TP prise en main de CUDA : introduction à la programmation des GPU

P. Kestener

25 novembre 2016

1 Prise en main des outils de développement CUDA

1.1 Environnement

On utilisera les stations de travail sous Fedora de la salle de TP pour se connecter à distance vers le serveur de calcul : **`rhum`** (addresse IP 147.250.33.51), utilisant des GPU de type **Kepler K80**.

Chaque binôme se voit attribuer un login et un mot de passe sur rhum [1].

— login : b133gr[00-19]

— mot de passe : `43xtc$24`

L'objectif du TP est de se familiariser avec le flot de compilation NVIDIA/NVCC et d'apprendre les bases du modèle de programmation CUDA.

NOTE IMPORTANTE :
Le compilateurs nvcc CUDA 8.0 est installé dans le répertoire `/usr/local/cuda-8.0`. Les variables d'environnement `PATH` et `LD_LIBRARY_PATH` sont déjà prédéfinies pour une bonne utilisation du *toolkit* 8.0 (i.e. chaîne de compilation) CUDA.

On utilisera l'aide en ligne de CUDA :
http://docs.nvidia.com/cuda/cuda-runtime-api/index.html

1.2 Compilation du SDK CUDA

Sur la machine `rhum`, copiez les exemples du SDK CUDA [2], i.e. dans votre `HOME`, tapez la commande suivante :

```
cuda-install-samples-8.0.sh .
```

Le SDK contient des exemples d'applications et de programmes en CUDA/C. Chaque exemple peut être compilé indépendamment des autres en se plaçant dans le sous-répertoire correspondant et en tappant `make`.

1. exemple de connection : `ssh -X b133gr01@rhum`
2. SDK : Software Development Kit

1.3 deviceQuery

Le code source de l'exemple `deviceQuery` se situe dans le sous-répertoire `1_Utilities` du SDK.

Tappez `cd 1_Utilities/deviceQuery` pour aller dans le répertoire source de cet exemple et ensuite `make`. Exécutez l'exemple `deviceQuery` qui permet d'avoir toutes les informations sur le matériel disponible [3].

On pourra activer le mode `verbose` de compilation :

```
make clean
make verbose=1
```

On pourra constater ici que le compilateur `nvcc` n'est même pas appellé ; on ne compile pas de *kernel* CUDA, on ne fait qu'appeller des routines qui interrogent le driver de la carte graphique.

1. Exécuter `deviceQuery`.
2. De combien de GPU dispose-t-on sur la machine `rhum` ?
3. De combien de *streaming multiprocessor* sont faits les GPU de `rhum` ?
4. Utiliser l'information sur le bus mémoire pour en déduire la valeur de la bande passante mémoire crête en GBytes/s.

Pour info, une liste à jour des GPU NVIDIA et leurs caractéristiques :
http://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

1.4 HelloWorld en CUDA/C

Utilisation des variables intrinsèques `threadIdx` et `blockIdx`

Le but de l'exercise est d'écrire un premier kernel CUDA et d'apprendre à utiliser les variables intrinsèques du modèle de programmation qui dimensionne la grille de bloc de *threads* et les blocs de *threads* eux-même.

Récupérez le fichier `/home/gpu/kestener/M2S_2016/TP_CUDA/basic/helloworld.cu`. Il s'agit de l'exemple qui consiste simplement à additionner deux tableaux d'entier et à afficher le résultat (vu en cours).

1. Tapper `nvcc -help`, et explorer l'aide en ligne du compilateur nvidia
2. Compiler le code source `helloworld.cu` avec le compilateur `nvcc` en utilisant l'option `-arch=sm_13` ; à quoi sert cette option ? Que se passe-t-il si on omet cette option ?
3. Exécuter le programme avec les paramètres par défaut : le *kernel* `add` avec la configuration 1 bloc de *threads*, 1 *thread* par bloc.
4. Recompiler avec l'option `-keep`. Illustration du flot de compilation.
5. Manipulation :

3. Complément d'information :
http://devblogs.nvidia.com/parallelforall/how-query-device-properties-and-handle-errors-cuda-cc/

— Pour les petites valeurs de N (taille des tableaux), afficher à l'écran le résultat du calcul. Vérifier que le code écrit peut être exécuté de multiple configurations, par exemple en 2 blocs de *threads* avec N/2 *threads* par bloc (en donnant évidemment les mêmes résultats).
— Augmenter la taille du tableau d'entrée jusqu'à au moins 16 millions. Que constatez-vous ? Quelle limitation matérielle explique ce comportement (utiliser les informations de `deviceQuery`) ?
— Augmenter `nbThread`. Que constatez-vous ? Comment l'expliquer ?

1.5 Bande passante mémoire : GPU-GPU et CPU-GPU

On se concentre dans un premier temps sur la bande passante GPU-GPU.

1. Reprendre le code de l'exemple `helloworld`.
2. Ecrire le code d'un kernel CUDA qui prend en argument deux pointeurs vers des tableaux alloués en mémoire globale (i.e. alloués avec `cudaMalloc`) et donc la fonction est de recopier le premier tableau dans le deuxième.
   Faites en sorte que le programme prenne en argument de la ligne de commande le nombre d'éléments des tableaux à allouer [4].
3. Dans un premier temps, on adopte la même stratégie que pour `helloworld` : 1 *thread* par élement du tableau à copier. Ecrire le code du kernel `copy`.
4. Dans un deuxième temps, comment modifier le kernel CUDA si on considère que la taille de la grille de *thread* est de taille fixe (indépendante de la taille des tableaux) ? Ecrire le code du kernel `copy2` correspondant. On définira le nombre de blocs comme un multiple du nombre de *streaming multiprocessor*.

```
// utiliser cudaGetDeviceProperties
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, 0);  // pour le device 0
// utiliser prop.multiProcessorCount
```

5. Utiliser les *timer* pour mesurer les temps d'exécution et afficher la bande passante mémoire en GBytes par seconde.

```
// exemple d'utilisation
#include "CudaTimer.h"
...
CudaTimer timer;
timer.start();
// do something
timer.stop();
// use timer.elapsed_in_second() to get elapsed time
```

4. Utiliser la routine `atoi` pour convertir une chaîne de caractère en entier.

6. Exécuter le code plusieurs fois pour différentes tailles de tableau et pour les 2 versions du kernel copy. Que constatez-vous sur la valeur de la bande passante mémoire ?
7. Comparer avec la bande passante maximale possible :

```
// utiliser cudaGetDeviceProperties
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, 0);  // pour le device 0
printf("  Peak Memory Bandwidth (GB/s): %f\n",
2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
```

On revisite l'étude de la bande passante mémoire (interne au GPU et Pci-Express).

1. Compiler l'exemple `bandwidthTest` du SDK CUDA/C et lire l'aide (`./bandwidthTest --help`)
2. Exécuter la version *release* avec l'option `--mode=quick`
3. Vérifier les ordres de grandeur discutés en cours des 3 différentes bandes passantes mémoire.
4. Exécuter l'exemple avec l'option *range* pour des tailles de transfert de données comprises entre 0 et 100kB par pas de 10kB. Que constatez-vous ?
5. Reprendre l'exemple en utilisant la mémoire dite *pinned*.

1.6 Manipulation en mémoire partagée

Compiler et exécuter l'exemple *transposition* du SDK CUDA/C.

Des explications sur les notions d'accès coalescent à la mémoire et de conflit de banc mémoire seront données pendant le TP.

1.7 SAXPY

SAXPY est une des fonctions de base que l'on trouve dans les librairies d'algèbre linéaire de type BLAS [5] qui réalise l'opération $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$, où $\mathbf{x}$ et $\mathbf{y}$ sont des vecteurs 1D de réels simple précision.

Récupérer l'archive `/home/gpu/kestener/M2S_2016/TP_CUDA/blas/saxpy_cuda_c.tar.gz`. Utiliser le Makefile pour compiler l'exemple. La compilation fournit un exécutable `saxpy` qui calcule la fonction *saxpy* de 3 manières :

1. version séquentielle sur le CPU
2. version parallèle sur le GPU avec kernel CUDA écrit *à la main*
3. version parallèle sur le GPU en utilisant les routines de la librairie cuBlas [6]

Comparer les performances des 3 versions en faisant varier la taille du tableau d'entrée (paramètre `N` en début du fichier).

5. http://en.wikipedia.org/wiki/SAXPY
6. cuBlas est fournie par NVIDIA en installant le toolkit CUDA. Cf /usr/local/cuda-8.0/doc/pdf/CUDA_CUBLAS_Users_Guide.pdf.

1.8 Optimisation dirigée par l'analyse du code

Cet exemple permet de comprendre comment on peut déterminer quel est le facteur limitant dans l'exécution d'un kernel CUDA : les transferts de/vers la mémoire globale (i.e. externe au GPU) ou le flot d'instruction. On utilise pour cela une technique qui étant donné un kernel, consiste à éteindre l'une ou l'autre de ces deux composantes.

- Examiner et exécuter le code CUDA/C `limitingFactor.cu`. Vous trouverez au début de chaque fichier les instructions pour compiler.
- Mesurer les temps d'exécution en utilisant la commande `nvprof` placée devant l'exécutable.
- Comparer les temps d'exécution des 3 kernels : `memory_and_math`, `memory`, `math`. Est-ce que la somme des temps associés à `memory` et `math` est égale à celui de `memory_and_math` ? Expliquer, interpréter.

Ne pas hésiter à demander des explications pendant le TP et/ou lire la section 2.2 du livre de Ruetsch et Fatica [7].

Voir également les planches de G. Ziegler, *Analysis-driven Optimization*.

1.9 Algorithmes de reduction

[8]

Placez-vous dans le répertoire des sources de l'exemple `reduction` du SDK CUDA (`6_Advanced/reduction`).

1. Éditez le code source pour comprendre comment appeller les différentes versions de `kernel`.
2. Ouvrez le document PDF (sous répertoire `doc` dans les sources) pour avoir des explications claires sur les différentes `kernel`.
3. Le document PDF nous informe que la bande passante mémoire maximale entre le GPU et sa SDRAM externe est de 86.4 GBytes/s pour le GPU FeForce GTX 8800 (toute première version de l'architecture CUDA, fin 2006). Quelle la valeur correspondante pour notre GPU (Tesla S1070) ? Utiliser les informations de l'exemple `deviceQuery` (celui du SDK).
4. Executer l'exemple reduction pour les différents *kernels*, retrouve-t-on les chiffres indiqués dans le document ?

2 Équation de la chaleur 2D/3D sur GPU

On se propose de résoudre l'équation de la chaleur (cf annexe C) par la méthode des différences finies sur une grille cartésienne en 2D puis 3D en explorant plusieurs variantes d'implantations avec CUDA.

7. CUDA Fortran for Scientists and Engineers, Elsevier, 2013.
8. Cet exercice peut être laissé de coté dans un premier temps.

L'équation de la chaleur représente l'archétype du problème parallélisable [9] par décomposition de domaine. Le travail proposé permet d'explorer les diverses façons d'implanter cette décomposition dans le modèle de programmation CUDA.

1. Décompresser l'archive `TP_CUDA/chaleur/heat2d3d_ensta2013_sujet.tar.gz` sur votre compte local.
2. Tout au long de l'exercice, il faudra éditer le `Makefile` pour permettre la compilation des différentes variantes.
3. Une version de référence en C++ est fournie. Elle est constituée de 4 fichiers :
   (a) `heat_solver_cpu.cpp` : contient le `main`, les allocations mémoire et les sorties dans des fichiers pour visualisation,
   (b) `heat_kernel_cpu.cpp` : les routines de résolution du schéma numérique (à l'ordre 2 et à l'ordre 4),
   (c) `param.cpp` : definition des structures de données pour paramètres du problème (taille des tableaux, nombre de pas de temps, sortie graphique, etc...),
   (d) `heatEqSolver.par` : exemple de fichier de paramètres (utilisant le format GetPot)
   (e) `misc.cpp` : les routines utiles annexes (initialisation des tableaux).
4. Compiler, exécuter la version de référence avec les valeurs par défaut des paramètres et de la condition initiale.
5. Reprendre la question précédente en modifiant la condition initiale/condition de bord (fichier `misc.cpp`, routine `initCondition2D`, mettre par exemple tous les bords à 0 sauf un bord à 1) et en calculant 1000 pas de temps avec une sortie graphique tous les 100 pas de temps sur un domaine 2D de taille 256×256. Visualiser les résultats (images PNG ou fichiers VTK) en vous aidant des informaions contenues dans le sous-répertoire `visu`.

On se propose de porter cet algorithme sur GPU graduellement en terme d'optimisation.

version naïve 2D Dans un premier temps, on n'utilisera pas la mémoire partagée du GPU, tous les accès mémoire se feront à partir des tableaux situés en mémoire globale.

1. Editer le fichier `heat2d_solver_gpu_naive.cu`, et remplir de façon appropriée les endroits signalés par `TODO`. Consulter la documentation en ligne de CUDA pour savoir comment utiliser les routines `cudaMalloc` et `cudaMemcpy` [10].
2. Editer de même le fichier `heat2d_kernel_gpu_naive.cu` qui contient le code du *kernel* exécuté sur le GPU
3. Editer le `Makefile` et décommenter les lignes correspondantes à la compilation de cette version.

9. Voir le site http://www.cs.uiuc.edu/homes/snir/PPP/ pour avoir plus d'informations sur les différents archétypes de problèmes parallèles
10. Voir la doc en ligne CUDA : `/usr/local/cuda-8.0/doc/html/index.html` sur la machine `rhum`

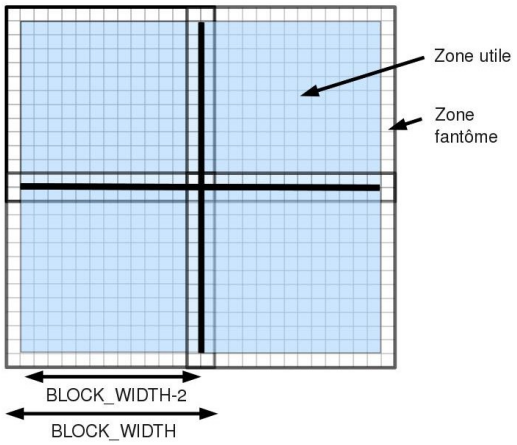

FIGURE 1 – Schéma représentant la grille de blocs de threads utilisée dans le kernel défini dans le fichier `heat2d_kernel_gpu_shmem1.cu` ; les blocs de thread adjacent se recouvre sur une zone de largeur 2.

4. Vérifier que le code est fonctionnel (en comparant aux résultats de la version de référence).
5. Comparer les performances de cette première version sur des tableaux qui ont des tailles en puissance de 2 et non-puissance de 2.

version simple avec mémoire partagée 2D Reprendre les questions précédentes en utilisant les fichiers `heat2d_solver_gpu_shmem1.cu` et `heat2d_kernel_gpu_shmem1.cu`. La mémoire coté GPU est à présent allouée avec les routines `cudaMallocPitch` (voir la documentation de l'API CUDA : `/usr/local/cuda/doc/html/group__CUDART__MEMORY.html`) pour respecter les alignements mémoire. On se propose dans cette version d'utiliser la mémoire partagée (meilleurs temps d'accès) ; en revanche la mémoire partagée étant privée à un bloc de *threads*, il est nécessaire d'utiliser un découpage du domaine en blocs qui se chevauchent (voir la figure 1) pour assurer la continuité de l'accès aux données (voir les explications données pendant le TP). On pourra s'aider de ce schéma pour déterminer quel *thread* accède à quelle case mémoire.

1. Après avoir testé cette version, que se passe-t-il en terme de performance si on échange les rôles des indices de *thread* `tx` et `ty` ? Expliquez.

version simple avec mémoire partagée 2D - variante Il s'agit d'une légère variante de la version précédente. A présent, on alloue un tableau en mémoire partagée plus grand que la taille des blocs de threads pour tenir compte des cellules *fantômes* qui permettent aux blocs de threads de travailler sur des blocs complètement indépendants. Ne pas hésiter à demander des explications pendant le TP.

version optimisée avec mémoire partagée 2D (facultatif) Afin de profiter au maximun de la copie en mémoire partagée, on demande à chaque *threads* de calculer plusieurs cellules du tableau de sortie. Le *kernel* est divisé en 2 *sous-kernels* travaillant respectivement sur les lignes puis les colonnes.

1. Le travail demandé consiste à écrire le code du *kernel* travaillant sur les colonnes en s'inspirant de celui travaillant sur les lignes. On éditera pour cela le fichier `heat2d_kernel_gpu_shmem3.cu`. On pourra commencer par analyser le code du *kernel* sur les lignes pour comprendre le schéma d'accès aux données (faire un dessin).
2. On comparera les performances globales de cette version avec les 2 autres en particulier pour des tailles de domaine qui ne sont pas des puissances de 2.

version 3D naïve Les limitations de CUDA font que l'on ne peut pas créer des grilles de bloc de threads de n'importe quelle dimension dans la direction `z`. On se propose de reprendre la version 2D naïve et de l'étendre en 3D, chaque *thread* s'occupant de toute une colonne suivant `z`

1. Ecrire le code du *kernel* `heat3d_ftcs_naive_kernel` dans le fichier `heat3d_kernel_gpu_naive.cu`
2. Vérifier qu'il est fonctionnel en comparant les résultats avec ceux de la version de référence.

version 3D optimisée On reprend la version 2D qui utilise la mémoire partagée. La partagée ayant une taille maximale qui ne permet pas d'allouer dans la direction `z` la même taille que dans les directions `x` et `y`, on se contente d'allouer le tableau suivant :

```
__shared__ float shmem[3][BLOCK_HEIGHT][BLOCK_WIDTH];
```

qui contient les données de 3 plans en `z` consécutifs. On peut ainsi accéder à toutes les cases mémoires voisines nécessaires à la mise à jour de $\phi_{i,j,k}^{n+1}$ (cf Eq. (4)). A chaque fois qu'un plan est calculé complètement (dans un bloc), on avance en permuttant les indexes des plans (`shmem[z]`) et en chargeant les données du plan suivant dans `shmem[3]`.

1. Remplir les trous laissés dans le *kernel* `heat3d_ftcs_sharedmem_kernel` dans le fichier `heat3d_kernel_gpu_shmem1.cu` ; vérifier la fonctionalité du programme et tester ses performances.
2. On pourra également développer une version où seul le plan médian est stocké en mémoire partagée, les données des plans $z-1$ et $z+1$ étant mise dans des registres (variables locales du thread courant). Cette version présente l'avantage de mieux utiliser les resources matérielles (**équilibrer registres et mémoire partagée**).

Pour aller plus loin...

Etude de l'influence de certains paramètres sur les performances CPU/GPU.

1. impact de l'ordre du schéma numérique (ordre 2 ou 4).
2. impact de la double précision (ajouter le symbol `USE_DOUBLE` aux flags de compilation, voir le Makefile en tête de fichier)
3. impact de la dimension des tableaux de simulation
   — Essayer la version 3D naïve avec des tailles de tableaux en puissance de 2 et non-puissance de 2.
   — Faîtes la même chose avec la version 3D en mémoire partagée. Que constatez-vous ?
4. impact de la dimension des blocs de *threads* et de leur forme (bloc carré ou alongé suivant x)

Interoperabilité CUDA-OpenGL On se propose d'illustrer l'inter-opérabilité entre CUDA et OpenGL, c'est à dire la possibilité d'échanger *facilement* des données entre ces deux contexts sans repasser intermédiairement par le CPU. On va voir comment on peut transmettre un buffer CUDA existant dans la mémoire globale du GPU aux routines GLUT/OpenGL chargées de l'affichage graphique.

Voir le code source `main_glut.cpp` qui sert aux deux versions :
— version CPU : calcul sur le CPU et affichage OpenGL
— version GPU : calcul sur le GPU (CUDA) et affichage OpenGL

On pourra voir comment mettre en œuvre le mécanisme de *callbacks* de la librairie GLUT qui permet via le clavier et la souris d'interagir avec le calcul.

A API CUDA

On pourra se servir de la documentation officielle de l'API CUDA :
`/usr/local/cuda/doc/html/index.html`

B Editeur de texte

Pour avoir la coloration syntaxique du c++ dans emacs sur les fichiers d'extension .cu, tapper : `Echap-x c++-mode` et `entrée`.

C Schéma numérique pour l'équation de la chaleur

Dans la section 2, on utilise la méthode des différences finies et plus précisément un schéma explicite de type FTCS [11] pour résoudre l'équation de la chaleur

$$\partial_t \phi = D \left[\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} \right], \; 0 \leq x \leq L_x, \; 0 \leq y \leq L_y, \; t \geq 0 \tag{1}$$

sur un domaine rectangulaire muni de conditions de bord et d'une condition initiale.

11. Forward Time Centered Space

On étudie deux versions du schéma, qui correspond à deux approximations de la dérivée seconde :

— différence centrée à 3 points, ordre 2 (erreur en h^2)

$$\begin{aligned} D^2\phi(x_0) &= \frac{1}{h^2}\left[\phi(x_0-h) - 2\phi(x_0) + \phi(x_0+h)\right] \\ &= \phi''(x_0) + \frac{1}{12}h^2\phi^{(4)}(x_0) + O(h^4) \end{aligned}$$

— différence centrée à 5 points, ordre 4 (erreur en h^4)

$$\begin{aligned} D^2\phi(x_0) &= \frac{1}{12h^2}\left[-\phi(x_0-2h) + 16\phi(x_0-h) - 30\phi(x_0) + 16\phi(x_0+h) - \phi(x_0+2h)\right] \\ &= \phi''(x_0) + \frac{1}{90}h^4\phi^6(x_0) + O(h^6) \end{aligned}$$

Le schéma FTCS (à 3 points) met à jour le tableau 2D $\phi_{i,j}$ (ou 3D $\phi_{i,j,k}$) au temps $t = t_{n+1} = (n+1)\Delta t$ par la relation :

$$\phi_{i,j}^{n+1} = \phi_{i,j}^n + D\frac{\Delta t}{\Delta x^2}\left[(\phi_{i+1,j}^n - 2\phi_{i,j}^n + \phi_{i-1,j}^n) + (\phi_{i,j+1}^n - 2\phi_{i,j}^n + \phi_{i,j-1}^n)\right] \quad (2)$$

$$= R_2\phi_{i,j}^n + R\left[\phi_{i+1,j}^n + \phi_{i-1,j}^n + \phi_{i,j+1}^n + \phi_{i,j-1}^n\right] \quad (3)$$

où $R = D\Delta t/\Delta x^2$ et $R_2 = 1 - 4R$ (en 2D). De manière similaire en 3D, on trouve

$$\phi_{i,j,k}^{n+1} = R_3\phi_{i,j,k}^n + R\left[\phi_{i+1,j,k}^n + \phi_{i-1,j,k}^n + \phi_{i,j+1,k}^n + \phi_{i,j-1,k}^n + \phi_{i,j,k+1}^n + \phi_{i,j,k-1}^n\right] \quad (4)$$

avec $R_3 = 1 - 6R$ en 3D.

N.B. : En pratique, pour résoudre l'équation de la chaleur, on préfére utiliser un schéma implicite comme celui de Crank-Nicolson, qui conduit à l'inversion d'un système linéaire tridiagonal et qui possède l'avantage d'être inconditionnellement stable.

On pourra consulter les rérefences suivantes :

— *Finite Difference Methods for Ordinary and Partial Differential Equations*, R. J. LeVeque, SIAM, 2007.
http://www.amath.washington.edu/~rjl/booksnotes.html

— Sur l'implantation GPU d'un algorithme de résolution de système tridiagonal :
http://research.nvidia.com/publication/fast-tridiagonal-solvers-gpu