Université Joseph Fourier
Département Licence Sciences & Technologie

RAPPORT DE STAGE
Calcul parallèle sur GPU

D'Aguanno Carlotta

Laboratoire d'accueil : INRIA
Directeur du laboratoire : GROS Patrick
Responsable du stage : HUARD Guillaume

Licence MIN - 1ère année
Année Universitaire : 2014-2015

Sommaire

1 Remerciements **4**

2 Introduction **5**

2.1 INRIA 5

2.2 Le calcul parallèle 5

3 Mon expérience **7**

3.1 Introduction 7

3.2 Calcul 7

3.2.1 Objectif 7

3.2.2 Méthode expérimentale et résultats 7

3.2.3 Conclusion 10

3.3 Mémoire 10

3.3.1 Introduction 10

3.3.2 Méthode expérimentale et résultats 11

3.3.3 Conclusion 13

3.4 Les outils utilisés 13

4 Bibliographie **16**

5 Conclusion **17**

1 Remerciements

Je tiens tout d'abord à remercier mon responsable de stage Mr Guillaume HUARD qui a eu le gentillesse de m'accueillir au sein de son équipe et grâce à qui j'ai pu découvrir le métier de chercheur. Je le remercie tout particulièrement pour le temps qu'il m'a accordé au cours de mon stage pour m'expliquer de nouvelles notions et m'encourager, mais également pour le temps qu'il m'a consacré à la suite de ce stage pour m'aider à rédiger mon rapport.

Je remercie ensuive Mr David BENIAMINE, actuellement en thèse à l'INRIA, pour le temps qu'il a passé avec moi, à me montrer comment me servir des différents outils et à me guider tout le long du stage vers de nouveaux objectifs.

Je remercie également Mme Annie SIMON, l'assistante de l'équipe de recherche avec laquelle j'ai travaillé pour avoir passé du temps à s'occuper de mon dossier et pour m'avoir aidé à regler mes problèmes de conventions de stage ainsi que toute l'équipe pour m'avoir accueillie si gentiment et pour m'avoir donné des conseils.

Enfin, je remercie le DLST et tout particulièrement Mme Patricia CAJOT pour m'avoir donné l'opportunité de faire ce stage.

2 Introduction

Mon stage s'est déroulé à l'INRIA et a porté sur le calcul parallèle sur GPU dont je vais vous faire une petite introduction ci-dessous.

2.1 INRIA

L'INRIA est un centre de recherche crée à l'époque de De Gaule et compte aujourd'hui presque une dizaine de centres en France. Le centre Rhône-Alpes a été crée en 1992. A Grenoble le directeur du centre est Patrick GROS depuis le 1er Décembre 2014. La plupart des équipes-projets Inria grenobloises sont des équipes communes avec le CNRS, l'université Joseph Fourier, et Grenoble INP au sein des laboratoires :

— Laboratoire d'informatique de Grenoble (LIG)

— Laboratoire Jean Kuntzmann (LJK)

— Laboratoire Grenoble Images Parole Signal Automatique (GIPSA)

— Laboratoire Adaptation et Pathogénie des Microorganismes (LAPM)

L'équipe qui m'a accueillie est l'équipe MOAIS (Multi-programmation et Ordonnancement pour les Applications Interactives de Simulation) dont le thème de recherche est le calcul distribué et à haute performance et le responsable est Jean louis ROCH. Les axes de recherche de l'équipe-projet MOAIS sont centrés sur le problème de l'ordonnancement avec un objectif de performance multi-critère : précision, réactivité.

2.2 Le calcul parallèle

Le processeur (CPU : Central Processing Unit) est l'élement central d'un ordinateur, il permet l'éxecution des différentes commandes de l'utilisateur, ainsi que le fonctionnement de tous les programmes présents dans la machine. Avec l'évolution constante de ces programmes informatiques, les processeurs doivent être toujours plus puissants afin de permettre de résoudre des problèmes toujours plus gros et complexes. Chaque nouvelle génération de processeur a largement gagné en fréquence et a permis à l'informatique de progresser rapidement jusqu'en 2003 où on arrive aux limites de ce genre d'améliorations. A la suite de ça, les industriels mettent en place les processeurs multi-coeurs, ce qui permet de maintenir la progression des performances jusqu'à nos jours. Cependant, un programme séquentiel ne va fonctionner que sur un seul coeur donc la mise en place des multiprocesseurs ne va pas forcément permettre d'augmenter les performances ce qui a conduit au début de la programmation parallèle pour le grand public.

Les processeurs graphiques (GPU : Graphics Processing Unit) existent déjà à cette époque mais ne sont utilisées que pour le traitement des images, ces dernières pouvant être schématisées de façon très simplifiée par de grandes matrices qui représentent les pixels et sur lesquelles des calculs sont effectués en parallèle. Certains programmeurs se sont rendus compte qu'ils pourraient se servir des cartes graphiques afin d'appliquer ces calculs à d'autres données que des images et profiter du parallélisme massif mis en oeuvre par le GPU. De plus en plus de personnes s'intéressent à cette façon de procéder puisqu'elle permet de diminuer de façon significative les temps d'éxecution losrqu'on a de nombreux calculs à effectuer et qu'il est possible de paralléliser l'application.

La programmation sur carte graphique est très difficile au début puisque chaque chose que doit faire l'ordinateur doit être détaillée et qu'il faut exprimer les choses à faire en termes de transformations d'images.

Nvidia invente donc un nouveau langage de programmmation appelé CUDA qui permet de simplifier le code. CUDA permet de décrire ce que fait la machine mais en une seule fois si on a que des calculs identiques à effectuer, c'est ensuite le langage qui va permettre la distribution du travail sur les différents processeurs de la carte graphique.

Les nouveaux programmes ainsi écrit sont executés de la façon suivante : l'ordinateur commence par exécuter le programme de façon séquentielle sur le CPU, puis quand il rencontre une partie de code en CUDA il lance de travail de façon parallèle sur la carte graphique, les résultats sont ensuite récuperés et l'exécution reprend sur le CPU.

Chaque génération de GPU voit toujours plus de coeurs apparaitre pour toujours plus de performances. La division du problème grâce à CUDA peut être décrite de la façon suivante : on divise le problème en grilles, elles-mêmes divisées en blocs possédant des threads. Chacun des threads effectue les calculs de façon parallèle. Voici un schéma qui représente cette division : [1]

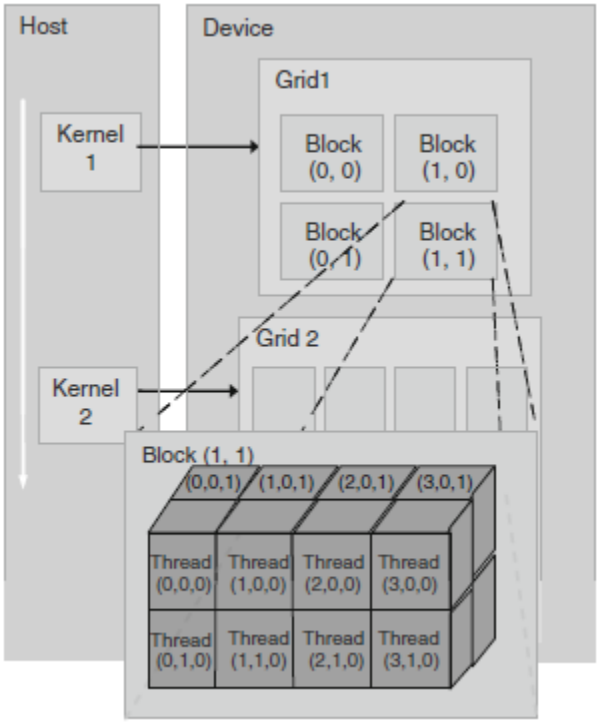


FIGURE 1 – Division en CUDA

3 Mon expérience

3.1 Introduction

Dans cette partie, nous allons nous intéresser à deux façons de rendre un programme plus rapide, tout d'abord nous allons voir l'effet apporté par l'utilisation de la carte graphique, et du parallélisme qu'elle apporte, au niveau des temps d'éxecution et ensuite les différences de temps apportées par une bonne utilisation de la mémoire. Pour cela, nous allons nous intéresser à un problème classique de la programmation parallèle : la multiplication de matrices. En effet, le nombre de calculs à effectuer pour multiplier deux matrices croit de façon cubique quand la taille des matrices augmente, d'où l'interêt de trouver des solutions pour en diminuer le temps de calcul. Enfin, nous verrons les différents programmes que j'ai eu l'occasion de découvrir tout au long de mon stage.

3.2 Calcul

3.2.1 Objectif

L'objectif de cette sous-partie est d'étudier et de comparer les temps de calcul de multiplications de matrices de différentes tailles afin de savoir si l'utilisation du GPU est réellement plus avantageuse que l'utilisation du CPU seul.

3.2.2 Méthode expérimentale et résultats

J'ai tout d'abord commencé par écrire un programme qui calcule la multiplication de matrices sur le CPU de façon linéaire en effectuant les calculs une ligne après l'autre en affichant les résultats à l'écran pour vérifier que les calculs effectués par l'ordinateur étaient justes et que mon programme fonctionnait correctement. Voici un exemple de programme qui calcule une multiplication de matrice sur CPU : [1]

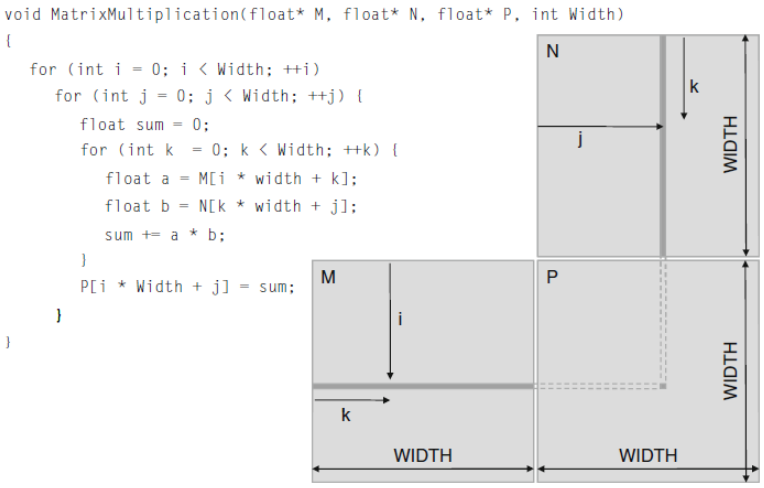


FIGURE 2 – Exemple de multiplication de matrices sur CPU

Je me suis ensuite beaucoup renseignée sur la programmation sur GPU en lisant de la documentation avant de programmer une version sur GPU qui se limitait à des matrices de taille 32*32 à cause de la limitation en nombre de threads de ma carte graphique. N'ayant utilisé qu'un seul bloc elle se limite a 1024 threads d'où la matrice de taille 32*32. J'ai ensuite effectué des test sur des tailles de matrices qui augmentent de 1*1 a 32*32 pour comparer les temps de calcul sur GPU et sur CPU. Le tout premier constat qu'on peut faire est que le temps de calcul est inférieur sur le GPU, cependant, la version GPU du programme contient une partie de transfert de données vers la carte graphique, si on prend en compte ce temps de transfert, la version GPU est plus couteuse en temps que la version CPU sur des matrices aussi petites, d'où la nécessité de faire les tests sur des matrices plus grandes.

J'ai donc modifié le programme de façon à ce qu'il puisse calculer la multiplication de matrices de taille arbitraire. Voici l'exemple d'un tel programme de multiplication de matrices sur GPU : [1]

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // Calculate the row index of the Pd element and M
 int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
  // Calculate the column idenx of Pd and N
 int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

 float Pvalue = 0;
 // each thread computes one element of the block sub-matrix
 for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

 Pd[Row*Width+Col] = Pvalue;
}
```

FIGURE 3 – Exemple de multiplication de matrices sur GPU

Pour cela, il a fallu comprendre la séparation de la carte graphique en grilles et en blocs de façon à utiliser tous les threads disponibles sur celle-ci. En séparant la matrice résultat en blocs on peut calculer des matrices de tailles aribitraire et ainsi comparer avec le CPU. Dans cette version il est nécessaire de donner à l'éxecution la taille de bloc que l'on souhaite telle que taille-bloc*taille-bloc soit le nombre de threads par bloc. Voici la façon dont on appelle la fonction dans le main en donnant la taille de bloc et de grille : [1]

```
// Setup the execution configuration
  dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH);
  dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);


// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

FIGURE 4 – Exemple d'appel de fonction sur GPU

Pour tester quelle serait la taille de bloc idéale on effectue des tests où on garde une taille de matrice fixe une taille de bloc qui varie, ainsi on peut observer l'influence de la taille des blocs sur le temps de calcul. On a tout d'abord lancé une expérience où on a pris des matrices de tailles 1500*1500 et des tailles de blocs qui

varient de 1*1 à 150*150 et on a obtenu des résultats étranges puisque la taille des blocs semblaient avoir une influence sur le temps de transfert.

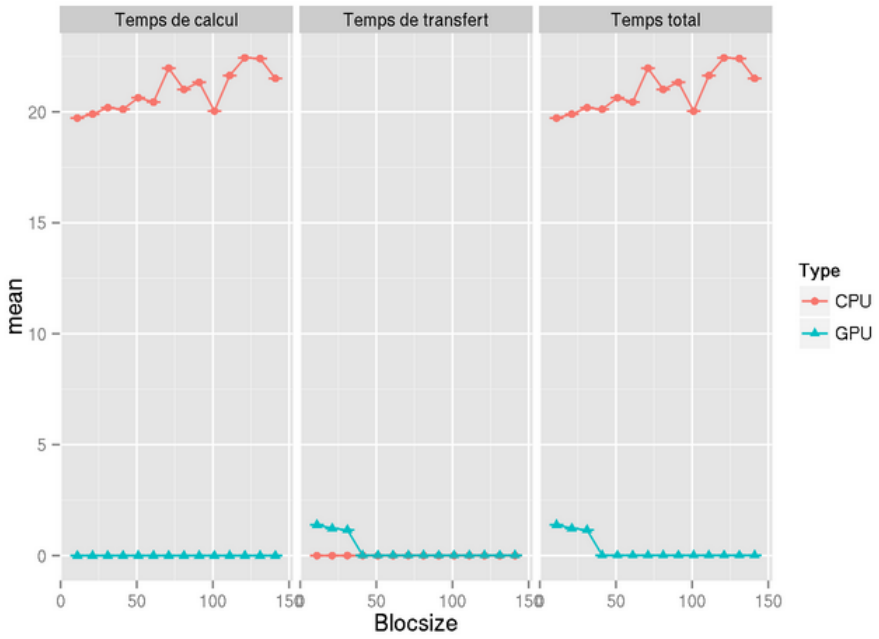


FIGURE 5 – Courbe avec un temps de transfert étrange

Pour verifier cette hypothèse on relance la même expérience mais en faisant décroitre la taille des blocs pour vérifier si c'est bien la taille des blocs qui semble avoir une influence ou si c'est l'ordre dans lequel on effectue les calcul. On obtient exactement le même résultat qu'avant. Après recherche, on s'est rendu compte qu'il y avait un problème dans la façon de calculer les temps, en effet, le GPU fait une partie de ce qu'on lui demande de façon asynchrone donc les temps relevés ne correspondaient pas. Après modification du programme en ajoutant des synchronisations des threads on a pu obtenir de meilleures courbes. Sur ces courbes on constate que le temps de calcul sur des matrices de tailles 1500*1500 est largement inférieur sur le GPU que sur le CPU même en prenant en compte le temps de transfert.

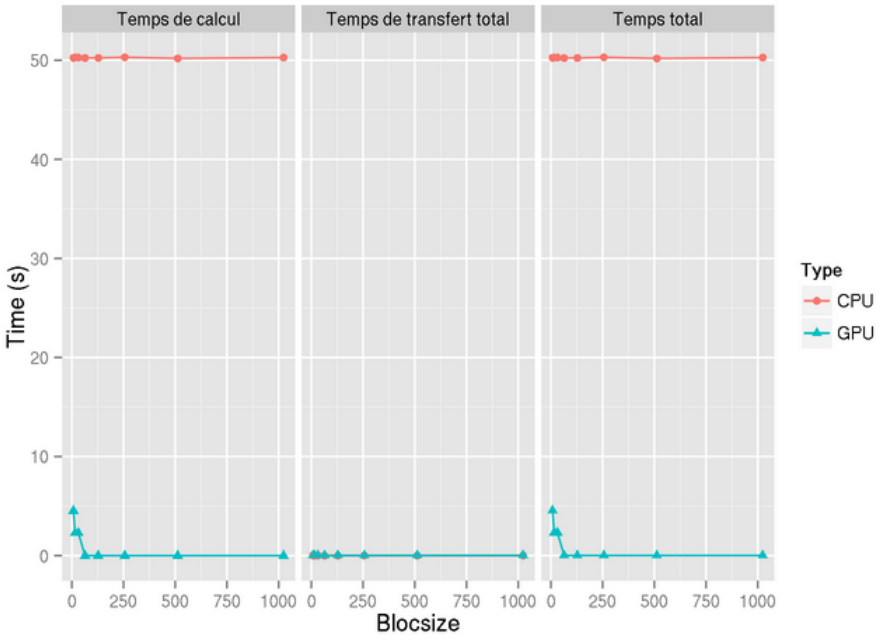


FIGURE 6 – Temps de transfert lissé

En utilisant des tailles de matrices et de blocs qui sont des puissances de 2 on obtient une courbe de calcul sur CPU qui est plus lisse puisqu'on utilise la mémoire d'une meilleure façon, quand on utilise des tailles qui ne sont pas des puissances de deux on a un problème d'alignement en mémoire qui fait qu'on n'utilise pas le cache et le bus mémoire de manière optimale. On voit effectivement que la courbe CPU est complètement lisse sur l'image ci-dessus.

3.2.3 Conclusion

A la fin de cette série de tests on a pu conclure que le temps de calcul était plus court sur GPU que sur CPU dans tous les cas mais que le temps de transfert ayant un cout il n'était plus avantageux qu'à partir d'une certaine taille de matrice.

3.3 Mémoire

3.3.1 Introduction

On va maintenant effectuer une série de tests qui permettent de visualiser l'importance d'utiliser correctement la mémoire de l'ordinateur pour obtenir de meilleurs résultats. En effet, l'ordinateur possède différents niveaux de cache qui ont des couts temporels différents. Plus le niveau est petit et rapproché plus il sera rapide de récuperer des informations dedans. En utilisant ces différents niveaux de cache on peut éviter des lectures mémoires répétées à l'ordinateur et ainsi rendre le programme plus rapide.

3.3.2 Méthode expérimentale et résultats

On commence tout d'abord par écrire une nouveau programme sur GPU qui va utiliser la mémoire partagée. C'est une sorte de cache dont l'accès est explicité. Cela consiste à créer un tableau dans lequel tous les threads d'un bloc pourront écrire et lire. De cette façon, chaque thread qui calcule une ligne du tableau résultat copie les lignes lues des matrices de départ et les met dans ce tableau de façon à ce que le thread suivant puisse également s'en servir sans devoir aller récuperer l'information jusque dans la memoire. De cette façon on améliore le temps d'exécution. On peut effectivement voir sur la courbe ci-dessous qu'il y a de meilleurs résultats avec un tableau partagé :

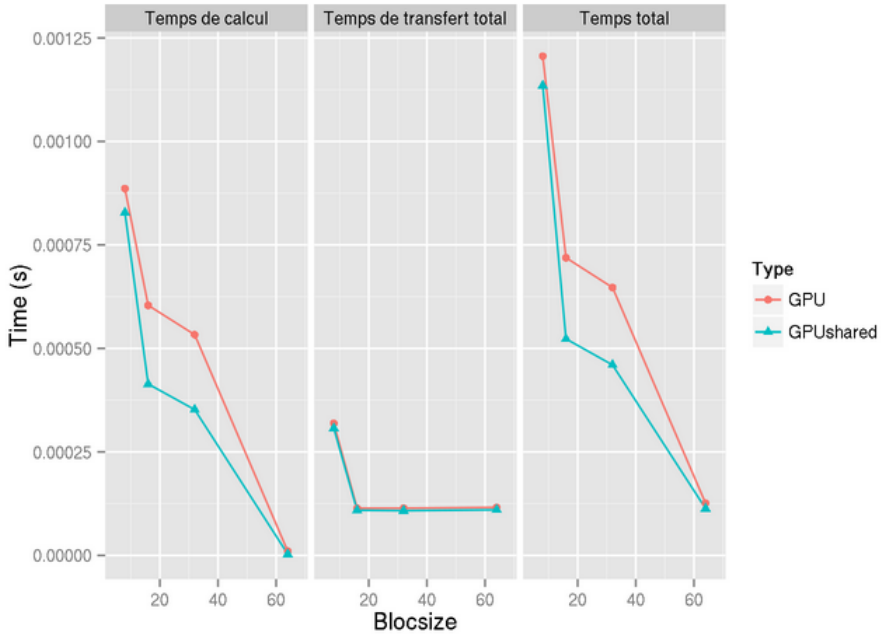


FIGURE 7 – Comparaison entre la version GPU et celle avec des tableaux partagés

On veut ensuite essayer d'observer les différents niveaux de cache sur le GPU et sur le CPU, pour cela on choisit une taille de bloc fixe de 64*64 et on fait ensuite varier la taille de la matrice de 8*8 a 4096*4096. On obtient des courbes qui ont la bonne allure mais qui n'ont pas assez de points pour observer réellement le phénomène.

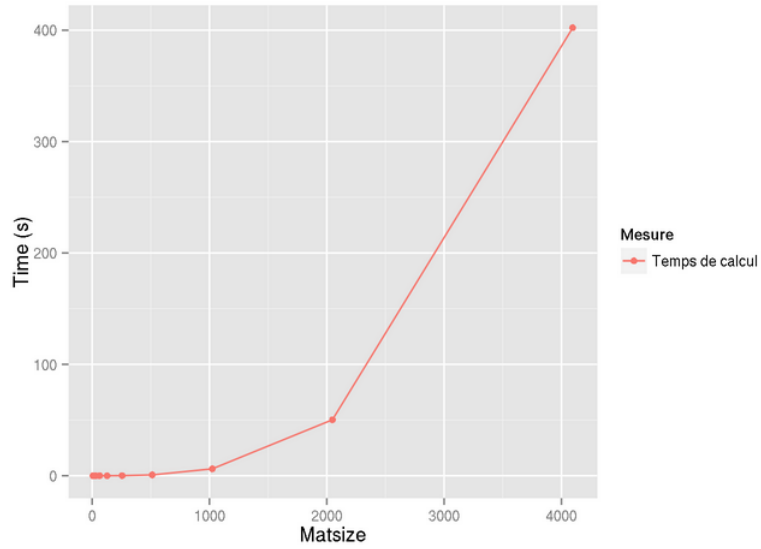


FIGURE 8 – Courbe sur CPU

On devrait voir des paliers en marche d'escalier quand on sort d'un niveau de cache puisque les temps de lecture mémoire sont plus long d'un coup. On relance donc l'expérience avec plus de points. On fait varier la taille de la matrice de 8*8 à 10000*10000 en la multipliant par deux jusqu'à 512 puis en rajoutant 512 à chaque fois et on calcule ça sur le CPU. On obtient une courbe bien plus lisse mais on ne voit toujours pas d'allure escalier.

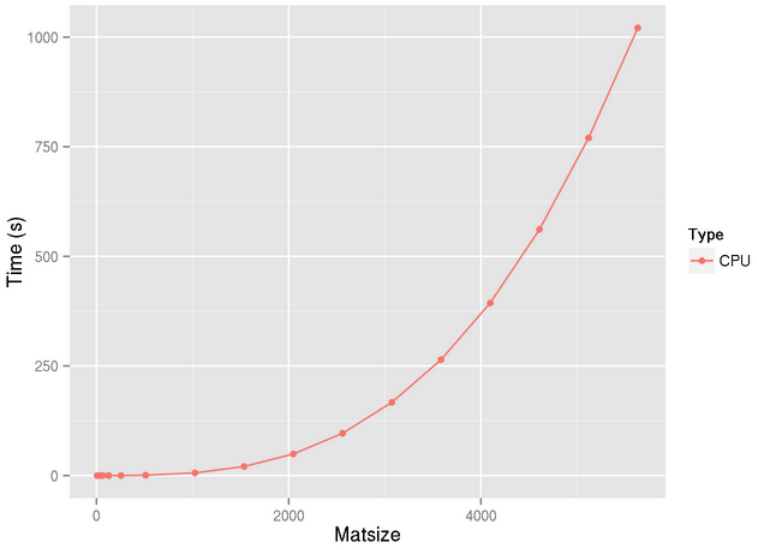


FIGURE 9 – Courbe sur CPU avec plus de points

On effectue le même test sur le GPU shared et normal et on obtient la même chose que pour le CPU.

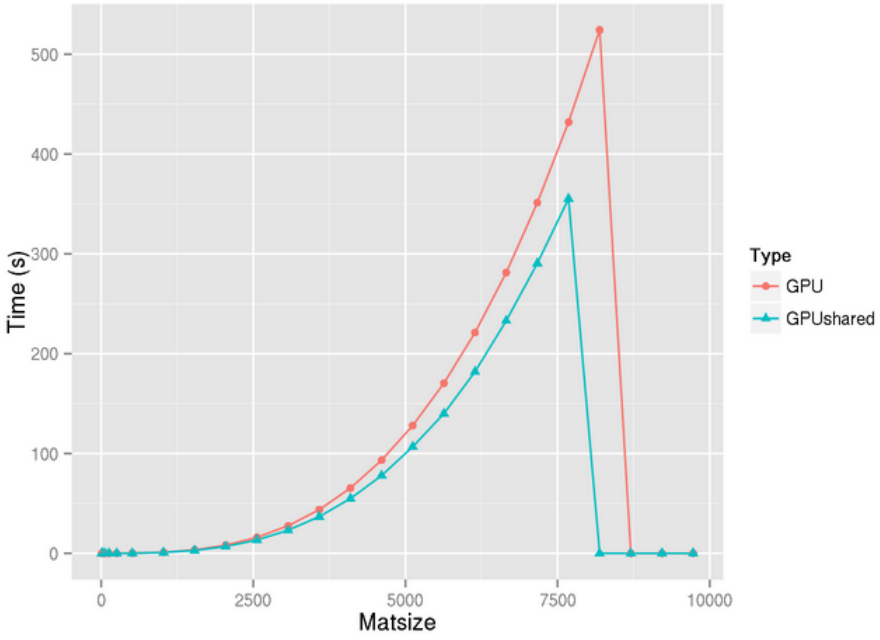


FIGURE 10 – Courbe sur GPU et GPUshared

Ce qu'il est intéressant de voir sur cette courbe c'est le gain qu'on obtient avec un tableau partagé par rapport à la version précédente. En effet, on devrait gagner un facteur constant égal à la largeur du bloc qu'on arrive à mettre en cache. On voit bien que les deux courbes ont une allure cubique et qu'elles semblent séparées par un facteur constant.

Le fait qu'on ne voit pas l'allure escalier peut s'expliquer par le fait qu'on cherche peut être à voir un phénomène trop petit ou que les nouveaux systèmes de cache ont un système de lissage qui ne permet pas de bien voir les sauts. De plus, on observe sur la courbe du GPU que la courbe s'écrase à partir 8000, ceci s'éplique surement par le fait que les matrices deviennent trop grosses et que le transfert vers la carte graphique n'est plus possible puisqu'elles ne rentrent plus dans la mémoire.

3.3.3 Conclusion

On a donc pu voir que l'utilisation de la mémoire est très importante quand il s'agit d'optimiser un programme. De plus, ces expériences nous suggèrent que si on faisait une version bloc qui utilise le cache de façon efficace sur CPU on pourrait améliorer les temps de calcul.

3.4 Les outils utilisés

Tout au long de mon stage j'ai été amenée à utiliser des logiciels que je ne connaissais pas et à apprendre de nouveaux langage de programmation, parmis ceux-là se trouve latex avec lequel j'ai rédigé ce rapport, c'est un outil qui permet de rédiger des textes sans avoir à se préocuper de la mise en page, quelques commandes s'occupent ensuite à la compilation de tout mettre au bon endroit. Voici quelques exemples du langage de

latex :

rapport.tex

```
%!TEX encoding=UTF-8 Unicode

\documentclass{article}[12pt]
\usepackage{fullpage}
\usepackage{setspace}


%=========================encodage fontes et langue============================

\usepackage[utf8]{inputenc}
\usepackage[francais]{babel}
\usepackage[T1]{fontenc}    % (pour les accents)

%==============================================================================

%=========================Listings: code mis en forme==========================

\usepackage{listings}
%definition d'un language algorithmique:
\lstdefinelanguage{algo}%
{
    alsoletter={\\,[,],/,*,\,},%
    morekeywords=[2]{si, sinon, alors, finSi, pour tout, finPour,
    tantQue, finTantQue, ou, et, non, vrai, faux},%
    otherkeywords={},%
```

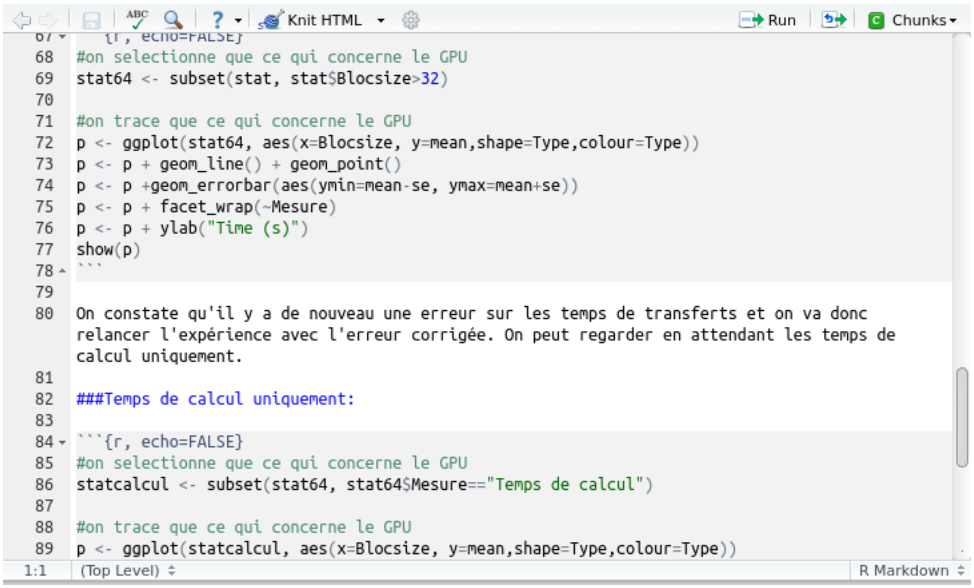
Log and Messages | Output | Konsole | Preview

rapport.tex

```
\null
\newpage

\begin{center}
        \renewcommand{\contentsname}{Sommaire}
        \setcounter{tocdepth}{5}
    \tableofcontents
\end{center}

\newpage


\section{Remerciements}

\newpage

\section{Introduction}
        \subsection{INRIA}

        L'INRIA est un centre de recherche crée en 1992 et qui est installé à Grenoble et Lyon.
        \begin{itemize}
        \item Laboratoire d'informatique de Grenoble (LIG)
    \item Laboratoire Jean Kuntzmann (LJK)
    \item Laboratoire Grenoble Images Parole Signal Automatique (GIPSA)
    \item Laboratoire Adaptation et Pathogénie des Microorganismes (LAPM)
    \end{itemize}
```

Log and Messages | Output | Konsole | Preview

Je me suis également beaucoup servie de rstudio qui est un outil pour tracer des courbes et faire des statistiques. Il utilise le langage de programmation R markdown qui permet de mélange langage de programmation et texte. Voici un exemple de code utilisé pour tracer une courbe :


Knit HTML
Run
Chunks
```{r, echo=FALSE}
#on selectionne que ce qui concerne le GPU
stat64 <- subset(stat, stat$Blocsize>32)
#on trace que ce qui concerne le GPU
p <- ggplot(stat64, aes(x=Blocsize, y=mean,shape=Type,colour=Type))
p <- p + geom_line() + geom_point()
p <- p +geom_errorbar(aes(ymin=mean-se, ymax=mean+se))
p <- p + facet_wrap(~Mesure)
p <- p + ylab("Time (s)")
show(p)
```
On constate qu'il y a de nouveau une erreur sur les temps de transferts et on va donc relancer l'expérience avec l'erreur corrigée. On peut regarder en attendant les temps de calcul uniquement.
###Temps de calcul uniquement:
```{r, echo=FALSE}
#on selectionne que ce qui concerne le GPU
statcalcul <- subset(stat64, stat64$Mesure=="Temps de calcul")
#on trace que ce qui concerne le GPU
p <- ggplot(statcalcul, aes(x=Blocsize, y=mean,shape=Type,colour=Type))
1:1
(Top Level)
R Markdown
```

4 Bibliographie

Références

[1] David B. Kirk and W. Hwu Wen-mei. *Programming massively parallel processors : a hands-on approach.* Newnes, 2012.

5 Conclusion

Pendant ce mois de stage dans le laboratoire INRIA j'ai pu découvrir une nouvelle manière de programmer : la programmation parallèle. J'ai pu effectuer des tests qui comparent les temps d'exécution de la programmation linéaire et de la programmation parallèle ce qui m'a permis de me rendre compte de l'importance d'optimiser les programmes informatiques afin d'accroitre les performances. J'ai également eu la chance d'apprendre à utiliser des outils informatiques qui pourront m'être très utiles pour la suite de mes études.

L'utilisation de tous ces outils et le contact avec les chercheurs du laboratoire m'ont permis de comprendre quelles étaient les étapes importantes qu'il faut impérativement effectuer quand on effectue des recherches et que l'on écrit des rapports qui présentent ces recherches.

Ce stage m'a également permis de me rendre compte des difficultés que l'on pouvait rencontrer lorsqu'on se retrouve à devoir chercher seul une solution sans être guidés (comme dans nos TP au DLST par exemple) et quelles étaient les solutions pour avancer petit à petit.

En conclusion je peux donc dire que ce stage a été très bénéfique pour moi, il m'a permis de me rendre compte de ce qu'étais réellement le métier de chercheur, m'a apporté beaucoup de connaissances et m'a aidé à savoir un peu plus si ce métier étais fait pour moi.