* sur GPU

5.2.4 Analyse et interprétation des résultats

L'ensemble des résultats obtenus sont visibles dans la figure 5.9. Ces données sont représentées en utilisant pour l'axe des *threads* :

— une échelle linéaire dans la figure 5.9a et

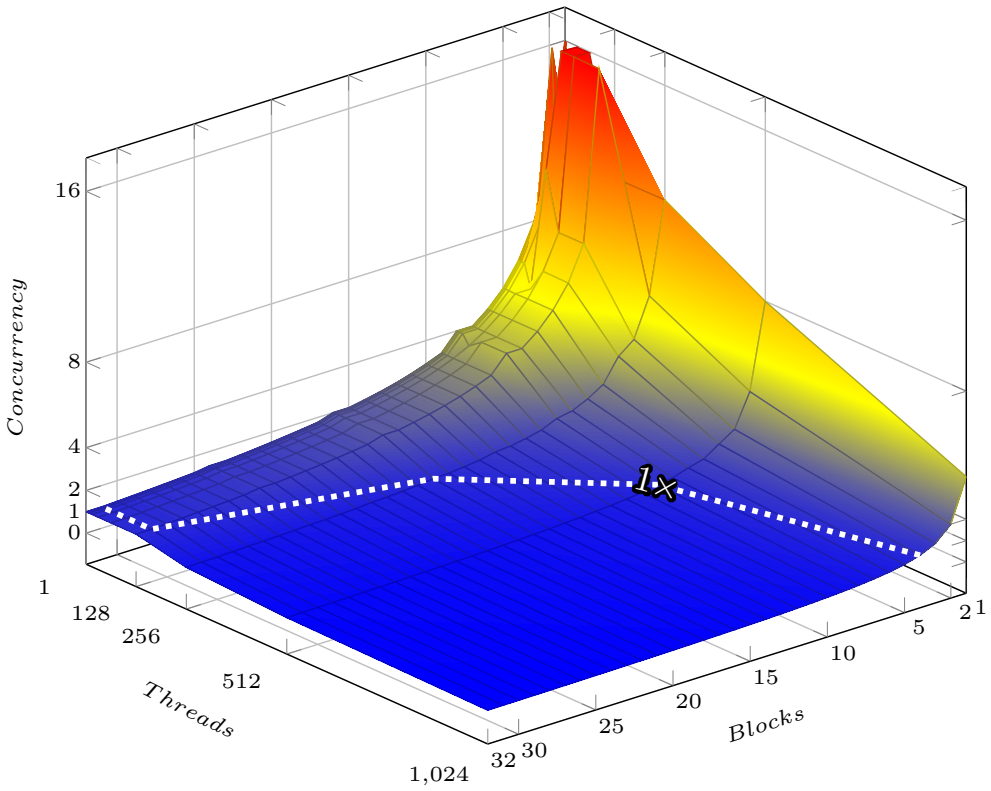
— une échelle logarithmique dans la figure 5.9b.

Pour représenter les résultats de cette expérimentation, nous avons utilisé la métrique *concurrency*, définie dans la formule 5.8.

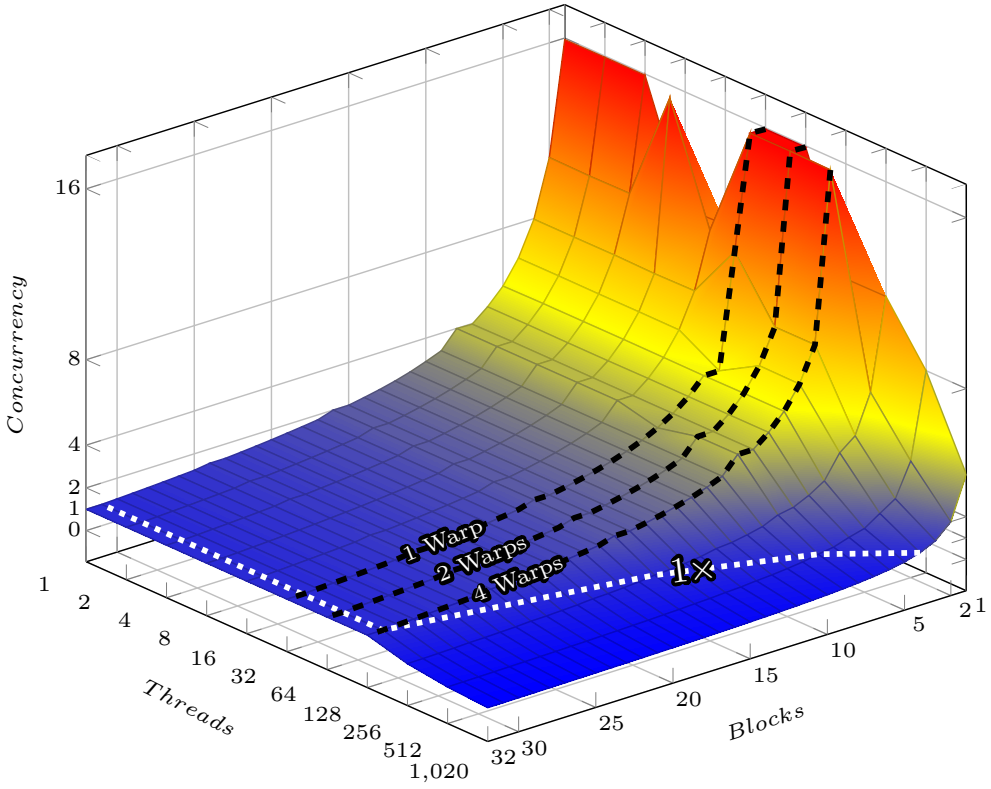
$$concurrency = \frac{\Delta t_{theor.}}{t_{max} - t_{min}} \tag{5.8}$$

Dans cette formule, pour chaque configuration de test (*blocks/threads*) :

— t_{max} correspond à la date la plus récente de fin d'exécution de *kernel*,



(a) Échelle de *threads* linéaire



(b) Échelle de *threads* logarithmique

FIGURE 5.9 – Analyse de la concurrence de kernels intra-GPU sur architecture Nvidia Kepler

— t_{min} à la date la plus ancienne de début d'exécution de *kernel* et
— $\Delta t_{theor.}$ aux temps global théorique donné par la formule 5.9. Pour ce dernier :
  — $nb_{kernels}$ correspond au nombre de *kernels* utilisés par configuration de test.
  — $t_{theor.}$ correspond au temps d'exécution théorique d'une instance de *kernel*.

$$\Delta t_{theor.} = nb_{kernels} \times t_{theor.} = 32 \times 10\,ms = 320\,ms \tag{5.9}$$

La métrique *concurrency* nous donne donc le rapport entre :
— le temps d'exécution théorique global des 32 *kernels*, pour une unique instance par *kernel*
— et le temps d'exécution global mesuré.

Nous considérons ainsi les cas :

concurrency > 1 : lorsque le GPU est sous-utilisé par un unique *kernel*. Dans le cadre d'un placement de type *coarse grain*, les unités de calcul inutilisées sont alors exploitées pour exécuter de manière concurrentielle les *kernels* des différents *CUDA streams*. La valeur de *concurrency* correspond dans ce cas au nombre moyen de *kernels* exécutés simultanément sur le GPU.

concurrency < 1 : lorsque la quantité d'instances de *threads* à traiter pour un même *kernel* vient surcharger la capacité de traitement des unités SM. Le temps d'exécution de ce *kernel* s'allonge et devient alors supérieur à $10\,ms$. L'exécution concurrente de *kernels* n'est alors plus adaptée et le placement converge vers l'exploitation d'un parallélisme à grain fin. La surcharge des SMs a pour bénéfice de masquer les bulles au sein des *pipelines* d'instructions selon un principe similaire à l'*hyperthreading* sur les CPUs *Intel*.

concurrency $= 1$: lorsque la capacité de traitement des unités SM est tout juste atteinte. Ce cas correspond au point d'équilibre entre les deux cas précédents.

En fonction des configurations de test (*blocks*/*threads*), le détail des données acquises, correspondant à la métrique *concurrency*, est fourni dans la table 5.1. Le code couleur utilisé dans cette table correspond à :

rouge : la zone de données où *concurrency* > 1.0.

vert : la zone de données où *concurrency* < 1.0.

bleu : la zone de données où *concurrency* ≈ 1.0.

Lors de leur représentation, l'intensité de chacune de ces couleurs est corrélée respectivement à la quantité de *kernels* exécutés en concurrence (rouge), à la saturation du GPU (vert) et à la proximité du point d'équilibre (bleu).

En complément, nous avons ajouté deux représentations graphiques de ces données (figure 5.9) afin d'améliorer la visualisation et l'interprétation du phénomène de concurrence sur GPU. Nous étudions à présent la surcharge des unités SMX et l'exécution concurrentielle de *kernels*.

Saturation des unités SMX

Nous avons tracé sur les figures 5.9a et 5.9b l'évolution du point d'équilibre (*concurrency* $= 1.0$) en fonction du nombre de *blocks* et de *threads* utilisés. Les valeurs intermédaires ont été extrapolées en utilisant une interpolation linéaire. La courbe résultante, représentée par des tirets blancs, est identifiable par la mention "1×". Cette courbe délimite, entre autres, la zone de données correspondant à la saturation des unités SMX (*concurrency* < 1.0).

concurrency		*blocks*															
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
threads (warps)	1 (1)	15.93	10.63	7.97	6.38	5.30	4.56	3.97	3.99	3.18	2.90	2.65	2.45	2.28	2.13	2.00	2.00
	2 (1)	15.94	10.61	7.97	6.39	5.32	4.56	3.99	3.99	3.19	2.90	2.66	2.45	2.28	2.13	1.99	2.00
	4 (1)	15.94	10.63	7.98	6.38	5.32	4.55	3.99	3.99	3.19	2.90	2.66	2.46	2.28	2.13	2.00	1.99
	8 (1)	10.55	15.94	7.95	6.39	5.32	4.55	3.99	3.99	3.19	2.90	2.66	2.45	2.28	2.13	1.99	2.00
	16 (1)	10.63	10.58	7.98	6.38	5.32	4.56	3.99	3.99	3.19	2.90	2.66	2.46	2.28	2.13	2.00	2.00
	32 (1)	15.94	15.98	10.65	5.31	5.31	4.56	3.99	3.55	3.19	2.90	2.66	2.46	2.28	2.13	2.00	2.00
	64 (2)	15.94	15.98	7.97	6.39	5.32	4.56	3.99	3.98	3.19	2.90	2.66	2.46	2.28	2.13	2.00	2.00
	128 (4)	10.62	15.94	7.98	6.37	5.32	4.56	3.99	3.99	3.19	2.90	2.66	2.46	2.28	2.13	2.00	1.99
	256 (8)	10.63	6.37	4.56	3.99	2.91	2.46	2.13	2.00	1.68	1.52	1.39	1.33	1.18	1.10	1.03	1.00
	512 (16)	7.99	3.99	2.46	2.00	1.52	1.33	1.10	1.00	0.86	0.78	0.71	0.67	0.60	0.57	0.52	0.50
	1024 (32)	3.99	2.00	1.33	1.00	0.80	0.67	0.57	0.50	0.44	0.40	0.36	0.33	0.31	0.29	0.27	0.25

		blocks															
		17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
threads (warps)	1 (1)	1.77	1.68	1.60	1.52	1.45	1.39	1.33	1.33	1.23	1.18	1.14	1.10	1.06	1.03	1.00	1.00
	2 (1)	1.77	1.68	1.60	1.52	1.45	1.39	1.33	1.33	1.23	1.18	1.14	1.10	1.07	1.03	1.00	1.00
	4 (1)	1.78	1.68	1.60	1.52	1.45	1.39	1.33	1.33	1.23	1.18	1.14	1.10	1.06	1.03	1.00	1.00
	8 (1)	1.78	1.68	1.59	1.52	1.45	1.39	1.33	1.28	1.23	1.18	1.14	1.10	1.07	1.03	1.00	1.00
	16 (1)	1.78	1.68	1.60	1.52	1.45	1.39	1.33	1.33	1.23	1.18	1.14	1.10	1.06	1.03	1.00	1.00
	32 (1)	1.78	1.68	1.60	1.52	1.45	1.39	1.33	1.33	1.23	1.18	1.14	1.10	1.07	1.03	1.00	1.00
	64 (2)	1.77	1.68	1.60	1.52	1.45	1.39	1.33	1.33	1.23	1.18	1.14	1.10	1.07	1.03	1.00	1.00
	128 (4)	1.78	1.68	1.60	1.52	1.45	1.39	1.33	1.33	1.23	1.18	1.14	1.10	1.07	1.03	1.00	1.00
	256 (8)	0.91	0.86	0.82	0.80	0.74	0.71	0.68	0.67	0.63	0.60	0.58	0.57	0.54	0.52	0.51	0.50
	512 (16)	0.46	0.44	0.42	0.40	0.38	0.36	0.34	0.33	0.32	0.31	0.29	0.29	0.27	0.27	0.26	0.25
	1024 (32)	0.24	0.22	0.21	0.20	0.19	0.18	0.17	0.17	0.16	0.15	0.15	0.14	0.14	0.13	0.13	0.12

TABLE 5.1 – Résultats de l'expérimentation sur la concurrence de *threads*

La figure 5.9b met en évidence, pour ce seuil de saturation, un phénomène de rupture se produisant à partir de 4 *warps* (128 *threads*). Il s'en dégage deux tendances distinctes dans la courbe :

1. une première partie évoluant de manière constante jusque 4 *warps* puis
2. au-delà de 128 *threads*, une seconde partie dont l'évolution est fonction du nombre de *blocks* utilisés.

L'évolution globale du seuil de saturation correspond alors à la formule 5.10 où :
— **warps** correspond au découpage en groupe de *threads* de chaque *block* selon la formule 5.11 et
— **blocks** correspond au nombre de *blocks* utilisés.

$$\begin{cases} warps \times blocks = 128 & warps \geq 4 \\ blocks = 32 & warps < 4 \end{cases} \tag{5.10}$$

$$warps = \lceil \frac{threads}{32} \rceil \ tq \ threads \in [1, 1024] \tag{5.11}$$

Nous en déduisons ainsi, que :
— les SMX sont saturés à partir de 128 *threads* par *blocks* et
— les SMX sont sous-exploités en dessous de 128 *threads* par *blocks*.

Afin de maximiser le taux d'occupation des SMX, nous avons ainsi défini dans le second critère de placement, en section 3.3.2, le paramètre d'optimisation :

$$\prod_{p=0}^{T} |I_{j+p}| \geqslant 4 \times 32$$

Exécution concurrente de *kernels*

L'architecture des unités SMX de la *Quadro K2000* intègre quatre *warp schedulers*, visibles dans la figure 4.1. Ces derniers sont ainsi capables de gérer quatre *warps* en concurrence. Chaque *warp scheduler* est associé à deux *intruction dispatch units* permettant ainsi d'exécuter en concurrence deux instructions indépendantes pour un même *warp*. Chaque SMX a ainsi la capacité d'exécuter jusque 8 instructions distinctes de manière simultanée. La *Quadro K2000* intégrant deux SMX, le total est ainsi porté à 16 instructions distinctes sur cette plateforme.

Dans la figure 5.9, le parallélisme de tâches sur GPU correspond à la zone de données où $concurrency > 1.0$ et dont la couleur de représentation évolue du bleu (faible concurrence) vers le rouge (forte concurrence). Pour cette zone, nous pouvons observer que la valeur de *concurrency* évolue en fonction du nombre de *blocks* et de *threads* affectés à chaque *kernel*. Lorsque l'une de ces deux dimensions augmente, la *concurrency* diminue, ce qui implique que les *kernels* peuvent être exécutés en concurrence non seulement entre les SMs mais aussi entre les *warp schedulers*. Enfin, les résultats de *concurrency* sont plafonnés à 16 *kernels* exécutés simultanément ce qui correspond aux spécifications du *compute capability 3.0* de Nvidia. Ce maximum est cependant atteint de manière irrégulière. L'alimentation de 32 *CUDA streams* nous apparait délicate dans cette expérimentation du fait que :
— les *CUDA streams* soient alimentés à partir d'un unique *thread* sur le CPU et
— le temps d'exécution de chaque *kernel*, lorsque la valeur de *concurrency* est supérieure à 1.0, reste assez court ($10ms$).

Le temps d'exécution des *kernels* et la puissance du CPU sont donc deux critères a prendre en considération pour l'exploitation du parallélisme de tâches sur GPU.