**RenPar'18 / SympA'2008 / CFSE'6**
**Fribourg, Suisse, du 11 au 13 février 2008**

# Parallélisation de la recherche de similarités entre séquences protéiques sur GPU

Van Hoa NGUYEN, Dominique LAVENIER

IRISA, Campus de Beaulieu, 35042 Rennes cedex France
vhnguyen@irisa.fr, lavenier@irisa.fr

---

**Résumé**
Ce papier présente une nouvelle approche pour accélérer la recherche de similarités entre séquences protéiques. Elle repose sur une indexation complète des données en mémoire permettant d'exhiber un parallélisme massif sur des traitements simples, indépendants et bien adaptés au calcul sur GPU (*Graphics Processing Units*). Deux programmes ont été réalisés, iBLASTP et iTBLASTN et testés sur une carte NVIDIA GeForce 8800 GTX. Ces 2 programmes effectuent respectivement des recherches de similarités entre deux banques protéiques ou entre une banque protéique et un génome complet. Les facteurs d'accélération mesurés varient de 5 à 10 par rapport aux deux programmes de référence dans le domaine : BLASTP et TBLASTN.

**Mots-clés :** Recherche de similarités, séquence protéique, indexation, parallélisation, GPU.

---

## 1. Introduction

La recherche de similarités entre séquences génomiques est une des tâches fondamentales de la bioinformatique. L'objectif est de localiser des régions semblables dans des séquences d'ADN ou des séquences protéiques. Une application typique est l'interrogation d'une banque avec un gène dont la fonction est inconnue. Les résultats retournés correspondent à des segments similaires présentant un indice de ressemblance élevé. Plus exactement, l'information utile est un *alignement*, c'est à dire deux portions de séquence où sont précisément indiqués les appariements entre nucléotides (pour l'ADN) ou les appariements entre acides aminés (pour les protéines).
Aujourd'hui, les progrès continus des biotechnologies conduisent à un accroissement exponentiel des données génomiques. La banque d'ADN GenBank [1], par exemple, contient plus de 180 milliards de nucléotides (novembre 2007) et sa taille est multipliée par un facteur compris entre 1,4 et 1,5 chaque année. La recherche d'information dans ce type de banque – ou, plus généralement, le traitement global de ces informations – prend donc de plus en plus de temps et ce, malgré l'augmentation continue des puissances des ordinateurs. En fait, le volume de données (exprimé en nombre de nucléotides) croît plus vite que la puissance des machines (exprimée en MIPS) [7].
La recherche de similarités – ou d'alignements – entre séquences génomiques est principalement une recherche par le contenu sur des données brutes ; elle ne dépend pas d'analyses réalisées sur d'éventuelles annotations (mots clés). On doit parcourir systématiquement l'ensemble des banques, de la première à la dernière séquence. Il existe plusieurs algorithmes pour extraire des alignements. Les premiers, comme celui de Smith-Waternam élaboré en 1981 [13], utilisent des techniques de programmation dynamique et possèdent une complexité quadratique. Les seconds, apparus en 1990, comme le programme BLAST, se basent sur une heuristique très efficace (recherche de points d'ancrage) permettant de cibler directement de courtes zones identiques potentiellement intéressantes [11][12]. Ces algorithmes, très rapides par rapport aux précédents, ont essentiellement été concus pour procurer une recherche efficace dans les banques génomiques. Par contre, leur usage en mode comparaison intensive n'est pas réellement adapté, ce qui entraîne des temps d'exécution non optimisés.
Néanmoins, pour accélérer ces traitements, plusieurs pistes existent : indexation de l'ensemble des données en mémoire principale [10] ; parallélisation à gros grain sur des machines de type cluster [9] ; pa-

rallélisation à grain fin sur des structures spécialisées à base de VLSI/FPGA [6] ou, plus récemment, parallélisation sur des GPUs [4] [5].
Ce papier présente une combinaison entre indexation des données génomiques et parallelisation sur GPU. Il se focalise sur la comparaison de banques de séquences protéiques. Contrairement à beaucoup d'autres approches, notamment BLAST, le programme de référence [8], notre méthode évite un parcours systématique des banques. Elle repose sur une double indexation des deux banques en mémoire centrale permettant ainsi de localiser immédiatement des zones d'intérêt entre séquences. Cette indexation permet d'exhiber des milliers de petits traitements indépendants qui peuvent alors être déportés et exécutés en parallèle sur le GPU.
La suite du papier est organisée de la façon suivante : la section 2 introduit l'algorithme de recherche de similarités par points d'ancrage et expose les principes de notre méthode. La section 3 détaille la parallélisation sur GPU. La section 4 présente les résultats et la section 5 conclue.

## 2. Algorithme de recherche de similarités

Dans cette section, nous présentons successivement l'heuristique du point d'ancrage, l'algorithme de BLASTP et notre algorithme baptisé iBLASTP.

### 2.1. L'heuristique du point d'ancrage

L'idée de l'heuristique repose sur une constatation : les alignements significatifs possèdent, en général, une zone de forte similarité qui se caractérise par une suite de W caractères identiques (un mot) entre les portions des deux séquences qui composent cet alignement. On localise donc, dans un premier temps, les endroits où ces mots apparaissent. Lorsqu'un tel mot est localisé sur deux séquences distinctes, il sert de point d'ancrage pour calculer un alignement plus important en essayant d'étendre, à droite et à gauche, le nombre d'appariements (*match*) entre symboles identiques. Si l'extension conduit à une similarité significative (mesurée en nombre de *match* / *mismatch*) on procède alors à une dernière phase consistant à étendre davantage l'alignement en considérant l'inclusion ou l'omission de symboles (*gap*).
Le schéma ci-dessous montre un exemple de construction d'alignement entre deux séquences protéiques. Le mot de 3 caractères ARV, présent dans les deux séquences, sert de point d'ancrage (étape 1). A partir de celui-ci, on étend à droite et à gauche tant que l'on obtient un ratio *match/mismatch* intéressant (étape 2). Finalement, on étend encore l'alignement en prenant en considération les erreurs de *gap* (étape 3).

```
étape #1                           étape #2                           étape #3

K V I T A R V T G S A Q W C D N    K V I T A R V T G S A Q W C D N    K V I T A R V T G S A Q W C D N
        | | |                      |   |   | | |   | |                |   |   | | |   | |   |   |   |
K L I S A R V K G S Q F C T N P    K L I S A R V K G S Q F C T N P    K L I S A R V K G S - Q F C T N
```

C'est donc un processus en 3 étapes : (1) recherche d'un point d'ancrage ; (2) extension **sans** gap ; (3) extension **avec** gap. Cette heuristique est implémentée avec moult variantes dans de nombreux programmes, dont les programmes de la famille BLAST faisant référence en la matière.

### 2.2. L'algorithme de BLASTP

L'algorithme prend en entrée une banque protéique et une ou plusieurs séquences requête (protéines également). Pour simplifier, nous ne considérerons qu'une seule séquence requête. L'algorithme fonctionne en 5 étapes :

| | Algorithme de BLASTP | |
|---|---|---|
| 1 : | indexation de la requête sur la base de mots de W caractères | – étape 1 |
| 2 : | pour tous les mots M de la banque | |
| 3 : | si M $\in$ requête | – étape 2 |
| 4 : | extension sans gap | – étape 3 |
| 5 : | si score $\geq S_1$ | |
| 6 : | extension avec gap | – étape 4 |
| 7 : | si score $\geq S_2$ | |
| 8 : | afficher l'alignement | – étape 5 |

**Etape 1 : indexation**. Chaque requête est découpée en mots chevauchants de taille W. Ces mots sont stockés dans une table de hachage.
**Etape 2 : recherche de point d'ancrage**. La banque est lue séquentiellement, du début jusqu'à la fin, en considérant chaque mot chevauchant de W caractères. Pour chaque mot, on recherche s'il est présent dans la table de hachage. Dès qu'un mot existe, alors on est en présence d'un point d'ancrage.
**Etape 3 : extension sans gap**. A partir du point d'ancrage on essaie d'étendre à droite et à gauche pour produire une extension sans gap. Plus précisément, BLASTP effectue une extension sans gap uniquement s'il y a deux points d'ancrage proches l'un de l'autre. Par défaut, il faut que la distance entre ces deux points d'ancrage soit inférieure à 40 acides aminés. Un alignement sans gap incluant ces 2 points d'ancrage est alors calculé. Si le score affecté à cet alignement dépasse une valeur seuil $S_1$, alors on passe à l'étape 4.
**Etape 4 : extension avec gap**. Dans cette étape, on cherche encore à étendre l'alignement en incluant des erreurs de gap. Cette opération est réalisée par des techniques de programmation dynamique et s'appuie sur les résultats de l'étape précédente : on part des extrémités de l'alignement sans gap et on explore, à partir de ces endroits, la possibilité d'inclure ou de supprimer certains caractères. Si le score de ce nouvel alignement dépasse un seuil $S_2$, alors l'étape suivante est effectuée.
**Etape 5 : visualisation des résultats**. Au cours de cette dernière phase, BLASTP prépare la visualisation des alignements en effectuant une procédure de *traceback* sur la structure de données pour déterminer exactement les positions des appariements entre symboles.
L'avantage de BLASTP est sa rapidité par rapport aux techniques de programmation dynamique. Cette rapidité est d'autant plus marquée que la taille W du point d'ancrage est importante. En effet, plus W est grand, plus la possibilité de trouver un mot de cette de taille est faible et moins les étapes 3, 4 et 5 seront exécutées. Pour la comparaison de protéines la taille est usuellement de 3 caractères.
En revanche, lorsque BLASTP doit traiter un groupe important de requêtes, il doit parcourir systématiquement la banque pour chaque séquence requête. Cette contrainte est partiellement levée par une option qui permet l'indexation d'un groupe de séquences jusqu'à hauteur d'une taille cumulée de 20 000 acides aminés. Cette taille reste cependant bien trop faible dans le cadre de la comparaison intensive de séquences protéiques.

### 2.3. L'algorithme de iBLASTP

L'algorithme de iBLASTP indexe à la fois la banque et les requêtes. En réalité, la distinction entre banque et requête disparait puisque le but est de comparer deux banques protéiques, ou une banque protéique contre des génomes complets (ce qui revient au même).
L'avantage d'une double indexation est que la notion de parcours de la banque n'existe plus. A travers une structure de données adéquate, on peut alors directement pointer sur l'ensemble des mots identiques de part et d'autre des deux banques. Si un mot M apparait $n1$ fois dans la banque 1 et $n2$ fois dans la banque 2, il y a alors $n1 \times n2$ points d'ancrage et donc $n1 \times n2$ calcul d'alignements sans gap à calculer. L'algorithme de iBLASTP se décompose également en 5 étapes comme illustré ci-dessous :

| | Algorithme de iBLASTP | |
|---|---|---|
| 1 : | indexation des 2 banques sur la base de mots de W caractères | – étape 1 |
| 2 : | pour chaque mot M différent | |
| 3 : | construire une liste l1 de $n1$ sous-séquences banque 1 | – étape 2 |
| 4 : | construire une liste l2 de $n2$ sous-séquences banque 2 | |
| 5 : | pour chaque sous-séquence de la liste l1 | |
| 6 : | pour chaque sous-séquence de la liste l2 | |
| 7 : | extension sans gap | – étape 3 |
| 8 : | si score $\geq S_1$ | |
| 9 : | début d'extension avec gap | – étape 4-1 |
| 10 : | si score $\geq S_2$ | |
| 11 : | extension complète avec gap | – étape 4-2 |
| 12 : | si score $\geq S_3$ | |
| 13 : | afficher l'alignement | – étape 5 |

**Etape 1 : indexation**. Le principe de l'heuristique avec point d'ancrage est conservé. La différence est que l'indexation est réalisée sur les deux banques. L'indexation consiste à stocker les positions des mots

identiques dans une liste dont le début est indexé dans une table à $20^W$ entrées (20 est le nombre d'acides aminés).
**Etape 2 : recherche de points d'ancrage**. La recherche est immédiate car il s'agit simplement de consulter les listes établies précédemment. Pour les $20^W$ mots possibles, nous construisons alors deux listes de sous-séquences de taille N. Une sous-séquence correspond à une instance du mot entouré de son voisinage immédiat.
**Etape 3 : extension sans gap**. Le calcul d'extension est effectué entre toutes les sous-séquences, soit $n1 \times n2$ fois. Contrairement à BLASTP, l'extension sans gap est réalisée sur des sous séquences de taille bornée. Une autre différence majeure est qu'une extension sans gap démarre dès qu'un seul point d'ancrage est identifié (et non pas deux points d'ancrage à proximité l'un de l'autre). Les points d'ancrage sont cependant différents et reposent sur le concept de graines multiples [2].
**Etape 4 : extension avec gap**. Cette étape est divisée en deux sous étapes, la première intervenant comme un filtre pour la seconde. Dans un premier temps (étape 4-1), l'idée est de limiter l'espace de recherche de la programmation dynamique. Cette procédure est donc contrainte à la fois sur le nombre d'erreurs de gap autorisé et sur la taille de l'alignement. Si le score de l'alignement ainsi calculé dépasse un seuil $S_2$, alors la procédure de programmation dynamique standard est lancée (étape 4-2). La première étape, beaucoup moins coûteuse en terme de calculs que la seconde, filtre environ 90% des alignements.
**Etape 5 : visualisation des résultats**. Elle est similaire à celle de BLASTP et n'appelle pas de commentaires particuliers.
Cette structuration du code a été imaginée en ayant pour objectif une implémentation sur GPU. Ainsi, les étapes 3 et 4-1 qui œuvrent sur des sous-séquences de taille fixe représentent la majorité du temps de calcul comme le montre le tableau 1.

| étapes | 1 | 2 | 3 | 4-1 | 4-2 | 5 |
|---|---|---|---|---|---|---|
| iBLASTP | 0,4 % | 0,6 % | 60 % | 25 % | 11 % | 3% |
| iTBLASTN | 0,4 % | 0,6 % | 72 % | 23 % | 2,7 % | 1,3% |

TAB. 1 – *Pourcentage moyen du temps CPU à chaque étape*

Deux implémentations séquentielles ont été réalisées : iBLASTP et iTBLASTN. La première considère deux banques de protéines. La seconde considère une banque de protéines et un génome. Dans ce dernier cas, le génome est traduit dans les 6 phases de lecture (i.e. les 6 manières de faire correspondre un texte d'ADN vers des protéines). Ces deux versions qui s'exécutent sur des processeurs standards nous permettent de mesurer plus précisément l'apport des GPUs.

## 3. Parallélisation sur GPU

L'implémentation a été mise en oeuvre sur une carte graphique NVIDIA GeForce 8800 GTX. Cette carte intègre un GPU composé de 8 multiprocesseurs SIMD, chaque multiprocesseur contenant 16 processeurs. Le langage de programmation utilisé est CUDA [3], un language de programmation dont la syntaxe est proche du langage C. Le modèle de programmation considère une grille composée d'un ensemble de blocs qui exécutent simultanément un maximum de 256 *threads*. Ces threads déroulent le même programme (*kernel*).
Les sections suivantes présentent successivement la mise en œuvre globale de l'algorithme iBLAST, la parallélisation du calcul d'extension sans gap (étape 3) et la parallélisation du calcul du début d'extension avec gap (étape 4-1).

### 3.1. Mise en œuvre de iBLASTP sur GPU

La parallélisation sur GPU n'est efficace que si le nombre de tâches indépendantes à exécuter est élevé. L'implémentation sur GPU doit donc veiller à ne solliciter la carte que dans un tel cas.
L'étape 3 ne pose pas de problème : pour des banques conséquentes, le nombre de calculs d'extension sans gap ($n1 \times n2$), pour un mot donné, est toujours important. En revanche, le nombre d'alignements produits par cette étape peut être très faible. Les envoyer directement à la carte graphique pour réaliser

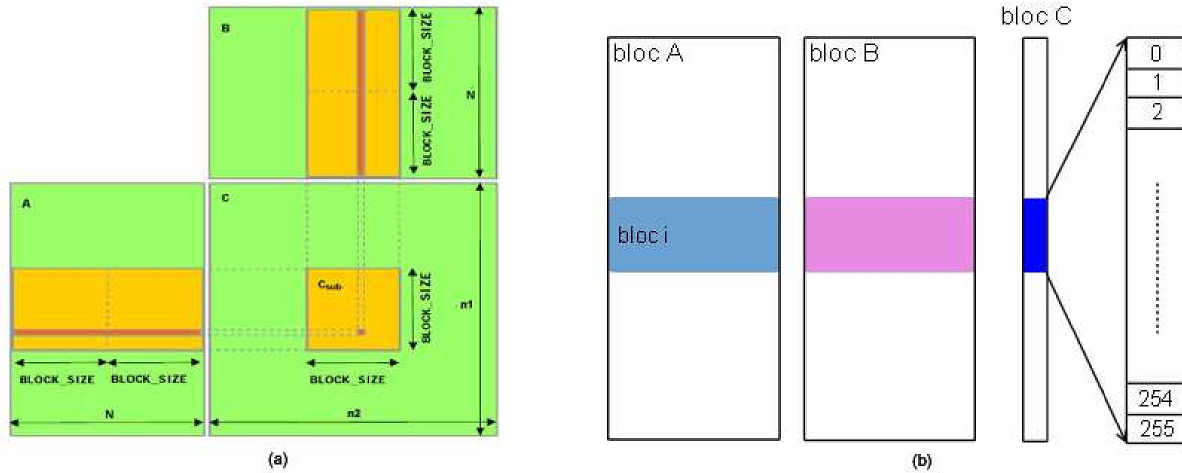


FIG. 1 – *Parallélisation de l'extension sans gap (a) et de l'extension avec gap sur le GPU (b)*

un début d'extension avec gap (étape 4-1) peut alors être extrêmement pénalisant car le GPU ne sera utilisé qu'à une fraction de ses capacités.
La solution est de stocker les résultats produits par l'étape 3 et de réaliser l'étape 4-1 que lorsqu'un nombre suffisant d'extensions sans gap a été trouvé. L'algorithme suivant (iBLASTP-GPU) schématise les modifications apportées à l'algorithme précédent (iBLASTP).

| | Algorithme de iBLASTP-GPU | |
|---|---|---|
| 1 : | indexation des 2 banques sur la base de mots de W caractères | – étape 1 |
| 2 : | pour chaque mot M différent | |
| 3 : | construire une liste de $n1$ sous-séquences banque 1 | – étape 2 |
| 4 : | construire une liste de $n2$ sous-séquences banque 2 | |
| 5 : | effectuer $n1 \times n2$ extensions sans gap sur GPU | – étape 3 |
| 6 : | stocker les extensions avec un score $\geq S_1$ dans R | |
| 7 : | si la liste R contient au moins K éléments | |
| 8 : | effectuer K début d'extensions avec gap sur GPU | – étape 4-1 |
| 9 : | si score $\geq S_2$ | |
| 10 : | extension complète avec gap | – étape 4-2 |
| 11 : | si score $\geq S_3$ | |
| 12 : | afficher l'alignement | – étape 5 |
| 13 : | supprimer K éléments de R | |

La liste R mémorise les résultats de l'étape 3. Les étapes suivantes ne sont déclenchées que si la liste R possède un nombre suffisant de résultats pour exploiter efficacement le GPU.

### 3.2. Parallélisation de l'extension sans gap

Pour un mot M, il y a $n1$ occurrences dans la banque 1 et $n2$ occurrences dans la banque 2. Autrement dit, il y a $n1 \times n2$ extensions sans gap à réaliser. Chacune des extensions étant indépendantes les unes des autres, nous pouvons paralléliser ces calculs sur le GPU. Il faut cependant veiller à ce que chaque processeur accède à des données différentes à chaque cycle machine.
Dans une première phase, pour réduire les coûts d'échange entre l'hôte et la carte graphique, nous construisons deux blocs de sous-séquences : le bloc 1[N, $n1$], avec N la longueur des sous-séquences et le bloc 2[$n2$,N].
Dans une deuxième phase, nous envoyons les deux blocs à la carte graphique pour effectuer le calcul des extensions sans gap. Les scores résultants sont stockés dans un bloc C[$n1$, $n2$]. La valeur de chaque cellule[i,j] du bloc C est le score d'extension sans gap entre la sous-séquences j du bloc 1 (correspondant à la ligne j) et la sous-séquence i de bloc 2 (correspondant à la colonne i).

Le calcul est ensuite partitionné en threads de la manière suivante : la matrice [n1,n2] est divisée en bloc [16,16] correspondant à 256 threads qui peuvent s'exécuter indépendamment. Chaque thread calcule un élément de la matrice 16x16 en plusieurs itérations. La figure 1-a illustre ce partitionnement qui, à l'origine, a été établi pour le calcul matriciel [3]. Nous avons repris ce schéma et nous l'avons adapté au calcul de distance entre séquences protéiques.
La mémoire partagée du GPU est divisée en 16 bancs qui permettent 16 accès simultanés à des données d'adresses différentes. Une lecture à une même adresse entraîne un conflit et, par conséquent, des cycles machines supplémentaires pour séquentialiser les accès. Il est donc extrêmement important, lors de la programmation, de prévoir un ordonnancement optimal des accès mémoire.
Dans l'implémentation du calcul des distances, la granularité est le calcul d'un bloc [16,16]. Celui-ci déclenche le transfert dans la mémoire partagée des données nécessaires au calcul (2 x 16 sous-séquences de 16 acides aminés). Les 16 sous-séquences du bloc 1 doivent être comparées avec les 16 sous-séquences du bloc 2. Il y a donc une forte réutilisation des données entre les calculs qui est gérée de la façon suivante : au début du calcul, chaque thread transfère un acide aminé de la mémoire principale vers la mémoire partagée. Cet acide aminé est ensuite partagé par plusieurs threads, mais à des instants différents. Il en résulte aucun conflit et une efficacité maximale.

### 3.3. Parallélisation de l'extension avec gap

A l'issue de l'étape 3, nous récupérons des alignements sans gap que nous cherchons à étendre en ajoutant un type d'erreur : l'insertion ou l'omission de caractères. Cette extension s'effectue à chaque extrémité de l'alignement. Aussi, pour un alignement particulier trouvé à l'étape 3, nous pouvons considérer qu'il est la source de deux tâches d'alignement avec gap. Plus généralement, K alignements sans gap vont impliquer la parallélisation de 2K tâches sur le GPU.
Comme pour l'extension sans gap nous construisons d'abord deux blocs de sous-séquences : le bloc 1 et le bloc 2. Un bloc correspond à 2K sous séquences de taille M. Ensuite, ces blocs sont transférés vers la carte graphique pour effectuer 2K extensions.
Dans ce cas, un bloc de threads est une matrice [1,256]. Dans un bloc, chaque thread correspond au calcul d'une extension. Par conséquent, une grille sera composée de $\frac{2K}{256}$ blocs de threads. Deux blocs de sous-séquences sont mappés dans la mémoire de texture du GPU. Ce type de mémoire est partagé à tous les threads en lecture seule. Mais la taille de la mémoire de texture pour chaque multiprocesseur est limitée à quelques dizaines de kilo octets. C'est la raison pour laquelle on ne peut pas garder 256 paires de sous-séquences dans cette zone mémoire.
De plus, l'extension avec gap utilise un algorithme de programmation dynamique qui demande un accès aux données plus complexe que dans le cas précédent. Pour éviter tout conflit mémoire, en début de calcul, chaque thread charge sa paire de sous-séquences dans sa mémoire locale.
Les scores résultants des 2K extensions sont stockés dans un bloc C[1,2K] comme le montre la figure 1-b ; le score d'extension d'une paire de sous-séquences à la position i est stocké dans la cellule C[1,i].

## 4. Résultats

Dans cette section, nous présentons les résultats de notre implémentation. Deux critères sont considérés par rapport au programme de référence BLAST : le temps d'exécution et la sensibilité.
Les tests ont été effectué sur un processeur Intel 2 Dual Core, 3 GHz, 2 Mo de cache L2, 2 Go de RAM, fonctionnant sous Linux (Fedora 6). Nous utilisons la carte graphique GeForce 8800 GTX ayant les caractéristiques suivantes : 16 multiprocesseurs SIMD à 675 MHz ; chaque multiprocesseur est composé de huit processeurs fonctionnant à deux fois la fréquence d'horloge ; le nombre maximum de threads par bloc est de 512 ; la taille de la mémoire sur la carte est de 768 Mo à 1,8 GHz ; la vitesse maximale observée de la bande passante entre la mémoire système et la mémoire de carte est de 2 Go/s. Le langage de programmation utilisé pour la carte graphique est CUDA.
Les jeux de données considérés sont :

- une partie de la banque PIR (version 80 01/2005) contenant 141.708 séquences protéiques de longueur moyenne 340 ;
- 4 extraits de la banque SWISSPROT (version 05/2007) contenant respectivement 5.000, 10.000, 20.000 et 40.000 séquences protéiques de longueur moyenne 367 ;

– une partie de GenBank (version 05/2007) contenant 27.360 séquences d'ADN de longueur moyenne 5.450.

Le temps d'exécution est calculé avec la commande time, la machine étant uniquement dédiée pour ce calcul. BLAST est lancé avec les paramètres par défaut, sauf pour un paramètre statistique, la e-value, qui est positionnée à $10^{-3}$ pour se positionner dans un contexte de comparaison intensive réel.

La sensibilité est évaluée par rapport aux nombres d'alignements trouvés dans les deux cas. Plus précisément, nous regardons si les alignements trouvés commencent et finissent aux mêmes endroits dans les deux banques avec une marge de P% calculée sur la taille moyenne des 2 alignements. Par exemple, pour comparer deux alignements de taille 100 à 5% près, nous vérifions que les positions de début et de fin des sous-séquences qui composent cet alignement ne sont pas éloignées de plus de 5 acides aminés.

Ces deux critères, temps d'exécution et sensibilité, ont été évalués sur deux programmes : iBLASTP-GPU et iTBLASTN-GPU. Ils sont les pendants des versions séquentielles iBLASTP et iTBLASTN.

Le tableau 2 compare : (1) les temps d'exécution des programmes iBLASTP-GPU et BLASTP avec comme paramètres la banque PIR et les 4 banques SWISSPROT (SWP) ; (2) les temps d'exécution des programmes iTBLASTN-GPU et TBLASTN avec comme paramètres GenBank et les 4 banques SWISSPROT (SWP). Des facteurs d'accélération de 5,5 et 10 sont respectivement obtenus pour iBLASTP-GPU et iTBLASTN-GPU.

| SWP (nb. séq.) | BLASTP (sec) | iBLASTP-GPU (sec) | Accélé. | TBLASTN (sec) | iTBLASTN-GPU (sec) | Accélé. |
|---|---|---|---|---|---|---|
| 5k | 3521 | 640 | 5,5 | 7063 | 773 | 9,1 |
| 10k | 6832 | 1186 | 5,6 | 13784 | 1372 | 10,0 |
| 20k | 13597 | 2420 | 5,6 | 27147 | 2613 | 10,4 |
| 40k | 26111 | 4581 | 5,6 | 52232 | 4942 | 10,5 |

TAB. 2 – *Temps d'éxécution de (i)BLASTP(-GPU) et de (i)TBLASTN(-GPU)*

Si on analyse plus précisément l'influence de la parallélisation sur les temps d'exécution des différentes étapes, on mesure des facteurs d'accélération compris entre 7,9 et 10,3 sur les étapes 3 et 4-1, comme le montre le tableau 3 :

| étapes | iBLASTP | | iBLASTP-GPU | | Accélé. | iTBLASTN | | iTBLASTN-GPU | | Accélé. |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2643 | 55,7% | 295 | 25,1% | 8,9 | 7264 | 74,9% | 703 | 53,1% | 10,3 |
| 4-1 | 1232 | 26,0% | 131 | 11,6% | 9,4 | 2054 | 21,1% | 259 | 19,5% | 7,9 |

TAB. 3 – *Temps d'exécution des deux étapes 3 et 4-1 de iBLASTP(-GPU) et de iTBLASTN(-GPU)*

La sensibilité a été évaluée en considérant deux valeurs de P : 2% et 10%. Les mêmes jeux de données que précédemment sont considérés. Le tableau 4 résume les résultats. Pour chaque exécution, le nombre d'alignements trouvés par BLASTP et iBLASTP-GPU (TBLASTN et iTBLASTN-GPU) est précisé ainsi que le pourcentage d'alignements jugés équivalents.

Les deux programmes détectent sensiblement le même nombre d'alignements, et environ 95% des alignements sont équivalents. La différence s'explique par le fait que les programmes BLASTP et iBLASTP-GPU (TBLASTN et iTBLASTN-GPU) ne se basent pas sur le même système de point d'ancrage. Dans les deux cas, les programmes passent à côté d'alignements significatifs dans des proportions quasiment égales. Les sensibilités sont donc équivalentes.

**5. Conclusion**

Dans ce papier, nous avons présenté la parallélisation d'un algorithme de recherche de similarités de séquences protéiques sur GPU basé sur la technique de points d'ancrage. A notre connaissance, il s'agit de

| SWP (nb. seq) | BLASTP (nb. align) | iBLASTP-GPU (nb. align) | 2% | 10% | TBLASTN (nb. align) | iTBLASTN-GPU (nb. align) | 2% | 10% |
|---|---|---|---|---|---|---|---|---|
| 5k | 305435 | 305939 | 94,7% | 95,4% | 290016 | 290446 | 95,6% | 95,9% |
| 10k | 611031 | 611393 | 95,3% | 95,8% | 572608 | 573930 | 96,0% | 96,4% |
| 20k | 1047794 | 1062602 | 94,8% | 95,4% | 1172466 | 1173633 | 96,4% | 96,6% |
| 40k | 2237076 | 2237797 | 94,5% | 95,6% | 2208330 | 2214335 | 96,4% | 96,7% |

TAB. 4 – *Comparaison de la sensibilité entre BLASTP et iBLASTP-GPU et entre TBLASTN et iTBLASTN-GPU*

la première tentative pour ce type d'algorithme, les implémentations sur GPU relevées jusqu'à présent dans la littérature faisant toutes référence aux algorithmes de programmation dynamique.
Deux programmes ont été développés et testés : iBLASTP-GPU et iTBLASTN-GPU. Ils font référence aux programmes BLASTP et TPLASTN de la famille BLAST quotidiennement utilisés par des milliers de biologistes. Pour ces deux programmes, nous obtenons respectivement des gains d'un facteur 5 et 10 sur la globalité des traitements, et des gains allant de 8 à 10 pour les étapes de l'algorithme effectivement parallélisées sur GPU.
Il est certain que les futurs GPU seront encore plus puissants et offriront encore plus de parallélisme. La parallélisation proposée ne pourra que bénéficier de ces futures améliorations : aujourd'hui, la limitation provient du fait que le GPU utilisé ne peut exécuter qu'un maximum de 256 threads simultanément alors que le nombre de petites tâches élémentaires indépendantes (les extensions avec et sans gap) est de plusieurs ordres de grandeur supérieur.
Ceci dit, une des limitations actuelle de notre mise en oeuvre est la part séquentielle des traitements (étape 4-2) qui bride de manière non négligeable l'accélération globale. La solution vient naturellement des travaux mentionnés précédement : cette étape utilise essentiellement un algorithme de programmation dynamique dont l'efficacité sur GPU a été démontré [4]. Nous pouvons donc raisonnablement encore espérer un gain substantiel en parallélisant sur GPU sur cette dernière tâche.

**Bibliographie**


1. Benson, D., Karsch-Mizrachi, I., Lipman, D., Ostell, J., Wheeler, D., GenBank, Nucleic Acids Research, Vol. 35, pp. 21-25, 2007.
2. Peterlongo, P., Noé, L., Lavenier, D., Georges, G., Jacques, J., Kucherov, G., Giraud, M., Protein similarity search with subset seeds on a dedicated reconfigurable hardware, Parallel Bio-Computing, Gdansk, Poland, 2007.
3. NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, version 1.0, 23/6/2007.
4. Liu, W., Schmidt, B., Voss, G., Schroder, A., Wolfgang, M., Bio-sequence database scanning on a GPU, International Workshop on High Performance Computational Biology, Rhodes Island, Greece, 2006.
5. Liu, W., Schmidt, B., Voss, G., Muller-Wittig, W., GPU-ClustalW Using graphics hardware to accelerate multiple sequence alignment, HiPC 2006, LNCS 4297, pp. 363-374.
6. Lavenier, D., Xinchun, L., Georges, G., Seed-based genomic sequence comparison using a FPGA / FLASH accelerator, International IEEE Conference on Field Programmable Technology, Bangkok, Thailand, 2006.
7. Lavenier, D., Fine-Grained Parallelism for Genomic Computation, SIAM Conf. on Parallel Processing for Scientific Computing, San Francisco, CA, 2006.
8. Cameron, M., Williams, H. E., Cannane, A., Improved gapped alignment in BLAST, IEEE Transactions on Computational Biology and Bioinformatics, Vol. 1, No. 3, 2004.
9. Darling, EA., Carey, L., Feng, W., The design, implementation, and evaluation of mpiBLAST, 4th International Conference on Linux Clusters, San Jose, USA, 2003.
10. Kent, W. J., BLAT–the BLAST-like alignment tool, Genome Research, Vol. 12, No. 4. April 2002, pp. 656-664.
11. Altschul, S., Madden, T., Schaffer, A., Zhang, J., Zhang, Z., Miller, W., Lipman, D., Gapped BLAST and PSI-BLAST : A new generation of protein database search programs, Nucleic Acids Research, Vol. 25, No. 17, pp. 3389-3402, 1997.
12. Altschul, S., Gish, W., Miller, W., Myers, E. W. Lipman, D., Basic local alignment search tool, J. Mol. Biology, 215, pp. 403-410, 1990.
13. Smith, T.F., Waterman, M.S., Identification of common molecular subsequences, J Mol Biol 1981, 147(1), 195-7.