Université Joseph Fourier
Département Licence Sciences & Technologie

RAPPORT DE STAGE

Calcul parallèle sur GPU

Picard Michaël
6 juin 2016 - 1er juillet 2016

Laboratoire d'accueil : LIG - INRIA
Équipe : POLARIS
Directeur du laboratoire : GAUSSIER Eric (LIG)
GROS Patrick (INRIA)
Responsable du stage : HUARD Guillaume

License MIN - 1ère année
Année universitaire : 2015 - 2016

Cette page a été laissé intentionnellement blanche.

Sommaire

1 Remerciement

Je tiens à remercier Mr HUARD Guillaume pour son accueil au sein de l'équipe Polaris, pour sa gentillesse et pour le temps qu'il m'a accordé pour m'expliquer de nouvelles notions, tant utilitaire, méthodique qu'algorithmique, pour ses encouragements et son soutien durant ce stage, qui m'a permis de découvrir le métier d'enseignant-chercheur.

Je remercie aussi Mr BENIAMINE David, actuellement en thèse à l'INRIA, pour le temps passé à m'accueillir sur les locaux et ses conseils utiles pour me permettre d'avancer durant le stage.

Je remercie Mme SIMON Annie, l'assistante de l'équipe Polaris pour son temps sur mon dossier, et pour m'avoir aidé à régler les nombreux imprévus à propos des conventions de stages et autres formalités.

Je remercie aussi l'INRIA et le LIG ainsi que leur directeur, Mrs GAUSSIER Eric et GROS Patrick, pour m'avoir permis de participer à une présentation d'algorithmie à des lycéens, dans leur locaux de Montbonnot, sous la tutelle de Mr HUARD.

Je remercie enfin l'UGA, le DLST et Mmes MANDON Nina, COGNE Lydie, DARRACQ Marie-Cécile et CAJOT Patricia pour m'avoir donné l'opportunité de participé à ce stage.

2 Introduction

Mon stage à l'Inria portant sur le Calcul parrallèle sur GPU, j'ai été amené à utilisé de nombreux outils.
Je vais donc vous présenter en premier lieu la structure d'accueil du stage, puis je vous parlerai des différents outils utilisée durant ce stage, la raison pour laquelle j'ai choisi ce stage, avant de vous présenter mon évolution durant mes 4 semaines en tant que stagiaire.

2.1 Environnement d'acceuil : INRIA & LIG

L'INRIA est un centre de recherche crée à sous le régime du Général DeGaule et compte aujourd'hui presque une dizaine de centres en France. Le centre Rhône-Alpes a été crée en 1992.

À Grenoble le directeur du centre est Mr GROS Patrick 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)

Le LIG est quant à lui composé d'une multitude d'équipe :

- POLARIS
- DATAMOVE
- DRAKKAR
- ERODS
- ...

L'équipe qui m'a accueilli est l'équipe POLARIS, nouvellement installée au batîment IMAG sur le campus universitaire de Grenoble. Le responsable du LIG est Mr GAUSSIER Eric, tandis que le responsable de l'équipe POLARIS est Mr Legrand Arnaud. Le thème de recherche de cette équipe est « Calcul distribué et à haute performance », plus précisément « Performance analysis and optimization of LARge Infrastructures and Systems ».

2.2 Outils et Méthodes utilisés

Dans cette partie, je vais vous présenter les différentes méthodes de travail et les languages (et logiciel) appris pour ce stage.

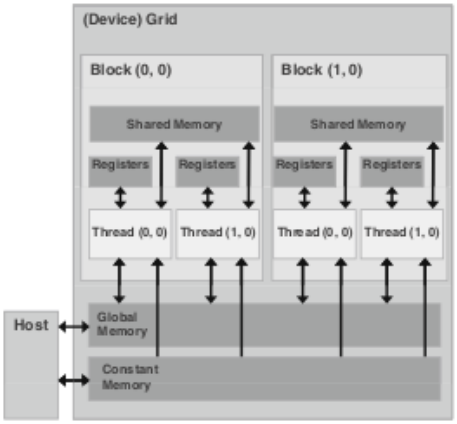
2.2.1 CUDA et les cartes graphiques NVIDIA

Tout d'abord, il est nécessaire de rappeler le sujet du stage : le calcul parallèle, plus précisément sur GPU. Pour expliquer les différents choix implés ici, il est nécessaire de faire quelque petits rappels.

Le CPU (Central Processing Unit ou processeur) est le cœur de l'ordinateur, c'est lui qui exécute toutes les commandes et calculs requis par l'ordinateur. Avec l'évolution des programmes informatiques, la puissance de processeur cœur a dû augmenter, jusqu'en atteindre une limite physique en 2003, qui induisit la création de processeur multicœur pour augmenter les performances de notre matériel informatique. Mais le principal problème est qu'un programme séquentiel ne va s'exécuter que sur un seul coeur du processeur, ce qui ne permettrai pas d'obtenir de gain efficace, d'où les débuts de la programmation parallèle.

Dans le même temps, les cartes graphiques (qui sont accessible au grand public depuis 1981) ont été dévelopé : principalement utilisées pour le rendu d'image en 2D/3D, il a été rapidement découvert que, grâce à leur structure si particulière (elle possède un grand nombre de cœur de calcul simple au détriment de la complexité des instructions et de la mémoire fourni), il est possible d'exécuter de multiples calculs simultanément, gagnant un temps de calculs dépendant de la taille des données (plus il y a de données, plus le gain est important) si l'on compare à un cœur de CPU. Donc les principales entreprises de production de cartes graphiques (NVidia et AMD) ont dévellopé leur propre language de programmation parallèle, adapté des languages de programmation existants, tel que le C et le C++ (il s'agît d'extension des languages).

FIGURE 1 – Structure d'un GPU



Pour en revenir avec le sujet du stage, nous allons donc faire du calcul parallèle et ce, sur le GPU. Étant donné que les cartes graphiques du laboratoire sont des cartes graphiques NVidia, il a été choisi d'utiliser le language CUDA, dévellopé par NVidia exclusivement pour leur produit. À noter qu'il aurait tout aussi bien pu être possible d'utiliser un autre language.

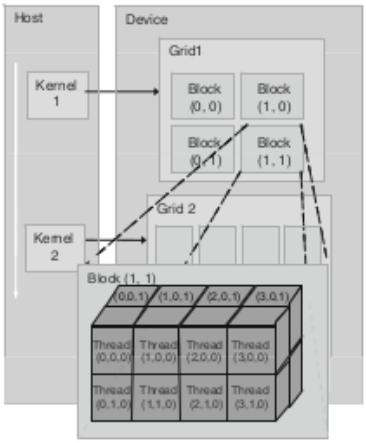
Pour bien utiliser comprendre le CUDA (à partir de maintenant, le CUDA sera considéré comme une extension du C, car nous avons travaillé avec celle originaire du C, mais il existe des version C++,Python,...), il me parait nécessaire de comprendre comment est formée une carte graphique. Dans la figure 1 est décrit la structure d'un GPU, la carte graphique étant composée :

- d'une grille (Grid) qui comprend tout les éléments du GPU ;
- de 2 DRAM (Dynamic Random Access Memory), qui stocke les données modifiables pour l'une ou constantes pour l'autre, accessible par l'hôte (i.e. le CPU, et seulement par transfert) et par tout les autres composants du GPU ;
- de multiples Blocks (en quantités dépendant du GPU) qui rassemble :
  + d'une mémoire cache (Shared Memory) et d'une mémoire texture (non affichée sur le schéma, de très faible capacitè et principalement pour les graphiques) accessible par tout les composants du Block ;
  + de multiples cœurs de calculs(Thread, en quantités dépendant du GPU) de calculs ;
  + de 2 mémoires locales (Registered et Local) propres à chaque Thread qui permet le stockage de tableau de données dans la mèmoire Local et les autres données dans la mémoire Registered.

De plus, le GPU ne peux pas acceder à des données stockées sur les disque durs, la RAM et les mémoires internes du CPU, tandis que le CPU ne peux que faire des échanges de données entre la RAM (de lui-même) et la DRAM (du GPU).
D'où l'introduction de nouvelles fonctions, liées au concept de la parallèlisation des calculs :

- l'équivalent du malloc() et du free sont cudaMalloc et cudaFree, pour allouer sur la Global Memory ;
- pour effectuer des calculs sur le GPU, il est nécessaire de créer des fonctions répondant à la norme : *parametre type fonction(variable)* où le paramètre est un identifiant que indique si l'appel se fait du CPU vers le GPU (*__global__*) ou du GPU vers le GPU (*__device__*), de type au choix sauf si le parametre __global__ est défini et le type *void* est alors obligatoire. L'appel de cette même fonction se fait de la forme *fonction<<<dimGrid,dimBlock>>>(variable)*, où dimGrid et dimBlock sont les dimensions de la grille et de chaque block, que l'on défini et limité à une dimension dépendant de la carte graphique ;
- dans le corps d'une fonction du GPU, chaque thread executera sa propre version de la fonction, et toute variable interne à la fonction sera propre à chaque thread (sauf si elle est déclarée *__shared__*) si qui implique que l'on peux avoir à faire appel à des variables tels que threadIdx.x (ou .y/.z) pour connaitre l'emplacement du thread dans son block (voir figure 2 pour un exemple) ;
- la compilation se fait avec nvcc (il est possible que la compilation est des erreurs sans réponses, pour les corriger [si il ne s'agit pas d'erreur de votre part] il suffit de renommer vos fichiers *.c en *.cpp car le compilateur nvcc est plus adapté au C++ qu'au C).

FIGURE 2 – Exemple d'une grille défini par une fonction

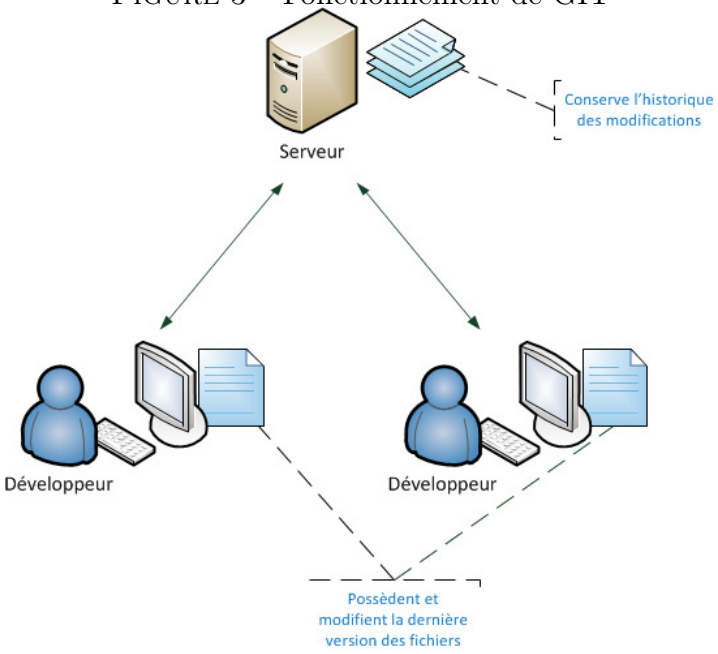


2.2.2 GIT

Selon `https://git-scm.com/book/fr/v1/D%C3%A9marrage-rapide-Une-rapide-histoire-de-Git` :
Comme de nombreuses choses extraordinaires de la vie, Git est né avec une dose de destruction créative et de controverse houleuse. Le noyau Linux est un projet libre de grande envergure. Pour la plus grande partie de sa vie (1991–2002), les modifications étaient transmises sous forme de patchs et d'archives de fichiers. En 2002, le projet du noyau Linux commença à utiliser un DVCS propriétaire appelé BitKeeper.
En 2005, les relations entre la communauté développant le noyau Linux et la société en charge du développement de BitKeeper furent rompues, et le statut de gratuité de l'outil fut révoqué. Cela poussa la communauté du développement de Linux (et plus particulièrement Linus Torvalds, le créateur de Linux) à développer son propre outil en se basant sur les leçons apprises lors de l'utilisation de BitKeeper. Certains des objectifs du nouveau système étaient les suivants :

- *vitesse ;*
- *conception simple ;*
- *support pour les développements non linéaires (milliers de branches parallèles) ;*
- *complètement distribué ;*
- *capacité à gérer efficacement des projets d'envergure tels que le noyau Linux (vitesse et compacité des données).*

FIGURE 3 – Fonctionnement de GIT



Ainsi, GIT est plus un gestionnaire de version en ligne, centralisé pour permettre un accès depuis plusieurs appareils. Ce service fonctionne sous le principe suivant (voir figure 3) : un serveur possède chaque version du projet, et permet à

chaque client de posséder une version commune à tous du projet, qu'il modifiera de manière locale avant de la retransmettre au serveur, en prenant garde de régler tout conflit qui pourrait lui être signaler (dû à la mise à jour du projet par un autre client avant vous), ce qui permet d'avoir un suivi complet du projet par tous les développeur (client) de pouvoir travailler en collaboration sans problème de gestion. À noter qu'il est possible de multiplier ou diviser les branches de développement, pour permettre de partir dans plusieurs direction de recherche sans compromettre le travail exécuter précédemment. Ce qui en fait un des gestionnaire de version les plus puissants existants.
De fait, il existe des dépôts en ligne tel que GitHub.com qui permettent d'obtenir gratuitement un "serveur" pour centraliser ses données gratuitement (mais les donné]es sont accessibles et non modifiables par tous) ou en payant pour un "serveur" privé. Par exemple, ceci est mon GIT pour ce stage : `https://github.com/mipicard/Stage_parrallel_programming`

2.2.3 Abstraction et factorisation du code

Ici, nous allons expliquer quelques méthodes de travail nécessaire en laboratoire (surement aussi en entreprise) de programmation :

- **l'abstraction** : lorsque l'on créé un projet, il est plus utile de crée ses propres types (en renommant un type existant ou en construisant le sien) car il est ainsi possible de changer les composants de ce type sans devoir changer tout ce qui utilise ce type et ses fonctions (sauf possiblement les fonctions dédiées ce type, bien entendu) ce qui permet de mettre à jour bien plus facilement ses projets sans devoir changer tout son code. De plus, il est utile de séparer son code en de multiple petit fichier, par exemple en créant un fichier pour chaque type, ou par action : le choix est à la discrétion du développeur.

- **la factorisation** : il s'agit d'un effet à éviter lors du développement qui consiste a répéter inutilement des parties de codes. Il est donc nécessaire de rester attentif lorsque l'on code, et si l'on répète du code, de vérifier que si c'est obligatoire (autre type) ou si l'on pourrait faire mieux sans répétition.

2.2.4 R et Rmarkdown

Selon `https://fr.wikipedia.org/wiki/R_(langage_de_programmation_et_environnement_statistique)` :
R est un logiciel libre de traitement des données et d'analyse statistiques mettant en œuvre le langage de programmation S. C'est un projet GNU fondé sur l'environnement développé dans les laboratoires Bell par John Chambers et ses collègues. Depuis plusieurs années, deux nouvelles versions apparaissent au printemps et à l'automne. Il dispose de nombreuses fonctions graphiques.
Le logiciel R est considéré par ses créateurs comme étant une exécution de S, avec la sémantique dérivée du langage Scheme. C'est un logiciel libre distribué selon les termes de la licence GNU GPL et disponible sous GNU/Linux, FreeBSD, NetBSD, OpenBSD, Mac OS X et Windows.
Une enquête menée par Rexer Analytics auprès de 1 300 analystes retrouve que R est le logiciel le plus souvent utilisé lorsqu'il s'agit d'un travail en entreprise, dans le monde académique, au sein d'organismes publics ou d'ONG et chez les analystes travaillant comme consultants.

R est ainsi un language très utilisé par les chercheurs en informatique pour leur permettre d'obtenir de manière rapide et optimisée des courbes et des résultats leur permettant de répondre à leur sujet de recherche.
R est très souvent utilisé en Rmarkdown, i.e en script convertible en page web ou pdf, pour permettre de rassembler de manière efficace le sujet et les résultats d'expériences (voir figure 4 pour un exemple tiré de mon propre fichier rmd).

FIGURE 4 – Exemple de code Rmarkdown

2.2.5 LaTeX

Selon `http://www.grappa.univ-lille3.fr/FAQ-LaTeX/1.1.html` :
TeX (1978) est le formateur de texte de D. E. Knuth. À l'origine, Knuth a développé TeX notamment pour réaliser de beaux documents et écrire des formules mathématiques.
LaTeX, écrit par L. Lamport (1982), est un jeu de macros au-dessus de TeX, plus facile à utiliser que ce dernier. Il propose notamment différents styles de document auxquels correspondent des classes de document et une grande diversité de macros qui répondent à divers besoins des auteurs. LaTeX a été conçu pour rédiger des articles, des rapports, des thèses ou des livres ou pour préparer des transparents. On peut insérer dans le texte, des dessins, des tableaux, des formules mathématiques et des images sans avoir à se soucier (ou presque) de leur mise en page. Les documents produits avec LaTeX et TeX sont d'une excellente qualité typographique.

LaTeXest donc utilisé pour de la mise en forme de texte : il se différencie de OpenOffice et Microsoft Word par sa compilation, qui permet un meilleur rendu (puisque l'utilisation d'algorithme de traitement plus lourd et efficace) que les autres (qui utilise des algorithmes léger pour un rendu graphique en temps réel). Ainsi, il est nécessaire d'écrire dans le texte les spécification graphique comme l'inclusion d'image, la taille, la couleur et le style du texte, etc (voir figure 5 pour un exemple tirée de notre fichier pour obtenir ce rapport).

FIGURE 5 – Exemple de code LaTeX

```
>{}>(variable)}, o\`u dimGrid et dimBlock sont les dimensions de la grille et de chaque block, que l'on defini et limit\'e \ a une
dimension d\'e{}pendant de la carte graphique ;
        \item dans le corps d'une fonction du GPU, chaque thread executera sa propre version de la fonction, et toute variable interne \`a
la fonction sera propre \`a chaque thread (sauf si elle est d\'e{}clar\'e{}e \textit{__shared__}) si qui implique que l'on peux avoir \`a
faire appel \`a des variables tels que threadIdx.x (ou .y/.z) pour connaitre l'emplacement du thread dans son block (voir figure~\ref
{fig:structGrid} pour un exemple).
        \item la compilation se fait avec nvcc (il est possible que la compilation est des erreurs sans r\'e{}ponses, pour les corriger [si
il ne s'agit pas d'erreur de votre part] il suffit de renommer vos fichiers *.c en *.cpp car le compilateur nvcc est plus adapt\'e au C++
qu'au C).
    \end{itemize}
    \begin{figure}
        \centering
        \caption{Exemple d'une grille d\'e{}fini par une fonction}
        \label{fig:structGrid}
        \includegraphics[scale=0.65]{Structure_grille_GPU.eps}
    \end{figure}
    \subsubsection{GIT}
    \subsubsection{Abstraction, g\'e{}n\'e{}ralisation et factorisation du code}
    \indent Ici, nous allons expliquer quelques m\'e{}thodes de travail n\'e{}cessaire en laboratoire (surement aussi en entreprise) de
programmation :
    \begin{itemize}
        \item \textbf{la g\'e{}n\'e{}ralisation} :
        \item \textbf{l'abstraction} :
        \item \textbf{la factorisation} :
    \end{itemize}
    \subsubsection{R et Rmarkdown}
    \subsubsection{\LaTeX}
    \indent \LaTeX est utilis\'e pour de la mise en forme de texte : il se diff\'erencie de OpenOffice et Microsoft Word par sa
compilation, qui permet un meilleur rendu (puisque l'utilisation d'algorithme de traitement plus lourd et efficace) que les autres (qui
utilise des algorithmes l\'e{}ger pour un rendu graphique en temps r\'e{}el). Ainsi, il est n\'e{}cessaire d'\'e{}crire dans le texte les sp
\'e{}cification graphique comme l'inclusion d'image, la taille, la couleur et le style du texte, etc.
    \subsection{Pourquoi ce stage?}
    \indent J'ai choisi ce stage en regardant r\'e{}gulièrement sur le site du DLST, o\`u j'avais remarqu\'e ce stage propos\'e pour l'ann
\'e{}e derni\`ere et qui m'avait tr\`es int\'e{}ress\'e : \'e{}tant un grand joueur sur PC, je poss\`ede et j'exploite ma carte graphique
pour les jeux et je voulais savoir ce que je pouvais faire d'autre avec, ainsi que l'id\'e{}e de pouvoir r\'e{}ellement faire des ex\'e{}
cutions parall\`eles au lieu de toujours le simul\'e en encha\^inant tr\`es rapidement des instructions sensées être parallèles.
```

2.3 Pourquoi ce stage ?

J'ai choisi ce stage en regardant régulièrement sur le site du DLST, où j'avais remarqué ce stage proposé pour l'année dernière et qui m'avait très intéressé : étant un grand joueur sur PC, je possède et j'exploite ma carte graphique pour les jeux et je voulais savoir ce que je pouvais faire d'autre avec, ainsi que l'idée de pouvoir réellement faire des exécutions parallèles au lieu de toujours le simulé en enchaînant très rapidement des instructions sensées être parallèles.

3 Mon parcours

Le but de ce stage était de découvrir la programmation parallèles sur GPU, en utilisant CUDA pour les cartes graphiques NVidia. Pour ce faire, nous avons utilisé l'exemple de la multiplication matricielle, qui permet de montrer efficacement les différences de vitesse d'exécution entre des algos parallèlisés ou non.

3.1 L'avancement chronologique

Durant la première semaine de stage, nous avons dû suivre le programme de ce livre *Programming massively parallel processors* (disponible ici *http://www.hds.bme.hu/~fhegedus/C++/programming_massively_parallel_processors.pdf*) . Ainsi, nous avons suivi le tutoriel en apprenant le language et les exemples, qui seront aussi nos algorithmes de calcul durant tout notre stage, le tout écrit dans un seul et même fichier. Durant cette même semaine, nous avons aussi découvert et commencé à utilisé GIT, ce qui nous a permis de garder des traces de notre évolution plus facilement (lien de notre GitHub : *https://github.com/mipicard/Stage_parrallel_programming*).
Liste des algorithmes : voir en Annexe, les algorithmes CPU, GPU1 et GPU2.

Durant la deuxième semaine, suite au conseil de Mr HUARD, nous avons abstrait tout notre code, en séparant notre code en petits fichiers :

- un fichier Element.cpp, qui décrit l'élément constituant de nos matrices, ainsi que les foncions associées ;
- un fichier MatriceCPU.cpp et un fichier MatriceGPU.cu, qui décrivent chacun un type de matrice en fonction du cœur de calcul utilisé, dont chaque case de la matrice est défini par le type Element ;
- un fichier Matrice.cu, qui permet de faire la correspondance entre les deux type de matrice définit avant, permettant de ainsi de centraliser les appels au matrice ;
- un fichier Timer.cpp, qui fournit des méthodes de mesures du temps, à la microseconde près ;
- un fichier main.cu, utilisant tout ce qui a été définit précedemment, pour effectuer les calculs, pouvant changer les paramètres d'exécution du programmes via la ligne de commande ;
- un makefile, qui définit les règles de compilation du projet ;
- des fichiers bash, pour automatiser les tests de vitesses.

Le code complet est disponible ici : *https://github.com/mipicard/Stage_parrallel_programming/blob/master/Travail/Matrices.tgz*.
De plus, nous avons commencé à étudier le language R et Rmarkdown pour nous donner une première idé du fonctionnement de cet outil.

Durant la troisième semaine, nous avons développé un dernier algorithme GPU qui aurait dû nous permettre d'obtenir de meilleur rapidité d'exécution que ce produit précedemment et continué d'étudié le language R, pour utiliser les résultats d'expériences produit par les 3 algorithmes précédents.

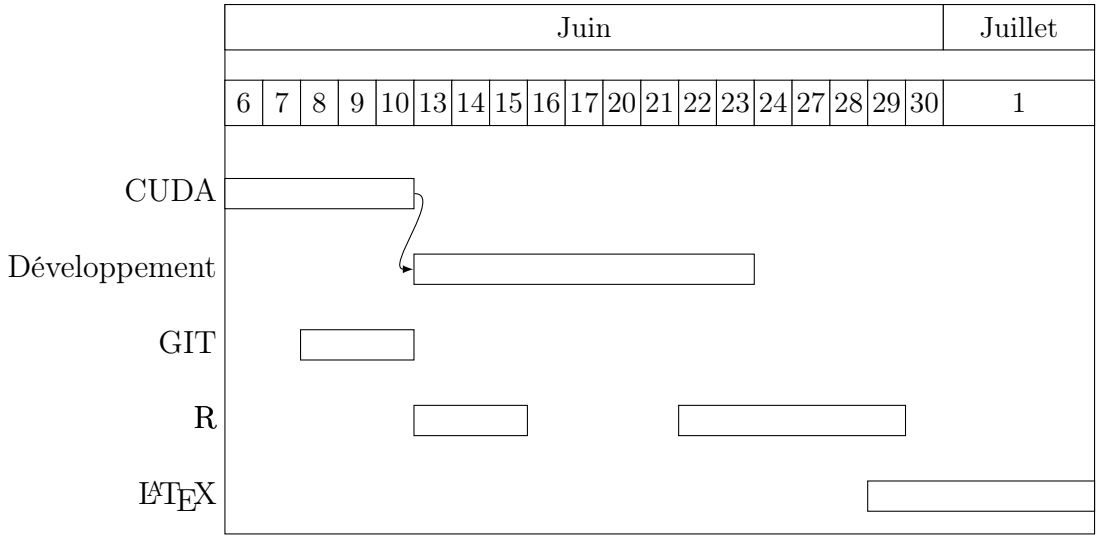
Durant la quatrième et dernière semaine, nous avons fini d'étoffer notre rapport d'expérience, qui est disponible en annexe, et débuter ce rapport de stage.

3.2 L'apprentissage

Durant ce stage, nous avons été confronté à différentes sous-problématiques et difficultés :

- **Les transferts mémoires** : lors d'opérations sur des cœurs de calculs différents, la mémoire utilisé est différente, ce qui implique un temps de copie entre les différentes mémoires, et donc une légère baisse de performance induite.
- **La mesure du temps** : il est nécessaire de séparer les temps d'allocations/transferts mémoires des temps de calculs, et donc de faire des mesures séparées des allocations/transferts et des opérations effectuées.
- **Les données (matrices)** : il est nécessaire de prendre en compte la taille mémoire de la matrice, qui doit être multiple de la taille d'un bloc mémoire sur le cœur de calcul utilisé. Par exemple, pour optimisé le calcul sur CPU, il faut que la matrice soit de taille multiple d'une ligne de cache et qu'elle soit toujours au début de cette ligne, tandis que sur le GPU étudié les memoires shaders de nos blocks permettaient d'utiliser 2 tableaux temporaires de taille 64*64 donc il est utiles d'utiliser des matrice de dimensions multiple de 64 (ou du moins des matrices utilisant un emplacement de cette taille même si elle est de taille inférieur). Il s'agit d'une notion qui donna lieu à un incompris jusqu'à l'écriture de ce rapport entre Mr HUARD et moi, car je n'avais pas saisi que je devais utilisé des matrices respectant ces critères pours les GPU, ce que je n'ai pas fait e qui a occasioné de cruelles pertes de performances pour le dernier algorithme développé, alors que des gains auraient dû être observé.

4 Conclusion



Pendant ce mois de stage à l'INRIA, j'ai pu avoir un aperçu de la vie en laboratoire de recherche et du métier de chercheur en découvrant par la même la programmation parallèles et tout les contraintes qui y sont liées.

J'ai aussi découvert moult nouvelles notions, languages et méthodes de travail qui me seront utiles dans la suite de mes études et surement dans mon futur métier.

En conclusion, ce stage à été plus qu'une bénédiction pour moi, dans un cadre de stage excellent.

Annexe : Rapport d'expérience

Comparaison de l'exécution des algorithmes naïfs de multiplication de matrice sur CPU et GPU

Picard Michaël

Le but de cette expérience est de montrer les différences de performances entre la multiplication sur CPU et sur GPU de matrices, en utilisant les algorithmes naïfs de calculs. Ceci dans le but de débuter en programmation parrallèle et d'expérimenter techniques et utilisation du language CUDA.

Nous allons tester le temps d'exécution de 4 algorithmes :

- le 1er en CPU naïf
- le 2eme en GPU naïf, avec division en blocks
- le 3eme en GPU avec mémoire shader,i.e la mémoire locale à chaque block du GPU, et avec division en blocks
- le 4ème est une modification du 3eme pour augmenter le nombre de calcul exécuté par thread à 4, en posant une taille de blocks et de variable locales fixes

Tout ces tests seront effectué avec des matrices alloué localement au centre de calcul (CPU ou GPU).

Nous nous attendons à des performances qui décroisse exponentiellement avec la taille de la matrice pour le 1er algorithme (CPU), et des performances à peu près stables pour les 2 algorithmes sur GPU, avec un léger gain de puissance pour le 3eme algorithme, ainsi qu'à une possible amélioration supplémentaire pour le dernier, avec un doute sur son efficacité dû aux transferts mémoires.

I Algorithme

CPU : Algorithme naïf exécuté sur le CPU, de la même manière qu'un humain le ferait.

```
void multiplicationMatriceCPU(const MatriceCPU *m1,const MatriceCPU *m2,
     MatriceCPU *resultat){
    Element tmp;
    for(int i=0;i<m1->dimension;i++){
        for(int j=0;j<m1->dimension;j++){
            tmp=ZERO_ELEMENT;
            for(int k=0;k<m1->dimension;k++)
                tmp=additionElement(tmp,multiplicationElement(
                  m1->matrice[i*m1->dimension+k],
                  m2->matrice[k*m1->dimension +j]));
            resultat->matrice[positionElement(i,j,resultat)]=tmp;
        }
    }
}
```

N.B (pour les algorithmes GPUx):

- divMaxDim renvoie la plus grand diviseur (<=32) de la dimension envoyé
- dimSup renvoie la dimension supérieur multiplie de 64

GPU1 : Algorithme naïf, qui divise la matrice en block carrés de même dimension et qui calcule de manière humaine chaque case de la matrice

```
__global__ static void multiplicationMatriceGPU_Kernel(const MatriceGPU m1,
                        const MatriceGPU m2,MatriceGPU resultat,
                        const int nbThreadPerBlock){

    // On détermine les coordonnées de la case à calculer
    unsigned long ligne= blockIdx.y*nbThreadPerBlock + threadIdx.y,
                  colonne= blockIdx.x*nbThreadPerBlock + threadIdx.x;
    Element sum = ZERO_ELEMENT;

    for(int k=0;k<resultat.dimension;k++)
        sum = additionElement(sum,multiplicationElement(
          m1.matrice[ligne*m1.dimension+k],
          m2.matrice[k*m2.dimension+colonne]));

    // On affecte le résultat à sa case (ordonné par ligne et colonne)
    resultat.matrice[positionElement(ligne,colonne,&resultat)] = sum;
}

void multiplicationMatriceGPU(const MatriceGPU *m1,const MatriceGPU *m2,
     MatriceGPU *resultat){
    //Matrice de dimension dimension*dimension
    const unsigned long dim=resultat->dimension;
    //dimension d'un block = div *div <= 32*32
    int div = divMaxDim(dim);
    //On sépare notre matrice en divG*divG block
    int divG = dim/div;
    dim3 dimBlock(div,div,1),dimGrid(divG,divG,1);
    // Appel du Kernel
    multiplicationMatriceGPU_Kernel<<<dimGrid,dimBlock>>>(*m1,*m2,*resultat,div);
    // On attend que tous les calculs soit terminés
    cudaDeviceSynchronize();
```

GPU2 : Amélioration de l'algorithme GPU1 en stockant temporairement des éléments de la matrice dans le but de réduire les appels à la mémoire globale GPU

```
__global__ static void multiplicationMatriceGPU_Kernel(const MatriceGPU m1,
                        const MatriceGPU m2,MatriceGPU resultat,
                        const int nbThreadPerBlock){
    Element sum = ZERO_ELEMENT;
    // On utilise la mémoire locale, en utilisant le fait que la
    // matrice est découpé en block de dimension <= 32*32
    __shared__ Element Mgshader[32][32];
    __shared__ Element Ngshader[32][32];

    // Index des block et thread
    int bx=blockIdx.x,by=blockIdx.x,tx=threadIdx.x,ty=threadIdx.y;
    // On détermine les coordonnées de la case à calculer
    int ligne = by*nbThreadPerBlock+ty, colonne = bx*nbThreadPerBlock+tx;
    unsigned long Width = resultat.dimension;

    for(int s=0;s<(Width/nbThreadPerBlock);s++)
    {
      // On remplit les tableaux temporaires locaux
```

```
        Mgshader[ty][tx]=m1.matrice[ligne*Width+(s*nbThreadPerBlock + tx)];
        Ngshader[ty][tx]=m2.matrice[colonne+Width*(s*nbThreadPerBlock + ty)];
        __syncthreads(); // On attend que les tableaux soient remplis

        for(int k=0;k<nbThreadPerBlock;k++){ // on calcule la valeur temporaire
            sum=additionElement(sum,multiplicationElement(
              Mgshader[ty][k],Ngshader[k][tx]));
        }
        __syncthreads(); // On attend la fin de tous les calculs
    }
    // On affecte le résultat à sa case (ordonné par ligne et colonne)
    resultat.matrice[ligne*Width+colonne] = sum;
}

void multiplicationMatriceGPU(const MatriceGPU *m1,const MatriceGPU *m2,
     MatriceGPU *resultat){
    //Matrice de dimension dimension*dimension
    const unsigned long dim=resultat->dimension;
    //dimension d'un block = div *div <= 32*32
    int div = divMaxDim(dim);
    //On sépare notre matrice en divG*divG block
    int divG = dim/div;
    dim3 dimBlock(div,div,1),dimGrid(divG,divG,1);
    // Appel du Kernel
    multiplicationMatriceGPU_Kernel<<<dimGrid,dimBlock>>>(*m1,*m2,*resultat,div);
    // On attend que tous les calculs soit terminés
    cudaDeviceSynchronize();
```

GPU3 : Amélioration de l'algorithme GPU2 en fixant comme paramètre que chaque coté des matrices en entrées est multiple de 64, que les tableaux locaux sont de taille 64*64 toujours rempli et que les blocks sont tous de taille 64*16

```
__global__ static void multiplicationMatriceGPU_Kernel(const MatriceGPU m1,
                        const MatriceGPU m2,MatriceGPU resultat){
    Element sum[4];
    // Tableau des 4 variables locales correspondant aux 4 cases de la
    // matrices que l'on calcule par thread
    for(int i=0;i<4;i++)
        sum[i]=ZERO_ELEMENT;

    // On crée les tableaux temporaires locaux de dimension
    // 4 fois (64*64) celle d'un block (64*16)
    __shared__ Element Mgshader[64][64];
    __shared__ Element Ngshader[64][64];

    // Index des blocks et threads
    int bx=blockIdx.x,by=blockIdx.x,tx=threadIdx.x,ty=threadIdx.y;
    for(int s=0;s<gridDim.x;s++){ //On remplit les tableaux temporaires locaux
        for(int i=0;i<4;i++){//
            Mgshader[ty+i*16][tx]=m1.matrice[resultat.dimension*(s*64+ty+i*16)+bx*64+tx];
            Mgshader[ty+i*16][tx]=m2.matrice[resultat.dimension*(by*64+ty+i*16)+s*64+tx];
        }
        __syncthreads(); // On attend que les tableaux soient remplis
```

```
        for(int i=0;i<4;i++){ // Pour chaque case,
            for(int k=0;k<64;k++) // on calcule la valeur temporaire
                sum[i]=additionElement(sum[i],multiplicationElement(
                  Mgshader[ty+i*16][k],Ngshader[k][tx+i*16]));
        }
        __syncthreads(); // On attend la fin de tous les calculs
    }
    // On affecte chaque résultat à sa case (ordonné par les blocks et les threads)
    for(int i=0;i<4;i++)
        resultat.matrice[resultat.dimension*(64*by+ty+i*16)+bx*64+tx]=sum[i];
    __syncthreads();
}

void multiplicationMatriceGPU(const MatriceGPU *m1,const MatriceGPU *m2,
     MatriceGPU *resultat){
    //Matrice de dimension dimension*dimension
    const unsigned long dim=resultat->dimension;
    unsigned long dimSup=dimSUP(dim); //dimSup = X*64 >=dim
    // On divise la matrice de dimension dimSup en nbBlock*nbBlock block
    unsigned long nbBlock=dimSup>>6;
    dim3 dimBlock(64,16,1),dimGrid(nbBlock,nbBlock,1);
        // On copie les matrices entrées dans des matrices
        // de dimension supérieures ou égales
        MatriceGPU *m1bis=initialiserMatriceGPU(dimSup);
        MatriceGPU *m2bis=initialiserMatriceGPU(dimSup);
        cpMatriceDimDiff(m1,m1bis);cpMatriceDimDiff(m2,m2bis);
        MatriceGPU *resultatbis=initialiserMatriceGPU(dimSup);
    // Appel du Kernel
    multiplicationMatriceGPU_Kernel<<<dimGrid,dimBlock>>>(*m1bis,*m2bis,*resultatbis);
    cudaDeviceSynchronize(); // On attend que tous les calculs soit terminés
        cpMatriceDimDiff(resultatbis,resultat); // on récupére le bon résultat
        freeMatriceGPU(m1bis);freeMatriceGPU(m2bis);freeMatriceGPU(resultatbis);
}
```

II Matériel

Description du GPU (via deviceQuery) :

```
  Quadro 600
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 1023 MBytes (1072889856 bytes)
  ( 2) Multiprocessors, ( 48) CUDA Cores/MP:     96 CUDA Cores
  GPU Max Clock rate:                            1280 MHz (1.28 GHz)
  Memory Clock rate:                             800 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 131072 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
```

```
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
```

Description du système (via *lstopo*) :

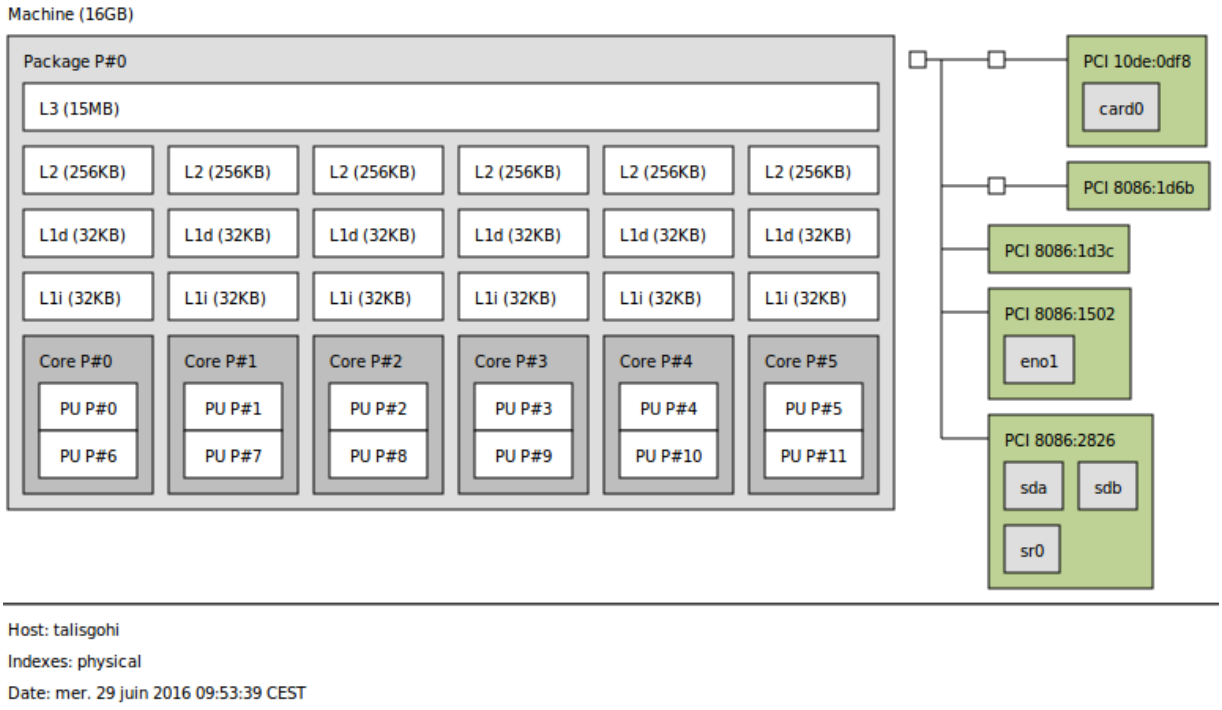


Figure 1: *Description du système*

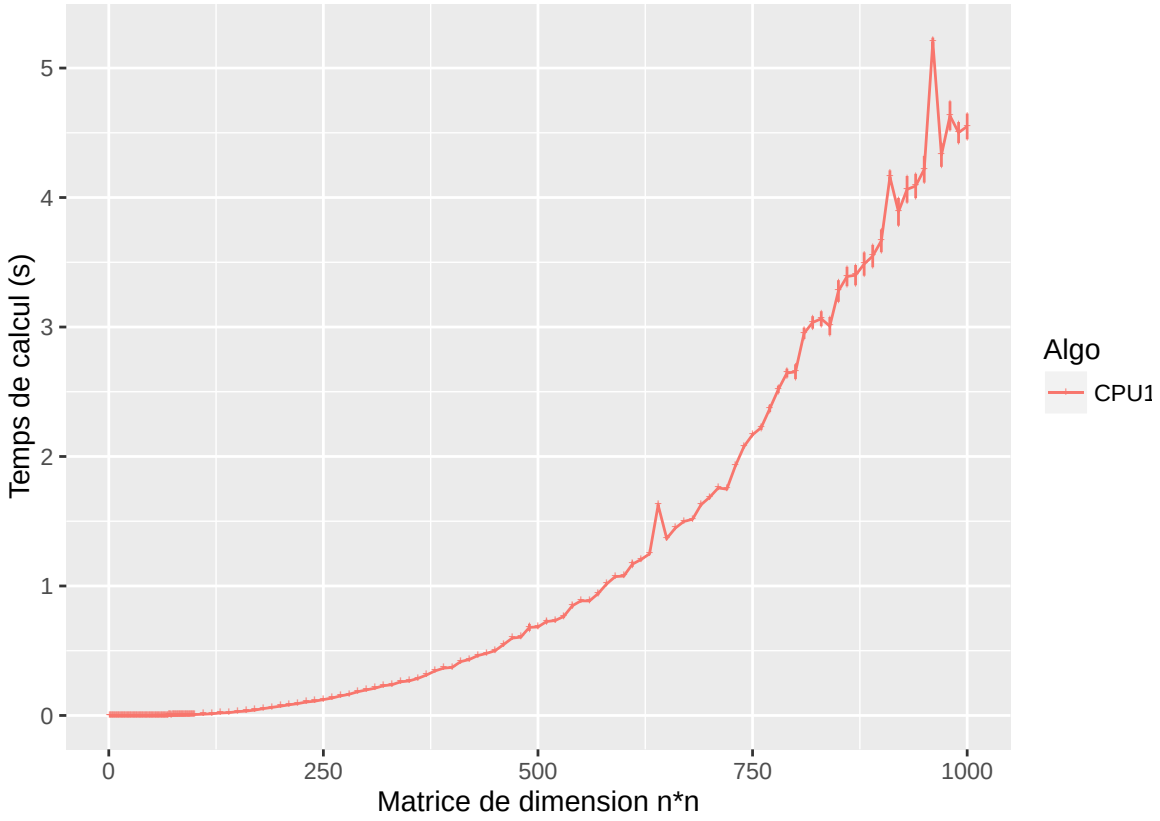
III Resultats d'expériences

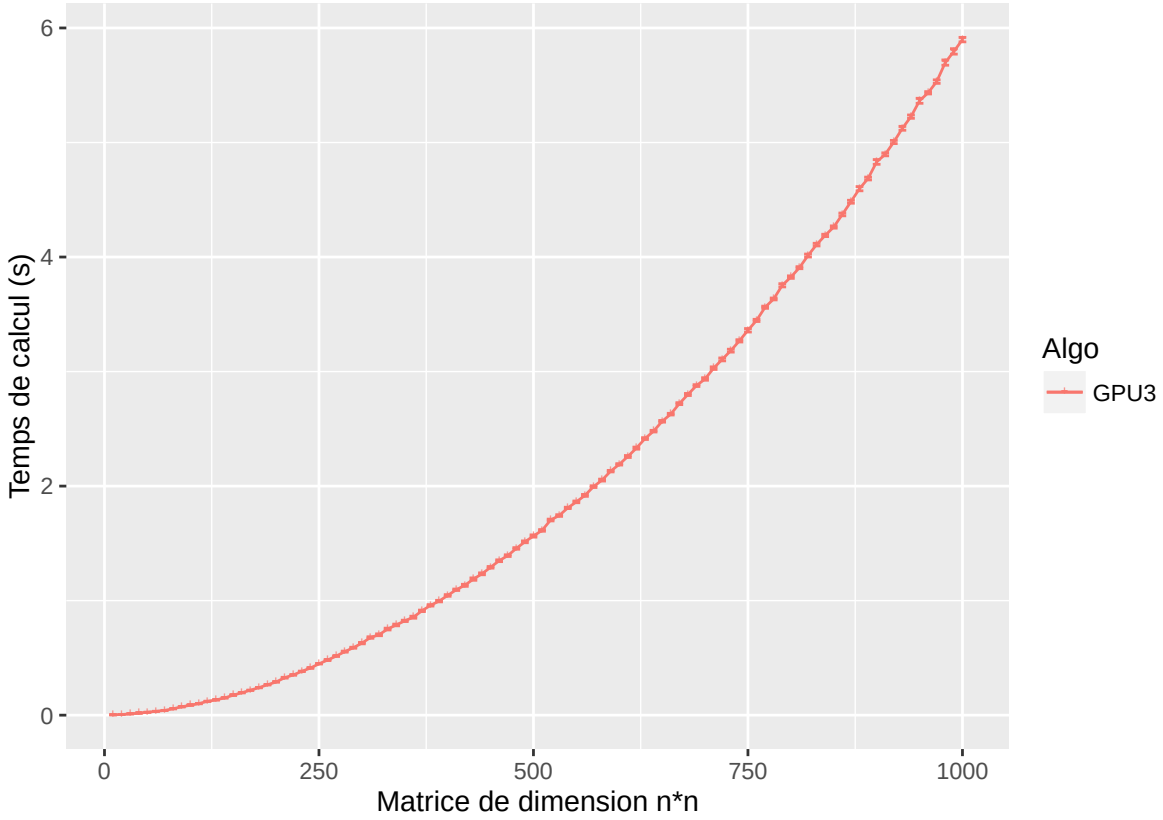
Nous avons choisi de séparer certain affichage :

- L'algorithme CPU est séparé des algorithmes GPU : il s'agit d'un choix nécessaire, car il est à la fois logique de séparé les coeurs de calculs et nécessaire de la faire pour une meilleur lisibilité des résultats.
- Les algorithmes GPU seront séparés entre eux s'il est nécessaire de le faire par soucis de lisibilité

Sur chacun des graphiques présentés ci-dessous, un point correspond à une moyenne de 32 run et les barres d'erreurs sont affichées.

1 Temps d'exécution

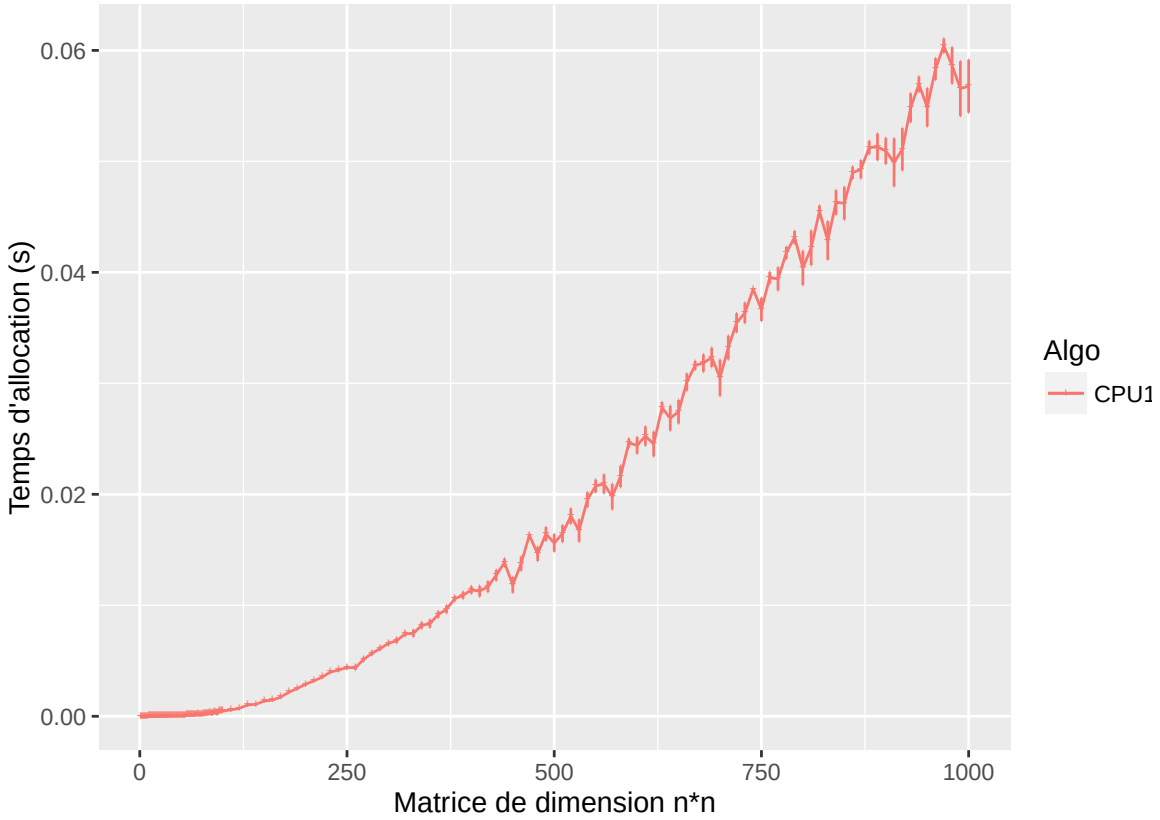




On remarque ici que, comme attendu, l'algorithme CPU en dimensions élevées de matrice (clairement lisible

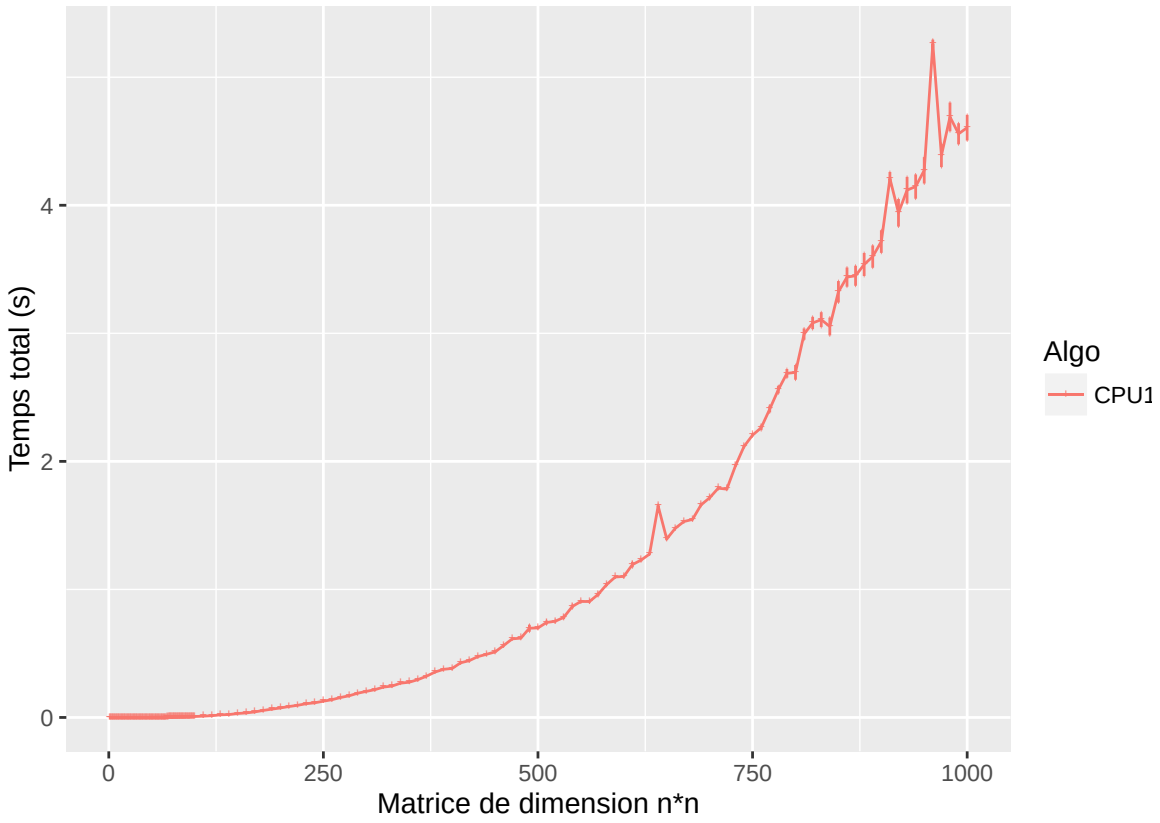
dès la dimension 500*500) est plus lent que les algorithmes GPU1 et GPU2, le GPU2 étant plus rapide que le GPU1. Fait surprenant, l'algorithme GPU3 que l'on espérait plus efficace que les autres est très rapidement le plus lent (très clairement visible dès la dimension 500*500).

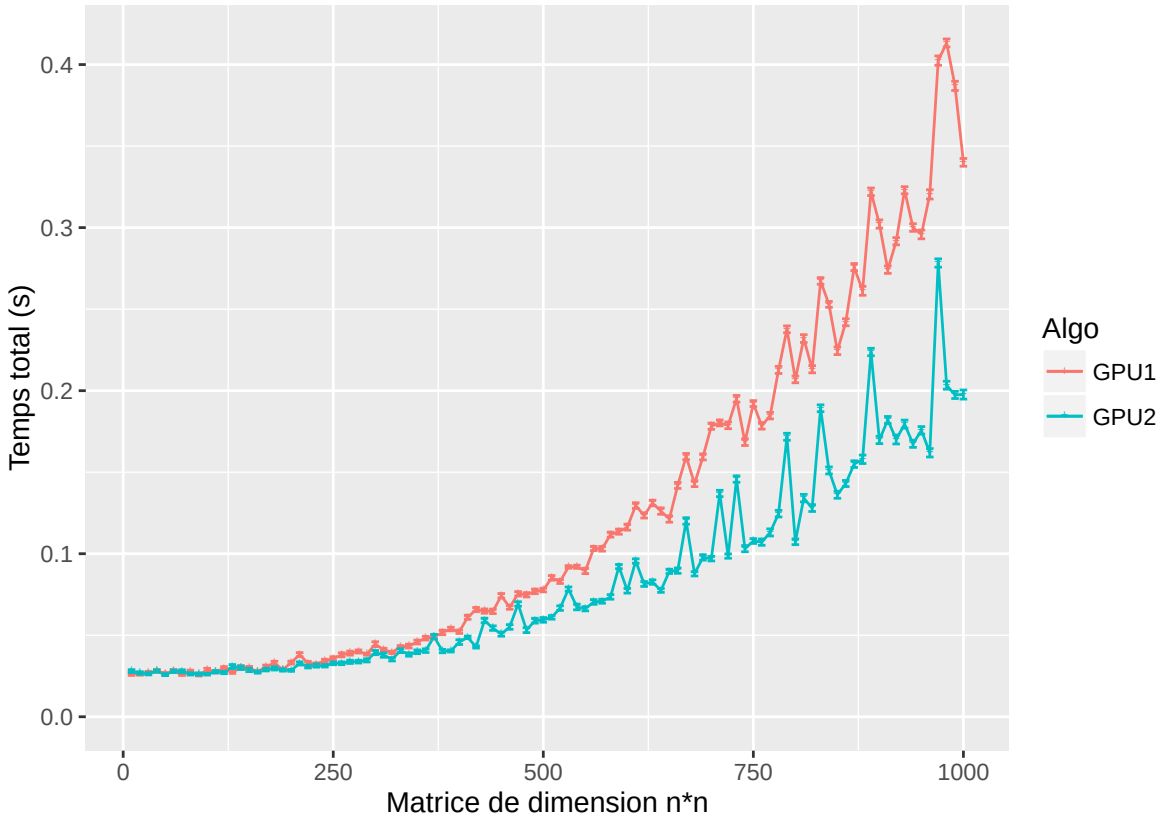
2 Temps d'allocation

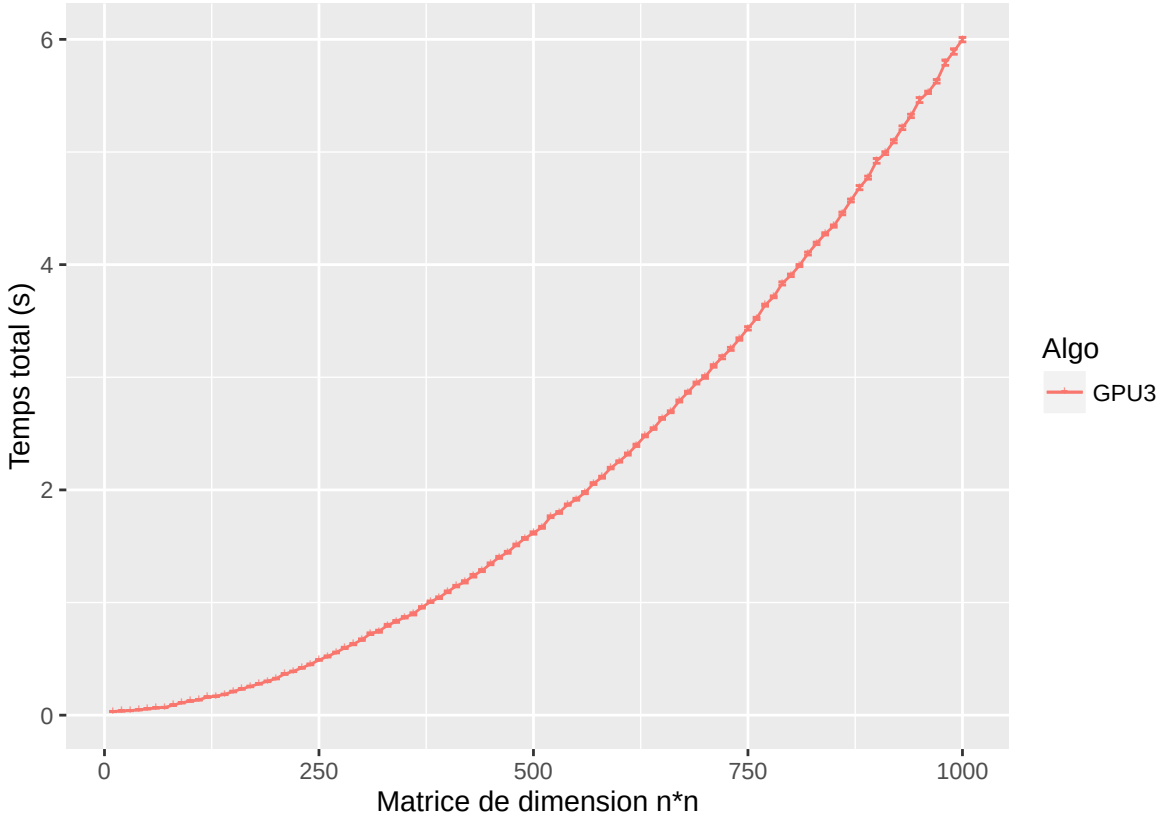


Comme attendu sur la théorie de l'allocation mémoire du CPU et du GPU, le temps d'allocation des matrices sur le GPU est clairement supérieur à celui du CPU. Un autre fait surprenant est que le temps d'allocation des matrices pour l'algorithme GPU3 est supérieur au 2 autres algorithmes GPU, alors qu'il s'agit d'une allocation strictement identique.

3 Temps total







Ici, nous avons rassemblé les temps d'allocations et de calculs de chaque algorithme : nous avons donc les

mêmes observation que pour les temps d'exécutions, car les temps d'allocations sont négligeables à partir d'un certain rang pour les algorithmes CPU et GPU3, et que les temps pour GPU1 et GPU2 n'ont pas d'incidence avec la comparaison entre eux et avec les autres algorithmes.

4 Observation

Globalement les résultats concordent avec nos attentes ; on peut dire qu'à partir du de la dimension 500*500, les algorithmes sont classé dans l'ordre attendu (CPU>GPU1>GPU2, en temps d'exécution) avec une vitesse d'allocation plus lente dans notre plage de test pour le GPU que pour le CPU. De plus, la différence de transfert de données pour les calculs entre les algorithmes GPU1 et GPU2 est bien visible est permet au GPU2 une meilleur efficacité temporelle.

Mais il y a quelques demi-surprises : le temps d'allocation, censé être identique pour chaque GPU est globalement le même, malgré une augmentation plus rapide de ce temps en fonction de la taille des matrices pour l'allocation nécessaire au GPU3. De plus, notre doute était fondé : les copies nécessaires dans des matrices plus grandes pour le GPU3 ajoutent des temps de calculs plus long même que la version CPU, que ce soit avec ou sans l'allocation de base, ce qui rend cet algorithme totalement incorrect et non valable. (face au algorithme GPU1 et GPU2). Enfin on remarque une lègere baisse de temps nécessaire pour l'allocation sur CPU entre les dimensions 32*32 et 256*256.

IV Conclusion

Grâce à cet expérience, nous avons pu saisir quelques prémices d'utilisation et de questionnement sur la programmation parrallèle sur GPU via le language CUDA, en nous basant sur la multiplication matricielle. Nous avons ainsi pu apréhender la méthode de pensées nécessaire à la parrallèlisation des calculs, ainsi que nous pencher sur les questions de transferts mémoires pouvant avoir un impact sur la vitesse de calculs.

Pour pousser plus loin sur la multiplication matricielle, il serait utile de faire varier l'emplacement mémoire pré-calcul pour introduire la variable dû à la localité ou non des données (ici, toutes les matrices étaient stockées sur la mémoire correspondant à leur unités de calcul [CPU ou GPU] respectives), ou de comparer en ne prenant que des tailles de matrice multiples de 2 (en améliorant peut être l'algorithme GPU3 qui dans le cas particulier où l'on a des matrices de dimension (X*64)*(X*64) ne ferait pas de copies). Il pourrait aussi être intéressant d'adapter ces algorithmes pour des matrices n'étant pas forcément carrées, et d'observer les différences de performances induites.