	Nom : SONG Prénom : Kun

Stage de fin d'année
Cycle Ingénieur 4ème année

Département : Electronique, Energie, Système

Tuteur école : CEDRIC COENIGUER
cedric.coeniguer@u-psud.fr, 0169337509

ACCÉLÉRATION DE LA DIAGONALISATION DES MATRICES RÉELLES SYMÉTRIQUES

Société : Fédération de Recherche Lumière Matière LUMAT
Adresse : Bât 210, Université Paris-Sud, 91405 Orsay CEDEX
Dates du stage : 15/05/2013 - 30/08/2013
Tuteur de l'entreprise : Philippe Dos Santos
Téléphone : 0169158255

Remerciements

Je tiens à remercier Philippe Dos Santos ainsi que Georges Raseev pour leur disponibilité et leur aide tout au long de ce stage. Je remercie également Yves Bergougnoux et Oliver de Kermoysan du service informatique de l'ISMO.

Je remercie aussi tout le personnel de l'ISMO pour leur accueil.

Enfin, je remercie Cédric Koeniguer qui m'a beaucoup aidé pour la recherche de stage.

Sommaire

Introduction..1
1 Présentation de la fédération LUMAT...2
2 La Grappe Massivement Parallèle de Calcul Scientifique (GMPCS)..4
2.1 Architecture matérielle et logicielle..4
2.2 Limitation du calcul parallèle..6
2.3 Les Processeurs de type CPU et GPU...8
3 Diagonalisation des matrices symétriques réelles en double précision...................................10
3.1 Théorie de la diagonalisation...10
3.2 Librairies spécialisées : MKL, CULA Tools, MAGMA...15
3.3 Diagonalisation des matrice avec le CPU et le GPU..16
Apports personnels 24
Conclusion...25

Introduction

La mission de mon stage est d'étudier l'accélération de la diagonalisation des matrices symétriques réelles en double précision avec trois différentes librairies de calcul scientifique : MKL, CULA et MAGMA. Le matériel utilisé est une grappe de calcul, la Grappe Massivement Parallèle de Calculs Scientifique (GMPCS) de la fédération LUMAT.

Dans la recherche scientifique, les calculs matriciels sont souvent utilisés, et la diagonalisation est un calcul dont la complexité est en $O(N^3)$ où N est la taille des matrices. Plus la taille des matrices est grande plus le temps de calcul est élevé. L'accélération de ce type de calcul devient donc intéressante.

La GMPCS possède 25 nœuds de calcul, chacun a deux processeurs de type CPU INTEL (4 à 8 cœurs) et cela permet de faire des calculs en parallèle. Parmi les nœuds de calcul il y a deux nœuds qui sont connectés à deux accélérateurs GPU (Graphics Processing Unit). Le GPU, spécialisé dans le traitement des images, permet depuis quelques années, de faire du calcul scientifique en double précision.

Les trois librairie associées au calcul scientifique que sont MKL, CULA et MAGMA nécessitent chacune des ressources matérielles différentes. Les performances de ces trois librairies sont intéressantes pour l'accélération de la diagonalisation.

Pendant ce stage, le but est d'obtenir les meilleures accélérations de diagonalisation des matrices symétriques réelles en double précision.

1 Présentation de la fédération LUMAT

La fédération LUmière MATière (LUMAT) est une fédération de recherche, regroupant quatre laboratoires de recherche de l'Université Paris Sud situés sur le campus d'Orsay : le Laboratoire Aimé Cotton (LAC, UPR 3321), le Laboratoire de Physique des Gaz et des Plasmas (LPGP, UMR 8578), le Laboratoire Charles Fabry (LCF, UMR 8501) et l'Institut des Sciences Moléculaires d'Orsay (ISMO, UMR 8214). Cette fédération est composée d'environ 320 permanents rattachés au Centre National de la Recherche Scientifique (CNRS), à l'Université Paris Sud et à l'Institut d'Optique Graduate School. Les 320 permanents de la fédération sont constitués de :

- ➢ 115 chercheurs ;
- ➢ 75 enseignants-chercheurs ;
- ➢ 130 ingénieurs, techniciens et administratifs.

Fédération Lumière Matière (LUMAT - FR2764)
Direction : D. Dowek
Secrétariat et Gestion : C. Jucha

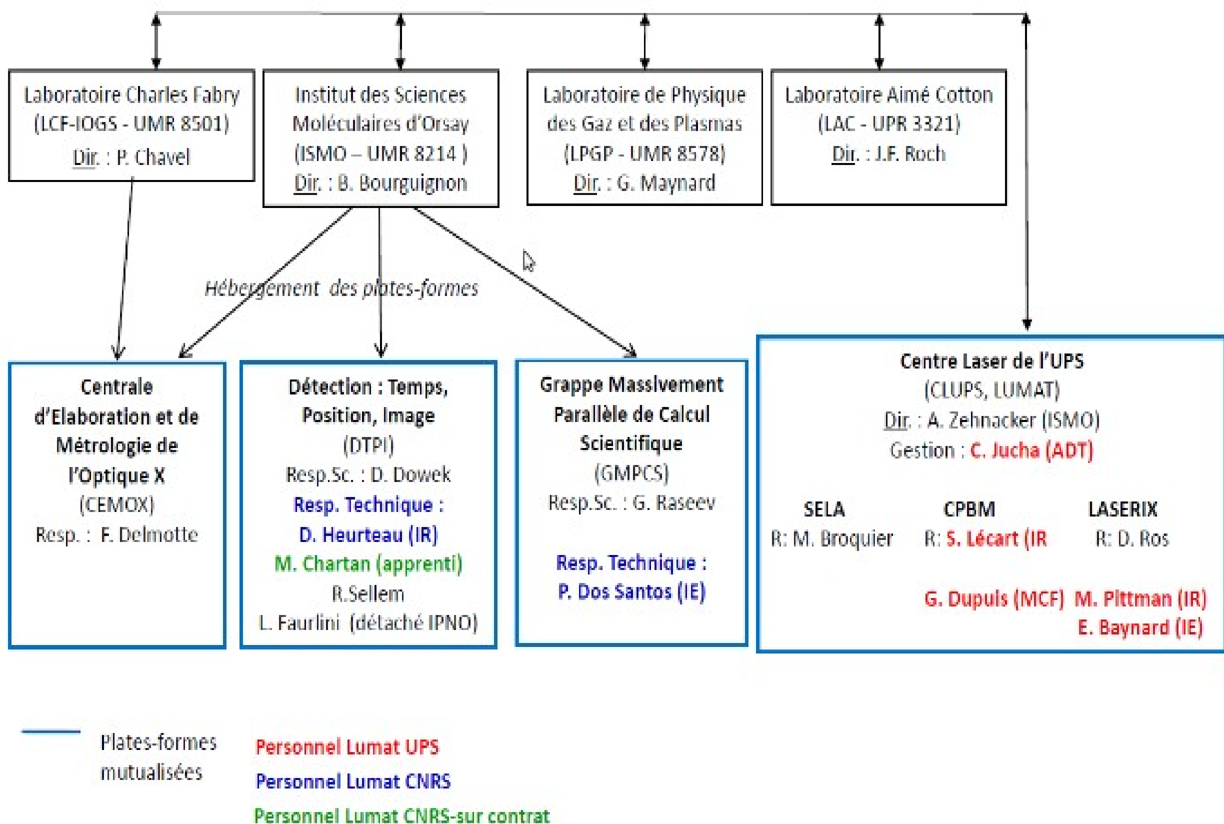

Figure 1.1 Organisation LUMAT

La fédération LUMAT a pour objectif de mutualiser les plates-formes et le savoir-faire entre les différentes unités de recherche qui la constituent, en passant par le développement des projets scientifiques communs.

La fédération LUMAT joue un rôle important dans l'animation scientifique entre les différentes équipes de recherche qui la composent grâce au financement des projets de recherche.

Pendant mon stage j'ai travaillé sur une plate-forme de la fédération LUMAT, la Grappe Massivement Parallèle de Calculs Scientifique GMPCS). Cette grappe permet de faire des calculs en parallèle en utilisant des processeurs de type Central Processing Unit (CPU) et de type Graphics Processing Unit (GPU). Le travail du service informatique rattaché à la GMPCS est de rendre la grappe plus performante, de surveiller son fonctionnement et d'optimiser son utilisation.

2 La Grappe Massivement Parallèle de Calcul Scientifique (GMPCS)

Les calculateurs sont classés en quatre catégories en fonction de leur puissance de calcul. Dans ce classement, la GMPCS fait partie des mésocentres (Tier2) car sa puissance de calcul est 5 TFLOPS crête. C'est un calculateur de puissance intermédiaire entre la station de travail (Tier 3) et les centres de calcul nationaux (Tier 1).

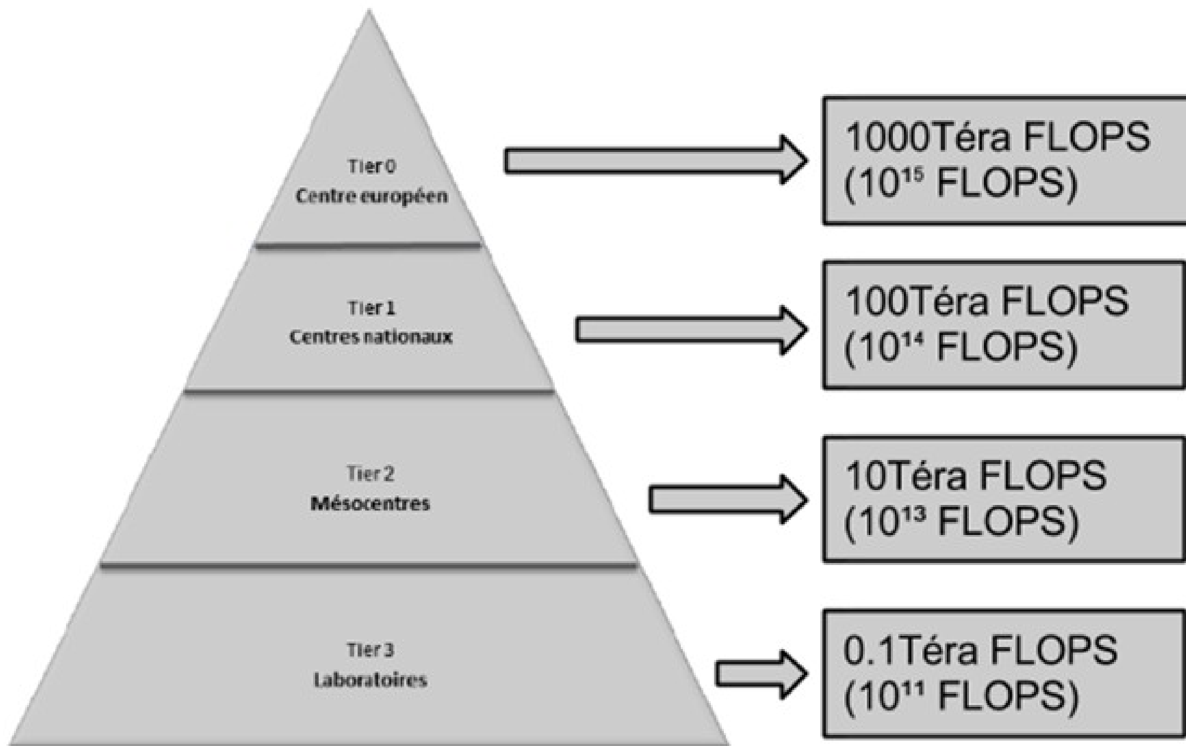

Figure 2.1

2.1 Architecture matérielle et logicielle

Assemblée par la société transtec, la GMPCS est un ensemble de nœuds de calcul connectés à un nœud maître dédié à la gestion de la grappe. Les nœuds de calcul sont des ordinateurs constitués de deux processeurs multi-coeurs et d'environ 50 Go de mémoire RAM chacun. Ces nœuds sont reliés à un réseau dédié au calcul (réseau InfiniBand à 40 Gbit/s), grâce à cette architecture il est possible d'effectuer des calculs en parallèle sur un seul nœud (sur plusieurs cœurs de processeurs en OpenMP) ou sur plusieurs nœuds de calcul (MPI). Cela permet de profiter de la puissance de plusieurs nœuds.
Voici le schéma de l'architecture de la GMPCS :

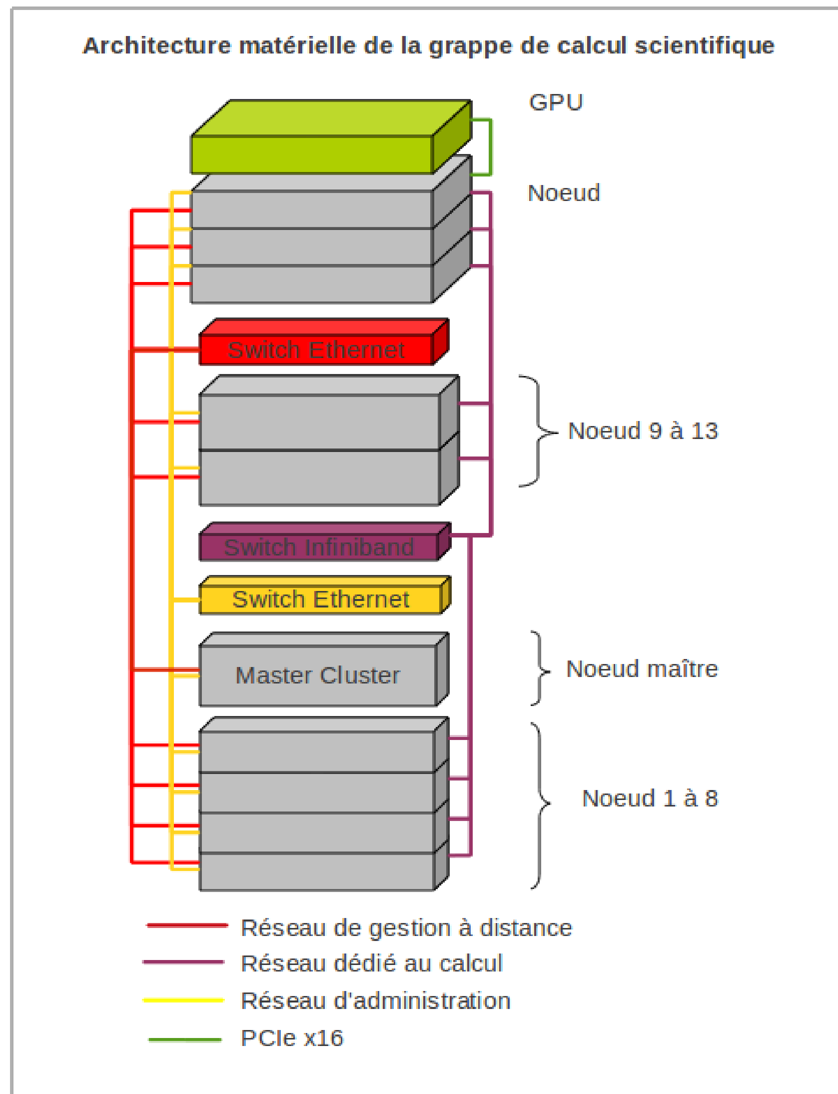

Figure 2.2 Architecture de la GMPCS

La GMPCS est composée :

- un nœud maître ;
- 25 nœuds dédiés au calcul.

Le nœud maître est le point unique d'entrée à la GMPCS, il joue donc le rôle :

- point d'accès : c'est le seul point d'entrée sur toute la grappe. Il est directement accessible via l'Internet par le protocole SSH.
- gestion des jobs : il réserve les ressources sur les nœuds de calcul (nombre de cœurs, quantité mémoire, durée) et distribue les jobs des utilisateurs en veillant à l'équilibre de la charge de la GMPCS via le logiciel SGE.
- gestion des nœuds : il gère le déploiement du système d'exploitation au démarrage des nœuds via le logiciel xCAT.
- suivi de la charge : il permet de suivre la charge de l'ensemble des nœuds via ganglia.
- suppression : il détecte tout défaut hardware et le signale via Nagios

- gestion des données : toutes les données envoyées par les utilisateurs pour le calcul sont tout d'abord stockées sur le nœud maître, puis par la suite accessibles sur les nœuds de calcul au moyen du partage interne de fichiers.

Les 25 nœuds de calcul sont dédiés uniquement aux calculs. Ces nœuds sont connectés à 3 réseaux avec des fonctionnalités complètement différentes :

- le réseau d'administration utilisant le protocole Ethernet, est un réseau utilisé pour le partage des dossiers entre les utilisateurs et également pour le transfert des codes sur les nœuds.
- le réseau de gestion à distance permet de gérer l'arrêt et le démarrage des nœuds.
- le réseau InfiniBand (réseau rapide dédié aux calculs), est un réseau principalement utilisé pour les calculs parallèles.

2.2 Limitation du calcul parallèle

Les processeurs actuels disposent de plusieurs cœurs et il est possible d'utiliser en parallèle tous les cœurs pour accélérer le calcul. Mais l'accélération du parallélisme est limitée. En effet, la loi de Amdahl présente l'accélération maximale qu'on peut obtenir en parallélisme.

L'accélération :
L'accélération (speedup en anglais) donne le gain de temps obtenu avec un programme parallèle par rapport à sa version séquentielle. La définition de l'accélération est :

$$A(N) = \frac{T(1)}{T(N)}$$

- N : nombre de processeurs utilisés
- A(N) : accélération
- T(N) : temps d'exécution total

Accélération maximale :
L'accélération d'un programme parallèle est contrainte par sa partie séquentielle. La loi de Amdahl est utilisée pour prévoir l'accélération maximale. La loi de Amdahl montre que l'accélération est bornée et dépend du nombre de cœurs N et de la portion séquentielle du programme, s :

$$A(N) = \frac{1}{s + (1-s)/N}$$

- A(N) : accélération
- s : la fraction de temps que le programme passe dans la portion séquentielle
- N : nombre de cœurs utilisés

Un programme ou une partie de programme sans proportion séquentielle (s=0), bénéficie directement de l'accélération :

$$A(N) = N$$

En réalité les programmes disposent d'une partie séquentielle et d'une partie parallèle, l'accélération est bornée par la partie séquentielle quelque soit le nombre de cœurs utilisés :

$$A(N) = \frac{1}{s + (1-s)/N} < \frac{1}{s}$$

La formule cd-dessus peut être réécrite en fonction de la portion parallèle du programme en posant

$$s = 1 - p$$

$$A(N) = \frac{1}{1 - p + \frac{p}{N}} < \frac{1}{1-p}$$

La Figure 2.3 donne le lien de l'accélération et la proportion de partie parallèle d'un programme en utilisant de différents nombres de processeurs.

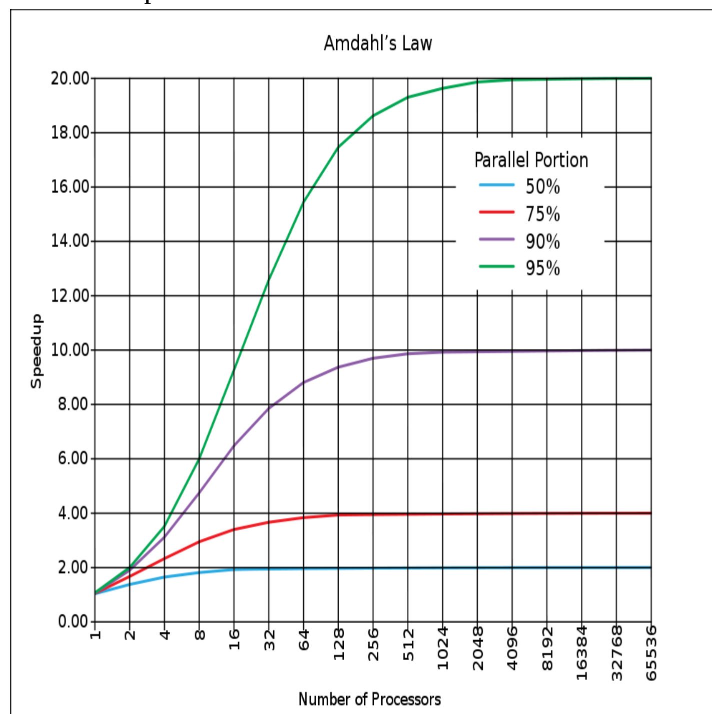

Figure 2.3

L'accélération présentée par la loi d'Amdahl est théorique car en réalité il y a plusieurs surcoûts qui vont diminuer l'accélération : la création, la destruction et la synchronisation des processus, la communication entre processus etc... Typiquement, plus il y a de processus plus les surcoûts sont élevés.

2.3 Les Processeurs de type CPU et GPU

CPU

Le processeur de type CPU (Central Processing Unit) est un composant central d'un ordinateur, il contient deux types d'éléments chargés du calcul sur les nombres entiers et les nombres flottant : ALU (Arithmetic Logic Unit), FPU (Floating Point Unit). Pour diminuer le temps d'inactivité des unités de calcul, le CPU possède des unités évoluées (Out of Order Execution, Branch-Prediction, Hyper-Threading). Ces unités permettent d'exécuter plus vite les programmes, le processeur devient par conséquent extrêmement compliqué et cela met en difficulté d'ajouter des cœurs sur une seule puce.

Les processeurs récents disposent de plusieurs CPUs appelés cœur ('core' en anglais) et chaque coeur traite ses données indépendamment. La figure 3.2 présente l'architecture d'un processeurs multi-cores.

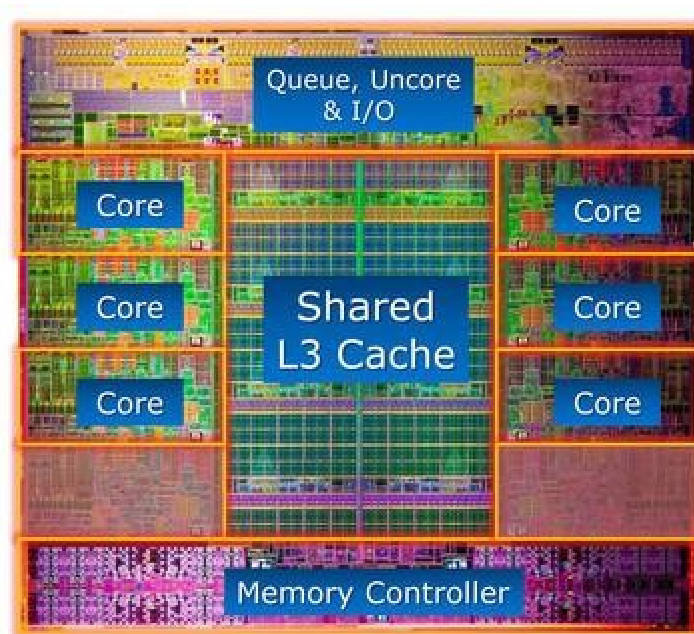

Figure 2.4 – Processeur multicoeurs

Néanmoins il y a peu de cœurs sur une seule puce (~10), le CPU est donc limité pour les programmes qui nécessitent beaucoup de calculs parallèles.

Les technologies évoluées sur le CPU:

Branch Prediction (prédiction de branchement): une fonctionnalité de processeur qui lui permet de prédire le résultat d'un branchement. Cette technique permet à un processeur rendre sa pipeline plus efficace, le branchement a commencé à exécuter avant la fin de l'instruction précédente lorsque la prédiction est correcte. Avec l'algorithme bien évoluée, le taux de prédiction réussite peut atteindre un niveau plus de 90%.

Out Of Order execution (l'exécution dans le désordre): au lieu de perdre du temps à attendre que toutes les instructions soient disponibles, le processeurs traite directement celles dont il dispose et réorganise l'ensemble à la fin du traitement, à fin de maintenir la cohérence chronologique indispensable. Cette technologie permet le CPU d'exécuter les instructions sur la disponibilité plutôt que la séquence, par conséquent le CPU évite d'attendre les instructions séquentielles et gagne du temps.

Accélérateur de type GPU

L'accélérateur de type GPU (Graphics Processing Unit) est un périphérique qui aide le processeur à accélérer les programmes de calcul. Il est composé d'un grand nombre de cœurs (~1000). Un cœur de GPU est plus simple qu'un cœur de CPU :

- les cœurs ne sont pas autonomes,
- les cœurs n'ont pas d'unités évoluées (Out of Order Execution, Branch-Prediction, Hyper-Threading)

Comme le GPU contient un nombre important de cœurs (beaucoup plus que le CPU multi-cores), le parallélisme est important sur l'accélérateur. L'accélérateur GPU est particulièrement bien adapté au calcul matriciel.

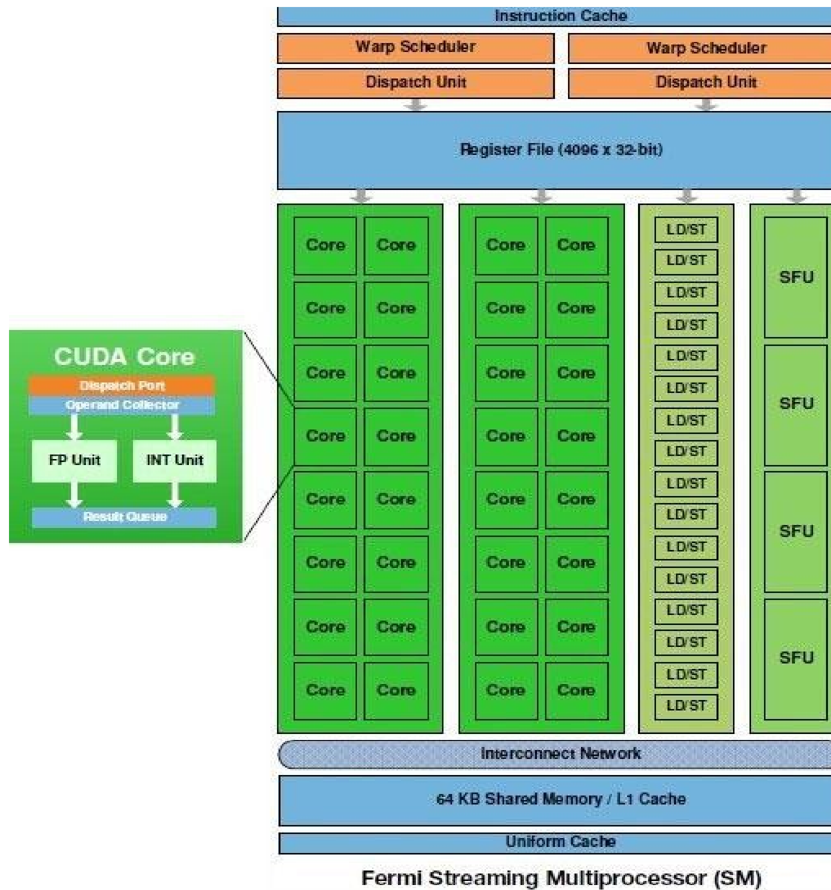

Figure 2.5 Architecture de GPU CUDA

3 Diagonalisation des matrices symétriques réelles en double précision

3.1 Théorie de la diagonalisation

Dans cette partie, je vais présenter la théorie élémentaire du problème de valeurs propres de matrices ainsi que quelques algorithmes associés. Je resterai aux matrices symétriques réelles mais certains éléments liés aux cas généraux sont également présentés.

3.1.1. Résumé des propriétés de matrices

On commence par la présentation de différents types de matrices réelles et leur propriétés:

1. La taille ou dimension d'une matrice carrée corresponds le nombre de lignes ou colonnes; le rang d'une matrice corresponds le nombre des valeurs propres non-nuls.

2. Opération de matrices :

transposée	$\boldsymbol{A}^T$	$[a_{ij}]^T = a_{ji}$
inverse	$\boldsymbol{A}^{-1}$	$\boldsymbol{A}^{-1}\,\boldsymbol{A} = \boldsymbol{A}\,\boldsymbol{A}^{-1} = \boldsymbol{I},\ \det\lvert A\rvert \neq 0$
multiplication (non commutative)	$\boldsymbol{A}\,\boldsymbol{B} \neq \boldsymbol{B}\boldsymbol{A}$	$\sum_{k=1}^{N} a_{ik}\,b_{kj} \neq \sum_{k=1}^{N} b_{ik}\,a_{kj}$

3. Types de matrices :

identité	$\boldsymbol{I}$	$i_{ij} = \delta_{ij}$
diagonale	$\boldsymbol{A}$	$a_{ij} = 0,\ \forall i \neq j$
symétrique réele	$\boldsymbol{A} = \boldsymbol{A}^T$	$a_{ij} = a_{ji}$
orthogonale	$\boldsymbol{A}\,\boldsymbol{A}^T = \boldsymbol{A}^T\,\boldsymbol{A} = \boldsymbol{I}$	
triangulaire supérieure/inférieure	$\boldsymbol{A}$	$a_{ij} = 0,\ \forall i > j\ /\ \forall i < j$
tridiagonale	$\boldsymbol{A}$	$a_{ij} = 0,\ \forall \lvert i-j\rvert < 1$

4. Matrice semblable et diagonalisation. Les matrices ***A*** et ***B*** sont dites semblables, s'il existe une matrice inversible telle que :

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{B}\boldsymbol{U}^{-1} \qquad \boldsymbol{B} = \boldsymbol{U}^{-1}\boldsymbol{A}\boldsymbol{U}$$

$$\Delta = \det\lvert\boldsymbol{A} - \lambda\,\boldsymbol{I}\rvert \qquad \text{polynome characteristique}$$

On appelle cette transformation la transformation de similarité et ce type de matrice la matrice semblable. Les matrices semblables ont le même polynôme caractéristique et ont donc les mêmes valeurs propres. La diagonalisation est une transformation de similarité où la matrice $A \equiv B$ est diagonale. Les valeurs propres sont les éléments diagonaux de la matrice ***E*** qui sont les racines du polynôme caractéristique. Les transformations de similarité sont exploitées dans la diagonalisation de matrices.

3.1.2. Diagonalisation de matrices symétriques réelles

Nous allons étudier les matrice pleine, symétrique et réelle avec les éléments non-nuls(en opposant les matrices creusée avec beaucoup d'éléments nuls).

L'une des stratégie pour diagonaliser une matrice consiste à effectuer une série de transformations de similarité

$$\begin{aligned} &\boldsymbol{A} \;\rightarrow \boldsymbol{P}_1^{-1}\boldsymbol{A}\boldsymbol{P}_1 \;\rightarrow\; \boldsymbol{P}_2^{-1}\boldsymbol{P}_1^{-1}\boldsymbol{A}\boldsymbol{P}_1\boldsymbol{P}_2 \;\rightarrow\; \boldsymbol{P}_3^{-1}\boldsymbol{P}_2^{-1}\boldsymbol{P}_1^{-1}\boldsymbol{A}\boldsymbol{P}_1\boldsymbol{P}_2\boldsymbol{P}_3 \;\rightarrow\; \text{etc} \\ &\boldsymbol{U} \;=\; \boldsymbol{P}_1\boldsymbol{P}_2\boldsymbol{P}_3\ldots, \end{aligned}$$

où les vecteurs propres ***U*** sont une matrice de production de ces transformations de similarité ***P***. S'il y a des valeurs propres dégénérées($e_{ii}=e_{jj}$) et/ou la rang de matrice est plus faible que la dimension/taille(i.e certaines valeurs propres sont nulles) il faut que la diagonalisation utilise un algorithme spécifique ou additions spécifiques supplémentaires.

3.1.2.1 Diagonalisation de Jacobi

Dans la diagonalisation de Jacobi, la matrice diagonale ***E*** des valeurs prores et les vecteurs propres ***U*** sont telles que:

$$\begin{aligned} \boldsymbol{E} &= \boldsymbol{U}^T\,\boldsymbol{A}\,\boldsymbol{U} \\ \boldsymbol{U} &= \boldsymbol{P}_1\,\boldsymbol{P}_2\,\boldsymbol{P}_3\,\ldots \end{aligned}$$

où Pi est une rotation de Givens élémentaire, en effet une transformation de similarité ***P*** donnée par :

$$^{pq}\boldsymbol{P} = \begin{pmatrix} 1 & \ldots & 0 & 0 & \ldots & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & c & 0 & \ldots & s & \ldots & 0 \\ 0 & \ldots & 0 & 1 & \ldots & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & -s & 0 & \ldots & c & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & 0 & 0 & \ldots & 0 & \ldots & 1 \end{pmatrix}$$

où l'indice "pq" signifie que une opération effectuée sur les éléments pq et qp de la matrice symétrique à diagonaliser. Les éléments c = cosø et s = sinø, $c^2 + s^2 = 1$ effectuent une rotation correspondante la transformation de similarité et permettra de mettre à zéro les éléments hors-diagonaux $a_{pq}=a_{qp}=0$ avec un angle particulier de rotation ø. La transformation de similarité de la matrice ***A*** devient:

$$\boldsymbol{A}' \;=\; {}^{pq}\boldsymbol{P}^T\,\boldsymbol{A}\,{}^{pq}\boldsymbol{P}.$$

Pour expliquer cet effet de la rotation dans la procédure de la diagonalisation de Jacobi, nous envisageons explicitement une 2×2 matrice de rotation. La transformation 12 a explicitement :

$$\boldsymbol{A}' \;=\; {}^{12}\boldsymbol{P}^T\,\boldsymbol{A}\,{}^{12}\boldsymbol{P} = \begin{pmatrix} c & -s \\ s & c \end{pmatrix}\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{11} \end{pmatrix}\begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

qui donne:

$$A' = \begin{pmatrix} c\,(c\,a_{11} - s\,a_{21}) - s\,(c\,a_{12} - s\,a_{22}) & s\,(c\,a_{11} - s\,a_{21}) + c\,(c\,a_{12} - s\,a_{22}) \\ c\,(s\,a_{11} + c\,a_{21}) - s\,(s\,a_{12} + c\,a_{22}) & s\,(s\,a_{11} + c\,a_{21}) + c\,(s\,a_{12} + c\,a_{22}) \end{pmatrix}$$

Maintenant on met à zéro les termes hors-diagonaux a'_{12} et a'_{21}

$$(c^2 - s^2)\,a_{12} + sc\,(a_{11} - a_{22}) = 0$$

Divisé par c² et on suppose que $t=s/c$ et $\theta=(a_{22}-a_{21}/2a_{12})$ on obtient une équation de second ordre :

$$t^2 + 2\,\theta\,t - 1 = 0$$

La racine la plus petite de cette équation qui correspond un angle de rotation à $\pi/4$:

$$t = -\theta \pm \sqrt{\theta^2 + 1} \qquad c = (1 + t^2)^{-1/2} \qquad s = t\,c$$

On met à zéro progressivement les lignes première, deuxième, troisième etc… On parcourt toutes les lignes et colonnes en utilisant cette procédure. Pour une matrice symétrique de taille N, le nombre total de rotations est $N(N-1)/2$. La convergence s'atteint après 6 à 10 parcours, i.e. $3N^2$ à $5N^2$. Chaque rotation nécessite 8N opération, cet algorithme nécessite donc $24N^3$ à $40N^3$ opérations. Calcul des valeurs propres réduira le nombre d'opération de $8N^3\,à\,12N^3$.

3.1.2.2 Diagonalisation de Householder

La transformation de Householder réduit la matrice symétrique(ou bien les matrices plus généraux) à la forme tridiagonale. Ensuite on diagonalise cette forme tridiagonale en utilisant un autre algorithme QL décomposition.

3.1.2.2.1 Tridagonalisation

Dans cette première partie d'algorithme j'ai utilisé un article de wikipedia (following Burden and Faires, Numerical Analysis,8-th edition). J'introduirai en parallèle les étapes de transformation à la forme tridiagonale avec un exemple numérique donnée dans wikipedia. Considérer une matrice A , symétrique réelle de taille **N** et un exemple numérique 4×4 A_{num}

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1N} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2N} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3N} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{N1} & a_{N2} & a_{N3} & \cdots & a_{NN} \end{pmatrix} \qquad \boldsymbol{A}_{num} = \begin{pmatrix} 4 & 1 & -2 & 2 \\ 1 & 2 & 0 & 1 \\ -2 & 0 & 3 & -2 \\ 2 & 1 & -2 & -1 \end{pmatrix}$$

On considère les expressions généraux pour A^k avec $k\in[1,N-1]$, et en parallèle on donne le résultat numérique pour $A_{num}^{(1)}$. On met à zéro les éléments de la première ligne et de la première colonne en excluant ceux du diagonal et du premier hors-diagonal de la matrice A . On introduit deux scalaires $\alpha^{(k)}$ et $r^{(k)}$:

$$\alpha^{(k)} = -\mathrm{sign}(a^{(k)}_{k+1\,k})\sqrt{\sum_{j=k+1}^{N}[a^{(k)}_{j\,k}]^2} \qquad \alpha^{(1)} = -3$$

$$r^{(k)} = \sqrt{\frac{\alpha^{(k)}}{2}(\alpha^{(k)} - a^{(k)}_{k+1\,k})} \qquad r^{(1)} = \sqrt{6}$$

où $\alpha^{(k)}_{k+1,k}$ au-dessus est un élément de la matrice $A^{(k)}$ obtenue après k transformations successives. On construit le vecteur $v^{(k)}$ à partir $\alpha^{(k)}$ et $r^{(k)}$

$$\boldsymbol{v}^{(k)} = \begin{pmatrix} 0 \\ (a^{(k)}_{2k} - \alpha^{(k)})/(2\,r^{(k)}) \\ a^{(k)}_{3k}/(2\,r^{(k)}) \\ \vdots \\ a^{(k)}_{Nk}/(2\,r^{(k)}) \end{pmatrix} \qquad \boldsymbol{v}^{(1)} = \begin{pmatrix} 0 \\ \sqrt{2/3} \\ -1/\sqrt{6} \\ 1/\sqrt{6} \end{pmatrix}$$

Maintenant on calcule la transformation de Householder

$$\boldsymbol{P}^{(k)} = \mathbf{1} - 2\,\boldsymbol{v}^{(k)}\,(\boldsymbol{v}^{(k)})^T \qquad \boldsymbol{P}^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1/3 & 2/3 & -2/3 \\ 0 & 2/3 & 2/3 & 1/3 \\ 0 & -2/3 & 1/3 & 2/3 \end{pmatrix}$$

et appliquer une transformation de similarité à la matrice A

$$\boldsymbol{A}^{(k)} = \boldsymbol{P}^{(k)}\boldsymbol{A}\boldsymbol{P}^{(k)} \qquad \boldsymbol{A}^{(1)}_{num} = \begin{pmatrix} 4 & -3 & 0 & 0 \\ -3 & 10/3 & 1 & 4/3 \\ 0 & 1 & 5/3 & -4/3 \\ 0 & 4/3 & -4/3 & -1 \end{pmatrix}$$

Dans notre exemple numérique, $a^{(1)}_{31}$, $a^{(1)}_{41}$, $a^{(1)}_{13}$ et $a^{(1)}_{14}$ sont à zéro. On met à les lignes et les colonnes $k=1,2,3,\ldots,N-1$ en utilisant la transformation de similarité et on obtient finalement une matrice tridiagonale $A^{(N-1)}$.

Le stockage exigée par cet algorithme de Householder consiste en une matrice initiale symétrique à diagonaliser et un vecteur pour les valeurs propres. En sortie les vecteurs propres se placent au lieu de la matrice initiale. Pour une N élevée, le nombre d'opérations dans la réduction Householder à la forme tridiagonale, est $2\mathrm{N}^3/3$ (valeurs propres seulement) et $4\mathrm{N}^3/3$ quand les valeurs propres et les vecteurs propres sont tous demandés.

3.1.2.2.2 Diagonalisation de matrices tridiagonales avec l'algorithme QL

Une matrice symétrique peut être décomposé comme un produit

$$\boldsymbol{A} = \boldsymbol{Q}\,\boldsymbol{L}$$

où Q est orthogonal et L est une matrice triangulaire inférieure. On peut définir une séquence s pour la transformation orthogonale en utilisant le produit inversé QL de la matrice au-dessus

$$\boldsymbol{A}_{s+1} = \boldsymbol{L}_s\,\boldsymbol{Q}_s = \boldsymbol{Q}_s^{-1}\,\boldsymbol{A}_s\,\boldsymbol{Q}_s = \boldsymbol{Q}_s^T\,\boldsymbol{A}_s\,\boldsymbol{Q}_s$$

où la dernière égalité est assurée car Q est orthogonale. Cette séquence de s transformations aboutit une matrice diagonal. Pour une matrice tidiagonale, le travail de cette tranformation est aussi bas que $O(N)$ (comparer avec $O(N^3)$ pour une matrice pleine symétrique).

Au lieu d'utiliser l'algorithme QL, on peut diagonaliser une matrice tridiagonale en utilisant de pleines rotations de Givens dans la section précédente 2.1.

La convergence d'un algorithme dépend de la dispersion des valeurs propres:

- D'abord si les valeurs propres sont dégénérées , alors que les éléments hors-diagonaux supérieure ou inférieure sont zéros et une telle matrice est diagonale par blocs. Ensuite chaque bloc peut être séparément diagonalisé, ce qui réduit le montant de l'effort de calcul. Les vecteurs propres correspondant aux valeurs propres dégénérés ne sont pas uniques et rotation n'auront aucun effet sur eux. Pour obtenir les vecteurs propres orthogonaux on peut effectuer par exemple l'orthogonalisation de Schmidt.
- Si les valeurs propres $|e_i - e_j| < e$, avec e petit, alors que la convergence est lente. La procédure générale d'accélérer la convergence est de déplacer par, dit k , quelques éléments diagonaux de la matrice tridiagonale, le valeur propre correspondante devient maintenant $e_i - k$. Il existe deux type de déplacements, déplacement implicite et déplacement explicite, mais on ne discutera pas l'algorithme de déplacement.

D'après la courte discussion au-dessus, c'est claire que la diagonalisation de matrices tridiagonaux utilise branchement et ils sont inefficace sur un GPU.

3.2 Librairies spécialisées : MKL, CULA Tools, MAGMA

MKL(Math Kernel Library)
La librairie MKL est une implémentation des routines BLAS[1] (Basic Linear Algebre Subroutines), proposée par Intel, supportant plusieurs langages de programmation (C, C++, Fortran). C'est une librairie de routines mathématiques optimisées pour les calculs scientifiques sur les CPUs Intel et les processeurs compatibles. Cette librairie nous permet de faire la diagonalisation en mode séquentiel(monothread sur un cœur) et en mode parallèle (multithread sur plusieurs cœurs).

CULA Tools
CULA Tools est une librairie algèbre linéaire accélérée basée CUBLAS (CUda Basic Linear Algebre Subroutines) qui est une implémentation des routines BLAS[1] sur GPU. Elle est optimisée pour les architectures GPU Nvidia. Cette librairie nous permet d $O(N^3)$ e faire la diagonalisation sur le GPU en utilisant le parallélisme.

MAGMA (Matrix Algebra on GPU and Multicore Architectures)
La librairie MAGMA est une librairie issue d'un projet de l'ICL (the Innovative Computing Laboratory of University Tenessee) contenant les implémentations MKL et CUBLAS des routines BLAS[1]. Cette libraire hybride combinant les avantages du CPU et du GPU, nous permet de faire les calcul en utilisant le CPU ainsi que le GPU. L'accélération est donc élevée, mais en revanche utiliser cette librairie demande plus de ressources (RAM et matériel).

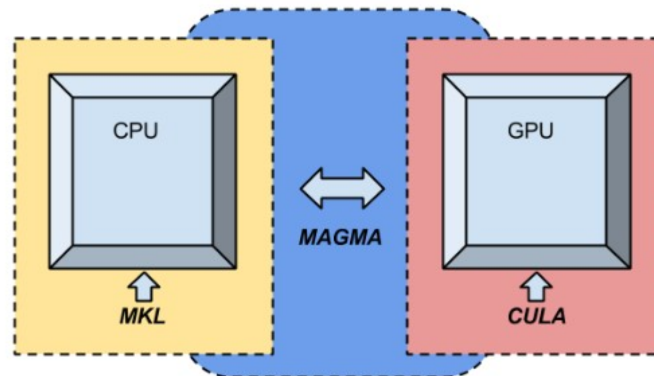

Figure 3.1

La figure 3.1 présente les relations entre les librairies et le matériel.

1 Site officiel: http://www.netlib.org/blas/faq.html

Un même algorithme pour MKL, CULA et MAGMA

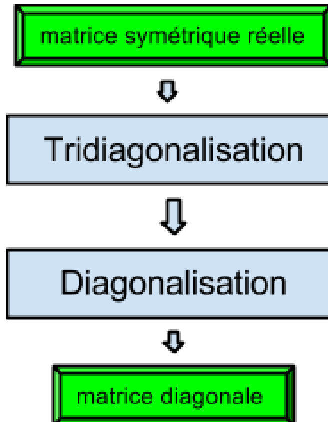

Figure 3.2 Procédure de diagonalisation

Les trois librairies de calcul (MKL, CULA et MAGMA) utilisent le même algorithme pour la diagonalisation des matrices réelles symétriques. La diagonalisation est décomposée en deux étapes :

1. tridiagonalisation (méthode de Householder) , complexité : $O(N^3)$;
2. diagonalisation (algorithme QL avec décalage implicite), complexité : $O(N)$.

Dans la première étape, le nombre d'opérations est connu à l'avance car il y a exactement N-2 transformations de Householder. Mais dans la deuxième étape, le nombre d'opérations n'est plus connu à l'avance, l'algorithme s'arrête quand la convergence est atteinte.

3.3 Diagonalisation des matrice avec le CPU et le GPU

3.3.1.Présentation des matrices utilisées :

Comme indiqué dans la partie théorique, la diagonalisation dépend de la convergence de l’algorithme. Les coefficients de la matrice ont une grosse influence sur le temps de diagonalisation. Il faut générer une matrice pour faire des tests. Il y a deux fonctions qui nous permettent de générer les nombres aléatoires:

- rande() ;
- dlarnv().

Deux matrices symétriques réelles en double précision sont générées : une "matrice 1" par la fonction rand() sur l’intervalle [1,256] notée **M1**, une "matrice 2" par la fonction dlarnv() sur l’intervalle [0,1] notée **M2**. Chaque élément des matrices **M1** et **M2** est un nombre aléatoire généré par les deux fonctions au-dessus. Il n’y a pas de différence entre les éléments diagonaux et ceux hors-diagonaux.

A la fin de stage, j’ai utilisé aussi une matrice à diagonale dominante **M3** donnée par un utilisateur de grappe pour faire la comparaison des trois librairies. Cette matrice est plus proche du travail réel dans la pratique.

3.3.2. Accélération CPU (MKL)

Le processeur récent porte souvent plusieurs coeurs physiques, au niveau d'exécution du programme, nous pouvons choisir le mode de fonctionnement :

MonoThread : le programme sera exécuté sur un seul coeur en séquentiel ;

MultiThread : le programme sera exécuté sur plusieurs coeurs en parallèle.

Avec la librairie MKL, tout le travail est fait uniquement sur le CPU. La fonction DSYEV() de la MKL permet de faire la diagonalisation d'une matrice symétrique réelle. Un compilateur Intel attaché avec la MKL peut compiler le code du programme en mode MonoThread ou bien en MultiThreads par un paramètre choisi.

Le temps de diagonalisation en mode monothread est utilisé comme référence pour mesurer l'accélération.

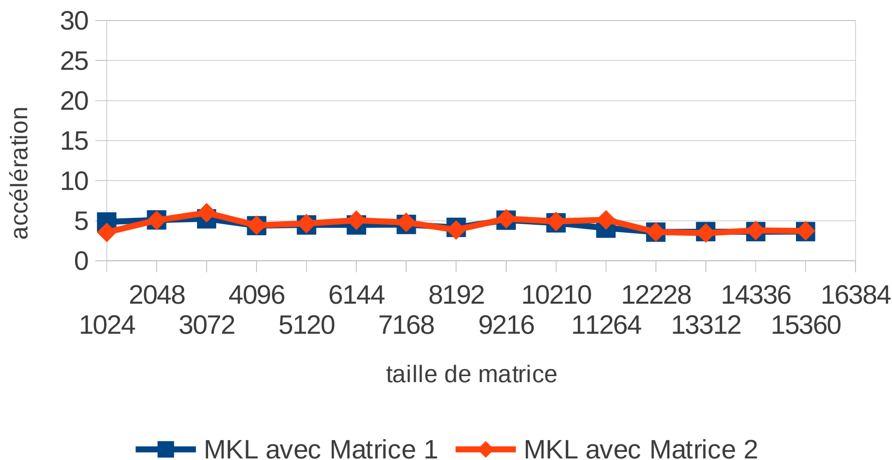

Figure 3.3

Dans ce test, j'ai comparé l'accélération MultiThread (6 cœurs de Xeon 5650 soit une puce complète) par apport au temps MonoThread, le test est fait sur deux matrices différentes, on voit bien que les deux courbes d'accélération sont proches l'une à l'autre. L'accélération du multithread avec 6 cœurs est d'environ 5. D'après la loi de Amdhal, l'accélération est définie de la façon suivante :

$$A(N) = \frac{1}{s + (1-s)/N} < \frac{1}{s}$$

Comme l'accélération du multithread vaut : $A(6)=5$, nous avons $\frac{1}{s}=25$. Donc la limite maximale de l'accélération ne peut pas dépasser 25 avec l'implémentation MKL de la diagonalisation.

Pour augmenter l'accélération, il suffit d'avoir un processeur avec plus de cœurs comme le processeur E5-2670 avec 8 cœurs.

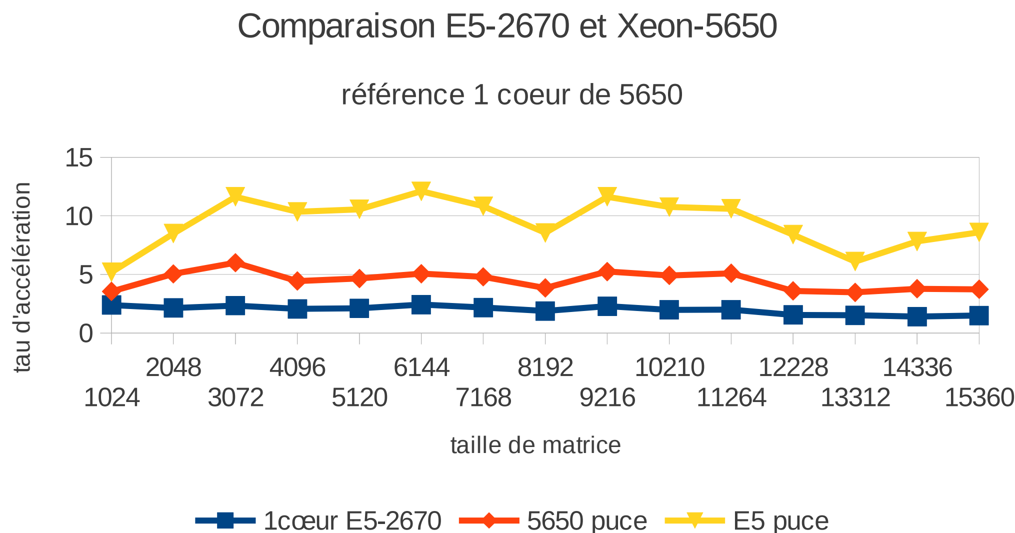

Figure 3.4

Pendant le stage, j'ai eu un accès à une grappe chez la société transtec avec les CPUs et les GPUs les plus récents. J'ai fait le même test sur la grappe chez transtec pour identifier la différence entre deux générations de processeurs. Dans la figure 3.4, la courbe bleue présente l'évolution du nouveau processeur. Il est bien claire qu'un cœur du nouveau processeur E5-2670 (un cœur) est presque deux fois plus rapide qu'un cœur de l'ancien (Xeon 5650) grâce à la nouvelle microarchitecture. La nouvelle technologie de fabrication permet d'ajouter plus de cœurs sur une seule puce, ce qui permet d'augmenter encore la puissance de calcul du processeur complet. Ceci est cohérent avec les données constructeurs fournies par Intel :

	Cœurs/Puce	Fréquence	FLOPS	FLOPS/coeur
Xeon-5650	6	2,66 GHz	63,984 GFLOPS	10,664 GFLOPS
E5-2670	8	2,6 GHz	166,4 GFLOPS	20,8 GFLOPS

Le FLOPS[2] présente la puissance de calcul, comme la puissance de calcul de E5-2670 est deux fois plus que celle de l'ancien, il est donc logique que l'accélération du nouveau processeur est deux fois plus que celle de l'ancien. Il y a aussi une chose à noter : quand j'ai fait le test avec la grappe chez transtec, il y avait un autre utilisateur qui travaillait sur le nœud de calcul en même temps. Il est possible que ce soit la raison pour laquelle l'accélération du E5-2670 est diminuée.

L'accélération de processeur actuel est environ 5, mais avec le processeur plus récent, l'accélération peut atteindre environ 10, soit deux fois plus rapide. Il y a encore de la marge pour atteindre 25, donc nous avons tout intérêt d'acheter des processeurs avec plus de cœurs.

2 FLOPS : (Floting-points Operations Per Secend).

3.3.3. Accélération GPU (CULA)

Le processeur graphique (GPU) n'est pas autonome. Pour faire des calculs, le GPU est sous le contrôle du processeur (CPU). Les programmes/calculs sont en effet exécutés sur le GPU en coopération du CPU. Le CPU et le GPU possèdent chacun ses mémoire propres (RAM). Pour faire la diagonalisation, il faut que le CPU prépare la matrice à diagonaliser dans sa mémoire. Ensuite le CPU la transfère à la mémoire du GPU pour qu'il puisse faire le calcul. A la fin de la diagonalisation, le CPU récupère le résultat.

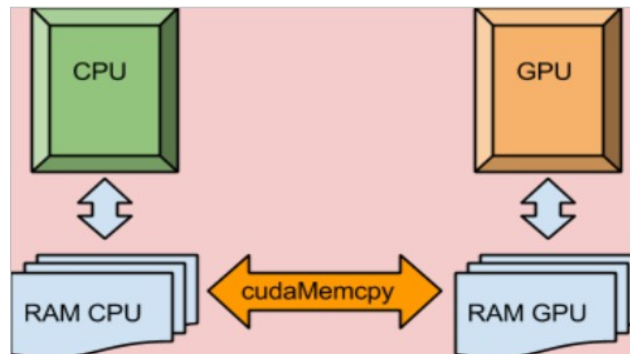

Figure 3.5

Le temps de diagonalisation est composé de deux parties: le temps de transfert et le temps de calcul. Voici un tableau qui présente le temps de transfert le temps de diagonalisation :

Matrix size	Temps_c2g(ms)	Temps_g2c(ms)	Temps_diago(ms)
64	0.027	0.042	10.735
128	0.069	0.090	61.092
256	0.239	0.297	80.511
512	0.735	0.858	306.312
1024	2.710	2.978	1215.530
2048	9.019	9.949	6047.181
4096	41.622	41.001	48589.246
8192	318.197	432.927	319048.812
16384	881.132	785.818	1105124.375

J'ai constaté que le temps de diagonalisation du GPU est important, le temps de transfert par rapport au temps de calcul est assez faible (moins de 1 %) donc on peut l'ignorer, c'est à dire que l'amélioration du temps de transfert n'est pas du tout intéressante.

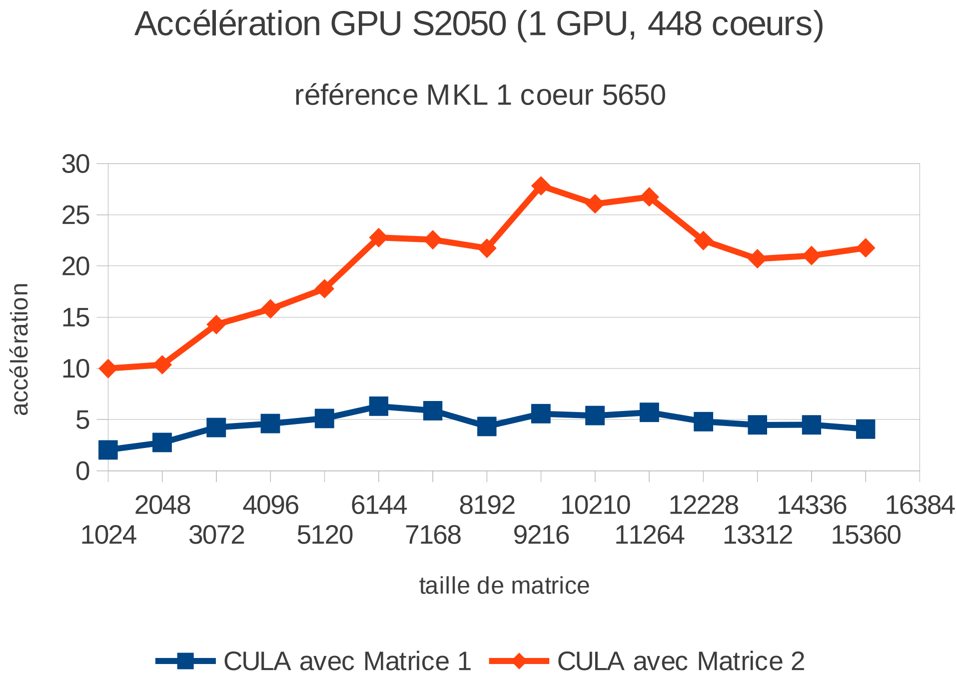

Figure 3.6

Dans ce test, j'ai utilisé la librairie CULA Tools pour la diagonalisation. Deux matrices **M1**, **M2** sont utilisées, la figure 3.6 présente l'accélération du GPU. La performance du GPU n'est pas stable par apport celle du CPU :

- Avec la matrice **M1**, il y a une accélération d'environ 5 ;
- Avec la matrice **M2** l'accélération est d'environ 25.

La différence est énorme entre les deux. Interprétation des résultats :

- Dans le cas de la matrice **M1**, nous supposons que le processeur GPU n'est pas capable de faire de l'exécution spéculative comme le processeur CPU. En effet, le CPU possède une unité de prédiction de branchement et d'exécution dans le désordre. Ceci lui permet d'optimiser l'exécution du programme.
- Dans le cas de la matrice **M2**, l'accélération ne dépasse pas 30. Contrairement au CPU, nous ne pouvons pas exécuter le programme sur un seul cœur du GPU. Nous ne pouvons donc pas estimer la portion séquentielle du programme sur GPU. Mais nous supposons que c'est lié à la portion séquentielle, et en comparant avec la loi de Amdhal dans le cas du CPU, l'accélération est bornée à 25. Nous dépassons 25, c'est sans doute lié à la différence du CPU et du GPU.

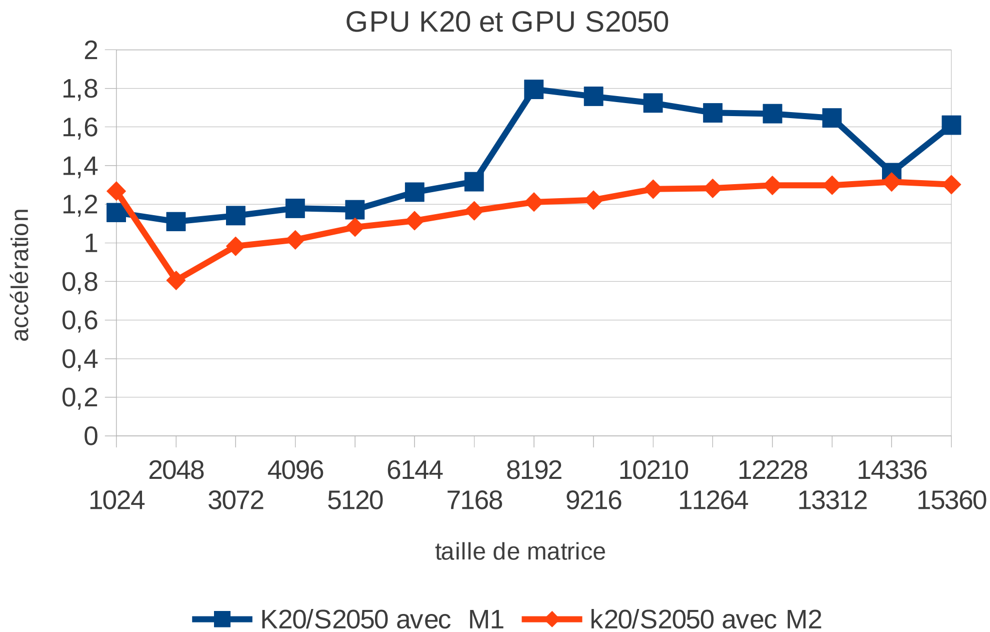

Figure 3.7

J'ai fait le même test sur un GPU (Kepler K-20) plus récent et plus puissant. La figure 3.7 présente la comparaison entre les deux GPUs de différentes générations. Nous constatons qu'il a un rapport entre 1,4 et 2,0 pour la matrice **M1**, si nous regardons le tableau des informations sur ces deux GPUs, nous trouverons qu'il y a un rapport 2,0 entre ses puissances de calcul. Pour la matrice **M2**, il y a rapport 1,2 entre les deux GPUs. Nous supposons que pour les deux GPUs, le parallélisme de l'algorithme pour **M2** est proche de l'accélération maximale prévue par la loi de Amdhal. Dans ce cas, le rapport n'est plus lié à la puissance de calcul.

Le tableau au-dessus présent les informations de ces deux GPUs :

	Cœurs/puce	Fréquence	FLOPS
Nvidia Tesla S2050	448	1,15 GHz	515 GFLOPS
Nvidia Kepler K-20	2496	0,745 GHz	1,17 TFLOPS

3.3.4. Accélération combinant CPU et GPU (MAGMA)

Les tests sont faits avec la librairie MAGMA, la figure 3.8 montre l'accélération de MAGMA, en utilisant le CPU ainsi que le GPU. Avec les deux différentes matrices, l'accélération n'est pas la même. L'accélération avec **M1** est d'environ 15, alors que celle de **M2** est d'environ 20.

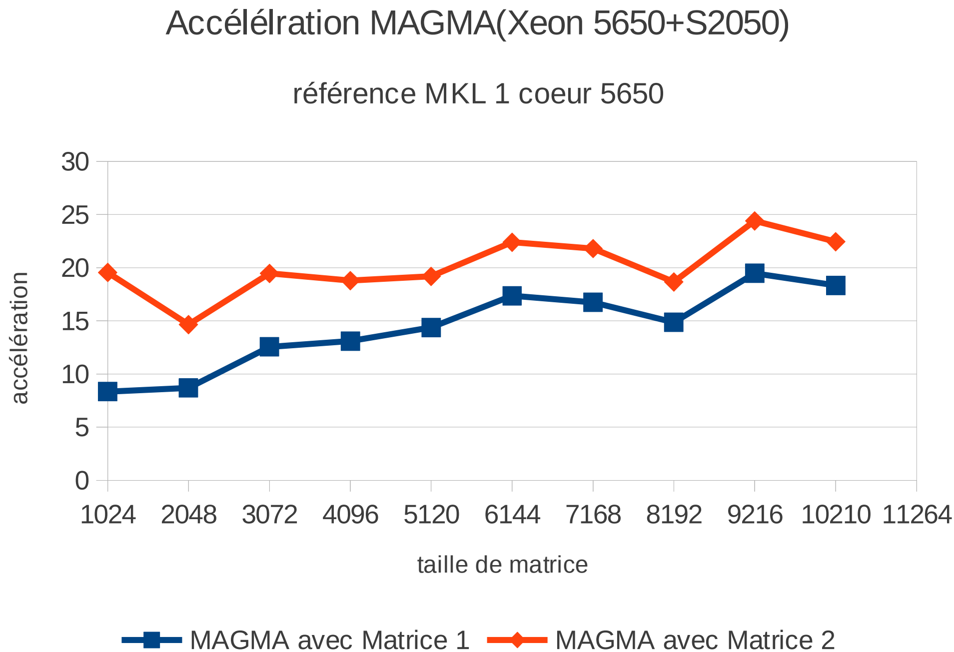

Figure 3.8

La fonction de diagonalisation dans la librairie MAGMA se fait en effet en deux étapes :

- tridiagonalisation par GPU. Le nombre d'opérations dans la première étape est finie, car il n'y pas de convergence et ce sont des opérations sur les matrices : l'exécution sur GPU est plus efficace que sur CPU.
- diagonalisation par CPU. Il s'agit d'un problème de convergence, tous les éléments hors-diagonaux doivent tendre vers zéro, il y aura donc beaucoup de tests et de branchements, le CPU est performant car il possède une unité de prédiction de branchement et d'exécution dans le désordre.

Par conséquent, l'accélération est toujours intéressante avec la librairie MAGMA.

Les tests en utilisant la librairie MAGMA sont également faits sur une grappe avec le nouveau CPU (E5-2670) et le nouveau GPU (Kepler K-20). La figure 3.9 au-dessous montre le résultat de l'accélération apportée par le nouveau matériel. Cette accélération est entre 1.5 et 2.0. Comparant avec l'accélération apporté par la librairie sur le même matériel, l'accélération apporté par nouveau matériel n'est pas élevée.

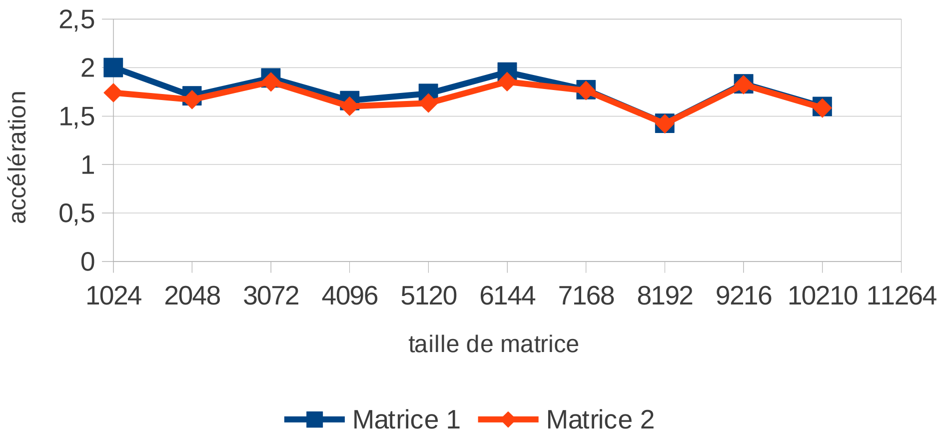

Figure 3.9

A la fin du stage, j'ai fait une comparaison entre trois matrices concrètes de taille 11000, **M1**, **M2** et une nouvelle matrice **M3** diagonale dominante[3]. Le résultat est dans la figure 3.10.

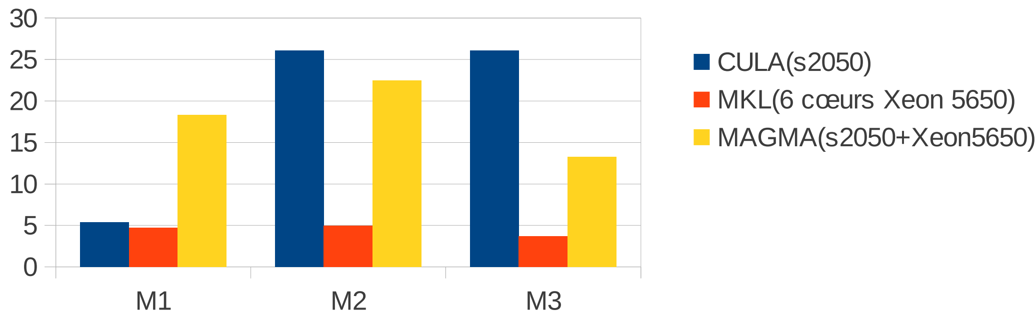

Figure 3.10

Les éléments hors-diagonaux de **M2** et de **M3** sont proches de zéro, l'algorithme QL dans la deuxième étape de diagonalisation est en effet pour mettre à zéro tous les éléments hors-diagonaux. Vue que ces éléments sont déjà proches de zéro ou bien plus petit que les diagonaux, il est donc plus facile de les mettre à zéro. Par conséquent, nous voyons bien que l'accélération de **M2** et **M3** par GPU est très élevée.

3 Diagonale dominante : les éléments hors-diagonaux sont proches de zéro ou bien plus petit que les diagonaux.

Apports personnels

Au sein de ce stage de trois mois, j'ai travaillé dans un équipe des services informatique, cela m'a apporté beaucoup de connaissances du travail réel que je n'ai jamais appris à l'école. L'ambiance du laboratoire a été agréable et amicale. J'ai également eu des occasions de pratiquer mes connaissances acquises à l'école à la pratique. Par exemple, les connaissances de l'architecture des processeurs m'ont bien aidé à l'interprétation du résultat des tests que j'ai fait.

D'ailleurs, dans mon travail j'ai comparé les différents facteurs sur l'accélération de la diagonalisation des matrices. Cela permet de trouver le meilleur choix sur l'utilisation du logiciel ou bien du matériel.

Conclusion

La diagonalisation des matrices est un travail lourd pour un processeur car la complexité est en $O(N^3)$. Une grappe de calcul permet d'accélérer le calcul grâce à du matériel performant et à des librairies spécialisées. Selon les tests effectués pendant le stage, le multicoeurs permet d'accélérer les calculs avec un rapport d'environ 5 en utilisant un CPU complet. Mais la performance du GPU est remarquable, même si l'accélération du GPU n'est pas stable (de 5 à 25), elle est au moins équivalent à celle du CPU. En général, le meilleur choix est d'utiliser la librairie MAGMA qui utilise le CPU et le GPU, car son accélération est stable et élevée. En revanche, utiliser MAGMA nécessite plus de ressources matérielles (GPU et CPU).

En plus, l'accélération apportée par les nouveau matériels est aussi considérable. La nouvelle génération de CPU porte une accélération d'environ 2, mais celle du GPU apporte une accélération faible et toujours inférieure à 2.0. Avec tous ces nouveaux matériels, la librairie MAGMA fonctionne encore mieux avec une accélération entre 1.5 et 2.0.

Le meilleur choix est donc :

1. choisir la librairie MAGMA pour avoir une plus grande accélération ;
2. opter pour du matériel récent.

Mais ce dernier est définitivement plus cher, car la librairie MAGMA est un projet open-source, elle gratuite à utiliser et les matériels nouveaux coûtent une fortune.

Bibliographie

[1] NVIDIA, CULA Programmer's Guide; Release R15 (CUDA 4.2) August 14, 2012, p4-8, 20-22.
[2] NVIDIA, CULA Reference Manual; Release R17(CUDA5.0) May 07, 2013, p4-9, 102-104.
[3] Intel, Math Kernel Library Reference Manual; Document Number: 630813-051US
MKL 10.3 Update 11, p984-986.
[4] Agner Fog (Copenhagen University College of Engineering), The microarchitecture of Intel, AMD and VIA CPUs; 2012-02-29, p8-34.
[5] William H.P., Saul A.T., William T.V.et Brian P.F., Numerical Recipies in C-The Art of Scientific Computing; 2nd edition(CAMBRIDGE UNIVERSITY PRESS), pp456-481.
[6] François Boulier (Université de Lille 1), Calcul Numérique; 17 mai, 2013 p54-66, 88-98.
[7] NVIDIA, CUDA C Programming Guide; PG-02829-001_v5.0, October 2012, p2-5, 63-65
[8] Jud Porter, CS264 Lab 3 CUDA Basics(cs264_lab3.pdf). p2-4
[9] Taxonomie de Flynn, sur wikipedia, dernière modification le 1 juin 2013, URL https://http://fr.wikipedia.org/wiki/Taxinomie_de_Flynn
[10] A. S. Householder, Householder Transformation, wikipedia, dernière modification le 9 juillet 2013 URL https://en.wikipedia.org/wiki/Householder_transformation
[11] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flanney, Numerical Recipies in Fortran 77. The art of scientific computing, Cambridge University Press, Cambridge(internet version),1992
[12] G. Lance, Numerical Methods for High Speed Computes, Iliffe & Sons LTD, London, 1960

Annexe

Script pour lancer le job sur la grappe :

```
#!/bin/bash
############### directives SGE ###############
#$ -N test
#$ -m base
#$ -M kun.song@u-psud.fr
#$ -l h_rt=05:00:00
#$ -cwd
#$ -j y
#$ -q cuda-12G300m
#$ -l gpu=1

############### commandes shell / programmes ###############

date
############### charger les modules / positionner les variables
###############

module load cuda/5.0
module load intel/13.1.2
module load cula/dense/R16a
export OMP_NUM_THREADS=6

############### verifier les modules / la valeur des variables
###############

echo "OMP_NUM_THREADS: "$OMP_NUM_THREADS
module list

echo "running program"

test

echo "program terminate"

date
```

Le code principal :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>
#include <sys/time.h>
#include <cula_lapack.h>
#include <cula_lapack_device.h>
#include <cuda_runtime.h>
#include "magma.h"
#include "magma_lapack.h"
#define NB_BOUCLE 1
#define TAILLE_MAX_MAGMA 11000

void checkStatus(culaStatus status)
{
   char buf[256];

   if(!status)
      return;

   culaGetErrorInfoString(status, culaGetErrorInfo(), buf, sizeof(buf));
   printf("%s\n", buf);

   culaShutdown();
   exit(EXIT_FAILURE);
}

void checkCudaError(cudaError_t err)
{
   if(!err)
      return;

   printf("%s\n", cudaGetErrorString(err));

   culaShutdown();
   exit(EXIT_FAILURE);
}

double getHighResolutionTime(void)
{
   struct timeval tod;

   gettimeofday(&tod, NULL);
```

```
    double time_seconds = (double) tod.tv_sec + ((double) tod.tv_usec / 1000000.0);
    return time_seconds;
}

void fillRandomDouble(int m, int n, double* A,double min, double max)
{
    int iseed[4];
    iseed[0] = 2001;
    iseed[1] = 2003;
    iseed[2] = 2005;
    iseed[3] = 2007;
    int i =0;
    int j =0;
    for( i;i<m;i++) {
      for( j;j<n;j++) {
        double random;
        int ii=1;
        dlarnv_(&ii,iseed,&ii,&random);
        A[i*m+j] = random;
        A[j*n+i] = random;
      }
    }
}
/*
void fillRandomDouble(int m, int n, double* a, double min, double max)
{
    int i, j;

    srand(1);

    for (j=0; j<m; j++)
    {
        for (i=0; i<n; i++)
        {
            a[j*n+i] = min + (max-min) * rand()/RAND_MAX;
        }
    }
}
*/
int main()
{
        int size=1024;
        int max = 29000;
        double timer_start;
        double timer_stop;
```

```
	double time_cula,time_mkl, time_magma;

	// point to host memory
	double* A = NULL;
	double* vals =NULL;

	// point to device memory
	double* Ad = NULL;
	double* valsd = NULL;

	double* A_stock =NULL;

	//variables pour MKL
	int lwork;
	int info;
	double* work;

	//variable pour MAGMA
	double* work_magma;
	double* wa;
	magma_int_t lwork_magma;
	int info_magma;
	int* iwork;
	int liwork;

	//Initialisation du CULA
	cudaError_t err;
	culaStatus status;
	status = culaInitialize();
	checkStatus(status);
	printf("  Matrix size |CULA_time(sec)| MKL_time(sec)| MAGMA_time(sec) \n");
	fflush(stdout);

while(size<=max){

	//Mémoire dynamique
	A_stock = (double*)malloc(size*size*sizeof(double));    //Matrice initiale: ne dois pas etre
changer

	//Obtenir l'espace du travail
	//------------------CULA&MAGMA----------------
	err = cudaMalloc((void**)&Ad,size*size*sizeof(double));          //Mémoire dynamique sur GPU:
size*size
	checkCudaError(err);
	err = cudaMalloc((void**)&valsd,size*sizeof(double));   //Mémoire dynamique sur GPU: 1*size
	checkCudaError(err);
```

```
        //--------------------MKL--------------------
        A = (double*) malloc(size*size*sizeof(double));
        vals = (double*)malloc(size*sizeof(double));
        lwork=-1;
        work = (double*)malloc(2*sizeof(double));
        lapackf77_dsyev("V","U",&size,A,&size,vals,work,&lwork,&info);
        lwork = (int)work[0];
        free(work);
        work = (double*)malloc(lwork*sizeof(double));

        //------------------MAGMA--------------------
        if(size<TAILLE_MAX_MAGMA){
                wa = (double*)malloc(size*size*sizeof(double));
                work_magma = (double*)malloc(1*sizeof(double));
                iwork = (int*)malloc(1*sizeof(int));
                magma_dsyevd_gpu('V','U',size,A,size,vals,wa,size,work_magma,-1,iwork,-1,&info);
                lwork_magma = (magma_int_t)work_magma[0];
                liwork = iwork[0];
                free(work_magma);
                free(iwork);
                work_magma = (double*)malloc(lwork_magma*sizeof(double));
                iwork = (int*)malloc(liwork*sizeof(int));
         }
//      printf("lapack lwork = %d,  magma lwork = %d \n", lwork, lwork_magma);

  for(int count=0;count<NB_BOUCLE;count++){

        //CULA
                //Transfert de donnée Host to Device
                timer_start = getHighResolutionTime();
                err = cudaMemcpy(Ad,A_stock,size*size*sizeof(double),cudaMemcpyHostToDevice);
                checkCudaError(err);
                //Diagoanlisation
                status = culaDeviceDsyev('V','U',size,Ad,size,valsd);      //device memory
                checkStatus(status);                    //check le fonctionnement du dsyev
                //Transfert de donnée Device to Host
                err = cudaMemcpy(A,Ad,size*size*sizeof(double),cudaMemcpyDeviceToHost);
                checkCudaError(err);
                err = cudaMemcpy(vals,valsd,size*sizeof(double),cudaMemcpyDeviceToHost);
                checkCudaError(err);
                timer_stop = getHighResolutionTime();
                time_cula = timer_stop-timer_start;

        //MKL
                timer_start = getHighResolutionTime();
```

```
			memcpy(A, A_stock, size*size*sizeof(double));
			lapackf77_dsyev("V","U",&size,A,&size,vals,work,&lwork,&info);
			timer_stop = getHighResolutionTime();
			time_mkl = timer_stop-timer_start;
			if(info>0)printf("Probleme de convergence\n");

		//MAGMA
			timer_start = getHighResolutionTime();
			if(size < TAILLE_MAX_MAGMA){
			cudaMemcpy(Ad,A_stock,size*size*sizeof(double),cudaMemcpyHostToDevice);


magma_dsyevd_gpu('V','U',size,Ad,size,vals,wa,size,work_magma,lwork_magma,iwork,liwork,&info);
			if(info>0)printf("Probleme de convergence\n");
			}
			timer_stop = getHighResolutionTime();
			time_magma = timer_stop-timer_start;

			printf("%13d | %12.3f | %12.3f | %12.3f|\n", size,time_cula,time_mkl,time_magma);
			fflush(stdout);
	}
		free(A);
		cudaFree(Ad);
		cudaFree(valsd);
		free(vals);
		free(work);
		if(size < TAILLE_MAX_MAGMA){
			free(iwork);
			free(work_magma);
			free(wa);
		}
		size+=1024;
		free(A_stock);
	}
		culaShutdown();
		return EXIT_SUCCESS;
}
```