Projet : Indexing genomes with k-mers

Alexandre Temperville, Master 2 Scientific Computing

23/02/2012

Introduction

Dans ce compte-rendu, je présenterai mon travail et tâcherai de répondre aux questions posées par l'énoncé. Je ne rentrerai pas trop en détail sur les fonctions et procédures de mon programme car s'y trouve des commentaires pour expliquer comment cela fonctionne.

J'ai supprimé la première ligne des fichiers de séquence commençant par le symbole '>' à l'aide de la commande shell : sed -i '/>/d' *fichier*
(on pouvait aussi le faire à la main mais ayant trouvé cette commande, je préfère la noter).

Le programme en CUDA se trouve dans le dossier 'Projet_Indexation_kmers', il est le fruit de plusieurs étapes progressives en suivant les questions que je ne mets pas car il réalise tout ce qu'il faut.
Pour le compiler j'effectuais la commande shell : nvcc TP.cu -O3 -o TP
Pour le lancer il fallait rentrer en ligne de commande :
TP *fichier1 fichier2 k (top)*
avec *fichier1* le fichier de référence, *fichier2* le fichier requête, k la taille des k-mers voulus et *top* le nombre des k-mers les plus fréquents et les moins fréquents que l'on souhaite afficher.

Le dossier 'Test_homme' contient des images de l'exécution du programme pour différentes valeurs de k sur la séquence du chromosome 1 humain. De même, on a dans le dossier 'Test_loup' les images de l'exécution du programme pour le chromosome 1 du loup. Ce sera exploité pour la question 7.

Enfin le dossier 'Comparaison_homme_loup' nous permet de comparer les séquences de l'homme avec celle du loup. J'attire l'attention sur la chose suivante : à partir de $k = 10$ inclus, ces résultats ne sont pas fiables, je l'explique dans la partie qui suit.

Limites du programme et premières remarques

Mon programme de base (sur CPU) ne fonctionne pas pour $k > 14$, où l'on gère des tableaux de taille 4^k ce qui devient vite très grand. Le type *unsigned long* ne permet pas d'aller plus loin que le nombre 2^{32} mais à $4^{14} = 2^{28}$ je ne l'atteins pas, je suppose que les performances machines sont aussi limitées en terme d'allocation mémoire étant donné que l'on utilise beaucoup de tableaux de ces tailles.

Mon programme sur GPU ne fonctionne pas pour $k > 9$, certainement à cause des limites en terme de nombre de threads.

Pour définir le nombre de blocks total (NB_BLOCK_SIZE), j'ai procédé à tâtons de sorte à obtenir des résultats cohérents, mais à partir de $k = 10$, j'aurais besoin de plus de threads que la machine

n'en a. Une solution serait de donner à un thread plusieurs tâches pour en réduire le nombre dont on a besoin. J'exploite partiellement cette approche pour la procédure *find_kmers_GPU* où chaque thread gère NB_BLOCK_MAX tâches au lieu d'un seul.

Il serait intéressant de pouvoir poursuivre cette idée pour toute les procédures sur GPU.

Remarque : sur un ordinateur 32 bits, les types int et long sont équivalents.

1 Décompte de k-mers

1.1 Fonction code

Cette fonction prend une chaîne de caractère et sa taille en paramètre et la convertit en un nombre représenté en base 4. Les nombres 0, 1, 2 et 3 correspondent respectivement à A, C, T et G. Si un caractère est un N, alors on utilise $rand()\%4$ pour lui faire correspondre aléatoirement à une lettre : A, C, T ou G.

1.2 Tableau index_kmers

La création de ce tableau contenant le nombre de k-mers présent dans un fichier est précisée dans le programme par des commentaires, on ne commence qu'à indexer dès lors que l'on a un mot de longueur k puis on le décale dans la liste de caractères de *tab_fichier1*.

1.3 k-mers plus ou moins fréquents

Une procédure de calcul et d'affichage des *top* k-mers les plus fréquents (*top* étant un nombre à rentrer en paramètre lors de l'exécution du programme) permet de savoir quels sont les k-mers que l'on voit le plus, est décrite dans la procédure *les10meilleurs*. De même, on crée facilement une procédure similaire permettant de calculer et afficher les *top* k-mers les moins fréquents (procédure *les10pires*).

Les mots du style AAAA..A, TTTT..T, CGCG..CG ... reviennent énormément et sont de faible complexité, j'ai donc créé une procédure pour créer un tableau regroupant les codes correspondants de 16 mots de faible complexité, il s'agit de la procédure *faible_complex*.

Une façon de se débarrasser de ces mots de faible complexité est d'utiliser dans les procédures *les10meilleurs* et *les10pires* un critère empêchant d'afficher un mot de faible complexité s'il a pourtant une valeur d'index satisfaisante pour être affiché. Ce critère est un booléen (variable *booleen*) qui vaut 1 si la valeur de l'index ne correspond pas à un mot de faible complexité et 0 sinon.

Dans le fichier 'Test_homme', j'ai fait des impressions écrans des résultats pour différentes valeurs de k sur la séquence du chromosome 1 de l'homme. On remarque aisément que les k-mers de grande taille les plus fréquents sont pour la plupart composés des k-mers les plus fréquents de plus petite taille. Ainsi on peut à partir de $k = 4$ par exemple tenter d'assembler certains k-mers pour deviner quels k-mers de plus grande taille pourraient apparaitre fréquemment. Cela se vérifie facilement en passant de $k = 4$ à $k = 6$ puis $k = 8$. Mais lorsque l'on passe à $k = 10$, cette fois ci des mots différents des meilleurs pour des valeurs de k plus petites apparaissent, il n'y a donc pas de critère de construction à partir des meilleurs mots de petite taille des plus grands, mais il est normal qu'il y ait une forte probabilité que lorsque l'on augmente un peu la taille k, on retrouve des similitudes entre ces mots.

On retrouve à peu près le même raisonnement pour les mots les moins fréquents, à la différence qu'à partir d'une certaine valeur de k (ici 10), évaluer les k-mers les moins fréquents n'a plus d'intérêt car la probabilité de tomber sur des k-mers n'apparaissant jamais augmente donc on afficherait des k-mers n'apparaissant jamais alors qu'il y en a beaucoup d'autres, cette donnée n'est donc pas représentative

pour k-grand (inversement, la procédure *les10meilleurs* n'a pas vraiment d'intérêt pour k petit).

1.4 Fonction find_kmers

Ma fonction *find_kmers* établit le nombre de mots en commun dans chaque fichier en prenant le minimum d'apparition de ce mot entre la séquence requête et la séquence de référence et l'additionne à un résultat successivement pour chaque mot. Cette façon de procéder permet de donner une estimation de correspondance assez intéressante pour comparer deux séquences mais ne signifie pas forcément que si 2 séquences ont un mot en commun le même nombre de fois, ce mot apparaisse aux mêmes positions dans les séquences autour de mots similaires et/ou ayant des propriétés équivalentes. Ce critère n'est qu'un indicateur, un indice qui ne se soustrait pas à l'utilisation d'autres indicateurs utilisés en biologie, certainement plus spécifiques que le mien. J'appelle ce critère indice de correspondance lors de l'affichage final après exécution de mon programme.

Une autre idée aurait été de dire de créer un critère de présence d'un mot dans les 2 séquences comparées, peu importe leur nombre de présence. Ce critère serait intéressant si l'on veut juste savoir si une séquence nucléique ayant une propriété peut être trouvée dans une autre.

2 Parallélisation OpenCL

Une tentative de parallélisation 'OpenCL' a été effectuée dans le dossier 'Essai_OpenCL', j'aurais aimé abordé cette approche plutôt que l'autre, mais je ne comprenais pas les erreurs que l'écran me retournait et n'ai pas réussi à comprendre et modifier la structure des fichiers adjacents que je devais utiliser. Je me suis donc mis à paralléliser sur CUDA.

3 Parallélisation CUDA

3.1 Version GPU1 : find_kmers sur le GPU

La version réalisant l'indexation de la séquence de référence sur le CPU et la recherche du nombre de k-mers en communs sur le GPU est appelée dans mon programme GPU1.

Pour faire le même travail sur GPU que sur CPU, dans un premier temps on construit une procédure de recherche des k-mers dans le fichier 'TP_kernel.c'.

Pour l'expliquer j'utilise la figure 'find_kmers_GPU.png'. Chaque thread stockera dans la case jaune de *index_kmers2_device* son nombre de k-mers calculés apparaissant dans la séquence de référence. Travaillant avec énormément de données et risquant de ne pas avoir assez de blocs, je demande à chaque thread d'effectuer plusieurs instructions représentées par les traits de couleurs verts et violets.

Chaque trait vert représente la première étape de chaque thread : le calcul du minimum de 2 cases des deux tableaux de séquences et son affectation à la case jaune. Ensuite, chaque gros trait violet représente le calcul du minimum du contenu des 2 cases, puis son ajout à la valeur de la case jaune à qui elle est liée par un trait violet plus fin. Chaque thread fait donc ici NB_THREAD opérations.

Ensuite chaque case jaune est introduite dans un nouveau tableau plus petit de sorte qu'elles soit successive. Et cela nous permet de somme de sorte à ne pas contraindre les threads à sommer sur une même case à chaque étape. Ce procédé est particulièrement utilise lorsque l'on veut diminuer rapidement le nombres de blocs ayant à s'échanger des données pour pouvoir additionner des éléments de tableaux

entre eux.

3.2 Version GPU2 : indexation des k-mers sur le GPU

Pour réaliser cela, il faut construire une procédure *code_ GPU* sur le GPU et une procédure de création de l'index (creation_index_GPU. Etant simple et ressemblant fortement à leurs équivalents sur CPU, il ne me semble pas nécessaire de les expliquer (voir le programme).

4 Comparaison entre plusieurs génomes

Pour voir les k-mers apparaissant le plus ou le moins dans les deux génomes, on regardera les dossiers 'Test_homme' et 'Test_loup' pour voir les résultats donnés.
Il serait intéressant d'afficher les mots apparaissant le plus en commun dans les deux fichiers, je n'ai pas eu le temps de le faire mais ce n'est pas compliqué à programmer. On peut regarder les indices de correspondances que j'ai créé pour différentes valeurs de k dans le dossier 'Comparaison_homme_loup'.

Vitesse d'exécution

J'ai voulu afficher le temps d'exécution de chaque méthode, i.e. celle n'utilisant que le CPU, la première version GPU (GPU1) puis la seconde version GPU (GPU2). Ces résultats sont visualisés en fin de programme. Pour chaque méthode, je compte toutes les étapes qui lui sont nécessaires. On observe que plus l'on fait de choses sur le GPU, plus la vitesse d'exécution diminue, ce qui est cohérent vu que l'on exécute énormément d'instruction en parallèle. Ainsi avec la version GPU2, je peux réaliser mes instructions en moins d'une seconde tandis qu'il m'en faudra énormément plus juste sur le CPU.