Table des matières

Liste des tableaux

Liste des figures

Liste des acronymes

ADN	Acide désoxyribonucléique
APU	Accelerated Processing Unit
CPU	Central Processing Unit
DDR	Double Data Rate
DMA	Direct Memory Access
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HBM	High-Bandwidth Memory
HMC	Hybrid Memory Cube
hUMA	heterogeneous Unified Memory Access
ICD	Installable Client Driver
MPI	Message-Passing Interface
RAM	Random Access Memory
SMT	Simultaneous Multi-Threading
TCL	Tool Command Language

Remerciements

J'aimerais remercier mon directeur de recherche, le Dr. Paul Fortier, pour m'avoir suggéré ce palpitant sujet de recherche. Je le remercie également, avec mon co-directeur de recherche, le Dr. Benoît Gosselin, pour m'avoir supervisé et financé pendant ma maîtrise.

Je dois aussi de sincères remerciements à mon Honey Bunny qui a su m'encourager à travailler sur mon projet quand j'en avais moins envie.

Pour terminer, mes parents et mon grand-père ont aussi joué un rôle crucial en démontrant un intérêt soutenu pour mon avancement.

Introduction

Depuis quelques années, on assiste à une stagnation des fréquences des puces dans les ordinateurs [32]. Pour continuer d'augmenter la performance des puces, les fabricants se sont rabattus sur l'intégration de plusieurs processeurs dans une même puce, maintenant appelés "coeurs". La performance disponible augmente encore à peu près linéairement, sauf que les logiciels doivent évoluer pour exploiter cette nouvelle architecture.

Du même coup, le marché du matériel informatique se métamorphose. L'intérêt se déplace de la performance brute la plus élevée vers la meilleure efficacité énergétique, et ce, pour tous les domaines. Pour le commun des mortels, l'explosion de la mobilité informatique en est le plus bel exemple ; les appareils portables sont de plus en plus puissants, pour les mêmes barêmes de dissipation thermique et d'autonomie, alors que la technologie des batteries évolue plus tranquillement. Dans le monde du calcul haute performance, le mouvement est aussi présent ; on opte pour des accélérateurs de toute sorte qui ont une architecture plus adaptée à certains types de problèmes afin de diminuer la consommation électrique.

Il s'agit ici de remplacer l'utilisation des processeurs centraux qui ont jusqu'à récemment fourni la majeure partie de la capacité de calcul des appareils. Parmi ces accélérateurs, on retrouve beaucoup de processeurs graphiques (GPU), qui ont acquis ces dernières années des capacités de calcul général. Plus récemment, les circuits logiques programmables (FPGA) ont fait leur entrée, alors qu'ils étaient auparavant limités à des utilisations plus particulières comme par exemple le traitement de signal et le prototypage. En juin 2015, on y retrouvait cependant aucun de ces derniers parmi les 500 super-ordinateurs les plus puissants. 90 de ceux-ci contenaient des accélérateurs autres que des FPGA, soit des processeurs graphiques NVIDIA ou AMD, ou des processeurs hautement parallèles de PEZY Computing et d'Intel [27].

Que ce soit pour n'importe lequel type d'accélérateur, on doit retravailler les logiciels traditionnels pour en faire usage. L'application relativement récente du calcul haute performance dont il sera question dans ce mémoire est l'assemblage d'acide désoxyribonucléique (ADN). Le projet qui sera présenté consiste à réécrire un algorithme destiné à cette fin, nommé Ray, afin de le paralléliser de manière générale pour différents accélérateurs, dont les FPGA. Cet algorithme a été développé par Sébastien Boisvert, de l'Université Laval [5]. Puisque l'ou-

til de parallélisation utilisé pour le présent projet est OpenCL, le programme résultant est simplement nommé OCLRay.

L'utilisation d'accélérateurs ne se limite pas au marché des super-ordinateurs, cependant. Placés dans des ordinateurs personnels ou des stations de travail, ils peuvent fournir la puissance de calcul requise pour les problèmes qui autrement seraient difficiles ou trop longs à résoudre. Le but de ce projet est de démontrer que l'utilsation d'un FPGA comme accélérateur peut accélérer ou rendre possible l'assemblage d'ADN dans des délais raisonnables sur des appareils autre que des super-ordinateurs.

De ce fait, les contributions de ce mémoire sont les suivantes :

— Réécriture du logiciel Ray avec parallélisation par OpenCL.

— Implantation sur circuits FPGA.

— Tests de comparaison de performance FPGA et processeur central.

— Tests de comparaison de consommation énergétique FPGA et processeur central.

— Modification de l'algorithme de base de la table de hachage afin de minimiser les accès mémoire en lecture.

— Tests sur l'effet du taux d'occupation de la table de hachage modifiée.

Au chapitre 1, il sera question du séquençage et de l'assemblage en général. Les quelques algorithmes développés au fil du temps pour ce dernier seront brièvement présentés, puis l'algorithme de Ray sera étudié en détail. Ensuite les différents travaux qui ont tenté d'accélérer l'assemblage seront résumés.

Au chapitre 2, il sera question des différentes puces capables de traitement parallèle. Les processeurs centraux, les processeurs graphiques ainsi que les outils utilisés pour la compilation seront présentés. Les accélérateurs de type FPGA seront ensuite discutés. Les détails de la compilation du code OpenCL comme l'extraction de la parallélisation et l'optimisation pour ces appareils sera étudiée. De plus, certaines caractéristiques d'accélérateurs leur donnant des avantages inhérents seront introduits, soit les différences de consommation énergétique et la mémoire unifiée.

Au chapitre 3, l'algorithme de parallélisation utilisé dans Ray sera discuté. Les changements apportés à l'algorithme afin de convertir sa parallélisation d'une interface de passage de messages (MPI) à OpenCL seront décrits au même titre que ceux apportés purement par souci de performance.

Au chapitre 4, les résultats d'accélération seront présentés et les accélérateurs testés, comparés sur le plan de la performance et de la consommation énergétique. Cela se terminera finalement avec une ouverture sur de possibles améliorations et le futur du traitement parallèle.

Chapitre 1

Revue de littérature

1.1 Assemblage d'ADN

L'ADN est composé d'une séquence de quatre nucléotides qui se répètent sans cesse. Un chromosome est constitué d'une telle séquence ainsi que de son complément renversé ; ces deux séquences que l'on appelle des brins sont retenus par des liaisons hydrogène. Le complément renversé d'une chaîne de nucléotides consiste en le complément de chacun de ces nucléotides individuellement. Le tout est ensuite renversé, car l'ADN est parcouru toujours dans une direction spécifique pour des raisons biologiques, qui est inversée pour le brin opposé. Cette direction va de $5'$ à $3'$, soit les noms donnés aux extrémités ; $3'$ désigne l'extrémité terminée par groupement hydroxyle OH, alors que $5'$ désigne l'extrémité terminée par un phosphate. Le tout est illustré à la figure 1.1.

L'assemblage d'ADN découle du séquençage de ce dernier, qui donne comme résultat des lectures de longeur plus ou moins importante, qu'on doit assembler pour obtenir la chaîne de nucléotides complète. Il existe plusieurs méthodes qui parviennent à ce but, séparées en deux types. Il y a l'assemblage dit "de novo" et l'assemblage dit "mapping". Le premier reconstruit le génôme sans connaissance à priori, tandis que le deuxième requiert un génome déjà assemblé

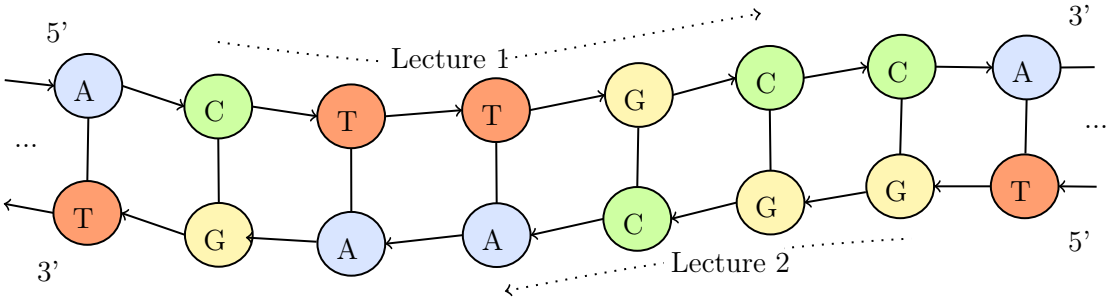

FIGURE 1.1 – Représentation d'une partie d'un chromosome, soit de l'ADN composé des quatres nucléotides.

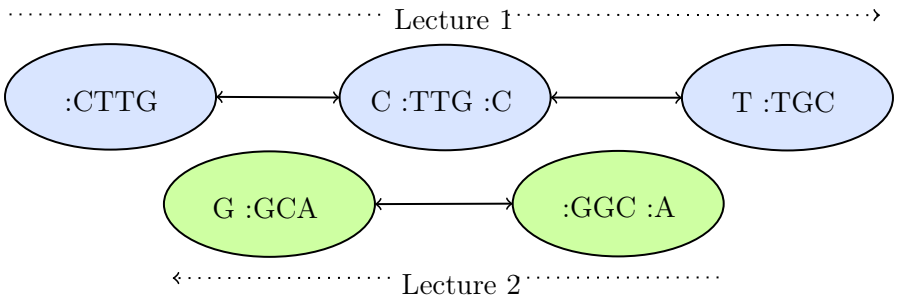

FIGURE 1.2 – Représentation d'un graphe de De Bruijn ayant cinq sommets.

afin de comparer à mesure qu'il construit son propre résultat. Évidemment, un assemblage "de novo" est plus complexe puisqu'il consiste en plus qu'une simple question d'alignement.

L'assemblage d'ADN a été complété avec différents algorithmes au fil du temps. Le premier d'entre eux est appelé le "greedy". Il consiste à agrandir les lectures itérativement, simplement en choisissant à chaque fois le plus grand chevauchement [20]. Par la suite, un autre appelé "Overlap, Layout and Consensus" a été inventé. Comme son nom l'indique, il utilise aussi les recouvrements dans les lectures afin d'assembler le génôme [15], mais calcule aussi l'ordre et l'orientation de celles-ci. Cependant, il est moins adapté à l'assemblage de lectures courtes, de par le critère minimal de chevauchement.

Une méthode plus récente dont il sera question dans ce mémoire consiste à utiliser un graphe de de Bruijn dans lequel on place des fragments de l'ADN. Un graphe de de Bruijn est un graphe orienté qui permet de représenter les chevauchements de longueur k-1 entre tous les mots de longueur k, appelés k-mers, sur un alphabet donné [8]. Cela résulte en une représentation compacte des lectures, car chaque k-mer présent dans celles-ci n'est stocké qu'une seule fois. Le nombre de fois que chacun a été aperçu est enregistré, ce qu'on appelle la couverture. Par exemple, pour une longueur de k-mer de 3 nucléotides, les lectures représentées à la figure 1.1 résulteront en cinq sommets, tous avec une couverture de 1. Ce graphe de De Bruijn est représenté à la figure 1.2. Il est finalement possible de chercher dans le graphe des chemins qu'on appelle des contigs représentant la séquence génomique orginale ou une partie de celle-ci.

1.2 Algorithme d'assemblage Ray

Les auteurs de Ray ont vu l'opportunité de designer un nouvel assembleur dans le fait qu'aucune méthode n'avait été proposée pour réaliser un assemblage de haute qualité à partir d'un mélange des résultats de séquençage de différentes technologies, sous forme de lectures courtes. C'est ce qu'ils ont accompli, et de par la nature ouverte de Ray, ils espèrent qu'il serve comme base pour d'autres développements [4].

Donc, étant donné une banque de lectures qui peuvent contenir des erreurs, être potentielle-

ment en paires et de différentes technologies, le problème d'assemblage consiste à construire des contigs de façon à ce que :

1. la proportion du génôme représenté soit maximale ;
2. le nombre d'erreurs d'assemblage soit minimal ;
3. le nombre de contigs soit minimal.

L'algorithme Ray se divise en plusieurs étapes réalisées séquentiellement. Ici, chacune d'entre elles sera décrite.

1.2.1 Construction du graphe

Comme mentionné dans l'introduction, Ray utilise un graphe de de Bruijn pour assembler l'ADN. Tout commence d'abord par la construction du graphe. Chaque lecture est découpée en k-mers, donc une séquence de nucléotides de longueur fixe, décidée par l'utilisateur lors du lancement du programme. Donc dans le graphe, un sommet est créé pour chaque k-mer. La longueur minimale donnant de bons résultats, déterminée par des observations expérimentales [10], est 19. Par défaut, puisque les k-mers doivent être significativement plus petits que les lectures elles-mêmes, Ray utilise une longueur de 21, pour les technologies de séquençage courantes. Il est cependant possible d'imposer une longueur différente, selon les lectures et le type de génôme qu'on souhaite assembler. La longueur des k-mers est imposée impaire, parce que, de par le fait que les nucléotides se rattachent avec leur complément renversé, cela permet de stocker les valeurs d'un brin ou de l'autre de l'ADN dans un seul sommet. Chaque k-mer de longueur k chevauche ses voisins sur $k-1$ nucléotides, donc les arrêtes entre les sommets indiquent la continuation de la lecture.

L'opération est répétée pour chaque lecture. Lorsqu'un k-mer est rencontré pour une deuxième fois ou plus, sa couverture est incrémentée ; il s'agit du nombre de fois que le dit k-mer a été croisé dans le lot de lectures.

1.2.2 Élimination des couvertures faibles

Ray, pour déterminer si les k-mers rencontrés sont fiables, se fie sur le nombre total de fois qu'ils ont été lus lors du séquençage ; ce nombre est appelé *couverture*. Dans l'algorithme, une couverture minimale du génôme est requise pour minimiser les erreurs et éliminer les régions répétées. En effet, il considère d'abord que tout k-mer rencontré qu'une seule fois dans le lot de lectures est une erreur.

Afin d'éliminer ces k-mers avec une couverture de 1, deux méthodes sont utilisées successivement. Premièrement, un filtre de Bloom est utilisé. Un filtre de Bloom est une structure compacte dans laquelle on peut insérer des éléments et en refaire la requête. Cependant, il y a une certaine probabilité de faux positifs, c'est-à-dire que le filtre peut divulguer la présence d'un élément alors qu'il n'a pas été inséré. Les éléments ne peuvent pas être retirés cependant,

mais dans l'algorithme ici, cette fonctionnalité n'est pas nécessaire. Concrètement, cette structure consiste en une série de bits dont certains sont mis à "1" lorsqu'un élément est inséré. La position des bits mis à "1" est déterminée selon des fonctions de hachage. Lors de la requête d'un élément, il s'agit de vérifier si tous ses bits sont à "1". Si l'un ou plusieurs d'entre eux est à "0", c'est que l'élément n'est pas présent dans le filtre. Soit le nombre de lectures N_L, la longueur des k-mers L_k, le nombre de brins N_B et le nombre de directions N_D. Ray détermine la taille optimale du filtre de Bloom, T_B, soit la quantité de bits, par :

$$T_B = N_L \cdot L_k \cdot N_B \cdot N_D = 4 \cdot N_L \cdot L_k \tag{1.1}$$

Pour les fonctions de hachage, Ray utilise une technique qui donne un résultat uniforme sur 64 bits [29]. Ensuite, un ou-exclusif est effectué avec plusieurs nombres de la même taille, donnant les positions des bits à vérifier pour le k-mer en question. Ces nombres sont générés en conservant les 64 premiers bits de la somme md5 de la date [1].

Chaque k-mer rencontré est placé dans le filtre, puis c'est à sa deuxième rencontre seulement qu'il est inséré dans le graphe de de Bruijn. Par conséquent, tous les sommets rencontrés une seule fois dans les lectures ne s'y retrouvent pas, et la taille du graphe nécessaire est amoindrie. Puisqu'un filtre de Bloom contribue à une perte d'information, un effet latéral est que les arcs décelés lors de la première rencontre sont perdus.

Deuxièmement, lorsque le graphe a été construit, Ray procède à une purge de celui-ci, afin d'éliminer les quelques k-mers qui auraient passé à travers le filtre de Bloom, par exemple grâce à des faux positifs. Lorsqu'un k-mer est présent une seule fois dans le graphe, après avoir passé par le filtre de Bloom, il est éliminé, ainsi que les arcs menant à d'autres k-mers. Si un k-mer se retrouve sans arc, il est aussi éliminé.

1.2.3 Annotation des lectures

Lorsque tous les k-mers ont été extraits et que le graphe a été construit, Ray viendra annoter les lectures dans le graphe, car il se réfèrera à elles pour faire l'assemblage. Cette opération consiste à placer sur un seul sommet une référence vers la lecture en question. Il a été déterminé que les lectures sont plus utiles lorsqu'annotées sur un sommet unique, ce qui signifie qu'il ne fait pas partie d'une section du génôme répétée, et qu'il n'est pas erroné. Par exemple, si nous avons la lecture GCAAATAT et que le premier sommet unique est AAA, l'annotation de cette lecture sera faite sur ce dernier. Ceci permet de conserver l'information contenue dans chaque lecture, ce qui est en contraste avec les assembleurs précédents comme Velvet, où les graphes de de Bruijn sont simplifiés et l'information des lectures perdue [31].

1. Fonction de hachage 128 bits populaire pour la signature de fichiers, décrite dans le RFC 1321.

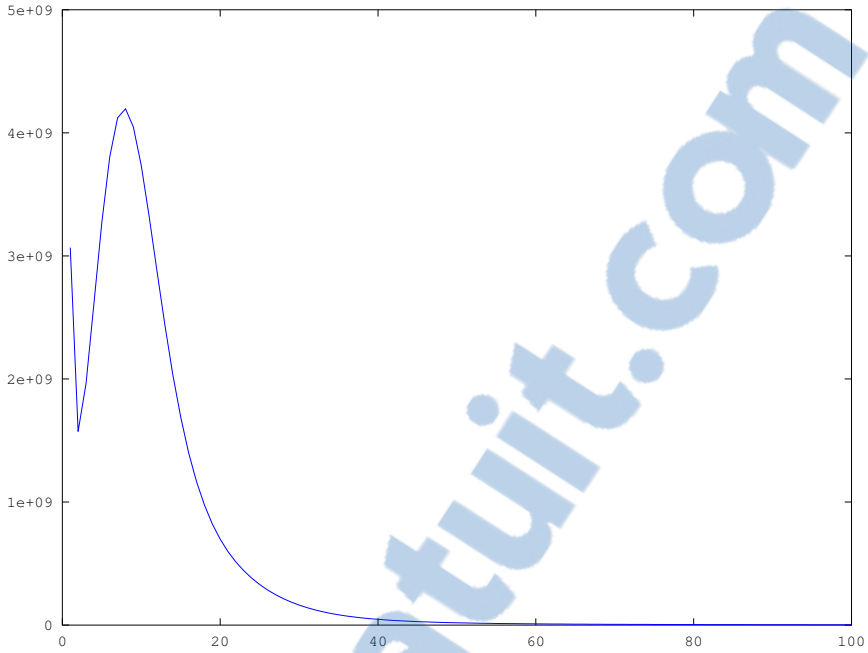

FIGURE 1.3 – Distribution de la couverture pour l'assemblage du génome de l'épinette blanche[1].

C'est la distribution de la couverture qui permet de classifier les sommets. Il s'agit du nombre de sommets qui ont une couverture donnée. Tel que spécifié par [7], la distribution de la couverture se modélise par la somme d'une exponentielle et d'une distribution de Poisson. La première représente les erreurs générées lors du séquençage, alors que la deuxième représente les k-mers véridiques, qui sont choisis parmi toutes les sous-séquences de la longeur donnée avec une probabilité à peu près uniforme. Un tel exemple est affiché à la figure 1.3. Il peut être possible de distinguer plusieurs maximums locaux dans une telle distribution, mais les sommets uniques doivent faire partie de la première crête. On choisit donc une valeur limite qui correspond à deux fois la couverture maximale de la première crête. Au delà de ça, le sommet représente fort probablement un k-mer répété. En dessous, c'est un sommet erroné.

La lecture est donc annotée sur le sommet représentant le premier k-mer unique de celle-ci. Il peut aussi y avoir plusieurs lectures annotées sur le même sommet. Quant aux lectures en paires, les annotation se font sur le même principe, mais l'information de la paire est conservée. La différence se situe au niveau des heuristiques, qui seront détaillés en 1.2.5.

1.2.4 Détermination des semences

La première étape de l'assemblage est de déterminer les semences, soit une série de nucléotides hautement fiable à partir de laquelle la solution proliférera. Dans Velvet, on requiert que l'uti-

1. Données générées par Sébastien Boisvert lors d'une exécution de Ray.

couverture	confiance (m)
$[2, 19]$	3
$[20, 24]$	2
$[25, \infty[$	1.3

Tableau 1.1 – Détermination de la confiance, m, d'un sommet, selon sa couverture.

lisateur indique manuellement ces semences, alors que Ray, lui, les calcule automatiquement. C'est à partir de la distribution de la couverture qu'elles sont déterminées. Cette distribution représente le nombre de k-mers qui ont une couverture donnée. Pour déterminer les semences, on considère le graphe de de Bruijn dont tous les k-mers ont une couverture supérieure à la valeur minimale, dénotée c_{min}. Celle-ci représente le point dans la distribution où le nombre d'erreurs devient plus petit que le nombre de vraies séquences. Du même coup, la dérivée de la distribution est nulle à cet endroit. La valeur maximale, dénotée c_{max}, représente le point dans la distribution correspondant à la moyenne de la couverture. À cet endroit, on retrouve aussi une inversion du signe de la dérivée de la distribution de couverture. Bref, la couverture c des k-mers potentiels pour la formation d'une semence doit respecter les bornes spécifiées par :

$$\frac{c_{min} + c_{max}}{2} \leq c \leq 2 * c_{max} \tag{1.2}$$

Finalement, tous les sommets composés de ces k-mers doivent avoir au maximum un arc entrant et un arc sortant. Le sous-graphe de de Bruijn formé par ces sommets contient évidemment beaucoup moins d'information, mais il ne sert que pour la détermination des semences, suite à quoi le graphe original est utilisé.

1.2.5 Heuristiques d'assemblage

Ce qui rend Ray particulier par rapport aux autres travaux est qu'il ne tente pas de faire du graphe un graphe eulérien pour déterminer la solution. Une telle méthode consiste à déterminer un ou des chemins bien définis pour lesquels il est possible d'identifier une extrémité de début et de fin. Il utilise simplement trois inégalités pour déterminer l'allongement d'un chemin en cours d'extension. Dans celles-ci, Ray introduit le concept de confiance, m, qui est simplement un facteur spécifié selon la couverture du k-mer concerné. Il est inversement proportionel à la couverture, tel que présenté au tableau 1.1. Ces valeurs ont été déterminées empiriquement par les créateurs de Ray.

Un autre concept est la fonction de décalage. Celle-ci représente la sécurité [2] d'un chemin $c+y$ donné dans le graphe. Plus précisément, la fonction de décalage pour un sommet donné est le nombre de lectures annotées, $nb(A_{x_i})$, qui spécifient que le nombre de nucléotides entre ce

2. Par sécurité, on entend à quel point le chemin risque d'être réel.

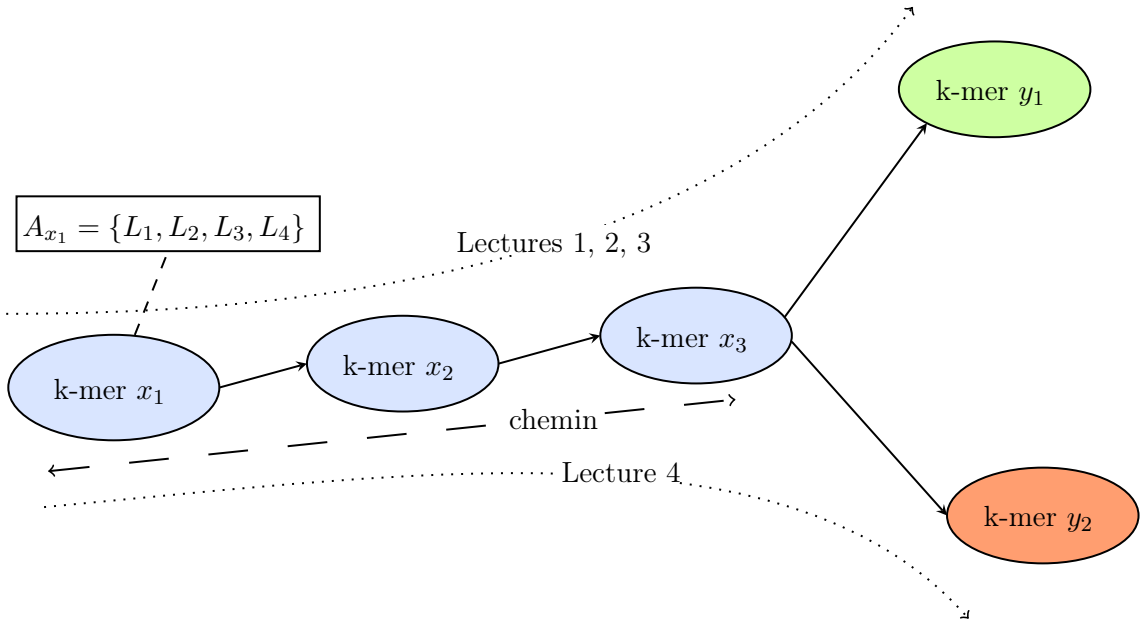

FIGURE 1.4 – Représentation de la fonction de décalage.

sommet et un autre, représenté par la fonction $\text{dist}_L(x_i, y)$, est la même que pour le chemin évalué.

$$D_i(c, y) = nb(A_{x_i}) \text{ tel que } \text{dist}_L(x_i, y) = \text{dist}_{c+y}(x_i, y), \tag{1.3}$$

où c est le chemin évalué dans le graphe, y est le sommet évalué, x_i est le sommet i dans le chemin et A_{x_i} est l'annotation d'une lecture sur ce dernier.

Par exemple, dans la figure 1.4, les lectures L_1, L_2 et L_3 parcourent le chemin $c{+}y_1$, tandis que seulement la lecture L_2 a été répertoriée pour $c{+}y_2$. Ici, $\text{dist}_{L_1}(x_1, y_1) = 3$ et $\text{dist}_{L_4}(x_1, y_2) = 3$. Alors, $D_1(c, y_1) = 3$ et $D_1(c, y_2) = 1$.

Pour des lectures en paires, soit L_1 et L_2, le même concept s'applique, sauf qu'on utilise une méta-lecture, L_M, pour le calcul du chevauchement. Cette méta-lecture est composée de L_1, ensuite d'un nombre d de nucleotides correspondant à la distance entre les deux lectures, puis du complément renversé de L_2. De plus, on ne requiert pas que $\text{dist}_{L_M}(x_i, y)$ soit égale, mais plutôt, on requiert que sa différence soit inférieure ou égale à l'écart-type des distances en paires, $\sigma_{\text{dist}_{L_M}}$, donc des caractéristiques du séquençage.

$$Dp_i(c, y) = nb(A_{x_i}) \text{ tel que } |\text{dist}_{L_M}(x_i, y) - \text{dist}_{c+y}(x_i, y)| < \sigma_{\text{dist}_{L_M}} \tag{1.4}$$

Finalement, des heuristiques permettent de choisir le sommet le plus adéquat pour l'allongement d'un chemin, ici dénoté c et sa longueur l, alors que les sommets présentant le dilemme

sont y_1 et y_2. Pour que le premier l'emporte, il s'agit que les trois inégalités suivantes soient respectées :

$$m \cdot \sum_{i=1}^{l}(l-i) \cdot Dp_i(c, y_2) < \sum_{i=1}^{l}(l-i) \cdot Dp_i(c, y_1) \quad (1.5)$$

$$m \cdot \sum_{i=1}^{l} Dp_i(c, y_2) < \sum_{i=1}^{l} Dp_i(c, y_1) \quad (1.6)$$

$$m \cdot \min_{i \epsilon 1..l}(Dp_i(c, y_2)) < \min_{i \epsilon 1..l}(Dp_i(c, y_1)), \quad (1.7)$$

où m est la confiance.

Pareillement, un sommet peut remporter face à un autre en utilisant les fonctions de chevauchement pour les lectures singulières, même si elles sont fournies en tant que paires. On accorde cependant la priorité aux fonctions de chevauchement pour les paires.

1.2.6 Fusion des chemins

La dernière étape consiste à fusionner les chemins déterminés à l'étape précédente. Cela est nécessaire car ils sont stockés séparément, puis des semences peuvent avoir fait débuter deux extensions qui en réalité sont contiguës. Ray exige un chevauchement minimal entre les chemins de 2000 paires de base, qui pourrait ne pas être respecté si par exemple les deux chemins sont différents. Cela pourrait être expliqué par le fait qu'ils n'ont pas été construits avec les mêmes lectures.

En plus des étapes coeurs de l'algorithme tout juste décrites, plusieurs s'ajoutent de par le partitionnement du problème sur plusieurs noeuds de calcul, comme il sera décrit plus en détails à la section 3.1. La figure 1.5 représente l'algorithme dans son ensemble ainsi que les différentes tâches de chaque étape. Ces tâches sont différentes selon le fait qu'un noeud soit maître ou esclave. Le premier rôle consiste à commander les tâches et parfois rassembler les données alors que le deuxième rôle consiste à exécuter les tâches.

1.3 Accélération matérielle

1.3.1 OpenCL

Dans le présent mémoire, il sera question de la spécification OpenCL 1.2, publiée en novembre 2011 [28]. OpenCL est un standard ouvert et sans royauté permettant une programmation haute performance en exploitant le parallélisme de l'architecture matérielle. Il est utilisé tant sur les super-ordinateurs que les systèmes intégrés et les appareils mobiles. Le standard divulgue l'interface à exposer aux utilisateurs, mais non l'implémentation ; chaque fabricant est

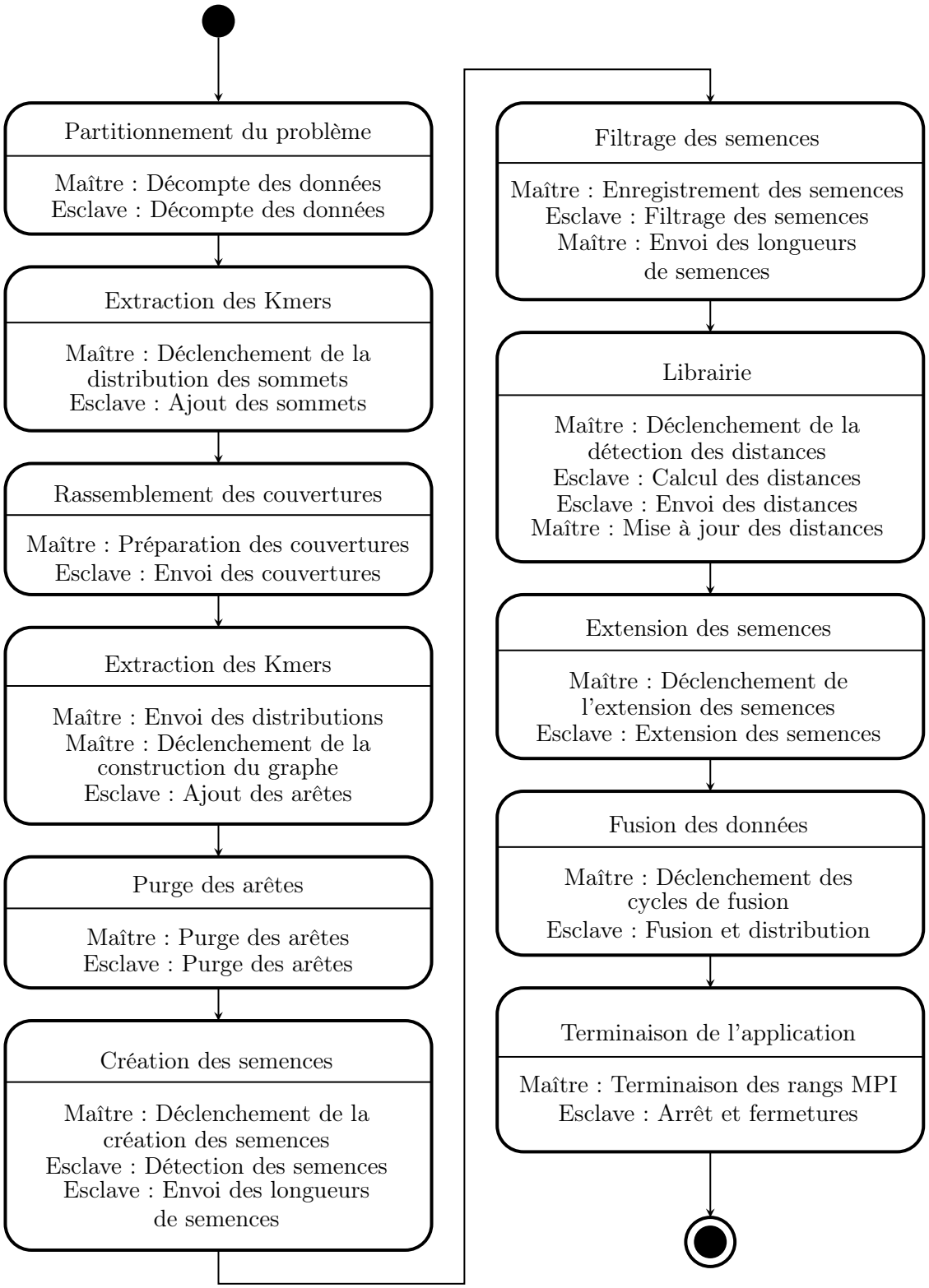

FIGURE 1.5 – Graphe de fluence de l'algorithme complet avec le partitionnement.

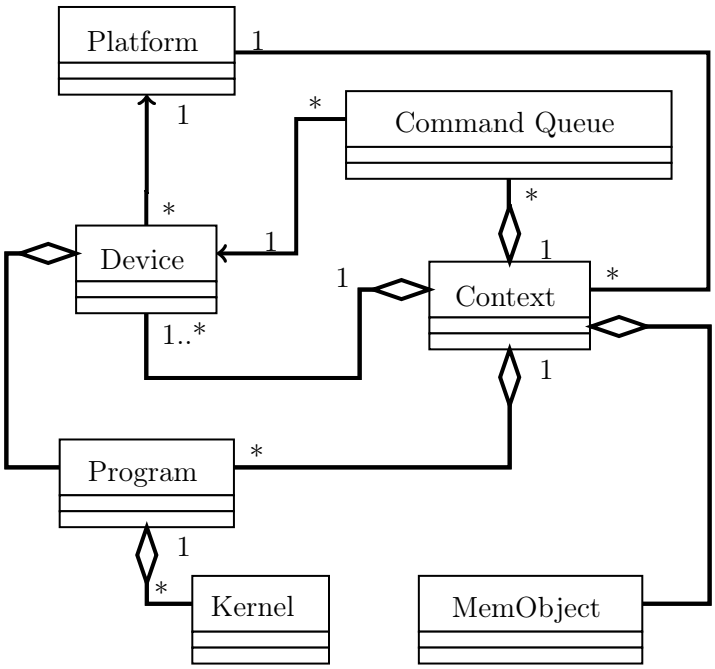

FIGURE 1.6 – Diagramme de classe de la spécification OpenCL.

libre à ce niveau. Il est donc facile de cibler, avec un même code, plusieurs architectures différentes. Le diagramme de classes de la spécification est affiché à la figure 1.6. Les différents types d'accélérateurs (Device) supportés seront présentés au chapitre 2, alors que certains objets pertinents pour la compréhension du contenu du mémoire (Kernel, MemObject) seront présentés dans cette même sous-section.

Concrètement, le programmeur OpenCL utilise un langage de programmation basé sur le C99 ou le C11 pour la version plus récente. Le parallélisme est décrit explicitement par ce dernier, selon une structure de groupes de travail (workgroups) et d'éléments de travail (work-items), assignés aux ressources physiques que sont les unités de calcul (compute unit) et les éléments de traitement (processing element). Chaque tâche doit, à la base, être découpée en petites étapes similaires et parallèles.

Plateforme OpenCL

La sélection de la plateforme OpenCL peut se faire de deux façons. Celle universelle consiste à utiliser des librairies faisant partie de la trousse de développement logiciel propre au fabricant lors de l'édition de liens du programme hôte. C'est donc à ce moment que la plateforme est sélectionnée, puis elle ne peut être changée lors de l'exécution. L'autre méthode, plus adaptée au déploiement commercial, consiste à utiliser une librairie universelle lors de l'édition de liens. Alors, à l'exécution, le système d'exploitation pourra informer le programme hôte des plateformes disponibles grâce à un pilote client installable (ICD) et une ou plusieurs seront

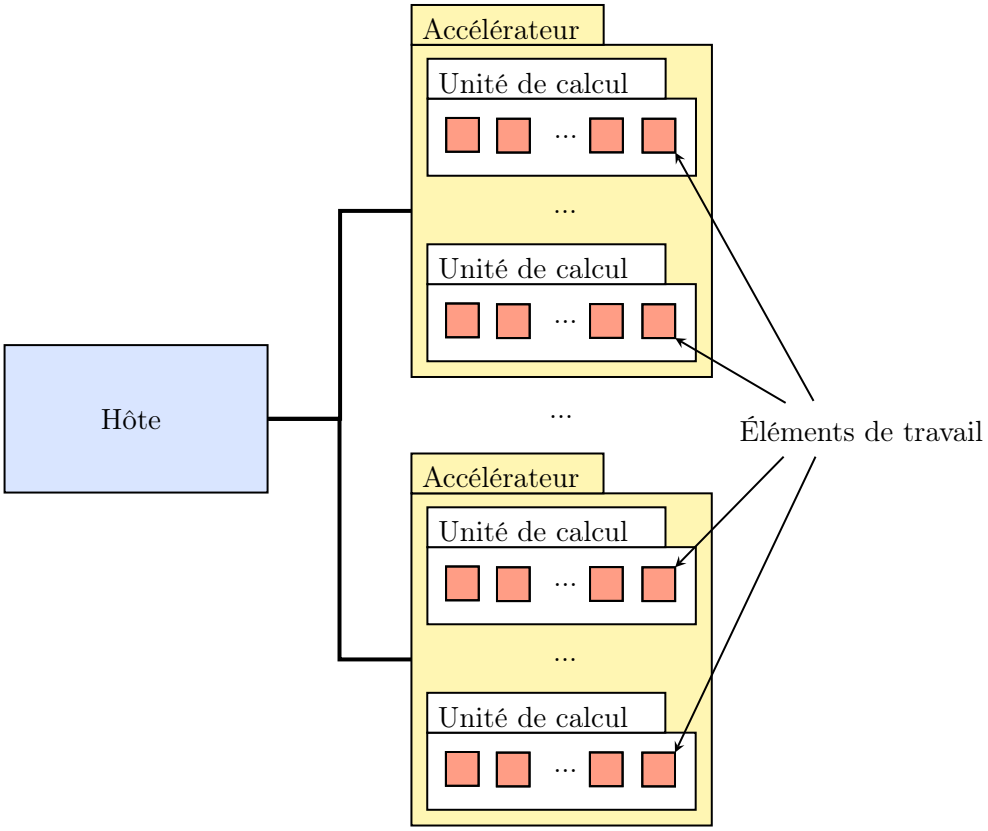

FIGURE 1.7 – Plateforme d'exécution OpenCL.

sélectionnées. On retrouve typiquement une plateforme par fabricant d'appareils OpenCL, donc c'est pourquoi en interrogeant un système avec plusieurs appareils de différents vendeurs, plusieurs plateformes peuvent être disponibles.

Architecturalement parlant, un système compatible OpenCL comprend en premier l'hôte, soit un processeur central. À celui-ci vient se connecter la ou les plateformes d'exécution OpenCL, via un bus de communication tel PCI-E [3] pour celles d'entres elles qui sont dédiées. Chaque plateforme d'exécution peut avoir une ou des unités de calcul, qui traiteront chacun plusieurs éléments. Une unité de calcul consiste en une architecture qui peut traiter un ou des fils d'exécution de manière indépendente et simultanément aux autres unités de calcul. Par exemple, un coeur de processeur est une unité de calcul, mais l'utilisation d'une technologie de traitement multiple simultané (SMT), par exemple le Hyper-Threading d'Intel, ne transforme pas ce coeur en deux unités de calcul. Finalement, chaque élément consiste en une série d'instructions à exécuter. Le tout est illustré à la figure 1.7.

3. Bus local série communément utilisé pour les périphériques.

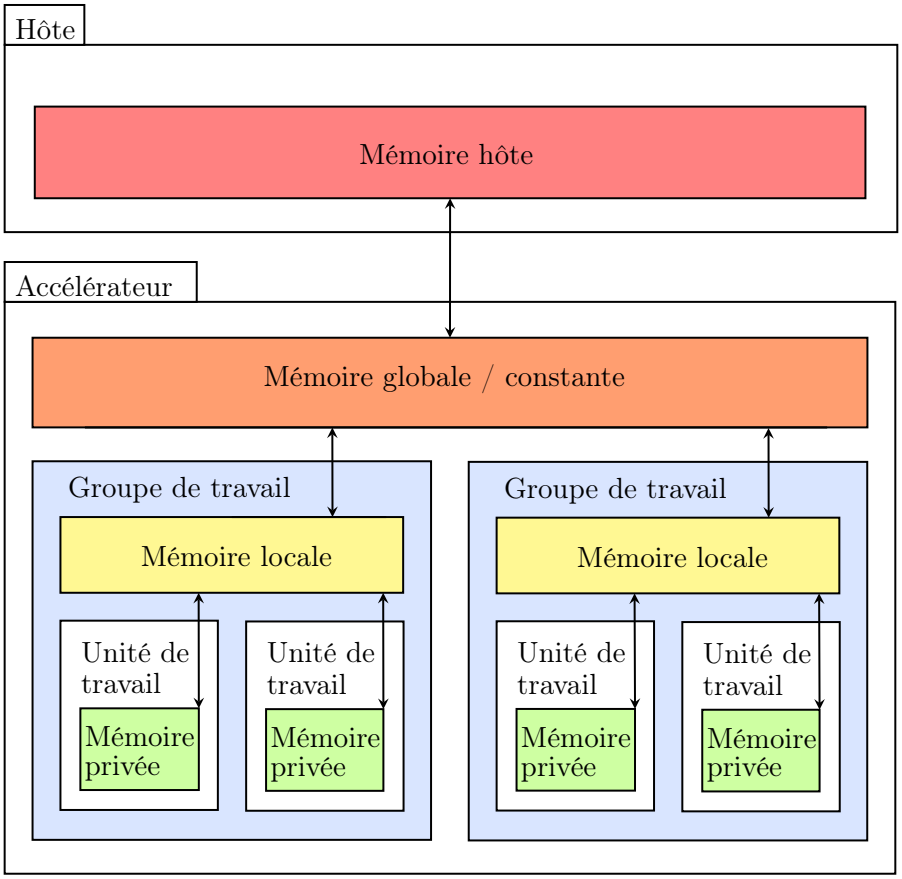

FIGURE 1.8 – Modèle de mémoire OpenCL.

Modèle de mémoire

Un modèle de mémoire fait partie du standard et est adopté par toutes les implémentations. Dans celui-ci, on retrouve d'abord la mémoire hôte, soit la mémoire principale, accessible par le processeur central, laquelle est utilisée pour l'exécution du programme hôte. Ensuite, du côté de la plateforme d'exécution, il y a la mémoire globale. Celle-ci est accessible à tous moments, de n'importe où. Puis, il y a la mémoire locale. Celle-ci est commune à tous les éléments de travail dans un même groupe de travail. Puis, une mémoire privée à chacune de ces unités est aussi présente. Finalement, il y a aussi une mémoire constante pour les données qualifiées comme tel. Le tout est illustré à la figure 1.8. C'est au manufacturier de l'appareil conforme OpenCL d'implémenter chaque mémoire à sa guise, mais typiquement, en terme de bande passante, la mémoire globale est la plus lente, alors que la mémoire privée est la plus rapide. La latence et la capacité sont inversement proportionelles.

Flux de travail

Le flux de travail OpenCL consiste en trois principales étapes. D'abord, des tampons de grosseur appropriée sont alloués et remplis dans la mémoire globale de l'appareil. Puis, l'exécution

du code OpenCL, appelé kernel, a lieu. Finalement, les données résultantes sont récupérées. Dans le cas d'un appareil ayant une mémoire globale distincte de la mémoire hôte, les données doivent être copiées ; c'est l'hôte qui commande un transfert de sa mémoire centrale à celle de l'accélérateur, souvent en utilisant un accès direct à la mémoire (DMA) pour alléger ce lourd travail. Dans le cas d'un accélérateur ayant une mémoire unifiée avec celle de l'hôte, les tampons originaux peuvent être utilisés tels quels. Le délai d'attente après le transfert des données n'a pas lieu. C'est le cas, par exemple, lors de l'exécution OpenCL sur un processeur central, qui peut aussi jouer le rôle d'accélérateur en même temps qu'hôte. Plus de détails seront donnés au chapitre 2.

Types de kernels

OpenCL permet deux types de kernels fondamentalement différents. Ils consistent en une tâche, ou une plage d'exécution de dimension N. Le premier type exécute la série d'instructions fournies de manière séquentielle et une seule fois, tandis que le deuxième l'exécute pour chaque élément de la plage d'exécution, pour chaque dimension. Dans ce cas, un groupe de travail peut occuper une plage non unitaire sur une ou plusieurs dimensions.

1.3.2 Accélérations précédentes

Les possibilités d'accélération pour des algorithmes en particulier ont déjà été prouvées à maintes reprises. Dans le domaine de la bioinformatique, l'algorithme de Smith-Waterman a fait l'objet de plusieurs recherches, utilisant notamment les unités de traitement vectoriel des microprocesseurs récents ainsi que les accélérateurs de type GPU et FPGA. Cet algorithme est utilisé pour l'alignement de séquences biologiques, testant des segments de toutes les longueurs possibles. Les résultats obtenus pour une implémentation OpenCL sur FPGA laissent entendre une amélioration de la performance absolue d'un facteur de 9.9 et de l'efficacité énergétique d'un facteur de 26 par rapport à l'implémentation sur microprocesseur [24].

Un autre aligneur de lectures accéléré appelé CUSHAW, cette fois-ci utilisant la transformation de Burrows-Wheeler [6], a été réalisé sur processeur graphique à l'aide de CUDA, un équivalent propriétaire à OpenCL. La transformation en question est une permutation réversible d'une chaîne de caractères, la rendant plus apte à être compressée. Cette fois, la performance se retrouve aussi améliorée face aux logiciels exploitant un processeur central multi-coeurs ; pour des lectures en paires, on rapporte un temps d'exécution plus rapide d'un facteur allant de 1.65 jusqu'à 10.4, dépendamment du lot de données. Pour des lectures simples, le facteur d'accélération va de 3.4 jusqu'à 24.3. Les meilleurs résultats sont obtenus lorsque les lectures sont plus courtes, alors que les pires sont pour les lectures les plus longues [19].

Des GPU ont aussi été utilisés pour l'alignement haut-débit de lectures en paires sur une seule séquence de référence, sous le programme MUMmerGPU. Une accélération de dix fois par rapport à la version séquentielle sur processeur central serait obtenue, ainsi qu'un temps total

d'exécution 3.5 fois plus court pour les projets de séquençage récents avec lectures courtes à moyennes, soit les technologies 454, Illumina et Sanger. Pour ce faire, le langage de programmation CUDA a été utilisé, pour les NVIDIA GeForce 8800GTX [23].

Bien sûr, il y a aussi Ray. Comme discuté à la section 1.2, il s'agit cette fois d'un assemblage de novo basé sur les graphes de de Bruijn. ABySS utilise aussi un graphe du même type qu'il fragmente sur plusieurs noeuds via une interface de passage de messages [26]. SOAPdenovo a également parallélisé la méthode avec graphe de de Bruijn en utilisant plusieurs fils d'exécution [17]. Ray assemble le génôme humain en environ 1300 heures·processeurs.

Un dernier ouvrage notable consiste en MEGAHIT. Ce logiciel fait la construction d'un graphe de de Bruijn succint sur GPU [16]. Ce type de graphe constitue une méthode bien particulière de stockage qui utilise la transformation de Burrows-Wheeler pour n'utiliser que $O(a)$ bits, a étant le nombre d'arcs. MEGAHIT utilise l'algorithme CX1 [18] qui permet la parallélisation nécessaire à l'exécution sur GPU. Au final, pour l'assemblage méta-génomique de sol, il n'utilise que 44% du temps requis comparativement à une implémentation sur processeur central seulement.

Chapitre 2

Types d'accélérateurs compatibles

2.1 Processeurs classiques ou centraux

Le processeur classique est utilisé en tant qu'hôte dans le système compatible OpenCL. Cependant, il peut en même temps jouer le rôle d'accélérateur. Tel qu'expliqué en 1.3.1, chaque coeur est alors une unité de calcul. La correspondance entre l'architecture logique OpenCL et l'architecture réelle est directe, comme par exemple pour la puce présentée à la figure 2.1, dans laquelle on retrouve huit unités de calcul. Parmi celles-ci, il est possible d'utiliser le traitement vectoriel pour traiter plusieurs éléments en parallèle à l'aide de jeux d'instruction tels AVX-256.

La compilation du code OpenCL pour le processeur central peut se faire pendant l'exécution du programme hôte sans délai perceptible par l'utilisateur. De plus, le code compilé peut être déverminé en utilisant les mêmes outils que le programme hôte, comme par exemple le GNU Debugger (GDB) [14]. Pour ces raisons, les kernels OpenCL sont souvent testés sur un processeur central d'abord. Cela est exactement ce qu'Altera suggère de faire lors du prototypage, en utilisant l'option en ligne de commande *-march=emulator* au lieu de compiler pour ses propres accélérateurs. [2]

2.2 Processeurs graphiques

Les processeurs graphiques sont aujourd'hui des bêtes de calcul à usage général. Une multitude d'applications commerciales sont accélérées avec ces appareils, puisque tout ordinateur en possède un. La correspondance entre l'architecture logique OpenCL et matérielle est similaire d'un fabricant à un autre, quoique un peu moins évidente que pour un processeur central. Par exemple, le GPU NVIDIA représenté à la figure 2.2 contient deux unités de calcul, appellés Streaming Multiprocessor (SMX). Chacune de ces unités de calcul peut effectuer une opération arithmétique ou logique sur une quantité fixe d'éléments, en parallèle, à l'aide de ses 192 Streaming Processors (SP). D'autres sections ont aussi un rôle crucial, comme par exemple les

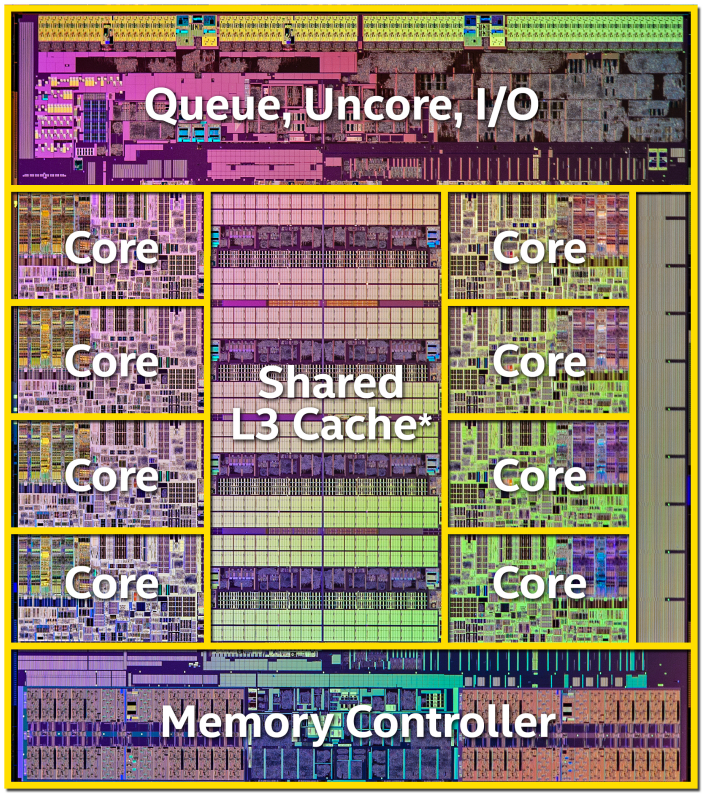

FIGURE 2.1 – Organisation logique de la puce au coeur du processeur Intel Haswell-E Core i7-5960X[1].

unités de textures qui font office de mémoire constante. C'est donc une architecture de type single-instruction, multiple-data (SIMD). Les processeurs graphiques voient leur performance très dégradée lors d'un branchement dans le code, ce qui fait que les instructions pour chaque élément à traiter sont différentes. Puis, comme les autres accélérateurs, plusieurs groupes de travail peuvent être exécutés en parallèle sur la même unité de calcul.

Bien sûr, une vaste gamme de niveaux de performance existe, allant de processeurs graphiques intégrés au processeur central jusqu'aux cartes dédiées consommant plusieurs centaines de watts. La différence principale entre les différentes gammes de produits est la quantité de mémoire à bord, son type et le nombre d'unités de calcul contenues dans la puce. Le GPU présenté n'en avait que deux, mais d'autres puces ayant la même architecture en ont jusqu'à 15 dans une seule puce.

1. Diagramme en provenance de http://www.anandtech.com/show/8426/the-intel-haswell-e-cpu-review-core-i7-5960x-i7-5930k-i7-5820k-tested.

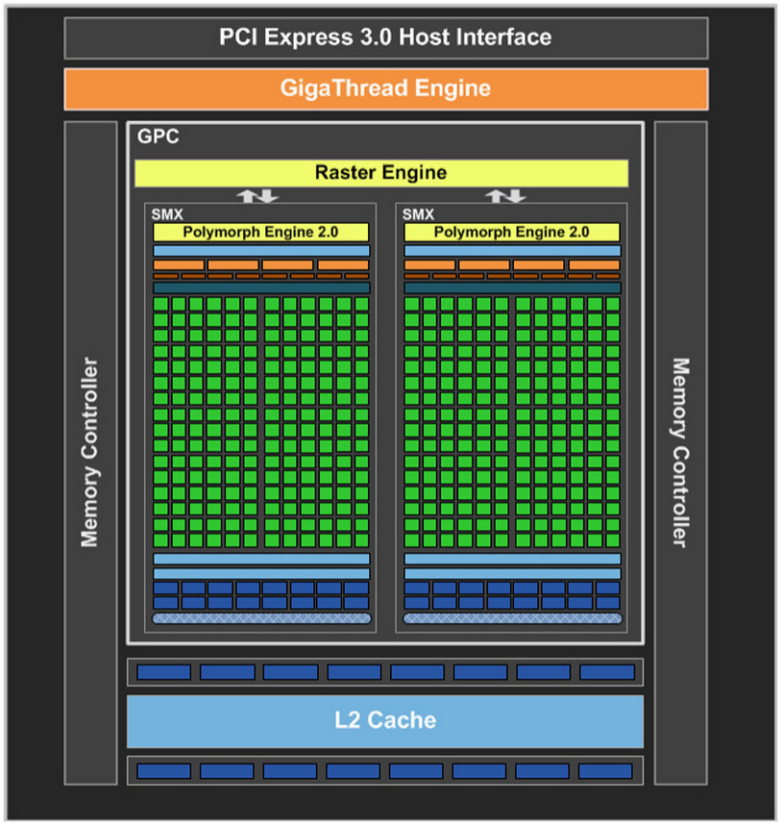

FIGURE 2.2 – Architecture de la puce GK107B au coeur du GPU NVIDIA GTX 650[1].

2.3 Circuits logiques programmables

Les circuits logiques programmables sont de nouveaux venus dans la liste des appareils compatibles OpenCL. Autrement, leur programmation requiert un langage de description matérielle (HDL) tel que le VHDL ou le Verilog. Ceux-ci requièrent à l'utilisateur d'avoir une certaine expertise en architecture matérielle pour pouvoir générer des circuits traitant le problème de façon optimale. En OpenCL, c'est le compilateur et l'optimiseur qui s'occupent de générer une architecture adéquate selon les instructions à exécuter, indiquées en C. Le programmeur n'a qu'à respecter un certain nombre de règles, sans quoi le compilateur et l'optimiseur se voient imposer des contraintes. Par exemple, il est dit que le traitement de données en point flottant ainsi que la division et le modulo d'entiers sont très coûteux en terme de logique utilisée dans le FPGA. L'arithmétique sur des pointeurs ainsi que les fonctions atomiques sont également à éviter [1]. Il y a aussi quelques bonnes pratiques à respecter pour optimiser les accès à la mémoire. Pour le reste, le SDK OpenCL promet des performances souvent meilleures qu'une architecture codée en HDL, un temps de développement moindre et aussi un code portable qui pourra être migré à de plus récents FPGA automatiquement [3].

1. Diagramme en provenance de http://www.bit-tech.net/hardware/2012/09/13/nvidia-geforce-gtx-660-2gb-review/3.

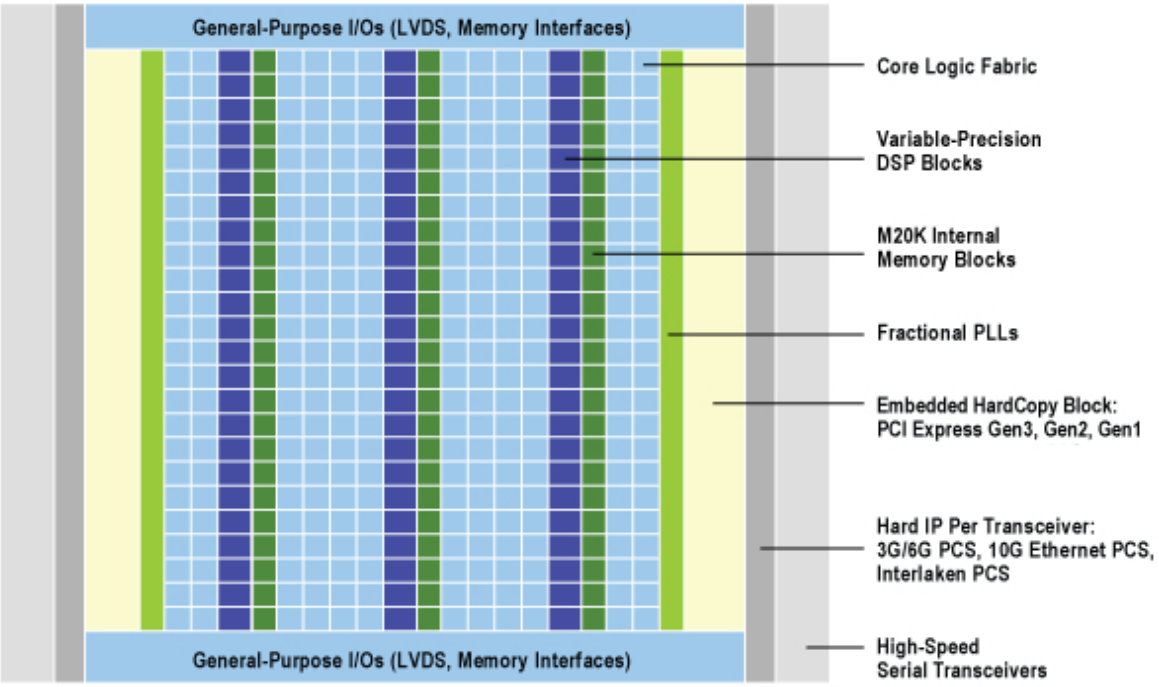

FIGURE 2.3 – Architecture de la puce au coeur du FPGA Altera Stratix V[1].

Physiquement, le FPGA est un amalgame de plusieurs structures bas-niveau tel des blocs de mémoire, de la logique combinatoire, des blocs de traitement de signal numérique, tous reliés par un réseau d'interconnexion programmable. Leur répartition est assez constante, hormis les entrées et sorties, tel que présenté à la figure 2.3. Il n'y a donc pas de bloc prédéfini comme étant une unité de calcul ou un élément de traitement.

Lorsque ce projet était en cours, le seul fabricant de FPGA à supporter OpenCL était Altera, avec son kit de développement logiciel (SDK) OpenCL. Il sera donc seulement question des FPGA de cette société dans le présent ouvrage. Cependant, récemment, le fabricant Xilinx a aussi annoncé le support OpenCL avec son environnement de développement SDAccel. Une exploration de ses outils a été réalisée et est présentée à la section 2.6.

2.3.1 Compilation

Il faut faire la distinction avec les outils existants permettant de générer une architecture selon un code C, appelés "C-to-Gates". Ceux-ci n'adoptent aucun standard et leur performance peut laisser à désirer. Avec OpenCL et le parallélisme explicite, il est facile de générer une architecture sur mesure et adaptée au problème.

Afin d'exploiter ce parallélisme, le SDK OpenCL génère d'abord un pipeline afin d'obtenir une

1. Diagramme en provenance de https://www.altera.com/products/fpga/stratix-series/stratix-v/features.html.

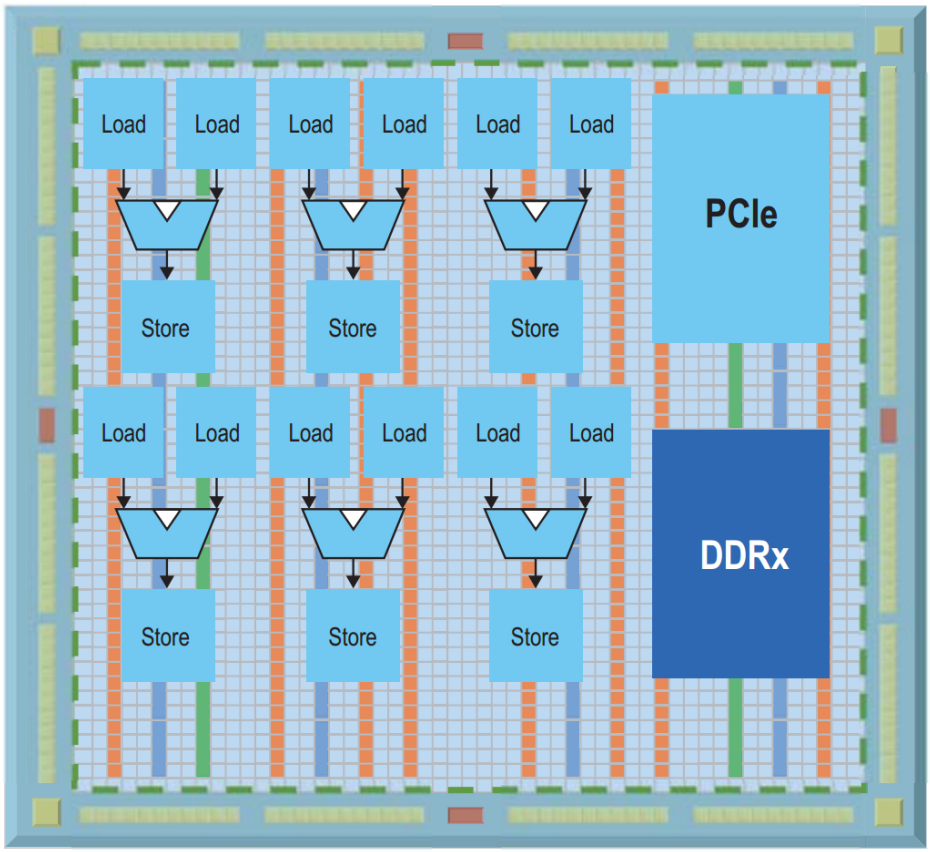

FIGURE 2.4 – Instanciation d'une architecture OpenCL à pipelines répliqués dans un FPGA[1].

cadence de traitement d'un élément par coup d'horloge. Ensuite, il est possible de rendre ce pipeline plus large en y traitant plus d'éléments à la fois, à la manière SIMD. Puis, le pipeline en entier peut être dupliqué pour un nombre augmenté d'unités de calcul. L'architecture résultante contient aussi les contrôleurs de mémoire et de bus PCI-e, requis par le standard OpenCL. Le tout est présenté à la figure 2.4.

Entre chaque étape dans le pipeline, des registres servent à conserver les résultats intermédiaires. C'est la mémoire privée, tel qu'expliqué en 1.3.1. Puis, pour la mémoire locale, il s'agit des blocs de mémoire à accès direct intégrés, ainsi que les puces externes de mémoire vive pour la mémoire globale.

Les branchements dans le code dédoublent le matériel nécessaire dans le FPGA afin de rendre possible l'exécution de chaque branche. Cependant, ils n'infèrent aucune pénalité de performance comparé aux GPU.

Tel qu'expliqué à la section 1.3.1, deux types de kernels sont possibles. Ceux de type tâche

1. Diagramme en provenance de https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01173-opencl.pdf.

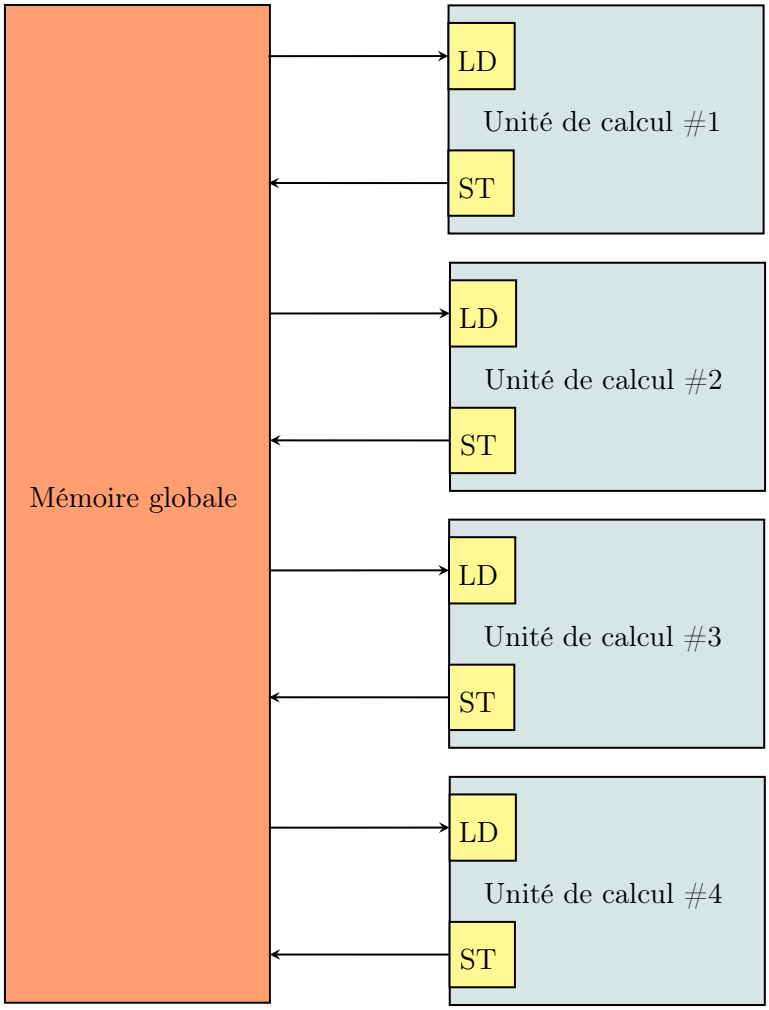

FIGURE 2.5 – Réplication d'une unité de calcul.

doivent contenir une grande boucle principale qui peut être pipelinée, afin d'obtenir une performance adéquate. Dans ce cas, il n'y a pas de groupe de travail, puis les mémoires locales et privées sont accessibles pour toutes les itérations. Il est conseillé d'utiliser ce type de kernel lorsque des dépendances sont présentes entre chaque élément à traiter.

Dans le cas des kernels qui consistent en une plage d'exécution de dimension N, c'est leur mise en file d'attente qui détermine la grosseur globale du problème au lieu du nombre d'itérations de la boucle. La mémoire locale est spécifique à un groupe de travail, tandis que la mémoire privée n'est valide que pour une même unité de travail.

En guise d'optimisation, si les ressources dans le FPGA le permettent, multiplier les unités de calcul est possible, tel que représenté à la figure 2.5. En supposant que les accès mémoires ne soient pas un facteur décisif de la performance, la cadence de traitement du kernel se voit multipliée par le nombre d'unités de calcul.

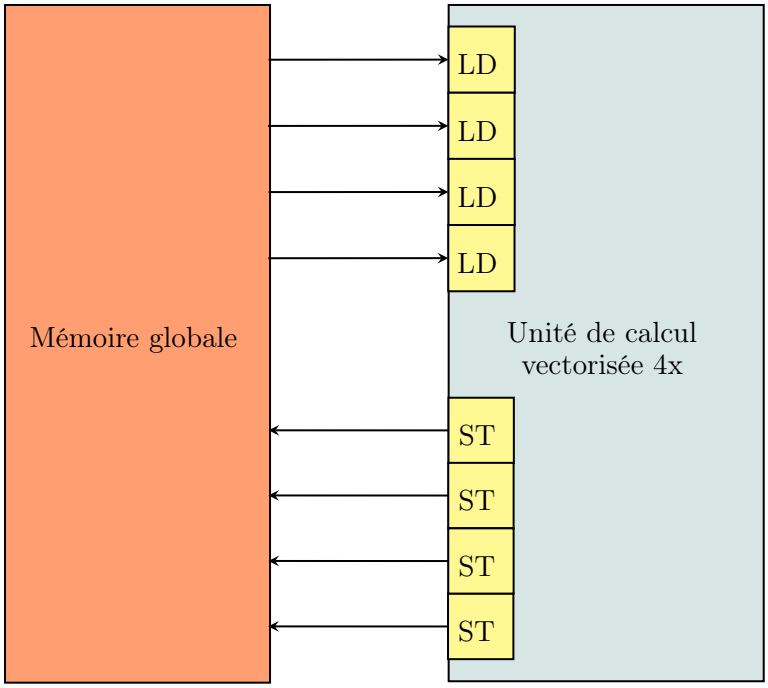

FIGURE 2.6 – Vectorisation d'une unité de calcul.

Le SDK permet aussi de vectoriser les unités de calcul afin d'exécuter les mêmes instructions sur plusieurs éléments en parallèle, tel que représenté à la figure 2.6. On assiste aussi à une multiplication de la cadence de traitement, sauf qu'une vectorisation est moins coûteuse en ressources et est généralement moins demandante sur le système de mémoire à cause de la prochaine optimisation qui sera expliquée. La vectorisation est donc préférée à la réplication d'unités de calcul.

Ces deux techniques permettent d'augmenter la performance même si le pipeline généré est souvent bloqué par des accès mémoire. Une optimisation possible qui permet de diminuer la bande passante requise consiste à fusionner les accès mémoire de façon statique, c'est-à-dire lors de la compilation, tel que représenté à la figure 2.7. Cette opération est réalisée automatiquement lors d'une vectorisation, mais il est possible de le faire manuellement en utilisant les types de données vecteur telles que *uint2*, *float4*, *long8*, etc.

Finalement, il est aussi possible de dérouler les boucles dans le code pour un parallélisme accru. Une boucle dans le flot de traitement résulte en une section du pipeline qui est surutilisée d'un facteur du nombre d'itérations de la boucle, puisque chaque élément de travail doit y passer ce nombre de fois au lieu de seulement une fois pour le reste du pipeline. Un déroulement partiel de la boucle permet d'alléger ceci, mais requiert bien évidemment de la logique de contrôle supplémentaire pour la gestion de l'éventualité où le nombre d'itérations ne serait pas un multiple du facteur de déroulement. Si on connaît le nombre d'itérations à la compilation, il devient possible de dérouler la boucle en entier, ce qui est le cas optimal. Le nouveau flot

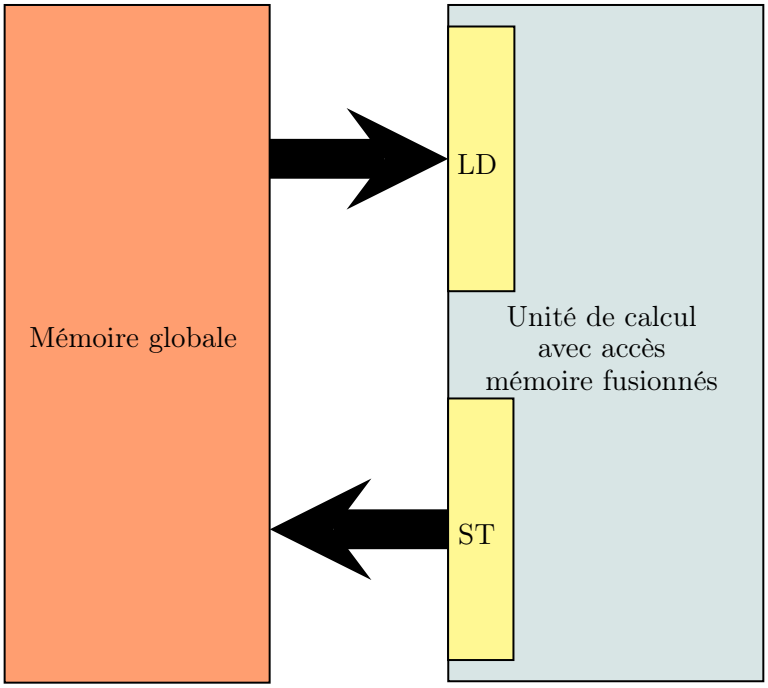

FIGURE 2.7 – Groupement des accès mémoire.

de traitement est représenté à la figure 2.8.

2.3.2 Évaluation de l'architecture

L'optimalité de l'architecture générée est jugée selon deux critères, soit la cadence de traitement des éléments, mesurée en éléments/seconde, et l'utilisation des ressources, mesurée en pourcentage des ressources disponibles dans le FPGA. Celles-ci sont distinguées en cinq types, soit la logique combinatoire, les unités de logique arithmétique, les registres, les blocs mémoire et les processeurs de traitement de signal. Ces résultats sont estimés immédiatement après la compilation du code OpenCL, mais avant la génération de l'architecture matérielle. En effet, cette dernière étape prend plusieurs heures à compléter, donc il serait malcommode de devoir attendre la fin de cette opération pour obtenir ces résultats. Du même coup, il devient évident que cette génération ne peut être faite lors de l'exécution du programme hôte, tel qu'expliqué en 2.1 ; il faut donc charger son résultat en différé.

Un facteur augmentant la volatilité de la performance est si les accès mémoire sont dépendants des données. C'est justement le cas avec OCLRay. Par exemple, c'est l'arc sortant d'un k-mer qui indique quel autre k-mer visiter lors du suivi d'un chemin. Si les deux k-mers sont consécutifs en mémoire, le deuxième peut être accédé avec une latence minimale. Considérant la grosseur du graphe et la fonction de hachage uniforme, il est peu probable que ce soit le cas, mais cela est un élément qui peut fausser les estimations de performance pré-génération. La meilleure mesure consiste donc à considérer le temps d'exécution des kernels sur un groupe

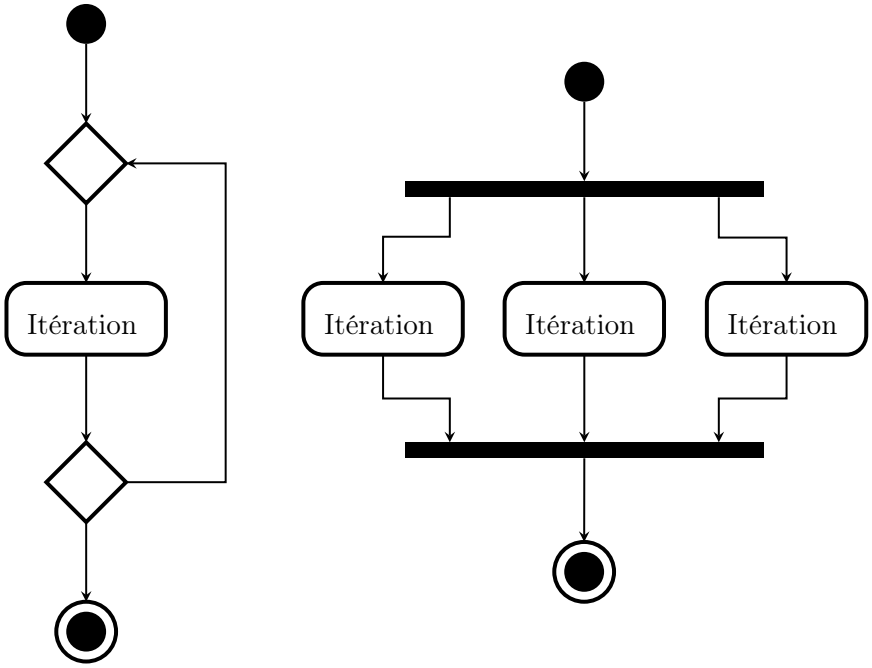

FIGURE 2.8 – Flot de traitement sans et avec déroulement complet d'une boucle.

de données véridique.

Altera fournit au sein de son SDK un outil qui permet de quantifier les blocages du pipeline dûs aux accès mémoire. Son profileur, lorsqu'activé, vient placer des registres supplémentaires entre chaque étape du pipeline afin de mesurer les délais.

2.4 Consommation énergétique

Les différents types d'accélérateurs se démarquent énormément sur la consommation énergtique. Le graphique 2.9 représente une comparaison approximative des plages de consommation possibles à pleine charge de travail pour chaque type d'accélérateur. Il est question ici de puces destinées au marché du calcul haute-performance et non au marché de la portabilité et bien sûr, en général il y a une forte corrélation entre la performance brute et la consommation pour un même type d'accélérateur.

2.5 Mémoire hétérogène unifiée

Certains accélérateurs possèdent une mémoire globale qui est physiquement la même que la mémoire centrale de l'hôte. De ceux-ci, un sous-groupe a la possibilité de partager les deux espaces mémoire de façon unifiée, comparativement aux autres qui ne font que réserver une portion pour l'accélérateur. Il est donc possible de n'envoyer qu'un pointeur à ce dernier en

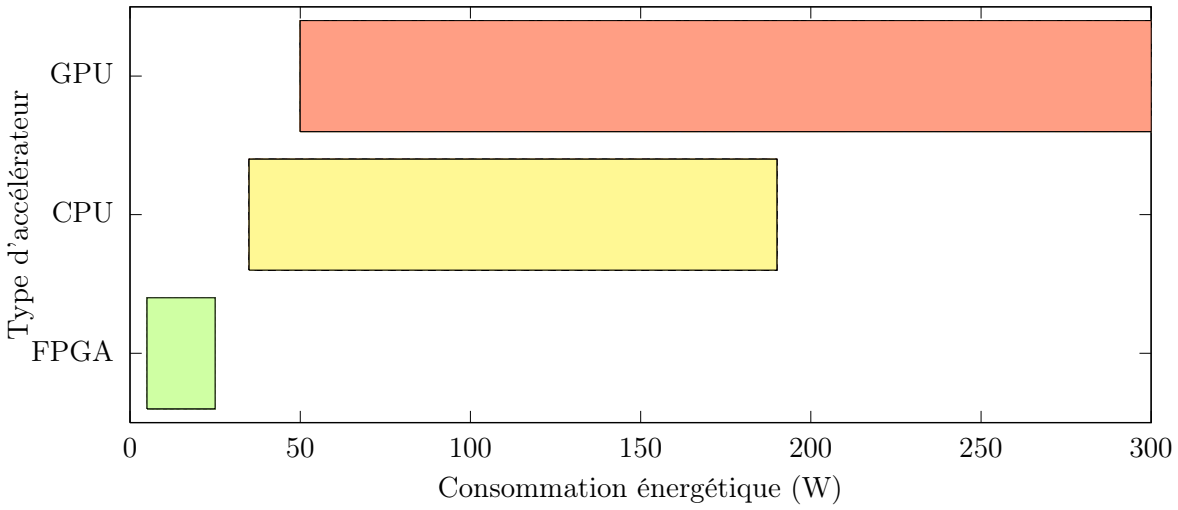

FIGURE 2.9 – Plages de consommation approximatives possibles pour les différents types d'accélérateurs.

guise de copie de mémoire. Le temps de transfert est donc pratiquement nul. On parle alors de plateformes avec accès mémoire hétérogène unifié (hUMA).

Présentement, les quelques plateformes ayant cette capacité sont tous les processeurs centraux, les récentes générations de processeurs graphiques intégrés dans la même puce que les processeurs centraux, ainsi que les systèmes sur puce d'Altera.

2.6 Support OpenCL chez Xilinx

La solution Xilinx pour le support d'OpenCL a aussi été explorée. Elle a ultimement été rejetée pour les tests de ce projet, puisque certains problèmes de compilation du kernel se présentaient. La solution étant encore en phase d'essai fermée, Xilinx a encore du temps pour arranger les problèmes. Néanmoins, une comparaison sommaire des outils a tout de même été effectuée.

La différence fondamentale entre les deux SDK réside dans le fait que Xilinx contrôle l'entièreté de l'environnement dans lequel le développement est fait. Dans un premier temps, le développeur doit écrire un script Tcl afin de spécifier le type de compilation, ses paramètres, etc. Lors d'exécution de ce script, un compilateur livré avec l'installation est appelé pour la compilation du code hôte. Il semble qu'il ne puisse pas être remplacé par un autre compilateur au choix du développeur. Le SDK Altera ne fait pas de différence quant à lui ; il s'agit simplement de faire l'édition de liens selon les librairies Altera fournies. De plus, pour Xilinx, la gestion du code source ne peut être fait à la guise du développeur ; il faut ajouter les fichiers manuellement dans le script Tcl et spécifier leur type. Conséquemment, il est impossible d'utiliser d'autres

méthodes plus courantes comme par exemple des Makefiles ou des projets Visual Studio.

Pour ce qui est du flux de travail, Xilinx suggère une procédure semblable à Altera. Il s'agit de coder une première version du kernel et de l'application hôte, compiler le tout en utilisant le CPU comme accélérateur et optimiser. Puis, il faut ensuite compiler pour l'accélérateur FPGA. Une simulation matérielle peut être roulée pour confirmer la fonctionnalité correcte de l'architecture générée, puis une autre phase d'optimisation peut prendre place, cette fois en utilisant des attributs de compilation dans le kernel. Ceux-ci influenceront l'architecture générée au même titre que que les commandes *#pragma* pour le compilateur Altera. Il est aussi possible de profiler l'exécution, cependant les informations comme la latence ne sont pas rapportées pour chaque ligne de code dans le kernel.

Les optimisations possibles au niveau de l'architecture mémoire sont semblables à celles de la solution Altera. Il y en a une supplémentaire cependant qui vaut la peine d'être mentionnée. À la section 1.3.1, il était expliqué que la mémoire globale, accessible à toutes les unités de travail dans tous les groupes, résidait dans la mémoire externe. Dans le cas où elle n'a pas besoin d'être accédée par le système hôte, Xilinx permet son instanciation directement dans la puce FPGA, ce qui résulte évidemment en une réduction de la latence d'accès et une amélioration de la bande passante. Cette optimisation fait partie de la spécification OpenCL seulement à partir de la version 2.0, mais elle a tout de même été implémentée. De plus, Xilinx permet aussi le contrôle du partitionnement de la mémoire locale, qui peut être soit en blocs, cyclique ou complète.

En résumé, l'approche Xilinx adopte plus une approche matérielle, tandis que les développeurs logiciels se sentiront plus en terrain connu avec la solution Altera.

Chapitre 3

Parallélisation

3.1 Parallélisation originale

Ray partionne le problème sur plusieurs noeuds de calcul d'un super-calculateur. En utilisant une interface de passage de message (MPI), un processus roule sur chacun d'entre eux. L'avantage d'une telle parallélisation est que, pourvu que suffisament de noeuds de calcul participent à l'exécution, une quantité suffisante de mémoire vive sera disponible pour stocker même les plus gros problèmes d'assemblage. Ray peut aussi exploiter plusieurs fils d'exécution sur le même noeud en utilisant l'option "–mini-ranks." Dans ce cas, la communication se fait via la mémoire principale. Cependant, si le problème d'assemblage est destiné à être résolu sur une seule machine, MPI (Message Passing Interface) peut tout de même être utilisé en exécutant au même endroit les nombreux processus qui seraient autrement sur des machines distinctes. Dans ce cas, une mémoire partagée sera aussi utilisée pour le passage de messages. Donc, dépendamment des paramètres de lancement fournis par l'utilisateur, Ray aura plusieurs processsus auxquels on attribue un rang et un ou plusieurs fils d'exécution, ayant chacun leur mini-rang. Les fonctionnalités de Ray sont codées en tant que modules d'extension. Chacun d'eux sont enregistrés puis construits. Ensuite, le "ComputeCore" est lancé. Celui-ci reçevra les messages provenant d'autres rangs, les traitera, fera le travail demandé sur les données appropriées, puis enverra les messages requis, tant et aussi longtemps qu'il est gardé en vie par le noeud maître. Le tout est illustré à la figure 3.1.

De plus, l'exécution de chaque étape est découpée en plusieurs petits morceaux séquentiels qui seront exécutés à raison d'un par unité de temps, ce que Ray appelle "tick".

Les méthodes permettant de traiter les messages et les données peuvent être décomposées encore plus, tel que montré à la figure 3.2. Premièrement, la réception d'un message est dépendante du fait que les mini-rangs ont été activés. Ensuite, le traitement du message inclut l'étape de routage si la fonction a été activée. Celle-ci permet d'influencer sur quels noeuds un message sera relayé selon une architecture de réseau virtuel comme il sera décrit plus tard.

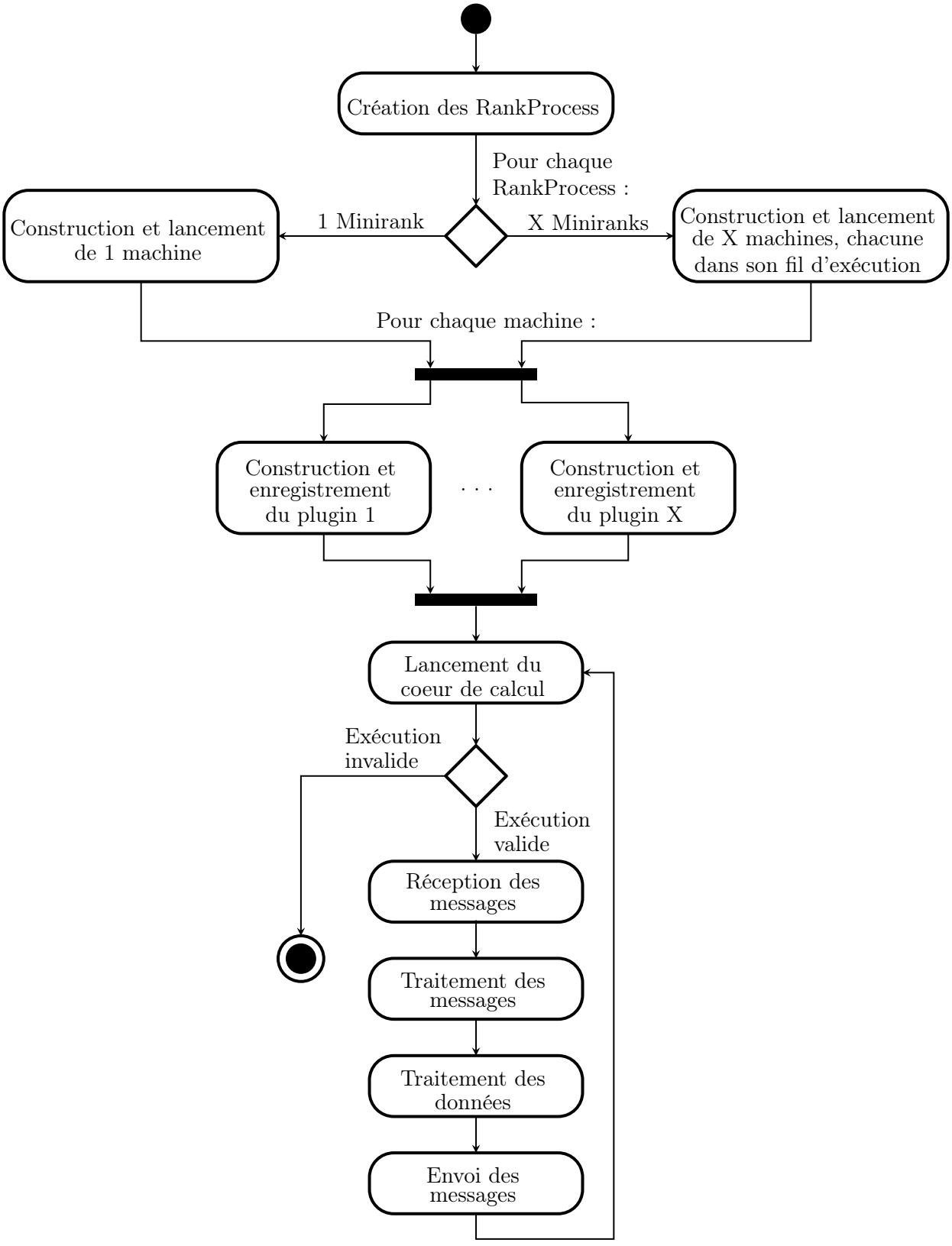

FIGURE 3.1 – Graphe de fluence du fonctionnement général de Ray avec le passage de messages.

Ensuite, lors du traitement des messages, c'est la classe SwitchMan qui contrôle le flot de travail pour chacun des rangs. Pour chacun de ceux-ci, elle décidera si un certain travail doit être exécuté, soit en appelant le gestionnaire des étiquettes de messages. Pour le traitement des données, c'est le gestionnaire d'exécution des tâches maîtres qui est appelé, puis le déroulement est enregistré dans le journal événementiel par unité de temps. Il en va de même pour les tâches esclaves. Finalement, pour l'envoi des messages, le routage est effectué à nouveau. Ces derniers sont aussi enregistrés dans le journal, puis envoyés.

Tel que décrit en 1.2, Ray utilise un graphe de Bruijn. Celui-ci est distribué parmi tous les noeuds de calcul participant à l'exécution. Chaque processus s'occupe des sommets pouvant avoir une valeur comprise dans une plage donnée, d'une grosseur variant selon le nombre de processus. Il devient alors évident que, lors d'une opération quelconque dans l'algorithme, si un sommet dans le graphe doit être consulté, une requête doit être placée sur le réseau afin de demander l'information nécessaire au bon noeud. C'est seulement lorsque l'information est retournée que le traitement de l'information courante peut être continué. Pour ce qui est du partionnement de l'assemblage en tant que tel, Ray exécute l'agrandissement d'une semence par fil d'exécution. Lorsqu'un k-mer résidant sur un autre noeud est requis, son obtention est demandée par le réseau d'interconnexion, puis l'assemblage continue. Il est donc clair que la rapidité d'exécution devient donc dépendante de non seulement la rapidité de calcul des processeurs, mais surtout de la vitesse du réseau d'interconnexion et de sa topologie. C'est pourquoi le routeur de messages existe : minimiser les délais de communications en ajustant l'architecture du réseau virtuel pour respecter le plus possible le schéma d'interconnexion réel entre les noeuds. Pour ce faire, lors du lancement de l'application, l'utilisateur doit préciser ce dernier avec les options en ligne de commande *-route-messages*, *-connection-type* et *-routing-graph-degree*. Les schémas d'interconnextion tels polytope, torus, debruijn et kautz [5] de différents degrés peuvent être choisis.

Il est évident que tout le code qui a trait à la parallélisation dans Ray ne fait aucun sens pour l'algorithme porté à OpenCL. La prochaine section traitera de cette nouvelle parallélisation.

3.2 Parallélisation OpenCL

3.2.1 Méthodologie

La méthodologie pour migrer une application existante sur FPGA avec OpenCL consiste en plusieurs étapes. Elles assurent un fonctionnement adéquat ainsi qu'un temps de développement optimal.

En premier, il est conseillé de réaliser bêtement un modèle avec le langage C, qui s'exécute sur le processeur central. Le but de cette première étape est d'obtenir des résultats de référence. Elle permet aussi au programmeur d'acquérir une connaissance approfondie de l'algorithme ainsi

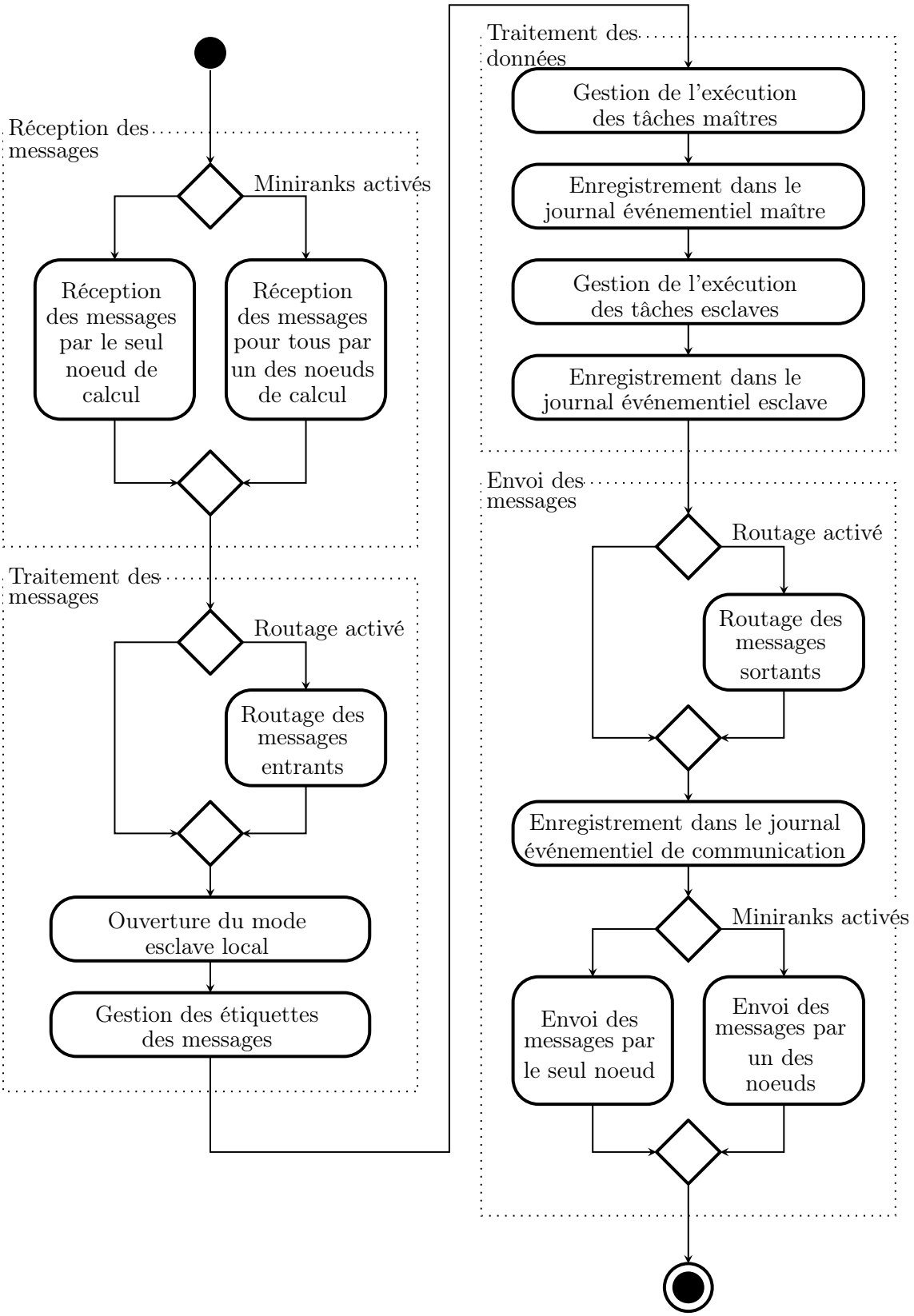

FIGURE 3.2 – Graphe de fluence des fonctions de traitement des messages et des données.

que de déterminer les tailles minimales des variables afin que les données ne produisent pas de débordement. Il est conseillé de mettre de côté, pour l'instant, les ajouts complémentaires à l'algorithme.

Ensuite, un kernel OpenCL est réalisé à partir du code C. Ici, la fonction principale réalisant l'algorithme peut être copiée directement dans un kernel de type tâche. De plus, les tampons temporaires doivent être évités, de par l'absence d'allocation de mémoire, donc une taille fixe pouvant contenir ces données doit être déterminée. Puis, l'émulateur Altera peut être utilisé pour valider que le fonctionnement est toujours adéquat.

Ici, les premiers tests de performance ainsi que d'utilisation des ressources peuvent être réalisés, en autant que ce modèle puisse être synthétisé et placé sur FPGA. L'effort qui sera requis afin d'obtenir une exécution sur FPGA respectant les contraintes de design peut ensuite être estimé.

Par défaut, le compilateur s'efforce d'extraire le parallélisme des boucles d'exécution, mais le kernel peut être optimisé autrement. La migration vers une plage d'exécution de dimension N peut être réalisée à la place. Des gains importants peuvent être obtenus si l'algorithme possède une boucle principale avec peu ou pas de dépendance entre les itérations. Si un kernel de type tâche est conservé, il est crucial d'incorporer un bon patron de conception maintenant. Par exemple, pour un flot de données continu, les registres à décalage sont à considérer. Les boucles doivent être déroulées le plus possible pour obtenir des indices d'accès constants aux registres, ainsi que pour ne pas bloquer le pipeline d'exécution. L'algorithme peut aussi être retravaillé s'il ne se prête pas facilement à la parallélisation de type SIMD.

La prochaine étape consiste à prototyper le tout sur FPGA et profiler l'exécution. De réelles informations sur le blocage du pipeline et les accès mémoire seront obtenus. Les paramètres de synthétisation du kernel peuvent être ajustés et les changements profilés à nouveau. Finalement, il est avantageux de faire bien des essais afin de converger vers la solution adéquate [9]. Le flot de la procédure en entier est représenté à la figure 3.3.

3.2.2 Modifications à l'algorithme dues aux contraintes OpenCL

Avec OpenCL, il n'y a pas d'allocation de mémoire possible à partir d'un kernel. Par conséquent, les structures utilisées doivent être adaptées à une telle contrainte ; les allocations doivent être gérées à l'avance, par le programme hôte.

Ray utilise une table de hachage afin de stocker le graphe, basée sur les "sparse tables" de Google. Cette implémentation consiste en un tableau de tableaux dynamiques, lequels sont responsables d'une certaine quantité d'indices. Cette implémentation a un avantage principal ; elle entrepose seulement les sommets qui ont été insérés. En effet, à chacune de ces opérations, un tableau dynamique doit être agrandi [25]. Afin d'obtenir une structure contiguë en mémoire et sans allocation dynamique, donc adaptée au traitement avec OpenCL, une table de hachage

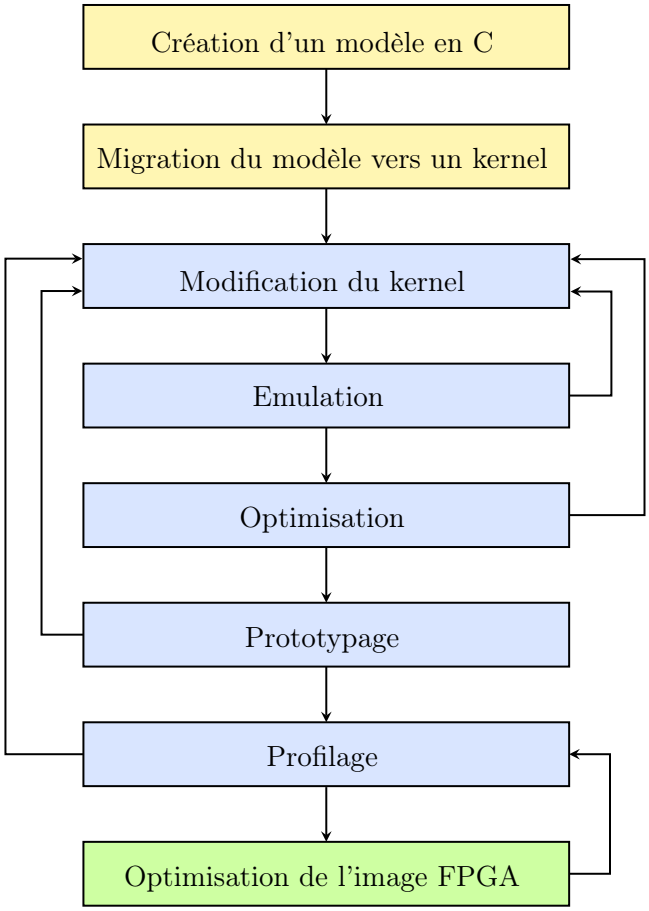

FIGURE 3.3 – Méthodologie de design d'une application OpenCL avec le SDK Altera.

à adressage ouvert a été choisie. Cela implique que toutes les entrées de la table sont stockées dans un même tableau. Il n'est donc pas nécessaire de négocier les transferts de mémoire OpenCL, qui seraient nécessaires pour chaque pointeur. Également, il n'y a aucune allocation de mémoire requise pour grossir un sous-tableau, puis afin d'éviter l'ajustement dynamique de la table maîtresse, on opte pour l'estimation de la grosseur du graphe à l'avance. Une seule allocation mémoire est effectuée, par le programme hôte.

Pour la même raison, un espace mémoire d'une grosseur estimée est préalloué pour contenir les contigs. Puisque chacun de ceux-ci sont assemblés en parallèle, chaque élément de travail est responsable de réserver un bloc parmi cet espace mémoire et y stocker une partie de son contig. Le propriétaire de ce bloc est déterminé par son identifiant unique. De son côté, Ray entrepose les résultats de l'assemblage dans un vecteur de vecteurs, ce dernier contenant les k-mers formant un chemin les uns à la suite des autres.

De plus, OpenCL n'a pas accès au système de fichiers. À cause de cela, autant la construction du graphe que l'écriture des contigs dans un fichier sont exécutés sur l'accélérateur. Ce sont la première et dernière étape de l'algorithme, respectivement.

Finalement, une contrainte sur l'allocation de mémoire maximale existe dans la spécification OpenCL, appelée `CL_DEVICE_MAX_MEM_ALLOC_SIZE`. Sa valeur minimale est le maximum entre le quart de la taille de la mémoire globale (`CL_DEVICE_GLOBAL_MEM_SIZE`) et 128 MB, pour les appareils qui ne sont pas du type `CL_DEVICE_TYPE_CUSTOM`. [13] Afin de contourner cette limite, le graphe est alloué en quatre espaces mémoires distincts, de taille égale. Selon l'indice du sommet à consulter, le chargement est aiguillé sur l'espace mémoire correspondant. Cela permet donc au graphe d'occuper plus que `CL_DEVICE_MAX_MEM_ALLOC_SIZE`.

3.3 Modifications à l'algorithme pour la performance et l'économie de mémoire

Hormis le port de l'algorithme à OpenCL, plusieurs changements ont été apportés afin d'assurer une performance optimale. Ici sont décrits les changements principaux réalisés.

3.3.1 Modification des pointeurs de début des lectures

Dans les autres assembleurs de lectures courtes tels Ray et Velvet, l'annotation d'une lecture mentionne la position du sommet dans celle-ci. Ici, nous procédons d'une autre manière, qui est permise par la structure différente. Puisque les nucléotides sont stockés les uns à la suite des autres, il suffit simplement de modifier le début de la lecture pour pointer à l'endroit où le premier sommet unique, celui annoté tel qu'expliqué en 1.2.3, a été rencontré. L'espace de stockage utilisé pour enregistrer la position est alors économisé. Il faut aussi ajuster la longueur de la lecture en conséquence.

3.3.2 Longueurs de k-mers fixes dans les kernels OpenCL

Ray original prend en argument la longueur des k-mers désirés. Du point de vue de l'utilisateur, un seul exécutable gère toutes les longueurs de k-mers. OCLRay conserve la même interface. Cependant, cette longueur est indiquée dès le lancement du programme et ne change pas durant l'exécution. Cela permet une compilation et un chargement paramétré des kernels OpenCL, selon la valeur indiquée par l'utilisateur. Des simplifications améliorant la performance s'en suivent.

Premièrement, la taille de la mémoire requise pour stocker un k-mer est connue. Dans un FPGA, il est donc possible d'instancier exactement la bonne quantité au lieu d'avoir un surplus pour les k-mers plus volumineux. Du même coup, cela n'engendre pas d'opérations mathématiques sur des pointeurs pour les accès à ces k-mers, ce qui est néfaste pour la performance de l'architecture générée.

Deuxièmement, pour plusieurs opérations, des boucles ayant un nombre d'itérations en fonction de la longueur des k-mers sont requises. Ne pas connaître ce paramètre à l'avance empêche

l'optimiseur de dérouler ces boucles. La performance est directement affectée, en plus de nécessiter des boucles imbriquées, une autre structure non-optimale pour un FPGA.

3.3.3 Grosseur de graphe d'une puissance de 2

Lors de la détermination de la grosseur du graphe au lancement du programme, on arrondit celle-ci à la puissance de deux supérieure la plus proche. Cela permet d'éviter les opérations modulo qui servent à calculer la position d'un sommet dans le graphe, suite à la fonction de hachage [29]. Cette opération peut être remplacée par une opération binaire "ET", tel qu'illustré dans l'équation suivante :

$$index_{vertex} = hash\%size_{table} = hash\&(size_{table} - 1) \tag{3.1}$$

Une opération de division servant à la détermination de l'espace mémoire approprié, tel qu'expliqué en 3.2.2, peut aussi être évitée puisqu'elle devient équivalente à un logarithme binaire ($sizeof_{bits}()$), un décalage de bit ($\gg$) et un scan binaire ($BSR()$), comme ceci :

$$index_{buffer} = hash/size_{table} = hash \gg sizeof_{bits}(size_{table}) - BSR(tableSize) \tag{3.2}$$

Cela peut avoir une influence sur le temps de calcul par un processeur central. En effet, l'architecture x86 Haswell d'Intel affiche une latence de 95 cycles pour une opération modulo (produit-dérivé de l'instruction assembleur DIV), tandis qu'une opération binaire "ET" a une latence de 1 cycle. Similairement, une opération de recherche binaire (BSR) prend 3 cycles, une sousraction prend 1 cycle et un décalage prend 1 cycle [12].

De façon plus importante, sur un FPGA, la division et le modulo sont des opérations à proscrire de par leur coût élevé en ressources, tel que mentionné à la section 2.3. Il a été constaté que sur un Altera Statix V GX, à l'intérieur d'une boucle déroulée 16 fois, le coût de ces opérations nécessaires au calcul des indices dans le graphe devient très prohibitif ; 6% de la logique combinatoire, 5% des unités de logique arithmétique, 1% des registres, 18% des blocs mémoire et 16% des processeurs de traitement de signal.

3.3.4 Résolution des collisions par sondage pseudo-aléatoire

C'est le même type de structure qui est utilisée pour stocker les paires clef-valeur qui sont insérées dans l'élément du type Dictionnaire en Python. Elle a été choisie notamment pour la possibilité d'y retirer un élément, d'itérer dans celle-ci et d'en obtenir un élément inséré très rapidement peut importe sa grosseur [21]. La méthode de résolution de collision dans ce dernier est fort intéressante, et elle est réutilisée pour OCLRay. La méthode la plus simple

```
/* Starting slot */
slot = hash;

/* Initial perturbation value */
perturb = hash;

while (slot.isFull() && slot.item != itemToFind) {
       slot = (5*slot) + 1 + perturb;
       perturb >>= 5;
}
```

FIGURE 3.4 – Méthode de résolution de collision dans la table de hachage.

serait une recherche linéaire ou quadratique, tandis que Ray procède à un hachage double. Cette dernière technique consiste à déterminer les prochaines cases mémoire à visiter selon une deuxième fonction de hachage. La méthode inspirée du Dictionnaire Python consiste à modifier le hash avec une valeur de perturbation, de telle sorte que tous les nombres entre 0 et la grosseur de la table de hachage moins 1 sont générés, dans un ordre pseudo-aléatoire, tel qu'indiqué à la figure 3.4.

Le calcul d'un nouvelle position devient donc très léger à exécuter, comparativement à ce que Ray propose. Cela promet un gain en performance ainsi qu'une économie de ressources pour le FPGA.

3.3.5 Élargissement de la table de hachage dans une seconde dimension

Le sondage pseudo-aléatoire a un effet pervers ; l'accès à des indices dans le tableau d'une telle façon affecte la performance. En effet, le fonctionnement de la mémoire globale du FPGA et de la mémoire hôte, soit de la mémoire de type DDR3, performe mieux pour un accès séquentiel, même s'il s'agit de mémoire dite à accès aléatoires (RAM). La solution finale consiste donc à étirer la table de hachage sur une deuxième dimension d'une largeur de plusieurs éléments. Lors d'une recherche dans cette structure, tous les éléments d'une même ligne sont vérifiés, ce qui résulte en plusieurs accès consécutifs et leur performance accrue de plusieurs ordres de grandeur [30]. La largeur utilisée dans OCLRay est de quatre ; deux lignes totalisent donc huit espaces, un nombre intéressant en terme d'économies de ressources à cause de la modification suivante.

3.3.6 Imposition d'un maximum de collisions lors d'une recherche et réordonnancement des entrées dans la table de hachage

Il a été calculé que dans une table de hachage à adressage ouvert utilisant une résolution de collisions pseudo-aléatoire, la quantité d'accès attendue pour trouver un élément est dictée par la formule suivante [11] :

$$A = \left\lceil -\frac{1}{\alpha} \ln(1-\alpha) \right\rceil \tag{3.3}$$

où α est le taux d'occupation de la table, entre 0 et 1. Plus ce dernier est élevé, plus il y a de collisions, donc un nombre supérieur d'accès mémoire est requis pour trouver un élément. À l'inverse, moins il est élevé, plus il y a d'espace mémoire non utilisé.

OCLRay vise 0.75%, un taux qui représente un bon équilibre. En utilisant ce chiffre dans la formule précédente, on obtient que la quantité d'accès moyenne requise est de 1.848. Lors de la création du graphe, nous imposons un maximum de 7 collisions dans la table ou huit espaces mémoire, bien au dessus de la quantité moyenne d'accès. Par conséquent, 2 rangées pleines tel qu'expliqué en 3.3.5 doivent être vérifiées. Le tout est représenté à la figure 3.5. Il est à noter que la courbe représentée est une moyenne, donc la fonction partie entière par excès de la formule originale n'est pas appliquée.

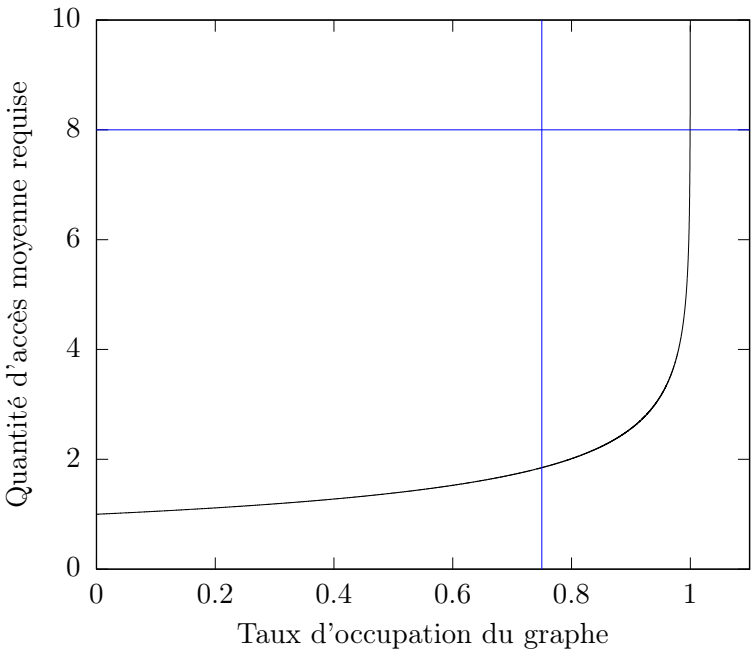

FIGURE 3.5 – Quantité moyenne d'accès requis vs. taux d'occupation de la table de hachage.

Il y aura cependant forcément bien des k-mers dans la table de hachage qui requiéront un nombre très grand d'accès pour être dénichés. Lorsque cela aura lieu, le pipeline dans le FPGA bloquera et ce, pour tous les éléments de travail en cours. Selon la longueur du pipeline, cela pourrait être très coûteux. Pallier cette éventualité est le but de l'imposition du maximum de collisions pour un même sommet. Du même coup, cette limite impose un nombre fixe d'itérations de la boucle de recherche d'un sommet dans le graphe. Elle peut donc être déroulée en entier et empêcher la formation d'une boucle imbriquée.

Le maximum de collisions est rendu possible grâce à un réordonnancement des entrées dans la table de hachage lors de son remplissage. Concrètement, lorsque le maximum est atteint, on échange l'entrée à ajouter avec une de celles qui occupe déjà un espace convoité, puis on recommence la recherche d'un espace libre pour cette dernière. Si celle-ci ne peut pas non plus être placée, on l'échange aussi avec une entrée différente de la première et on recommence. Ce manège peut potentiellement être très long, mais si le nombre de lectures par rapport au nombre d'écritures est suffisant, le temps d'exécution total s'en verra gagnant. Aucune mesure ne sera cependant faite à cet égard dans le cadre de ce projet, puisque la construction du graphe, exécutée sur processeur central, ne sera pas mesurée.

3.4 Partitionnement du problème

Toute parallélisation requiert que le problème soit partitionné, autant pour le cas d'une exécution en parallèle que d'une exécution en pipeline. Avec OpenCL, pour le type de kernel qui consiste en une exécution sur une plage de valeurs tel que décrit en 1.3.1, il s'agit de déterminer sous quel angle le problème est découpé, et cela peut être différent pour chacune des étapes de l'algorithme, décrites en 1.2.

Pour la partie de l'élimination des couvertures faibles, la séparation des unités de travail consiste simplement en chaque sommet dans le graphe. La plage d'exécution va donc de l'emplacement 0 de celui-ci jusqu'à l'emplacement $size_{table} - 1$.

Pour la partie du calcul de la distribution de la couverture, la séparation se fait aussi pour chaque sommet dans le graphe, mais une attention particulière est apportée à la grosseur des groupes de travail. Celle-ci doit être exactement égale à la plus grande valeur permise dans la distribution de la couverture, car chaque unité de travail est responsable de l'initialisation à zéro de la valeur correspondant à son identifiant local, variant entre 0 et $couverture_{max} - 1$, ainsi que de l'addition au total global de la valeur locale avec le même indice, et ce, à la toute fin du calcul.

Pour la partie de l'annotation des lectures, la séparation se fait au niveau de ces dernières. Dans le kernel, une boucle principale explore un nucléotide par itération. Plusieurs groupes de travail sont exécutés dans cette boucle au même moment, ce qui permet d'exploiter efficacement le pipeline. Les semences sont aussi générées lorsqu'elles sont détectées.

Puis, les éléments de travail du prochain kernel consistent en ces semences. Ici, la validité de chacunes d'elles doit être vérifiée et celles qui n'ont pas leur place, enlevées.

Finalement, les semences restantes sont les éléments de travail du dernier kernel, qui les agrandira pour former des contigs. Ici, une boucle principale procède à l'agrandissement de un nuclétide par itération.

Chapitre 4

Résultats

4.1 Montage de test

Pour effectuer les tests délivrant les résultats présentés dans ce chapitre, deux montages de test ont été utilisés. Ceux-ci sont dotés de cartes d'expansion PCI-e comprenant un FPGA sur lequel s'exécute les étapes parallélisées d'OCLRay :

Tableau 4.1 – Montage de test #1.

Processeur hôte	Intel *Haswell* Core i7-4770K
Carte mère	ASUS Sabertooth Z87
Mémoire vive hôte	Kingston HyperX 4x8GB DDR3-1600
Accélérateur 1	Intel *Haswell* Core i7-4770K - 4 coeurs / 8 threads - Base 3.5 GHz / turbo 3.9 GHz
Accélérateur 2	Nallatech PCIe-385N - Altera Statix V GX - 2x4 GB DDR3-1600

4.2 Temps d'exécution

Il est d'abord important de mentionner que dans tous les résultats de performance considérés, seulement les temps d'exécution des kernels sont mesurés. Nous écartons donc le temps de chargement ou de compilation de ces derniers ainsi que les transferts de mémoire. Ceci est dans le but d'obtenir des résultats les plus comparables et répétitifs possibles.

Les premiers tests de performance consistent à assembler des banques de lectures synthétiques qui ont été réalisées à partir d'ADN de plusieurs bactéries. Les figures 4.1, 4.2, 4.3, 4.4 et 4.5 sont les performances affichées par les jeux de données pour les bactéries Acidianus hospitalis, Agrobacterium vitis, Ecoli, Acaryochloris marina et Actinoplanes respectivement. Dans ces

figures, sur l'axe des x, les cinq étapes parallélisées avec OpenCL sont présentées. L'axe des y représente le temps requis en secondes pour l'exécution du kernel. Il est très clair que le FPGA est compétitif en termes de performance face au processeur central.

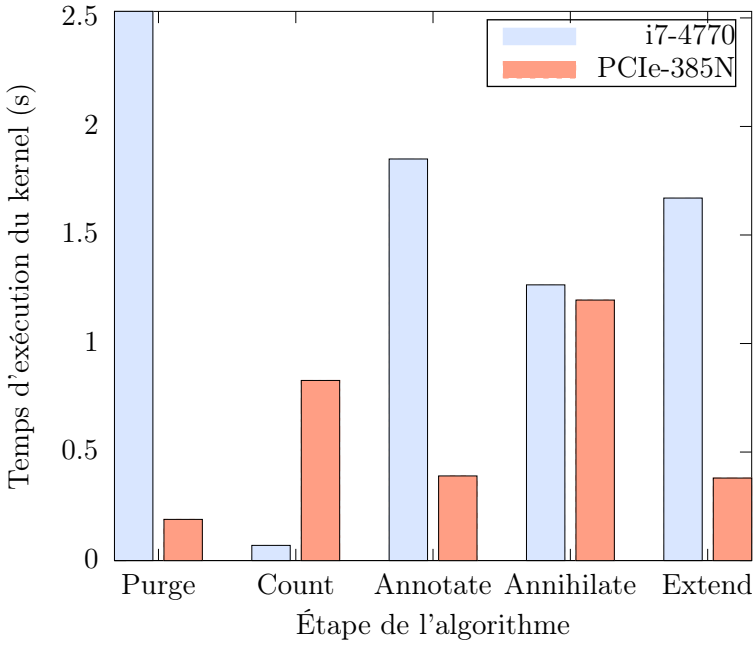

FIGURE 4.1 – Temps d'exécution des kernels pour la bactérie Acidianus hospitalis.

OCLRay a aussi été testé sur des lectures réelles. Le jeu de données affichant une performance présentée à la figure 4.6 consiste en la bactérie Salmonella enterica.

Dans le tableau 4.2, les temps d'exécution des kernels sur FPGA sont normalisés par rapport au processeur central. Les moyennes sont affichées tant par rapport aux jeux de données que par rapport aux étapes de l'algorithme.

Tableau 4.2 – Temps d'exécution des kernels sur FPGA, normalisés par rapport au CPU.

	Purge	Count	Annotate	Anihilate	Extend	Total
Acidianus hospitalis	0.075099	11.857143	0.210811	0.944882	0.227545	0.404601
Agrobacterium vitis	0.077449	7.153846	0.219331	0.609375	0.194937	0.305394
Ecoli	0.076106	6.200000	0.157585	0.648464	0.315385	0.249699
Acaryochloris marina	0.077049	6.200000	0.146384	1.235294	0.250000	0.267684
Actinoplanes	0.074572	3.724138	0.147841	0.345538	0.318332	0.221627
Salmonella enterica	0.076059	3.303030	0.189222	0.588583	0.310417	0.249740
Moyenne	0.076056	6.406360	0.178529	0.728689	0.269436	0.283124

Il est intéressant de voir que les kernels relativement simples qui ne peuvent être vectorisés,

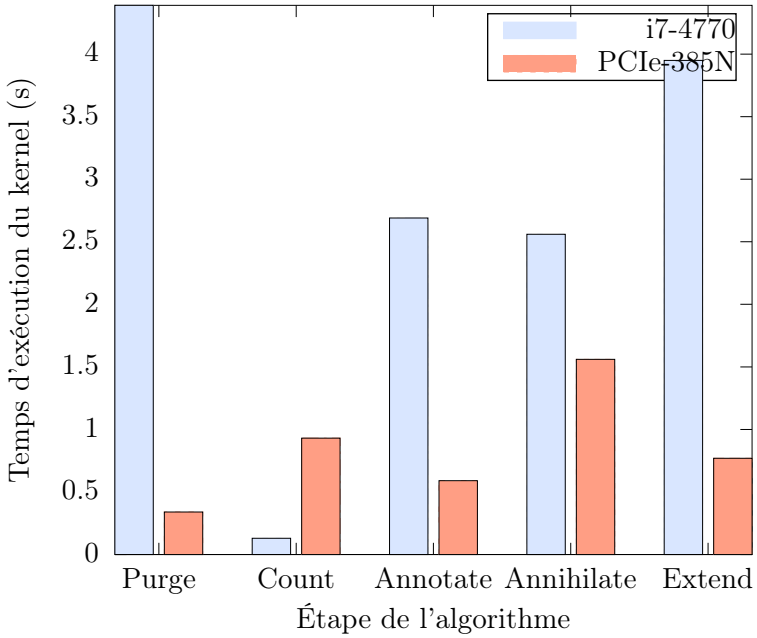

FIGURE 4.2 – Temps d'exécution des kernels pour la bactérie Agrobacterium vitis.

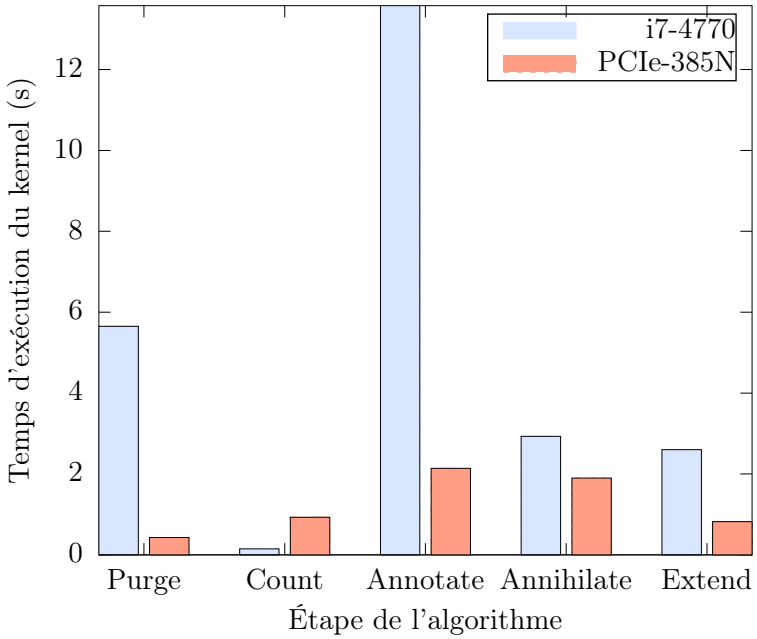

FIGURE 4.3 – Temps d'exécution des kernels pour la bactérie Ecoli.

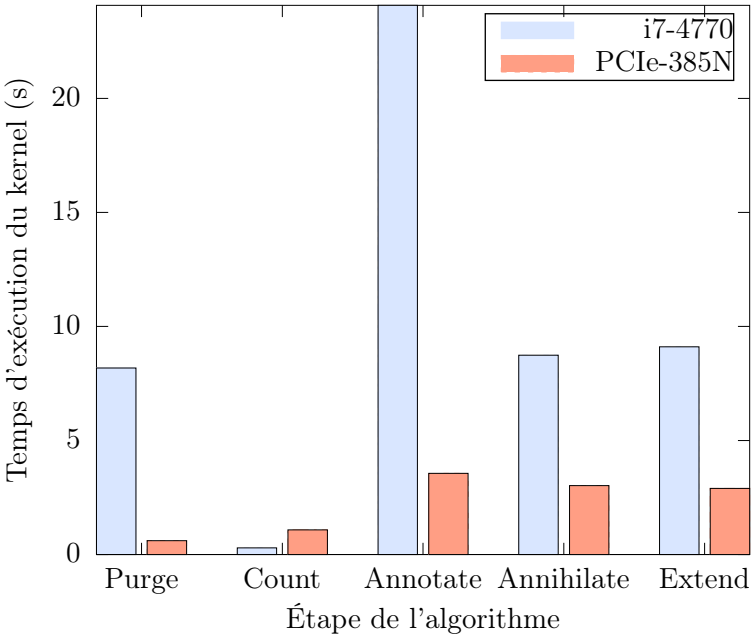

FIGURE 4.4 – Temps d'exécution des kernels pour la bactérie Acaryochloris marina.

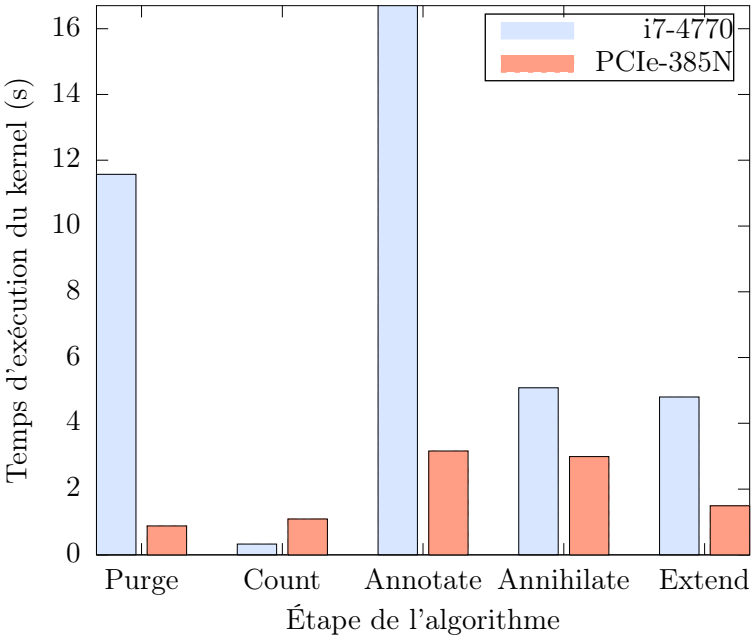

FIGURE 4.5 – Temps d'exécution des kernels pour la bactérie Actinoplanes.

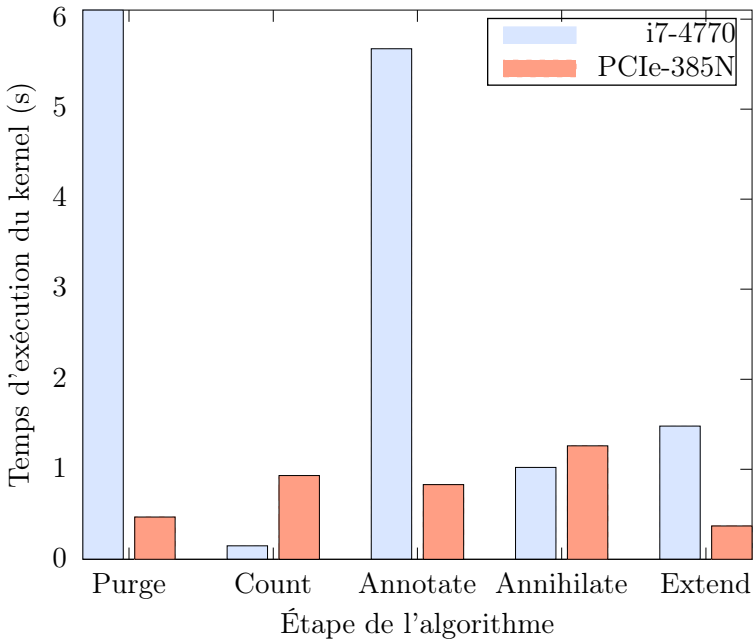

FIGURE 4.6 – Temps d'exécution des kernels pour la bactérie Salmonella enterica.

soit les étapes de dénombrage et de suppression des semences, favorisent le CPU au lieu du FPGA. À l'inverse, le kernel de suppression des arcs inutiles est complètement vectorisable et est celui qui est le plus favorable pour le FPGA. On voit aussi la performance du pipeline dans le FPGA par rapport au processeur dans les kernels d'annotation et d'extension qui ne sont pas vectorisables, mais très complexes. On entend qu'elles ont énormément d'instructions à exécuter pour chaque élément de travail, de par leur boucle principale qui fait plusieurs dizaines d'itérations.

4.3 Consommation énergétique

La mesure de consommation énergétique a été effectuée avec un appareil destiné à cette fin, nommé Kill-A-Watt et fabriqué par P3 International. La consommation à la source, soit après l'inefficacité de l'alimentation électrique, est obtenue lorsque les accélérateurs testés fonctionnent à plein régime. Pour la carte Nallatech PCIe-385N, nous obtenons une consommation augmentée de 28 W par rapport à l'état de repos, alors que le processeur central faisant office d'accélérateur induit une augmentation de 78 W pour un total de 111 W. Cette augmentation concorde avec l'enveloppe thermique de 84 W spécifiée par le fabricant, puisqu'elle est légèrement en dessous. Ces chiffres seront utilisés pour calculer l'énergie totale consommée pour la résolution des problèmes utilisés à des fins de comparaison dans ce chapitre. La formule permettant de calculer celle-ci est la suivante :

$$E = Pt \tag{4.1}$$

où P est la puissance requise en watts pour alimenter l'accélérateur, t est le temps requis en secondes et E est en joules. Par exemple, un accélérateur de 35 W fonctionnant pendant 30 secondes aura consommé 1050 joules. Les figures 4.7, 4.8, 4.9, 4.10, 4.11 et 4.12 présentent l'énergie consommée pour chacun des jeux de données.

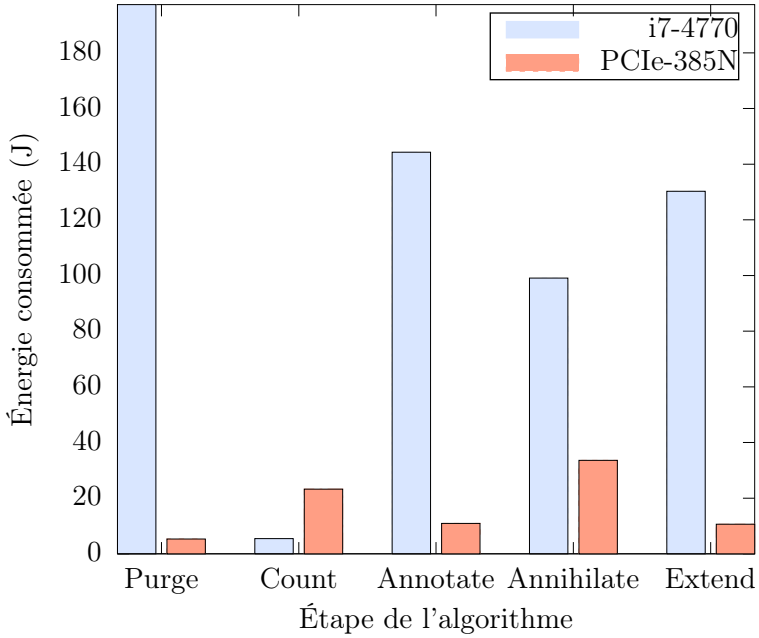

FIGURE 4.7 – Consommation énergétique des kernels pour la bactérie Acidianus hospitalis.

Finalement, le facteur d'économie d'énergie entre le processeur central et le FPGA peut être calculé comme suit :

$$F = \frac{t_{cpu}}{t_{fpga}} \frac{P_{cpu}}{P_{fpga}} \tag{4.2}$$

Le tableau 4.3 présente l'économie d'énergie pour les six jeux de données ainsi que globalement.

Pour les problèmes d'assemblage présentés ici, il n'y a qu'une étape, soit celle du décompte de la distribution de la couverture, qui ne vale pas la peine d'être exécutée sur FPGA. Cependant, cette étape se retrouve grandement éclipsée par les autres et au total, une somme d'énergie non-négligeable peut être économisée si un super-calculateur est basé en entier ou en partie sur des accélérateurs de type FPGA. Les coûts en électricité en seraient diminués, la complexité de la structure permettant le refroidissement efficace de ce dernier pourrait également être simplifiée et bien d'autres avantages pourraient potentiellement en découler.

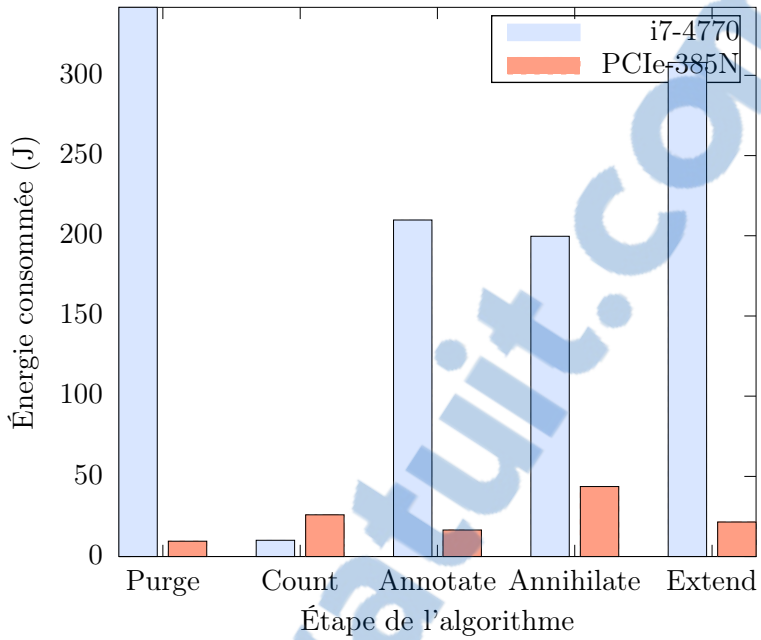

FIGURE 4.8 – Consommation énergétique des kernels pour la bactérie Agrobacterium vitis.

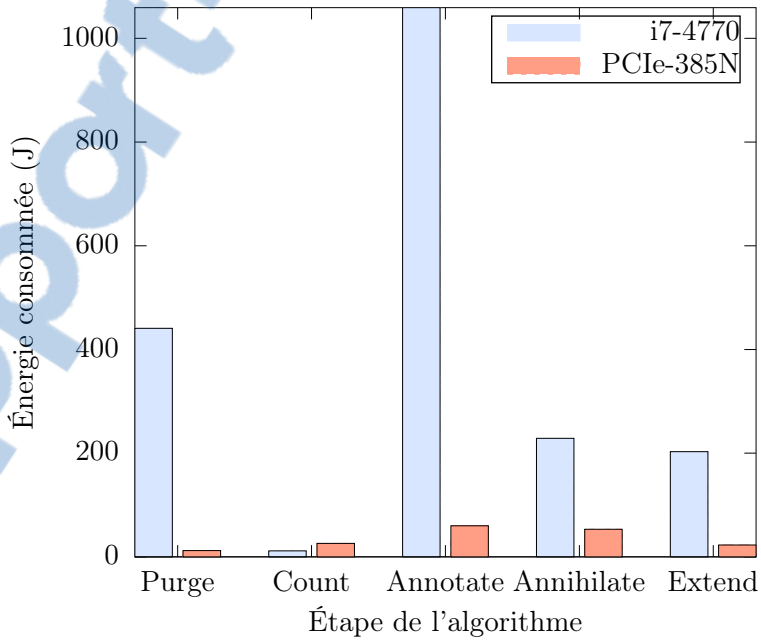

FIGURE 4.9 – Consommation énergétique des kernels pour la bactérie Ecoli.

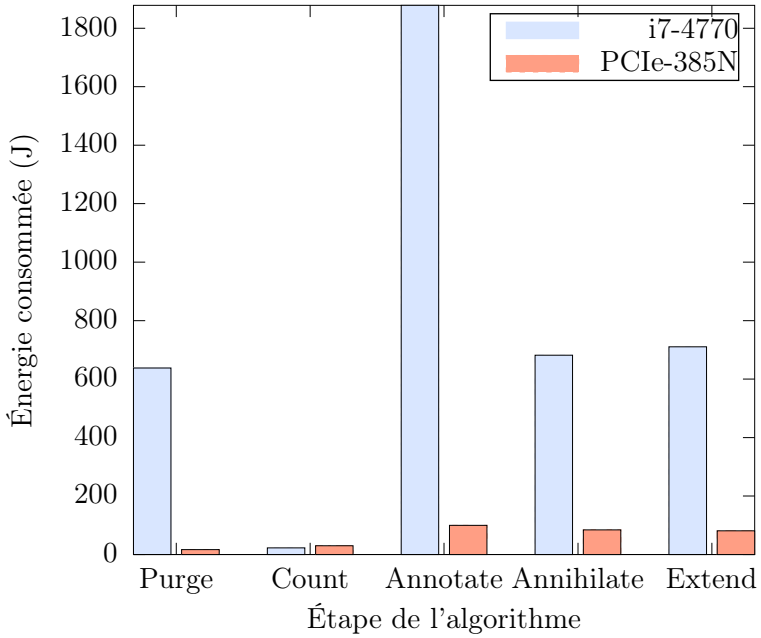

FIGURE 4.10 – Consommation énergétique des kernels pour la bactérie Acaryochloris marina.

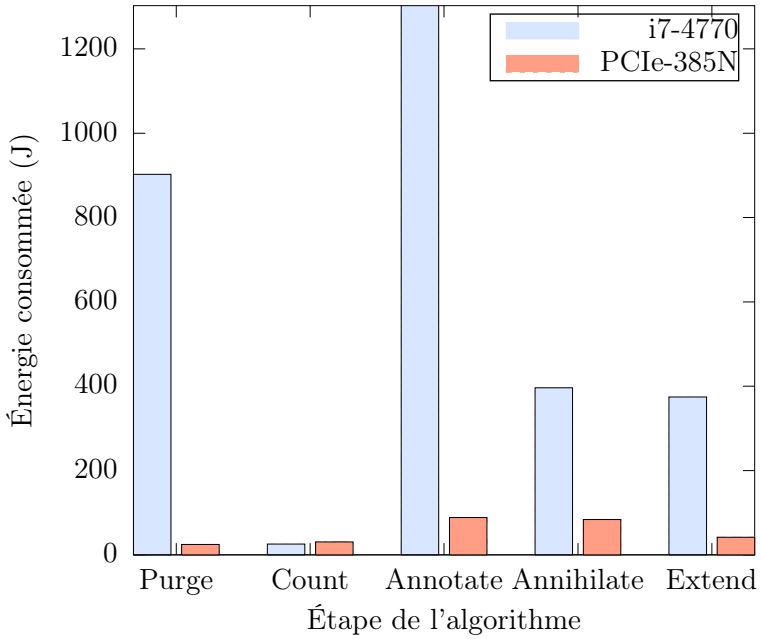

FIGURE 4.11 – Consommation énergétique des kernels pour la bactérie Actinoplanes.

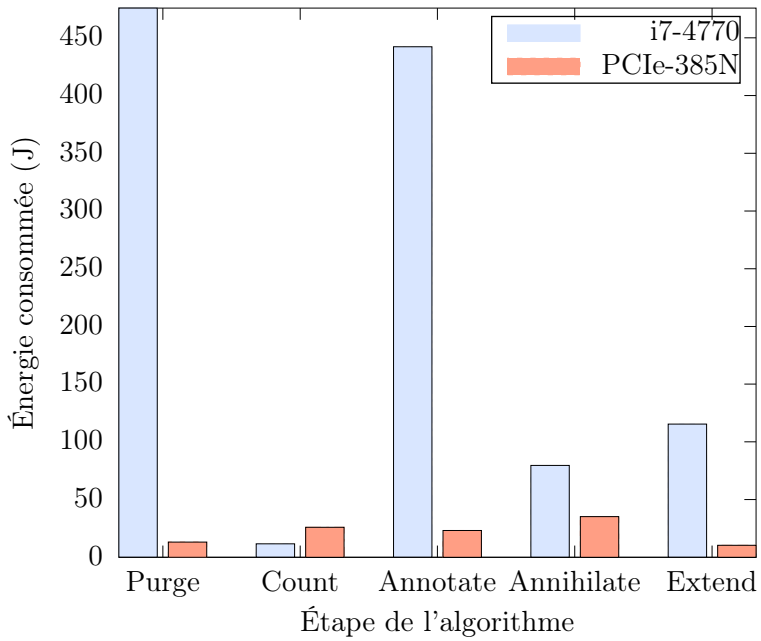

FIGURE 4.12 – Consommation énergétique des kernels pour la bactérie Salmonella enterica.

Tableau 4.3 – Facteur d'économie d'énergie en utilisant la carte d'expansion Nallatech PCIe-385N au lieu du Intel Core i7-4770.

	Purge	Count	Annotate	Anihilate	Extend	Total
Acidianus hospitalis	37.09398	0.23494	13.21429	2.94821	12.24248	6.88509
Agrobacterium vitis	35.96849	0.38940	12.70097	4.57143	14.29035	9.12172
Ecoli	36.60299	0.44931	17.67757	4.29586	8.83275	11.15629
Acaryochloris marina	36.15502	0.44931	19.03012	2.25510	11.14286	10.40674
Actinoplanes	37.35597	0.74802	18.84270	8.06197	8.75099	12.56938
Salmonella enterica	36.62581	0.84338	14.72197	4.73292	8.97411	11.15445
Moyenne	36.62732	0.43484	15.60372	3.82291	10.33907	9.83920

4.4 Effet du taux d'occupation du graphe

Parmi les assemblages présentés, tous avaient des taux d'occupation du graphe relativement différents. Le tableau 4.4 présente les grosseurs de graphe de puissance deux pour la raison expliquée en 3.3.3 ainsi que le taux d'occupation.

Afin de déterminer l'effet du taux d'occupation du graphe sur la performance, les durées d'exécution du kernel de purge peuvent être pris en compte. Cette étape, tel qu'expliqué en 1.2.2, visite les k-mers voisins à partir de chacun d'eux et procède à l'élimination si requis. Puisqu'il n'y a aucune autre structure de données d'impliquée comme dans les étapes suivantes, par exemple pour les lectures ou les annotations, le lien entre la performance et le taux d'occupa-

Tableau 4.4 – Grosseurs de graphes et taux d'occupation engendrés par les jeux de données testés.

	Grosseur du graphe	Taux d'occupation
Acidianus hospitalis	4194304	0.508
Agrobacterium vitis	8388608	0.441
Ecoli	8388608	0.537
Acaryochloris marina	8388608	0.580
Actinoplanes	16777216	0.377
Salmonella enterica	16777216	0.549

tion peut être observé directement. Dans les graphiques 4.13 et 4.14, le temps d'exécution par k-mer est représenté pour chaque taux d'occupation.

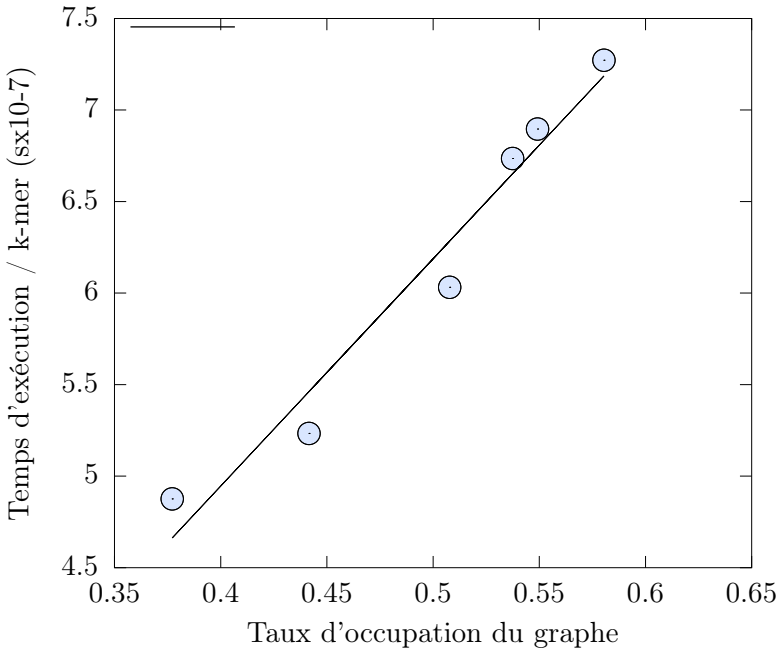

FIGURE 4.13 – Temps d'exécution moyen du kernel par k-mer vs. taux d'occupation du graphe pour l'exécution sur processeur central.

Pour chacun des accélérateurs, il devient évident que le taux d'occupation du graphe influence grandement le temps d'exécution des kernels. Plus le graphe est rempli, plus les probabilités que le k-mer recherché se trouve plus loin dans la chaîne de sondage augmentent et donc un délai d'accès supplémentaire est engendré plus régulièrement. Selon l'équation 3.3, pour une table de hachage à adressage ouvert utilisant une résolution de collisions pseudo-aléatoire, la relation entre le taux d'occupation et le nombre d'accès mémoire serait exponentiel. Cependant, avec des accès mémoire groupés et une table en deux dimensions tel qu'expliqué en 3.3.5, trois accès sur quatre sont épargnés. De plus, le réordonnancement des k-mers lors du remplis-

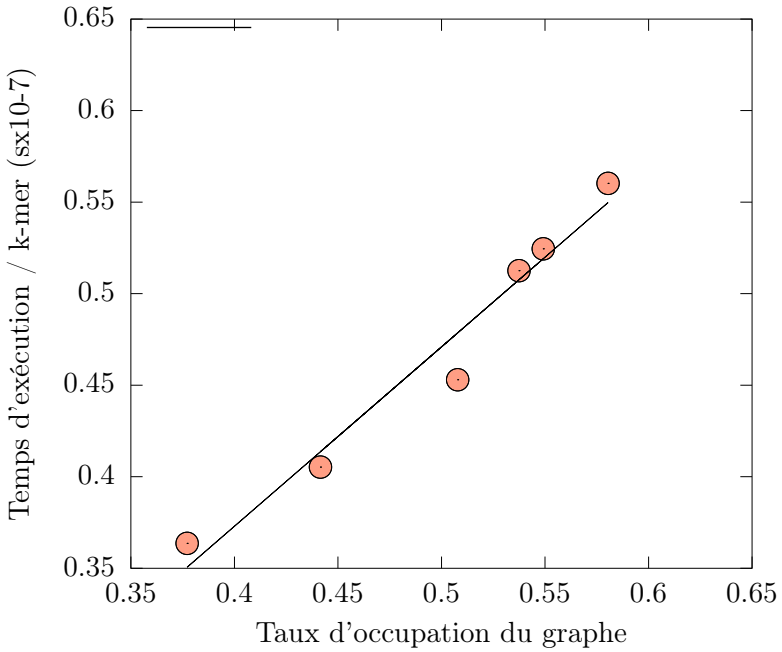

FIGURE 4.14 – Temps d'exécution moyen du kernel par k-mer vs. taux d'occupation du graphe pour l'exécution sur FPGA.

sage du graphe, tel qu'expliqué en 3.3.6, déplace la complexité $O(n^2)$ de l'algorithme lors du remplissage du graphe. Dans le cas des lectures pour cette implémentation simple, deux lignes du tableau de hachage maximum doivent être vérifiées. Cette modification à l'algorithme de base de la table de hachage a pour effet de camoufler la relation exponentielle et présenter un comportement plus linéaire. Pour le processeur central, la pente de la droite de régression est de 12.42 alors que pour le FPGA, elle est de 0.98. Cela démontre une fois de plus que la performance du processeur central est inférieure au FPGA.

4.5 Discussion

Au début du projet, les accélérateurs FPGA ayant 4 ou 8 Go de mémoire DDR3 étaient la norme. Aujourd'hui, il est possible de s'en procurer avec 32 Go de mémoire. Des problèmes d'assemblage beaucoup plus volumineux pourront être solutionnés. Le SDK Altera supporte les configurations avec de multiples FPGA, chacun sur son accélérateur ou non. Dans un poste de travail standard, il est possible d'ajouter jusqu'à huit accélérateurs pour un total de 256 Go de mémoire utilisable dans les kernels OpenCL. Cette quantité devrait être suffisante pour résoudre des problèmes d'assemblage tels le génôme humain.

De plus, certains accélérateurs sont maintenant disponibles avec de nouveaux types de mémoire tels le Hybrid Memory Cube (HMC) ou High-Bandwidth Memory (HBM). En plus de permettre une densité de mémoire beaucoup supérieure à la DDR3 et la DDR4, la bande

passante nettement meilleure et la latence plus faible ralentira moins le traitement hautement parallèle des plus récents FPGA.

Quant aux autres fabricants de FPGA, il y a aussi Xilinx qui supporte OpenCL. Le SDK de ce fabricant a été testé lors de ce projet, mais il s'est avéré qu'il ne pouvait compiler certains kernels ayant leur fonctionnement validé par les SDK d'autres vendeurs, soit celui de Altera, AMD et Intel. N'obtenant pas de messages d'erreurs mais seulement une trace d'appels et un support géré par une tierce-partie, il a été impossible de faire fonctionner OCLRay sur les FPGA Xilinx.

Conclusion

Ce mémoire a examiné l'utilisation d'accélérateurs basés sur des FPGA afin d'accomplir des problèmes d'assemblage d'ADN, ainsi que leur comparaison avec les processeurs centraux. Des généralisations ont été tirées quant aux résultats obtenus.

Les notions de base d'assemblage d'ADN ainsi que la provenance des données, soit le séquençage, ont été introduites. Les quelques algorithmes d'assemblage développés au fil du temps ont été présentés. Puis, celui qui a été à l'étude dans ce mémoire a été approfondi. Chaque étape a ainsi été décrite en détails. Par la suite, la méthode d'accélération adoptée pour ce projet ainsi que son fonctionnement et ses outils ont été discutés. Les quelques travaux ayant comme sujet l'accélération matérielle des algorithmes d'assemblage ont finalement été présentés pour clore la revue de littérature.

Les différents types d'accélérateurs compatibles avec OpenCL ont été présentés et comparés sur leurs points de différentiation. Puisque les FPGA étaient le but premier du projet, une attention particulière a été portée à leur égard ; les détails de la compilation, de l'optimisation ainsi que de l'évaluation de l'architecture ont été dévoilés.

Ensuite, la parallélisation originale avec l'interface de passage de message a été expliquée. Il était par la suite question des modifications apportées à l'algorithme pour rendre la parallélisation efficace avec OpenCL sur un seul accélérateur. On note la modification des débuts des lectures au lieu du stockage d'un pointeur d'annotation, l'utilisation de kernels précompilés avec des longueurs de k-mers fixes, l'utilisation d'une grosseur de graphe de puissance 2, la résolution des collisions par sondage pseudo-aléatoire, l'élargissement de la table de hachage dans une seconde dimension et l'imposition d'un maximum de collisions lors d'une recherche. Puis, pour chaque kernel, le partionnement du problème est expliqué.

Au chapitre de résultats, le temps d'exécution et la consommation énergétique pour chaque étape de l'algorithme et chaque accélérateur testé sont présentés. La première contribution importante de ce mémoire consiste en la démonstration que les accélérateurs basés sur des circuits logiques programmables peuvent performer mieux que des processeurs multi-coeurs ayant une logique interne fixe, tout en consommant drastiquement moins d'énergie.

L'effet du taux d'occupation du graphe sur le temps d'exécution de l'algorithme est aussi

analysé. Suite à un réordonnancement des k-mers, il est intéressant de voir que la complexité algorithmique de recherche d'un sommet dans le graphe évolue plus selon $O(n)$ que $O(n^2)$. Cela constitue une autre contribution de ce mémoire. Il serait intéressant de faire plus de tests suite à cette preuve de concept dans un cadre plus général et non dans un cadre d'accélération matérielle.

Quant aux publications, des résultats préliminaires avaient été présentés lors de la conférence EMBC'15 [22], puis un article de revue plus complet a été soumis et est présentement en attente de revue.

Aujourd'hui, si nous voulions pousser le projet à de plus hauts niveaux, il serait simple de recompiler les kernels pour les tous derniers modèles de FPGA qui possèdent plus de logique pour former plus d'éléments de traitement. Aussi, il y a sur le marché des cartes ayant jusqu'à quatre FPGA à leur bord, ce qui ferait un total de 32 FPGA. En utilisant le temps d'exécution moyen normalisé présenté au chapitre 4, il faudrait un total de 320 processeurs comme celui testé pour égaliser un tel poste de travail ! Cependant, pour les processeurs centraux de type x86, il est possible d'en utiliser un maximum de quatre avant de tomber dans la parallélisation passant par un réseau, comme celle originale. Hélas, celle-ci engendre des latences considérables.

Les conclusions de ce projet pourraient inspirer d'autres travaux sur des sujets semblables. Pour l'avenir, il serait intéressant d'exploiter les capacités d'OpenCL 2.0. Quoique les implémentations pour FPGA ne sont pas conformes à cette révision, elles possèdent plusieurs de ses fonctionnalités avancées telles que les canaux de communication inter-kernels. Des optimisations très intéressantes pourraient en découler.

Au niveau matériel, l'utilisation d'un SoC avec mémoire unifiée serait à explorer. Les toutes dernières puces Altera possèdent plusieurs coeurs ARM 64-bit embarqués qui permettraient de bien gérer les opérations sérielles restantes de l'algorithme sans occasionner de longs transferts de mémoire. Il en va de même pour les APU d'AMD. Plus globalement, il serait aussi intéressant d'explorer l'avenue des stations de travail et même des super-ordinateurs basés sur ces puces.

Bibliographie

[1] ALTERA. Altera sdk for OpenCL : Best practices guide. http://www.altera.com/literature/hb/opencl-sdk/aocl_optimization_guide.pdf, 2014.

[2] ALTERA. Altera SDK for OpenCL : Programming guide. http://www.altera.com/literature/hb/opencl-sdk/aocl_programming_guide.pdf, 2014.

[3] ALTERA. Implementing FPGA design with the OpenCL standard. http://www.altera.com/literature/wp/wp-01173-opencl.pdf, 2014.

[4] Sébastien Boisvert, François Laviolette, and Jacques Corbeil. Ray : Simultaneous assembly of reads from a mix of high-throughput sequencing technologies. *Journal of Computational Biology*, 17(11), 2010.

[5] Sébastien Boisvert, François Laviolette, and Jacques Corbeil. Assemblage adaptatif de génomes et de méta-génomes par passage de messages. http://www.theses.ulaval.ca/2014/30598/30598.pdf, 2013.

[6] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf, May 1994.

[7] Mark J. Chaisson and Pavel A. Pevzner. Short read fragment assembly of bacterial genomes. *Genome Research*, 18(2), 2008.

[8] Nicolaas Govert de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49 :758–764, 1946.

[9] Dmitry Denisenko. Lucas kanade optical flow – from C to OpenCL on CV SoC. In *CMC Microsystems Altera Training on OpenCL*, 2014.

[10] Paul Flicek and Ewan Birney. Sense from sequence reads : methods for alignment and assembly. *Nat. Methods*, 6(11 Suppl) :S6–S12, November 2009. [DOI :10.1038/nmeth.1376] [PubMed :19844229].

[11] Gaston H. Gonnet. Expected length of the longest probe sequence in hash code searching. *Department of Computer Science, University of Waterloo*, December 1978.

[12] Torbjörn Granlund. Instruction latencies and throughput for AMD and Intel x86 processors. https://gmplib.org/~tege/x86-timing.pdf, July 2014.

[13] KHRONOS GROUP. OpenCL 1.2 reference pages. http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/, November 2011.

[14] Advanced Micro Devices Inc. OpenCL programming guide. http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf, November 2013.

[15] Konstantinos Krampis. Genome assembly algorithm and software. http://www.slideshare.net/agbiotec/overview-of-genome-assembly-algorithms, November 2011.

[16] Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. MEGAHIT : An ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph. *HKU-BGI Bioinformatics Algorithms Research Laboratory*, January 2014.

[17] Ruiqiang Li, Hongmei Zhu, Jue Ruan, and al. De novo assembly of human genomes with massively parallel short read assembly. *Genome Research*, décembre 2009.

[18] Chi-Man Liu, Ruibang Luo, and Tak-Wah Lam. GPU-accelerated BWT construction for large collection of short reads. *HKU-BGI Bioinformatics Algorithms and Core Technology Research Center*, January 2014.

[19] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. CUSHAW : a CUDA compatible short read aligner to large genomes based on the burrows-wheeler transform. http://www.nvidia.com/content/tesla/pdf/CUSHAW-CUDA-compatible-short-read-aligner-to-large-genomes.pdf, May 2012.

[20] Encyclopedia of Mathematics. Greedy algorithm. http://www.encyclopediaofmath.org/index.php?title=Greedy_algorithm&oldid=11679, February 2011.

[21] Andy Oram and Greg Wilson. *Beautiful Code*. O'Reilly Media, 2007.

[22] Carl Poirier, Benoit Gosselin, and Paul Fortier. DNA assembly with de bruijn graphs on FPGA. *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 6489–6492, Aug 2015.

[23] Michael C. Schatz, Cole Trapnell, Arthur L Delcher, and Amitabh Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 12 2007.

[24] Sean O. Settle. High-performance dynamic programming on FPGAs with OpenCL. http://ieee-hpec.org/2013/index_htm_files/29-High-performance-Settle-2876089.pdf, 2013.

[25] Craig Silverstein. Implementation of sparse_hash_map, dense_hash_map, and sparsetable. *Journal of Computational Biology*, January 2005.

[26] Jared T. Simpson, Kim Wong, Shaun D. Jackman, Jacqueline E. Schein, Steven J. M. Jones, and Inancx Birol. ABySS : A parallel assembler for short read sequence data. *Genome Research*, February 2009.

[27] TOP500.org. List statistics. http://www.top500.org/statistics/list/, 2015.

[28] Neil Trevett. OpenCL introduction. In *SIGGRAPHASIA2013*, 2013.

[29] Thomas Wang. Integer hash function. http://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm, 2007.

[30] Xilinx. How to get more than two orders of magnitude better power/performance from key-value stores using FPGA. In *IEEE Communications Society Tutorials*, 2014.

[31] Daniel R. Zerbino and Ewan Birney. Velvet : Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, mars 2008.

[32] Victoria Zhislina. Why has CPU frequency ceased to grow ? https://software.intel.com/en-us/blogs/2014/02/19/why-has-cpu-frequency-ceased-to-grow, 2014.