

Multiplication matrice creuse - vecteur

Séminaire GPU

Eric Lombardi

LIRIS

Janvier 2016

Sommaire

- 1 Introduction
- 2 Matrice creuse au format CSR
- 3 SpMV-CSR, implémentation séquentielle
- 4 SpMV-CSR, implémentation parallèle
- 5 Matrice creuse au format ELL
- 6 SpMV-ELL, implémentation parallèle
- 7 SpMV-CSR, implémentation parallèle version 2
- 8 Méthode hybride ELL-COO
- 9 SpMV-ELL, implémentation cpu et effet du cache
- 10 Conclusion

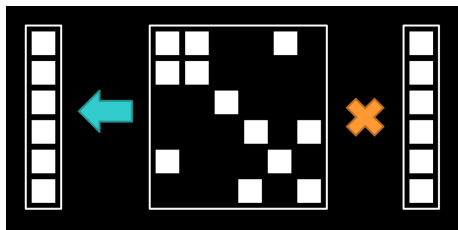
Introduction

- matrice creuse (sparse matrix)
- SpMV : Sparse Matrix Vector multiplication
- commentaire du code fourni (mult_mat_vect.cpp, mult_mat_vect_cuda.cu)
- usage : `./mult_mat_vect mat_10000x15000_0.50`

Introduction

Rappel : produit matrice-vecteur classique

```
// Matrice : m[], Vecteur : v[]  
// Sortie : y[] = m * v  
for(int r = 0; r < rNbr; r++)  
{  
    for(int c = 0; c < cNbr; c++)  
        y[r] += m[r*cNbr + c] * v[r];  
}
```



Nombreuses opérations inutiles (multiplications par zéro)

Matrice creuse au format CSR

- Compressed Sparse Row
- une matrice est stockée sous la forme de 3 tableaux :
 - ▶ values: valeurs non nulles
 - ▶ col_ind: index de la colonne de chaque valeur non nulle
 - ▶ row_ptr: pointeur vers la 1ère valeur de chaque ligne dans values[]
- exemple :

$$M = \begin{pmatrix} 0 & 0 & 12 & 13 \\ 21 & 22 & 23 & 0 \\ 0 & 32 & 0 & 34 \end{pmatrix}$$

values = [12 13 21 22 23 32 34]

col_ind = [2 3 0 1 2 1 3]

row_ptr = [0 2 5 7]

SpMV-CSR, implémentation séquentielle

Multiplication matrice creuse - vecteur :

```
// Matrice : values[], col_ind[], row_ptr[]
// Vecteur : v[]
for(int r = 0; r < rNbr; r++)
{
    int row_beg = row_ptr[r];
    int row_end = row_ptr[r+1];

    for(int i = row_beg; i < row_end; i++)
        y[r] += values[i] * v[col_ind[i]];
}
```

SpMV-CSR, implémentation parallèle

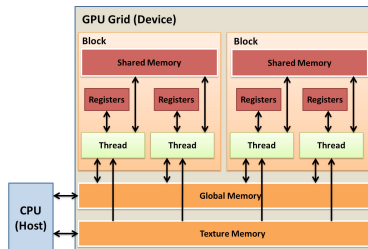
- un thread calcule une ligne de la multiplication
- la grille a une seule dimension
- la ligne traitée par le thread est l'indice du thread dans la grille
- tester si l'indice de la ligne est valide (extra-threads)

A vous de coder ...

SpMV-CSR, implémentation parallèle

*** Voir version avec solution ***

SpMV-CSR, implémentation parallèle



- améliorer la solution précédente en stockant le résultat des calculs intermédiaires dans un registre (utiliser une variable automatique dans le kernel)
- mais attention aux abus : "register spilling" (déversement des registres dans la mémoire locale) si trop de registres utilisés

A vous de coder ...

SpMV-CSR, implémentation parallèle

*** Voir version avec solution ***

SpMV-CSR, implémentation parallèle

Performances sur Xeon 2.4 Ghz / GTX 580 pour une matrice de dimensions 10000x15000 :

Méthode	Temps de calcul pur en ms
classique/cpu	294
CSR/cpu	123
CSR/GPU	98
CSR/GPU avec registres	54

SpMV-CSR, implémentation parallèle

Problèmes :

- les accès mémoire ne sont pas coalescents donc l'utilisation de la bande passante n'est pas optimisée
- 'branch divergence' dûe au nombre variable de valeurs non nulles par ligne

$$M = \begin{pmatrix} 0 & 0 & 12 & 13 \\ 21 & 22 & 23 & 0 \\ 0 & 32 & 0 & 34 \end{pmatrix}$$

values = [12 13 21 22 23 32 34]

	<i>thread</i> #0	<i>thread</i> #1	<i>thread</i> #2
<i>iter</i> #1	12	21	32
<i>iter</i> #2	13	22	34
<i>iter</i> #3	x	23	x

Matrice creuse au format ELL

- reconstituer des lignes de longueurs identiques en réintroduisant des zéros
- stocker la matrice dans l'ordre 'column major' pour que des threads successifs accèdent à des données contigües en mémoire
- exemple :

$$\begin{pmatrix} 0 & 0 & 12 & 13 \\ 21 & 22 & 23 & 0 \\ 0 & 32 & 0 & 34 \end{pmatrix} \Rightarrow \begin{pmatrix} 12 & 13 & 0 \\ 21 & 22 & 23 \\ 32 & 34 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 12 & 21 & 32 \\ 13 & 22 & 34 \\ 0 & 23 & 0 \end{pmatrix}$$

values = [12 21 32 13 22 34 0 23 0]

col_ind = [2 0 1 3 1 3 2]

- le tableau *row_ptr* devient inutile, la ligne *n* commence à la valeur *values*[*n*]
- la *kième* valeur de la ligne *n* est *values*[*n* + *k* * *h*], où *h* est le nombre de lignes

SpMV-ELL, implémentation parallèle

- un thread calcule une ligne de la multiplication
- la grille a une seule dimension
- la ligne traitée par le thread est l'indice du thread dans la grille
- tester si l'indice de la ligne est valide (extra-threads)

A vous de coder ...

SpMV-ELL, implémentation parallèle

*** Voir version avec solution ***

SpMV-ELL, implémentation parallèle

Performances sur Xeon 2.4 Ghz / GTX 580 pour une matrice de dimensions 10000x15000 :

Méthode	Temps de calcul pur en ms
classique/cpu	294
CSR/cpu	123
CSR/GPU	98
CSR/GPU avec registres	54
ELL/GPU	8

SpMV-ELL, implémentation parallèle

Avantages :

- accès mémoire coalescents, utilisation optimale de la bande passante
- pas de 'branch divergence'
- dans notre exemple

values = [12 21 32 13 22 34 0 23 0]

	<i>thread</i> #0	<i>thread</i> #1	<i>thread</i> #2
<i>iter</i> #1	12	21	32
<i>iter</i> #2	13	22	34
<i>iter</i> #3	0	23	0

Problème :

- une ligne beaucoup plus longue que toutes les autres fait ré-introduire de nombreux zéros

SpMV-CSR, implémentation parallèle version 2

- une solution pour obtenir des accès mémoire coalescents avec CSR : un WARP (et non un thread) calcule une ligne de la multiplication
- le résultat final pour une ligne est obtenu par une opération de réduction à l'intérieur du warp

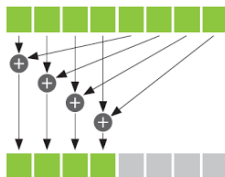


Figure 5.4 One step of a summation reduction

- utiliser la shared memory pour faire collaborer les threads

A vous de coder ...

SpMV-CSR, implémentation parallèle version 2

*** Voir version avec solution ***

SpMV-CSR, implémentation parallèle version 2

*** Voir version avec solution ***

SpMV-CSR, implémentation parallèle version 2

- tuning : utiliser NVVP pour dimensionner les blocs de façon à améliorer l'"occupancy"
- "occupancy" faible (16%) dans le cas 1 warp/bloc sur GTX 580

Results

⚠ GPU Utilization Is Limited By Block Size


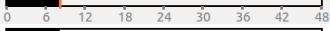

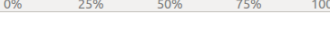
The kernel has a block size of 32 threads. This block size is likely preventing the kernel from fully utilizing the GPU. Device "GeForce GTX 580" can simultaneously execute up to 8 blocks on each SM. Because each block uses 1 warp to execute the block's 32 threads, the kernel is using only 8 warps on each SM. Chart "Varying Block Size" below shows how changing the block size will change the number of warps that can execute on each SM.

Optimization: Increase the number of threads in each block to increase the number of warps that can execute on each SM.

[More...](#)

Variable Achieved Theoretical Device Limit Grid Size: [15000,1,1] (15000 blocks) Block Size: [32,1,1] (3

Occupancy Per SM

Active Blocks		8	8	
Active Warps	7,94	8	48	
Active Threads		256	1536	
Occupancy	16,5%	16,7%	100%	




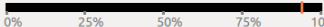
SpMV-CSR, implémentation parallèle version 2

- "occupancy" forte (93%) dans le cas 6 warps/bloc sur GTX 580

i Occupancy Is Not Limiting Kernel Performance

The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU.

[More...](#)

Variable	Achieved	Theoretical	Device Limit	Grid Size: [2500,1,1] (2500 blocks)Block Size: [192,1,1] (19
Occupancy Per SM				
Active Blocks		8	8	
Active Warps	44,88	48	48	
Active Threads		1536	1536	
Occupancy	93,5%	100%	100%	

SpMV-CSR, implémentation parallèle version 2

Performances sur Xeon 2.4 Ghz / GTX 580 pour une matrice de dimensions 10000x15000 :

Méthode	Temps de calcul pur en ms
classique/cpu	294
CSR/cpu	123
CSR/GPU	98
CSR/GPU avec registres	54
ELL/GPU	8.4
CSR/GPU v2, 1 warp/block	7.4
CSR/GPU v2, 6 warps/block	4.5

Méthode hybride ELL-COO

- exemple :

$$\begin{pmatrix} 0 & 0 & 12 & 13 & 0 \\ 21 & 22 & 23 & 0 & 25 \\ 0 & 32 & 0 & 34 & 0 \end{pmatrix} \xRightarrow{ELL} \begin{pmatrix} 12 & 13 & 0 & 0 \\ 21 & 22 & 23 & 25 \\ 32 & 34 & 0 & 0 \end{pmatrix}$$

- méthode hybride :

- ▶ stocker la partie **régulière** au format ELL
- ▶ stocker les **autres valeurs non nulles** au format COO (COOrdinates)

values = [23 25]

col_ind = [2 4]

row_ind = [1 1]

- ▶ le GPU traite la partie ELL
- ▶ puis le CPU traite séquentiellement la partie COO

```
for (int i = 0; i < valuesNbr; i++)  
    y[row_ind[i]] += values[i] * v[col_ind[i]];
```


SpMV-ELL, implémentation cpu et effet du cache

- implémenter SPMV-ELL sur cpu une 1ère fois en faisant la boucle extérieure sur les lignes
- implémenter SPMV-ELL sur cpu une 2nde fois en faisant la boucle extérieure sur les colonnes
- comparer les performances

SpMV-ELL, implémentation cpu et effet du cache

Performances sur Xeon 2.4 Ghz / GTX 580 pour une matrice de dimensions 10000x15000 :

Méthode	Temps de calcul pur en ms
classique/cpu	294
CSR/cpu	123
CSR/GPU	98
CSR/GPU avec registres	54
ELL/GPU	8.4
CSR/GPU v2, 1 warp/block	7.4
CSR/GPU v2, 6 warps/block	4.5
ELL/cpu par ligne	486
ELL/cpu par colonne	134

Conclusion

Résumé des techniques que nous avons appliquées :

- utiliser les registres du GPU pour les calculs intermédiaires (variables automatiques dans le kernel), mais attention au "register spilling"
- rendre les accès à la mémoire globale coalescents
- limiter les "branch divergence" à l'intérieur d'un bloc (plus précisément à l'intérieur d'un warp)
- technique de la reduction à l'intérieur d'un bloc ou d'un warp (nécessite d'utiliser la shared memory)

Références :

- Livre "Programming Massively Parallel Processors" par D.B.Kirk et W.W Hwu
- Efficient Sparse Matrix-Vector Multiplication on CUDA, Nathan Bell and Michael Garland, 2008