Parallelization on graphic hardware : contributions to RNA folding and sequence alignment

Guillaume Rizk

N° d'ordre : 4267 **ANNÉE 2010**

THÈSE / UNIVERSITÉ DE RENNES 1

sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Guillaume Rizk

préparée à l'unité de recherche IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires

Parallelization on graphic hardware: contributions to RNA folding and sequence alignment

Thèse soutenue à Rennes le 12 janvier 2011

devant le jury composé de :

Denis BARTHOU
rapporteur
Thierry LECROQ
rapporteur
Sanjay RAJOPADHYE
examinateur
Gregory KUCHEROV
examinateur
Jean-Jacques CODANI
examinateur
Dominique LAVENIER
directeur de thèse

Abstract

La bioinformatique nécessite l'analyse de grandes quantités de données. Avec l'apparition de nouvelles technologies permettant un séquençage à haut débit à bas coût, la puissance de calcul requise pour traiter les données a énormément augmenté. De plus, la puissance des CPU évolue comparativement moins vite, aboutissant à des temps d'exécution de plus en plus longs. En revanche, les processeurs graphiques (GPU) sont maintenant suffisamment programmables pour permettre de réaliser des calculs non graphiques, fournissant ainsi une puissance de calcul élevée bon marché. Cette thèse examine la possibilité d'utiliser les GPU pour des applications de bioinformatique. Dans un premier temps, ce travail s'intéresse au calcul des structures secondaires d'ARN. Ce problème est en général calculé par programmation dynamique, avec un algorithme qui pose de sérieux problèmes pour un code GPU: des relations de récurrence complexes et un accès mémoire compliqué. Nous introduisons une nouvelle implémentation tuilée qui fait apparaitre une bonne localité des accès mémoire, permettant ainsi un programme GPU très efficace. Cette modification permet également de vectoriser le code CPU et donc de faire une comparaison honnête des performances entre GPU et CPU. Dans un deuxième temps, ce travail aborde le problème d'alignements de séquences sur un génome de référence. Nous présentons une parallélisation GPU d'une méthode utilisant une indexation par graines. L' implémentation sur GPU n'étant pas efficace, nous nous tournons vers le développement d'une version CPU. Notre contribution principale est le développement d'un nouvel algorithme éliminant rapidement les nombreux alignements potentiels, basé sur le précalcul de portions de la matrice de programmation dynamique. Ce nouvel algorithme a conduit au développement d'un nouveau programme d'alignement très efficace. Notre travail fournit l'exemple de deux problèmes différents dont seulement un a pu être efficacement parallélisé sur GPU. Ces deux expériences nous permettent d'évaluer l'efficacité des GPU et leur place en bioinformatique.

Keywords : GPGPU, Next Generation Sequencing, mapping, RNA folding.

Abstract

Bioinformatics require the analysis of large amounts of data. With the recent advent of next generation sequencing technologies generating data at a cheap cost, the computational power needed has increased dramatically. However, CPU performance is not keeping up with this evolution, leading to unreasonable execution times and storage issues. On the other hand, Graphic Processing Units (GPU) are now programmable beyond simple graphic computations, providing high performance for general purpose applications at a low cost. This thesis explores the usage of GPUs for bioinformatics applications. First, this work focuses on the computation of secondary structures of RNA sequences. It is traditionally conducted with a dynamic programming algorithm, which poses significant challenges for a GPU implementation: complex data dependency scheme and memory access pattern. We introduce a new tiled implementation providing good data locality and therefore very efficient GPU code. We note that our algorithmic modification also enables tiling and subsequent vectorization of the CPU program, allowing us to conduct a fair CPU-GPU comparison. Secondly, this works addresses the short sequence alignment problem. We present an attempt at GPU parallelization using the seed-and-extend paradigm. Since this attempt is unsuccessful, we then focus on the development of a program running on CPU. Our main contribution is the development of a new algorithm filtering candidate alignment locations quickly, based on the pre computation of tiles of the dynamic programming matrix. This new algorithm proved to be in fact more effective on a sequential CPU program and lead to an efficient new CPU aligner. Our work provides the example of both successful an unsuccessful attempts at GPU parallelization. These two points of view allow us to evaluate GPUs efficiency and the role they can play in bioinformatics.

Keywords : GPGPU, Next Generation Sequencing, mapping, RNA folding.

Résumé étendu

Contexte

La bio-informatique est le développement de méthodes informatiques permettant l'analyse de données biologiques. Les champs d'applications concernent par exemple l'analyse des séquences d'ADN qui constituent un génome, ou la comparaison et la prédiction des structures des protéines ou des séquences d'ARN. De plus, la quantité de données à analyser a augmenté de manière exponentielle au cours des dernière années.

En ce qui concerne le séquençage de l'ADN, les progrès ont été très rapides. la structure en double hélice a été découverte en 1953 par Watson et Crick. En 1977, le premier génome entier d'un organisme fut séquencé, celui du bacteriophage ϕ X 174 composé de 5386 bases. Le génome humain, composé de 3 milliards de paires de bases, a été séquencé en 2000 à un coût de plusieurs milliards de dollars. De nos jours, plusieurs milliards de paires de bases peuvent être séquencées pour quelques milliers de dollars.

En conséquence, la quantité de données à analyser est de plus en plus grande. D'un autre côté, la puissance de calcul des ordinateurs augmente, mais moins rapidement. Cela conduit à un allongement important des temps de traitement. Une solution pour réduire les temps de calcul est de paralléliser les calculs. La première solution est l'utilisation de CPU multi-coeurs ainsi que des clusters de CPU. Bien qu'efficace, cette solution est coûteuse. Une autre solution est l'utilisation de matériel haute performance spécialisé. Par exemple les FPGAs peuvent fournir une puissance de calcul très élevée, mais sont également très coûteux.

Une autre solution vient de l'utilisation des processeurs graphiques (GPU). Les GPUs sont des processeurs spécialement conçus pour accélérer l'affichage de graphismes complexes sur un écran. C'est une tâche avec beaucoup de parallélisme de données: des traitements identiques doivent être appliqués à tous les pixels et objets géométriques qui constituent l'image. Les GPU ont été conçus pour traiter efficacement ce type de calcul: ils comportent de nombreuses unités de calcul capables d'appliquer le même calcul à différentes données. A l'origine limités à des calculs graphiques de base, les GPU ont graduellement acquis la capacité d'effectuer des traitements de plus en plus complexes, afin que les programmeurs graphiques puissent développer des effets plus réalistes. Cette programmabilité accrue est arrivée à un point ou des calculs non graphiques ont aussi pu être exécutés sur des cartes graphiques, donnant naissance à un nouveau champ d'application, le calcul générique sur cartes graphiques (GPGPU en anglais). NVIDIA a introduit en 2006 le langage de programmation CUDA, spécialement conçu pour faciliter le développement de programmes GPGPU.

Utiliser les GPUs pour exécuter des programmes non graphiques a plusieurs avantages. D'abord, une forte concurrence tirée par l'industrie du jeu vidéo a conduit à des GPUs bon marché, avec une puissance de calcul en constante augmentation. Tous les ordinateurs peuvent être équipés d'un GPU haut de gamme, augmentant considérablement sa puissance totale de calcul à faible coût.

Ensuite, les GPU sont naturellement bien adaptés pour tout problème avec un fort parallélisme de données, apparaissant dans de nombreux domaines de recherche scientifique. Par exemple en imagerie médicale, pour des simulations de dynamique des fluides, en finance, traitement du signal, ou encore pour la cryptanalyse. La bio-informatique comporte aussi beaucoup de parallélisme de données: l'analyse de données biologiques nécessite souvent d'appliquer le même calcul sur des données différentes. A priori, il semble donc naturel d'utiliser les GPU pour accélérer les programmes utilisés en bio-informatique.

Objectifs

L'objectif de cette thèse est d'évaluer les capacités des GPU dans des applications de bio-informatique typiques, et de comparer les résultats obtenus avec une parallélisation plus traditionnelle sur CPU multi-coeurs. Pour cela notre étude se penche sur deux problèmes distincts, à la fois bien représentatifs de ce qui se fait en bio-informatique, et différents en termes de méthodes de calculs utilisés pour les résoudre. Le premier problème est l'étude de la prédiction de structure secondaire d'ARN. L'algorithme utilisé est un calcul intensif qui se prête *a priori* bien à une parallélisation sur GPU. Le second problème porte sur l'alignement de séquences de petite taille, générées par les nouvelles technologies de séquençage, sur un génome de référence. Le calcul est moins intensif et nécessite au contraire plus d'accès à la mémoire, en utilisant une structure de données élaborée. Ces deux problèmes représentent donc un spectre varié de ce que l'on peut trouver en bio-informatique, et leur étude permet de répondre à la problématique de cette thèse: les algorithmes utilisés en bio-informatique peuvent-ils être efficacement implémentés sur GPU ?

L'évaluation des GPU doit s'effectuer sur plusieurs niveaux. Le premier point important est le temps nécessaire au développement d'applications sur GPU. Les GPUs sont a priori plus simples à programmer que des FPGAs, mais plus difficiles que des CPU. Toutefois cela nécessite d'être confirmé en pratique. Deuxièmement, les temps d'exécutions doivent être mesurés et comparés à un code tournant sur CPU. Pour une comparaison honnête, il faut comparer le code GPU à un programme bien optimisé sur CPU, utilisant tous les coeurs disponibles ainsi que les instructions vectorielles qui augmentent significativement la puissance des CPUs. Enfin, les coûts d'achat et de fonctionnement des GPUs doivent aussi être évalués.

Chapitre 1 : Architecture

la première étape pour savoir si les GPUs peuvent accélérer des programmes de bio-informatique est de connaître leur fonctionnement et l'architecture en détail, de déterminer les points faibles et points forts et de les comparer avec l'architecture CPU.

1.1 Introduction

Le temps d'exécution total d'un programme dépend de plusieurs facteurs, communs aux architectures CPU et GPU. Nous pouvons distinguer trois caractéristiques principales: la puissance de calcul brute, la bande passante mémoire, et la latence.
La puissance brute détermine le nombre d'opérations qui peuvent être exécutées par unité de temps. Elle est généralement fournie par les constructeurs en nombre d'opérations par cycles d'horloge. La multiplication de ce chiffre par la fréquence fournit le nombre d'opérations par secondes, usuellement défini en milliards d'opérations flottantes par secondes, *GFLOPS* en anglais. Par exemple la puissance maximale d'un CPU Xeon X5482 est de 102.4 GFLOPS pour des opérations en 32 bits, en utilisant les instructions SSE et tous les coeurs disponibles. Pour une carte graphique Tesla C1060, la puissance maximale théorique est de 624 GFLOPS. Cela signifie que l'on peut s'attendre à facteur d'accélération de x6 en utilisant un GPU. Toutefois, le facteur d'accélération réel dépend bien évidemment de la fraction de cette puissance maximale que l'on arrive à exploiter en pratique.
Le deuxième facteur est la bande passante mémoire, *i.e.* la quantité de données transférée par unité de temps, exprimée en Goctets/sec (Go/sec). Schématiquement, les données sont transférées de la mémoire pour alimenter le processeur, puis dans l'autre sens pour stocker les résultats. Pour une Tesla C1060 et un CPU Xeon 5500, la bande passante maximale est respectivement de 102.4 Go/sec et 32.0 Go/sec. Le rapport entre la bande passante et la puissance brute détermine l'intensité arithmétique nécessaire pour exploiter au maximum les ressources disponibles. Par exemple pour la Tesla C1060, un programme doit effectuer au moins 24 opérations flottantes par accès mémoire pour atteindre la puissance de calcul maximale. Toutefois, si le jeu de données d'un algorithme tient dans la mémoire présente sur la puce, les données peuvent être chargées une seule fois de la mémoire principale et réutilisées plusieurs fois. Dans ce cas, la bande passante mémoire est moins souvent une limitation.
Le troisième point est la latence, le délai entre le lancement d'une opération et le moment ou le résultat est disponible. La latence concerne à la fois les les instructions processeurs, et les transactions mémoires. Dans un système purement séquentiel la latence est un problème important. C'est pourquoi les CPU intègrent de nombreux niveaux de mémoire cache pour diminuer la latence d'accès à la mémoire, et des systèmes complexes tel que l'exécution *out-of-order* des instructions. D'un autre côté, le parallélisme de tâches permet aussi de combattre efficacement les problèmes de latence: pendant qu'une tâche est en attente, une autre tâche est exécutée, les unités de calcul ne sont donc pas bloquées. C'est ce qui est utilisé par les GPU, et qui leur permettent de se passer de grandes quantités de mémoire sur la puce.

1.2 Comparaison CPU/ GPU

Nos expérimentations ont été effectuées sur un système 8 coeurs composé de deux quad-core Xeon E5430 fonctionnant à 2.66GHz. Chaque Xeon contient 32KB de cache L1 par coeur, et 6MB de cache L2 commun à deux coeurs. Ce CPU supporte le parallélisme de tâches, traditionnellement implémenté avec les threads POSIX ou avec l'API OpenMP. Il supporte aussi le parallélisme de données avec les instructions vectorielles SSE.

Nous avons utilisé l'architecture GPU CUDA de Nvidia, qui combine hautes performances et facilité de programmation avec le langage *C for CUDA*. En résumé, les GPUs échangent le grand nombre de transistors dédiés à la mémoire cache et aux mécanismes de contrôle de flux présent sur les CPUs contre plus d'unités de calculs. A la place, les GPU comptent sur le multithreading pour masquer la latence. Un programme GPU est un *kernel*, appelé sur un ensemble de trheads, répartis en un grille de blocs. les blocs sont subdivisés en *warps* de 32 threads qui s'exécutent en SIMD. Les threads d'un bloc peuvent être synchronisés et partager un petite mémoire rapide partagée de 16 Ko.
Quelques points particuliers sont: (i) la bande passante maximale n'est atteinte que si les différents accès mémoire sont regroupés, ce qui nécessite que chaque thread accède à des données consécutives en mémoire, (ii) chaque multiprocesseur de la carte peut gérer au maximum 32 warps à la fois. L'*occupation* mesure le pourcentage de ce maximum qui est utilisé en pratique. Une *occupation* plus élevée permet en général de mieux utiliser les ressources de la carte, mais nécessite d'utiliser peu de registres par thread. (iii) La mémoire partagée est un point crucial pour obtenir une bonne performance, elle permet de stocker sur la puce des données susceptibles d'être utilisées par différents threads. Elle est gérée entièrement manuellement par le programmeur.

Les architectures CPU et GPU sont assez différentes dans le sens où le CPU contient beaucoup d'éléments dont le but est d'exécuter rapidement une tâche séquentielle, alors que le GPU est très lent s'il exécute une seule tâche. Comme le GPU compte sur le multi-threading pour ses performances, il requiert des milliers de threads pour fonctionner efficacement, contrairement au CPU qui est déjà très rapide avec un seul thread. Toutefois, il y aussi un certain nombre de similarités: un GPU est un ensemble de multiprocesseurs, chacun contenant une unité SIMD, un CPU est un ensemble de coeurs, chacun contenant également un unité SIMD. De plus, les architectures semblent converger: les GPU évoluent vers une programmation plus souple, tandis que les CPUs évoluent vers plus de parallélisme. Par conséquent, les GPUs ne font qu'anticiper ce qu'il sera nécessaire de faire pour les CPU de demain.

Chapitre 2 : Prédiction de structure secondaire d'ARN

La prédiction des structures secondaires d'ARN est un problème étudié depuis longtemps par la communauté scientifique, et est utilisée régulièrement dans des pipelines de bio-informatique. C'est un algorithme de calcul intensif, dont le temps d'exécution pose problème pour des séquences de grande taille. Une parallélisation GPU n'a jusqu'à présent à notre connaissance jamais été tentée.

2.1 Introduction

La molécule d'ADN est plus connue sous sa forme double brin qui forme une double hélice. Toutefois, la molécule d'ARN se trouve plus généralement en simple brin dans la cellule. Elle est composée de 4 nucléotides, aussi appelées *bases*: adenine (A), cytosine (C), guanine (G) et uracil (U). Deux nucléotides peuvent se lier formant ainsi une paire de bases, selon la règle de complémentarité de Watson et Crick: A avec U, G avec C, mais aussi la paire "bancale" plus instable G avec U. Les liaisons intra-moléculaires de ce simple brin d'ARN forcent la chaîne à se replier sur elle même en un ensemble de boucles de différent types : des épingles à cheveux, des hernies, des boucles internes ou multiples. Cette conformation 2D est appelée la structure secondaire, dont la connaissance est nécessaire à de nombreux problèmes de bio-informatique. Les algorithmes calculant cette structure ont une complexité en $\mathcal{O}(n^3)$, ce qui signifie que le temps d'exécution devient rapidement problématique pour des grandes séquences. L'algorithme le plus connu est celui de Zuker introduit en 1981, qui utilise un modèle thermodynamique précis pour mesurer la stabilité des différents éléments de la structure [85]. Il est encore largement utilisé aujourd'hui et disponible dans deux packages, ViennaRNA [29] et Unafold [56]. Notre objectif est de développer une implémentation GPU efficace de l'algorithme donnant les mêmes résultats que Unafold.

En résumé, l'algorithme de prédiction de structure fonctionne de la manière suivante. Il détermine la structure la plus stable, c'est à dire celle qui maximise un schéma de score fixé, ou plutôt, qui minimise l'énergie libre totale de la molécule d'ARN. L'énergie de la séquence entière se décompose facilement en la somme des énergies de ses sous-structures. Un modèle thermodynamique définit l'énergie des structures élémentaires en fonction de leur taille, type, et composition en nucléotides. Le problème est un problème d'optimisation: parmi toutes les structures possible, il s'agit de trouver celle d'énergie minimum. Ce problème est résolu par programmation dynamique: le problème entier est décomposé en sous-problèmes plus faciles à résoudre. Le problème est résolu pour des sous-séquences de la séquence principale, dont les solutions permettent ensuite de reconstituer les solutions pour des problèmes de plus en plus grands, jusqu'à obtenir la solution pour la séquence entière. L'algorithme introduit les 3 quantités $QP_{i,j}$,$QM_{i,j}$ et $QS_{i,j}$ qui représentent les solutions pour les sous-séquences $i \dots j$ dans trois cas de figures différents. $QP_{i,j}$ représente le score d'un séquence dont les extrémités sont appariées, $QM_{i,j}$ une séquence contenant au moins deux paires de bases, et $QS_{i,j}$ une séquence contenant au moins une paire de bases. Les relations de récurrence suivantes permettent de reconstruire de proche en proche la solution pour des séquences de plus en plus grandes :

$$QP_{i,j} = \begin{cases} \min \left\{ \begin{array}{l} Eh(i,j) \\ Es(i,j) + QP_{i+1,j-1} \\ \min\limits_{\{k,l|i<k,l<j\}} Ei(i,j,k,l) + QP_{k,l} \\ QM_{i+1,j-1} \end{array} \right\} & i \cdot j \textit{ est autorisée} \\ \infty & \textit{si paire } i \cdot j \textit{ interdite} \end{cases} \tag{2.1}$$

$$QM_{i,j} = \min_{i<k<j} (QS_{i,k} + QS_{k+1,j}) \qquad (2.2)$$

$$QS_{i,j} = \min\{QM_{i,j}, \min(QS_{i+1,j}, QS_{i,j-1}), QP_{i,j}\} \qquad (2.3)$$

Les trois quantités $QP_{i,j}$,$QM_{i,j}$ et $QS_{i,j}$ définissent la matrice de programmation dynamique pour i, j variant entre 1 et n. C'est une matrice symétrique: on a $QP_{i,j} = QP_{j,i}$. Les équations de récurrence impliquent un schéma de dépendance des données et donc un certain ordre de calcul de cette matrice. En séquentiel, les calculs s'effectuent colonnes après colonnes de la gauche vers la droite, et chaque colonne de bas en haut. L'équation qui calcule $QM_{i,j}$ est en $\mathcal{O}(n^3)$, les deux autres en $\mathcal{O}(n^2)$. Une mesure du temps passé dans les différentes parties du code montre que pour des séquences de taille 8000, plus de 80% du temps total est passé dans le calcul de QM. En revanche, pour des séquences de taille 1000, la répartition est respectivement de 30%,40% et 20% pour QM,QP et QS. Pour avoir une bonne accélération globale pour différentes taille de séquences, il faut donc accélérer efficacement ces 3 parties de l'algorithme. Pour des grandes séquences, l'accélération globale obtenue avec une parallélisation GPU va donc approcher l'accélération de la partie en $\mathcal{O}(n^3)$, qui calcule QM.

2.2 Principe de parallélisation

Les équations de récurrence impliquent une parallélisation a priori évidente des calculs: les différentes cases qui composent une diagonale $i - j = constante$ peuvent être calculées en parallèle. Trois kernels sont conçus pour le calcul de $QP_{i,j}, QM_{i,j}$ et $QS_{i,j}$, correspondant aux 3 équations de récurrences. Chaque kernel calcule une diagonale, avec un thread par case. La matrice complète est ensuite calculée séquentiellement avec une boucle sur toutes les diagonales. La synchronisation nécessaire entre chaque diagonale s'effectue donc naturellement entre l'exécution des différents kernels.

La partie la plus problématique concerne la parallélisation de la partie en $\mathcal{O}(n^3)$ qui calcule QM. La parallélisation naïve par diagonale requiert en effet beaucoup de bande passante, car chaque case de la matrice nécessite l'accès à $\mathcal{O}(n)$ données pour $\mathcal{O}(n)$ calculs, et il n'y a pas de partage possible des accès entre les différentes cases d'une diagonale. En théorie, il pourrait y avoir une réutilisation des données, car une valeur $QS(i, k)$ va être utilisée pour le calcul des différentes cases $QM(i, j)$ avec $j \in [k; n]$. Cependant, le schéma de dépendance des données rend impossible d'en tirer avantage, car l'ensemble des cellules $QM(i, j)$ $j \in [k; n]$ sont situées sur différentes diagonales et donc calculées dans des appels kernels distincts.
Nous avons développé une implémentation pavée qui fait apparaitre une bonne localité mémoire. L'observation clé est que le calcul d'une case est une minimisation sur $i < k < j$, qui peut être découpée en plusieurs parties. Notre approche est inspirée de l'implémentation systolique du problème de parenthésage optimal développé par Guibas-Kung-Thompson[27, 65]. L'idée est de décomposer et réordonner la réduction afin de faire apparaitre une bonne localité mémoire et donc un code efficace sur GPU. Considérons que les cases jusqu'à une certaine diagonale d_0 sont déjà calculées, c'est à dire toutes les cellules (i, j) telles que $j < i + d_0$. Pour (i_1, j_1) fixé en dehors de la zone

connue, avec $j_1 = i_1 + d_0 + T, T > 0$, le calcul entier $\min\limits_{i_1<k<j_1} (QS_{i_1,k} + QS_{k+1,j_1})$ n'est pas possible car de nombreuses valeurs QS sont encore inconnues.
Toutefois, il est possible de calculer une partie de la réduction pour laquelle les valeurs QS sont connues: pour $\delta_1 < k < \delta_2$ avec $\delta_1 = j - d_0$ et $\delta_2 = i + d_0$. Cela signifie qu'en enlevant la partie de la réduction qui viole la relation de dépendance des données, il est possible de calculer en parallèle dans un pavé un ensemble de cases de la matrice. Dans ce pavé, la bonne localité mémoire permet de charger en mémoire partagée du GPU les données nécessaires au calcul. Chaque thread va ensuite accéder à cette mémoire partagée pour effectuer ses opérations. De cette manière, les besoins en bande passante sont réduits d'un facteur égal à la taille du pavé (16 dans notre implémentation GPU). Le kernel n'est plus limité par la bande passante et peut au contraire exploiter au maximum les capacités de calcul du GPU. Le reste de la réduction est calculé de manière traditionnelle diagonale après diagonale.

Ce schéma de pavage est aussi applicable au code CPU. La bonne localité mémoire est aussi bénéfique sur un processeur traditionnel: elle garantit un faible nombre de défauts de cache. De plus, le code pavé rend relativement aisé une vectorisation du code avec les instructions SSE. Une implémentation sur CPU avec les instructions SSE a été développée. De plus, un second niveau de pavage permet une réutilisation des chargements mémoire au niveau des registres SSE. Le code CPU peut également facilement tirer partie d'une parallélisation multi-coeurs avec openMP en parallélisant sur les différentes cases ou pavés d'une diagonale. Cela permet une vectorisation efficace du code CPU, et donc une comparaison honnête des performances entre CPU et GPU.

2.3 Résultats

Pour des grandes séquences, le code pavé sur GPU est jusqu'à 150x plus rapide que le code séquentiel UNAfold disponible sur internet. Le code pavé et vectorisé sur CPU est quant à lui jusqu'à 12x plus rapide que UNAfold. Cette accélération provient à la fois de la bonne localité fournie par le pavage et de la vectorisation SSE. Le code CPU vectorisé et parallélisé sur 8 coeurs de CPU est quant à lui "seulement" deux fois plus lent que le code GPU.
Des algorithmes proches du calcul de la structure secondaire optimale existent, en particulier le calcul de structures sous-optimales et la fonction de partition [84, 58]. Nous avons également appliqué notre approche à ces algorithmes et avons obtenus des accélérations similaires.
Notre implémentation GPU pavée va faire l'objet d'un chapitre dans l'ouvrage de NVIDIA "GPU computing gems":

Rizk, G., Lavenier, D., and Rajopadhye, S. (2010). GPU computing Gems, chapter GPU accelerated RNA folding algorithm. Addison-Wesley Professional.

2.4 Discussion

Dans ce chapitre, l'objectif était de déterminer s'il était possible d'avoir une implémentation efficace sur GPU du calcul de la structure secondaire de l'ARN. Nous avons développée une première parallélisation “naïve” fortement limitée par la bande passante mémoire de la carte graphique. Nous avons ensuite développé une nouvelle approche pavée de l'algorithme qui fait apparaître une bonne localité mémoire, permettant de passer d'un code limité par la bande passante à un code limité par la puissance brute de la carte. L'accélération obtenue va jusqu'à 150x par rapport au programme UNAfold. Le réponse est donc positive, il est possible de paralléliser efficacement un algorithme de programmation dynamique complexe de l'envergure de l'algorithme de calcul de structure secondaire. Toutefois, le facteur 150x est à prendre avec des pincettes, il ne provient pas uniquement de l'accélération GPU, mais aussi de notre nouvelle approche pavée. Nous avons noté que cette nouvelle approche permettait également une bonne accélération du programme CPU, fournissant ainsi une comparaison honnête des performances entre GPU et CPU. L'accélération GPU par rapport à notre code vectorisé sur CPU est de 2x sur un système octo-coeurs. Cela correspond aux accélérations observées dans [40] quand le code CPU est bien vectorisé. Cela peut paraître une faible accélération mais constitue en réalité déjà une optimisation très intéressante pour les pipelines de bio-informatique nécessitant d'exécuter ce programme de manière intensive.

Chapitre 3 : Alignement de séquences

L'alignement de séquences est un domaine majeur de la bio-informatique. Avec l'avènement des nouvelles générations de séquenceurs, le développement de nouveaux programmes d'alignements efficaces est plus que jamais une priorité. Dans ce chapitre l'objectif est de déterminer si une parallélisation GPU peut résoudre ce problème efficacement. Nous avons développé une stratégie qui élimine la plupart des alignements potentiels provenant de la phase d'indexation, avant qu'ils aient à être étendus par le coûteux algorithme de programmation dynamique de type Needleman-Wunsch. En revanche, cette approche rend difficile une implémentation GPU efficace. Il a donc été jugé préférable de développer et optimiser la version CPU du programme, qui a aboutit à la création d'un nouveau programme d'alignement, appelé GASSST.

3.1 Introduction

Les nouvelles technologies de séquençage (NGS) produisent de grandes quantités de données. Elles sont utilisées dans un large spectre d'applications, incluant le re-séquençage de génome, la détection de polymorphisme ou l'étude des éléments transcrits. Les données consistent en un grand nombre de petites séquences appelées *reads*. Un problème récurrent est de localiser ces reads dans un génome de référence, en autorisant quelques erreurs, mismatchs, insertions ou délétions. Pour cela, la stratégie *seed and extend* est la plus utilisée en pratique. L'idée est que des alignements significatifs vont inclure des régions avec une correspondance parfaite entre la séquence requête et la séquence

de référence. Par exemple, tout alignement de 50 paires de bases avec 3 erreurs contient au moins 12 bases identiques consécutives. En conséquent, dans la stratégie *seed and extend*, seules les séquences partageant des k-mers identiques avec la référence sont considérées comme alignements potentiels. La détection de k-mers communs est par exemple effectuée en utilisant un index localisant tous les k-mers sur la séquence de référence. Ensuite, tous les k-mers chevauchants, appelés *graines*, de cette séquence sont extraits. Pour chaque graine l'index fournit une liste de positions dans la référence, qui sont autant d'alignements potentiels, appelés CAL pour *candidate alignment location*. Chaque alignement potentiel doit ensuite être "étendu" pour savoir s'il peut générer un alignement suffisamment similaire. Pour autoriser les insertions/délétions, cette étape est en générale effectuée avec un algorithme de programmation dynamique de type Needleman-Wunsch. Cette méthode bénéficie d'une bonne précision et d'une bonne sensibilité, mais est aussi extrêmement lente à cause du nombre très élevé de coûteuses instances de programmation dynamique à résoudre.
Les solutions utilisées dans la littérature pour accélérer cette stratégie sont l'utilisation de graines espacées ou multiples. C'est par exemple le cas des programmes BFAST [30], SHRiMP [71] et MAQ [44]. Une autre solution développée dans SHRiMP est la vectorisation en SSE du code de programmation dynamique. La principale méthode concurrent à la stratégie *seed and extend* est celle qui utilise un indexation basée sur une transformée de Burrows-Wheeler (BWT) [9]. La méthode utilise une technique appelée *backward search* [22] qui permet d'utiliser l'index pour obtenir des résultats similaires à une recherche utilisant un arbre des suffixes. Cette méthode est utilisée par les programmes Bowtie [38], SOAPv2 [47], et BWA [42]. L'avantage de cette méthode est des temps d'exécution très courts et peu de besoins de mémoire vive, mais elle ne fonctionne bien en général que pour des alignements avec un faible taux d'erreurs.

3.2 Principe

Notre objectif est la création d'un nouveau programme d'alignement efficace, fonctionnant dans un grand nombre de situations : avec des reads courts ou longs, et des taux d'erreurs faibles ou élevés. Nous avons exploré deux pistes pour le développement de cet algorithme. Premièrement, une approche par *filtrage*, et deuxièmement, une parallélisation sur GPU.

L'approche par *filtrage* vient de l'observation que la plupart des alignements candidats produits par la stratégie *seed and extend* ne génèrent pas de bons alignements. Seul un très faible pourcentage, généralement inférieur à 1%, conduit à des alignements significatifs. De plus, il n'est probablement pas nécessaire d'effectuer le complexe calcul de Needleman-Wunsch pour se rendre compte qu'un alignement va être mauvais. L'idée du filtrage est alors de mettre au point des tests consistant en calculs simples et rapides qui permettent d'éliminer rapidement les alignements candidats avec un faible score de similarité. Les alignements candidats qui passent ces filtres sont ensuite recalculés avec Needleman-Wunsch pour obtenir l'alignement complet. Le filtre que nous avons développé est basé sur l'utilisation de tables de scores pré-calculés qui contiennent le score d'alignement de Needleman-Wunsch de toutes les paires de mots de taille l. En stockant chaque score dans un octet, la taille de la table est de 4^{2l} octets. En pratique, nous utilisons $l = 4$ ce qui conduit à une table de 64Ko. C'est une table suffisamment petite pour tenir dans les premiers niveaux de cache du processeur et peut donc être accédée très rapidement dans un algorithme. Nous avons développé un nouvel algorithme qui réutilise cette table un grand nombre de fois pour calculer une approximation de l'algorithme de Needleman-Wunsch. En choisissant un schéma de score simple, i.e. 1 pour une erreur, mismatch, insertion ou délétion et 0 sinon, notre algorithme calcule une borne inférieure du score obtenu par Needleman-Wunsch. C'est à dire que l'algorithme détermine une borne inférieure du nombre d'erreurs présentes dans l'alignement. Nos mesures montrent que cette borne est calculée 10 à 100 fois plus vite que le score exact de Needleman-Wunsch. Son utilisation constitue donc un filtre très efficace pour éliminer rapidement des alignements candidats. Le fait que l'algorithme calcule une borne inférieure permet d'éliminer sans se tromper des alignements candidats avec trop d'erreurs. Aucun bon alignement n'est éliminé par le filtre, la sensibilité n'est donc pas diminuée. En revanche, certains alignements avec trop d'erreurs peuvent passer le filtre. Ceux-ci sont éliminés dans la phase finale de l'algorithme ou le calcul exact de Needleman-Wunsch est effectué sur les alignements candidats qui passent le filtre.

La deuxième piste explorée est la parallélisation sur GPU. Il y a deux possibilités pour cette parallélisation : soit la parallélisation de l'approche filtrée, soit la parallélisation de l'algorithme sans filtres. L'algorithme sans filtres est facile à paralléliser sur GPU, mais nos mesures ont montré que l'approche sans filtres est 10 à 100 fois plus lente que l'approche filtrée. Sa parallélisation GPU est donc inutile. L'algorithme filtré est rapide sur CPU, mais est en revanche très difficile à paralléliser efficacement sur GPU: l'utilisation de filtres introduit des divergences dans le traitement des différents alignements candidats: pour certains le calcul s'arrête dès le premier filtre, pour d'autres le traitement passe tous les tests. Le programme ne convient donc plus à un parallélisme de données ou les différents unités de calcul doivent effectuer exactement la même tache, comme c'est le cas sur GPU. Il convient par contre tout de même à un parallélisme de tâches, pour une parallélisation sur multi-coeurs. Pour cette raison nous avons donc décidé d'abandonner la parallélisation GPU de cet algorithme. En revanche, la nouvelle approche filtrée développée a permis la création d'un nouveau programme d'alignement efficace, appelé GASSST, pour *Global Alignment Short Sequence Search Tool.*

3.3 Résultats

Les résultats du programme GASSST ont été comparés en particulier avec ceux des programmes BFAST et BWA qui sont actuellement les plus utilisés. Pour cela, des jeux de données simulés ont été utilisés. Des reads de différentes tailles et taux d'erreurs ont été généres à partir du génome humain. Pour chaque expérience, la sensibilité, la précision et le temps de calcul ont été mesurés. Des jeux de données réels ont aussi été utilisés, pour lesquels la sensibilité ne peut être qu'estimée. Ces évaluations ont montré que le programme GASSST est efficace dans beaucoup de situations différentes: pour des reads courts et longs, avec un taux d'erreurs autorisées dans les alignements faible ou élevé. GASSST est aussi rapide que BWA pour des reads courts avec un faible taux d'erreurs, et devient beaucoup plus sensible pour des taux d'erreurs plus élevés.

Le programme GASSST est disponible en licence CeCILL, et a été l'objet d'un article

dans la revue *bioinformatics* :
Rizk, G. and Lavenier, D. (2010). GASSST: global alignment short sequence search tool. Bioinformatics, 26(20), 2534–2540.

3.4 Discussion

Dans ce chapitre, l'objectif était de déterminer si les GPU pouvaient être utilisés pour créer des algorithmes rapides d'alignement de séquences. Nous avons développé une approche par filtrage qui a donné de très bons résultats mais qui n'a pas pu être parallélisée sur GPU. Le dilemme pour la parallélisation GPU est le suivant: soit paralléliser l'approche non filtrée, qui est régulière et donc efficacement parallélisée sur GPU, mais qui est à la base très lente; soit essayer de paralléliser l'approche filtrée, qui est très rapide mais aussi très difficile à paralléliser. Les deux options menant à une impasse pour la parallélisation GPU, il apparaît donc impossible d'utiliser les GPU pour paralléliser la stratégie *seed and extend* .

Chapitre 4 : Conclusion

4.1 Contributions

Nous avons contribué à deux problèmes différents: le calcul de structures secondaires d'ARN et l'alignement de séquences générées par les nouvelles technologies de séquençage. Pour le premier problème, nous avons développé un code GPU qui va jusqu'à 150x plus vite que le programme UNAfold. Cela a été rendu possible par l'utilisation d'une méthode de pavage qui fait apparaître un bonne localité mémoire. Nous avons aussi développé un programme CPU qui exploite le pavage et vectorisé en SSE, qui est jusqu'à 10 x plus rapide que UNAfold. Nos méthodes ont également été appliquées au calcul des structures secondaires sous-optimales et au calcul de la fonction de partition. La méthode de pavage que nous avons développée est un bon exemple de ce qu'il faut faire pour obtenir des bonnes performances sur GPU. Tandis que notre première implémentation était limitée par la bande passante et n'utilisait donc qu'une faible fraction des capacités de calcul du GPU, la version pavée n'est plus limitée par la bande passante et utilise donc au maximum les capacités de calcul du GPU. Des méthodes similaires pourraient être utiles à d'autres applications GPGPU. Pour le second problème, l'alignement de séquences, nous avons développé un nouvelle approche filtrée basée sur l'utilisation de tables pré-calculées, qui a conduit à la création du programme GASSST fonctionnant sur CPU.

4.2 Perspectives

Pour le repliement d'ARN, il serait possible d'explorer des variations sur la méthode de pavage, apportant éventuellement encore un gain pour le GPU. Il serait également intéressant d'essayer d'appliquer notre méthode à des algorithmes similaires, par exemple ceux qui calculent les structures secondaires avec pseudo-noeuds, ou les algorithmes calculant d'autres classes de structures sous-optimales. Pour l'alignement de séquences, il semble possible d'optimiser davantage le programme GASSST. Dans un premier temps, il serait possible d'essayer des méthodes d'indexation plus élaborées, telles que des graines espacées. Il serait aussi utile d'étudier l'impact de l'utilisation de schéma de scores plus complexes pour notre approche de filtrage. Bien qu'il semble difficile pour la méthode actuelle d'intégrer des scores plus complexes (avec un coût différent pour le début ou l'extension d'une délétion), des méthodes approchées pourraient être développées. Plus généralement, il reste probablement de nombreuses pistes à explorer pour accélérer l'alignement de séquences. La stratégie *seed and extend* bénéficie d'une bonne sensibilité, mais souffre aussi d'une certaine lourdeur: elle génère un très grand nombre d'alignements candidats, générant beaucoup de calculs inutiles. Pour obtenir de meilleures performances, il semble nécessaire d'élaborer une stratégie radicalement différente.

4.3 Discussion

La problématique de cette thèse est de déterminer si les algorithmes utilisés en bioinformatique peuvent ou non être efficacement implémentes sur GPU. Par l'étude de deux problèmes différents, il nous est possible d'avoir un début de réponse à cette question. Pour les algorithmes de calcul intensif du type de l'algorithme de repliement d'ARN, les GPU peuvent fournir une implémentation efficace, même s'il peut être difficile d'exposer une bonne localité mémoire, nécessaire pour obtenir des bonnes performances. En revanche, pour le domaine du traitement des séquences en général, la réponse est non. Plusieurs facteurs en sont la cause: premièrement ils nécessitent en général la manipulation de grandes quantités de données, qui peuvent être longs à transférer sur GPU. Deuxièmement, l'intensité arithmétique (le ratio entre le nombre d'opérations et la quantité de transactions mémoire) est en générale faible, ce qui conduit à une sous-utilisation de la puissance de calcul des GPUs. Troisièmement, des structures de données complexes sont souvent utilisées, qui requièrent des accès aléatoire à la mémoire, et donc induisent une faible bande passante sur GPU.
En ce qui concerne les performances, dans notre cas le code GPU est environ équivalent à 16 coeurs de CPU. C'est moins qu'espéré initialement, mais déjà un bon résultat si on compare le coût de fonctionnement d'un GPU et de 16 coeurs de CPU. Enfin, la question du temps de développement sur GPU est importante. Notre expérience a montré que les temps de développement sur GPU sont assez longs, du fait d'un grand nombre de paramètres à tester et optimiser. Bien que subjectif, car fortement dépendant de l'expérience de chacun, il nous semble plus simple de paralléliser pour des instructions vectorielles SSE que pour GPU. Comme les instructions vectorielles des CPU fournissent déjà un gain de performance significatif, il semble judicieux pour de nombreux algorithmes de commencer par une implémentation SSE que par un portage sur GPU.

Contents

Introduction

Context

Bioinformatics deals with techniques to analyze biological data. Common fields include DNA or protein sequence alignments, analysis and comparison of protein or RNA structures. For all these fields, the amount of data available has been growing exponentially in the past decades.

As regards DNA sequencing, progress has been very fast. The double helix structure of the DNA was discovered in 1953 by Watson and Crick. The first organism to be sequenced was the bacteriophage ϕ X 174 composed of 5386 bases in 1977. The first draft of the human genome, more than 3 billion base-pair long, was released in 2000 at a cost of several billion of dollars. With nowadays next generation sequencing technologies, several giga base-pairs can be sequenced in a single run for approximately 10K$.

As a consequence, database size has grown exponentially. More and more data need to be analyzed. Furthermore, because of the dramatically decreasing sequencing costs and comparatively much slower evolution of storage capacity, it has sometimes become paradoxically cheaper to re-sequence than to store the sequence for later reuse. This stresses the need to analyze huge quantities of data on the fly, and store only interesting results. Moreover, CPU performance is not keeping up with this evolution. As a result, program execution takes more and more time and biologists are facing increasingly serious storage issues.

Contrary to the past, we cannot just wait for the next generation of processors to speed up slow sequential programs. In the past, ever increasing processor frequencies provided easy to use new processing power. This trend has now reached a dead-end, for two main reasons: the *memory wall* and the *power wall*. The former is the increasing gap between memory and processor speed, requiring larger and larger cache memories. The latter is the exponentially increasing power consumption as frequency increases.

The only way left to get more processing power is through parallelism. The first solution is the use of multi-core CPUs and clusters of computers. While effective, this is also an expensive solution. Another solution is the use of high performance specialized hardware. For example FPGAs are able to provide very high performance, but still suffer from high development costs.

Yet another solution comes from Graphic Processing Units (GPU). A GPU is a specialized processor designed to accelerate the rendering of graphics on screen. It is a highly data-parallel problem: identical computations need to be executed on many vertices and pixels. Therefore, GPUs were designed as massively data-parallel architectures:

they are composed of many computational units able to apply the same computation efficiently on different input data. Moreover, from fixed basic graphic functionality, GPUs gradually included more and more programmability options to allow game programmers to generate more realistic effects. This programmability and flexibility came to a point where non-graphic computation became also possible, giving birth to General Purpose computing on Graphics Processing Units (GPGPU). NVIDIA introduced in November 2006 the CUDA programming language designed entirely to facilitate GPGPU on their GPUs.

Using GPUs for standard computation has many advantages. First, strong competition driven by the game industry lead to cheap GPUs, and ever-increasing processing power. Every standard computer can be fitted with a high-end GPU, dramatically increasing its total computational power at a low cost. Secondly, GPUs are naturally well suited for any massively data parallel problem, arising in many scientific research domains. This includes computational fluid dynamics, medical imaging, finance, signal processing, cryptanalysis. Bioinformatics also exhibits a lot of data parallelism: the analysis of large data sets often requires to apply the same computation on different data. A priori it therefore seems natural to use GPUs to speed up bioinformatics applications.

Objectives

The objective of this thesis is to estimate GPU capabilities in typical bioinformatics applications. Our study focuses on two different problems, the computation of RNA structures and sequence alignments. Both have already been studied for a long time by the scientific community, but still poses computational issues, because of the rapid growth of biological data.

The fact that a problem shows a high level of data parallelism is not enough to predict that GPUs will be effective in practice. In fact, GPU architecture has many specificities that restrict the scope of possible applications. Our objective is also to evaluate the impact of these GPU specificities.

The evaluation of GPU has to be conducted on several different aspects. First, an important point is the development time required for an efficient GPU implementation. For instance, GPUs are a priori easier to program than FPGAs, but more difficult than CPUs. This needs to be evaluated in practice. Secondly, the execution speed. For a fair evaluation, GPU contribution needs to be compared with optimized CPU code. In particular, comparison with the vector computation units of CPU has to be conducted. Nowadays, all CPUs contain vector units able to provide small scale data parallelism, but still rarely used in practice. Then, the cost. Both the device initial cost and running costs need to be considered. The latter is directly dependent on power consumption, hence often evaluated as a performance per watt ratio.

Finally, the objective is more than just a GPU evaluation. It is also to develop, if possible, general methods to transform efficiently a program for an efficient GPU implementation. Techniques developed for the GPU implementation of the two problems studied here might prove to be useful for other unrelated problems using similar algorithmic schemes.

Outline

Chapter 1 - This first chapter explains the operating principles of the GPU hardware. Comparing it with the traditional CPU architecture enables to shed light on the strong and the weak points of GPUs. In a first part, the basic characteristics—computational power, latency, bandwidth—are introduced and detailed. Then, GPU specificities are explained.

Chapter 2 - This chapter is an optimization case study of what can be done with various high performance hardware on a well known bioinformatic algorithm: the Smith-Waterman algorithm, which computes optimal sequence alignments. This review shows the advantages and drawbacks of different hardware and puts GPU performance in perspective. It also provides a state of the art of parallelization methods for dynamic programming algorithms.

Chapter 3 - In this chapter, we present our first contribution: the GPU implementation of the RNA folding computation. This computation is conducted with a dynamic programming (DP) algorithm. Although other DP algorithms have already been successfully implemented on GPU, the RNA folding computation poses significant challenges: complex data dependency scheme and memory access pattern. We introduce here a new tiled implementation which provides highly efficient GPU implementation. This tiled scheme is also applied to a CPU implementation, fully optimized with the use of vectorized instructions. Algorithms computing sub-optimal foldings and partition function were also studied and implemented on GPU and vectorized on CPU.

Chapter 4 - Our second contribution is developed in this chapter: the design of a new sequence alignment program. More precisely, the short read mapping problem, coming with the advent of next generation sequencing technologies, is targeted. After a state of the art of the short read mapping techniques, we introduce our attempt at GPU parallelization. Our main contribution is the development of a new algorithm filtering candidate alignment positions quickly, based on the precomputation of tiles of the dynamic programming matrix. This new algorithm proved to be in fact more effective on a sequential CPU program and lead to the creation of the GASSST software, standing for Global Alignment Short Sequence Search Tool.

Chapter 1

Architecture

Our objective is to accelerate bioinformatics programs through the use of high-performance commodity hardware: GPUs. The first step is to study the hardware in details, to know its weak and its strong points and to see how it competes with traditional CPU architectures.

First in section 1.1 the basic notions required to compare different architectures are detailed. We distinguish three main characteristics: raw power computation, memory bandwidth and latency. We explain how these characteristics impact the total execution time of a program and how they differ between GPU and CPU architectures.

Then in section 1.2 the CPU and GPU devices used in our work are presented. The different forms of parallelization provided by CPU are quickly explained, and GPU specificities are described. Finally GPU and CPU differences are discussed.

1.1 Common concepts

The total execution time of a program depends on many factors. In this section, we will address the hardware point of view. For a given program, which hardware characteristics determine the speed at which it will be executed ?

We distinguish three main characteristics: raw power computation, memory bandwidth and latency. These features are important because they give the basis for comparing different architectures. If the performance of each architecture are known on each of these characteristics, and also how the execution speed of a program is related to them, we can determine the theoretical speedup which can achieved when moving from one architecture to another.

1.1.1 Throughput

Throughput is the computational raw power of an architecture. It is the number of operations that can be executed within a given time. Architecture manufacturers usually give figures in operations per clock cycle. Sometimes *reciprocal throughput* is given, i.e. the average number of cycles it takes to perform an operation.

Some processors are designed to perform multiple operations on each clock cycle, greatly increasing their computational throughput. We distinguish three classes, possibly cumulative: scalar, vector and superscalar processors.

***Scalar* processors.** Instructions handle a single data item at a time, therefore the maximum number of operations per clock cycle is one.

***Vector* processors.** Each instruction operates simultaneously on many data items. The processor is therefore able to perform several operations at each clock cycle.

***Superscalar* processors.** Such processors can execute several different instructions on each clock cycle by dispatching instructions to several functional units contained in the processor. Again, the processor is able to perform several operations per clock, possibly of different kinds.

Current processors are usually both *superscalar* and *vector* processors, with several Arithmetic Logic Units (ALU), Floating Point Units (FPU), and vector units.

Multiplying operations per clock cycle by the clock rate returns operations per seconds, usually defined as Giga Floating point Operations per Seconds (GFLOPS) or GOPS for other operations.

Table 1.1 shows the throughput for different operations on several architectures. Figures are computed from the Nvidia and Intel optimization guides providing the throughput in operation per clock cycle [31, 63]. For the Xeon, we display the overall throughput when using the four cores of the CPU. Those are only theoretical maximum values and are sometimes in fact only obtained for very special cases.

For example the 38.4 GOPS of the Xeon for the 32-bit *add* (without using SSE) is obtained with the formula 4 cores $\times$ 3.2 Ghz $\times$ 3 execution units. The XeonX5482 is indeed able to dispatch an integer *add* to any of three execution units and therefore able to perform 3 integer *add* each cycle. This peak *add* throughput can be reached only if the processor finds 3 independent instructions in the code at each clock cycle, which is highly unlikely.

The GPU hardware possesses fixed function units for the computation of transcendental operations like *exp, log, cos, ...*, whereas those operations are evaluated through algebraic expressions in CPU. This explains the huge difference showed table 1.1 for *cos*: the Tesla C1060 is hundreds of times faster than the Xeon. Other than this special case, Tesla C1060 peak throughput is between 3 and 6 times that of the quad core Xeon using SSE for 32-bit operations. In practice the actual speedup will of course depends on how much of this peak throughput each architecture is able to harness.

Besides, computation throughput is not the only factor to consider, as we shall see next.

1.1.2 Bandwidth

Bandwidth is the amount of data transferred per unit of time, usually expressed in Gbits/sec (Gb/s) or GBytes/sec (GB/s).

Table 1.1. Instruction Throughput on GPU and CPU

Operation	Tesla C870	Tesla C1060	Xeon X5482	Xeon X5482, with SSE
32-bit integer add	166.4	312	38.4	102.4
32-bit floating point add	166.4	312	12.8	51.2
32-bit floating point divide	18 - 36	34 - 68	≈ 0.64	≈ 2.56
32-bit floating point cosinus	41.6	78	≈ 0.15	-
64-bit floating point add	-	39	12.8	25.6
peak 32-bit GFLOPS, best case	332.8	624	25.6	102 .4
peak 64-bit GFLOPS, best case	-	78	12.8	51.2

Instruction Throughput is given in giga operations per seconds. Figures are inferred from NVIDIA and Intel optimization guides [31, 63]. The best GFLOPS case is attained with multiply-add operations. The claimed GFLOPS peak is sometimes 933 for the the Tesla C1060 when mixing multiply-add and mul operations, but this does not appear to work in practice, even on simple kernels [personal tests].

Table 1.2. Maximum theoretical and measured bandwidth

Model	peak in GB/s	Measured	formula
GPU C870	76.8	60.5	384 bit bus $\times$ 1.6Ghz / 8
GPU C1060	102.4	81.6	512 bit bus $\times$ 1.6Ghz / 8
GPU GTX280	140.8	114	512 bit bus $\times$ 2.2Ghz / 8
CPU Xeon 5400	10.6	≈ 4.5	64 bit bus $\times$ 1.3Ghz FSB / 8
CPU Xeon 5500	32.0	-	64 bit bus $\times$ 1.3Ghz FSB $\times$ 3 channels / 8
PCI 2.0 16x	8	5.7	-
GPU C1060 shared memory[a]	41.6	-	32 bit $\times$ 0.65Ghz $\times$ 16 banks /8
Xeon 5500 L1 cache[b]	51.2	-	128 bit $\times$ 3.2Ghz /8

Bandwidth between memory and processor for several GPU and CPU models, and over the PCI link between GPU and CPU. Figures are given in GB/sec with GB=10^9 bytes. Second column is actual measured bandwidth with handwritten benchmarks. Last column shows the formula used to compute the peak bandwidth.
[a] Bandwidth per GPU multiprocessor, C1060 has 30 MP.
[b] L1 bandwidth per CPU core.

Schematically, data transfers happen to feed the processor with data stored in the memory, and to store back results. The actual situation is however quite complex, because of the complex memory hierarchy in a typical system. Bandwidth can refer to transfers between main memory and processor, between L1 or L2 cache and processor, or between CPU and GPU main memory over the PCI bus.

Table 1.2 shows the maximal theoretical bandwidth for the memory to processor transfer of various GPU and CPU. GPU bandwidth is approximately ten times greater that Xeon 5400 bandwidth. However, with the new *nehalem* architecture, CPU bandwidth has also recently seen a large increase.

Depending on the arithmetic intensity (the ratio of arithmetic operations over memory operations) the memory bandwidth or the computational throughput can be the bottleneck. For example, with the Tesla C1060, according to peak throughput and bandwidth, a program has to perform at least 24 floating point operations per memory load/store to reach the full GFLOPS peak. However, if the working set of an algorithm fits in the on-chip storage, data can be loaded once and reused several times. In this case, main memory bandwidth is less often a limitation.

Table 1.3. Latency table

	Latency, in clock cycles
GPU main memory	400-800
CPU main memory	≈ 100
Xeon 5482 L1 cache	3
Xeon 5482 L2 cache	14
Xeon 5482 floating point add	3
Xeon 5482 floating point divide	6-20
Xeon 5482 integer add	1

Indicative latency of different memories and some CPU instructions.

1.1.3 Latency

Latency is the time delay between issuing an operation and receiving its result. It is usually expressed in nano-sec or in clock cycles.

First, it concerns any processor instructions, and tells the time it takes to finish some computation. In a non-pipelined system, latency of the instructions equals the reciprocal throughput. On the contrary, in a pipelined architecture, the processor does not have to wait for the result of an operation before issuing the next instruction. In this case, *latency* and *throughput* are different and should not be confused.

Secondly, latency concerns memory transactions: the time delay between asking for some data and actually receiving it. In this case latency greatly depends on the memory physical location: off-chip memory latency is several orders of magnitude higher than on-chip memory latency.

Table 1.3 gives the indicative latency of several different memory operations and instructions.

1.1.4 Analysis

In a purely sequential system latency is a major issue. For example if an operand resides in main memory, the latency will be high and the processor will stall until it becomes available. Similarly, if some operand is the result of a previous computation, the processor will stall until this previous operation is complete.

The first solution to reduce memory-induced latency is the use of a hierarchical memory system: L1, L2 and sometimes L3 cache between the CPU registers and the main memory. Data used often will reside in small memory closer to the CPU and thus have lower access latency. Since this solution works well, CPU caches have seen their size increased a lot in the past few years.

A second solution to the latency problem is the *out-of-order* execution implemented in most CPUs. It consists in reordering differently instructions of a program: if some instruction is waiting for some operand, the processor will try to execute another upcoming instruction of the program. In that case, some latency is effectively hidden. This reordering depends on the amount of Instruction Level Parallelism (ILP) of the program—ILP measures how many instructions a program can perform simultaneously. In a series of dependent operations the processor will not be able to re-order instructions.

Yet another solution is to exploit task-parallelism. If a task is waiting for some data, the processor will execute instructions from another task—i.e. another thread—so that the processor is not stalled. This is implemented by Intel with their *hyper-threading* technology.

Likewise in the highly parallel GPU architecture, the hardware is often able to effectively hide most of the latency, as we shall see in next section detailing the GPU architecture.

1.2 Platforms

In this section the CPU and GPU architectures used for our experiments are presented.

1.2.1 CPU

Description

Experiments were mainly conducted on an 8-core system composed of two quad-core Xeon E5430. It was introduced by Intel in November 2007 and is based on a *core microarchitecture* using a 45nm fabrication process.

Each Xeon E5430 contains 4 CPU cores. There are 32KB of L1 data cache per core, and 6MB of L2 cache shared between two cores. The Xeon E5430 runs at 2.66Ghz, and some tests were also ran on the E5482 variant running at 3.2Ghz.

Task parallelism

Definition 1. *Task parallelism is an execution scheme where multiple processing elements are working on different independent tasks.*

This form of parallelization is handled by the different cores of the CPU. It is traditionally conducted with the POSIX threads (Pthreads) or OpenMP APIs.

Data parallelism

Definition 2. *Data parallelism is an execution scheme where multiple processing elements are executing the same task on different data.*

Current Intel processors exploit data parallelism with a set of instructions working with vectors, i.e. the same instruction is applied on each vector element. This is a Single Instruction Multiple Data (SIMD) system. The instruction set is called Streaming SIMD Extension (SSE). Xeon E5430 includes SSE up to version 4.1. Vectors are 128 bit wide, meaning that four 32-bit floating point operations can be handled simultaneously, or up to sixteen 8-bit integer operations.

The first way to implement SSE in a program is directly via the corresponding assembly code. It is obviously the most challenging approach, but potentially yielding the best results.
The second way is through the use of *intrinsics*. It is an API extension set built into the compiler giving access to the low-level SIMD instructions. It makes possible an easy implementation in a C/C++ program and, contrary to direct assembly code, relieves programmer from tedious register management.
The last way is via automatic vectorization handled by the compiler itself. This obviously does not work in all cases and is an ongoing field of research.

1.2.2 GPU

The GPU architecture we used is the Compute Unified Device Architecture (CUDA) developed by NVIDIA. It combines high performance with the ease of use of a C-like environment with the *C for CUDA*[1] programming language [63].

GPUs are massively parallel architectures providing cheap high performance computing, originally designed for image processing. Their first use for general computing were done through the use of graphic programming languages like openGL, which has the advantage of running on all graphic cards. However, this approach requires the painful work of mapping algorithms into graphic concepts and is not fully efficient. That is why GPU vendors have developed specific programming languages for general computation, CTM (Close To Metal) for AMD, C for CUDA for NVIDIA, and more recently OpenCL by the Khronos consortium which aims at providing one universal computation language for all graphic cards.

Shortly speaking, GPUs trade the high amount of transistors devoted to on-chip memory cache and flow-control in traditional CPUs for more processing units. Instead, they rely on multithreading to hide latency, which requires a high arithmetic intensity (many calculations per memory access) to work efficiently.

Description

Work was conducted with two generations of NVIDIA graphic cards, G80 and GT200. The latest Fermi architecture was not available at development time.

The GT200 is divided into 30 multiprocessors each being a SIMD unit of 8 32-bit scalar processors (SP), as shown figure 1.1. Each multiprocessor also contains one double-precision execution unit, two special function units (SFU) dedicated to transcendental functions, and a single instruction fetch unit.

A GPU procedure is a *kernel* called on a set of threads, divided in a *grid* of *blocks* each running on a single multiprocessor. Furthermore *blocks* are divided into warps of

[1]The “C for CUDA” programming language is often simply referred to as *CUDA*, although *CUDA* is originally just the name of the architecture.

32 threads executed in a SIMD-like fashion. Moreover, only threads within a block can be synchronized and share the fast on-chip shared memory.

One key difference with a traditional CPU implementation is that the programmer has to explicitly handle several memory spaces having different performance, size, scope and lifetime:

- Global memory: slow, large, read-write, not cached, per *grid*, application lifetime.
- Texture memory: slow, large, read only, cached with spatial locality, per *grid*, application lifetime.
- Constant memory: slow, small, read only, cached, per *grid*, application lifetime.
- Shared memory: fast, small, read-write, on-chip, per *block*, kernel lifetime.
- Registers: fast, small, read-write, on-chip, per thread, kernel lifetime.

In order to achieve good performance, the programmer has to isolate key data components and allocate them to different memory spaces according to their size and access pattern. This is a difficult task since it requires careful tuning and benchmarking of numerous different options.

Specificities

SIMT. NVIDIA emphasizes that strictly speaking, the hardware is not *SIMD* but *SIMT*, meaning Single-Instruction, Multiple Threads. Whereas a SIMD system applies one instruction to multiple data lanes, SIMT applies one instruction to multiple independent threads. All threads of a *warp* start together at the same program address, but each has its own instruction address and register states meaning that threads are free to branch and execute independently. At each cycle, the multiprocessor executes one common instruction of a warp at a time. Therefore, full efficiency is reached when all threads of a warp take the same code path.

The first difference appears in software: contrary to SIMD which exposes the vector width explicitly to the programmer, a SIMT program specifies the execution and branching behavior of a single thread. This new coding paradigm appears to be one of the major reason of the CUDA success in the community.

Secondly, it is believed but not clearly documented by NVIDIA that the SIMT hardware may be more efficient than SIMD when handling divergent code path, thanks to the fact that each thread has its own program counter.

Width. Although a multiprocessor is made of 8 SPs, warps are grouped into 32 threads. The instruction fetch unit works at half the clock rate of the SPs, and may dispatch instructions alternatively to the 8 SP or to the two SFUs who work independently, hence the 4x ratio. Branching (if-then-else control flow instructions) occurring across different warps does not affect performance.

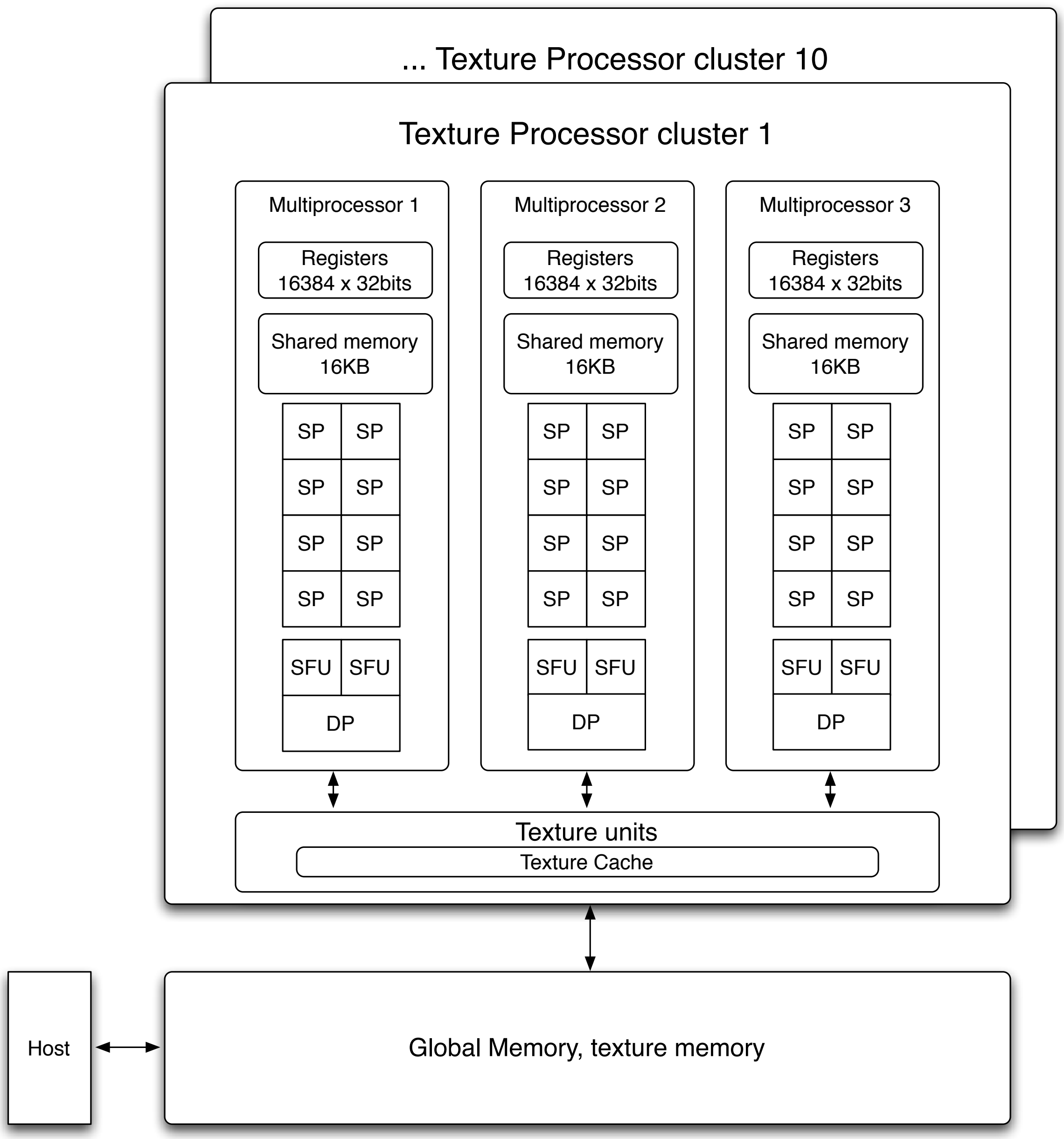


Figure 1.1. GT200 architecture. The GT200 has 30 multiprocessors each containing 8 stream processors, 2 special function units and one double precision unit. Multiprocessors are grouped by 3 in texture processor clusters with a texture unit. Each multiprocessor contains 16KB of on-chip shared memory and 16384 32-bit registers.

Coalesced reads. Maximum memory bandwidth is achieved only when threads of a warp coalesce their memory accesses. Detailed requirements for coalesced reads depend on device generation, but roughly are to have consecutive threads access consecutive memory locations at the same time.

Shared memory. It is a fundamental element responsible of the architecture efficiency. As stated in section 1.1.2, although already quite high the main memory bandwidth is generally not sufficient to feed with data the numerous execution units. The issue is addressed by storing data reused several times or by different threads in the small shared memory. Moreover, full performance is achieved only when simultaneous accesses hit the 16 different memory banks composing the shared memory. Avoiding bank conflicts is however generally a late optimization stage.

Occupancy. Each multiprocessor in the GT200 chip can handle 32 warps at a time, i.e. 1024 threads. The ratio between the actual amount of warps residing in a multiprocessor and the maximum is called *occupancy*. A high number of threads allows the hardware to hide most of the memory latency: while a warp waits for some data, another warp is scheduled for execution. This mechanism is the reason why the GPU architecture does not need large data cache, but it works best only if the occupancy is high enough.

Hardware multithreading. The execution context of each active warp in a multiprocessor is stored and maintained on-chip. This implies that switching from one warp to another is completely free. It also means that on-chip resources (registers, shared memory) are shared between all active warps. This implies that occupancy is limited by the registers and by the shared memory consumption of each warp.

1.2.3 CPU/GPU comparison

From a general point of view, CPU and GPU architectures might look similar: a GT200 chip is an array of multiprocessors each containing a SIMD unit, while the Xeon E5430 is an array of cores each containing scalar and SIMD units. However, taking a closer look reveals many key differences.

The CPU contains many components aimed at making a single instruction stream run fast: out of order execution, complex branch prediction, memory pre-fetch system, and large data cache. The GPU does not need such components, because massive multithreading provides an easy way to keep pipelines execution full.

GPU can therefore devote more space to computation units, which explains the difference in peak throughput observed. The drawback is however that full efficiency is only reached with a high enough number of threads. The GPU relies on multi-threading for its performance, therefore, contrary to the CPU which can run at full efficiency with one thread per core, the GPU needs thousands of threads to make full use of its 240 “cores”.

Despite these differences, GPU and CPU appear to follow a converging path. GPU are evolving toward easier programmability while CPU are evolving toward more and more parallelism: latest GPU Fermi architecture includes an automatic cache, CPUs have more and more cores and will have wider SIMD units in the future. Therefore, GPUs are merely anticipating what will need to be done for tomorrow's CPUs.

Chapter 2

Optimization case study: Smith Waterman algorithm

The Smith-Waterman (SW) algorithm is a well-known bioinformatic algorithm performing local alignment [74, 25]. It is the most sensitive algorithm, but also computationally demanding with its quadratic complexity. Therefore heuristic approaches are often preferred, such as BLAST [4, 5] although it may miss alignments.

To achieve both high sensitivity and reasonable execution times, the solution is to make SW algorithm run faster. That is why SW has been thoroughly studied and optimized in the past decades. It has been implemented on many high-performance architectures: VLSI, FPGA, GPU, Cell, SIMD CPU.

It is therefore a good example to have an overview of the different high performance architectures and to see how a GPU implementation competes with them.

2.1 Smith Waterman algorithm

SW is a dynamic programming (DP) algorithm: the global problem is recursively computed from smaller subproblems. The similarity score $H(i,j)$ of two sequences S_1 and S_2 ending at position i and j is computed recursively with the equations:

$$\begin{aligned}
E(i,j) &= \max(E(i,j-1) - G_{ext}, H(i,j-1) - G_{init}) && (2.1)\\
F(i,j) &= \max(F(i-1,j) - G_{ext}, H(i-1,j) - G_{init}) && (2.2)\\
H(i,j) &= \max(0, E(i,j), F(i,j), H(i-1,j-1-W(S_1(i),S_2(j)))) && (2.3)
\end{aligned}$$

G_{init} and G_{ext} are the penalties for starting or extending a gap, and $W(S_1(i),S_2(j))$ the substitution scoring matrix for all nucleotide pairs. The computation of each cell H depends on its left, upper and upper-left neighbors meaning cells along a minor diagonal can be computed in parallel.

Performance of the SW algorithm is usually given in Cell Updates Per Second (CUPS), calculated as $|S_1| \times |S_2| /$ execution time.

2.2 VLSI accelerators

An application specific integrated circuit (ASIC) is a chip customized for a particular use. It provides highest performance but also highest development costs.

High performance is achieved with parallelization of the $\mathcal{O}(n^2)$ DP matrix. A common scheme is to parallelize work across the minor diagonal. For two sequences S_1 and S_2, a linear array of $|S_1|$ processing elements (PE) can finish computation in $|S_1|+|S_2|-1$ steps. Each PE is assigned a letter of S_1 (i.e. a line in the DP matrix), then at each step a database sequence is shifted through the array. Each PE computes one cell of the matrix at each step. Since the number of PEs available is usually smaller than $|S_1|$, some partitioning strategy is needed.

In 1987, Lopresti [51] presented the first VLSI implementation. He designed a chip with 30 PEs with fixed substitution score matrix and linear gap score, limited to global alignments. They reported a 125x speedup compared to a DEC VAX 11/785. They noted that their design was severely limited by I/O bandwith, and that PEs were actually running at a tenth of their full capacity.

Chow *et al.* [12] designed in 1991 a chip with 16 PEs. Their chip permits local and global alignment and parameterizable insertions/deletions and score penalties. They describe a system with 16 connected chips controlled by a Motorola 68020 microprocessor. Their system can be linearly extended with many chips to allow alignment of large sequences without the help of external software.

In 1996, Guerdoux-Jamet and Lavenier [26] presented the SAMBA accelerator, a systolic array of 128 VLSI processors designed to compute cells of the DP matrix. Key aspects are: (i) the algorithm is parameterized so that computation of ungapped, gapped local or global alignment are all possible, (ii) an FPGA interface handles the partitioning of computation required when input sequences are longer than 128. SAMBA yielded a 96x speedup compared to a 200 MHz Pentium Pro .

VLSI accelerators suffer from high development and production costs. This is the reason why they have gradually been replaced by FPGAs. FPGAs allows programmers to implement architectures similar than with VLSI, with the added benefit that a design can be regularly upgraded to use state-of-the art technology.

2.3 FPGA accelerators

FPGAs are high density arrays of logic components that programmers can configure to approximate specialized hardware accelerators. They are a good alternative to special-purpose hardware since they combine high performance with reconfigurability. Moreover the SW algorithm only requires simple integer arithmetic for which FPGAs excel. They usually implement the SW algorithm with the same architecture as VLSI accelerators, i.e. with a linear array of processing elements.

Oliver *et al.* [64] implemented the SW algorithm on FPGA by parallelizing work across the minor diagonal. They designed an array of 252 PEs implementing SW with linear gap penalties, and composed of 168 PEs in the case of affine gap penalties. They described a partitioning strategy to implement database scan with query sequences of any length. They reported a 125x speedup over contemporary P4 1.6 GHz processor.

More recently Allred *et al.* [2], Zhang *et al.* [83] followed the same parallelization scheme and obtained 9 to 25.6 GCUPS.

Li *et al.* [45] proposed another FPGA implementation scheme: they still have one PE compute a cell, but create a square 8x8 array of them. Within this tile the cell scores are calculated through unclocked signal propagation. The whole matrix is subdivided in as many 8x8 tiles as necessary. They reported a peak 24.5 MCUPS.

Cornu *et al.* [14] described how to use the ImpulseC C-to-hardware compiler tool to quickly implement a traditional linear systolic array architecture computing the SW algorithm. ImpulseC is a high level synthesis tool based on ANSI C aimed at facilitating the design process. They reported that ImpulseC indeed significantly accelerated design process and allowed to try several different design options much faster than with traditional methods. They obtained a peak 38.4 GCUPS.

2.4 CPU vectorization

Wozniak [81] proposed a vectorization scheme with vectors parallel to the minor diagonal using SIMD-like video oriented instructions on an ULTRA SPARC processor. He reported a twofold speedup over a traditional implementation.

Rognes and Seeberg [70] proposed another vectorization scheme with vectors parallel to the query sequence using the Intel MMX SIMD instructions. This new vector orientation greatly facilitates loading of substitution scores: a query profile—a kind of query specific scoring matrix—is computed once for an entire database search and permits to load all substitution scores of a vector of cells in one operation. However, in this scheme each element of a vector is dependent on the element above it. They however noted that the vertical contribution of F does not occur very often and can therefore be evaluated in a lazy fashion afterwards. They reported a six-fold speedup over sequential code.

Farrar [19] then also used this scheme with vectors parallel to the query but in a *striped* manner: a vector contains non consecutive values. This key improvement made it possible to move conditional branches of the lazy F evaluation outside of the inner critical loop. Sixteen 8-bits operations are conducted per SIMD vector, which limits scores to 255 maximum but also doubles computational throughput. Higher 16-bit precision computation is conducted when necessary. Farrar reported up to six-fold speedup over previous SIMD implementations.

Szalkowski *et al.* [76] slightly improved Farrar's implementation with optimizations when loading the query profile and in the lazy F loop evaluation. They also provided multi-core support. They reached a maximum of 15.7 GCUPS on a Quad core Q6600 2.4 GHz.

2.5 Cell

The Cell broadband engine is a processor developed by IBM, present in particular in the Playstation 3 [36]. It is a heterogenous multi-core system: it combines a Power Processor Element (PPE) with 8 Synergistic Processor Elements[1] (SPE). Each SPE has a 128-bit SIMD unit and 256 Ko of local storage. It therefore combines task parallelism and data parallelism.

The difficulty especially lies in the fact that SPEs cannot directly access main memory, instead software controlled DMA transfers are used to copy data to the local storage. Moreover, it does not natively support saturated arithmetics, and 16-bit half word is the smallest supported element, which reduces throughput compared to SSE implementation on CPU.

Wirawan *et al.* [80] implemented SW on a Playstation 3 with SIMD vectors parallel to the query sequence, similar to the Rognes and Seeberg [70] scheme. The reference database is split among the different SPEs. They reported a speedup of 1.64 over the Farrar's *striped* vectorization running on a dual core Core2 2.4 GHz, and a peak 3.6 GCUPS.

Szalkowski *et al.* [76] then presented an improved Cell implementation using the Farrar's *striped* scheme. They obtained a peak 8 GCUPS on a Playstation 3.

Farrar also presented in an unpublished article a Cell implementation with his striped scheme [20], where he claimed a 12 GCUPS on a Playstation 3. He also reported that a Cell with 8 SPE(in an IBM QS20 blade) running at 3.2 GHz is 20% slower than an 8-core Xeon 5130 running at 1.6 GHz, in particular because of the lack of saturated arithmetic support.

2.6 GPU

The first GPU implementation was conducted in OpenGL before CUDA was available by Liu *et al.* [48]. General purpose computation on GPU through OpenGL requires to map the algorithm to the graphic pipeline: ask the GPU to render a quad, which will be decomposed into pixels by the GPU rasterizer, and provide a fragment shader program that will be applied to every pixel in parallel. In their implementation, a fragment program computes one cell of the matrix and the quad covers a minor diagonal (skewed to a column). They reported up to 650 MCUPS on a 7800GTX.

CUDA then made it significantly easier and more efficient to implement SW on GPUs. Manavski and Valle [55] provided the first CUDA implementation. Their implementation is *inter-task*: each GPU thread computes the whole alignment between a query sequence and a database sequence. The database is sorted according to sequence lengths so that different threads stop computing approximately at the same time. Their implementation performance is 1.8 GCUPS on a 8800 GTX and about 3.0 GCUPS on a GTX280.

Liu *et al.* [49] then provided a significantly more efficient CUDA implementation.

[1]In the Playstation 3, only 6 SPEs are available.

They still used the *inter-task* parallelization, with several optimizations: (i) database sequences and DP matrices are stored in a special way so that memory accesses are coalesced across threads, (ii) in a 8x8 tile of the matrix, intermediate results are stored in registers in order to decrease the number of global memory accesses. They obtained 9 GCUPS on a GTX 280.

Liu *et al.* [50] further improved their CUDA implementation by packing query profile and sequence data in char4 vectors, fetched in a single texture access. These optimizations yielded 17 GCUPS on a GTX 280.

2.7 Four-Russians speedup

Contrary to previous sections detailing hardware accelerations of the algorithm, we now present an algorithmic optimization of the SW algorithm, known as the *Four-Russians speedup.* Although not directly related to our study focused on hardware acceleration, we deem it necessary to also describe this approach.

It was introduced in 1970 by Arlazarov *et al.* [6] and initially developed for boolean matrix multiplication. The principle is to partition the whole DP matrix into blocks of t x t cells. In each block, the outputs are the last row and last column. They can be computed from the block inputs: the first row and column, and substrings of the two sequences spanning the block interval.

The idea is to precompute the block output results for every possible input and store them in a lookup table. Then, partition the whole DP matrix in blocks having one overlapping line and column, so that a block output becomes a neighbor's block input. Results of each block are obtained in constant time through the lookup table, hence the overall computation is conducted in $\mathcal{O}(n^2/t^2)$ steps. When choosing $t = log(n)$, complexity becomes $\mathcal{O}(n^2/(log(n))^2)$.

One issue is the time taken for pre-computation. One key observation is that when using a simple scoring scheme such as -1 for mismatch and gap open/extension, a block input row and column can be coded as an offset vectors of $-1, 0, +1$ values. The number of different possible inputs for a block is then $3^{2t} \times 4^t$ on a 4-letter alphabet, each taking t^2 steps to compute. With $t = log(n)$, pre-computation is conducted in $\mathcal{O}(n(log(n))^2)$ steps.

In practice the method would be used with t fixed resulting in $\mathcal{O}(n^2/t^2)$ complexity. It is however not commonly used, in particular because of the restriction it requires on the scoring scheme. The general idea is nevertheless very interesting and we believe related methods could prove to be very useful in some cases.

2.8 Discussion

Table 2.1 shows the peak performance of the various architectures presented. The first thing that should be noted is that for all architectures, finding the optimal implementation is not obvious and was not obtained at first time. It is the most striking with GPUs, where implementation from Manavski and Valle on a GTX280 first yielded 3

Table 2.1. Smith Waterman performance on various architectures

Author	Architecture	Device model	peak GCUPS
Rognes and Seeberg	SIMD CPU	Xeon Core2 2.0 Ghz	0.865
Farrar	SIMD CPU	Xeon Core2 2.0 Ghz	3.0
Farrar, Szalkowski *et al.*	SIMD CPU	quad core Q6600 2.4 Ghz	15.7
Liu *et al.*	GPU	GTX 7800	0.65
Manavski and Valle	GPU	GTX 8800	1.8
Manavski and Valle	GPU	GTX 280	3.0
Liu *et al.*, first version	GPU	GTX 280	9.0
Liu *et al.*, second version	GPU	GTX 280	17.0
Liu *et al.*, second version	GPU	GTX 295	28.0
Wirawan *et al.*	Cell BE	Playstation 3	3.6
Szalkowski *et al.*	Cell BE	Playstation 3	8.0
Oliver *et al.*	FPGA	Virtex II XC2V6000	5.8
Allred *et al.*	FPGA	XD2000i	9
Zhang *et al.*	FPGA	XD1000/ Altera StratixII	25.6
Cornu *et al.*	FPGA	XD2000i	38.4

Peak GCUPS performance on CPU, GPU and FPGA. Figures for CPU, GPU and FPGA are, respectively, taken from [19, 76], [48, 55, 50] and [64, 83, 2, 14].

GCUPS, then Liu *et al.* obtained 9 GCUPS and finally 17 GCUPS, realizing a 5.6x speedup on the same GPU. Similarly with SSE on CPU, there is a 3.5x performance difference between the Rognes and Seeberg, and Farrar implementations running on the same CPU.

This shows that although the SW algorithm is apparently simple, with each cell being computed from only three neighbors with simple arithmetic, getting the most of an architecture is a difficult task. Concerning GPUs, it seems that memory bandwidth is the most problematic point: improvements from Liu *et al.* mainly concern decreasing the number of global memory accesses when possible and developing a data layout allowing threads to access memory in a coalesced way.

GPU demonstrated its ability to provide cheap performance compared to other architectures: a GTX 295 card, costing around 450 $ contains 2 GT200 GPU chips and reaches a high 28.0 GCUPS. However, vectorized code running on CPU is also performing remarkably well. The ability to perform operations on 8-bit elements is probably largely responsible for this good result. There is not a large gap between vectorized CPU code and the best GPU or FPGA implementations: the GTX 295 is less than 2 times faster than the quad core CPU. When considering hardware costs, the benefit of high performance hardware is therefore not obvious.

What can be learned from this case study is first that GPUs are able to compete with other high-performance hardwares. Secondly, getting high performance is a difficult task even on a simple dynamic programming algorithm, it is therefore a real challenge to obtain a good speedup on a significantly more complex algorithm such as the RNA folding problem, presented in the following chapter. Finally, CPU is in fact a serious competitor.

Chapter 3

RNA folding

The RNA secondary structure prediction problem has been studied for a very long time and is routinely used in many bioinformatic pipelines. It is a computationally intensive algorithm whose execution time is often a problem when dealing with large data sets of sequences, therefore making it worthwhile to work on its optimization. However a GPU parallelization was never attempted before.

In this chapter, we investigate whether GPU architecture is able to provide a good speedup for this problem. After an introduction of the principles of RNA folding in section 3.1 and a state-of-the-art in 3.2, a profiling of the standard sequential algorithm is conducted in 3.3. We then detail our GPU implementation in sections 3.4 to 3.5, and the CPU implementation in 3.6. Results are presented in section 3.7, then variation of the algorithms are studied in 3.8.

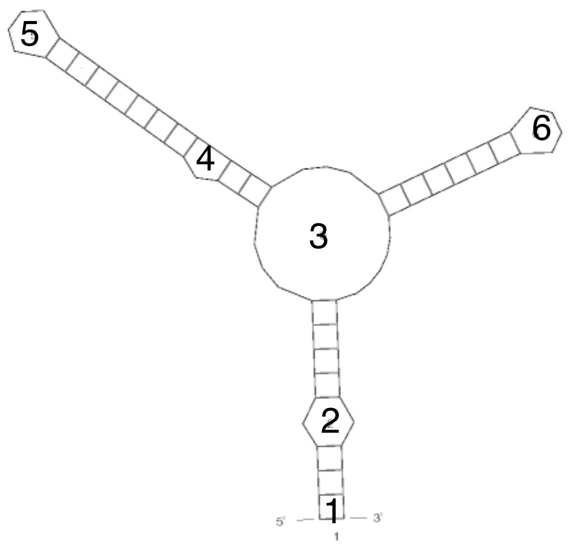
3.1 Main algorithm

3.1.1 Problem Statement

RNA or Ribonucleic acid is a single-stranded chain of nucleotide units. There are four different nucleotides, also called *bases*: adenine (A), cytosine (C), guanine (G) and uracil (U). Two nucleotides can form a bond thus forming a *base pair*, according to the Watson-Crick complementarity: A with U, G with C; but also the less stable combination G with U, called wobble base-pair. Because RNA is single-stranded, it does not have the double-helix structure of DNA. Rather, all the base pairs of a sequence force the nucleotide chain to fold in "on itself" into a system of different recognizable domains like hairpin loops, bulges, interior loops or stacked regions.

This 2-D space conformation of RNA sequences is called the secondary structure, and many bioinformatic studies require its knowledge. Figures 3.1 and 3.2 show different representations of a secondary structure along with the different kinds of loops. Algorithms computing this 2-D folding runs in $\mathcal{O}(n^3)$ complexity, which means computation time quickly becomes prohibitive when dealing with large data sets of long sequences.

The first algorithm was introduced in 1978 by Nussinov, which finds the structure with the largest number of base pairs [62]. In 1981 Zuker and Stiegler proposed an



```
AAAAAAGGGAAAAGAACAAAGGAGACUCUUCUCCUUUUUCAAAGGAAGAGGAGACUCUUUCAAAAAUCCCUCUUUU
((((.(((((...(((.((((((((....)))))))))))...(((((((....))))))).....))))).)))) (-24.5)
```

Figure 3.1. Secondary structure of an RNA sequence. An RNA sequence is a chain of four different nucleotide units A,C,G,U. In the bracket representation, a matching pair of parenthesis represents a base pair, and a dot an unpaired nucleotide. The figure shows the different kinds of loops: (1) a stacked base pair, (2) an interior loop, (3) a multi-loop, (4) a bulge loop and (5,6) hairpin loops. The score -24.5 represents the stability of the structure: the lower it is, the more stable the folding is.

algorithm with a more realistic energy model than simply the count of the number of pairs [85]. It is still widely used today and is available in two widely used packages, ViennaRNA [29] and Unafold [56]. Our goal was to write a GPU efficient algorithm with same usage and results as the one in the Unafold implementation.

RNA folding algorithms compute the most stable structure of an RNA sequence, i.e., the structure maximizing a given scoring system, or rather, minimizing the total free energy of the RNA molecule.

Although this may seem a daunting task since there is an exponential number of different possible structures, a key observation makes it computationally feasible in reasonable time: the optimal score of the whole sequence can be defined in terms of optimal scores of smaller subsequences, leading to a simple dynamic programming formulation.

In the following, we note $i \dots j$ the subsequence starting at position i and ending at position j. Let us look at the optimal score E of the whole sequence. As shown in recursion diagram 3.3.A, there are two possible ways it can be constructed: the last base n is either paired or unpaired. If it is unpaired, the scoring system implies that E can be directly derived from the score of sequence $1 \dots n-1$. In the second case, it is paired with some base k, thus breaking the sequence in two parts: subsequences $1 \dots k-1$ and $k \dots n$, with k, n forming a base pair. The scoring system implies that E is the sum of the score of these two parts. Now, if determining optimal scores of both parts are self-contained subproblems, meaning they are independent of the rest of the sequence, then they need to be computed only once, stored and reused wherever the subsequence

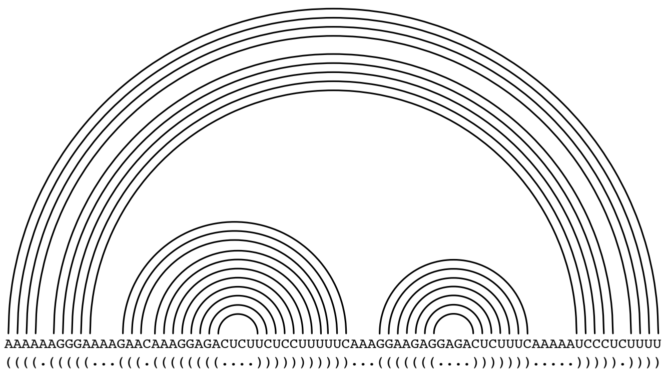

AAAAAAGGGAAAAGAACAAAGGAGACUCUUCUCCUUUUUCAAAGGAAGAGGAGACUCUUUCAAAAAUCCCUCUUUU
((((.(((((...(((.((((((((....)))))))))))...(((((((....))))))).....))))).))))

Figure 3.2. Secondary structure of an RNA sequence, arc representation. In this representation the polymer backbone is a straight line and base pairs are connected by arcs. In the absence of pseudoknots all loop structures are perfectly nested. This means that the structure of a subsequence $i \dots j$ in the case where i, j form a base pair does not depend on exterior element of $i \dots j$, i.e the subproblem is self-contained. This opens the path to dynamic programming.

occurs again. This opens the path to dynamic programming, E is computed as the optimal way to construct it recursively with the equation:

$$E_j \quad = \quad \min\left\{E_{j-1}, \min_{1<k<j}(E_{k-1} + QP_{k,j})\right\} \tag{3.1}$$

With E_j the optimal score of subsequence $1 \dots j$ and $QP_{k,j}$ the optimal score of subsequence $k \dots j$ when k, j form a base pair. The assumption that sub-problems are self-contained implies that there cannot exist a base pair d, e with $d < k$ and $k < e < j$, i.e., something that would make their scores dependent on exterior elements. Such forbidden base pairs are called pseudoknots.

Quantity $QP_{i,j}$ is also broken down in smaller sub-problems as shown in recursion diagram 3.3.B. Either subsequence $i \dots j$ contains no other base pairs—i, j is then called a *hairpin loop*—or there exists one internal base pair k, l forming the *interior loop* i, j, k, l, or there are at least two interior base pairs, forming a *multiloop.* The recursion can then be conceptually written in the simplified form:

$$QP_{i,j} \quad = \quad \begin{cases} \min\begin{cases} \text{Hairpin loop} \\ \text{Interior loops} \\ \text{Multi loops} \end{cases} & \textit{if pair } i \cdot j \textit{ is allowed} \\ \infty & \textit{if pair } i \cdot j \textit{ is not allowed} \end{cases} \tag{3.2}$$

3.1.2 Recurrence equations

We first describe with more details the principles of the folding algorithm as implemented in the Unafold package in the function *hybrid-ss-min* [56]. Recurrence equations of the

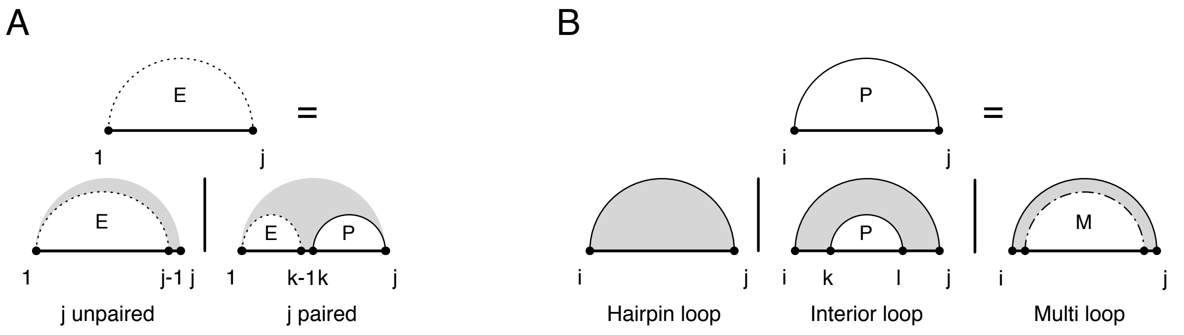


Figure 3.3. Recursion diagrams [66] represent the recursive equations of the dynamic programming algorithm. The flat line represents the backbone of an RNA sequence. A base pair is indicated by a solid curved line, and a dashed curved line represents a subsequence with terminal bases that may be paired or unpaired. Shaded regions are parts of the structure fixed by the recursion, and white regions denote sub-structures whose scores have already been computed and stored in previous steps of the recursion. **A.** Recursion of subsequence $1 \dots j$ assuming no exterior dependence. Last base j is either paired or unpaired. **B.** Recursion for subsequence $i \dots j$ assuming i, j are paired. The subsequence either contains none, one or several internal base pairs. This corresponds, respectively, to hairpin loops, interior loops and multi loops. In the case of multi loops another quantity QM is introduced and computed through another recursion.

RNA folding algorithm are well known in the bioinformatic community [85, 29]. Recurrence equation 3.1 shows that finding the most stable structure of an RNA sequence involves computing table $QP_{i,j}$, itself constructed in three possible ways as shown in equation 3.2. The first possibility is a hairpin loop, whose score is given by a function Eh mainly depending on loop length. The second possibility is interior loops, containing the special case where $k = i + 1$ and $l = j - 1$ corresponding to a stacked base pair. Functions Es and Ei give, respectively, the scores of stacked pairs and interior loops. The last possibility is when subsequence $i \dots j$ contains at least two internal base pairs, forming a multi loop. Another quantity QM is introduced, representing the score of a subsequence assuming it is forming a multi-loop. Recursive equation 3.2 can then be detailed as:

$$QP_{i,j} = \begin{cases} \min \left\{ \begin{array}{l} Eh(i,j) \\ Es(i,j) + QP_{i+1,j-1} \\ \min\limits_{\{k,l|i<k,l<j\}} Ei(i,j,k,l) + QP_{k,l} \\ QM_{i+1,j-1} \end{array} \right\} & \textit{if pair } i \cdot j \textit{ is allowed} \\ \infty & \textit{if pair } i \cdot j \textit{ is not allowed} \end{cases} \tag{3.3}$$

The value QM represents the score of a subsequence with at least two internal base pairs. It is simply broken down as two subsequences $i \dots k$ and $k+1 \dots j$ containing both at least one internal base pair. A third value QS is introduced to represent the score of such subsequences. $QS_{i,j}$ decomposition is then straightforward: either a pair is on i, j giving $QP_{i,j}$, or there are several pairs giving $QM_{i,j}$, or i or j are unpaired, giving $QS_{i+,j}$ or $QS_{i,j-1}$.

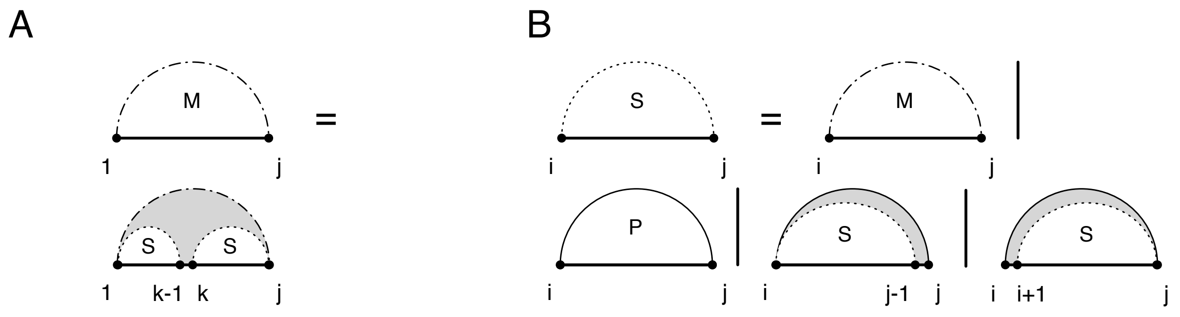


Figure 3.4. **A.** Recursion of subsequence $i \ldots j$ containing at least two internal base pairs. Both subsequences $i \ldots k-1$ and $k \ldots j$ each contains at least one base pair. **B.** Recursion for subsequence $i \ldots j$ containing at least one internal base pair. It contains either several or one base pair on i, j, or on $i, j-1$ or $i-1, j$ if i or j are unpaired.

Recursion diagrams 3.4 shows their decomposition patterns. Their recurrence equations are written as:

$$QM_{i,j} = \min_{i<k<j} (QS_{i,k} + QS_{k+1,j}) \tag{3.4}$$

$$QS_{i,j} = \min \{QM_{i,j}, \min(QS_{i+1,j}, QS_{i,j-1}), QP_{i,j}\} \tag{3.5}$$

The third term of equation 3.3 is computed over the four variables i, j, k, l, hence implying a complexity of $\mathcal{O}(n^4)$. However, a common approximation is to limit internal loops sizes to $K = 30$, thus reducing complexity to $\mathcal{O}(n^2 \cdot K^2)$. Equation 3.4 is in $\mathcal{O}(n^3)$, leading to an overall $\mathcal{O}(n^2 \cdot K^2 + n^3)$ complexity. The corresponding secondary structure is then obtained by a trace-back procedure.

3.2 Related work

3.2.1 Alternative approaches

The most used algorithm for RNA secondary structure prediction is the Minimum Free Energy (MFE) method, implemented in the two packages Unafold and ViennaRNA [56, 29]. This method relies on thousands of experimentally measured thermodynamic parameters giving the energy of the many different kinds of possible loops. On the contrary, the Stochastic Context Free Grammar (SCFG) approach does not rely on such thermodynamic parameters, it instead uses automated statistical learning algorithms to derive model parameters. It is obviously a major advantage, since experimentally measuring parameters is both expensive and not necessarily accurate.

However, first attempts at this method did not yield very accurate results [37, 18]. Then Do *et al.* [16] significantly improved the method and obtained even better accuracy than the MFE method. The SCFG approach, while promising, is still not often used. We therefore preferred to focus our work on the widely accepted and used MFE method.

The last approach is the comparative analysis method, which predicts RNA secondary structures from multiple sequence alignments. Extensive work has also been conducted in this field, see in particular the review of [24].

One particularly noteworthy approach was conducted by Frid and Gusfield [23]. They used the Four-Russians technique presented in section 2.7 to achieve a worst-case time bound of $\mathcal{O}(n^3/\log(n))$. They however used the simple Nussinov scoring scheme (the count of base pairs) and acknowledged that they did not provide yet a finished competitive program. Their work nonetheless illustrates that it is possible to obtain in practice a speedup with the Four-Russians approach even on a complex dynamic programming algorithm. They also claim that their method could be extended to more complex scoring schemes and energy functions.

There is a huge number of programs computing the secondary structures [1], detailing the different approaches is out of scope of this thesis. We instead focus now on the previous attempts at parallelizing the MFE method.

3.2.2 Parallelization

Parallelization on cluster, i.e. distributed memory

First parallelization efforts were aimed toward parallelization of the computation of a single sequence over multiple computers via MPI [28, 21]. A diagonal is distributed across the different nodes of the cluster. The problematic point is the amount of data transfer needed which greatly reduces overall speedup. Subsequent work aimed at developing a tiling scheme minimizing those costly transfers [3].

Those pioneer parallelization work are however not as useful today, since the computation of even a very large sequence can be conducted on a single computer. Clusters potential is now generally trivially exploited by distributing many different sequences across a cluster nodes.

Parallelization on multi core CPU, i.e. shared memory

Mathuriya *et al.* [57] implemented the MFE folding algorithm for multi cores using OpenMP in their GTfold program. Their implementation splits the elements of a diagonal across the available CPU cores. They use for the last diagonals the finer grain parallelization level consisting in splitting the minimization loop of the different kinds of structure loops. They also implemented the fast $\mathcal{O}(n^3)$ algorithm for the computation of internal loops introduced by Lyngsø *et al.* [53].

They ran their program on a 32 core P5-570 cluster and reported a 19x speedup compared to the sequential version. They also reported a 84s running time on a 9781 nucleotide-long HIV virus sequence.

Parallelization on FPGA

Jacob *et al.* [32] first parallelized both the Nussinov and Zuker algorithms [62] on FPGA.

Since the Nussinov algorithm does not use energy tables but only counts the number of base pairs, it is significantly simpler than the Zuker algorithm and as such a good first step to evaluate the FPGA approach. For the Zuker algorithm, they reported a 103x speedup with their FPGA implementation over a single core of a 3.0 Ghz Core 2 duo, but their implementation is limited to 273-nucleotide-long sequences [33].

3.3 Algorithm profiling

The first step before a GPU implementation is a code analysis to see which part is the most time-consuming, and if it can be parallelized. Due to Amdahl's law, it is of the outmost importance to check that the non-parallelizable portion is negligible. Even a small 5% non parallelizable part would seriously impact overall speedup.

We profiled the *hybrid-ss-min* function of the Unafold 3.7 package which computes the minimum free energy of folding of an RNA sequence. Figure 3.5 shows the percentage time spent by the different parts of the algorithm for sequences from 100 to 8000 nt.

For short sequences, computation of *QM*, *QS*, *QP*—corresponding to equations 3.4, 3.5 and 3.3—represents, respectively, about 22%, 20% and 52% of total time. The remainder is composed of the *Final reduction*, computed with equation 3.1 (about 4% total time) and of the initialization phase of the different matrices (approximately 1% of total time). For longer sequences, computation of *QM* gradually becomes more and more important since it has the highest $\mathcal{O}(n^3)$ complexity, and reaches 88% of total time for a 8000 nt sequence.

These results show that in order to obtain a good overall speedup, a GPU implementation needs to parallelize all these different parts, including the *Final reduction* and the initialization step. Fortunately, all these parts are parallelizable, as we shall see in the next section. We can also expect that overall speedup will be different on short sequences and long sequences: long sequences overall speedup will be dominated by the speedup of the *QM* part (which becomes predominant for large sequences), and by that of *QP* for short sequences.

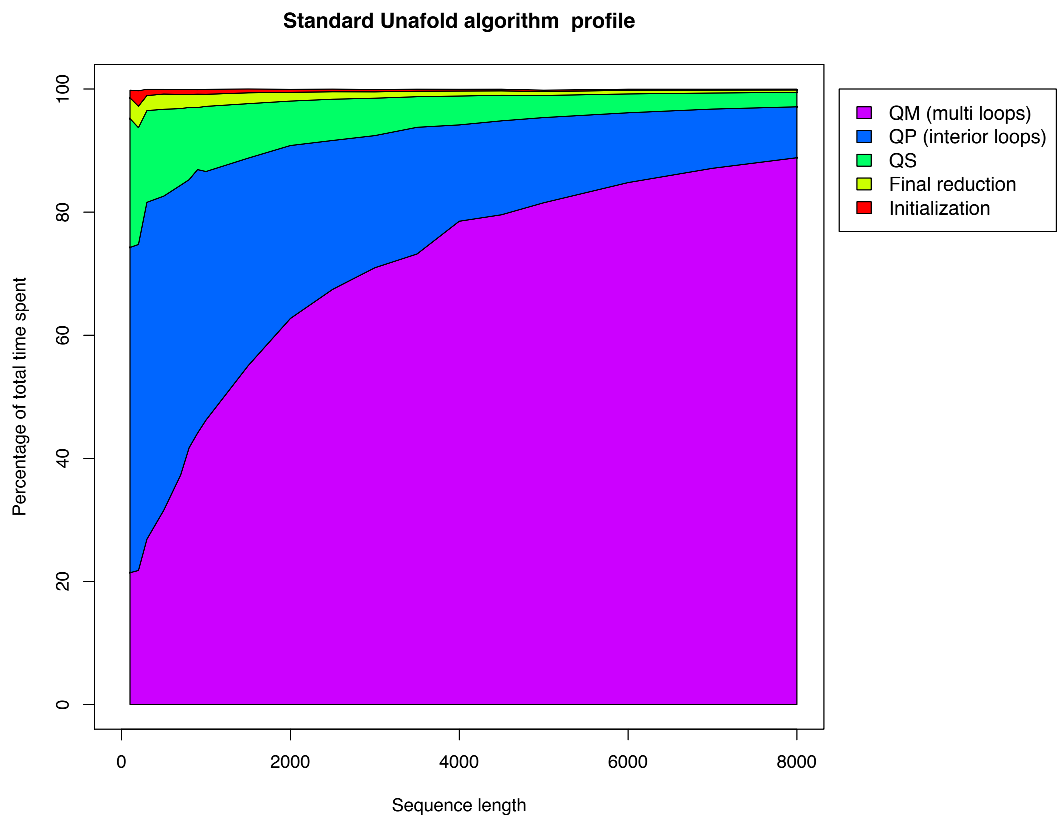


Figure 3.5. Percentage of time spent in the different parts of the *hybrid-ss-min* function of the standard Unafold package computing the minimum free energy of sequences from 100 to 8000 nucleotides. QM is the computation of multi-loops in equation 3.4. It is the only $\mathcal{O}(n^3)$ part of the algorithm, therefore it gradually becomes preponderant. QP, QS and *Final reduction* refer, respectively, to equations 3.3, 3.5 and 3.1

3.4 First implementation

3.4.1 General parallelization scheme

Figure 3.6 shows the data dependencies coming from the recursion (3.3) to (3.5). They imply that, once all previous diagonals are available, all cells of a diagonal can be processed independently. Three kernels are designed for the computation of $QP_{i,j}$, $QM_{i,j}$ and $QS_{i,j}$, corresponding to equations (3.3) to (3.5). Each one computes one full diagonal. The whole matrix is then processed sequentially through a loop over the diagonals. The next step corresponding to equation (3.1) is a combination of reductions (search of the minimum of an array) which is parallelized in another kernel. The pseudo-code for the host side of this parallelization scheme is given in algorithm 1.

Algorithm 1 Host code, first implementation

```
1: Input: sequence of length $N$
2: Output: minimal energy of the sequence
3: for diagonal $d$ in $[1; N]$ do
4:     launch kernel $GPU\_QP(i \in [1; N-d], j = i+d)$
5:     launch kernel $GPU\_QM(i \in [1; N-d], j = i+d)$
6:     launch kernel $GPU\_QS(i \in [1; N-d], j = i+d)$
7: end for
8: launch kernel GPU_reduction $E_N$
```

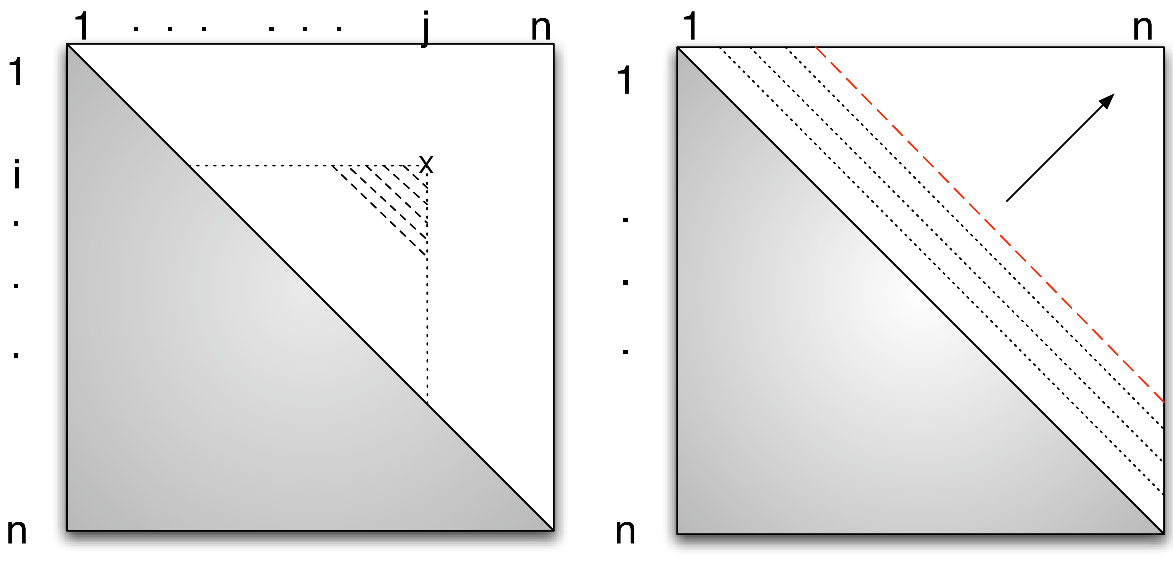


Figure 3.6. Left: Data dependency relationship. Each cell of the matrix contains the three values QP, QM and QS. As subsequence i, j is the same as subsequence j, i only the upper half of the matrix is needed. The computation of cell i, j needs the lower left dashed triangle and the two vertical and horizontal dotted lines. **Right: Parallelization.** According to the data dependencies, all cells along a diagonal can be computed in parallel from all previous diagonals.

3.4.2 $\mathcal{O}(n^2)$ part: Internal loops

The third term of equation 3.3 corresponds to the computation of internal loops. Their size is usually limited to $K = 30$, which means the minimization is conducted over the domain defined by (k, l) with $i < k < l < j$ and $(k - i) + (j - l) < K$. This domain is represented by the dashed triangle in figure 3.6. This term is therefore responsible in total for $\mathcal{O}(n^2 \cdot K^2)$ computations. The kernel GPU_QP is designed to compute all cells $QP(i, j)$ over a diagonal d.
One major issue with this kernel is the divergence caused by the two branches in equation 3.3: if pair i, j is not allowed then $QP(i, j)$ is assigned the score ∞ and no computations are required. Different cells of the diagonal are following different code paths, causing a huge loss in performance. To tackle this problem we compute on CPU an index of all diagonal cells i, j where base pair i, j is allowed. The kernel computation domain is then the set of cells pointed by this index.

3.4.3 $\mathcal{O}(n^3)$ part: Multi-loops

Equation 3.4 computes the most stable multi-loop formed by subsequence i, j, and requires $\mathcal{O}(n)$ computations per cell, thus $\mathcal{O}(n^3)$ overall for the n^2 cells of the matrix. It requires the knowledge of the line and column represented by dotted lines in figure 3.6 and is simply computed by finding the minimum of $(QS_{i,k} + QS_{k+1,j})$ over $k \in [i; j]$. Although it is a simple computation, it is the only $\mathcal{O}(n^3)$ part of the algorithm and as such the most important one for large sequences.

First implementation

The obvious way to implement it on GPU is to compute cells of a diagonal in parallel, with the reduction computation of each cell being itself parallelized over several threads. This way is implemented in kernel GPU_QM, shown in algorithm 2 and used in our first implementation. The scheme is simple: BX threads cooperate to compute the reduction, with each thread sequentially accessing memory locations BX cells apart, so that the memory accesses to $QS(i, k)$ are coalesced among threads. The code shows that each thread does 2 operations for every 2 global memory elements accessed, on line 7. This is a very poor ratio, worsened by the fact that accesses $QS(k + 1, j)$ are not coalesced. Performance for this piece of code is measured at only 4.5 GOPS (Giga Operations Per Second) and a bandwidth usage of 17 GB/s for a 10kb sequence. It is obviously severely bandwidth limited. It may be possible to improve and coalesce the second reference in the loop body by storing QS and its transpose in global memory, but we did not pursue this approach since tiling leads to significantly higher performance as we shall see next.

Algorithm 2 QM kernel, first implementation

```
1: **BlockSize**: One dimension = $BX$
2: **Block Assignment**: Computation of one cell $QM_{i,j}$
3: $tx \leftarrow threadIdx.x$
4: $k \leftarrow i + tx$
5: Shared_data[$tx$] $\leftarrow \infty$
6: **while** $k < j$ **do**
7:     Shared_data[$tx$] $\leftarrow$ min (Shared_data[tx] , QS(i,k) + QS(k+1,j))
8:     $k \leftarrow k + BX$
9: **end while**
10: Syncthreads
11: $QM_{i,j} \leftarrow$ Minimum of array Shared_data
```

3.5 Multi loops second implementation

This section explains the second parallelization scheme with a new efficient reordering of the computation of the multi loops part, allowing a tiled implementation with good data locality.

3.5.1 Reduction split and reordering

The main issue with our first implementation is that it requires a lot of memory bandwidth, as each cell of the diagonal requires $\mathcal{O}(n)$ data, and there is no reuse of data between the computation of different cells. Theoretically there is a lot of data reuse, as a single value $QS(i,k)$ will be used for the computation of all cells $QM(i,j)$ with $j \in [k;n]$. Unfortunately the dependency pattern makes it *a priori* impossible to take advantage of this, as the set of cells $QM(i,j)$ $j \in [k;n]$ are located on different diagonals and thus computed in separate kernel calls.

In our “tiled implementation,” we designed a way to exploit this data reuse. The key observation is that the computation of a single cell is a minimization over $i < k < j$, which can be split into several parts. Our approach is inspired by the systolic array implementation of the optimum string parenthesization problem by Guibas-Kung-Thompson [27, 65], which completely splits reductions in order to have a parallelization scheme working for an array of $\mathcal{O}(n^2)$ processors. Although not going as far, our approach exploits the same idea: we split and reorder the reductions, in order to obtain a parallelization scheme better suited for GPU.
Let's say we have already computed all cells up to a given diagonal d_0, meaning that all cells (i,j) such that $j < i + d_0$ are known. For a given (i_1, j_1) outside of the known area (marked with an x in Figure 3.7) with $j_1 = i_1 + d_0 + T, T > 0$, the whole computation $\min\limits_{i_1<k<j_1} (QS_{i_1,k} + QS_{k+1,j_1})$ is not possible as many QS values are still unknown.
Nevertheless, we can compute a *part* of the minimization for which QS values are already known: for $\delta_1 < k < \delta_2$ with $\delta_1 = j - d_0$ and $\delta_2 = i + d_0$ (shown as the pair of thin blue lines in the left part of Figure 3.7). Similar “blue portions” of the accumulations of

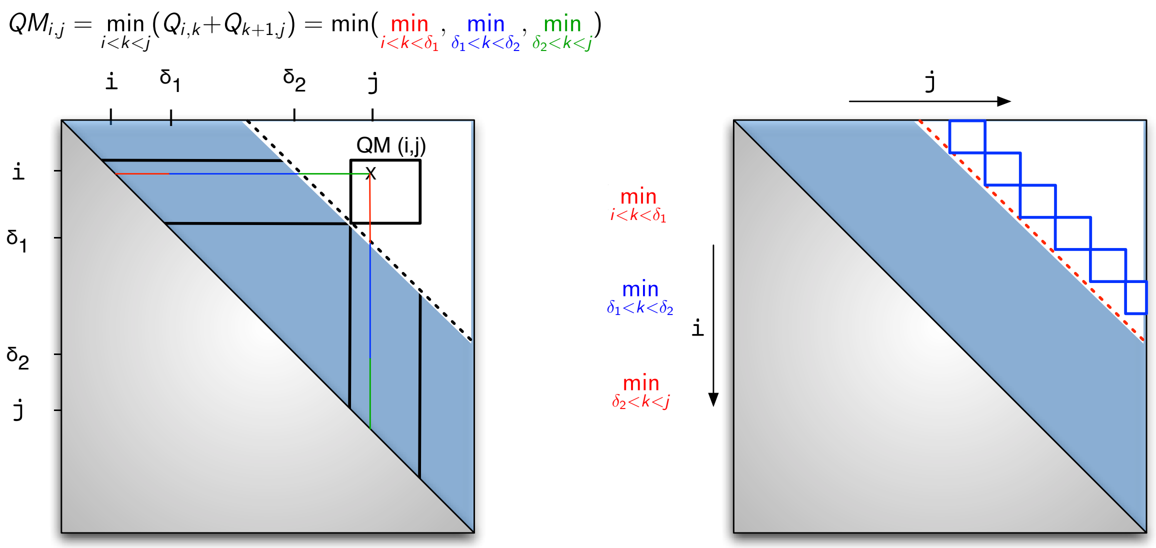

$QM_{i,j} = \min_{i<k<j}(Q_{i,k}+Q_{k+1,j}) = \min(\min_{i<k<\delta_1}, \min_{\delta_1<k<\delta_2}, \min_{\delta_2<k<j})$
i
δ1
δ2
j
QM (i,j)
$\min_{i<k<\delta_1}$
$\min_{\delta_1<k<\delta_2}$
$\min_{\delta_2<k<j}$

Figure 3.7. Left: Reduction split. All cells in light blue are already available. For a point (i, j) located outside of the known area, the middle part of the reduction represented by the thin blue line can be computed. **Right: Tiling scheme.** The middle part of the reduction is done "in advance" in the blue tiles. The remaining part is done on the red cells, diagonal after diagonal.

all the points in the tile can be computed in parallel. This means that by taking apart the range that violates the dependency pattern, we can compute in parallel a tile of cells and thus benefit from a good memory reuse scheme. Figure 3.7 shows such a tile and how the minimization is split.

3.5.2 Tiling scheme

We still move sequentially diagonal after diagonal, but we compute "in advance" the middle part of the minimization of equation 3.4 in tiles. This tiled computation is implemented in the kernel *GPU_QM_tiled*, and called every *Tile_Size* diagonals. The remaining part of the minimization is done in kernel *GPU_finish_QM* called every diagonal, as shown in algorithm 3 and depicted with dotted red line in the right part of Figure 3.7.

3.5.3 GPU implementation

Thanks to the tiled implementation the GPU program is very efficient: each tile is computed by a thread block, which can sequentially load into shared memory blocks of lines and columns, similar to a standard tiled matrix-matrix multiplication. Kernel code of *GPU_QM_tiled* is shown in algorithm 4. *QS* values are stored in shared memory lines $9, 10$ and reused *BS* times each, on line 13. Bandwidth requirement is therefore reduced by a factor of $Tile\_Size = BS$ compared to the first implementation. Performance of this kernel is now 90 GOPS with a bandwidth usage of 23 GB/s.

Algorithm 3 Host code, tiled implementation

```
1: **Input**: sequence of length $N$
2: **Output**: minimal energy of the sequence
3: for diagonal $d$ in $[1; N]$ do
4:     launch kernel $GPU\_QP(i \in [1; N-d], j = i+d)$
5:     if ($d$ modulo Tile_Size == 0) then
6:         launch kernel $GPU\_QM\_tiled(d)$
7:     end if
8:     launch kernel $GPU\_finish\_QM(i \in [1; N-d], j = i+d)$
9:     launch kernel $GPU\_QS(i \in [1; N-d], j = i+d)$
10: end for
11: launch kernel $GPU\_reductionE_N$
```

Algorithm 4 QM kernel, tiled implementation

```
1: BlockSize: Two dimension $= BS \cdot BS$
2: Block Assignment: Computation of “middle part” of $QM$ for a tile of $BS \cdot BS$
   cells, with upper left corner $(i_0, j_0)$
3: $tx \leftarrow threadIdx.x$, $ty \leftarrow threadIdx.y$
4: $iter \leftarrow$ beginning of “middle part”
5: $bound \leftarrow$ end of “middle part”
6: $temp \leftarrow \infty$
7: while $iter < bound$ do
8:     Load blocks in shared memory:
9:     Shared_L $[tx][ty] \leftarrow QS(i_0 + ty, iter + tx)$
10:    Shared_C $[tx][ty] \leftarrow QS(iter + 1 + ty, j_0 + tx)$
11:    Synchthreads
12:    for $k = 0$, $k \leq BS$, $k++$ do
13:        temp $\leftarrow$ min(temp,Shared_L [ty][k] $+$Shared_C [k][tx] )
14:    end for
15:    $iter \leftarrow iter + BS$
16: end while
17: $QM_{i_0+tx, j_0+ty} \leftarrow$ temp
```

3.6 CPU new implementation

Vectorization

The tiled approach we developed for GPU also has significant advantages for the CPU version. In the standard sequential Unafold implementation, the computation of QM is not very efficient because of the memory accesses to $QS_{k+1,j}$ which are made column-wise. Such accesses are inefficient as each data is in a different cache line.

A simple solution, implemented in the Vienna RNA package, is to store the column in a temporary array so that consecutive accesses are in the same cache line. This temporary array can be loaded once and reused many times since in the sequential code the matrix is filled column after column, from bottom to top.

Our tiled approach is also a good fix to this problem: the tiling scheme naturally provides a very good temporal locality, each QS cell is reused many times for the computation of other cells of the tile. Moreover, this tiling scheme also provides an obvious way to implement an efficient vectorization: four consecutive cells of a tile are computed at the same time.

Additionally, a second level of tiling is applied. While the first level permits data to fit in the CPU L1 cache, the second level permits data to fit in the CPU SIMD vector registers. Data is loaded once in a SIMD register and reused several times for the computation of cells inside the second sub-tile. This way the overhead cost of loading data in SIMD registers is amortized.

The code for vectorized implementation is presented in algorithm 5. Variables *VEC* denote SIMD registers able to contain four 32-bit values. The second level of tiling operates on a 4 by 4 sub-tile, lines 6 and 7. The sixteen corresponding QM values are stored in the 4 SIMD registers *VEC0* to *VEC3*. Each QS value loaded in a SIMD register, at line 14 and 16, is reused 4 times. The innermost loop at line 15 is unrolled in the implementation.

A larger sub-tile would exhibit more data reuse and thus would amortize loading costs more. However, the number of SIMD registers is very restricted, generally 8 or 16. Our implementation only uses 6 registers and therefore works well on all CPUs with SSE. The *min* operation was introduced in SSE4, but can also be decomposed in four native SSE2 instructions. We provide a flag at compilation to select the correct version.

Figure 3.9 shows a break down of the time spent in the different part of the algorithm for our new vectorized implementation. The time spent in the tiled vectorized part is now only 30% of total time for a 8Kb sequence. It has however the dominating $\mathcal{O}(n^3)$ complexity and is therefore growing, although at a slow rate.

Multi core parallelization

We implemented the multi core parallelization via OpenMP across the different tasks of a diagonal. Thanks to the ease of use of OpenMP this was effectively a 5-minute transformation of the code.

Algorithm 5 CPU vectorized algorithm

```
1: **Input**: A tile of dimension $BS\_CPU \cdot BS\_CPU$, with upper left corner $(i_0, j_0)$ .
2: **Output**: Computation of “middle part” of $QM$.
3: $start \leftarrow$ beginning of “middle part”
4: $bound \leftarrow$ end of “middle part”
5: **for** $tk = start$, $tk < bound$, $tk += BS\_CPU$ **do**
6:    **for** $ti = 0$, $ti < BS\_CPU$, $ti += 4$ **do**
7:       **for** $tj = 0$, $tj < BS\_CPU$, $tj += 4$ **do**
8:          *Load a 4 by 4 sub tile in SIMD registers:*
9:          $VEC0 \leftarrow QM(ti + i_0, tj + j_0), \ldots, QM(ti + i_0, tj + j_0 + 3)$
10:         $VEC1 \leftarrow QM(ti + i_0 + 1, tj + j_0), \ldots, QM(ti + i_0 + 1, tj + j_0 + 3)$
11:         $VEC2 \leftarrow QM(ti + i_0 + 2, tj + j_0), \ldots, QM(ti + i_0 + 2, tj + j_0 + 3)$
12:         $VEC3 \leftarrow QM(ti + i_0 + 3, tj + j_0), \ldots, QM(ti + i_0 + 3, tj + j_0 + 3)$
13:         **for** $k = 0$, $k < BS\_CPU$, $k++$ **do**
14:            $VEC\_C \leftarrow QS(tk + k, tj + j_0), \ldots, QS(tk + k, tj + j_0 + 3)$
15:            **for** $i = 0$, $i < 4$, $i++$ **do**
16:               $VEC\_L \leftarrow$ four times the same value : $QS(ti + i_0 + i, tk + k)$
17:               *Core SIMD computation:*
18:               $VECi \leftarrow min(VECi,\ VEC\_C + VEC\_L)$
19:            **end for**
20:         **end for**
21:         *Store back the four by four sub tile results*
22:         $QM(ti + i_0 \text{ to } ti + i_0 + 3, tj + j_0 \text{ to } tj + j_0 + 3) \leftarrow VEC0, VEC1, VEC2, VEC3$
23:      **end for**
24:   **end for**
25: **end for**
```

3.7 Results

Our GPU implementation uses exactly the same thermodynamic rules as Unafold, thus the results and accuracy obtained on GPU is exactly the same as the standard CPU Unafold function. We therefore only compare the execution time between programs, including the host-to-device transfer for the GPU implementation. Our tests are run on a Xeon E5450 @ 3.0GHz and a Tesla C1060. CPU code is running on fedora 9, compiled with gcc 4.3.0 with -O3 optimization level.

3.7.1 Tiled implementation

Figure 3.8 A shows the speedup obtained by the tiled implementation on a Tesla C1060 over the standard Unafold algorithm running on one core of a Xeon E5450 for randomly generated sequences of different lengths. Sequences are generated with each of the four letters being equiprobable. Tests are run with a large enough number of sequences ensuring GPU starting costs are amortized. The speedup starts at around 20x for short sequences and goes up to 150x for large sequences. For small sequences, the $\mathcal{O}(n^2)$ part of the algorithm is preponderant. Thus, the overall speedup is roughly that of the kernel computing internal loops, called *GPU_QP* and explained in section 3.4.2. As sequence size increases, the $\mathcal{O}(n^3)$ multi-loop computation becomes more important so the speedup approaches that of kernel *GPU_QM_tiled* which is very efficient thanks to our tiled code.

In order to do a fair, "apples-to-apples" comparison, we applied our tiling technique on the CPU and used SSE vectorized instructions to get the full potential of CPU performance. Execution speed of the vectorized part of the code is 5.4 GOPS for a 10 kb sequence, which is quite high considering first GPU implementation speed was only 4.5 GOPS. Speedup obtained by the second GPU implementation against this optimized code is presented figure 3.8 B. Here, speedup is more stable across sequence lengths as like the GPU, CPU tiled code is very efficient. GPU speedup of the $\mathcal{O}(n^3)$ part of the code compared to CPU code using same tiling technique and vectorized code is 90 GOPS / 5.4 GOPS = 16.6x. This is therefore the theoretical maximum speedup that can be achieved when running on very large sequences when the $\mathcal{O}(n^2)$ part—exhibiting different speedup—becomes negligible.

The global memory to device bandwidth GPU usage of the $\mathcal{O}(n^3)$ part of the code computing multi loops was measured at 23 GB/s. The maximum bandwidth is around 100 GB/s for the GPU used, the code is not bandwidth limited. Although we do not attain peak GPU 312 GOPS computational throughput, the code is actually computation bound.
First, the real peak GOPS is only 66% of the maximum when doing computation from shared memory, consequently the maximum is only 205 GOPS [40, 77]. The remainder is probably explained by some inefficiencies in the code.

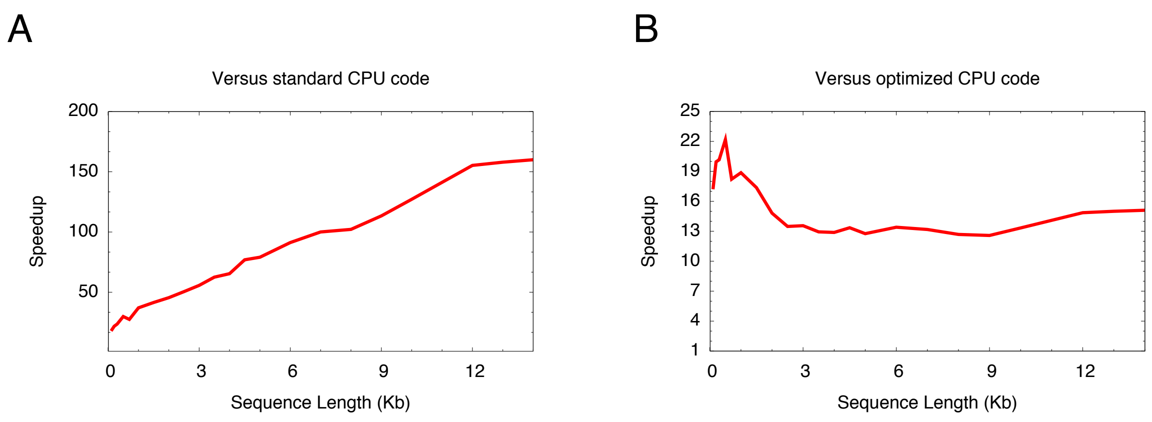


Figure 3.8. Tesla C1060 speedup against one core of Xeon E5450 @ 3.0Ghz for various sequence lengths.**A:** Speedup against standard Unafold code. **B:** Speedup against tiled and vectorized CPU code.

3.7.2 Overall comparison

Figure 3.11 compares performance on a 9781-nucleotide-long virus sequence on a larger set of programs: the first untiled GPU implementation, the tiled one, the standard Unafold CPU code [56], the GTfold algorithm [57] running on 8 CPU cores, and the new SSE-optimized CPU code running on one or eight threads.

We can see that our new CPU code is running 10 times faster than the regular Unafold code, thanks to both SSE and better memory locality of the tiled code. Even against this fast CPU code, the Tesla C1060 is still more than 11 times faster than the Xeon. It is also worth noting the 6x speedup obtained between the GPU tiled and untiled code.

For the CPU tiled and vectorized however, going from one to eight threads yielded a 6.2x speedup. We measured memory bandwidth usage of the tiled part of the code at 0.6 GB/s when running on one CPU core, therefore running 8 cores at full potential would require 4.8 GB/s, exceeding the memory bandwidth limit of the Xeon E5450. As a result, the single GPU tested is approximately 2 times faster than this 8-core CPU system.

This may seem a low speedup, especially to those used to seeing hyped up GPU performance gains. However, this is a more reasonable expectation – for example the commercially released CULA package for linear algebra is “only" about 3-5 times faster than a vectorized and optimized linear algebra library on a 4-core CPU.
Also note that when dealing with huge numbers of sequences, GPU computing provides significant advantages for this application; when considering performance/watt, or the ability to stack several GPU cards in a single system, hence rivaling with a small cluster at a fraction of the cost.

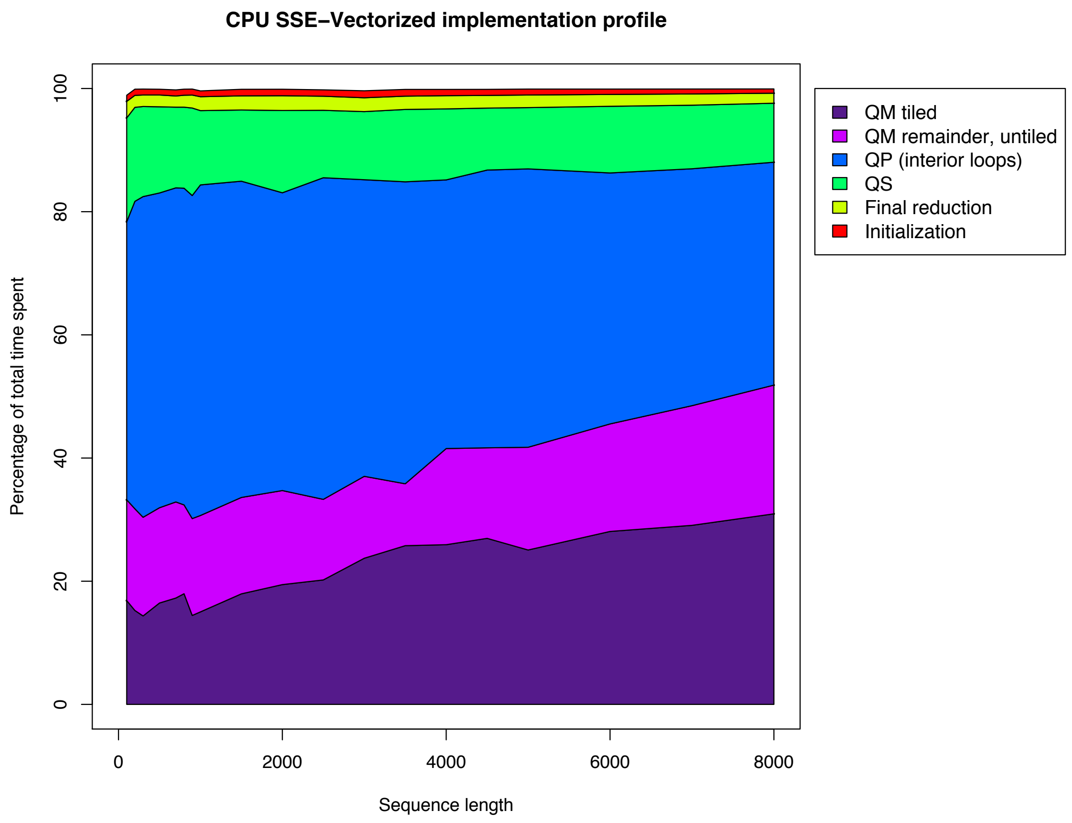
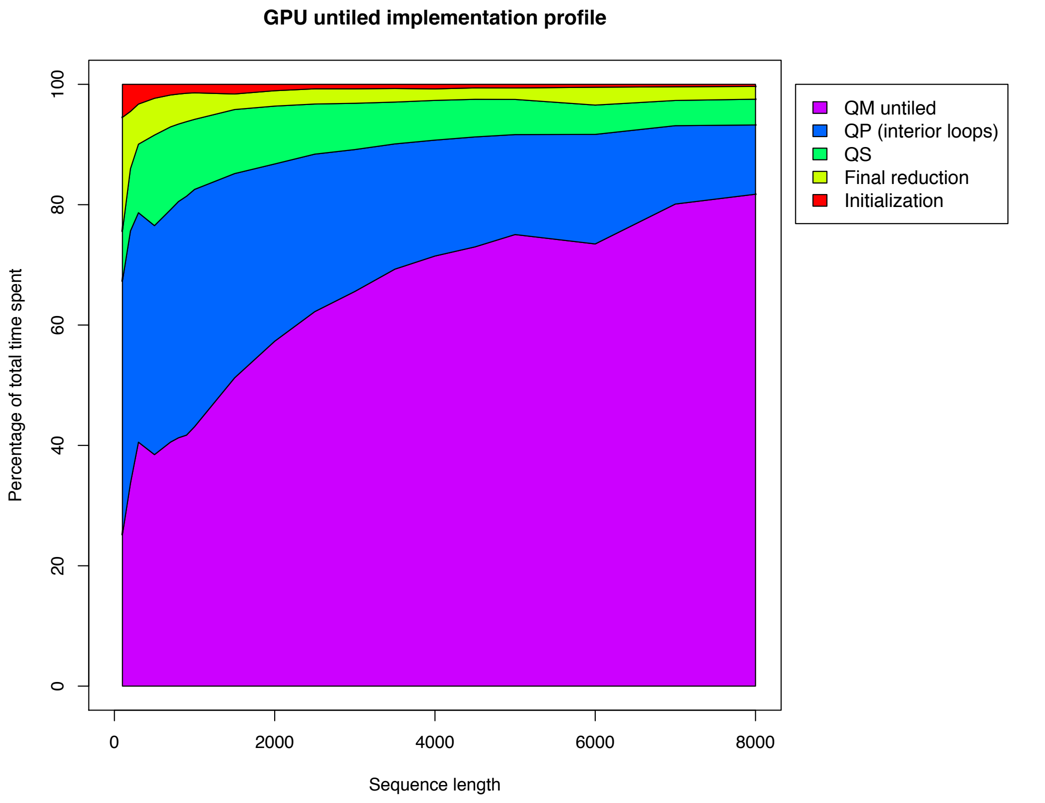


Figure 3.9. Percentage of time spent in the different parts of our *SSE* vectorized implementation of *hybrid-ss-min* function, computing the minimum free energy of sequences from 100 to 8000 nt. *QM* is the computation of multi-loops in equation 3.4. The dark purple segment representing the tiled portion of the computation of *QM* is the only $\mathcal{O}(n^3)$ part of the algorithm. Its vectorized SSE implementation is so efficient compared to other parts of the code—of complexity $\mathcal{O}(n^2)$—that its proportion is growing only very slowly. Even for large 8000 nt sequences, it represents only 30% of total execution time.



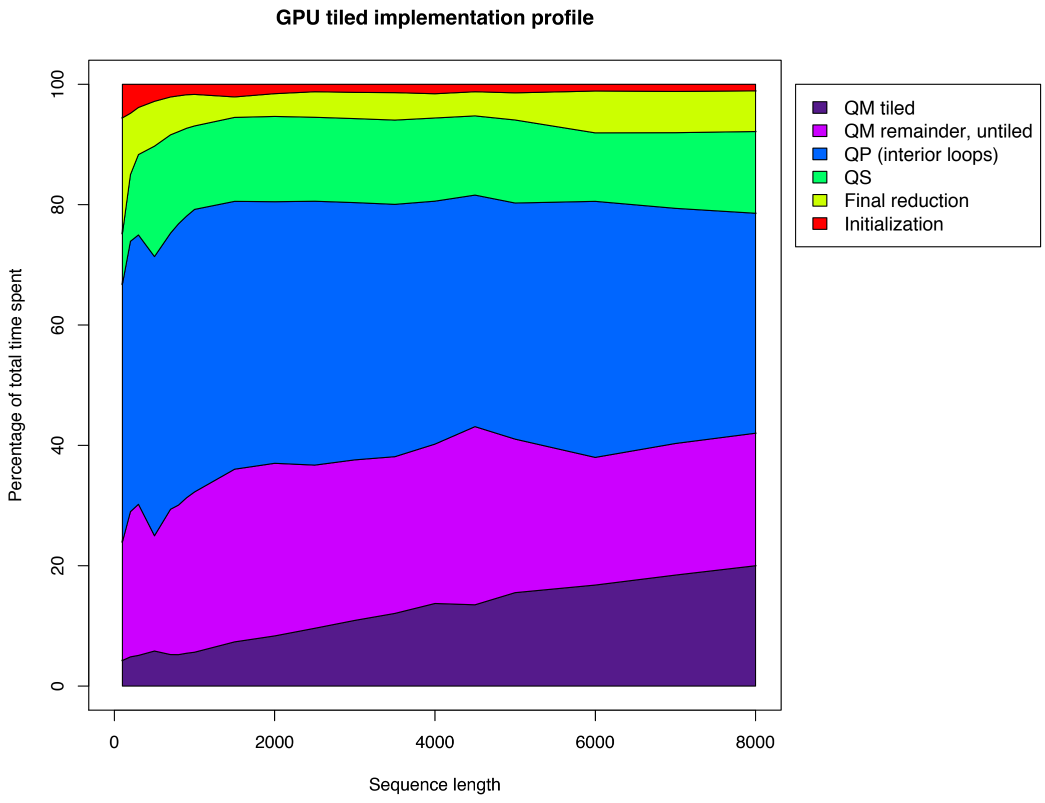


Figure 3.10. Percentage of time spent in the different parts of our GPU implementation of *hybrid-ss-min* function. **Top.** Profile for the untiled first implementation. **Bottom.** Profile for the tiled second implementation.

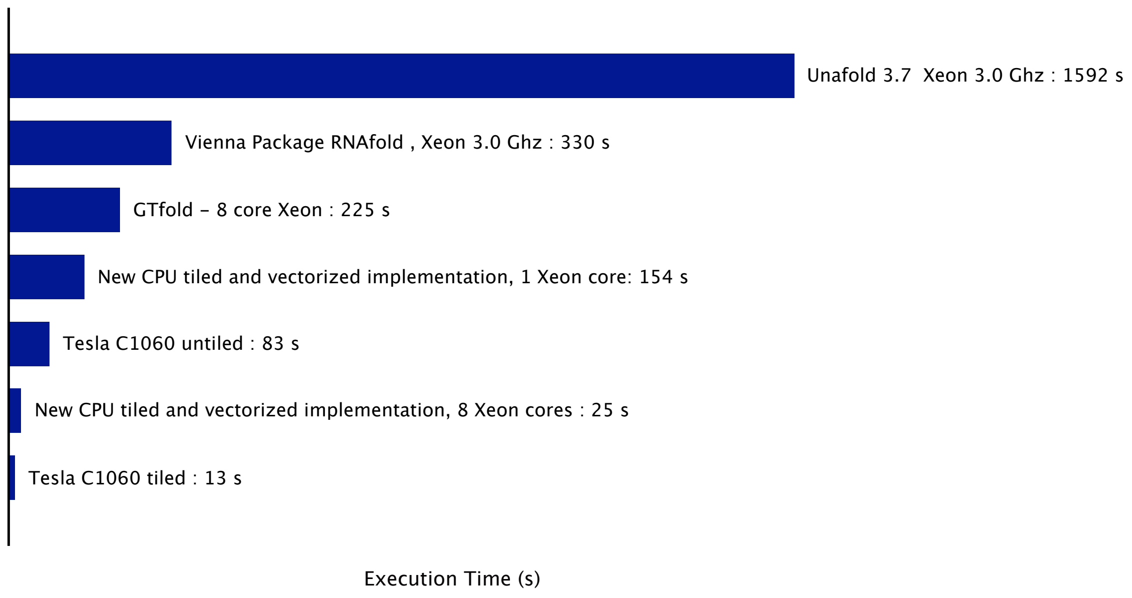


Figure 3.11. Execution time on a 9781-nucleotide-long virus sequence. Various implementations are compared: standard Unafold and Vienna packages, the GTfold program conducting multi core parallelization, our new CPU tiled and vectorized code, and GPU implementation with or without tiling. It is striking to see the difference in speedup we obtain when changing the reference: GPU tiled implementation is 122 times faster than standard Unafold code, but only 1.9 times faster than a eight core CPU system used at its full potential.

3.7.3 Multi-GPU implementation

Usually many independent sequences need to be folded. In that case, this provides a coarse grain level of parallelism. We exploit this level to parallelize work across several GPUs. Our program send sets of sequences to the different GPUs available. This is easily implemented with the Pthreads API and by creating a CUDA context in each one of them. Synchronization is only needed when the different threads write in the same result file. Since no synchronization or data transfer is needed between GPUs, performance effectively scales linearly with the the number of GPU used.

3.8 Related algorithms

We also investigated the computation of sub-optimal foldings and the partition function of a sequence, whose algorithms are similar to the one presented for optimal foldings in section 3.1. Sub-optimal foldings and partition function are very useful to give a more accurate view of a sequence ensemble of possible foldings, since the unique "optimal" folding is not necessarily always representative of biological reality.

3.8.1 Sub-optimal foldings

Finding the optimal folding of a sequence is a good mathematical result, but is in practice often not enough. First, there can be many foldings within a few percents of the minimum free energy, and these foldings can be very different form the optimal one [86]. Moreover, since folding takes place during transcription, RNA sequences are in fact often kinetically trapped in locally optimal structures. Therefore having an overview of the possible sub-optimal foldings of a sequence is essential to many bioinformatic studies.

Steger *et al.* [75] first observed that by computing additional values representing the minimum energy of the "excluded fragments", one could compute the minimum energy of a sequence containing a given base pair i, j. In section 3.1, $QP_{i,j}$ was introduced as the optimal score of of subsequence $i \dots j$ when i, j form a base pair. Similarly, Steger *et al.* [75] defined[1] $QP_{i,j}^{ext}$ as the sum of optimal score of both subsequences $1 \dots i$ and $j \dots n$. $QP_{i,j} + QP_{i,j}^{ext}$ then represents the minimum energy over the whole sequence of a structure containing base pair i, j.

Zuker [84] then used this extension to compute a collection of sub-optimal representative foldings. For every pair i, j, his algorithm computes the optimal structure containing this pair. A distance criterion also ensures to output structures that are not too similar. This is implemented in the *hybrid-ss-min* function (–mfold option) of the Unafold package [56].

Following this seminal work, Wuchty *et al.* [82] then introduced an algorithm to compute the exhaustive list of suboptimal foldings within a Δ band of the MFE. This

[1]originally with notation $V(j, i)$ in [84], as opposed to $V(i, j)$ for the internal part. We prefer to use an ^{ext} exponent to denote the *external* part of a subsequence.

is implemented in the *RNAsubopt* function of the ViennaRNA package [29]. Finding relevant sub-optimal foldings is still an ongoing field of research [7, 52, 72].

In the following, we study the sub-optimal folding algorithm introduced by Zuker [84] and examine its GPU implementation.

Zuker sub-optimal folding Algorithm

Algorithm 6 is the Zuker algorithm as implemented in the *hybrid-ss-min* function of the Unafold package. The list $\mathcal{L}$ is the collections of all base pairs i, j along with the minimum free energy of any folding that contains the given base pair. It contains therefore at most $n^2/2$ elements.
A quick profiling of the code showed that lines 1 and 3 are the most time consuming parts of the algorithm. Since the list contains at most $n^2/2$ elements, the sort (line 7) complexity conducted with an efficient merge-sort[2] is in $\mathcal{O}(n^2 \log n)$, less than the $\mathcal{O}(n^3)$ complexity required to fill the QP matrix.

GPU computation of line 1 is already detailed in section 3.5. Computation of $QP^{ext}_{i,j}$ can also be conducted recursively as depicted with recursion diagrams 3.12, with the following equations:

With $i < j$:

$$QP^{ext}_{i,j} = \begin{cases} \min \left\{ \begin{array}{l} Es(i,j) + QP^{ext}_{i-1,j+1} \\ \min\limits_{\{k,l \mid k<i,j<l\}} Ei(i,j,k,l) + QP^{ext}_{k,l} \\ QM^{ext}_{i-1,j+1} \\ E_{i-1} + B_{j+1} \end{array} \right\} & \textit{if pair } i \cdot j \textit{ is allowed} \\ \infty & \textit{if pair } i \cdot j \textit{ is not allowed} \end{cases} \tag{3.6}$$

$$QM^{ext}_{i,j} = \begin{cases} \min\limits_{j<k<n} (QS_{j,k} + QS^{ext}_{k+1,i}) \\ \min\limits_{1<k<i} (QS_{i,k+1} + QS^{ext}_{k,i}) \end{cases} \tag{3.7}$$

$$QS^{ext}_{i,j} = \min\left\{QM^{ext}_{i,j}, \min(QS^{ext}_{i-1,j}, QS^{ext}_{i,j+1}), QP^{ext}_{i,j}\right\} \tag{3.8}$$

Quantities $QS, Es(), Ei()$ are those already defined in section 3.1.2. E_i is the MFE of subsequence $1 \dots i$ as defined in equation 3.1. B_j is a new value corresponding to the MFE of subsequence $j \dots n$. It is the symmetric of E_i and computed the exact same way. $QP(i,j)^{\,ext}, QS(i,j)^{\,ext}, QM(i,j)^{\,ext}$ are, respectively, the equivalent of the QP, QS, QM values but referring to the exterior part of subsequence $i \dots j$.

Equation 3.7 is particularly noteworthy: it is decomposed the same way as QM in equation 3.4, however since the exterior part of $i \dots j$ is composed of the two parts $1 \dots i$ and $j \dots n$ the decomposition must also be broken down in two parts. Although still in $\mathcal{O}(n^3)$, the computation of $QM^{\,ext}$ is roughly twice as long as that of QM.

[2]As of version 3.8, Unafold implements this sort with a quadratic sort, resulting in overall $\mathcal{O}(n^4)$ complexity. Unafold implementation is obviously designed to compute small sequences only.

Algorithm 6 Zuker sub optimal foldings algorithm

```
1: Compute $QP_{i,j}$
2: $QP_{i,j}$ is the minimum energy of the subsequence $i \ldots j$ when $i, j$ are paired.
3: Compute $QP_{i,j}^{ext}$
4: $QP_{i,j}^{ext}$ is the minimum energy of the exterior of subsequence $i \ldots j$ when $i, j$ are paired.
5: $\Delta G(i,j) = QP_{i,j} + QP_{i,j}^{ext}$
6: $\Delta G(i,j)$ is the the minimum energy of any folding that contains the $i, j$ base pair.
7: Sort the list $\mathcal{L}$ of triplets $\{i, j, \Delta G(i,j)\}$ in order of increasing $\Delta G(i,j)$.
8: while $\mathcal{L} \neq empty$ do
9:     Compute the optimal folding containing the $i, j$ pair at the top of the list $\mathcal{L}$
10:    Remove from $\mathcal{L}$ all base pairs in the computed folding as well as those within a distance of $W$ base pairs.
11:    Output computed folding if it contains at least W base pairs not present in previous foldings
12: end while
```

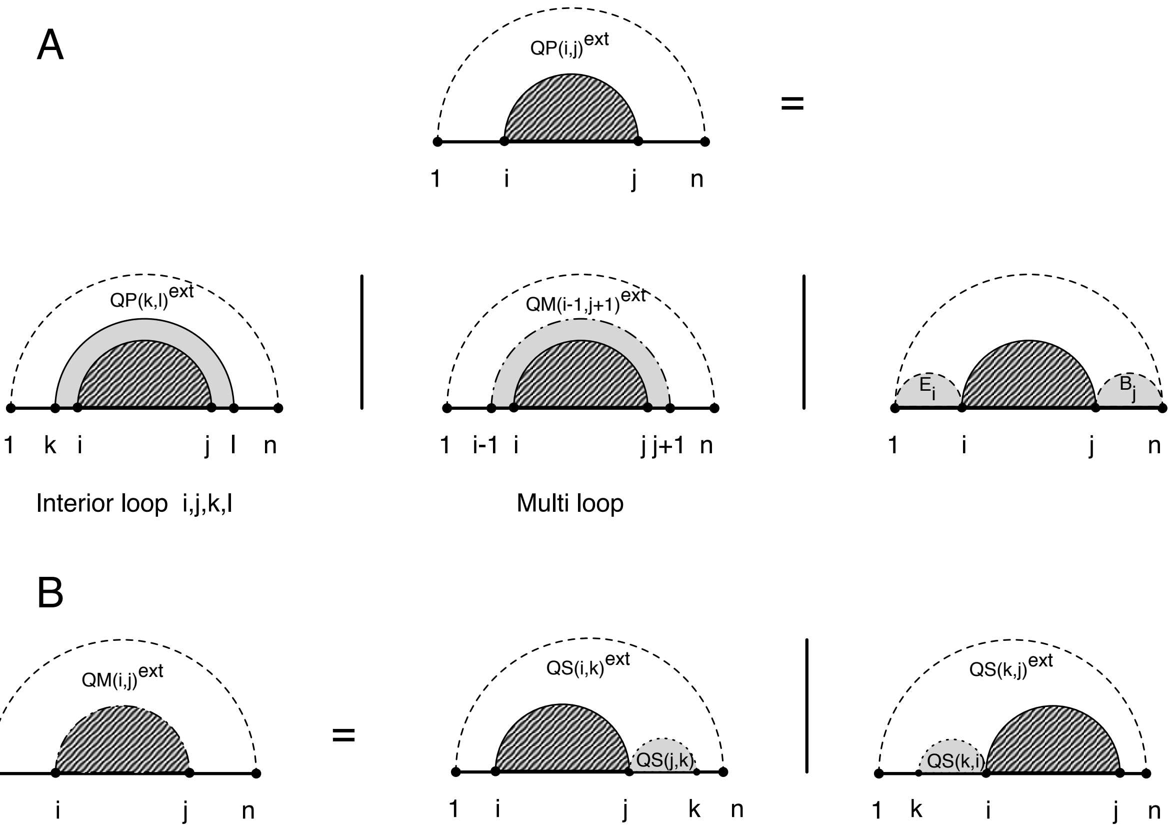


Figure 3.12. Recursion diagrams for the exterior part of $i \cdots j$. A base pair is indicated by a solid curved line, and a dashed curved line represents a subsequence with terminal bases that may be paired or unpaired. Shaded regions are parts of the structure fixed by the recursion, while white regions denote sub-structures whose scores have already been computed and stored in previous steps of the recursion. The dashed region represents the interior segment enclosed by bases i and j which is not considered here. **A.** Recursion for $QP_{i,j}^{ext}$ representing the MFE of the external part of subsequence $i \ldots j$ when i, j is a base pair. **B.** Recursion for $QM_{i,j}^{ext}$ representing the MFE of the external part of subsequence $i \ldots j$ when there is a multi-loop in the external part.

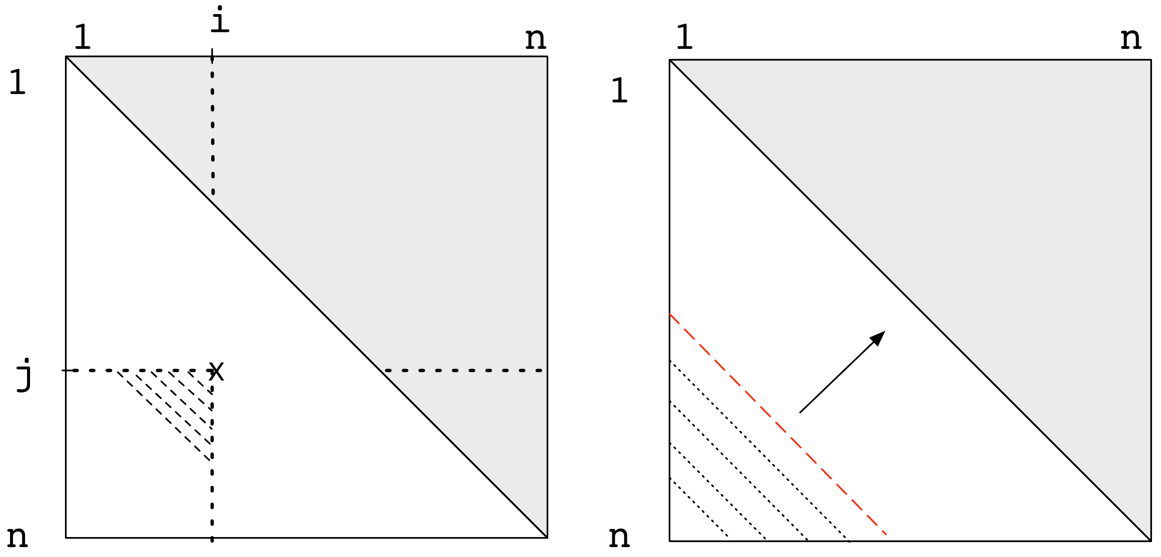


Figure 3.13. Left: Data dependency relationship. The lower left triangle of the matrix contains the three values QP^{ext}, QM^{ext} and QS^{ext}. The computation of cell j, i needs the lower left dashed triangle and the vertical and horizontal dotted lines. The computation requires values in the upper right triangle, i.e the algorithm must first fill the values QP, QM and QS of the upper triangle before beginning the computation of the lower left triangle. **Right: Parallelization.** According to the data dependencies, all cells along a diagonal can be computed in parallel from all previous diagonals.

GPU implementation

The GPU implementation is conducted in a similar way as that of the minimum energy of folding. We use the same general scheme as the one implemented in the Unafold package: a square matrix is allocated to hold the $QM, QP, QS, QM^{ext}, QP^{ext}, QS^{ext}$ values. The upper right triangle is used to store the *internal* values, while the lower left triangle is used to store the *external values.* With $i < j$, cell (i, j) of the matrix contains $QM(i, j), QP(i, j), QS(i, j)$, and cell (j, i) contains[3] $QM(i, j)^{ext}, QP(i, j)^{ext}, QS(i, j)^{ext}$.

Implementation of equations 3.8 and 3.6 is the same as for the *internal* case. Figure 3.14 details equation 3.7 which is slightly different. Tiling is conducted the same way, the only difference is that data needed are both in the upper and lower parts of the matrix. Special care is needed to handle properly the boundary cases, i.e tiles cut because of their proximity to the diagonal.

Results

We compared our GPU implementation of the Zuker sub-optimal foldings algorithm with the the standard UNAfold 3.8 package on sequences of lengths up to 8 kb. We

[3]This implementation probably explains the original $V(j, i)$ notation to refer to *external* fragments [84].

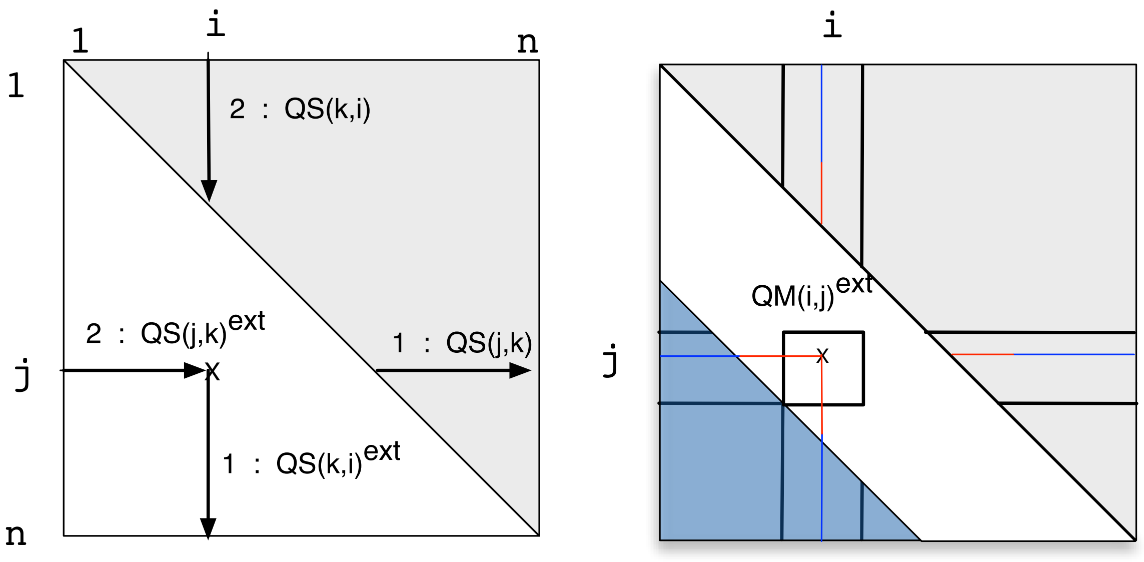


Figure 3.14. Tiling implementation of QM^{ext} computation. Left: QM^{ext} decomposition. The two steps of equation 3.7 are shown. **Right: Tiling.** The red line shows the part of the minimization which is taken apart so that elements of a tile can be computed in parallel in a GPU block of threads. This red part is done in a later step, in the regular untiled way.

also implemented an SSE-vectorized code with our tiling technique to provide a fair CPU-GPU comparison[4]. Speedup is presented in figure 3.15. The speedup obtained is up to 80x for long sequences compared to the standard UNAfold package, and between 10x and 17x compared to our optimized CPU code.

[4]Our CPU implementation also includes an efficient merge-sort as opposed to the quadratic sort of the UNAfold package for line 7 of algorithm 6.

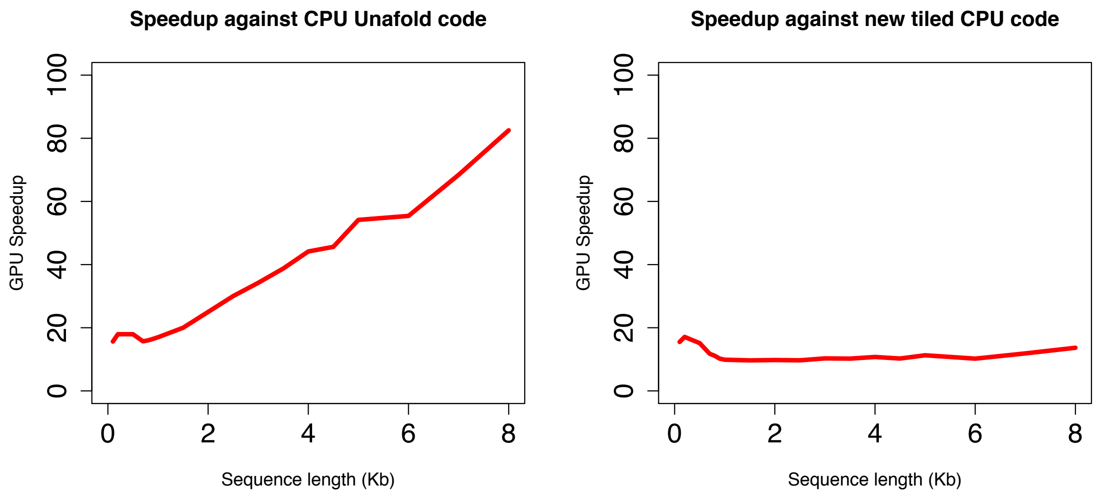


Figure 3.15. Suboptimal folding algorithm GPU speedup. Tesla C1060 speedup against one core of Xeon E5450 @ 3.0GHz for various sequence lengths for the algorithm computing sub-optimal foldings (option –mfold="5,-1,100"). **Left.** Speedup against standard Unafold code. **Right.** Speedup against our new tiled and vectorized CPU code.

3.8.2 Partition function

Contrary to the MFE which gives an information only about the most stable secondary structure of a sequence, the partition function aims to give an information about the ensemble of all possible foldings of a sequence. It therefore provides a more accurate view of a sequence foldings.

The partition function Z is the sum over the set $\mathcal{S}$ of all possible secondary structures:

$$Z = \sum_{s \in \mathcal{S}} \exp^{-\Delta G_s / RT} \tag{3.9}$$

With R the universal gas constant, ΔG_s the energy of a structure s and T the temperature. The equation is best understood with its relation to thermodynamic notions.

Thermodynamic reminder

Consider a general chemical equilibrium with stoichiometric coefficients $\alpha, \beta, \gamma, \delta$

$$\alpha A + \beta B \rightleftharpoons \gamma G + \delta D$$

In solution the equilibrium constant is defined as :

$$K = \frac{[G]^\gamma [D]^\delta}{[A]^\alpha [B]^\beta}$$

With [ ] denoting the concentration of the different species in the solution.
The *free energy* ΔG of the equilibrium is linked to K with the relation:

$$K_{(T)} = \exp^{-\Delta G / RT}$$

The partition function Z can therefore be seen as the equilibrium constant of the reaction:

$$S_{unfolded} \rightleftharpoons S_{folded}$$

Where $S_{unfolded}$ represents an RNA sequence in its unfolded state, and S_{folded} the sequence folded in any of its possible secondary structure. We consequently also have :

$$Z = \frac{[S_{folded}]}{[S_{unfolded}]}$$

Z provides the relative concentration of the folded and unfolded states.
The partition function can also be used to compute the probability of a structure s:

$$P(s) = \frac{\exp^{-\Delta G_s / RT}}{Z}$$

Computation

McCaskill [58] first developed an algorithm to compute the partition function. It is based on the observation that since the energy is additive—energy of sequence $1 \cdots n$ is the sum of energies of $1 \cdots k$ and $k \cdots n$—the partition function is multiplicative. With $1 < k < n$, partition function of the whole sequence $1 \cdots n$ is the product of partition function over $1 \cdots k$ and over $k \cdots n$.

Following work in the community focused on computing the partition function including some classes of pseudoknots [15, 8, 66]. Dirks and Pierce [15] article also provides a very clear introduction on partition function computation.

Partition function can therefore be computed with dynamic programming as the MFE. Similar recurrence equations are used, by replacing the minimization over different structures with a sum, and the sum over the different subsequences by a product.
The decomposition looks the same: quantities $ZP(i,j)$, $Z1(i,j)$, $ZS(i,j)$ are introduced representing the partition function over a subsequence $i \cdots j$ in different cases.

Base pair probability

In the same way as for suboptimal foldings, quantity $ZP(i,j)^{ext}$ can be computed. It represents the partition function of the external part of sequence $i \cdots j$ when i,j forms a base pair. Due to the multiplicativity of the partition function, $ZP(i,j)^{ext} \cdot ZP(i,j)$ represents the partition function over the set of structures that contains the i,j base pair.

Therefore the probability of i,j forming a base pair can be derived as:

$$P(i,j) = \frac{ZP(i,j)^{ext} \cdot ZP(i,j)}{Z}$$

Implementation

Although using similar recurrence equations as the MFE, the computation of the partition function is much more computationally intensive. The main reason is that it operates on double precision floating point numbers instead of integers. Secondly, since Z grows exponentially with the length of the sequence, computation inevitably overflows. Unafold counteracts this with a scaling system, which, while effective, also significantly burdens computations in critical sections of the algorithm.

Our tiling approach introduced in section 3.5 can still be applied here. We implemented again an SSE-vectorized CPU code. The method used is the same as the one already introduced in section 3.5. Some adjustments had to be made in the tiling scheme to fit the particularities of partition function computation, yet globally the method remains the same.

It is nevertheless interesting to see how the SSE-vectorized CPU implementation behave in this case, where computation is conducted on double precision floating point numbers. In the MFE code, SSE was able to compute four 32-bit integers at a time with its 128-bit SIMD vector, here it can compute only two 64 bits floating point values. We therefore expect to have a lower SSE speedup than before.

We also tried a GPU implementation. Preliminary results were not very good, which was expected since GPU hardware double precision support is still limited[5]. We therefore considered it was not worth the effort to pursue the GPU implementation of the partition function. We instead focused on the SSE vectorized implementation, whose results are presented in the next section.

Results

We computed the partition function on the 9781 nucleotide long hiv virus sequence. On the standard Unafold 3.8 package and with our new tiled and vectorized implementation, execution time was, respectively, 184 minutes and 20 minutes, a 9.2x speedup. This is a good result considering the SSE vectorization only provides at best a 2x speedup here. The remainder of the speedup comes from better memory locality of our approach, and from some computation we were able to factorize thanks to our tiling approach.

3.9 Applications

This section gives examples of bioinformatic studies where the computation of RNA secondary structures was successfully conducted with our GPU implementation.

3.9.1 MicroRNAs Prediction

MicroRNAs (miRNAs) are small (18-25 nt) non coding RNAs playing crucial roles in the regulation of gene expression in eukaryotes. miRNAs precursors have a distinctive hairpin secondary structure between 65 and 100 nt long. In insects, they are involved for example in embryonic development and tissue differentiation [34].

In collaboration with bioinformaticians and biologists working at INRA (Fabrice Legai, Denis Tagu, Stéphanie Jaubert-Possamai) we developed a tool to identify new miRNAs of the pea aphid *Acyrthosiphon pisum*. Contrary to common methods relying in part on comparative analysis of related genomes to identify miRNAs, the new method relied only on the analysis of the pea aphid genome and known miRNA characteristics. This work lead to a publication in *BMC genomics* [41].

The whole pea aphid genome was scanned to find miRNA-like hairpins. For that the secondary structures of 120 nucleotides long windows with an overlap of 100 nucleotides were computed. On the whole genome, this represents a total of 21 million windows.

Hairpins longer than 63 nucleotides were selected, leading to approximately 2.5 million genomic hairpins. A set of features was developed to discriminate between miRNA

[5]Tests were conducted with a Tesla C1060 where double precision throughput is only 1/8 that of single precision. At the time of development the new Fermi architecture with improved double precision support was not yet available.

and non miRNA hairpins: folding energy, total and maximum internal loop size and symmetry, arms and terminal loop sizes; GC% and complexity score.

Using 38 pea aphid miRNAs already found by homology search as a learning set, we constructed a classifier using all these features. We first used a decision tree implemented in the R tree package to build a prototype classifier and then hand tuned it to improve its selectivity. Out of the 2.5 million genomic hairpins, the classifier identified 4500 hairpins as putative miRNAs, out of which 98 were confirmed to be expressed by comparison with deep sequencing data.

Although not related to our GPU parallelization work, this bioinformatic study provided a real-world test of our GPU implementation. The 21 million 120 nt long windows were successfully folded in less than two hours with a Tesla C870 and a GTX280, whereas this would have taken 24 hours if running on a typical dual core CPU. Work is still in progress to generalize the tool so that it can predict miRNAs of other organisms.

3.10 Discussion

In this chapter, the goal was to find whether GPUs were able to provide an efficient implementation of the RNA secondary structure prediction problem. We developed a first parallelization which yielded a 17x speedup compared to the standard Unafold package [67]. This first version was however severely bandwidth limited.

We then developed a new tiling approach of the algorithm. This approach exhibits a lot of data reuse between threads. This technique therefore allowed us to move from a memory-bound algorithm to a compute-bound kernel, thus fully unleashing raw GPU power. This tiled implementation is detailed in a chapter of the NVIDIA GPU computing gems book to be published [69]. The speedup obtained compared to the sequential standard Unafold algorithm is up to 150x for large sequences, a lot more than our initial expectations.

Moreover we also successfully applied the method to the computation of suboptimal foldings. The answer is thus definitely positive, it is possible to parallelize efficiently on GPU a dynamic programming algorithm of the scope of the complex RNA folding algorithm.

However, the 150x speedup needs to be considered carefully. It does not solely comes from GPU parallelization, but also from the benefits of our new tiling approach. We noted that our algorithmic modification also enables tiling and subsequent vectorization of the standard CPU implementation. This allowed us to a do a fair apples-to-apples comparison of the CPU and GPU implementations, and avoid the hype surrounding GPGPU.

The speedup compared to our well optimized and vectorized CPU code is around 2x compared to an eight core system. This corresponds to the typical speedup observed when comparing to vectorized CPU code [40]. This may seem a low speedup but in fact already very interesting for people having to conduct RNA folding intensively.

To sum up, our contribution to the RNA folding problem are manifold: we developed a GPU parallelization for the MFE and suboptimal problems, and a new SSE vectorized CPU implementation for the MFE problem, suboptimal foldings and the partition function. The new CPU vectorized code is in itself a very good result.

An interesting point is that the method we developed for GPU was also able to provide a large speedup of the CPU code. It is in fact not surprising, since both architectures equally benefit from good data locality and from efficient vectorization.

Chapter 4

Short Sequence Alignment

Sequence alignment is a major field of bioinformatics. With the advent of next generation sequencing technologies, developing highly efficient alignment programs is more than ever a priority. In this chapter the goal is to find whether a GPU parallelization is able to tackle this problem efficiently.

Section 4.1 first explicits the problem, then section 4.2 details state of the art methods used to tackle the alignment problem: algorithms based on *seeds* or on *suffix trees*.

Our GPU study is based on the ORIS algorithm already developed by Lavenier in [39], initially targeting ungapped alignments. This algorithm exhibits good data locality and provides many indepedent tasks, making it *a priori* a good match for GPU parallelization. It is presented section 4.3. However, first profiling tests showed that in the case of gapped alignments, significant optimizations were possible.

We developed a filtering strategy that discards most alignments coming from the seed phase before they are checked by the costly dynamic programming algorithm (section 4.4). Promising results were obtained with this strategy on a first prototype running on CPU.

GPU parallelization of this first prototype is conducted in section 4.5. Although necessary, the filtering strategy also makes GPU implementation significantly more difficult. It was therefore decided to move back to a CPU program in order to be able to develop the filtering strategy to its full potential (section 4.6). This lead to the creation of the GASSST program, whose results are analysed in section 4.7.

4.1 Problem presentation

Next generation sequencing (NGS) technologies are now able to produce large quantities of genomic data. They are used for a wide range of applications, including genome resequencing or polymorphism discovery. A very large amount of short sequences are generated by these new technologies. For example, the Illumina-Solexa system can produce over 50 million 32-100 bp reads in a single run. A first step is generally to map these short reads over a reference genome, in other words, find the initial location of the read in the genome. To enable efficient, fast and accurate mapping, new alignment programs have been recently developed. Their main goals are to *globally* align short sequences to *local* regions of complete genomes in a very short time. Furthermore, to increase sensitivity, a few alignment errors are permitted.

The *seed and extend* technique is mostly used for this purpose. The underlying idea is that significant alignments include regions having exact matches between two sequences. For example, any 50 bp- read alignments with up to 3 errors contains at least 12 identical consecutive bases. Thus, using the *seed and extend* technique, only sequences sharing common k-mers are considered for a possible alignment. Detection of common k-mers is usually performed through indexes localizing all k-mers.

Recently, several index methods have been investigated and implemented in various bioinformatic search tools. The first method, used by SHRiMP [71] and MAQ [44], creates an index from the reads and scans the genome. The advantage is a rather small memory footprint. The second method makes the opposite choice: it creates an index from the genome, and then aligns each read iteratively. PASS [10], SOAPv1 [46] and BFAST [30] use this approach. The last method, used in ORIS and CloudBurst indexes both the genome and the reads [39, 73]. Although more memory is needed, the algorithm should exhibit better performance due to memory cache locality. Another short read alignment technique, used in Bowtie [38], SOAPv2 [47], and BWA [42], uses a method called backward search [22] to search an index based on the Burrows-Wheeler Transform (BWT) [9]. Basically, it allows exact matches to be found before using a backtracking procedure that allows the addition of some errors. Although this technique reports extremely fast running times and small memory footprints, some data configurations lead to poor performances[1].

Moreover, in order to speed-up computations, some methods restrict the type or the number of errors per alignment to a few mismatch and indel errors. In the building alignment process, computing the number of mismatches requires linear time, whereas indel errors require more costly algorithms such as the dynamic programming techniques used in the Smith-Waterman [74] or Needleman-Wunsch [59] algorithms. For instance, MAQ, Eland and Bowtie do not allow gaps. EMBF [79], SOAPv1 and SOAPv2 allow only one continuous gap, while PASS, SHRiMP, BFAST and SeqMap [35] allow any combination of mismatch and indel errors. In most applications, when reads are very short, dealing with a restricted number of errors is acceptable. On the other hand, when longer reads are processed, or when more distant reference genomes are compared, this restriction may greatly affect the quality of the search.

4.1.1 Problem definition

The *mapping* problem can be formally stated as follows:

Problem 1. *Let $R = \{r_1, r_2, ...\}$ and $Q = \{q_1, q_2, ...\}$ be two collections of sequences, usually referred to as reference and query set. For a given scoring function $\mathcal{F}$, find for every query sequence q_i the substring s of any sequence in R which maximizes the optimal alignment score $\mathcal{F}(s, q_i)$.*

[1]Bowtie online manual at http://bowtie-bio.sourceforge.net/manual.shtml specifies for example that reporting multiple alignments per read or allowing more "backtracks" to be performed significantly slows down the program.

Additionally, users can impose a minimal alignment score, i.e a maximal number of errors above which alignments are deemed not significant enough and not outputted. Queries, also called *reads*, can be as short as 36nt, however their length is rapidly increasing. One particular issue is when a query aligns equally well to many positions of the reference. In that case, it is difficult for the mapping algorithm to choose one position over another. The typical solution is to pick one randomly and warn the user that the read is probably not correctly mapped. Errors in the alignment originate from two sources: mutations between the sequenced chromosomes and the reference genome, and sequencing errors.

4.2 State of the art

Alignments algorithms have to deal with millions of short sequences, and map them on a reference sequence, which can be as long as several giga base-pairs (Gbp) in the case of the human genome. This is a daunting task if data are not handled correctly. The key is the use of data structures organizing the data. The data structures commonly used can be grouped in two major categories: algorithm based on hash tables containing the position of *k-mers* and algorithm based on suffix trees.

4.2.1 Seed Based Algorithms

Introduction

This is the most commonly used strategy. The idea behind this strategy is that a significant alignment will include some region having an exact match between the two sequences. More formally, a l-nt long sequence with k differences from a reference sequence is guaranteed to share at least a $l/(k+1)$ long substring with the reference. The alignment problem can then be reduced to the search for common substrings between the query and the reference. This idea, initially developed in BLAST [4, 5], is implemented by creating a hash-table containing the position of every k-mer of a sequence. The other sequence is then scanned: k-mers substrings—also called *seeds*—are extracted and looked up in the hash-table. A couple of matching seeds in the two sequences is potentially the start of a good alignment, sometimes called a *hit*, or *Candidate Alignment Location* (CAL) [30]. This first phase, called *seed* step, provides all potentially homologous areas in the genome with a given query sequence. Each CAL generated by the *seed* step needs to be checked to know if a good enough alignment can be generated, i.e. with an alignment score below a given threshold. This is called the *extend* step. To include gaps, the *extend* step is carried out with a dynamic programming algorithm (Needleman-Wunsh or Smith-Waterman). This approach provides high accuracy, but is prohibitively expensive due to the high number of costly Needleman-Wunsh extensions that have to be performed.

To tackle this issue, several approaches have been proposed. The first idea is to restrict the size of the candidate hit space, with an improved *seed* step.

Reduction of the hit space

Spaced seeds Ma *et al.* [54] found that non consecutive matches, called spaced seed, significantly increase sensitivity. The relative positions of the k required matches are a model, for example 110100110010101111. k is the weight of the spaced seed. Comparatively to contiguous seeds of same weight, spaced seeds both increase sensitivity and decrease the average number of generated CALs. Patternhunter was the first program to implement such spaced seeds [54], and was followed by many other programs. In particular, YASS [61] introduced a more elaborate model called *transition-constrained seeds* defined with a third symbol defining a match or a transition (mutation $A \leftrightarrow G$ or $C \leftrightarrow T$).
Eland (A.J. Cox, unpublished results) was one of the first program able to quickly map millions of short reads to the human genome. A set of 6 spaced seeds spanning the whole 32 bp alignment are used, designed to find 32-bp alignments with at most 2 mismatches. The idea is that if the alignment is divided in four 8-nt long regions, then any alignment with 2 mismatches will have at least two 8-nt long regions which are perfect matches. There are $\binom{4}{2} = 6$ possible ways of choosing these two regions, creating the 6 associated spaced seeds (i.e. 11111111000000001111111100000000, 11111111000000000000000011111111, ...). The algorithm is however restricted to ungapped alignment of size 32bp.
SOAPv1 and MAQ [46, 44] then used the same idea. SOAPv1 added the possibility of one continuous gap of size 1 to 3 bp, while MAQ extended the principle to be able to align reads up to 128 nt-long with more than two errors. The search is restricted to alignments with at most 2 errors in the first 28 bp of the read. MAQ also introduced the key concept of *mapping quality score*, which is a measure of confidence a read actually comes from the position found by the mapping algorithm. Such a mapping quality score is now commonly provided by most mapping algorithms.
BFAST [30] also uses spaced seeds. Its design idea is to decrease the number of CALs generated by the seed step with the use of large spaced seeds. In order to achieve high sensitivity, a set of different seeds is used, usually ten. The associated indexes are loaded iteratively in memory to generate a list of CALs. Gapped alignment is then performed with standard dynamic programming algorithm.

Multiple seeds A second idea is to generate CALs only when multiple seed matches occurs. A large spaced seed of the form 111111110000000011111111 might seem similar as it apparently also allows two distinct 8-nt seed matches. However it is in fact different as the spaced seed does not allow gaps to occur in the '0' regions, whereas the multiple seed technique does. Multiple seeds are used for example in the SHRiMP and in the LSPMUL algorithms [71, 17]. It is based on the observation that a l-long query string with k differences from a reference sequence contains at least $l+1-(k+1)w$ substrings of length w in common with the reference sequence. In SHRiMP the reads are indexed, then a window is moved along the genome. If a read is found to have more than a specified number of matches in that window then it is considered as a candidate alignment location.

Filter step

The last idea is to include an additional *filter* step after CALs have been generated and before they are checked by the costly dynamic programming algorithm used to obtain the actual gapped alignment. The idea is to discards CALs with some computation that is faster than the full dynamic programming. SHRiMP uses a vectorized Smith-Waterman algorithm that computes only the score of the alignment, i.e. which does not perform the traceback procedure providing the actual alignment. PASS [10] implements another solution: a pre-computed table of all possible short words, aligned against each other, is built to perform a quick analysis of the flanking regions adjacent to seed words. The number of errors in the flanking region is directly given by the pre-computed table. If it is greater than the maximum number of allowed errors, the CALs can be discarded. In practice PASS works on 7-nt long flanking regions, resulting in a $4^{14} = 256$ MB pre-computed table.

4.2.2 Algorithms based on suffix tries

The use of suffix/prefix trees solves efficiently the problem of finding exact matches. The main advantage of this data structure is that multiple identical substrings occurring in a sequence are grouped in a single path in the prefix tree. This saves a lot of computations compared to the hash table method where every copy in the reference need to be treated separately. On the other hand, prefix trees requires a lot of memory. However, the backward search technique developed by Ferragina and Manzini [22] allows to mimic the traversal of a prefix tree without actually putting the tree in memory, but just through the use of the Burrows-Wheeler Transform of the sequence [9]. This key innovation enables programs to benefit from the fast resolution of the exact matching problem with a very small memory footprint (between 2 and 3 Gb for the human genome). If an exact match is not found with this procedure, the search is generally extended to strings with some substitutions/indels. This approach is very efficient when dealing with sequences with a low error rate. This backward search method is used in BWA, SOAPv2 and Bowtie [42, 47, 38].

4.3 Starting point

In [39] Dominique Lavenier proposed the ORIS algorithm based on the *seed-extend* paradigm, aimed toward faster execution time, with a new way of manipulating seeds. The key idea comes from the observation that a lot of time is spent when accessing the index. Traditionally the reference sequence is indexed, then for each seed in the sequence query, the index is looked up to get the list of occurrences of the seed on the reference sequence. Since the index is large (several gigabytes) and accesses are random, this results in many time consuming cache-misses. The ORIS algorithm was designed to address this issue.

The principle is to order seeds such that memory accesses exhibit better locality. The two sequence banks are indexed for contiguous seeds, then seeds are enumerated in a strict order. For each seed s, two sets Q_s and R_s of occurrences are obtained, one from the reference sequence (R_s) and one from the query (Q_s). Each pair in the cartesian product $R_s \times Q_s$—representing a candidate alignment position—is then extended to see if it generates a correct alignment. By doing so every access to elements of R_s and Q_s happen in a short time: the algorithm exhibits good *temporal locality.* In the case of gapped alignment, the extension is conducted with a dynamic programming algorithm to obtain the exact alignment, i.e. the Needleman-Wunsch algorithm for a global alignment of the query sequence.

However, this special seed ordering also raises new problems. Since an alignment can be generated by different seeds, a method to ensure unique alignments are generated had to be developed. The binary encoding of the seeds provides a straight-forward strict ordering of seeds. Then, when an alignment is generated with a seed s_1 every potential seed s_2 able to produce the same alignment are compared to s_1. If $s_2 < s_1$ then the alignment is discarded. Each alignment is uniquely generated by its lowest seed. The drawback is that, in the case of gapped alignment, this check can only be conducted after dynamic programming. This results in a waste of computation.

Lavenier reported a speedup of 5 to 28 compared to BLASTN. The initial idea, developed in the next section, was to capitalize from these good results to create a new GPU-accelerated program, suitable for sequences generated by next generation sequencing technologies, i.e efficient for a massive number of short sequences. Several levels of parallelization can be exploited: different seeds can be computed in parallel, and each seed generates a list of candidate alignment locations that can also be computed independently. Moreover the good data reuse of the algorithm makes it *a priori* a good candidate for GPU implementation.

4.4 Preliminary work

Since the ungapped mapping problem was already tackled efficiently with BWT approaches, whose GPU implementation seems problematic, the idea was to target gapped alignments. Common techniques often restrict indels in the alignment to improve speed. The goal is to find whether GPUs are able to provide an efficient algorithm for short read gapped alignments.
As with every GPU implementation we first need to: (i) profile the CPU sequential code and identify the most time consuming sections; (ii) make sure those sections are really needed and cannot be further optimized, (iii) identify a parallelization scheme suitable for GPU. These points are examined in the three following paragraphs.

4.4.1 Profiling of the sequential code

The program implementing the ORIS algorithm, referred to as *Prototype_ V0*, presented in [39] and based on the *seed-extend* paradigm, contains two major steps: (i) searching for exact matching *seeds* between the reference and the query sequences, (ii) computing the full gapped alignment with the Needleman-Wunsch (NW) algorithm. Alignments with a number of errors below a user-specified threshold are then printed. It should be noted that since both sides of a Candidate Alignment Location (CAL) need to be extended, the number of NW algorithm instances is *a priori* twice the number of CALs.

However, CALs can be discarded after the extension of only one side if the one-sided alignment already contains too many errors.

The code has been profiled with a small test case of a 20 Mb reference sequence, and one million query sequences of 36 bases. The program was run to output all alignment with at most 2 errors. Results show that 96% of total time was spent in the dynamic programming part computing gapped alignment. More interesting, we also measured a number of 1.4×10^9 CALs generated by the *seed* phase, 1.8×10^9 instances of NW computed, and 3.4×10^6 alignments outputted in 53 minutes. First, it shows that only a small fraction (0.2%) of CALs generate good alignments. Most NW instances are only used to see that a given alignment is not good enough and needs to be discarded. Do we actually need to compute the full optimal gapped alignment to realize that? Probably not.

Secondly, as $1.8 = 1.0 + 2 \cdot 0.4$ we can see that for 1.0×10^9 CALs only one side of the seed was extended, after what the alignment was discarded. The fact that the extension is split in two parts saves a lot of computation. The sooner the algorithm can discard an alignment, the better. This key observation can probably lead to further optimizations, if we find ways to split the extension in smaller parts and discard a false-positive alignment sooner. Hence, optimizations are needed before considering GPU implementation. Parallelizing is good, but avoiding computation is even better.

PASS [10] followed a similar approach, as it uses pre-computed table corresponding to the first part of the dynamic programming algorithm. They discard alignments if this first small part already contains too many errors. However, PASS only checks at most a 7×7 sub-matrix. This is enough in the case of small 36 bp sequences, but ineffective when considering longer reads with higher error rates. Moreover, in PASS, the size of a 7×7 lookup table is $4^{14} = 256$ MB, as a score is memorized in a single byte. This fits into any computer's main memory, but not in the first level CPU cache. Hence, random access of the table, even if many computations are avoided, may still be costly. Our idea was to extend further this method. The next section presents our new algorithm using pre-computed tables.

4.4.2 Major optimization: Tiled-NW filter

A lookup table, called PST_l for *Precomputed Score Table*, containing all the alignment scores of all possible pairs of l-nt long words is first computed. PASS uses a PST_7 to analyze discrepancies near the 7-nt long flanking region adjacent to the seeds. The bigger the table, the better the filtering. But unfortunately, the table grows exponentially with l. To address this issue, the idea is to analyze discrepancies in a region of any length thanks to the re-use of a small PST_4 table. The goal is to provide a lower bound approximation of the real Needleman-Wunsch score along the whole alignment. If estimated lower bounds are greater than the maximum number of allowed errors, then alignments are eliminated. If not, alignments are passed to the next step.

The following values are used for the Needleman-Wunsh score computation: 0 for a match, and 1 for mismatch and indel errors. Consequently, the final score indicates the number of discrepancies in the alignment.

Definition 3. *In the following we call* semi-global *alignment of two sequences $S1$ and*

$S2$ an alignment that starts at the beginning of both sequences and includes the end of one or the other sequence.

The PST_l contains the optimal semi-global alignment score of all 4^{2l}possible pairs of l-nt long words. For two sequences $S1$ and $S2$ of size l, the strings participating in the alignment are therefore $S1(1,i)$ and $S2(1,j)$, with $i = l$ or $j = l$. When seeing an alignment as a path in the $l \times l$ dynamic programming matrix, the score is the minimum number of errors of any path starting at the top left corner and ending somewhere in the right column or bottom line—only the starting point of the path is constrained.

In the following, only the right side of the seed is considered, the processing of the other side being symmetrical.

Definition 4. *We call $TNW_l(n)$ the score of a region of n nucleotides which are adjacent to the right of the seed and which are computed with a PST_l. TNW stands for* Tiled Needleman Wunsch*. $TNW_l(full)$ operates on the maximum length available, i.e. from the right of the seed to the end of the query sequence.*

Definition 5. *If $S1$ and $S2$ are the two sequences directly adjacent to the right of the seed, we call $PST_l(i,j)$ the pre-computed NW score for the two l-nt long words $S1(i, i+l-1)$ and $S2(j, j+l-1)$.*

If G_m is the maximum number of allowed gaps, $TNW_l(n)$ is computed with the following recursion:

If $n \leq l$ then

$$TNW_l(n) = PST_l(1,1) \tag{4.1}$$

Else

$$TNW_l(n) = \min(A, B, C) \tag{4.2}$$

With

$$\begin{aligned} A &= TNW_l(n-l) + PST_l(n-l+1, n-l+1) \\ B &= \min_{1 \leq s \leq G_m} (\max(s, TNW_l(n-l)) + PST_l(n-l+1, n-l+1-s)) \\ C &= \min_{1 \leq s \leq G_m} (\max(s, TNW_l(n-l)) + PST_l(n-l+1-s, n-l+1)) \end{aligned}$$

Figure 4.1.B shows a graphical representation of this recursion.

The beginning of the recursion is straight forward: the alignment path must start right after the seed. Hence, $PST_l(1,1)$ is the first level. At the second level, the TWN computation minimizes over all the different points an alignment can go through when exiting the first sub-matrix.

Theorem 1. *The score $TNW_l(n)$ is a lower bound of the score of the optimal semi-global alignment of $S1(1,n)$ and $S2(2,n)$ computed by the Needleman-Wunsch algorithm.*

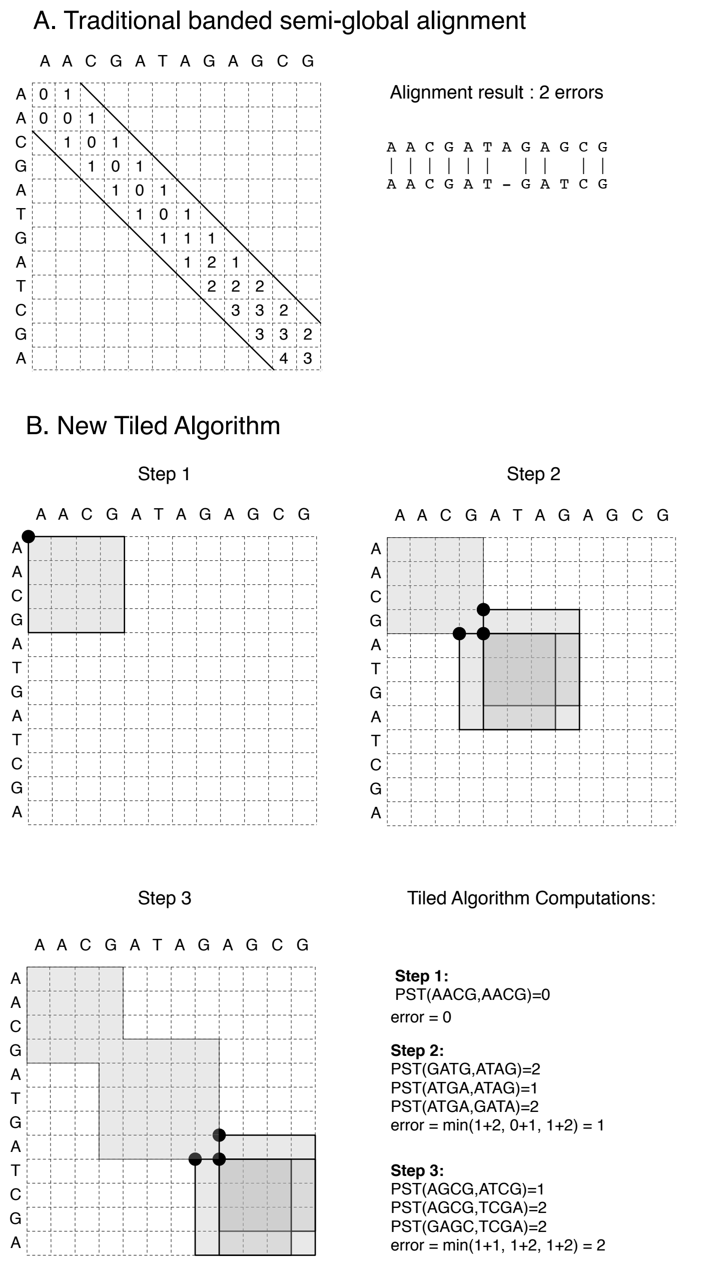


Figure 4.1. Computation of a semi-global alignment (only the query sequence needs to be globally aligned), in the case of a maximum of 1 error. **A. Dynamic Programming.** With 12-nt long sequences and a traditional dynamic programming algorithm, cell calculations can be limited in a band, but there are still 34 cells to compute. **B. Tiled Algorithm.** With a pre-computed table score of size 4×4, the tiled algorithm is performed in only three steps. The first step requires 1 table access, while the second and third steps both require 3 table accesses. Here, the score given by the tiled algorithm is the same as the full dynamic programming algorithm - there are 2 errors in the alignment. In the general case, the tiled algorithm only gives a lower bound of the number of errors present in the alignment.

Theorem proof is in appendix A.1.

Since the score is the number of errors in the optimal alignment (scoring scheme is 0 for a match, 1 for mismatch or indel), theorem 1 implies that $TNW_l(n)$ is a lower bound of the number of errors in the alignment.

$TNW_l(n)$ can therefore be used as a filter which will not decrease overall sensitivity: if $TNW_l(n)$ is greater than the user-specified maximum number of errors, we can safely discard a CAL before computing its actual NW score, knowing for sure it would have been eliminated anyway. The filter never eliminates CALs which could have produced good alignments.

In other words, the theorem implies $TNW_l(n)$ filter has 100 % sensitivity. It is an essential point which greatly differentiates this approach to commonly used heuristics where a speed/sensitivity tradeoff is involved. Here the approach increases execution speed at no cost to sensitivity.

On the other hand, the filter does not have 100 % specificity: as it computes a lower bound, some CALs having more than the maximum number of errors will pass the filter. Such CALs will be discarded by the next step of the algorithm: the dynamic programming algorithm computing the exact number of errors in the alignment.

Complexity of the filter With a PST_4 lookup table storing each score in a single byte, memory storage is equal to 64 KB. This small size allows the PST_4 to fit into a CPU cache of today's desktop computers. PST_4 hence requires only a few clock cycles to be computed.

In the general case, the number of PST_l accesses $N_{acc}[TNW_l(n)]$ needed for the computation is in $\mathcal{O}(n)$. If G_m is the maximum number of allowed gaps, the exact number of accesses N_{acc} is given by:

$$N_{acc}[TNW_l(n)] = (\left\lceil \frac{n}{l} \right\rceil - 1) \cdot (1 + 2 \cdot G_m) + 1 \tag{4.3}$$

The filter works with iterative l-sized levels. The first one takes 1 access, then each following level requires $(1 + 2 \cdot G_m)$ PST_l accesses. Each new level filters more and more false-positive alignments. Large PST_l tables lead to fewer levels and higher specificity, but they imply longer execution times since the number of memory cache misses increases rapidly with bigger tables.

Implementation of the filter For each seed, in the index, the position and the flanking regions are binary encoded. Two 32-bit integers are required for the position: one indicating the sequence number and the other its index. Up to 16 nucleotides are stored on the left and on the right of the seed in order to speed up the filtering steps, hence requiring 2 additional 32-bit integers. The total size of the index is therefore four 32 bit integers per seed, i.e a $16 \cdot N$ bytes index with N the number of nucleotides in the bank.

The PST_4 is constructed to take as input 2 binary encoded strings of 4 nucleotides. Before computing the full NW alignment score, $TNW_4(16)$ is computed on each side of the seed. The filter is therefore composed of 4 levels. After each level, the partial scores $TNW_4(4), TNW_4(8), TNW_4(12), TNW_4(16)$ are compared with the maximum number of errors allowed to see if CALs can already be eliminated. The filter is therefore a series of 4 filters. This exactly performs what we intended: the extension process is split in small fast computed parts, and alignments are discarded as soon as possible.

4.4.3 Analysis

We implemented the $TNW_4(16)$ on CPU, in a program we will refer to as *Prototype_ V1*. The only difference with *Prototype_ V0* is the new filter.

We conducted a profiling of *Prototype_ V1* on the same small test case of section 4.4.1. The 3.4×10^6 alignments were now outputted in about 5 minutes, a 10x speedup compared to the first non-filtered version.

Out of the 1.4×10^9 CALs generated by the seed step, 13×10^6 (0.9%) passed the $TNW_4(16)$ filter. Profiling shows that 70% of total time is spent in the filtering process, only 7% in the NW extension and most of the remainder in the indexing step. These are indicative figures, indexing part would decrease when dealing with large data banks (indexing complexity is $\mathcal{O}(N1 + N2)$ while alignment is $\mathcal{O}(N1 \cdot N2)$ with $N1$ and $N2$ the size of the two banks), and filtering percentage depends on the maximum number of errors allowed.

These results nevertheless show that this filtering step is an essential optimization, and that GPU parallelization should focus on its parallelization. A GPU implementation without this filtering step would be much easier since more *regular*. However, it would be inefficient considering the 10x speedup filtering already provides by TNW optimization. The parallelization scheme seems straightforward: thousands of independent seeds provide more than enough work for the GPU.

4.5 GPU approach

In this section, we explore the GPU implementation of the filter introduced in section 4.4.2. On one hand, there is plenty of parallelism to exploit with good data reuse. On the other hand, tasks than need to be executed on GPU cannot be mapped perfectly on a SIMD execution model—different instances of the $TNW_4(16)$ filter can stop at different levels. In this case, it is difficult to predict the outcome of the GPU implementation.

4.5.1 Parallelization scheme

With w the size of the seed, there are 4^w seeds to enumerate. Each of them can be computed independently, and will give two lists Q_s and R_s of occurrences, one from the reference sequence, the other from the set of queries. Each pair in the cartesian product $R_s \times Q_s$ represents a Candidate Alignment Location (CAL) that can also be treated independently from each other.

We exploit both levels of parallelization: we issue one GPU thread *block* per seed. We choose to have bidimensional thread blocks of $BX \cdot BY$ threads, with the $TNW_4(16)$ filter of each CAL being computed by a single thread. The $BX \cdot BY$ threads of a block will iterate sequentially over the $|Q_s \times R_s|$ different CAL generated by the seed. Figure 4.2 shows a graphical representation of this parallelization scheme.

Data specific to each occurrence of a seed are the two 32-bits integers representing the two 16-nucleotides flanking regions of a seed. Computation of the $TNW_4(16)$ filter only requires these flanking regions along with the PST table itself. Therefore the GPU, computing only this filter, does not need access to the whole data bank, the index containing the binary-encoded flanking regions is enough. Each occurrence of a seed is reused many times, allowing the two flanking regions to be stored in the fast shared memory.

Algorithm 7 GPU kernel code

```
1: BlockSize: Two dimensions = $BX \cdot BY$
2: Block Assignment: Computation of $TNW_4(16)$ filter for every CAL generated by
   seed $s$
3: Total number of blocks = number of seeds = $4^w$
4: parameter: $n$ (maximum errors allowed).
5: $tx \leftarrow threadIdx.x$, $ty \leftarrow threadIdx.y$
6: $T1 \leftarrow$ number of occurrences of $s$ in Bank1
7: $T2 \leftarrow$ number of occurrences of $s$ in Bank2
8: for $i = ty$, $i \leq T2$, $i = i + BY$ do
9:    if $tx == 0$ then
10:      Store occurrence $i$ of $s$ in Bank2 in shared memory
11:   end if
12:   for $j = tx$, $i \leq T1$, $j = j + BX$ do
13:      if $ty == 0$ then
14:         Store occurrence $j$ of $s$ in Bank1 in shared memory
15:      end if
16:      if $TNW_4(16)\{ CAL(i,j)\} < n$ then
17:         Push CAL(i,j) in result list.
18:      end if
19:   end for
20: end for
```

NB: Push result is done with an atomic incrementation of a global counter

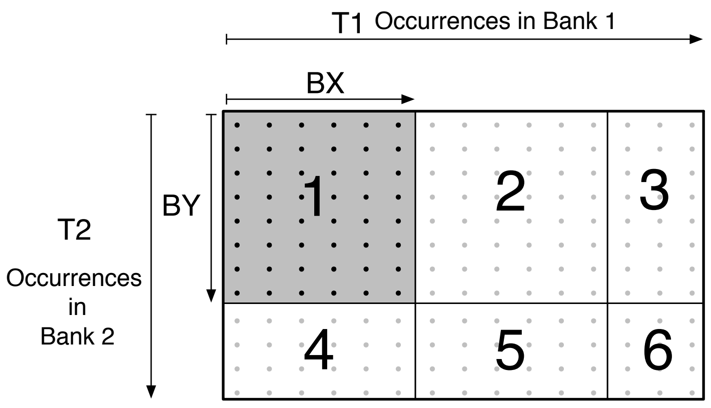


Figure 4.2. Parallelization scheme. Each GPU thread block is mapped to a single seed s. A lookup in the index for seed s gives $T1$ occurrences in the reference and $T2$ occurrences in the queries. This produces a total of $T1 \cdot T2$ candidate alignment locations represented by this $T1 \cdot T2$ matrix. For each one the $TNW_4(16)$ filter is computed by a single GPU thread. A thread block is a $BX \cdot BY$ sub-matrix. The whole space is computed by a single thread block by sequentially moving the $BX \cdot BY$ sub-matrix from left to right, then up to bottom. Data needed for each occurrence can be stored in GPU shared memory. Data for occurrences in bank 2 are reused $T1$ times while data concerning occurrences in bank 1 are reused BY times. For each cell, the outcome of the $TNW_4(16)$ filter is a boolean value.

4.5.2 Key optimizations

PST table

The first optimization to consider is memory access. Flanking regions are stored in fast shared memory, but the PST table itself, simultaneously accessed by all threads, also requires careful consideration. A PST_4 requires 64 KB if each score is stored in a byte. This is too large to fit in the 16 KB shared memory.

The first idea would be to compress this table. First, the table is symmetric, $score(S1,S2)$ equals $score(S2,S1)$. Secondly, with the scoring scheme used and 4 nt-long sequences, the score is in the range $0-4$. We could clamp this range to $0-3$(resulting in a decrease in filtering power) and store each score with 2 bits. The overall size of this compressed PST_4 would then be 8 KB and would fit in shared memory. This would however render accesses to the table very clumsy.

The second idea, which we chose to implement, is to leave the table in global memory. This could seem a bad idea, as this both causes a lot of latency and greatly reduced peak bandwidth. However, (i) latency can be completely hidden if *occupancy* is high enough, (ii) since the table will be "out of cache" anyway, we might as well have a bigger table, PST_6 or PST_7, (iii) potential problem is memory bandwidth, but we cannot be sure this will be a bottleneck before we try. Aside from the potential access latency problem, a larger PST table is better since it allows the filter on the 16-nt region to be computed with fewer steps.

Respective sizes of tables PST_4, PST_5, PST_6, PST_7 are 64 KB, 1 MB, 16 MB and 256 MB. We were surprised to observe that performance degraded significantly with the use of the PST_7 table. It seems that for the large PST_7 stored in 256 MB access latency doubled. This phenomenon, not documented by NVIDIA, is explained in [13, 77]. Nearly double latency is observed when misses in the TLB[2] occurs. Random accesses to a large 256 MB table are indeed susceptible to many TLB misses, hence the PST_6 was preferred.

Result storage

The second problematic point is how to transfer results computed by the GPU back to the CPU. The result is a boolean value for each possible CAL—a CAL either passes the $TNW(16)$ filter or it does not. CALs that passes the filter needs to be extended with the NW algorithm on CPU.

A naive solution would be to store the result in a large bit array and transfer it back to the CPU. Considering that the filtering rate on a typical test case is about 99%, this would be inefficient. It would require a lot of space, and the CPU would need to scan the whole array to find the 1% cases passing the filter, a relatively costly operation.

Instead, we implemented another solution: the GPU outputs the list of all CALs that pass the filter. Each CAL can be uniquely identified by two 32-bits integers representing the two occurrences of a seed. As this is only stored for 1% instances, this still requires less space and less bandwidth than a whole bit array containing boolean values.

This is implemented on GPU with the use of *atomic* operations allowing all threads to push elements in a shared list residing in global memory. Atomic operations have to be used with care since they serialize computation. However, here, it is not a problem as only 1% threads actually use it.
Since CUDA code cannot dynamically allocate memory, we statically allocate at the beginning a “big enough” array which will be used to store the list. This can be problematic since we do not know for sure in advance what the filtering rate will be, and therefore how much space will be required. When the array is full, the GPU stops outputting CALs, resulting in a loss of sensitivity of the algorithm.

Needleman-Wunsch computation

For each CAL passing the $TNW_4(16)$ filter, we need to compute the true Needleman-Wunsch alignment. Profiling showed that this operation was about ten times faster than the computation of every $TNW_4(16)$ filter on CPU. Therefore, if computing the filter on GPU and the NW extension on CPU, both operations should take roughly the same time. We overlap the CPU computation of NW with the GPU computation of the filter. This way, we exploit to its maximum the ressources of the hardware, and “hide” most of the computation time of the NW computation.

[2]Translation Lookaside Buffer, cache used by the memory hardware to improve virtual address translation speed. TLB implementation on NVIDIA GPUs is studied in [13, 77].

4.5.3 Results

We call this GPU implementation *Prototype_ V1_ GPU*. We compared it to *Prototype_ V1* on various test cases. In the best scenario, we obtained a 10x speedup. It is a good result when considering that the parallelization is in contradiction with the SIMD model. Since the $TNW_4(16)$ filter is in fact a series of 4 filters and execution can stop at any level, different threads will compute filter instances that do not stop at the same level. This results in many idle threads. The computation overlap between CPU and GPU dramatically helped achieve this 10x speedup.

However, it is highly dependent on the data set and configuration. One major problem is when dealing with longer reads or with higher error rates, which both leads to a lower filtering rate. In that case, (i) the part parallelized on GPU is less important, resulting in a much lower overall speedup, (ii) the list storing CALs passing the filter no longer fits in GPU memory, resulting in a major loss of CALs and decrease in sensitivity. We could implement the NW extension on GPU to improve overall speedup, but this would not solve the “storage” issue.

In its current development state, this GPU sequence alignment program only works well on a small set of configurations. Some more work could be done to improve it. However, the work already done revealed that this filtering scheme is not a good match for GPU parallelization. In particular, the storage of elements passing a filter will always be highly problematic.

Although it makes efficient parallelization very difficult on GPU, the filtering approach is interesting, since it saves a lot of computation. Therefore, instead of trying to improve its clumsy GPU implementation, we choose to develop the filtering strategy further, even if it means it will only work on CPU.

4.6 CPU approach

A CPU architecture is now targeted. Free of the constraints of GPU parallelization, we can develop the *filtering* strategy and bring it to a higher level. This lead to the creation of a new short sequence alignment CPU program, referred to as *GASSST*[3] in the following.

4.6.1 Overview

We use a traditional anchoring method followed by a Needleman-Wunsch extension step. However, fast computation is achieved thanks to a very efficient filtering step. Candidate positions are selected through a carefully designed series of filters of increasing complexity and efficiency. There are two main types of filter. One is related to the computation of an Euler distance between nucleotide frequency vectors as defined by [79]. The idea is the following: if one sequence has, for example, 3 more ’T’ nucleotides than another one, then the alignment will have at least 3 errors (mismatches or indels).

[3]Global Alignment Short Sequence Search Tool

The other is the filter introduced section 4.4.2. It is now used in different forms: it is not restricted to a 16-nt long flanking region, but computed up to the end of the query sequence. In this way, the filter is much more selective and discards a very large number of false-positive CALs, thus drastically decreasing the time spent in the final *extend* step.

GASSST algorithm has three stages: (1) searching for exact matching *seeds* between the reference genome and the query sequences, (2) quickly eliminating hits that have more than a user-specified number of errors, and (3) computing the full gapped alignment with the Needleman-Wunsch algorithm. These three steps are referred to respectively as *seed-filter-extend.* The novelty of the GASSST approach relies on a new highly-efficient *filter* method.

4.6.2 Seed

First, contrary to *Prototype_ V1*, we chose to move back to a single index scheme. The double index scheme becomes inefficient when dealing with gaps since it can detect duplicate alignments only after dynamic programming. Moreover the double index scheme does not make it possible to output only the k best alignments of a read.

On the contrary the single-index scheme can perform per-read operations that would otherwise be impossible. It makes it possible to sort alignments of a given read according to their similarity score and output only the k best, an essential feature for short read mapping. Efficiently taking care of *duplicate* alignments is also much easier with the single index.

GASSST creates an index of all possible k-mers in the reference sequence and then iterates over every query sequence. For each read, every overlapping seed it contains is looked up in the index to get a set of CALs.

The index contains a list $\mathcal{L}$ of all k-mers occurring in the reference sequence along with their list of occurrences. For each k-mer occurrence in the reference sequence, the index contains the sequence number, the position where it occurs, and the flanking regions (binary encoded). Up to 16 nucleotides are stored on the left and on the right of the seed in order to speed up the next filtering steps. The size of this index is equal to $16 \times N$ bytes, with N the size of the reference sequence.

A structure is then needed to know, for a given k-mer, if it actually occurs in the reference, and if yes, its position in the list $\mathcal{L}$. For seeds shorter or equal to 14, this information is stored in an array containing all possible 4^k k-mers. This is the fastest way, although space-inefficient. For k such that $4^k >> Reference\ Genome\ Size$ only a small subset of every possible k-mer actually occurs in the reference sequence.

When seeds longer than 14 are selected the algorithm uses a simple hash-table mechanism. A hash function associates to every k-mer a 28-bit key used to index a table with 2^{28} entries. By using a hash-function with a good scattering behavior, we expect to have less than 10 collisions per entry with most reference sequences.

If large reference sequences which exceed the memory size are considered, a simple partitioning scheme is provided; the reference sequence is split into as many parts as necessary to fit in the available RAM. Each part is then processed iteratively.

4.6.3 Tiled-NW filter

We use the same $TNW_l(n)$ as introduced in section 4.4.2. With different values of l and n we have some variations of the filter. A 16-nt long seed flanking region is stored in the index, allowing computation of the filter to be fast for $n \leq 16$. Additionally from the already used $TNW_4(16)$, we now also use $TNW_5(full)$ which operates until the end of the query sequence with a larger PST_5 table. The objective of this second filter is twofold. First, using PST_5 instead of PST_4 shifts the starting position of the different levels of the filter. Since some potential mismatches are missed in these "junctions" between levels[4], using a PST_5 allows errors that could have been missed with the PST_4 to be detected. Secondly, computing the filter on the full length of the sequence, even if it requires more computation, allows a larger number of false-positive CALs to be discarded.

4.6.4 Frequency vector filter

The frequency vector filter of GASSST is the same as the one used in EMBF [79]. The idea is quite simple: if one sequence has, for example, 3 more 'G' nucleotides than another sequence, then their alignment will have at least 3 errors. If the user-specified maximum number of errors is 2, then the alignment can be directly eliminated.

For a sequence $S = s_1 s_2 ... s_n$ of characters in the alphabet $\Sigma = \{a_1, a_2, ..., a_p\}$, the frequency vector $F = \{f_1, f_2, ..., f_m\}$ is defined as:

$$\forall i \in [1; m] \;\; f_i = \sum_{1 \leq k \leq n} \delta_{s_k, a_i} \tag{4.4}$$

With

$$\delta_{s_k, a_i} = \begin{cases} 1 & \text{if } s_k = a_i \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

The Euler distance has to be computed on similar frequency vectors, i.e., referring to sequences of equal length. The distance is computed with:

$$ECD(F, G) = \sum_{1 \leq i \leq m} \frac{|u_i - v_i|}{2} \tag{4.6}$$

In GASSST, frequency vectors of sequences of up to 16 nucleotides on both sides of the seed are computed. Actually, this value is often limited by the length of the reads, and forbids vectors of the reference sequence to be computed once and for all. Indeed, the size of subsequences for which the frequency vector is computed depends on the size available on the read, which is often less than 16 for short reads and which is only known at compute-time. The Euler distance is computed between the frequency vectors of the read and the genome. If the result is greater than the maximum number of errors then the alignment is discarded.

[4]See proof of theorem 1

The computation of the frequency vectors is vectorized and quickly performed thanks to the binary format of the seed flanking regions. The frequency vectors and the Euler distance computations are both vectorized to benefit from *vector* execution units present in modern processors. Since a 16-nt sequence is 32-bit encoded, counting the frequency of a single nucleotide can be done with bit-level logical operations that operate on traditional 32-bit words. The vectorization consists of simultaneously computing values for the 4 nucleotides of the alphabet and operates on 128-bit wide words. The frequency distance vectorized filter is referred to as *FD-vec*.

4.6.5 Filters combination

The goal of the *filter* step is to eliminate as many false-positive alignments generated by the *seed* step as possible, and in the fastest possible way. This is done by ordering the individual filters from the fastest to the most complex and powerful. Algorithm 8 shows the main computation loop of GASSST with the series of filters. $TNW_4(4)$ is first since it is the fastest. It requires only one PST_4 table access, hence is computed in only a few clock cycles. It is followed by the vectorized frequency filter that rules out remaining false-positive alignments. Then a more thorough filter is used ($TNW_4(16)$). The last filter is a $TNW_5(full)$ filter which is applied on the full length of the sequence and with a PST_5 in order to eliminate a maximum number of false-positive alignments. It is computationally intensive, but it comes at the end when most alignments have already been ruled out.

Finally, the true Needleman-Wunsch alignment is computed on alignments that go through all filters. This combination of filters ensures an efficient and fast filtering in a wide range of configurations, from short to longer reads, with low or high polymorphism.

One important point is that these filters only discard alignments which are proven to have too many errors and that would have been eliminated by the Needleman-Wunsch algorithm. They never eliminate good alignments, hence they do not decrease sensitivity.

Moreover, to reduce execution time, the search is stopped when a maximal number of occurrences of a seed in the reference sequence has been reached. This kind of limitation is present in most other aligners. The only difference here is that a threshold is also checked in some stage of the filtering process. The threshold is automatically computed according to seed length and reference sequence size. Users can control the speed/sensitivity trade-off of this heuristic through a parameter s in $0-5$ which modulates this threshold.

Algorithm 8 GASSST main computation loop

```
1: Input:
2: reference genome, short query sequences.
3: parameter: $n$ (maximum errors allowed).
4: Pre-calculation:
5: Compute the reference genome index
6: for each query $q$ do
7:    for each overlapping seed $s$ in $q$ do
8:       for each occurrence $o$ of $s$ in reference genome do
9:          if $TNW_4(4)\{q,s,o\} < n$ then
10:            if FD-vec$\{q,s,o\} < n$ then
11:               if not isDuplicate$\{q,s,o\}$ then
12:                  if $TNW_4(16)\{q,s,o\} < n$ then
13:                     if $TNW_5(full)\{q,s,o\} < n$ then
14:                        if $NW\{q,s,o\} < n$ then
15:                           Print Alignment $\{q,s,o\}$
16:                        end if
17:                     end if
18:                  end if
19:               end if
20:            end if
21:         end if
22:      end for
23:   end for
24: end for
```

A triplet query, seed, occurrence {q,s,o} defines a Candidate Alignment Location (CAL)

4.6.6 Duplicate alignments

Since every overlapping seed in a query is considered, a single alignment can be generated multiple times by different seeds of a same query sequence. For example, an alignment of a 50 bp query with two mismatches contains at least nine 14-nt seeds in common with the reference. If not detected early, this can dramatically increase computation time. In GASSST, a simple mechanism is used: a bit array stores if each position has already been tested for the query currently considered. The function *isDuplicate* simply checks this array to determine if a given position has already been tested, if yes the associated CAL is discarded. The array is then reset to zero between each query. This mechanism was not possible with the double-index model of *Prototype_ V1* where the enumeration was seed-ordered and not query-ordered.

4.6.7 Long reads

For long reads, the key is to use larger seeds. Larger seeds are much more selective and lead to a much lower number of CALs. However, large keys can also lead to a lower sensitivity in the presence of more errors. It is known that a l-long alignment with k errors (mismatches and gaps) contains at least $l/(k+1)$ consecutive bases. This is

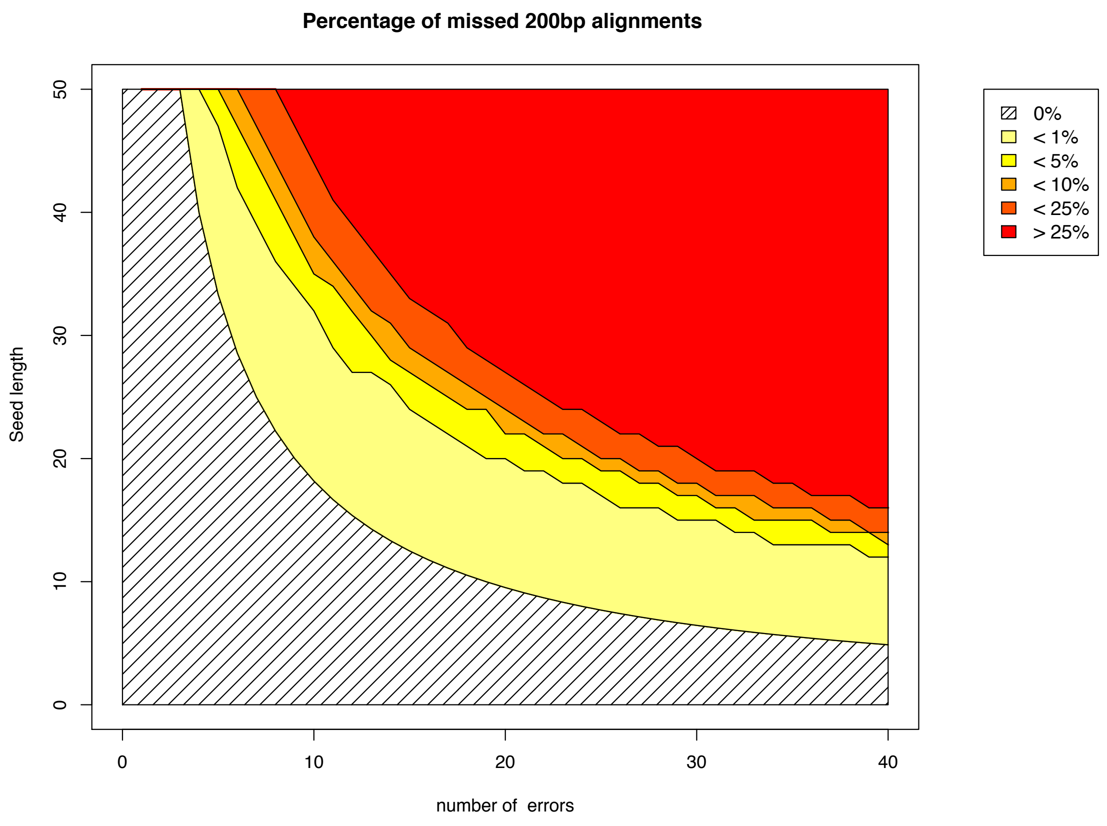


Figure 4.3. The figure shows, for 200-nucleotide long reads, the percentage of missed alignments for a given seed length (y axis) and number of errors (x axis). Data is computed with a uniform distribution of errors across the alignment. Below the $y = 200/(x+1)$ curve (dashed region), we are sure not to miss any alignment. It is interesting to see that if accepting to miss at most 1% alignments, we can select a much larger seed that will result in dramatically faster execution. For example with 20 errors a conservative choice would be a 9-nt long seed, while a 20-nt long seed will only miss 1% alignments with a uniform distribution of errors. In practice, the distribution of errors is non uniform, resulting in an even lower miss ratio. GASSST includes a model to automatically selects a good seed length, guaranteeing both fast execution and high sensitivity, according to the read length and maximum number of errors allowed in the alignment.

very restrictive, since this bound is attained when errors are equally scattered across the alignment, which is in practice highly unlikely. We included in GASSST a simple statistical model estimating a good seed size according to the read length and selected error rate. For example, for a 200-bp read and 10% errors maximum, a conservative approach would select a 9-bp long seed (200/21), whereas a statistical study shows that if the 20 errors are placed randomly in the alignment, a 20-bp long seed will only miss 1% of total alignments. This phenomenon is more pronounced with larger reads, where the probability of errors being equally scattered is less and less likely. Figure 4.3 shows the percentage map of 200-bp long alignments missed by a seed of length l (y axis) in the presence of r errors (x axis).

4.6.8 Extend

The *extend* step receives alignments that passed the *filter* step. It is computed using a traditional banded Needleman-Wunsch algorithm. Significant alignments are then printed with their full description. It should be noted that if the *filter* step provides good efficiency, no optimization of the *extend* step is required. Indeed, if most false-positive alignments have already been ruled out, the *extend* step should only take a negligible fraction of the total execution time.

4.6.9 Task parallelism

The parallelization over multiple CPU threads is straight-forward: the loop over different queries (line 6 of algorithm 8) is split among the available CPU cores. It is conducted with the Pthreads API. The only two critical sections of the code needing mutual exclusions are (i) when the main thread gives tasks to worker threads, (ii) when worker threads write results to the same file. This is not an issue with standard multi-core platforms but might become a bottleneck with massively multi-core systems.

The indexing part is not parallelized. Thus, it can significantly decrease overall speedup. It however becomes a negligible part when dealing with huge numbers of reads.

4.7 Results

GASSST results are evaluated both on simulated and real data, and compared against four state-of-the-art aligners. Performance and behavior of GASSST filters are also analyzed in detail.

4.7.1 Implementation

GASSST is implemented in C++ and runs on Linux, and OS X. It benefits from vector execution units with the use of SSE instructions, and is multi-threaded. It performs accurate and fast gapped alignments and allows the user to specify the number of hits given per read ($-h$ option). GASSST outputs either the best alignments or all alignments. GASSST is distributed under the CeCILL software license (http://www.cecill.info). Documentations and source code are freely available from the following web site: http://www.irisa.fr/symbiose/projects/gassst/.

4.7.2 Evaluated Programs

The performance of GASSST is compared with four other programs: PASS 0.74 [10] BFAST 0.6.3c [30], BWA 0.5.7 [42] and SSAHA 2.5.2 [60].

PASS indexes the genome and scans reads. It uses a pre-computed Needleman-Wunsch table to filter alignments before conducting the extension with a dynamic programming algorithm. BFAST is currently one of the most popular tools. It relies on large spaced-seeds for a fast execution, and on many different indexes for sensitivity. SSAHA also creates a hash table of the reference genome to store occurences of seeds. For each read a list of hits is created and sorted, then regions with enough hits are aligned with a banded SW algorithm.

BWA uses another approach, based on the Burrows-Wheeler Transform, and is probably one of the fastest aligners to date for alignments with a low error rate. We also tested the BWA-SW variant intended to work best for longer reads.

The computer used for the tests is an Intel Xeon E5462 with 32 GB RAM running at 2.8 GHz. Although all programs tested are able to benefit from multi-threaded computations, we choose to compare performance on a single thread, as it is enough to assess their respective strong or weak points.

We ran experiments with real data sets to give an indicative behavior. Detailed program analysis was conducted on simulated data where alignment correctness could be assessed.

4.7.3 Evaluation on real data

Performance was evaluated on three real datasets of short reads obtained from the NCBI Short Reads Trace Archive. The three sets contain, respectively, 11.9 million sequences of 36 bases, 6.8 million sequences of 50 bases and 8.5 million sequences of 76 bases of accession numbers SRR002320, SRR039633 and SRR017179. They are all aligned with the whole human genome.

BWA short read aligner was run with default options, BFAST was run with its ten recommended indexes and default options, GASSST and PASS were set to search for alignments with at most 10% errors. SSAHA was not initially designed for short reads and hence only tested with longer reads in section 4.7.4.

We measured the execution time and the percentage of mapped reads having a mappinq quality greater than or equal to 20. The results are presented in Table 4.1. For PASS which does not compute mapping quality, we measured percentage of reads having a unique best alignment.

Evaluation on real data is difficult since true alignment locations are unknown. However it is possible to compare results of different aligners to estimate accuracy, as it is done for BWA-SW evaluation in [43]. If an aligner A gives a high mapping quality to a read and another aligner B finds an alignment at another position for that same read with an alignment score better or just slightly worse, then A alignment is probably wrong. A score for each read is computed as the number of matches minus three multiplied by the number of differences (mismatches and gaps). We say that A alignment is questionable if the score derived from A minus the score derived from B is less than 20. Since this evaluation method is approximate, a full evaluation was conducted on simulated data in section 4.7.4.

On short 36bp reads GASSST performance is comparable to BWA. On longer 76 bp reads both PASS anf BFAST execution time increases a lot. On the other hand, the GASSST combination of filters still works well for the 76 bp dataset.

Table 4.1. GASSST evaluation on real data

Read length	Software	Index (s)	Align (s)	Q20%	Q20 Err rate %
36 bp	GASSST	1712	3211	34.5	0.14 %
	BFAST	520800	17520	41.8	0.12 %
	BWA	5158	3739	35.4	0.17 %
	PASS	2312	5072	41.4*	-
50 bp	GASSST	1719	4090	73.7	0.04 %
	BFAST	520800	22799	80.4	0.10 %
	BWA	5158	3043	74.5	0.17 %
	PASS	2144	5384	79.3*	-
76 bp	GASSST	1701	8483	81.3	0.04 %
	BFAST	520800	161220	85.4	0.28 %
	BWA	5158	3101	86.4	0.53 %
	PASS	1951	118541	87.7*	-

Datasets consisted, respectively, of 11.9, 6.8 and 8.5 millions of reads of size 36, 50 and 76 bp, of accession numbers SRR002320, SRR039633 and SRR017179. The three datasets were aligned with the whole human genome. The time required to compute the genome index is shown in column 3. For BFAST and BWA, the index was computed only once and stored on disk. Column 4 shows the time required in seconds to align reads, running on a single core of a 2.8 GHz Xeon E4562. Column 5 shows the percentage of reads with a mapping quality greater than or equal to 20 (Q20). Last column is the percentage of Q20 alignments that are probably wrong given the results of other aligners: if a program gives a high mapping quality to a read and another program finds a different alignment of similar alignment score for that read, then the first program is probably wrong. (*) Since PASS does not compute mapping qualities, fifth column shows for PASS the percentage of reads with a unique best alignment, and error rate is not computed.

GASSST and PASS currently cannot store the index on disk, yet index computation time is amortized when working on very large sets of reads. BFAST index time is very long because it uses ten different indexes and because its hash table relies on a sorted list of k-mers, however they are computed only once.

4.7.4 Evaluation on simulated data

We simulated twelve datasets containing one million reads each from the entire human genome, with lengths of 50, 100, 200 and 500 bp and with error rates of 2%, 5% and 10%. The error rate is the probability of each base being an error, twenty percent of errors are indel errors with indel length l drawn from a geometric distribution of density $0.7 \cdot 0.3^{l-1}$. BFAST is run with options $-K$ 8 and $-M$ 1280 with ten indexes. GASSST was tested in two different configurations, a *fast* one with option $-s$ 0 and an *accurate* one with $-s$ 3, which controls the speed/sensitivity trade-off of the algorithm by setting a maximum number of occurrences per seed. It is roughly the equivalent of options $-K$ and $-M$ of BFAST. Other GASSST options such as seed length and maximum number of allowed errors were tuned accordingly to the different datasets. BWA and

PASS options were also tuned to allow more mismatches and gaps when necessary, and BWA-SW was run with default configuration. Finding the most efficient set of options for each program and each dataset is a lengthy process, so although we did our best to tune options and provide a fair evaluation, in some cases different options may yield better performance. The results are presented in Table 4.2. Appendix A.2 gives the detail of options parameters used.

For each run we reported accuracy, sensitivity and execution time. A read is considered to be mapped if it has a unique best alignment. A mapped read is considered correct if it is within 10 bases of the true location. We filtered all alignments that had a mapping quality less than 20, except for PASS which does not compute mapping quality. Sensitivity and accuracy are then respectively defined as the percentage of total or mapped reads that are correct.

The execution time reported does not include index loading or computing phase. Although this choice penalizes BWA, which is the only one not spending noticeable time recomputing or reloading indexes, it is relevant since time spent for the index of the genome is constant and is amortized when dealing with a very large number of reads.

Tests were also conducted with SHRiMP 1.3.1 but not included here since running times were tens to hundreds of times larger than GASSST, rendering evaluation impractical. This corresponds to observations in other studies that had to extrapolate SHRiMP running time [30].

With a low 2% error rate GASSST performance is comparable to BWA. For example on short 50 bp reads, GASSST in fast mode obtained 45.8%/99.2% sensitivity/accuracy in 584 s compared to 48.2%/99.2% in 792 s for BWA. For higher error rates GASSST becomes better than BWA, for example on the 100 bp reads with 10% error rate GASSST obtained 43.5%/97.5% sensitivity/accuracy in 3262 s compared to 7.3%/97.9% in 3364 s for BWA

On datasets simulated with a high 10% error rate, GASSST consistently reports better results than other aligners. For example on the 200 bp 10% error data set, GASSST was the only one able to provide high sensitivity and accuracy within a reasonable amount of time.

On the 500 bp dataset, experiments are only conducted on GASSST, BWA-SW and SSAHA2 since other aligners are not designed for this read length. They show that GASSST still performs very well. For example, on the 2% error rate data set, GASSST obtained results comparable to BWA-SW. With 10% error rate GASSST is 8 times faster than SSAHA2 for similar results, whereas BWA-SW accuracy is dropping. BFAST is efficient on short 50 bp reads only, its execution time increases a lot for longer reads. The ability to perform well on long reads is important since read lengths are following an increasing trend.

BWA is efficient only with low error rates, BFAST is efficient only with short reads. On the contrary, these experiments show that GASSST performs well on a wide range of configurations, as expected with the design of our series of filters. This versatility is the strong point of our approach.

Table 4.2. GASSST evaluation on simulated data

Program	Mode	Metrics	50bp			100bp			200bp			500bp		
			2%	5%	10%	2%	5%	10%	2%	5%	10%	2%	5%	10%
GASSST	fast	Align sec	584	781	1720	794	981	1160	2030	2314	3051	6573	8453	11859
		Sensitivity%	45.8	42.8	36.1	54.2	53.3	44.9	58.6	56.3	53.8	61.0	59.8	58.8
		Accuracy%	99.2	98.5	93.8	90.5	89.4	86.9	91.9	91.5	89.7	93.0	92.8	91.7
GASSST	accurate	Align sec	1709	2741	4706	1290	1887	3262	3452	6173	8744	12864	18737	34222
		Sensitivity%	46.7	44.6	37.5	51.0	50.3	43.5	53.4	52.8	51.7	57.5	55.7	55.5
		Accuracy%	99.8	99.3	93.5	99.7	99.4	97.5	99.9	99.8	99.3	99.9	99.9	99.7
BFAST		Align sec	2279	2044	1756	15263	15787	11452	-	-	-	-	-	-
		Sensitivity%	46.5	43.0	32.1	52.7	51.0	48.5	-	-	-	-	-	-
		Accuracy%	98.8	96.1	85.2	99.0	98.7	95.8	-	-	-	-	-	-
BWA		Align sec	792	1392	1572	1862	4941	3364	4660	2145	185	-	-	-
		Sensitivity%	48.2	38.6	16.8	54.8	41.0	7.3	53.3	11.9	0.1	-	-	-
		Accuracy%	99.2	97.4	93.5	99.7	99.0	97.9	99.8	99.6	96.7	-	-	-
BWA-SW		Align sec	-	-	-	-	-	-	4699	3546	2365	13027	9646	7835
		Sensitivity%	-	-	-	-	-	-	54.9	50.3	25.2	57.3	56.1	45.4
		Accuracy%	-	-	-	-	-	-	99.4	96.9	85.7	99.2	96.8	85.2
SSAHA2		Align sec	-	-	-	27740	41978	45295	22285	27504	65420	179095	415252	275622
		Sensitivity%	-	-	-	45.3	43.5	38.6	53.2	51.4	48.4	59.5	58.8	55.8
		Accuracy%	-	-	-	99.8	99.1	95.3	99.8	99.1	96.0	99.8	99.2	95.3
PASS		Align sec	2012	2281	5085	14387	26033	30022	103338	139436	180943	-	-	-
		Sensitivity%	50.0	43.8	31.3	51.6	37.5	16.4	49.3	16.6	2.8	-	-	-
		Accuracy%	96.6	93.2	82.4	98.5	94.0	86.8	97.2	93.6	92.4	-	-	-

Data sets containing one million reads each are simulated from the human genome with different lengths and error rate. Twenty percent of errors are indel errors with indel length l drawn from a geometric distribution of density $0.7 \cdot 0.3^{l-1}$. The alignment time in seconds only includes the fraction of the total time proportional to the number of reads, i.e, not the time spent in computing or loading the index of the human genome, running on a single core of a 2.8 GHz Xeon E4562. Computed alignment coordinates are compared to the true simulated coordinates to find sensitivity and accuracy. Sensitivity is the percentage of reads correctly mapped, while accuracy is the percentage of mapped reads that are correctly mapped. A read is considered mapped if it has a unique best alignment. A mapped read is considered correct if it is within 10 bp of its true coordinates. We filter alignments having a mapping quality less than 20, except for PASS, which does not give mapping quality.

4.7.5 Filter behavior analysis

We measured the filtering power of the filter combination on the same three datasets of the section 4.7.3 to validate their efficiency on different configurations. The allowed similarity rate was set to 90% minimum so the number of allowed errors was 3, 4 and 7, respectively. Other program options are possible and would result in different behavior, so the results presented here cannot be generalized and are only intended to provide an indicative example.

Hits coming from the *seed* step are pipelined through the filters. At each filter step, we calculate the filtering percentage of the individual step as well as the cumulative percentage of hits filtered so far. The Needleman-Wunsch alignment is seen as the last filter and its result is also included. Table 4.3 shows the results. The first thing that can be noted is that in all cases the percentage of hits arriving at the Needleman-Wunsch step is limited to 4.0% of the number of hits generated by the *seed* phase. This means that without filters, this step would take at least 25 times longer. In the 36 and 50 bp data sets it is less than 1.0%, meaning the NW step would take at least 100 times longer without filtering.

Secondly, one interesting thing to observe is the number of false positive alignments that passed the filters and still arrive at the Needleman-Wunsch step. The NW step filters about 80% of incoming hits, meaning that previous filters have efficiently ruled out most false-positive alignments - one alignment out of five that enters the NW step is valid. As expected, on the last dataset (7 allowed errors), the $TNW_4(4)$ filter efficiency is very poor since it can only manage a maximum of 4 errors on each side of the seed. In the current implementation, this filter is deactivated when the number of errors exceeds 7. The filtering rate can also be presented as a bar plot where the percentage of total CALs discarded by each filter is drawn, shown in figure 4.6.

Figure 4.4 shows the filtering rate on the nine simulated data sets.

In the case of reads simulated with higher error rates, the program is set to allow more errors in final alignments. It is very clear that the series of filters behave differently with different levels of errors. As the error rate is increasing, the first filters $TNW_4(4)$, FD-vec and $TNW_4(16)$ gradually filters less and less CALs. They can manage, respectively, a maximum of 4,16 and 16 errors on each side of the seed. On the contrary the $TNW_5(full)$ filter still performs well for any level of errors. For low error levels very few CALs pass the first filters and reach $TNW_5(full)$, hence it discards a low percentage of total CALs. For high level of errors a lot of CALs pass the first filters and reach $TNW_5(full)$, hence it discards a higher percentage of total CALs.

It is also worth noting the amount of duplicate alignments that are discarded by the *isDuplicate* filter. A l-long query string with k differences from a reference sequence contains at least $l + 1 - (k + 1)w$ substrings of length w in common with the reference sequence [11]. This is the number of duplicate alignments that are potentially generated by our algorithm. We indeed observe in Figure 4.4 that the quantity of CALs discarded by the *isDuplicate* filter is increasing for larger reads, and decreasing for higher error rates. On the 500 bp data set with 2% errors and 20 bp long seed, the formula says each alignment is generated by at least 280 different seeds. It is therefore of the outmost importance to quickly discards CALs generating duplicate alignments. Figure 4.4 shows the *isDuplicate* filter discards nearly 80% of all CALs in this case.

These tests showed that (i) computing short or long reads, with low or high level of errors is radically different, (ii) our series of filters performs very well on all different cases, the percentage of total CALs reaching the NW step is in the worst case 10% and lower that 1% in the best case.

To complete this analysis, we measured the time spent inside each filter. The program was profiled for a typical execution with the 50bp and 500bp dataset. For each step, the computation time of a single hit in nano-seconds was measured. Figures are probably not very accurate, however knowing the order of magnitude is already very interesting. The results are shown in table 4.4. The *General Overhead* mostly represents the cost of line 8 of Algorithm 8, which iterates through hits and loads the associated information. The $TNW_4(4)$ filter takes only 5.8 ns, which represents, as expected, only a few clock cycles of the processor. We can also verify that the filters are indeed ordered from the most simple to the most complex.

For all simulated datasets, total execution time of each filter was also measured and

Table 4.3. Filtering rate on three datasets

Data Set	Filter Step	Step filter (%)	Total filter (%)
36 bp	$TNW_4(4)$	54.9	54.9
	FD-vec	61.6	82.7
	Duplicate	57.9	92.7
	$TNW_4(16)$	84.9	98.9
	$TNW_5(full)$	32.9	99.3
	NW	81.2	99.9
50 bp	$TNW_4(4)$	12.1	12.1
	FD-vec	57.3	62.4
	Duplicate	49.7	81.1
	$TNW_4(16)$	84.9	97.7
	$TNW_5(full)$	67.7	99.1
	NW	86.9	99.9
76 bp	$TNW_4(4)$	0.0	0.0
	FD-vec	4.2	4.2
	Duplicate	75.1	76.1
	$TNW_4(16)$	26.6	82.5
	$TNW_5(full)$	81.1	96.7
	NW	82.2	99.4

The first column gives the % of hits filtered by a given filter. The second column gives the total % of hits filtered by the sequence of filters. Reads were aligned with 10 % errors maximum, which means a maximum of 3, 4 and 7 errors, gaps or mismatches, respectively. Figure 4.6 is a bar plot representation of the second column.

presented in figure 4.5. With low error rates the NW step takes less than 10 % total time which validates the assessment formulated in section 4.6.8 that we do not need to be concerned about its optimization. However, with high error rates it still represents up to 40 % of total time and may thus benefit from further optimization.

The time spent in accessing the index is significantly higher with reads length ≥ 100, where the seed length selected is ≥ 15. In that case the algorithm switches to a hash table mechanism instead of direct table lookup, which explains this major difference.

Table 4.4. GASSST filters execution time

Step	50 bp dataset time for one hit (ns)	500 bp dataset time for one hit (ns)
General Overhead	13.9	860
$TNW_4(4)$	5.8	-
FD-vec	22.4	24
$TNW_4(16)$	78.5	$\approx$ 300
$TNW_5(full)$	356.7	$\approx$ 3000
NW	4486.5	$\approx$ 135000

Average time taken by each filter step for one hit of 50-bp and 500-bp query, in nano-seconds. Note that only the times of $TNW_5(full)$ and NW should actually depends on query size since the others only work on a bounded size subsequence.

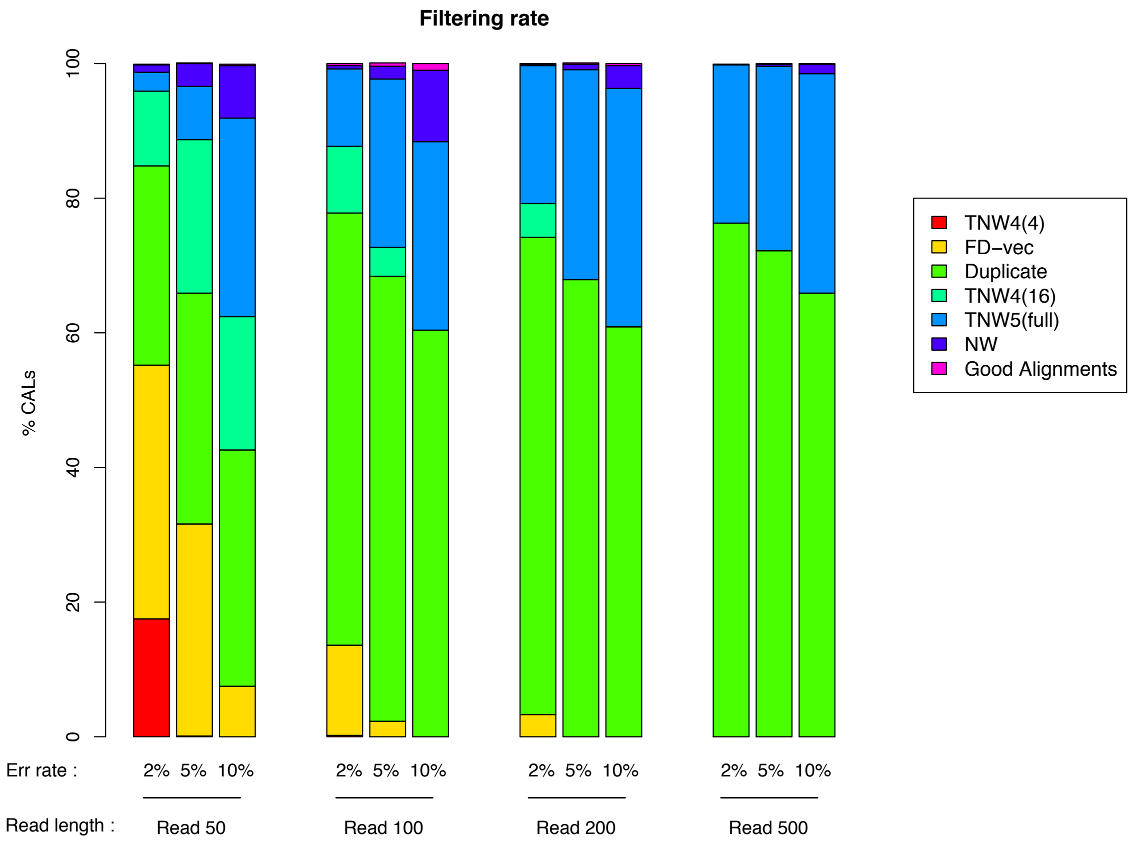


Figure 4.4. Detailed filtering rate of GASSST series of filters on simulated data sets. Data sets containing one million reads each are simulated from the human genome with different lengths and error rate. Twenty percent of errors are indel errors with indel length l drawn from a geometric distribution of density $0.7 \cdot 0.3^{l-1}$. The series of filters is composed in this order of $TNW_4(4)$, *FD-vec*, *isDuplicate*, $TNW_4(16)$, $TNW_5(full)$, NW. The percentage of total Candidate Alignment Locations (CALs) discarded by each filter is represented as a bar plot. CALs passing all filters are correct alignments and outputted. It should be noted that the cumulative filtering rate of the series of filter is very high, most of the time between 95 % and 99%. This is what makes GASSST efficiency: the costly NW algorithm is computed on a few number of CALs only.

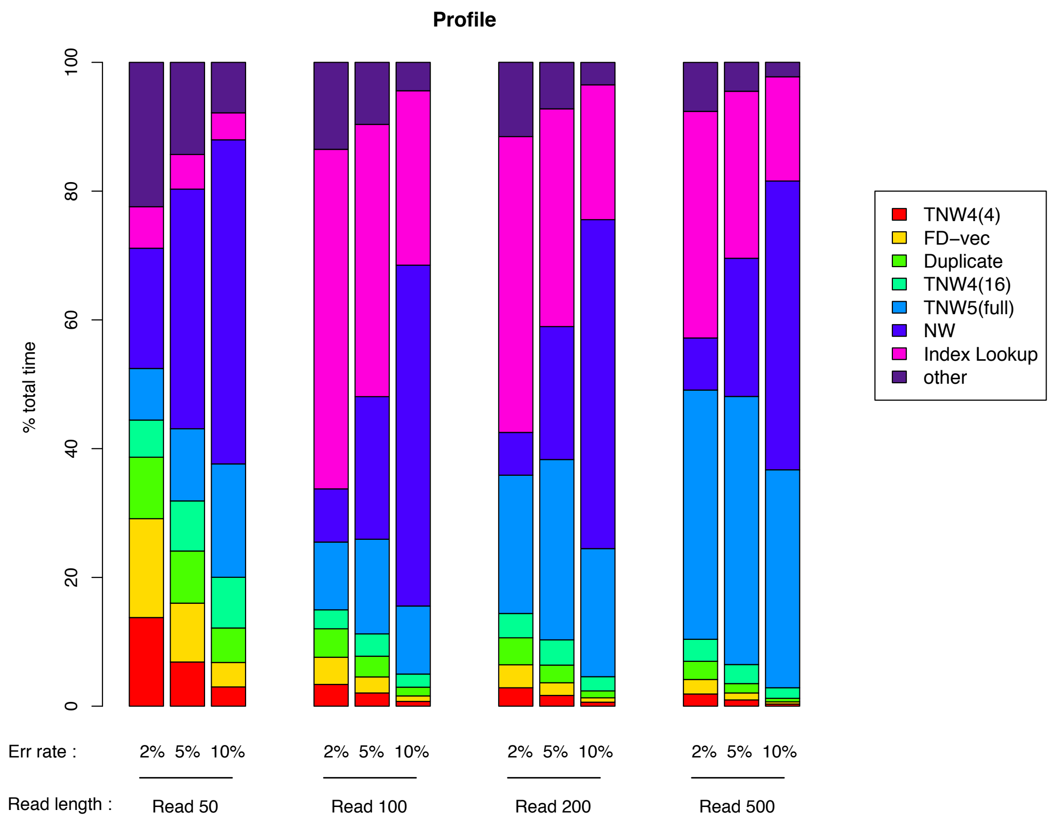


Figure 4.5. GASSST algorithm profile on the twelve simulated data sets. The figure shows the percentage time spent in each part of the algorithm.

4.7.6 Hash table statistics

It is essential to check whether the hash function (used for seeds longer than 14 bp) effectively scatters seeds well in the hash table. Since collisions are stored in a linked list in each entry, if the average number of collisions for a seed is N then it takes in the worst case N steps to find the seed in the index. If N is large overall execution time can be significantly increased.

In GASSST the hash function is a commonly used function consisting of a combination of *xor*, *mult* and *shift* on the whole input seed. The result is then truncated to 28 bits. We measured the behavior of the hash function on a 500 Mb partition of the human genome, and compared it to a simple hash function taking the 28 first bits of the input seed, such as the one used in BFAST [30]. Since GASSST hash table has 2^{28} entries, the best case result would be to have an average number of 1.86 collisions per seed.

The results are presented in Table 4.5. The average number of collisions is 2.55, a satisfying result. With the simple hash function, it is 11.2. In BFAST, this simple hash function is used with an additional feature: seeds are sorted. It means that locating a

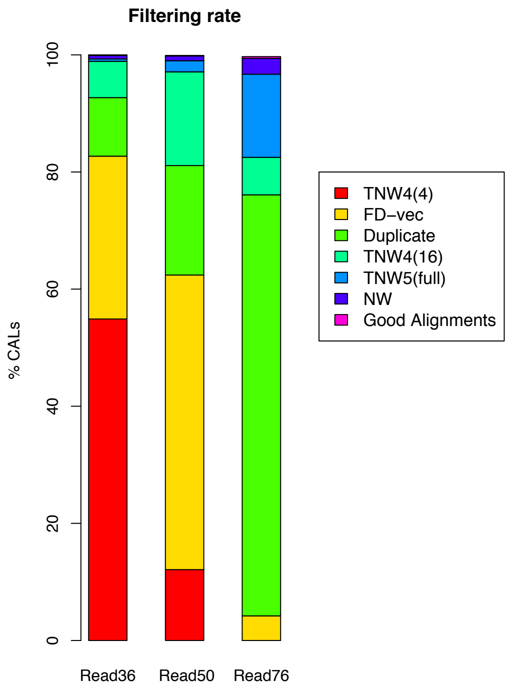


Figure 4.6. Filtering rate on real data sets, consisting, respectively, of 11.9, 6.8 and 8.5 millions of reads of size 36, 50 and 76 bp, of accession numbers SRR002320, SRR039633 and SRR017179. The three datasets were aligned with the whole human genome. The percentage of total Candidate Alignment Locations (CALs) discarded by each filter is represented as a bar plot.

Table 4.5. Hash Table statistics

hash function	Collisions	Seed 16	Seed 18	Seed 20
standard	Average	2.22	2.52	2.55
	Maximum	12	13	14
simple	Average	4.2	7.5	11.2
	Maximum	16	256	3650

Number of collisions in the index hash table with two different hash functions, on a 500MB partition of the human genome. The *simple* hash function takes the 14 first nucleotides of the seed, while the *standard* hash function is a combination of *xor*, *mult* and *shift* operations on the whole input seed, then truncated to the 28 first bits.

seed can be done by bisecting the list of collisions. In BFAST, where the whole genome is usually indexed entirely in one time, sorting seeds seems a sound choice to reduce lookup access time.

However, with GASSST the genome is usually partitioned in smaller parts, for example 500 millions nucleotides. With these relatively small partitions, sorting seeds is inadequate: since (i) sorting seeds significantly increases indexing time (see indexing time of the ten indexes of BFAST in table 4.1), (ii) using a good hash function renders the bisecting procedure useless, as $\log_2(11.2) = 3.5$ is still greater than 2.55.

4.8 Discussion

In this chapter, the goal was to find whether GPUs were able to provide new fast algorithms for short sequence alignment. We started from the ORIS algorithm developed by Lavenier [39], and tried both a GPU approach and a CPU approach. The GPU approach gave some results, but not satisfactory enough. The CPU approach yielded good results and lead to the creation of the GASSST program, whose results were published in the *Bioinformatics* journal [68].

The evolution of the place of GPU along this work is very interesting. The starting point was an algorithm with a very high level of parallelism, highly regular and with good data reuse. Millions of costly instances of Needleman-Wunsch computations were indeed conducted, with a good temporal locality coming from the ordered index seed algorithm. That made it *a priori* a perfect match for a GPU implementation.
Then, from benchmarks to optimizations, we gradually realized it was beneficial to break the regularity of the code to avoid a lot of computations. For some candidate alignment locations, it is very easy to see that they will not generate good enough alignments, while for others it is necessary to conduct a full NW extension to be able to make a decision. Computation is then no longer regular: the different CALs stop at different levels in the pipeline of filters, each CAL does not require the same amount of computation. With a complex series of filters as the one used in GASSST, it appears no longer possible to implement an efficient GPU code.

The first reason is that it is not possible to map a thread to a CAL since the computation of different CALs, stopping at different levels of the sequence of filters, is not in accordance with the SIMD GPU model. The other solution would be to compute the series of filters *level after level*. In other words, compute in parallel for every CAL the first filter, store the list of CALs passing the filter, then compute in parallel the second filter, and so on. This approach would fit the SIMD model. However, that would require storing the list of CALs passing each filter, which is highly problematic. Even if a way was found to conduct it successfully, the overhead cost associated with this operation—absent in the sequential CPU code—would most probably seriously impact the overall speedup of such a GPU implementation.
Therefore, it appears nearly impossible to obtain a good speedup with a GPU implementation of the GASSST algorithm in its current state.

The GPU approach we tried in section 4.5 was working on an algorithm already halfway between the original regular ORIS algorithm and the final GASSST algorithm with his complex series of filters, i.e it was already using a filter. On one hand, benchmarks showed that filtering was essential; on the other hand it is what rendered the GPU parallelization tricky. This illustrates well the dilemma we faced with the GPU approach: is it better to parallelize the regular, inefficient algorithm, or try to parallelize the highly efficient but irregular one ?
The answer here is that both options lead to a dead-end for GPU parallelization, as (i) parallelizing the regular algorithm is useless, since its speedup compared to the sequential efficient and irregular algorithm would be abysmal, (ii) as explained, the GPU parallelization of the irregular algorithm seems impossible.

To conclude, GPU architecture seems not to be able to provide good speedup for short sequence alignment, at least not for the *seed-and-extend* paradigm. On the other hand, more optimizations are yet possible on the sequential algorithm.

Figure 4.5 shows that the *General Overhead*, the cost of iterating through hits, can represent up to a third of the total execution time. This reveals that for the simple indexing scheme we use, our filtering technique is close to the optimal solution, in the sense that further improvements of filters would not bring much overall speedup. However, combining state-of-the-art indexing techniques with our series of filters should achieve better performance. We could, for example, replace our simple unique contiguous seed index with spaced seeds or with the multiple indexing technique used by BFAST. It would reduce the number of hits to explore, probably increasing performance even further.

Moreover, GASSST currently uses a simple gap and mismatch scoring scheme, whereas affine gap penalties are sometimes necessary. It could very easily be integrated in the *extend* step. Yet if the *filter* and *extend* step uses different scoring schemes, filters will no longer be guaranteed to discard only false-positive alignments. This could be a problem that needs to be investigated. Another solution would also be to include affine gap penalties in the *filter* step. Although apparently problematic for our tiled NW algorithm, approximate solutions might be designed and should be the focus of future work.

Chapter 5

General conclusion

5.1 Contributions

We contributed to two different problems: computation of RNA secondary structures and short sequence alignments.

For the RNA folding problem presented in chapter 3, we successfully developed a GPU implementation yielding in the best case more than a 100x speedup compared to the standard Unafold sequential code. This was made possible by a tiling technique exposing good data locality. We also developed a new tiled and vectorized CPU program running up to ten times faster than standard Unafold code. Our method was applied to the computation of suboptimal foldings with Zuker algorithm, for which both GPU and vectorized CPU programs were developed. We obtained similar speedup compared to optimal foldings computation. Finally, we also explored the algorithm computing the partition function. Both GPU and vectorized CPU codes were tried. However, only the vectorized CPU program gave good results since the GPU available at that time did not have enough double precision floating point capabilities required for the partition function computation. The tiling scheme we developed provides a good example of what needs to be done to exploit GPU to its full potential. Whereas our first untiled implementation was bandwidth limited, the tiled algorithm is limited by GPU peak computational throughput. Similar techniques might be useful for other GPGPU programs.

In chapter 4, the initial goal was to develop a GPU program performing fast gapped short sequence alignment. It was a failure, but our work however lead to the development of an algorithm computing an approximation of the NW algorithm. It allowed us to develop a new approach to quickly filter candidate alignment locations. It lead to the development of the GASSST software, a new program to align sequences on a reference genome. GASSST provides significant advantages compared to other state-of-the-art aligners: it works well on short and long sequences, with low or high error rates.

5.2 Discussion

We focused our study on two different problems: RNA folding and sequence alignments. For the first one, the GPU implementation was successful, whereas it was not for the second problem. The first issue with the sequence mapping problem for a GPU implementation is that the problem has not reached "maturity". Many new aligners are proposed every year, and still many new approaches are probably yet to be discovered. In that sense the GPU implementation of this problem may have been conducted too early. Moreover, the new approach we developed is not consistent with the SIMD model. This is also a reason why the GPU implementation failed.

Those two different problems provided an insightful view of GPU architecture. In the next sections, we discuss what we learned and conduct our evaluation of GPU usage.

5.2.1 General design rules

It is very difficult to guess if a problem is going to benefit from a GPU implementation or not. General rules can however be given. First, the problem should contain thousands of independent tasks. Secondly, their should not be "too much" divergence within a warp. Then, there should be some data locality to benefit from shared memory, and some way to coalesce memory accesses. If all these rules are verified, a GPU implementation will probably yield a good speedup. However, in most cases at least one of these rules is violated to some degree. It is very difficult to determine a precise boundary, i.e. it is difficult to define what is "too much" divergence. It is therefore difficult to predict the outcome of a GPU implementation.

The most important point to obtain an efficient GPU implementation is to find a way to expose data locality. As explained in Chapter 1, the bandwidth to computational throughput ratio implies that, in the best case, i.e when memory accesses are fully coalesced, more than 20 operations need to be performed for each memory load/store in order to reach peak throughput. Without good data locality, a program will inevitably be bandwidth limited. Exposing data locality is harder than exposing parallelism. In many cases, parallelism is natural or requires minor transformations of the code. On the contrary, locality generally requires to drastically change computation order, and is a concept more difficult to grasp than parallelism.

5.2.2 GPU evaluation

Development time

The development time required for a GPU implementation is longer than what we initially expected. The main reason is that there are numerous different possible implementation options for a GPU program. First, a parallelization scheme has to be chosen. What will be the task of a single thread? We can have many threads performing small basic tasks, or fewer threads conducting more complex computation, i.e. fine to large grain parallelization. Secondly, the GPU kernel parameters: the number of threads per block, and grid dimensionality. Then, the choice of memory space to use for the different data needed by the algorithm: should it be placed in registers, shared memory, constant memory, texture memory, global memory? Additionally, data packing and layout in memory also has a crucial impact on memory bandwidth. To make things worse, all these parameters deeply interact with each other, making the optimization of a GPU implementation a complex optimization problem with a very large search space. It is difficult to predict which choice will yield the best performance. Hence, the only way is often to test and profile many different options. Profiling and tuning a program often leads to a local optimum of this large search space, and is the reason why most of the time subsequent optimizations are still possible.

All these optimizations aspects take time to master. Moreover, hardware is constantly evolving and regularly provides additional options. A good implementation for a given GPU might require modifications to run best on newer generations.

In our opinion, GPU development takes significantly more time than a vectorization on CPU using SSE instructions, its main competitor. While SIMD on CPU requires the programmer to explicitly handle vector length and may seem to be a lower level interaction with the hardware, the "optimization search space" associated and hardware aspects to master are in our opinion substantially easier than on GPU.

Execution speed

It is possible to claim large 100x speedups with GPU. Our RNA folding implementation can be more than 100x faster than standard sequential Unafold code. However, when comparing against well-optimized CPU codes, such speedups are not possible. In a fair comparison, when using all cores of a CPU and the vector units, the speedup generally lies in the range 2x-4x, according to a recent study [40]. This is also what we experienced: our GPU code is only 2x faster than an 8 core system exploiting all cores and vector units.

This speedup is lower than what we expected initially, but already a good result for some critical problems. For example it is possible to fit two GTX 295 GPU boards—each one containing in fact 2 single GPUs—in a computer, totalizing the equivalent power of 64 cores of CPU for our RNA folding implementation.

It is interesting to compare how much of the peak throughput we were able to harness for both GPU and CPU architectures. Let us consider the $\mathcal{O}(n^3)$ tiled part of the RNA folding computation. It uses simple arithmetic operations, *min* and $+$. For those operations, GPU and CPU can theoretically deliver, respectively, a peak 320 GOPS for a GTX 280 and 12 GOPS for one core of a Xeon 3.0 GHz. Our code achieved 90 GOPS on GPU, and 5.4 GOPS on the CPU., i.e. 28% of the peak for GPU and 45% for CPU. However, as reported by Volkov and Demmel [77] and Lee *et al.* [40], GPU performance is only 66% of the peak when operands are in shared memory. This brings the true GPU peak to 211 GOPS, and our implementation to 43% of this optimum. This figure is then very close to the percentage we get on the CPU with SIMD instructions.

Cost

For standard GPUs, the initial cost is low: high end GPUs rarely exceed 500 euros. In fall 2010, the previous example of two GTX 295 costs around 800 euros total, much less that what would cost 64 cores of CPU. Tesla versions, designed for reliability, are however more expensive.

The relevant cost metric is the performance/watt ratio, as it determines the running cost. Our code running on a Tesla C1060 is approximately 12x faster than a single core of a quad core Xeon E5450 running at 3.0 Ghz, i.e we would need 3 quad cores to have same computational power. The Tesla C1060 TDP is 190W, while the quad core Xeon TDP is 80W. In this case the GPU is therefore 1.26x ($80 \times 3/190$) more power efficient. Again, this is less than our initial expectations, but already interesting in the case of large scale clusters.

When considering specific sections of the code, the situation is different. When looking at the tiled $\mathcal{O}(n^3)$ part alone: GPU performance reaches 90 GOPS while 1 Xeon core with vectorized code gets 5.4 GOPS. Here, GPU is 1.74x more power efficient than CPU. The reason is that in this part of the algorithm, we exposed good locality allowing GPU to make full use of its execution units. In other parts of the code, the GPU is bandwidth bound and less power efficient, leading to the overall figure of only 1.26x more power efficient for the whole program.

We are however aware that this crude comparison is partly flawed. It is difficult to provide a precise comparison with CPUs. CPU and GPU peak power consumptions are provided by manufacturers. However, in the case of GPU, power consumption is for the whole board, i.e. including memory and various control logic and transfer units. On the contrary, for CPU, figures concern the CPU chip only. For a fair comparison, consumption of CPU motherboard and main memory should be added to some extent, but difficult to evaluate.

Discussion

For our implementation of RNA folding computation exhibiting good data locality, the GPU is very efficient. When considering the $\mathcal{O}(n^3)$ tiled part of the RNA folding computation alone, GPU is 16x faster than one core of a CPU using vectorized instructions, a very good result. However, exposing good data locality is a difficult task, and not always possible. Moreover, with newer GPUs the peak computational throughput is increasing faster than memory bandwidth. As a result, the ratio of operations per memory transactions required to reach full throughput is increasing, data locality is more and more important. This means that it is getting more and more difficult to exploit GPUs to their maximum.

In our opinion, GPU architecture is still interesting for the speedup of bioinformatic problems, although not as much as initially expected. For dynamic programming problems, our experience with the RNA folding computation showed that it is possible to obtain good speedup even on complex cases. GPU should be able to yield good speedup in other dynamic programming algorithms, although it may require substantial transformations.

Concerning sequence alignment and more generally sequence analysis, GPU efficiency is in fact not obvious. The first reason is that most algorithms do not bluntly apply some computation to every sequence of a data set. Instead, they usually rely on a data structure to organize data and quickly access sequences of interest. Such data structures are not well suited for GPUs: most of the time they require pointer-chasing or random access of the memory, yielding low bandwidth on GPU. Secondly, as is the case with our software GASSST, some schemes are suitable for task parallelism for a multi core implementation, but not for a fine grained data-parallelism required for a GPU implementation. Task parallelism is almost always possible, whereas data parallelism is in fact not always apparent in sequence analysis. Finally, in many cases getting the exact result is not necessary, algorithms can afford to make approximations and use lots of heuristics. Heuristics generally involve breaking the regularity of the algorithm and introduce conditional branches, which decreases GPU efficiency. For example for the BLAST algorithm which uses heuristics to speed-up alignment, as far as we know no efficient GPU implementation has yet been achieved. The most recent attempt has been made by Vouzis and Sahinidis [78], who report a 3.5x speedup at most. This is of course not enough to justify the use of GPU. For the Smith Waterman algorithm which is more regular since it does not use heuristics, GPU performance is better, as detailed in Chapter 2. However even in that case a well-optimized CPU code using all cores and vector units may be more interesting to use (Table 2.1). In our opinion, rather than attempting GPU parallelization, work on sequence analysis should preferably focus on the development of new efficient algorithms. We believe there remain unexplored approaches and data structures which could yield significant speedup in this domain.

The lesson we learned is that in many cases, it would be better to start with a vectorization on CPU associated with task parallelism to use all cores. CPU vector units provide high computational power, and they are already present in all desktop computers. In our opinion they can provide enough processing power to many high-performance problems.

However, because of the convergence between GPU and CPU mentioned in chapter 1, we also believe that a GPU implementation is never done totally in vain. Any parallelization work conducted will eventually prove to be useful for the ever-increasing level of parallelism needed to exploit CPU.

Moreover, conducting a GPU implementation provides in our opinion many educational benefits. First it forces programmers to conduct deep profiling of the code, generally leading to a high level of understanding of the algorithm. This in turn may lead to subsequent optimizations and the development of new approaches. Secondly, it forces programmers to understand well the GPU hardware and identify specific hardware bottlenecks. This knowledge is highly useful to program any high performance architecture. Finally, it renders in comparison CPU parallelization seem relatively easy.

5.3 Future works

For the RNA folding problem, other tiling schemes might be developed, possibly yielding better performance. Profiling however showed that further optimizations should focus on the computation of internal loops representing generally more than 40% of total time. Yet, it is unclear how much more performance is still left to exploit and we are not sure if it is worth the effort, since our current implementation already provides significant speedup compared to standard Unafold program. Instead, additional work may be needed to implement more options or to explore other related algorithms, such as computation of secondary structures including pseudoknots, or more representative suboptimal foldings.

Concerning GASSST, it would still be possible to tweak filters: there are many filters combination and parameters yet to explore. It would also be beneficial to include a more efficient indexing technique, for example with state of the art spaced seed methods. Then, the program still lacks useful functionalities, like the support of affine gap penalties, paired-end reads and sequences in ABI color-space format.

For the short sequence mapping problem, there are still many approaches to explore. Different indexing methods need to be investigated. The use of seeds to generate candidate alignment locations is efficient but we believe better schemes are possibles, that have yet to be discovered.

Appendix A

A.1 Theorem proof

Theorem. *The score $TNW_l(n)$ is a lower bound of the score of the optimal semi-global alignment of $S1(1,n)$ and $S2(2,n)$ computed by the Needleman-Wunsch algorithm.*

Proof of theorem 1. We make a proof by induction for every $n = k \cdot l$, with $k \in \mathbb{N}$.

• For $k = 1, n = l$ we have $TNW_l(n) = PST_l(1,1)$. By construction $PST_l(1,1)$ *is* the score of the optimal semi-global alignment between $S1(1,l)$ and $S2(1,l)$.

• We suppose the theorem true for $k = p$.
We now consider the optimal semi-global alignment A_{opt} between $S1(1,(p+1)l)$ and $S2(1,(p+1)l)$.
It can be divided in two parts P1 and P2:
(P1) the semi global alignment between $S1(1,pl)$ and $S2(1,pl)$, and (P2) the rest of the alignment.
The number of errors in A_{opt} is the sum of its errors in P1 and P2. Moreover with G_m the maximum number of allowed gaps, any alignment, including A_{opt}, has $(1 + 2 \cdot G_m)$ possible ways out of P1, i.e. every possible exit point out of P1 in a band corresponding to the maximum number of gaps allowed.

Equation 4.2 iterates over all these possible ways of constructing A_{opt}. Variable s represents the deviation from the diagonal of one possible exit point. For each possibility it computes the corresponding total score of the alignment as the sum of the scores in the two parts P1 and P2. It finally takes the minimum of all these $(1 + 2 \cdot G_m)$ possibilities.

For every given s, we have:

Number of errors in P1 in equation 4.2 is computed as $\max(s, TNW_l((p+1)l - l))$. As the theorem is true for $k = p$, $TNW_l(pl)$ is a lower bound of the number of errors in P1. If the alignment A_{opt} passes through a point s cells away from the diagonal then the semi-global alignment of $S1(1,pl)$ and $S2(1,pl)$ contains at least s gaps. Therefore $\max(s, TNW_l((p+1)l - l))$ is a lower bound of the number of errors in P1.

Number of errors in P2 in equation 4.2 is computed as $PST_l(pl+1, pl+1-s)$. By definition $PST_l(pl+1, pl+1-s)$ is the optimal semi-global alignment score between $S1(pl+1,(p+1)l)$ and $S2(pl+1-s,(p+1)l-s)$, i.e. of any alignment either including the end $S1[(p+1)l]$ or the end $S2[(p+1)l-s]$.

Moreover, the semi-global alignment A_{opt} either includes the end $S1[(p+1)l]$ or the end $S2[(p+1)l]$. If it includes $S1[(p+1)l]$ it may finish before or after $S2[(p+1)l-s]$.

- If A_{opt} finishes *before* $S2[(p+1)l-s]$ then it includes $S1[(p+1)l]$ and $PST_l(pl+1, pl+1-s)$ *is* by PST construction the minimum number of errors of A_{opt} in P2.
- If A_{opt} finishes *after* $S2[(p+1)l-s]$ then $PST_l(pl+1, pl+1-s)$ is the minimum number of errors of A_{opt} in P2 *minus* the number of errors contained in the part of the A_{opt} alignment starting at $S2[(p+1)l-s]$. It is therefore a lower bound.

Therefore $\max(s, TNW_l((p+1)l-l)) + PST_l(pl+1, pl+1-s)$ is a lower bound of the number of errors of A_{opt} for s fixed. The algorithm then takes the minimum over every possible s, then $TNW_l((p+1)l)$ is a lower bound of the number of errors of the optimal semi-global alignment between $S1(1,(p+1)l)$ and $S2(1,(p+1)l)$.

$\square$

A.2 Test Parameters

Program	Option	50bp			100bp			200bp			500bp		
		2%	5%	10%	2%	5%	10%	2%	5%	10%	2%	5%	10%
GASSST fast	-w	14	14	14	18	18	18	18	18	18	20	20	20
	-p	94	90	82	96	91	85	96	92	86	97	93	87
	-g	2	3	5	3	5	7	4	7	12	7	14	23
	-s	0	0	0	0	0	0	0	0	0	0	0	0
GASSST accurate	-w	13	13	13	18	18	18	18	18	18	20	20	20
	-p	92	86	78	94	89	81	95	90	83	96	92	86
	-g	3	4	6	4	6	9	5	9	13	9	16	26
	-s	3	3	3	3	3	3	3	3	3	3	3	3
BWA	-n	def	5	5	def	7	7	def	def	def	-	-	-
	-o	def	2	2	def	4	4	def	def	def	-	-	-
PASS	max_seed_hits	20	20	20	20	20	20	20	20	20	-	-	-
	-pst	11	9	5	8	4	0	4	0	0	-	-	-
	-fid	94	90	82	96	91	85	96	92	86	-	-	-
	-g	2	3	5	3	5	7	4	7	12	-	-	-

Program parameters. “def” means with default program options.

Parameters set for each program and dataset for experiments on simulated data corresponding to Table 4. BWA-SW and SSAHA2 were set with default options. Option -454 was set for SSAHA2 for 100 and 200 bp reads. BFAST was run with its ten recommended indexes, with options -K 8 and -M 1280 for all datasets. PASS was run with seed 111111101111111 and PST table W7M1m0G0X0.pst

Bibliography

[1] Al-Khatib, R., Abdullah, R., and Rashid, N. (2010). A comparative taxonomy of parallel algorithms for RNA secondary structure prediction. *Evolutionary bioinformatics online*, **6**, 27.

[2] Allred, J., Coyne, J., Lynch, W., Natoli, V., Grecco, J., and Morrissette, J. (2009). Smith-Waterman implementation on a FSB-FPGA module using the Intel Accelerator Abstraction Layer. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–4. IEEE.

[3] Almeida, F., Andonov, R., Gonzalez, D., Moreno, L., Poirriez, V., and Rodriguez, C. (2002). Optimal tiling for the RNA base pairing problem. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, page 182. ACM.

[4] Altschul, S., Gish, W., Miller, W., Myers, E., and Lipman, D. (1990). Basic local alignment search tool. *Journal of molecular biology*, **215**(3), 403–410.

[5] Altschul, S., Madden, T., Schaffer, A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D. (1997). Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic acids research*, **25**(17), 3389.

[6] Arlazarov, V., Dinic, E., Kronrod, M., and Faradzev, I. (1970). On economical construction of the transitive closure of an oriented graph. In *Soviet Math. Dokl*, volume 11, page 85.

[7] Benedetti, G. and Morosetti, S. (1995). A genetic algorithm to search for optimal and suboptimal RNA secondary structures. *Biophysical Chemistry*, **55**(3), 253–259.

[8] Bonhoeffer, S., McCaskill, J., Stadler, P., and Schuster, P. (1993). RNA multi-structure landscapes. *European Biophysics Journal*, **22**(1), 13–24.

[9] Burrows, M. and Wheeler, D. (1994). A block-sorting lossless data compression algorithm. *Digital SRC Research Report*.

[10] Campagna, D., Albiero, A., Bilardi, A., Caniato, E., Forcato, C., Manavski, S., Vitulo, N., and Valle, G. (2009). PASS: a program to align short sequences. *Bioinformatics*, **25**(7), 967.

[11] Cao, X., Li, S., and Tung, A. (2005). Indexing DNA sequences using q-grams. In *Database Systems for Advanced Applications*, pages 4–16. Springer.

[12] Chow, E., Hunkapiller, T., Peterson, J., and Waterman, M. (1991). Biological information signal processor. In *Application Specific Array Processors, 1991. Proceedings of the International Conference on*, pages 144–160. IEEE.

[13] Collange, S., Defour, D., and Tisserand, A. (2009). Power Consumption of GPUs from a Software Perspective. *Computational Science–ICCS 2009*, pages 914–923.

[14] Cornu, A., Derrien, S., and Lavenier, D. (2010). Designing efficient FPGA-based accelerator for genomic sequence alignments with ImpulseC high level synthesis tools. Technical report, INRIA.

[15] Dirks, R. and Pierce, N. (2003). A partition function algorithm for nucleic acid secondary structure including pseudoknots. *Journal of Computational Chemistry*, **24**(13), 1664–1677.

[16] Do, C., Woods, D., and Batzoglou, S. (2006). CONTRAfold: RNA secondary structure prediction without physics-based models. *Bioinformatics*, **22**(14), e90.

[17] Dudoignon, L., Glemet, E., Heus, H., and Raffinot, M. (2002). High similarity sequence comparison in clustering large sequence databases. In *IEEE Computer Society Bioinformatics Conference, 2002. Proceedings*, pages 228–236.

[18] Fang, X.-Y., Luo, Z.-G., and Wang, Z.-H. (2008). Predicting rna secondary structure using profile stochastic context-free grammars and phylogenic analysis. *J. Comput. Sci. Technol.*, **23**(4), 582–589.

[19] Farrar, M. (2007). Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, **23**(2), 156.

[20] Farrar, M. (2008). Optimizing smith-waterman for the cell broadband engine.

[21] Fekete, M., Hofacker, I., and Stadler, P. (2000). Prediction of RNA base pairing probabilities on massively parallel computers. *Journal of Computational Biology*, **7**(1-2), 171–182.

[22] Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. In *focs*, page 390. Published by the IEEE Computer Society.

[23] Frid, Y. and Gusfield, D. (2010). A simple, practical and complete O-time Algorithm for RNA folding using the Four-Russians Speedup. *Algorithms for Molecular Biology*, **5**(1), 13.

[24] Gardner, P. and Giegerich, R. (2004). A comprehensive comparison of comparative RNA structure prediction approaches. *BMC bioinformatics*, **5**(1), 140.

[25] Gotoh, O. (1982). An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, **162**(3), 705–708.

[26] Guerdoux-Jamet, P. and Lavenier, D. (1997). SAMBA: hardware accelerator for biological sequence comparison. *Bioinformatics*, **13**(6), 609.

[27] Guibas, L., Kung, H., and Thompson, C. (1979). Direct VLSI implementation of combinatorial algorithms. In *Proceedings of the Caltech Conference on Very Large Scale Integration: held at the California Institute of Technology, 22-24 January 1979*, page 509. Caltech Computer Science Dept.

[28] Hofacker, I., Huynen, M., Stadler, P., and Stolorz, P. (1995). RNA folding on parallel computers: The minimum free energy structures of complete HIV genomes. *Working Papers*.

[29] Hofacker, I. L., Fontana, W., Stadler, P. F., Bonhoeffer, L. S., Tacker, M., and Schuster, P. (1994). Fast folding and comparison of RNA secondary structures. *Monatsh. Chem.*, **125**, 167–188.

[30] Homer, N., Merriman, B., and Nelson, S. F. (2009). Bfast: An alignment tool for large scale genome resequencing. *PLoS ONE*, **4**(11), e7767.

[31] Intel (2009). *Intel 64 and IA32 Architectures Optimization Reference Manual*.

[32] Jacob, A., Buhler, J., and Chamberlain, R. (2008). Accelerating Nussinov RNA secondary structure prediction with systolic arrays on FPGAs. In *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, pages 191–196. IEEE.

[33] Jacob, A., Buhler, J., and Chamberlain, R. (2010). Rapid RNA Folding: Analysis and Acceleration of the Zuker Recurrence . In *Proceedings of the 2010 IEEE Symposium on Field-Programmable Custom Computing Machines*.

[34] Jaubert, S., Mereau, A., Antoniewski, C., and Tagu, D. (2007). MicroRNAs in Drosophila: the magic wand to enter the Chamber of Secrets? *Biochimie*, **89**(10), 1211–1220.

[35] Jiang, H. and Wong, W. (2008). SeqMap: mapping massive amount of oligonucleotides to the genome. *Bioinformatics*, **24**(20), 2395.

[36] Kahle, J., Day, M., Hofstee, H., Johns, C., Maeurer, T., and Shippy, D. (2010). Introduction to the Cell multiprocessor. *IBM journal of Research and Development*, **49**(4.5), 589–604.

[37] Knudsen, B. and Hein, J. (1999). RNA secondary structure prediction using stochastic context-free grammars and evolutionary history. *Bioinformatics*, **15**(6), 446.

[38] Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. (2009). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, **10**(3), R25.

[39] Lavenier, D. (2008). Ordered index seed algorithm for intensive dna sequence comparison. In *7TH IEEE INTERNATIONAL WORKSHOP ON HIGH PERFORMANCE COMPUTATIONAL BIOLOGY*.

[40] Lee, V., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., *et al.* (2010). Debunking the 100X

GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 451–460. ACM.

[41] Legeai, F., Rizk, G., Walsh, T., Edwards, O., Gordon, K., Lavenier, D., Leterme, N., Mereau, A., Nicolas, J., Tagu, D., *et al.* (2010). Bioinformatic prediction, deep sequencing of microRNAs and expression analysis during phenotypic plasticity in the pea aphid, Acyrthosiphon pisum. *BMC genomics*, **11**(1), 281.

[42] Li, H. and Durbin, R. (2009). Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform. *Bioinformatics*.

[43] Li, H. and Durbin, R. (2010). Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, **26**(5), 589.

[44] Li, H., Ruan, J., and Durbin, R. (2008a). Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, **18**(11), 1851.

[45] Li, I., Shum, W., and Truong, K. (2007). 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array(FPGA). *BMC bioinformatics*, **8**(1), 185.

[46] Li, R., Li, Y., Kristiansen, K., and Wang, J. (2008b). SOAP: short oligonucleotide alignment program. *Bioinformatics*, **24**(5), 713.

[47] Li, R., Yu, C., Li, Y., Lam, T., Yiu, S., Kristiansen, K., and Wang, J. (2009). SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*.

[48] Liu, W., Schmidt, B., Voss, G., Schroder, A., and Muller-Wittig, W. (2006). Bio-sequence database scanning on a GPU. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8. IEEE.

[49] Liu, Y., Maskell, D., and Schmidt, B. (2009). CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC research notes*, **2**(1), 73.

[50] Liu, Y., Schmidt, B., and Maskell, D. (2010). CUDASW++ 2. 0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*, **3**(1), 93.

[51] Lopresti, D. (1987). P-NAC: A systolic array for comparing nucleic acid sequences. *Computer*, pages 98–99.

[52] Lorenz, W., Ponty, Y., and Clote, P. (2008). Asymptotics of RNA shapes. *Journal of Computational Biology*, **15**(1), 31–63.

[53] Lyngsø, R., Zuker, M., and Pedersen, C. (1999). Internal loops in RNA secondary structure prediction. In *Proceedings of the third annual international conference on Computational molecular biology*, pages 260–267. ACM.

[54] Ma, B., Tromp, J., and Li, M. (2002). PatternHunter: faster and more sensitive homology search. *Bioinformatics*, **18**(3), 440.

[55] Manavski, S. and Valle, G. (2008). CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC bioinformatics*, **9**(Suppl 2), S10.

[56] Markham, N. and Zuker, M. (2005). DINAMelt web server for nucleic acid melting prediction. *Nucleic Acids Research*, **33**, W577–W581.

[57] Mathuriya, A., Bader, D., Heitsch, C., and Harvey, S. (2009). GTfold: a scalable multicore code for RNA secondary structure prediction. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 981–988. ACM.

[58] McCaskill, J. (1990). The equilibrium partition function and base pair binding probabilities for RNA secondary structure. *Biopolymers*, **29**(6-7), 1105–1119.

[59] Needleman, S. and Wunsch, C. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol*, **48**(3), 443–453.

[60] Ning, Z., Cox, A., and Mullikin, J. (2001). SSAHA: A fast search method for large DNA databases. *Genome Research*, **11**(10), 1725.

[61] Noé, L. and Kucherov, G. (2005). YASS: enhancing the sensitivity of DNA similarity search. *Nucleic Acids Research*, **33**(Web Server Issue), W540.

[62] Nussinov, R., Pieczenik, G., Griggs, J., and Kleitman, D. (1978). Algorithms for loop matchings. *SIAM J. Appl. Math*, **35**(1), 68–82.

[63] NVIDIA (2010). *NVIDIA CUDA C Programming Guide 3.1*.

[64] Oliver, T., Schmidt, B., and Maskell, D. (2005). Reconfigurable architectures for bio-sequence database scanning on FPGAs. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, **52**(12), 851–855.

[65] Rajopadhye, S. V. (1989). Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, **3**, 88–105.

[66] Rivas, E. and Eddy, S. (1999). A dynamic programming algorithm for RNA structure prediction including pseudoknots1. *Journal of Molecular Biology*, **285**(5), 2053–2068.

[67] Rizk, G. and Lavenier, D. (2009). GPU accelerated RNA folding algorithm. *Computational Science–ICCS 2009*, pages 1004–1013.

[68] Rizk, G. and Lavenier, D. (2010). GASSST: global alignment short sequence search tool. *Bioinformatics*, **26**(20), 2534–2540.

[69] Rizk, G., Lavenier, D., and Rajopadhye, S. (2010). *GPU computing Gems*, chapter GPU accelerated RNA folding algorithm. Addison-Wesley Professional.

[70] Rognes, T. and Seeberg, E. (2000). Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, **16**(8), 699.

[71] Rumble, S., Lacroute, P., Dalca, A., Fiume, M., Sidow, A., and Brudno, M. (2009). SHRiMP: Accurate Mapping of Short Color-space Reads. *PLoS Computational Biology*, **5**(5).

[72] Saffarian, A., Giraud, M., and Touzet, H. (2009). Paysage d'énergie et structures localement optimales d'un arn. In *Journées Ouvertes Biologie Informatique Mathématiques (JOBIM 2009) (poster)*. poster.

[73] Schatz, M. (2009). CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, **25**(11), 1363.

[74] Smith, T. and Waterman, M. (1981). Identification of common molecular subsequences. *J. Mol. Bwl*, **147**, 195–197.

[75] Steger, G., Hofmann, H., Fortsch, J., Gross, H., Randles, J., Sanger, H., and Riesner, D. (1984). Conformational transitions in viroids and virusoids: comparison of results from energy minimization algorithm and from experimental data. *Journal of biomolecular structure & dynamics*, **2**(3), 543.

[76] Szalkowski, A., Ledergerber, C., Krahenbuhl, P., and Dessimoz, C. (2008). Swps3 - fast multi-threaded vectorized smith-waterman for ibm cell/b.e. and x86/sse2. *BMC Research Notes*, **1**(1), 107.

[77] Volkov, V. and Demmel, J. (2008). Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Press.

[78] Vouzis, P. D. and Sahinidis, N. V. (2011). Gpu-blast: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, **27**(2), 182–188.

[79] Wendi, W., Peiheng, Z., and Xinchun, L. (2009). Short read DNA fragment anchoring algorithm. *BMC Bioinformatics*, **10**.

[80] Wirawan, A., Kwoh, C., Hieu, N., and Schmidt, B. (2008). CBESW: Sequence alignment on the playstation 3. *BMC bioinformatics*, **9**(1), 377.

[81] Wozniak, A. (1997). Using video-oriented instructions to speed up sequence comparison. *Computer applications in the biosciences: CABIOS*, **13**(2), 145.

[82] Wuchty, S., Fontana, W., Hofacker, I., and Schuster, P. (1999). Complete suboptimal folding of RNA and the stability of secondary structures. *Biopolymers*, **49**(2), 145–165.

[83] Zhang, P., Tan, G., and Gao, G. (2007). Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform. In *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications: held in conjunction with SC07*, pages 39–48. ACM.

[84] Zuker, M. (1989). On finding all suboptimal foldings of an RNA molecule. *Science(Washington)*, **244**(4900), 48–48.

[85] Zuker, M. and Stiegler, P. (1981). Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Res*, **9**(1), 133–148.

[86] Zuker, M., Jaeger, J., and Turner, D. (1991). A comparison of optimal and suboptimal RNA secondary structures predicted by free energy minimization with structures determined by phylogenetic comparison. *Nucleic Acids Research*, **19**(10), 2707.

List of Figures

List of Tables

List of Algorithms