Université Paris-Sud

**Thèse**

pour obtenir le grade de

**Docteur en Informatique de l'Université Paris-Sud**

préparée au Laboratoire de Recherche en Informatique
dans le cadre de *l'École Doctorale 427 : Informatique Paris-Sud*

présentée et soutenue publiquement

par

**Adrien Rémy**

8 Juillet 2015

---

# Résoudre des systèmes linéaires denses sur des architectures composées de processeurs multicœurs et d'accélérateurs

---

***Directeur de thèse :***
Mr. Marc Baboulin

---

**Jury**

| | | |
|---|---|---|
| M. Stef Graillat | Rapporteur | Professeur Université Paris 6 |
| M. Paulo Beleza Vasconcelos | Rapporteur | Professeur Universidade do Porto |
| M. Philippe Langlois | Examinateur | Professeur Unviversité de Perpignan |
| M. Marc Baboulin | Directeur de thèse | Professeur Université Paris Sud |
| M. Nicolas M.Thiéry | Examinateur | Professeur Université Paris Sud |

**Résoudre des systèmes linéaires denses sur des architectures composées de processeurs multicœurs et d'accélerateurs.**

**Résumé :**

Dans cette thèse de doctorat, nous étudions des algorithmes et des implémentations pour accélérer la résolution de systèmes linéaires denses en utilisant des architectures composées de processeurs multicœurs et d'accélérateurs. Nous nous concentrons sur des méthodes basées sur la factorisation LU. Le développement de notre code s'est fait dans le contexte de la bibliothèque MAGMA.

Tout d'abord nous étudions différents solveurs CPU/GPU hybrides basés sur la factorisation LU. Ceux-ci visent à réduire le surcoût de communication dû au pivotage. Le premier est basé sur une stratégie de pivotage dite "communication avoiding" (CALU) alors que le deuxième utilise un préconditionnement aléatoire du système original pour éviter de pivoter (RBT). Nous montrons que ces deux méthodes surpassent le solveur utilisant la factorisation LU avec pivotage partiel quand elles sont utilisées sur des architectures hybrides multicœurs/GPUs.

Ensuite nous développons des solveurs utilisant des techniques de randomisation appliquées sur des architectures hybrides utilisant des GPU Nvidia ou des coprocesseurs Intel Xeon Phi. Avec cette méthode, nous pouvons éviter l'important surcoût dû pivotage tout en restant stable numériquement dans la plupart des cas. L'architecture hautement parallèle de ces accélérateurs nous permet d'effectuer la randomisation de notre système linéaire à un coût de calcul très faible par rapport à la durée de la factorisation.

Finalement, nous étudions l'impact d'accès mémoire non uniformes (NUMA) sur la résolution de systèmes linéaires denses en utilisant un algorithme de factorisation LU. En particulier, nous illustrons comment un placement approprié des processus légers et des données sur une architecture NUMA peut améliorer les performances pour la factorisation du panel et accélérer de manière conséquente la factorisation LU globale. Nous montrons comment ces placements peuvent améliorer les performances quand ils sont appliqués à des solveurs hybrides multicœurs/GPU.

**Mots clés :** Systèmes linéaires denses, factorisation LU, bibliothèques logicielles pour l'algèbre linéaire dense, bibliothèque MAGMA, calcul hybride multicœur/GPU, processeurs graphiques, Intel Xeon Phi, ccNUMA, communication-avoiding, randomisation, placement des processus légers.

# Table des matières

# Table des figures

# Introduction

Dans de nombreuses applications informatiques, la tâche la plus couteuse en temps et en ressources consiste à résoudre un système d'équations linéaires de la forme $Ax = b$. Ensuite, le défi majeur consiste à calculer une solution $x$ aussi vite que possible tout en conservant une précision satisfaisante. Le but principal de cette thèse est d'étudier des solutions pour accélérer des solveurs linéaires denses en utilisant les architectures parallèles actuelles, qui comprennent souvent des accélérateurs.

Dans ce travail, nous nous concentrons sur la décomposition LU de matrices denses. Nous proposons différents algorithmes et implémentations afin d'accélérer cette factorisation. Ces solveurs denses peuvent être utilisés pour résoudre des systèmes d'équations directement ou en tant que noyaux pour des solveurs creux, directs ou itératifs.

Pour résoudre ces problèmes aussi rapidement que possible, les algorithmes doivent être adaptés pour être efficaces et évolutifs sur les machines parallèles actuelles. En outre, les implémentations de solveur doivent être adaptées aux caractéristiques architecturales de ces systèmes parallèles. Dans notre travail, nous prenons en compte plusieurs caractéristiques des ordinateurs parallèles : l'utilisation d'accélérateurs tels que les GPGPUs et les coprocesseurs Intel Xeon Phi, le parallélisme SIMD nécessaire pour programmer efficacement ces accélérateurs, et des architectures à accès non uniforme de la mémoire (NUMA) utilisés dans des ordinateurs multi-socket à mémoire partagée. Ces architectures parallèles fournissent une puissance de calcul de plus en plus grande et ont besoin de certaines exigences spéciales pour être exploitées efficacement.

Nos développements de code sont faites dans le cadre de la bibliothèque MAGMA qui est une bibliothèque d'algèbre linéaire numérique dense, conçu pour les architectures hybrides avec accélérateur. MAGMA implémente les algorithmes de la bibliothèque LAPACK largement utilisé.

Nous proposons différents algorithmes et implémentations de résoudre de grands systèmes linéaires denses d'équations via la factorisation LU et une partie du code résultant a été intégrée dans la bibliothèque MAGMA.

Dans le Chapitre 1, nous présentons un aperçu du contexte scientifique de cette thèse. Dans le Chapitre 2, nous examinons différentes méthodes de pivotement dans l'algorithme de factorisation LU, lorsqu'il est effectué sur une architecture hybride combinant les processeurs multicœurs et GPUs. Dans le Chapitre **??**, nous nous concentrons sur l'utilisation de la méthode appelée "Random Butterfly Transformation" (RBT), dans les solveurs de systèmes linéaires utilisant des accélérateurs. Dans le Chapitre 4, nous étudions et comparons différentes méthodes pour utiliser efficacement les plates-formes à accès non uniforme de la mémoire (NUMA) dans le contexte des bibliothèques d'algèbre linéaire denses. Nous donnons enfin quelques remarques finales et discutons de certaines pistes de recherche en cours ou possibles.

Chapitre 1

# Algorithmes, architectures et bibliothèques

## 1.1 Introduction

Résolution de systèmes d'équations linéaires a toujours été une approche utilisée pour résoudre les problèmes de la vie réelle dans de nombreux domaines tels que la physique, la biologie, la géométrie ... Il y a quatre mille ans, Babyloniens avaient déjà trouvé comment résoudreun système linéaire $2 \times 2$ [1].

Au début du $19^e$ siècle, Carl Gauss a développé une méthode appelée " élimination de Gauss " dans le but de résoudre des systèmes d'équations linéaires.

Le calcul matriciel prit un tournant autour de la Seconde Guerre mondiale avec l'apparition des premiers ordinateurs. Cela a permis aux méthodes d'algèbre linéaire de résoudre rapidement et plus précisément de grands systèmes d'équations. Notez que l'élimination de Gauss est encore la méthode la plus connue pour résoudre un système d'équations linéaires [2].

Durant les années 1950, les transistors,beaucoup plus petits et plus fiable, ont remplacé les tubes à vide dans les architectures. Pendant cette période, des percées technologiques telles que la création du microcode ou la mise en œuvre du premier langage de haut niveau : Fortran, ont contribué à répandre l'utilisation des ordinateurs dans les applications scientifiques et commerciales.

En 1971, Intel a publié le 4004, le premier microprocesseur commercial, réunissant tous les éléments d'un processeur dans une seule puce. Ensuite, les microprocesseurs suivront la prédiction de Gordon Moore : leur complexité doublera chaque année [3].

Depuis lors, l'évolution architecturale a aidé à construire des processeurs plus efficaces.

Dans les supercalculateurs, le nombre de processeurs utilisés en parallèle augmenté à des milliers et des nouvelles solutions comme les accélérateurs (GPGPU ou Intel Xeon Phi) sont développés pour améliorer les performances.

Pour profiter de ces développements architecturaux, des bibliothèques de logiciels ont été publiées pour donner à l'utilisateur la possibilité d'effectuer efficacement des calculs d'algèbre linéaire sur ces architectures.

Dans cette thèse, nous proposons des solutions pour utiliser ou à améliorer certaines des bibliothèques actuelles d'algèbre linéaire du domaine public afin qu'elles exploitent au mieux les possibilités des architectures parallèles modernes.

Ce chapitre présente le contexte de notre travail.

## 1.2 Résolution de systèmes linéaires denses

Les grands systèmes linéaires denses sont rencontrés dans différents domaines scientifiques tels que : l'électromagnétisme, la méchanique des fluides, la méchanique quantique [4], la tomographie, la l'évaluation des supercalculateurs [5], etc ...

En outre, les routines pour les systèmes linéaires denses sont couramment utilisés comme noyaux dans des méthodes plus générales pour résoudre des systèmes linéaires creux en utilisant des méthodes directes ou itératives [6, 7, 8].

La résolution de ces problèmes consiste généralement à résoudre un système d'équations linéaires : $Ax = b$. Pour résoudre de tels systèmes, il ya deux classes de méthodes : les méthodes directes et itératives. Dans notre travail, nous sommes préoccupés par les matrices denses et nous nous concentrons sur des méthodes directes. Les méthodes directes impliquent généralement la décomposition des matrices suivies par la résolution successive de systèmes triangulaires. Différentes méthodes de décomposition existent comme factorisation QR, Cholesky, $\mathrm{LDL}^{\mathrm{T}}$ ou LU [9].

Dans la suite, nous nous concentrons sur la décomposition de LU.

## 1.3 factorisation LU

### 1.3.1 Élimination de Gauss

Si $A$ est carrée, dense et non structurée, la méthode généralement choisie pour résoudre $Ax = b$ est l'élimination de Gauss. L'élimination de Gauss est constitué d'une séquence d'opérations de base sur les lignes de la matrice pour remplir les coefficients en dessous de la diagonale avec des zéros rendant la matrice triangulaire supérieure, et permettant ainsi au système d'être facilement résolu. La factorisation LU est une forme modifiée de l'élimination de Gauss où la matrice $A$ est exprimée comme un produit $LU$, avec $L$ une matrice unitaire triangulaire inférieure et $U$ une matrice triangulaire supérieure.

Ensuite, le système est facile à résoudre par la substitution avant pour $L$ et la substitution arrière pour $U$ :

$Ly = b, Ux = y \Rightarrow Ax = LUx = Ly = b$. Cela nécessite $\mathcal{O}(n^2)$ flops.

### 1.3.2 Le problème du pivotage

Avec la méthode décrite précédemment, si un 0 se trouve sur la diagonale de la matrice, une division par zéro se produit. Aussi, si des éléments de petite ampleur sont sur la diagonale, les valeurs sur les facteurs triangulaires vont croître de manière significative. Par conséquent les erreurs d'arrondi sont inévitables.

Pour cette raison, nous passons le plus grand élément de la colonne sur la diagonale en permutant les lignes. Cette méthode est appelée pivotage partiel. D'autres stratégies de pivotage parallèle seront abordées plus tard.

Même si pivotage augmente la stabilité et ne nécessite pas d'opérations flottante supplémentaires, il implique des mouvements irréguliers de données. Si $n$ est la taille de la matrice, le pivotage partiel implique $\mathcal{O}(n^2)$ comparaisons.

### 1.3.3 Factorisation LU par bloc

Pour permettre le parallélisme et une utilisation plus optimale de mémoire hiérarchique, nous pouvons organiser la factorisation LU de sorte que les multiplications de matrices deviennent les opérations dominantes. Pour ce faire nous effectuons la factorisation par bloc

.

Les trois algorithmes communs pour la factorisation LU par bloc sont : left-looking LU, right-looking LU et Crout LU.

Nous décrivons ici la factorisation LU par bloc vers la droite (right-looking).

On calcule la factorisation d'une matrice $A$ de taille $m \times n$. La matrice $A$ est partitionnée comme suit,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

où $A_{11}$est de taille $b \times b$, $A_{21}$ de taille $(m-b) \times b$, $A_{12}$ de taille $b \times (n-b)$ and $A_{22}$ de taille $(m-b) \times (n-b)$.

1. La factorisation LU du panel $\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ est effectuée par élimination de Gauss avec pivotage pariel (GEPP) à $\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$
2. Nous appliquons les permutations au reste de la matrice : $\begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}$
3. On calcule $U_{12}$ par résolution triangulaire $U_{12} = L_{11}^{-1} A_{12}$
4. On met à jour $A_{22}$ : $A_{22} = A_{22} - L_{21} U_{12}$
5. On applique la même méthode à $A_{22}$

### 1.3.4 Technique dite : "Communication avoiding"

Sur les architectures parallèles, à la recherche du pivot dans le bloc de décomposition LU génère une grande quantité de mouvements de données. En réduisant les communications à leurs minimum, il est possible d'obtenir de meilleures performances.

Dans [10],les auteurs ont proposé ce que l'on appelle les algorithmes communication avoiding. Nous nous concentrons sur l'algorithme CALU décrit dans [10, 11, 12, 13].

Cet algorithme propose une nouvelle stratégie pour la sélection du pivot minimisant les communications [13].

La particularité de cet algorithme est principalement la factorisation du panel réalisée avec l'algorithm Tall Skinny LU (TSLU) décrit dans [10, 11, 12, 13].

### 1.3.5 Random Butterfly Transformation (RBT)

Pour éviter de pivoter il est possible de randomiser la matrice, puis de la factoriser sans pivoter. La méthode pour randomiser a été décrite par Parker dans [14, 15]. Elle consiste a multiplier une matrice $A$ en $A_r = U^T AV$, où $U$ et $V$ sont des matrices "recursive butterfly". Nous alors pouvons factoriser $A_r$ sans pivoter. Résoudre le système linéaire général suit ces étapes :

1. Une matrice aléatoire $A_r$ est calculée : $A_r = U^T AV$.
2. $A_r$ est décomposée en $LU$ sans pivoter.
3. Le système est résolu en utilisant : $A_r y = U^T b$.
4. La solution est $x = Vy$.

Du fait de leur structure creuse,le coût de calcul pour appliquer la transformation multiplicatif $(U^T AV)$ est $4dn^2$ flops quand $U$ et $V$ sont des matrices récursives "random butterfly" de profondeur $d$.

## 1.4 Architectures parallèles

Dans le domaine de l'informatique moderne à haute performance (HPC), le parallélisme est une question cruciale. Dans cette section, nous présentons un aperçu des solutions architecturales actuelles développées pour le HPC.

Depuis 1945, l'architecture de Von Neumann est utilisé comme modèle pour la construction d'ordinateurs [16]. Les supercalculateurs des années 70 aux années 90 ont été la plupart du temps conçus avec des processeurs vectoriels (par exemple, les plates-formes Cray). Différents types d'architectures parallèles existent. La taxonomie de Flynn propose quatre catégories d'architectures [17] : SISD, MISD, SIMD, MIMD. Dans ce qui suit, nous présentons les éléments architecturaux que nous avons utilisé au cours de cette thèse.

### 1.4.1 SIMD extensions

Les extensions Single Instruction Multiple Data (SIMD) également appelés extensions multimédias ont été introduits dans les processeurs de la fin des années 90. Ils fournissent des registres spéciaux qui peuvent stocker des données multiples. Ensuite, les instructions peuvent être appliqués à ces entrées, le traitement de chaque élément de données à l'intérieur d'un registre en même temps, ce qui crée du parallélisme.

Le coprocesseur Intel Xeon Phi d'aujourd'hui utilise l'AVX-512 qui peut traiter 8 nombres à virgule flottante double précision ou 16 simple précision en même temps. Dans le domaine de l'algèbre linéaire dense, exploiter le parallélisme offert par les extensions SIMD est essentiel pour obtenir des performances optimales.

### 1.4.2 Processeurs multicœurs

Un moyen d'augmenter les performances d'un processeur est d'augmenter sa fréquence mais pose des problèmes de dissipation de chaleur et de consommation électrique. La solution choisie par les constructeurs pour continuer d'améliorer les processeurs fût d'introduire les processeurs multicœurs [18].

IBM a développé le processeur POWER4 en 2001, le première processeur multicœur. AMD et Intel ont produit les premiers multiprocesseurs x86 en 2006 avec l'AMD Opteron et les architectures Intel Core. Depuis lors, le processeur multicœur est devenu un standard pour les ordinateurs de bureau, des serveurs ou des plates-formes mobiles.

En dehors de l'architecture des cœurs utilisés, un processeur multicœur peut être décrit par [19] :

— Le nombre de cœurs sur la puce,
— le nombre de niveaux de mémoire cache,
— la quantité de mémoire cache partagée.

### 1.4.3 Architectures à accès de mémoire non uniforme (NUMA)

Depuis 1968, des ordinateurs ont été construit avec plusieurs processeurs pour permettre un traitement parallèle [20]. Ces machines sont composées de multiples processeurs identiques et une seule mémoire principale partagée (UMA). À un certain point, la congestion du bus mémoire devient un problème pour la performance [21]. Une autre solution a été développée dans les années 90 afin de surmonter ces limites : les architectures à accès mémoire non uniforme (NUMA).

Systèmes NUMA sont généralement composés de plusieurs processeurs multicœurs et leurs banques de mémoire. Chaque cœur de processeur est capable d'accéder à toute partie de la mémoire. Le coût de l'accès à la mémoire sera différent, en fonction de l'emplacement des données demandées, mais l'ensemble de la mémoire est partagée de façon transparente pour le développeur.

Pour obtenir un bon passage à l'échelle, les programmes parallèles sur les systèmes NUMA doivent faire bon usage de la mémoire cache pour minimiser accès à la mémoire et à assurer une bonne localité des données [19].

### 1.4.4 Calcul généraliste sur processeurs graphiques (GPGPU)

Les unités de traitement graphique (GPU) sont des circuits électroniques spécialisés conçus pour créer ou accélérer la génération d'images à afficher. Depuis le début des années 2000, les GPUs ont commencés a être utilisés pour réaliser des calculs matriciels. En 2007, Nvidia a publié le Unified Device Architecture Compute (CUDA) une plate-forme de programmation [22] fournissant un ensemble d'instructions virtuel, permettant le développement d'applications à usage général.

Le GPGPU est devenu monnaie courante dans le HPC et est souvent utilisé dans les architectures des superordinateurs. Les GPUs offrent une grande capacité de calcul à un faible coût et une bonne efficacité énergétique. Sur les dix plus puissants

supercalculateurs dans le dernier classement TOP500 [23] (Novembre 2014), trois utilisaient des GPGPUs.

L'inconvénient des GPGPUs vient de leur modèle de programmation hybride qui ne permet pas autant d'efficacité que des architectures utilisant des processeurs uniquement, en raison de sa nature purement SIMD et des limitations de bande passante du PCI-Express.

Leur modèle d'architecture hautement parallèle fait du GPGPU une solution appropriée pour les calculs matriciels et les programmes d'algèbre linéaire denses. Toutefois, programmer de manière efficace sur des architectures à base de GPU est un défi critique pour le calcul haute performance. Dans cette thèse, nous avons montré quelques solutions pour résoudre efficacement les systèmes linéaires denses, en utilisant des GPUs comme accélérateurs.

### 1.4.5 Accelerateurs Intel Xeon Phi

En 2010, Intel a annoncé leur architecture moult cœur intégrée (Intel MIC), une architecture de coprocesseur hautement parallèle constitué de plusieurs processeurs x86 et sa propre mémoire intégrée de GDDR5 [24].

La pierre angulaire de la performance du Xeon Phi est la presence d'unités de traitement vectoriel (VPU) dans chaque cœur, utilisant des registres SIMD de 512 bits [25].

En Novembre 2014, deux des dix plus puissants supercalculateurs utilisaient des accélérateurs Intel Xeon Phi, y compris le numéro 1 du classement Top500.

Pour les programmes avancés avec un parallélisme non trivial, atteindre de hautes performances avec le Xeon Phi peut être difficile. Dans les cas pratiques, des optimisations avancées de bas niveau tels que du code SIMD écrit à la main sont nécessaires [26].

## 1.5 Bibliothèques d'algèbre linéaire numérique pour les matrices denses

L'évolution de l'architecture des ordinateurs a été suivie par les bibliothèques logicielles. Une architecture différente nécessite une mise en œuvre différente pour être en mesure d'utiliser au mieux les capacités de la machine. En algèbre linéaire dense, l'efficacité de calcul est un défi majeur. Par conséquent, une mise en œuvre optimale de ces programmes est indispensable.

### 1.5.1 Les bibliothèques historiques

Au milieu des années 60, IBM a distribué le "Scientific Subroutine Package" [27], en 1974 Garbow a publié EISPACK [28] pour calculer des valeurs et vecteurs propres. BLAS a été développé entre 1973 et 1977 [29].

La bibliothèque de Linpack a proposé en 1979, un ensemble de sous-programmes conçus pour les supercalculateurs afin de résoudre des équations linéaires et des

problèmes moindres carrés [30].

Sorti en Février 1992, LAPACK [31] remplace Linpack et eispack et réalise de meilleures performances. LAPACK se concentre sur la résolution : systèmes d'équations linéaires, des problèmes de moindres carrés, des problèmes de valeurs propres et des problèmes de valeur singulière.

La plupart des bibliothèques d'algèbre linéaire numérique développées par la suite sont basées sur BLAS et LAPACK.

### 1.5.2 Implémentations parallèles

Certaines bibliothèques de fournisseurs tels que ACML [32] pour les processeurs AMD et MKL [33] pour les processeurs Intel fournissent des implémentations optimisées de BLAS et LAPACK pour leurs processeurs. Ces optimisations incluent des fonctions multithread et vectorisés. Des projets open source existent également tels que ATLAS [34], Goto BLAS [35] ou OpenBlas [36]. Pour GPU, NVIDIA propose Cublas [37], une implémentation de BLAS pour CUDA. ScaLAPACK [38] (Scalable LAPACK) est une implémentation de LAPACK pour les architectures distribuées. PLASMA est une bibliothèque logicielle conçue pour être efficace sur des processeurs multicœurs homogènes et systèmes multi-sockets de processeurs multicœurs [39]. PLASMA utilise des noyaux BLAS pour ses calculs internes, est basée sur des algorithmes tuillés et utilise l'ordonnanceur dynamique de tâches QUARK [40].

### 1.5.3 La bibliothèque MAGMA

De même pour Lapack, MAGMA [1] [41, 42, 43], est construit comme étant un effort communautaire, incorporant les derniers développements dans les algorithmes hybrides. L'objectif de ces efforts est de reconcevoir les algorithmes d'algèbre linéaire denses de LAPACK afin d'exploiter pleinement la puissance des systèmes hétérogènes actuels composés de processeurs et plusieurs accélérateurs. La bibliothèque MAGMA existe en trois versions : une pour les GPU NVIDIA CUDA aide, une utilisant OpenCL et une dédiée aux accélérateurs Intel Xeon Phi.

Dans cette thèse, nous avons utilisé MAGMA comme un cadre pour développer de nouveaux solveurs. Certains de ces solveurs ont été inclus dans la dernière version de MAGMA pour GPU et pour Intel Xeon Phi.

1. Matrix Algebra on GPU and Multicore Architectures, `http://icl.cs.utk.edu/magma/`

Chapitre 2

# Algorithmes hybrides CPU/GPU pour la factorisation LU

## 2.1 Algèbre linéaire dense sur des machines multicœurs accélérées

Le chapitre est organisé comme suit. Tout d'abord, nous décrivons dans la Section 2.2 comment la bibliothèque MAGMA implémente l'algorithme de LU avec pivotage partiel sur des architectures hybrides. Puis, dans la section 2.3 nous introduisons "tournament pivoting", une stratégie sur la base de la CALU que nous avons adapté spécifiquement pour les architectures CPU / GPU. Enfin, nous proposons dans la section 2.4 des résultats de performance pour le panel et la factorisation hybride. Des remarques finales sont donnés dans la Section 2.5.

## 2.2 Implémentations de la factorisation LU dans MAGMA

Illustrons comment l'approche hybride multicœur + GPU peut être appliquée à la factorisation LU, en décrivant l'algorithme tel qu'il est implémenté dans la bibliothèque MAGMA. La méthode est basée sur la division du calcul comme le montre la figure 2.1. La matrice initiale a été téléchargé vers le GPU et nous décrivons dans l'algorithme 1 l'itération courante :

**Algorithm 1** Iteration pour la factorisation LU avec MAGMA

1: Le panel courant (1) est envoyé vers le CPU.
2: (1) est factorisé par le CPU avec GEPP et le resultat est renvoyé GPU.
3: Le GPU met à jours (2) (prochain panel).
4: Le panel mis à jour (2) est renvoyé au CPU pour être factorisé pendant que le GPU met à jour le reste de la matrice (3).

La technique consistant a factoriser (2) tout en mettant à jour (3) est souvent appelée *look-ahead* [44]. Dans la section 2.3 nous utilisons une stratégie de pivotage différente qui se révèle être très efficace pour la factorisation du panel.

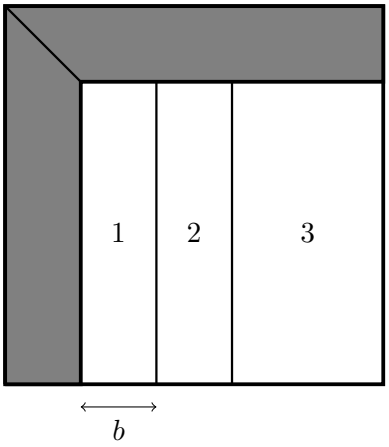



FIGURE 2.1 – Découpage en bloc dans la factorisation LU hybride

## 2.3 Implémentation hybride de "tournament pivoting LU"

De nouvelles approches ont été récemment proposées pour minimiser les communications dues au pivotage. Parmi elles, des algorithmes dits "Communication Avoiding" ont été introduits pour les machines à mémoire distribuée [13] et pour les architectures multicœurs [12]. Ces algorithmes sont efficaces en ce sens qu'ils réduisent considérablement les communications tout en étant stable en pratique. Dans CALU la factorisation du panel est effectuée avec l'algorithme TSLU.

l'algorithme CALU peut être représenté par un graphe acyclique orienté (DAG) où les noeuds sont des tâches élémentaires qui opèrent sur un ou plusieurs blocs de taille $b \times b$ et où arêtes représentent les dépendances entre eux. Dans la Figure 2.2 les tâches noires représentent la factorisation du panel via TSLU et les tâches grises représentent la mise à jour de la sous-matrice résultante.

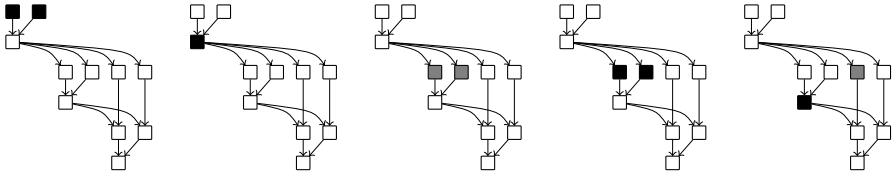


FIGURE 2.2 – Exemple de factorisation LU asynchrone utilisant CALU multithreadé (2 threads, 3 blocs de colonnes) sur CPU

Dans ce qui suit, nous décrivons une méthode qui améliore encore l'algorithme de MAGMA en minimisant les communications associées avec les factorisations des panels.

A chaque étape, un panel de taille $B$ est factorisé sur le CPU en appliquant CALU à une matrice rectangulaire et la mise à jour de la sous-matrice résultante est

effectué par le GPU. CALU factorise le panel en divisant le bloc initial de colonnes en blocs plus petits contenant $b$ colonnes qui sont factorisées de manière itérative en utilisant TSLU. Ainsi, la factorisation du panel est considéré comme une variante de l'algorithme au premier niveau dans lequel on factorise une matrice rectangulaire en utilisant uniquement le CPU.

La figure2.3 représente un exemple de la factorisation d'une matrice. Nous considérons que la matrice est d'abord stocké sur le GPU. Les tâches noires représentent la factorisation du panel avec CALU multithreadé et les Les tâches grises représentent la mise à jour de la sous-matrice résultante par GPU.

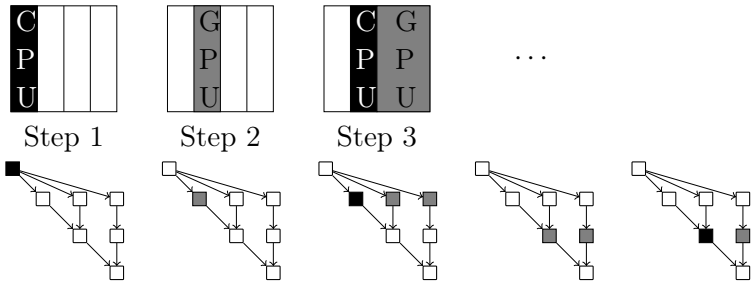



FIGURE 2.3 – Factorisation CALU hybride (4 panels).

## 2.4 Comparaison des performances

Dans cette section, nous présentons les résultats de performance pour les algorithmes décrits dans les sections 2.2 et 2.3. Ces expériences numériques ont été effectuées en utilisant un système hybride CPU / GPU.

### 2.4.1 Performance pour la factorisation du panel

Comme décrit dans la section 2.3, la factorisation du panel est effectuée par le CPU pendant que la mise à jour de la sous-matrice est exécutée par le processeur graphique. Nous évaluons spécifiquement les performances pour la phase panel de la factorisation dans une factorisation LU. Cette performance est mesurée en additionnant le nombre total de flops exécutés en factorisant successivement chaque panel durant la factorisation et en le divisant par le temps écoulé au cours de ces étapes.

Dans ces expériences (2.4,2.5), nous comparons les performances pour factorisation du panel pour les routines suivantes :

— La factorisation CALU modifiée pour le solveur H-CALU and linké avec la version séquentielle version de MKL.
— L'implémentation MKL de la routine LAPACK `dgetrf`, utilisée dans MAGMA.
— Une version récursive de GEPP `rgetf2` décrite dans [45].
— Une routine GENP `dgetrf_nopiv` (sans pivotage).

Les routines comparées dans cette section ont été sélectionnés du fait qu'elles peuvent être utilisées comme noyaux pour notre mise en œuvre hybride CPU / GPU.

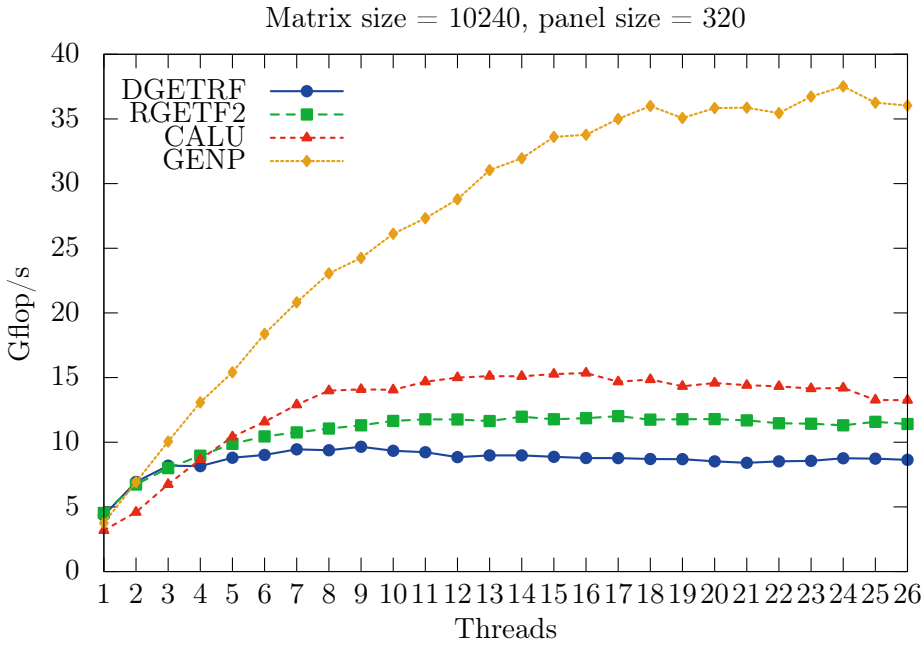



FIGURE 2.4 – Comparaison des performance multithreadées pour le panel.

### 2.4.2 Performances pour les implémentations hybrides de LU

Dans cette section, nous étudions les performances de LU en utilisant les ressources du multicœur (16 threads) et un GPU. Nous comparons dans figure 2.6 les routines suivantes, appliquées à matrices carrées de différentes tailles :

— La routine MAGMA `magma_dgetrf`, où le panel est factorisé avec MKL `dgetrf`,
— H-rgetf2, où le panel est factorisé avec la routine récursive pour GEPP `rgetf2`,
— H-CALU, où le panel est factorisé avec la routine CALUU mentionnée dans la section 2.4.1,
— Le solveur RBT (randomisation + GENP).

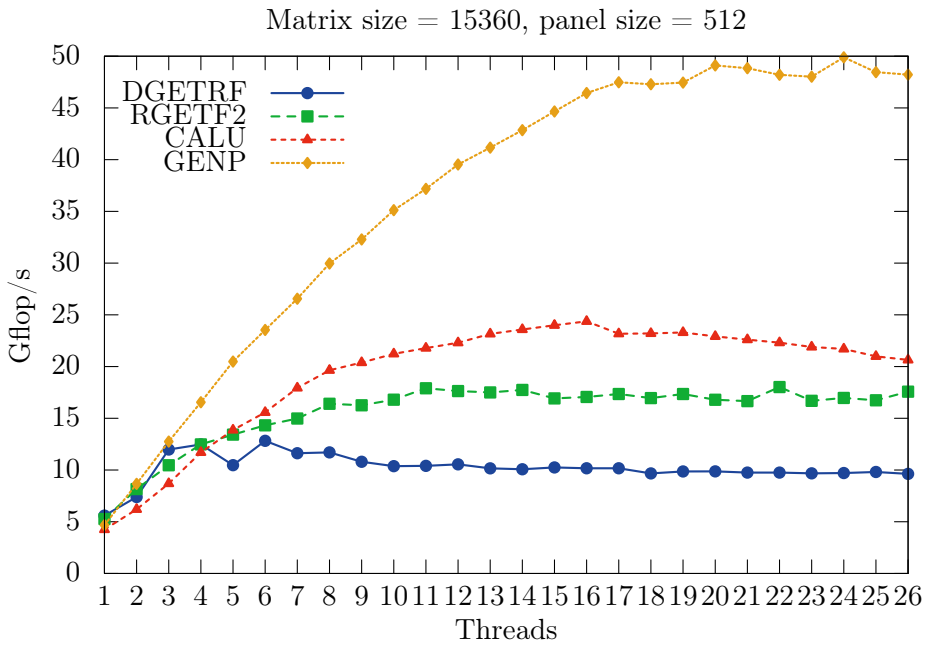



FIGURE 2.5 – Comparaison des performance multithreadées pour le panel.

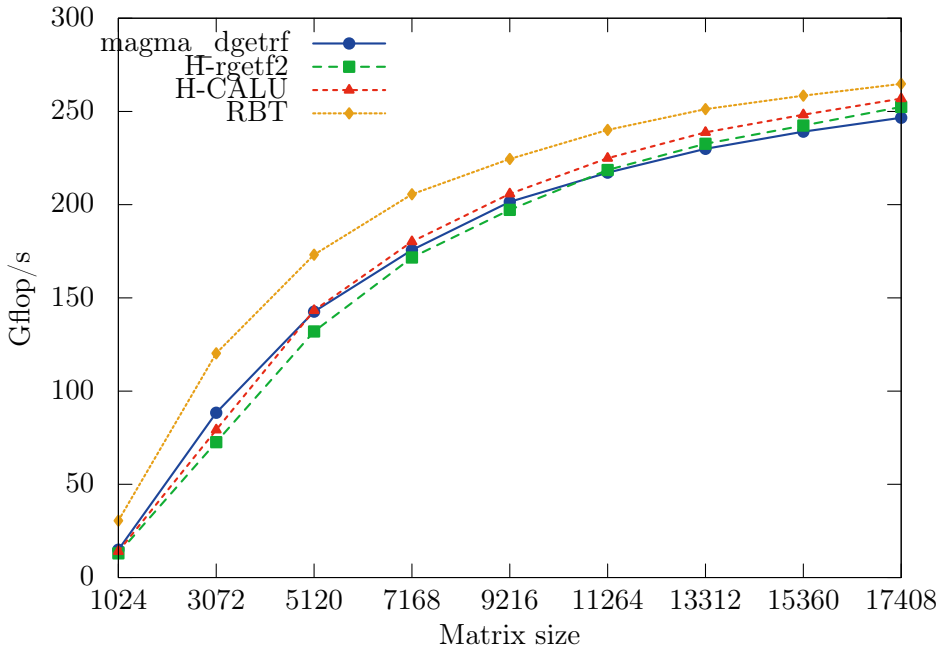



FIGURE 2.6 – Performances sur des matrices carrées

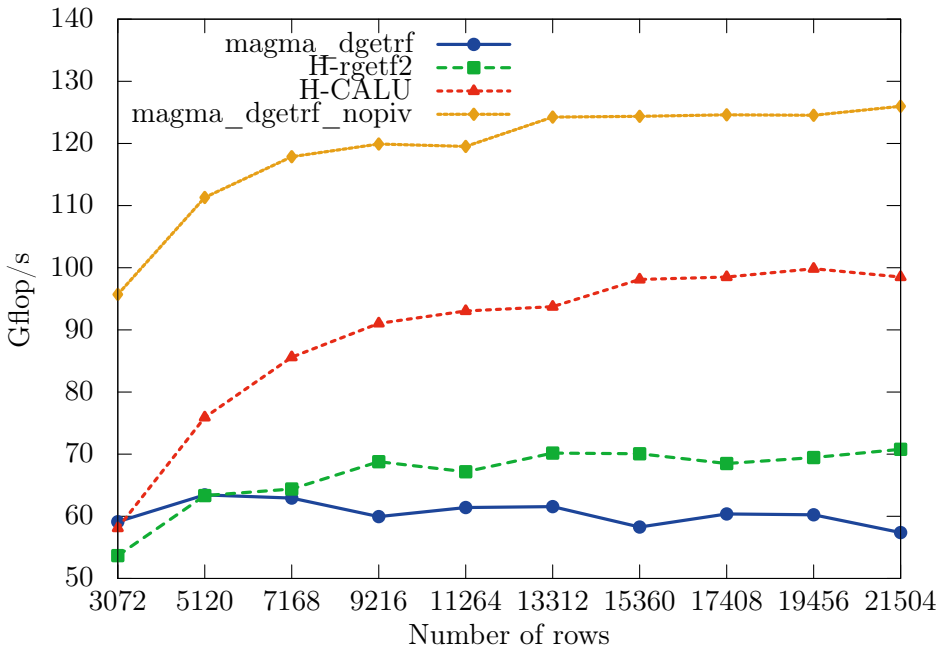



FIGURE 2.7 – Performance on rectangular matrices

On peut constater que H-CALU obtient de meilleures performances que GEPP, plus particulièrement sur des matrices rectangulaires. RBT et GENP nous sert de borne supérieure pour les performances.

## 2.5 Conclusion du chapitre 2

Dans ce chapitre, nous avons présenté différentes routines de factorisation LU en utilisant une machine multicœur accélérée avec un GPU. La différence entre ces approches provient de la stratégie de pivotage choisie pour la factorisation du panel. Nous avons proposé un nouveau solveur hybride minimisant les communications : H-CALU. Dans nos expériences, ce solveur se révèle être plus rapide que l'implémentation classique de GEPP dans MAGMA. Une partie du travail décrit dans ce chapitre a été publié dans [46].

Chapitre 3

# Un solveur rapide randomisé pour les architectures multicœurs accélérées

## 3.1 Introduction

Afin de résoudre des systèmes généraux denses, on utilise habituellement GEPP. Cependant le côut du pivotage est cher. Afin d'éviter de pivoter, la méthode "Random Butterfly Transformation" (RBT) à été proposée. Tout d'abord décrite par Parker dans [14, 15] elle fut récemment développée dans [47]. Cette méthode est tout à fait adaptée aux accélérateurs.

Dans ce chapitre nous décrivons d'abord la structure des matrices récursives random butterfly 3.2, puis dans la Section 3.3, comment le solveur peut être utilisé efficacement sur une machine hybride. Nous présentons ensuite les implémentations pour GPU 3.4 puis pour Xeon Phi 3.5 ainsi que les performances de ces implémentations. Finalement quelques remarques de conclusion dans la Section 3.6.

## 3.2 Le solveur RBT

Une matrice butterfly de taille $N$ par $N$ est définie comme suit :

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} R & S \\ R & -S \end{pmatrix},$$

Où $R$ et $S$ sont deux matrices random, singulières, diagonales.

Une matrice butterfly de taille $N$ par $N$ peut être stoquée dans un vecteur de taille $N$ en ne stoquant que les coéficients de $R$ et $S$.

Une matrice random butterfly récursive de profondeur $d$ comme définie dans [47], à la forme suivante :

$$W^{<n,d>} = \begin{pmatrix} B_1^{<n/2^{d-1}>} & & 0 \\ & \ddots & \\ 0 & & B_{2^{d-1}}^{<n/2^{d-1}>} \end{pmatrix} \times \cdots \times \begin{pmatrix} B_1^{<n/2>} & 0 \\ 0 & B_2^{<n/2>} \end{pmatrix} \times B^{<n>}, \tag{3.1}$$

Où chaque bloc $B_i^{<n>}$ est une matrice butterfly de taille $N$.

La méthode pour résoudre un système $Ax = b$ en utilisant RBT est décrite dans 1.3.5.

## 3.3 L'algorithme RBT hybride

Dans cette section nous décrivons l'implémentation du solveur RBT hybride. Nous utilisons deux niveaux de récursion pour la randomisation. Les tâches suivantes sont effectuées :

1. Les matrices random butterfly $U$ et $V$ sont générées au format compact sur l'hôte (CPU),pendant que la matrice $A$ est envoyé a l'accélérateur.
2. $U$ et $V$ sont envoyées sur l'accélérateur.
3. La randomisation est effectuée sur l'accélérateur. La mise à jour de $A$ est faite en place.
4. La matrice randomisée est factorisée avec GENP comme décrit précédemment.
5. On calcul $U^T b$, puis $A_r y = U^T b$ est aussi résolu sur l'accélérateur.
6. Si nécessaire on applique du raffinement itératif sur la solution calculée $y$ (sur l'accélérateur).
7. On calcul la solution $x = Vy$ sur l'accélérateur et la solution $x$ est envoyée à l'hôte.

## 3.4 Solveur RBT utilisant les processeurs graphiques (GPU)

Notre solveur RBT existe pour toutes les précisions utilisées dans LAPACK et a été intégré dans la bibliothèque MAGMA [1]. Cela inclu les routines de randomisation sur GPU, une routine de factorisation sans pivoter ainsi que de rafinement itératif sur GPU.

### 3.4.1 Implémentation

Sur architectures hybrides CPU/GPU, le solveur RBT est implémenté comme décrit dans la Section 3.3. Dans l'algorithme 2, nous détaillons la routine qui effectue la randomisation de la matrice $A$.

L'algorithme 3 détaille l'implémentation du noyau `Elementary Multiplication`.

Nous utilisons la "shared memory" de chaque bloc pour stocker les coefficients de $U$ et $V$ et ainsi améliorer les performances.

### 3.4.2 Performances

La section suivante présente les performance sur solveur RBT dans MAGMA pour GPU.

1. voir http://icl.cs.utk.edu/magma/news/news.html?id=351

**Algorithm 2** RBT avec deux récursions

**Result:** $A \leftarrow U^T AV$

1: $hauteur\_bloc \leftarrow 32$
2: $largeur\_bloc \leftarrow 4$
3: **Défintion** d'une grille de threads par bloc de taille : $(hauteur\_bloc, largeur\_bloc)$
4: **Definition** d'une grille de blocs de taille : $(\frac{N}{4\times hauteur\_bloc}, \frac{N}{4\times largeur\_bloc})$
{ Tous les noyaux GPU sont appelés avec les dimensions de threads et de grilles définies avant l'appel}
5: **Appel :** Elementary Multiplication($A$, $\&U(N)$, $\&V(N)$, $N/2$)
6: **Appel :** Elementary Multiplication($\&A(0, N/2)$, $\&U(N)$, $\&V(N + N/2)$, $N/2$)
7: **Appel :** Elementary Multiplication($\&A(N/2, 0)$, $\&U(N + N/2)$, $\&V(N)$, $N/2$)
8: **Appel :** Elementary Multiplication($\&A(N/2, N/2)$, $\&U(N + N/2)$, $\&V(N + N/2)$, $N/2$)
9: **Redéfinition** de la grille de blocs à la taille : $(\frac{N}{2\times hauteur\_bloc}, \frac{N}{2\times largeur\_bloc})$
10: **Appel :** Elementary Multiplication($A$, $U$, $V$, $N$) {application d'une récursion de niveau 1}

Dans la Figure 3.1 on peut constater que notre implémentation du solveur RBT obtient des performances jusqu'à 30% meilleures que le solveur GEPP de MAGMA.

Dans la Figure 3.2, nous constatons que la randomisation représente moins de 4% du temps total de la résolution du système.

## 3.5 Solveur RBT utilisant les coprocesseurs Intel Xeon Phi

De façon similaire a la section précédente, nous présentons ici notre implémentation sur solveur RBT mais cette fois ci pour les accélérateurs Intel Xeon Phi. Ce solveur et toutes les fonctions qui vont avec sont incluses dans la bibliothèque MAGMA MIC depuis la version 1.3.

### 3.5.1 Implementation

L'implémentation de la randomisation RBT de profondeur deux sur Xeon PHI est similaire à celle sur GPU excepté qu'il n'ai plus question de bloc de threads.

La fonction `Elementary multiplication Phi` décrite dans l'algorithme 4 utilise des instructions SIMD pour améliorer les performances et utilise OpenMP afin d'exploiter le parallelisme entre les cœurs.

### 3.5.2 Performance

Nous présentons les performances du solveur RBT dans MAGMA pour accélérateur Intel Xeon Phi.

**Algorithm 3** Noyau GPU : Elementary Multiplication($A$, $U$, $V$, $N$)

**for each** bloc de threads de taille $bsize.x \times bsize.y$ de coordonnées $b.x$ et $b.y$ **do**
  **for each** Thread de coordonnées $t.x$ et $t.y$ dans le bloc **do**
    $idx \leftarrow b.x \times bsize.x + t.x$
    $idy \leftarrow b.y \times bsize.y + t.y$
    **if** $idx < N/2$ **and** $idy < N/2$ **then**
      Déclare 4 vecteurs dans la "shared memory" : $U_1[bsize.x]$, $U_2[bsize.x]$, $V_1[bsize.y]$, $V_2[bsize.y]$
      $U_1(t.x) \leftarrow U(idx)$
      $U_2(t.x) \leftarrow U(idx + N/2)$
      $V_1(t.y) \leftarrow V(idy)$
      $V_2(t.y) \leftarrow V(idy + N/2)$
      **Synchronisation** des threads dans le bloc
      $a_{00} \leftarrow A(idx, idy)$
      $a_{01} \leftarrow A(idx, idy + N/2)$
      $a_{10} \leftarrow A(idx + N/2, idy)$
      $a_{11} \leftarrow A(idx + N/2, idy + N/2)$
      $b_1 \leftarrow a_{00} + a_{01}$
      $b_2 \leftarrow a_{10} + a_{11}$
      $b_3 \leftarrow a_{00} - a_{01}$
      $b_4 \leftarrow a_{10} - a_{11}$
      $A(idx, idy) \leftarrow U_1(t.x) \times V_1(t.y) \times (b_1 + b_2)$
      $A(idx, idy + N/2) \leftarrow U_1(t.x) \times V_2(t.y) \times (b_3 + b_4)$
      $A(idx + N/2, idy) \leftarrow U_2(t.x) \times V_1(t.y) \times (b_1 - b_2)$
      $A(idx + N/2, idy + N/2) \leftarrow U_2(t.x) \times V_2(t.y) \times (b_3 - b_4)$
    **end if**
  **end for**
**end for**

Dans la figure 3.3, nous constatons que notre version est jusqu'à 65% plus rapide que le solveur utilisant GEPP.

Dans la Figure 3.4, nous observons que la randomisation requière mois de 3% du temps total et même moins de 1% pour des grosses matrices.

## 3.6 Conclusion du chapitre 3

Dans ce chapitre nous avons présenté 2 implémentations du solveur RBT pour accélérateurs. Une pour GPU, l'autre pourr Intel Xeon Phi. Ces deux méthodes ont été intégrées à la bibliothèque MAGMA et sont significativement plus performantes que les solveurs basés sur GEPP.

Dans le chapitre suivant, nous présentons des méthodes de placement des threads et des données sur architecture NUMA.

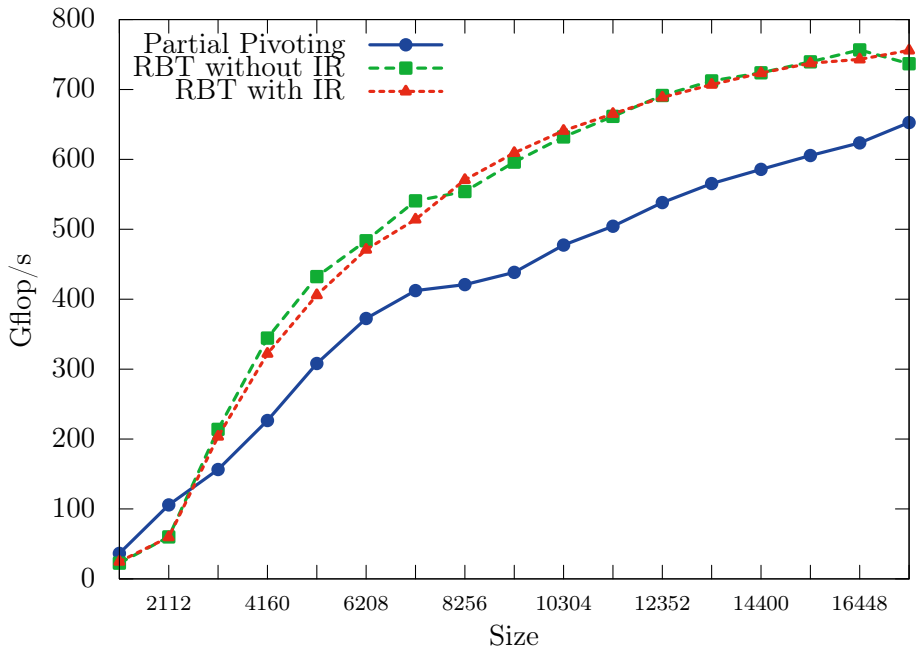



FIGURE 3.1 – Performances du solveur RBT sur GPU.

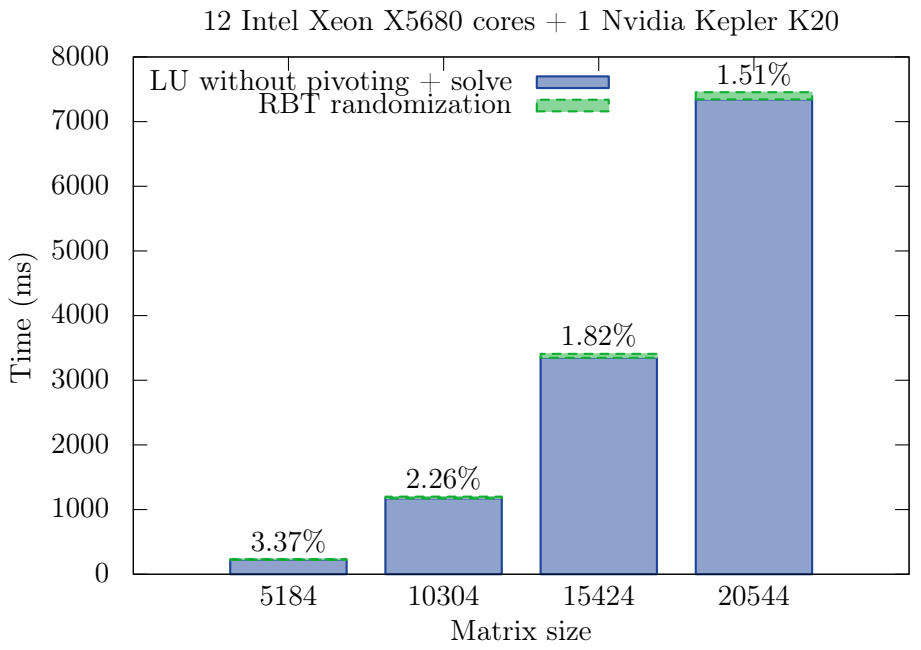



FIGURE 3.2 – Part de la randomisation dans le solveur RBT sur GPU.

**Algorithm 4** Fonction Phi : Elementary multiplication Phi($A$, $U$, $V$, $N$)

**OpenMP** parallel for
**for** $i = 0$ **à** $N/2$ **do**
  Déclaration de $V_1$ et $V_2$ deux registres vectoriels 512-bits.
  **Initialise** toutes les valeurs de $V_1$ avec $V(i)$
  **Initialise** toutes les valeurs de $V_2$ avec $V(i + N/2)$
  **for** $j = 0$ **à** $N/2$ **par pas de** 8 **do**
    Déclaration de $a_{00}$, $a_{01}$, $a_{10}$ et $a_{11}$ quatre registres vectoriels 512-bits.
    **LOAD** 8 valeurs de $A(i, j)$ dans $a_{00}$
    **LOAD** 8 valeurs de $A(i, j + N/2)$ dans $a_{01}$
    **LOAD** 8 valeurs de $A(i + N/2, j)$ dans $a_{10}$
    **LOAD** 8 valeurs de $A(i + N/2, j + N/2)$ dans $a_{11}$
    Déclaration $b_1$, $b_2$, $b_3$ et $b_4$ quatre registres vectoriels 512-bits.
    $b_1 \leftarrow$ **ADD**$(a_{00}, a_{01})$
    $b_2 \leftarrow$ **ADD**$(a_{10}, a_{11})$
    $b_3 \leftarrow$ **SUB**$(a_{00}, a_{01})$
    $b_4 \leftarrow$ **SUB**$(a_{10}, a_{11})$
    Déclaration de $U_1$ et $U_2$ deux registres vectoriels 512-bits.
    **LOAD** 8 valeurs de $U(j)$ dans $U_1$
    **LOAD** 8 valeurs de $U(j + N/2)$ dans $U_2$
    $a_{00} \leftarrow$ **MUL**($U_1$, **MUL**($V_1$, **ADD**$(b_1, b_2)$))
    $a_{01} \leftarrow$ **MUL**($U_1$, **MUL**($V_2$, **ADD**$(b_3, b_4)$))
    $a_{10} \leftarrow$ **MUL**($U_2$, **MUL**($V_1$, **SUB**$(b_1, b_2)$))
    $a_{11} \leftarrow$ **MUL**($U_2$, **MUL**($V_2$, **SUB**$(b_3, b_4)$))
    **STORE** 8 valeurs de $a_{00}$ dans $A(i, j)$
    **STORE** 8 valeurs de $a_{01}$ dans $A(i, j + N/2)$
    **STORE** 8 valeurs de $a_{10}$ dans $A(i + N/2, j)$
    **STORE** 8 valeurs de $a_{11}$ dans $A(i + N/2, j + N/2)$
  **end for**
**end for**

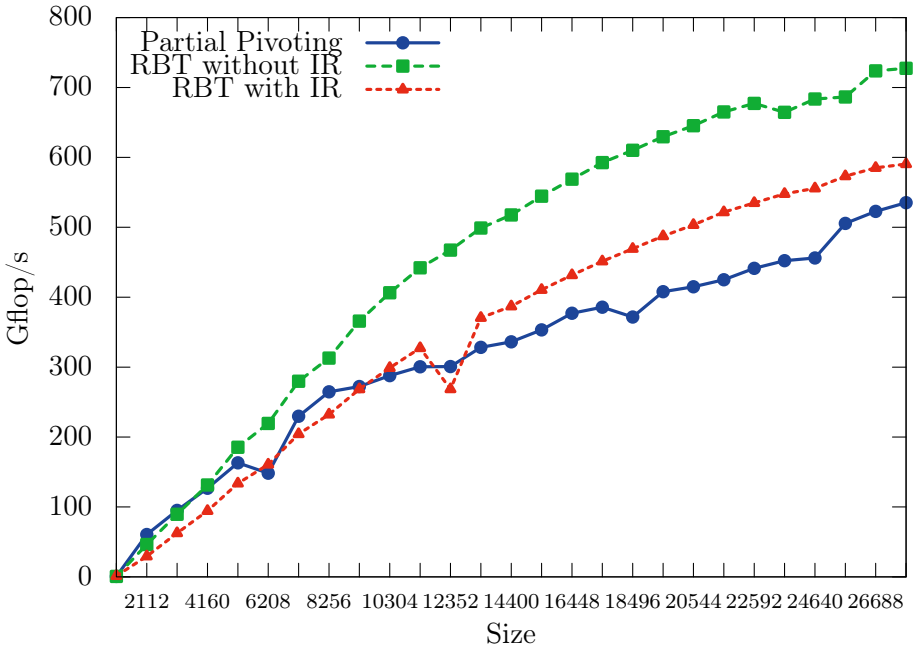



FIGURE 3.3 – Performance du solveur RBT sur coprocesseur Xeon Phi.

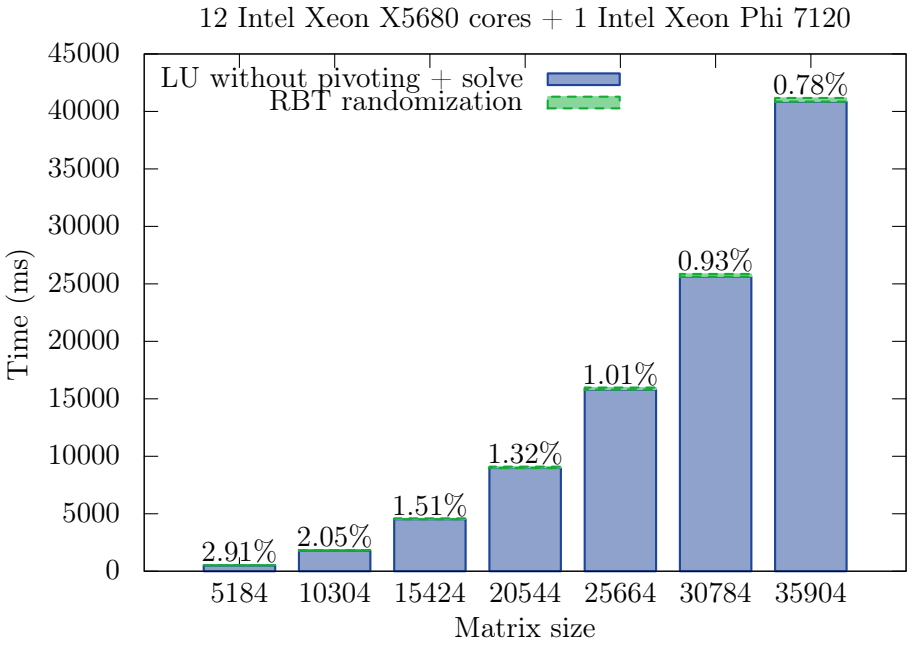



FIGURE 3.4 – Part de la randomisation sur Xeon Phi.

Chapitre 4

# Optimisation de la localité pour les architectures NUMA

## 4.1 Utiliser les architectures NUMA pour des systèmes linéaires denses

Dans les architectures parallèles modernes, la bande passante mémoire joue un rôle crucial pour les applications hautes performances. Il y a deux types de systèmes à mémoire partagée : Les systèmes UMA (Unified Memory Access) qui contient une seule banque de mémoire partagée par tous avec la même latence et la même bande passante pour tous les threads. L'utilisation de nombreux threads sur un système UMA peut entrainer la saturation du bus mémoire. Les systèmes ccNUMA (cache coherent Non Uniform Memory Access) où la mémoire est distribuée physiquement mais partagée logiquement [48]. Chaque banque de mémoire est associée avec un jeu de cœurs (ils forment un nœud NUMA). Du fait de leur distribution physique, les performances des accès mémoire varient en fonction de la localité des données et des threads [49].. Afin d'obtenir de bonnes performances sur ce type de système, la localité des données et des threads doit être prise en compte par différentes méthodes de programmation.

Dans ce chapitre nous étudions les effets NUMA sur la résolutions de systèmes linéaires généraux denses. Et comment le placement des threads et de la mémoire peut améliorer les performances sur la factorisation du panel lors d'une factorisation LU par bloc.

Le chapitre est organisé comme suit. Dans la section 4.2 nous décrivons le contexte applicatif de ces travaux. Dans la section 4.3 nous décrivons différentes stratégies de placement pour les threads et la mémoire. La section 4.4 expose les performances obtenues. Enfin une conclusion est donnée dans la section 4.5.

## 4.2 Contexte applicatif

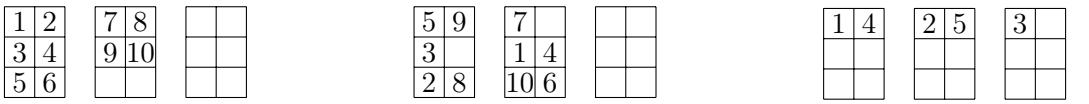
Nous considérons ici la factorisation LU hybrid implémentée dans la bibliothèque MAGMA et détaillée dans la section 2.2. Nous prenons en compte GEPP et GENP qui peut être utilisé pour RBT [47]. Du fait du pivotage, GEPP effectue de nombreux accès mémoire. Nous nous concentrons ici sur la factorisation du panel qui est la partie du problème limitée par la performance de la mémoire, ce qui la rend sujet à des effets NUMA.

## 4.3 Strategies de placement

Dans cette section nous décrivons comment les threads et la mémoire peuvent être placés.

Pour placer la mémoire nous utilisons la fonction `mbind()` de la bibliothèque `libnuma` [50]. Pour attacher les threads à un cœur nous utilisons la fonction Unix `sched_setaffinity()`. Nous répartissons les données entre les différents nœuds NUMA utilisés.

Pour le placement des threads nous considérons les 3 stratégies illustrées dans la Figure 4.1.




(a) Sequential pinning 1 (b) Sequential pinning 2 (c) Cyclic pinning

FIGURE 4.1 – Exemples de méthodes de placement

**No pinning :** Les threads sont placés par le système (`noPin`).

**Sequential pinning 1 :** Nous utilisons 1 thread par cœur du nœud NUMA, on remplit les nœuds au maximum. Comme illustré dans la Figure 4.1a ( `seqPin1`).

**Sequential pinning 2 :** Similaire à seqPin1 mais on essaye de mettre le même nombre de thread sur chaque nœud NUMA afin d'équilibrer la charge de travail. Illustré dans la Figure 4.1b (`seqPin2`).

**Cyclic pinning :** Les threads sont placés cycliquement sur tous les nœuds comme illustré dans la figure 4.1c (`cycPin`).

Le sequential pinning maximise le nombre d'accès mémoire locaux tandis que cyclic pinning maximise la quantité de mémoire cache L3 disponible par thread.

## 4.4 Application à la factorisation LU

### 4.4.1 Environnement expérimental

Nos expériences ont été effectuées sur un système comprenant 48 cœurs MagnyCours AMD Opteron 6172 répartis sur 8 nœuds NUMA ayant chacun 6 Mo de mémoire cache L3. Le GPU utilisé est un NVIDIA Fermi Tesla S2050.

### 4.4.2 Performance pour la factorisation du panel

Ici 4.2 nous comparons les performances pour la factorisation du panel en fonction du nombre de threads utilisés pour chaque stratégie de placement décritres précédemment. Nous testons nos méthodes sur GEPP et GENP.

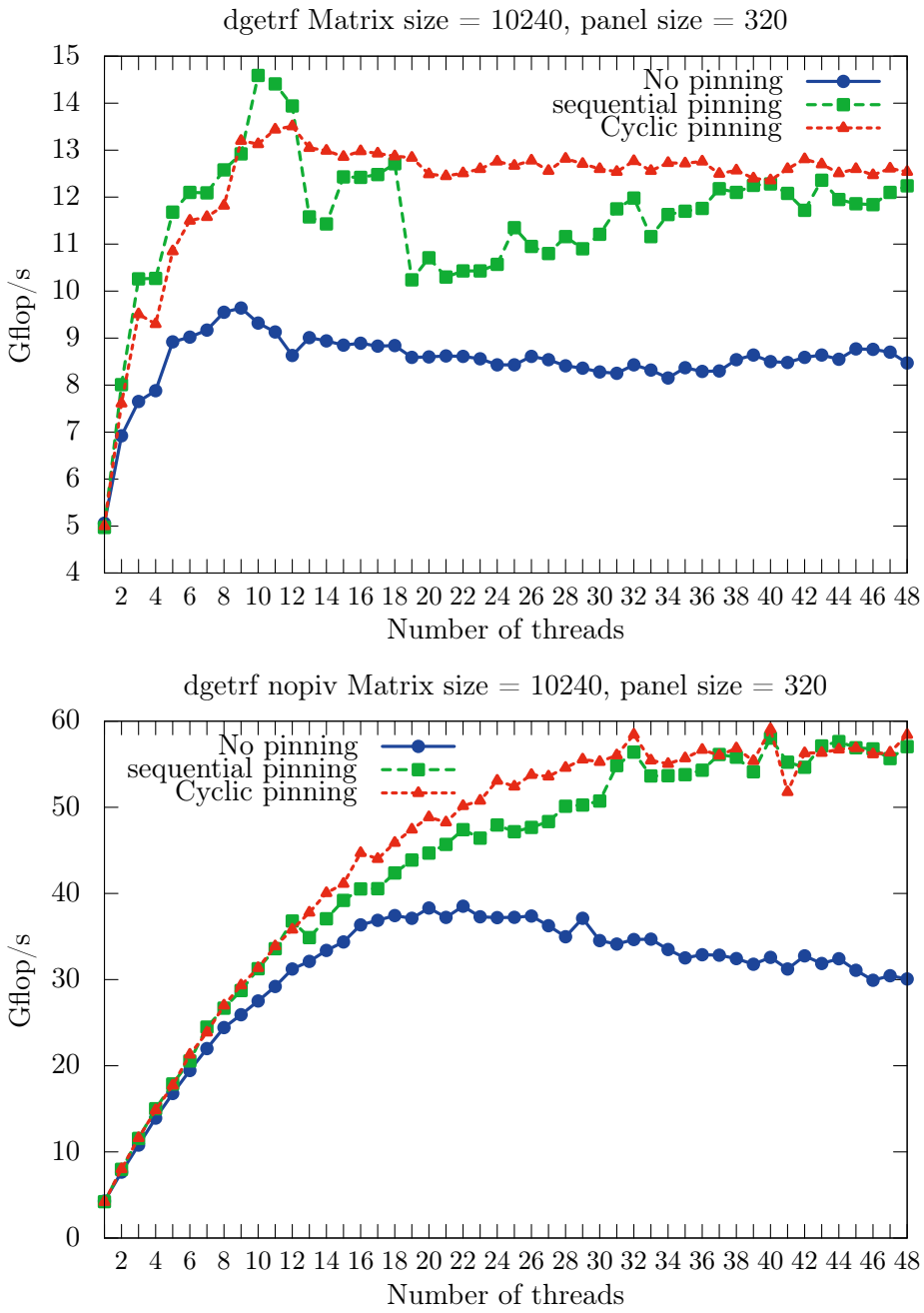



FIGURE 4.2 – Performances des stratégies de placement de threads pour la factorisation LU du panel avec pivotage (en haut) et sans pivotage (en bas). Taille du panel : $10240 \times 320$.

Dans ces expériences on s'intéresse au nombre de thread donnant les meilleures performances. En effet ici utiliser 48 threads n'est jamais la meilleure solution. Nous pouvons observer que le sequencial ou cyclic pinning donne de meilleurs résultats

que le placement de base des threads par le système. Pour GEPP les meilleurs résultats sont obtenus avec seqPin2 et 10 threads. Ceci est du à l'amélioration de la localité des données. GENP est moins affecté par la localité des données car il n'y a pas de recherche de pivot. De ce fait l'augmentation de la quantité de cache de L3 disponible par thread que fournit CycPin est plus profitable.

### 4.4.3 Performances pour le code hybride

Ici nous évaluons l'impact du placement des threads et des données sur une factorisation LU hybride utilisant CPU et GPU.

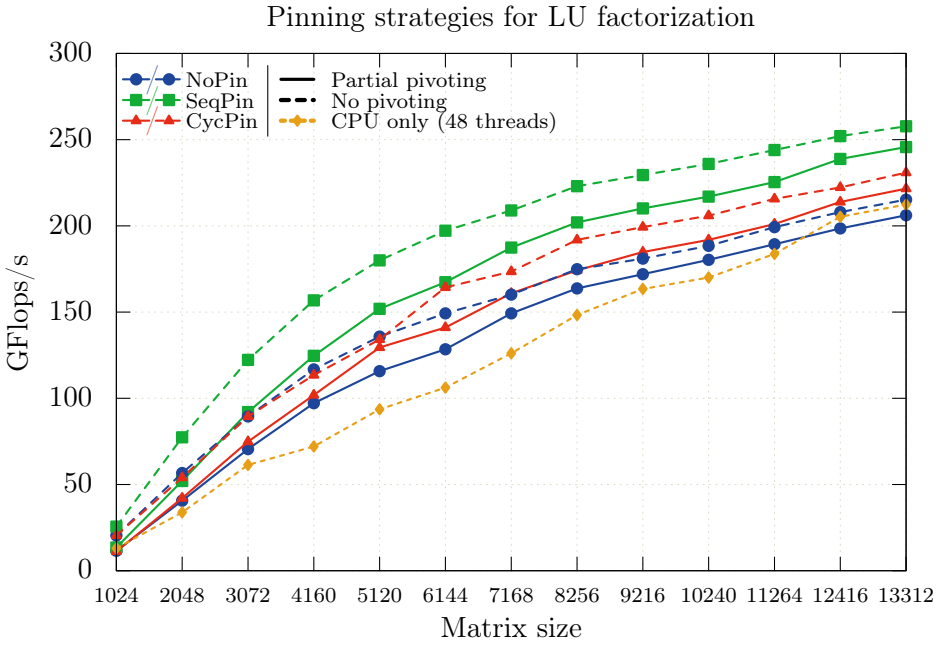



FIGURE 4.3 – Performances pour la factorisation LU hybride avec et sans pivotage (12 threads).

La figure 4.3 compare les performances de `seqPin2` et `cycPin` avec `noPin` pour GEPP et GENP.Nous pouvons constater que `seqPin2` donne de meilleurs résultats que les autres méthodes.

## 4.5 Conclusion du Chapitre 4

Dans ce chapitre nous avons étudié différentes méthodes de placement des threads et de la mémoire sur une architecture NUMA utilisant un GPU comme accélérateur, ceci dans le cadre de la factorisation LU. Les deux méthodes développées permettent une amélioration des performances jusqu'à un facteur 2 comparé au placement par

défaut. Le choix de la méthode la plus efficace dépend du l'algorithme, de l'architecture et de la taille des données. Une partie de ces travaux à été publiée dans [51].

# Conclusion et travaux futurs

Dans le domaine du calcul haute performance, les architectures parallèles fournissent de plus en plus de puissance de calcul mais sont de plus en plus difficiles à programmer efficacement. Dans cette thèse nous avons cherché à développer des algorithmes et des programmes efficaces, permettant de résoudre des systèmes linéaires denses. Nous avons proposé des solutions pour améliorer les performance sur architectures hybrides, en optimisant la factorisation du panel, en randomisant ou en prenant en compte les effets NUMA. Nous avons proposé un solveur hybride utilisant une stratégie de pivotage dite "communication avoiding" qui obtient de meilleures performance que le solveur dans MAGMA utilisant GEPP. Nous avons également étudié des solutions basées sur RBT afin d'éviter de pivoter tout en restant numériquement stable en pratique. Ces solveurs ont été intégrés dans les version de la bibliothèque MAGMA pour GPU et pour Xeon Phi. Enfin nous avons proposé des méthodes de placement pour les threads et les données sur des architectures NUMA utilisant un GPU comme accélérateur. Ces méthodes permettent d'améliorer les performances dans le cadre d'une factorisation LU.

## Perspectives

Les perspectives envisagées incluent l'implémentation de H-CALU pour plusieurs GPU, l'application de RBT à des systèmes symétriques indéfinis, appliquer RBT par lot sur de nombreux petits systèmes denses sur GPU. Le raffinement itératif sur Xeon Phi nécessite d'être optimisé. Pour la partie NUMA nous pourrions développer une heuristique afin de déterminer la meilleure méthode à utiliser. Enfin nous pourrions également implémenter ces méthodes de placement dans des ordonnanceurs tels que QUARK.

# Bibliographie


[1] L. Hodgkin. *A History of Mathematics : From Mesopotamia to Modernity.* Oxford University Press, 2005. (Cited on page 3.)

[2] A. Tucker. The growing importance of linear algebra in undergraduate mathematics. *College Mathematics Journal*, 24(1) :3–9, 1993. (Cited on page 3.)

[3] G. E. Moore et al. Cramming more components onto integrated circuits, 1965. (Cited on page 3.)

[4] J. K. Prentice. A Quantum Mechanical Theory for the Scattering of Low-Energy Atoms from Incommensurate Crystal Surface Layers. Technical report, New Mexico Univ., Albuquerque, NM (United States), 1992. (Cited on page 4.)

[5] A. Edelman. The first annual large dense linear system survey. *ACM SIGNUM Newsletter*, 26(4) :6–12, 1991. (Cited on page 4.)

[6] X. Li and J. Demmel. SuperLU_DIST : A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Softw.*, 29(2) :110–140, June 2003. (Cited on page 4.)

[7] P. Amestoy, J. Y. L'Excellent, F. H. Rouet, and M. Sid-Lakhdar. Modeling 1D distributed-memory dense kernels for an asynchronous multifrontal sparse solver. In *High-Performance Computing for Computational Science, VECPAR 2014, Eugene, Oregon, USA, 30/06/2014-03/07/2014*, http ://www.laas.fr, 2014. LAAS. (Cited on page 4.)

[8] C. Fu, X. Jiao, and T. Yang. Efficient sparse LU factorization with partial pivoting on distributed memory architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 9(2) :109–125, Feb 1998. (Cited on page 4.)

[9] G. H. Golub and C. F. van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1996. Third edition. (Cited on page 4.)

[10] J. Demmel, L. Grigori, M.F. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. Technical Report UCB/EECS-2008-89, EECS Department, University of California, Berkeley, Aug 2008. Current version available in the ArXiv at http ://arxiv.org/pdf/0809.0101 Replaces EECS-2008-89 and EECS-2008-74. (Cited on page 5.)

[11] L. Grigori, J. Demmel, and H. Xiang. CALU : a communication optimal LU factorization algorithm. *SIAM J. Matrix Anal. and Appl.*, 32 :1317–1350, 2011. (Cited on page 5.)

[12] S. Donfack, L. Grigori, and A. K. Gupta. Adapting communication-avoiding LU and QR factorizations to multicore architectures. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010. (Cited on pages 5 and 11.)

[13] L. Grigori, J. Demmel, and H. Xiang. Communication avoiding Gaussian elimination. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 29. IEEE Press, 2008. (Cited on pages 5 and 11.)

[14] D. S. Parker. Random Butterfly Transformations with Applications in Computational Linear Algebra. Technical Report CSD-950023, Computer Science Department, UCLA, 1995. (Cited on pages 6 and 16.)

[15] D. S. Parker and B. Pierce. The randomizing FFT : an aternative to pivoting in Gaussian elimination. Technical Report CSD-950037, Computer Science Department, UCLA, 1995. (Cited on pages 6 and 16.)

[16] J. von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, jun 1945. Report prepared for U.S. Army Ordinance Department under Contract W-670-ORD-4926. (Cited on page 6.)

[17] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9) :948–960, 1972. (Cited on page 6.)

[18] H. Sutter. The free lunch is over : A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3) :202–210, 2005. (Cited on page 7.)

[19] W. Stallings. *Computer Organization and Architecture - Designing for Performance (7. ed.)*. Pearson / Prentice Hall, 2006. (Cited on page 7.)

[20] B. I. Witt. M65mp : An experiment in OS/360 multiprocessing. In *Proceedings of the 1968 23rd ACM National Conference*, ACM '68, pages 691–703, New York, NY, USA, 1968. ACM. (Cited on page 7.)

[21] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. (Cited on page 7.)

[22] CUDA Nvidia. Compute Unified Device Architecture programming guide. 2007. (Cited on page 7.)

[23] *TOP500 Supercomputer Site*. `http://www.top500.org`. (Cited on page 8.)

[24] Intel. *Intel® Xeon Phi™ Coprocessor System Software Developers Guide*. 2012. `http://software.intel.com/en-us/articles/`. (Cited on page 8.)

[25] G. Chrysos. Intel® Xeon Phi™ Coprocessor-the Architecture. *Intel Whitepaper*, 2014. (Cited on page 8.)

[26] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu. Test-driving Intel Xeon Phi. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, pages 137–148. ACM, 2014. (Cited on page 8.)

[27] International Business Machines Corporation. *System/360 Scientific Subroutine Package (360A-CM-03X) Version II, Programmer's Manual*. IBM Technical Publications Department, White Plains, NY, 1967. (Cited on page 8.)

[28] B. S. Garbow. EISPACK-a package of matrix eigensystem routines. *Computer Physics Communications*, 7(4) :179–184, 1974. (Cited on page 8.)

[29] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3) :308–323, September 1979. (Cited on page 8.)

[30] J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK Users' Guide*, volume 8. SIAM, 1979. (Cited on page 9.)

[31] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. (Cited on page 9.)

[32] AMD. AMD Core Math Library (ACML). *URL http ://developer. amd. com/acml. jsp*, 2012. (Cited on page 9.)

[33] Intel. *Math Kernel Library (MKL)*. `http://www.intel.com/software/products/mkl/`. (Cited on page 9.)

[34] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, December 1997. URL : `http://www.netlib.org/lapack/lawns/lawn131.ps`. (Cited on page 9.)

[35] K. Goto. GotoBLAS. *Texas Advanced Computing Center, University of Texas at Austin, USA. http: // www. otc. utexas. edu/ ATdisplay. jsp* , 2007. (Cited on page 9.)

[36] Z. Xianyi, W. Qian, and Z. Chothia. OpenBLAS. *http: // xianyi. github. io/ OpenBLAS* , 2012. (Cited on page 9.)

[37] CUDA Nvidia. Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15, 2008. (Cited on page 9.)

[38] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997. (Cited on page 9.)

[39] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. PLASMA users' guide. Technical report, Technical report, ICL, UTK, 2009. (Cited on page 9.)

[40] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK users guide : QUeueing And Runtime for Kernels. Technical Report ICL-UT-11-02, University of Tennessee, Innovative Computing Laboratory, 2011. (Cited on page 9.)

[41] R. Nath, S. Tomov, and J. Dongarra. An improved MAGMA GEMM for Fermi GPUs. *International Journal of High Performance Computing Applications*, 24(4) :511–515, 2010. (Cited on page 9.)

[42] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5&6) :232–240, 2010. (Cited on page 9.)

[43] S. Tomov, R. Nath, and J. Dongarra. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Computing*, 36(12) :645–654, 2010. (Cited on page 9.)

[44] J. Kurzak and J. J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. *LAPACK Working Note 178*, September 2006. (Cited on page 10.)

[45] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6) :737–755, 1997. (Cited on page 12.)

[46] M. Baboulin, S. Donfack, J. Dongarra, L. Grigori, A. Rémy, and S. Tomov. A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines. In *International Conference on Computational Science (ICCS 2012)*, volume 9 of *Procedia Computer Science*, pages 17–26. Elsevier, 2012. (Cited on page 15.)

[47] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov. Accelerating linear system solutions using randomization techniques. *ACM Trans. Math. Softw.*, 39(2), 2013. (Cited on pages 16 and 23.)

[48] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2011. (Cited on page 23.)

[49] C. Lameter. Local and remote memory : Memory in a Linux/NUMA system. In *Linux Symposium (OLS2006), Ottawa, Canada*, 2006. `ftp://ftp.tlk-l.net/pub/linux/kernel/people/christoph/pmig/numamemory.pdf`. (Cited on page 23.)

[50] A. Kleen. A NUMA API for linux. Technical report, Novel Inc, 2004. `http://www.halobates.de/numaapi3.pdf`. (Cited on page 24.)

[51] A. Rémy, M. Baboulin, M. Sosonkina, and B. Rozoy. Locality optimization on a NUMA architecture for hybrid LU factorization. In *International Conference on Parallel Computing (PARCO 2013)*, volume 25 of *Advance in Parallel Computing*, pages 153–162. IOS Press, 2014. (Cited on page 27.)