Optimisation du produit matrice-vecteur creux sur architecture GPU pour un simulateur de réservoir

Corentin Rossignon

ComPAS'2013 : RenPar'21 / SympA'15/ CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

Optimisation du produit matrice-vecteur creux sur architecture GPU pour un simulateur de réservoir

Corentin Rossignon *

Total CSTJF - PAU
Avenue Larribau
64000 Pau - France
Corentin.Rossignon@Total.com

Résumé
Pour l'entreprise Total, la simulation de réservoir est une étape importante dans le processus d'optimisation de la production. Actuellement ces simulations s'exécutent entièrement sur CPU. Nous avons donc essayé d'accélérer les produits matrice-vecteur creux contenus dans le simulateur en utilisant des GPUs. Les bibliothèques GPU d'algèbre linéaire creux utilisent des formats génériques de stockage de matrices creuses qui sont plus ou moins performant sur GPU mais qui ne permettent pas d'exploiter la structure particulière des matrices utilisées dans le simulateur de réservoir. Pour exploiter cette structure, nous avons adapté pour nos matrices un format de stockage qui nous permet d'accélérer jusqu'à un facteur 20 le produit matrice-vecteur creux sur 3 GPUs par rapport à 8 coeurs CPU et d'un facteur 1,5 sur GPU par rapport aux formats génériques utilisée par NVIDIA dans cuSPARSE.

Mots-clés : solveur creux, GPU, SpMV, CSR

---

1. Introduction

La simulation de réservoir (réserve naturelle d'hydrocarbures) est l'élément central dans l'étude des champs pétrolifères ou gaziers. Elle permet d'optimiser le placement des puits d'extraction, de calculer le rendement en hydrocarbures et aussi d'expérimenter de nouvelles méthodes d'extraction.
Dans cette simulation, le temps est discrétisé avec un pas de l'ordre de la journée. Chaque pas nécessite de résoudre plusieurs systèmes d'équations linéaires. Ces résolutions représentent près de 70% du temps de calcul utilisé par le simulateur.
Actuellement ces simulations sont réalisées sur CPU, et le calcul est distribué avec MPI. L'émergence des solutions GPGPU et la promesse de performances font que le simulateur pourrait gagner à être porté sur GPU. La résolution de ces systèmes d'équations requiert l'utilisation de matrices creuses de grandes tailles. Pour réduire les ressources (CPU, RAM) utilisées par ces matrices, seuls les éléments non nuls des matrices sont stockés et leurs coordonnées peuvent être encodées de manière plus ou moins direct sous la forme d'un format de stockage.

∗. Doctorant CIFRE, Total - Inria Bordeaux - LaBRI, Université Bordeaux 1, 351 cours de la libération, 33405 Talence. Le texte a été relu par Pascal Henon (Total), Olivier Aumage (Inria) et Samuel Thibault (Inria).

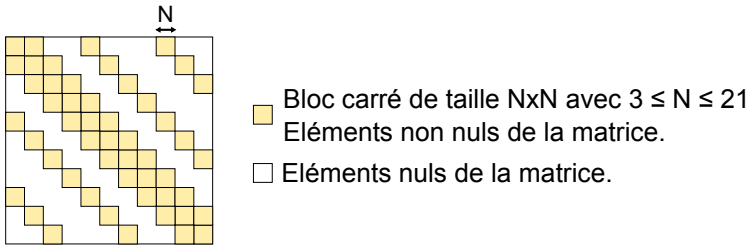

FIGURE 1 – Matrices utilisées pour la simulation de réservoir.

Sur GPU, ces formats de stockage auront un impact sur les performances, l'ordre des accès mémoire change en fonction du format et le GPU est un processeur sensible à l'ordre des accès mémoire.

2. Contexte

2.1. Simulation de réservoir

Dans la simulation de réservoir, le temps est discrétisé avec un pas de l'ordre de la journée. Pour chaque pas, il est nécessaire de faire converger un système non-linéaire, pour cela de nombreux systèmes linaires creux doivent être résolus. Les méthodes de résolution de systèmes linaires creux utilisent des matrices particulières, elles sont creuses et chaque entrée de la matrice est un bloc dense carrée de coefficients non nuls positionnés sur des diagonales. (Fig. 1)

2.2. Les matrices creuses

Les matrices creuses sont des matrices essentiellement composées de zéros. Pour ne pas avoir à stocker tous les zéros, seuls les éléments non nuls (NZ) sont stockés. Différents formats de stockage de matrices creuses existent comme le COO (*COOrdinate list*) ou le CSR (*Compressed Sparse Row*) et permettent d'encoder de différentes manières les coordonnées des éléments non nuls. Le format a un impact sur la taille de la matrice en mémoire et sur l'ordre des accès mémoires.

2.3. La programmation GPGPU

La programmation GPGPU consiste à détourner l'utilisation du processeur graphique (GPU), pour lui faire effectuer des calculs généralistes. Pour cela, nous devons écrire un code, aussi appelé Noyau (*Kernel*), dans un langage exprimant le parallélisme (CUDA pour une carte NVIDIA ou OpenCL pour la portabilité). Dans ces langages, le parallélisme est exprimé sous la forme de *Thread*, un *Thread* représente une instance du Noyau auquel on a donné un identifiant. Les *Threads* sont organisés en groupes appelés *Warp*, les *Threads* d'un même *Warp* sont totalement synchrones.
Le GPU, à l'inverse du CPU, est surtout composé d'unités de calcul. Cela permet de pouvoir faire s'exécuter un grand nombre de *Threads* en parallèle.
Avec l'architecture Fermi de NVIDIA, le GPU est composé d'unités SIMD appelées Streaming Processor (SP), ces unités sont capable à un instant T d'exécuter N instructions identiques sur N données potentiellement différentes, avec N la taille d'un *Warp*, 32 dans le cas de CUDA.
Il existe deux types de mémoire sur GPU :

— la mémoire partagée (*shared memory*), de petite taille et avec une faible latence, partagée entre les *Threads* d'un bloc.
— la mémoire globale (*global memory*), commune à tous les *Threads*, mais avec une latence élevée qui peut être masquée par l'utilisation d'un grand nombre de *Threads*.

Les accès à la mémoire globale présentent une contrainte, pour que les 32 instructions puissent s'exécuter en parallèle, les 32 données doivent être présentes dans le SP, or la mémoire globale ne peut envoyer que 128 octets à la fois (soit 32 float), les *Threads* doivent donc accéder aux données contiguës en mémoire, on appelle cela faire des accès coalescés. Si les données sont réutilisées plusieurs fois, il peut être avantageux de les placer dans la mémoire locale (*shared memory*) pour minimiser la latence et aussi pour pouvoir faire des accès désordonnés à la mémoire locale plutôt qu'à la mémoire globale.
Dans le cas du SpMV ($Ax = b$), seul les données du vecteur multiplicatif x sont lues plusieurs fois et de manière désordonnés, elles peuvent être donc mises en cache

3. Étude

3.1. Proposition

Maintenant que nous en connaissons un peu plus sur l'architecture des GPU, nous pouvons étudier l'écriture d'un produit matrice/vecteur efficace. Pour cela nous devons choisir un format de représentation adéquat des matrices creuses. En effet, sur GPU, l'ordre des accès mémoire est un élément essentiel à prendre en compte si nous souhaitons l'exploiter au maximum de ses capacités. Pour cela le format de stockage des matrices creuse doit nous permettre d'accéder correctement à la mémoire. Nous allons donc tester plusieurs formats existants pour trouver le meilleur. Puis, si possible, créer un nouveau format moins générique qui prendra en compte la structure particulière de nos matrices.

3.2. Expérimentation des principaux formats existants

La station de test est composée de 3 Tesla C2050 (puissance crête : 1030 Gflops et 3 Go de mémoire pour un GPU) et de 2 Xeon X5570 (4 coeurs, 2.93 GHz pour un CPU).
Les résultats sont exprimés en flops, pour chaque élément non nuls de la matrice, nous effectuons une multiplication et une addition, la formule pour calculer les flops est donc : $(2 * NNZ)/Temps$, NNZ représente le nombre d'éléments non nuls.
Les matrices utilisées sont générées par un programme développé par Total et sont représentatives de ce qui est réellement utilisé. Les matrices utilisées sont les suivantes :

— 64000 x 64000 avec des blocs de taille 8 x 8 et un total de 0.8 million de non zéros ;
— 128000 x 128000 avec des blocs de taille 16 x 16 et un total de 14 million de non zéros ;
— 216000 x 216000 avec des blocs de taille 8 x 8 et un total de 12 million de non zéros ;
— 432000 x 432000 avec des blocs de taille 16 x 16 et un total de 47 million de non zéros.

Nous les nommons respectivement 20_8, 20_16, 30_8 et 30_16.
Avec ces matrices, nous testons deux propriétés :

— La scalabilité des Noyaux GPU en fonction du nombre d'éléments non nuls ;
— l'influence de la taille des blocs sur les performances.

Pour les tests sur CPU, nous utiliserons la librairie MKL d'Intel [7] regroupant des routines d'algèbres linéaires efficaces. Pour les test sur GPU, nous utiliserons le framework CUDA de NVIDIA [1]. Les temps de transfert entre le CPU et le GPU ne sont pas pris en compte, ces transferts peuvent être éliminés par le portage complet sur GPU des autres opérations du solveur linéaire. La parallélisation sur plusieurs coeurs de calculs se fera via StarPU [2], il s'occupera des transferts mémoires entre le CPU et le GPU ainsi que de la gestion des contextes CUDA. Le

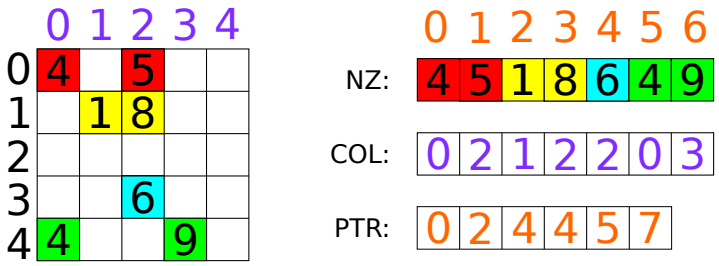

FIGURE 2 – Compressed Sparse Row

découpage de la matrice reste à notre charge.

3.3. CSR : Compressed Sparse Row

Les éléments non nuls (NZ) sont stockés par ligne, pour chaque élément nous avons sa colonne (COL) et sa ligne doit être calculée en utilisant PTR. Le tableau PTR correspond à l'indice dans NZ du premier élément d'une ligne. (Fig. 2)

L'implémentation du CSR sur GPU n'est pas évidente, pour avoir des accès mémoire coalescés à la matrice, il faut que les *Threads* d'un même *Warp* travail sur la même ligne, or cela pose deux problèmes :

- un problème d'accès concurrent au vecteur résultat, ce problème peut être réglé par une réduction du résultat de chaque *Thread* à la fin de chaque ligne, une technique efficace existe en CUDA [6] ;
- une charge de travail par *Warp* insuffisante, pour nos matrices les *Threads* d'un *Warp* n'ont qu'entre 2 et 4 opérations à effectuer.

Malgré ces problèmes, le choix d'un *Warp* par ligne reste plus efficace que le choix d'un *Thread* par ligne.

Nous avons comparé notre version avec les version implémentés dans cuSPARSE [9] et Cusp [3], nous obtenons les mêmes performances.

Sur la figure 6 nous pouvons remarquer qu'un SpMV sur un GPU atteint entre 10 et 19 Gflops alors que les 8 coeurs du CPU n'atteignent que 4 Gflops.

Dans nos cas, quelque soit la taille de la matrice, les performances sur CPU sont équivalentes. Par contre, sur le GPU, la taille des blocs a un impact sur les performances. Cela provient du fait que les accès mémoire à l'intérieur d'un *Warp* deviennent coalescés, pour une taille de bloc égale à 8 il faut lire 4 segments mémoires potentiellement non coalescés du vecteur multiplicatif alors qu'avec une taille de bloc égale à 16, nous n'avons plus que 2 segments à lire.

3.4. JAD : JAgged Diagonale storage ou JDS

La JAD [8] consiste d'abord à "pousser" tous les éléments vers la gauche, puis à trier les lignes par ordre décroissant d'éléments non nuls. Les permutions résultant du tri sont stockées dans un tableau PERM. Les données sont stockées par colonne (Fig. 3).

Le JAD est un format adapté aux machines vectorielles, pour le cas du SpMV les données contiguës en mémoire représente toujours une écriture dans des données contiguës en mémoire et sans recouvrement. Son implémentation sur GPU est plutôt simple, il suffit d'attribuer un *Thread* par ligne de la matrice, les données étant stockées par colonne, les accès mémoire à la

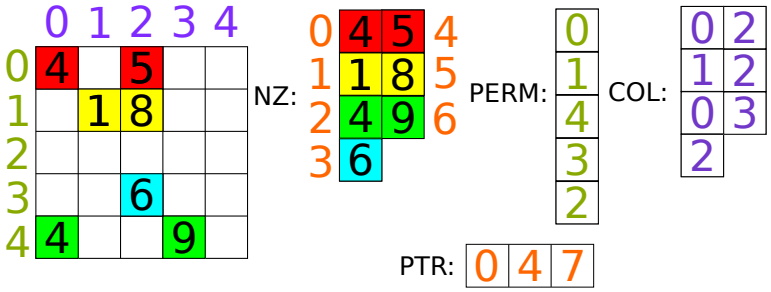

FIGURE 3 – JAgged Diagonale storage

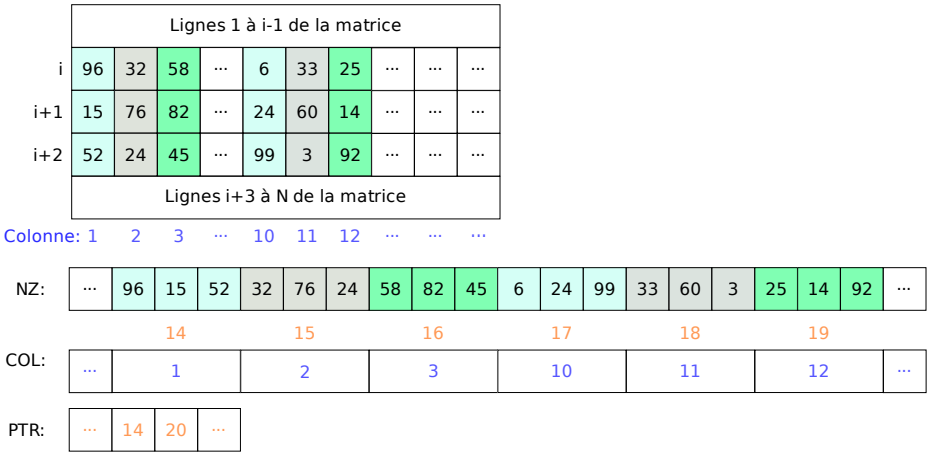

FIGURE 4 – Block Compressed Sparse Row, exemple d'une ligne contenant 2 blocs de taille 3x3

matrice sont coalescés.
Les performances du format JAD varie entre 18 et 25 Gflops sur GPU (Fig. 6). Pour ce format, la taille des blocs n'a pas d'importance, les matrices 20_16 et 30_8 obtiennent quasiment les même performances malgré la différence de taille de blocs.
Ce format est donc environ 1,5 fois plus efficace que le CSR pour nos matrices. Mais dans les tests que nous avons effectuées, nous n'avons pas pris en compte le temps de permutation des vecteurs, celui-ci pouvant être évité par un réordonnancement des inconnues dans la résolution.
Le format ELL [10] est un format proche du JAD, mais au lieu de trier les lignes par nombre d'éléments non nuls, le format ELL fait du remplissage en ajoutant des éléments nuls en fin de ligne pour avoir des lignes de même longueur, ce format est à privilégier au JAD dans le cas d'une matrice régulière. Ce format est utilisé par Cusp [3].

3.5. BCSR : Block Compress Sparse Row

Le BCSR est une adaptation du format CSR pour nos matrices. Il reprend le principe du CSR mais stocke des blocs de lignes (Fig. 4). Cette structure est naturellement adaptée à la structure

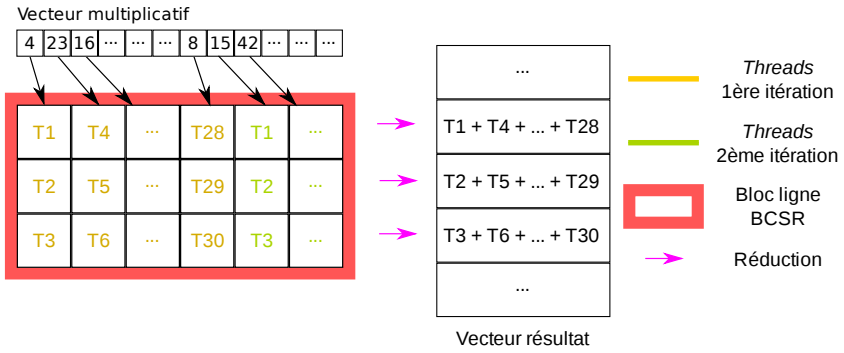

FIGURE 5 – Répartition des *Threads* à l'intérieur d'un bloc BCSR de taille 3x3 sur GPU, les *Threads* 31 et 32 sont exclus du travail.

de nos matrices et va permettre d'utiliser un *Warp* par bloc-ligne de manière plus efficace que le CSR. L'implémentation classique du BCSR correspond à un stockage continue des blocs par ligne [5], nous proposons de stocker les coefficients non nuls **par colonne** et de **stocker de manière scalaire** l'indice de chaque colonne : ainsi le produit d'un bloc-ligne par le vecteur multiplicatif peut se paralléliser indépendamment de la structure en bloc du bloc-ligne.

3.5.1. Implémentation sur GPU du SpMV

L'utilisation des blocs nous permet d'attribuer un *Warp* par bloc tout en garantissant assez de travail pour chaque *Thread*. A l'intérieur du bloc-ligne les *Threads* peuvent lire des données continues en mémoire et grâce au stockage par colonne, ils peuvent partagés en mémoire locale les données du vecteur multiplicatif (Fig. 5). Lorsque la hauteur h d'un bloc n'est pas un diviseur de 32, seuls les *Threads* ayant un identifiant inférieur au plus grand multiple de h inférieur à 32 effectuent des calculs, les autres ne font rien (Algo. 1). Par exemple, dans la figure 5, les *Threads* 31 et 32 sont inutilisés. A la fin d'un bloc ligne, une réduction dans la vecteur résultat doit être effectuée, le stockage par colonne permet d'effectuer plus efficacement la somme des produits de la ligne dans le vecteur résultat. En effet, à l'intérieur d'un bloc-ligne, un *Thread* ne participe qu'à une seule ligne ce qui réduit le nombre d'étapes pour effectuer la réduction finale. Pour faciliter l'identification des *Threads* appartenant à un même *Warp*, nous avons décider d'utiliser la première dimension de *threadIdx* comme identifiant à l'intérieur d'un *Warp* et la deuxième dimension de *threadIdx* (+ l'identifiant du *Block* CUDA) comme identifiant de *Warp*.

3.5.2. Résultats

Les performances sur un GPU sont doublées par rapport au format CSR. Ainsi nous atteignons de l'ordre de 45 Gflops sur un GPU alors que la puissance obtenue sur 8 coeurs CPU n'est que de 5 Gflops (cas 30_16 de la figure 6). Sur 3 GPUs, nous atteignons même 120 Gflops, soit 24 fois le résultat obtenu avec les CPUs de la station. Cette différence de performance avec le format CSR s'explique par trois améliorations :

- Le stockage par colonne qui permet le partage du vecteur multiplicatif ;
- une écriture coalescée dans le vecteur résultat à la fin d'une ligne ;
- une plus grande charge de travail par Warp.

3.6. Autres formats

Parmi les autres formats qui existent, nous pouvons citer :

Entrées : NZ, COL, PTR pour la matrice, X le vecteur multiplicatif
Sorties : Y le vecteur résultat
Const WARP_SIZE = 32;
Const BLOCK_SIZE = la taille des blocs de la matrice;
Const colonne_par_itération = (WARP_SIZE/BLOCK_SIZE);
si *threadIdx.x > colonne_par_itération*BLOCK_SIZE* **alors**
    `// Les threads inutilisés du warp`
    retourner
fin
Let ligne_courante = blockIdx.x + threadIdx.y;
Let colonne = threadIdx.x \ BLOCK_SIZE;
Let ligne_locale = threadIdx.x % BLOCK_SIZE;
Let sum_tmp = 0;
Let i = (PTR[ligne_courante]+colonne)*BLOCK_SIZE+ligne_locale;
Let j = PTR[ligne_courante]+colonne;
tant que *j < PTR[ligne_courante+1]* **faire**
    sum_tmp += NZ[i] * X[COL[j]];
    i += colonne_par_itération * BLOCK_SIZE;
    j += colonne_par_itération;
fin
Réduction des sum_tmp -> Y[ligne_courante+ligne_locale];

Algorithme 1: Pseudo-code du SpMV au format BCSR.

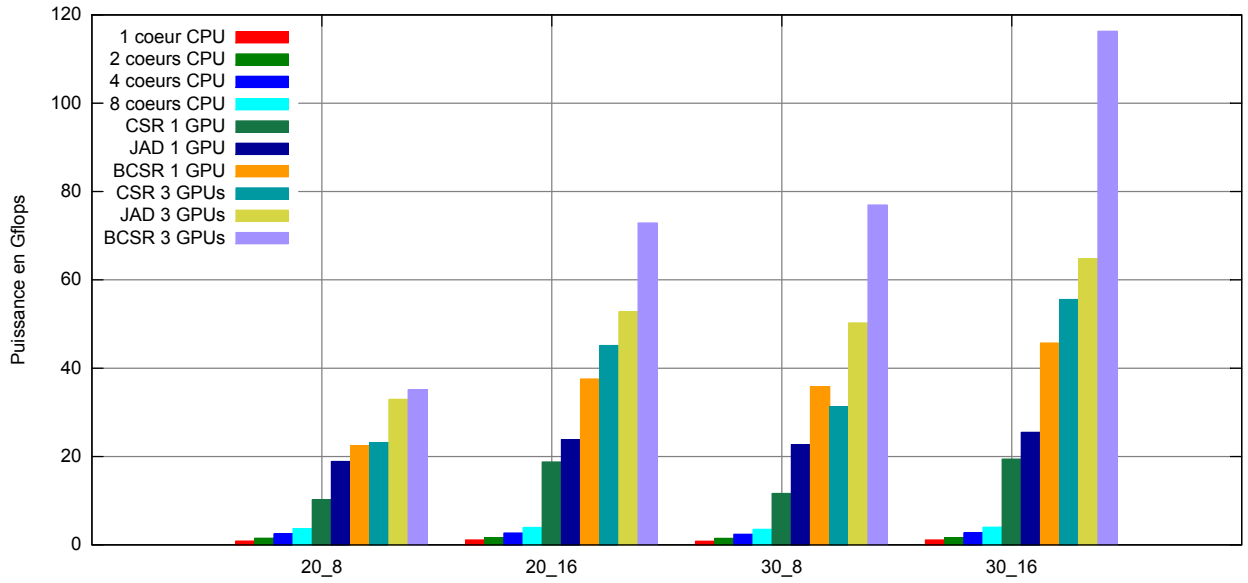

FIGURE 6 – Performance du SpMV sur les différents formats de stockage des matrices

— CSC : format identique au CSR mais le stockage se fait par colonne, le stockage complet de la matrice par colonne ne permet pas d'écrire un SpMV sur GPU à cause des accès concurrent en écriture au vecteur résultat ;
— DIA : seuls les diagonales contenant au moins un élément non nul sont stockées, dans notre cas les diagonales des bords de bloc ne contiennent pas assez de valeur, les performances sont équivalentes au CSR.

4. Conclusion

Le produit matrice/vecteur creux présente de nombreuses difficultés pour être porté sur GPU. De par son irrégularité, nous sommes obligés d'utiliser des tableaux d'indirections pour accéder aux éléments. Or l'utilisation de ces tableaux sur GPU est très coûteuse, c'est pour cela que nous avons cherché un format de matrices creuses pour la simulation de réservoir adapté aux GPUs.
Dans la littérature, nous avons trouvé le format JAD imaginé pour les anciennes machines vectorielles [4], ce format s'adapte très bien aux GPUs modernes.
Notre recherche d'un format adapté à nos matrices, nous a conduits au format BCSR dans lequel les blocs sont stockés par colonne. Malgré les bonnes performances du BCSR, les performances crêtes du GPU sont loin d'être atteintes. Cela s'explique par le fait que nous faisons trois accès mémoire, dont un indirect, pour seulement deux opérations.
Parmi les formats *standards* que nous avons essayés, le ELL et le JAD se sont montrés efficaces mais moins que le BCSR. Dans le cas général, l'utilisation du format JAD sur GPU offrira de meilleurs performances que le format CSR.
Maintenant que le SpMV est implémenté sur GPU, nous devons porter le reste des opérations du solveur sur GPU. Parmi ces opérations nous avons la factorisation ILU0 et les résolutions triangulaires, qui requièrent une gestion du parallélisme à grain fin. La version actuelle de CUDA ne nous permet pas d'implémenter ces deux opérations sur GPU, la prochaine version (CUDA 5) devrait le permettre.

Bibliographie

1. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2012.
2. Augonnet (C.), Thibault (S.), Namyst (R.) et Wacrenier (P.-A.). – StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation : Practice and Experience, Special Issue : Euro-Par 2009*, vol. 23, février 2011, pp. 187–198.
3. Bell (N.) et Garland (M.). – Cusp : Generic parallel algorithms for sparse matrix and graph computations, 2012. Version 0.3.0. http ://cusp-library.googlecode.com.
4. Blelloch (G. E.), Heroux (M. A.) et Zagha (M.). – *Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors*. – Rapport technique, 1993.
5. Choi (J. W.), Singh (A.) et Vuduc (R. W.). – Model-driven autotuning of sparse matrix-vector multiply on gpus. 2010, pp. 115–126.
6. Harris (M.). – *Optimizing Parallel Reduction in CUDA*. – NVidia, 2008.
7. Intel. – Math kernel library. http ://developer.intel.com/software/products/mkl/.
8. Li (R.) et Saad (Y.). – *GPU-Accelerated Preconditioned Iterative Linear Solvers*. – Rapport technique, 2010.
9. NVIDIA. – *CUDA Toolkit 4.2 CUSPARSE Library*, février 2012.
10. Vazquez (F.), Ortega (G.), Fernández (J.-J.) et Garzón" (E. M.). – Improving the performance of the sparse matrix vector product with GPUs. 2010.