

Introduction à la programmation // sur GPUs en CUDA et Python

Denis Robilliard

Équipe CAMOME: C. Fonlupt, V. Marion-Poty, A. Boumaza

LISIC — ULCO
Univ Lille Nord de France
BP 719, F-62228 Calais Cedex, France
Email: robilliard@lisis.univ-littoral.fr

Séminaire LISIC 19/05/2011

Introduction

Cadre de l'exposé

- Historique : suite à l'exposé de François Rousselle du 17 février 2011
- Objectif : présenter quelques bases de la programmation // sur GPU
- Matériel : machine "baru" 8 cartes Tesla C2050, fabricant Nvidia
- Langages : CUDA (seulement sur matériel Nvidia), Python
- ~~Application : "les nombres de Schur", problème de combinatoire~~

Note

On ne s'intéressera pas aux aspects calculs pour le graphisme

Plan

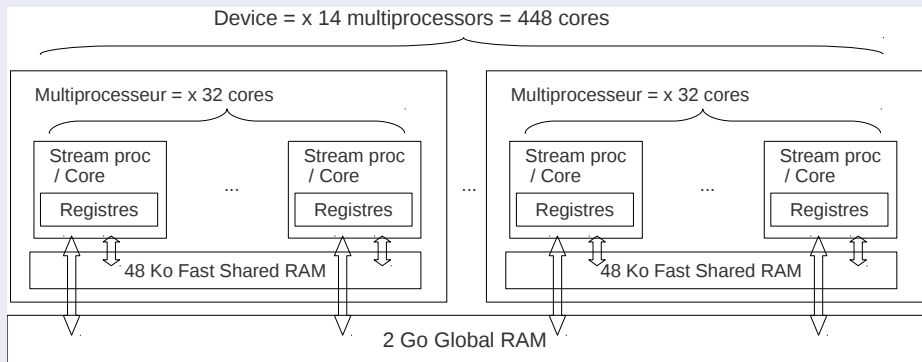
- 1 Architecture
- 2 Modèle de programmation
- 3 Principes de programmation
- 4 Exemple : produit terme à terme
- 5 Exemple : réduction
- 6 Conclusions

Architecture matérielle –1

Rappel

- Valable uniquement pour GPUs Nvidia post-2006 (GPU G-80 et suivants)
- Modèle de programmation vu à travers le langage CUDA

Schéma de l'architecture d'une carte (device) C2050



Constitution d'un multi-processeur : les cœurs

- Un cœur (*core / stream processor*) a son propre jeu de registres
- Un multiprocesseur = 32 cœurs + 48 Ko de mémoire partagée rapide (aussi rapide que les registres — accès en 4 cycles)
- Ils accèdent tous à la mémoire globale de la carte (2 Go — accès en 400 cycles)
- Les 32 cœurs d'un multiprocesseur sont en SIMD (*Single Instruction, Multiple Data*) : même instruction "en même temps", potentiellement sur des données différentes
- Cas des structures conditionnelles : tous les cœurs ne doivent pas forcément faire la même chose \Rightarrow certains cœurs seront oisifs (*idle*) et on parle de *divergence* (= perte d'efficacité)

Architecture matérielle –3

Multiprocesseurs d'une carte

- Les 14 multiprocesseurs (*multiprocessors*) ont leur propre pointeur d'instruction
- Fonctionnement en mode SPMD (*Single Program, Multiple Data*) : le même programme, pas forcément la même instruction
- ⇒ c'est donc en fait du MIMD (*Multiple Instruction, Multiple Data*)
- Pas de divergence entre cœurs de différents multiprocesseurs
- Ils peuvent être oisifs s'il n'y a pas assez de fils de calculs à traiter...

Cartes de l'hôte *baru*

- Le serveur *baru* comprend 8 cartes (*devices*) C2050, asynchrones.
- Chaque carte possède sa propre mémoire globale privée
- Un programme se décompose toujours en deux parties : une sur l'hôte (C, Python) qui va allouer et charger la mémoire d'une/plusieurs cartes pour y exécuter le/les (*kernels*) et récupérer le résultat sur l'hôte

Modèle de programmation –1

Blocs et fils de calcul

- Un *fil de calcul* (*thread*) est une instance du kernel exécutée sur un cœur
- Un *bloc* est un ensemble de fils exécutés :
 - ▶ **en //, donc a priori "en même temps"**
 - ▶ **sur un même multiprocesseur**
- Attention, on peut exécuter jusqu'à 1024 fils dans un bloc (seulement 32 cœurs !) — donc pas vraiment "en même temps" (voir suite)
- Les blocs sont ordonnancés en // sur les multiprocesseurs

Utilité de la synchronisation

- Le problème de la programmation en // : les conflits d'accès en lecture/écriture (*race conditions*)
- Pour éviter un résultat indéterminé lorsque des fils doivent coopérer, il faut pouvoir les synchroniser. Exemple :
 - si (fil-1) alors A = 0 ; A = A + 1 ; sortir A ; fsi ⇒ 1
 - si (fil-2) alors B = A ; B = B + 1 ; sortir B ; fsi ⇒ 1 ou 2 ou ???

Modèle de programmation –2

Blocs et fils de calcul

- Un *fil de calcul* (*thread*) est une instance du kernel exécutée sur un cœur
- Un *bloc* est un ensemble de fils exécutés :
 - ▶ **en //, donc a priori "en même temps"**
 - ▶ **sur un même multiprocesseur**
- Attention, on peut exécuter jusqu'à 1024 fils dans un bloc (seulement 32 cœurs !) — donc pas vraiment "en même temps" (voir suite)
- Les blocs sont ordonnancés en // sur les multiprocesseurs

Utilité de la synchronisation

- Le problème de la programmation en // : les conflits d'accès en lecture/écriture (*race conditions*)
- Pour éviter un résultat indéterminé lorsque des fils doivent coopérer, il faut pouvoir les synchroniser. Exemple :

si (fil-1) alors	A = 0 ;		A = A + 1 ;	sortir A ; fsi	⇒ 1
si (fil-2) alors			B = B + 1 ;	sortir B ; fsi	⇒ 1
			B = A ;		

Synchronisation entre fils et blocs

- Synchronisation fondamentale : la *barrière* \Rightarrow tous les fils s'arrêtent à la barrière, quand tous sont arrivés, on repart (ex : départ du tiercé)
- Les fils d'un même bloc sont ordonnancés par paquets (*warps*) de 32 fils :
 - ▶ **à l'intérieur d'un warp** : l'exécution est toujours complètement synchrone à l'instruction près
 - ▶ **entre fils dans des warps différents d'un bloc (cas général)** : on peut toujours forcer la synchronisation explicitement en plaçant une barrière
- **Entre fils de blocs différents** : pas de synchronisation, sauf à la terminaison du kernel \Rightarrow chaîner plusieurs kernels permet de les synchroniser

Modèle de programmation –4

Note –1

- Pourquoi ordonnancer en warps séquentiels ? mauvais matériel ?
- \Rightarrow l'ordonnement en warps permet de masquer les délais d'accès mémoire : technique issue des recherches en //isme des années 1990

Note –2

- Différentes architectures vont autoriser différentes possibilités de synchronisation. Actuellement (2011) Nvidia = beaucoup de blocs avec peu de fils, ATI = peu de blocs avec beaucoup de fils

Hôte et carte

- Appli GPU :
 - ▶ code sur l'hôte (*host*) : C, C++, Python...
 - ▶ kernel sur une carte GPU : **CUDA**, OpenCL, OpenGL+GLSL

Rôle de l'hôte

- Charger un kernel sur une carte
- Allouer les données sur la carte
- Charger les données sur la carte
- Lancer l'exécution du kernel
- Récupérer les données de la carte, les désallouer
- Gérer les E/S

Présentation de CUDA

- CUDA est une variante de C, avec ajout de :
 - ▶ **Qualificatifs (*qualifiers*) de types de variables et fonctions**
 - ▶ **Pseudo-variables pour connaître "l'identité" d'un fil**
 - ▶ **Fonctions de synchronisation et fonctions d'accès atomique**
 - ▶ **Types vecteurs de 1 à 4 composantes (ex : float4)**
⇒ essentiellement pour l'imagerie
 - ▶ Fonctions mathématiques, de gestion de texture spécifiques GPU
 - ▶ Sur les cartes récentes (compute capability ≥ 2.0) : allocation et E/S avec limitations

Note :

- on ne traitera que des parties en gras

Programmation –3

Principe d'écriture d'un kernel en CUDA

- Les variables *threadIdx* et *blockIdx* contiennent le numéro du fil et de son bloc : elles sont automatiquement de valeur différentes dans chaque fil
 - En accédant à un élément de tableau indexé par ces variables, on fait travailler chaque fil sur une donnée différente :
 $a[\text{threadIdx}] = b[\text{threadIdx}];$
 - Un test comme *if (threadIdx == valeur)* permet aussi d'attribuer une tâche spécifique à un fil (ou groupe de fils : *if (threadIdx >= valeur)*)
-
- Concrètement, les fils sont indicés/engendrés selon 3 axes/dimensions
 - Les suffixes *.x*, *.y*, *.z* spécifient l'index de fil dans la dimension souhaitée : *threadIdx.x* renvoie l'indice dans la 1ère dimension du bloc de fils
 - Similairement, les blocs sont indicés/engendrés selon 2 dimensions
 - \Rightarrow voir exemple page suivante

Exemple produit terme à terme –1

Exemple 1 : produit terme à terme de 2 vecteurs

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule

# loading and compiling kernel code
# __global__ qualifies the kernel "main" function
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")
multiply_them = mod.get_function("multiply_them")
```

Exemple produit terme à terme –2

Exemple 1 (suite) : produit terme à terme de 2 vecteurs

```
# prepare 2 random input vectors and an empty destination
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)
dest = numpy.zeros_like(a)

# calling kernel (automatic allocation and copy of params)
multiply_them(drv.Out(dest), drv.In(a), drv.In(b),
              block=(400,1,1), grid=(1,1))

# verification (result should be a vector of zeroes)
print dest-a*b
```

- *block=(400,1,1)* spécifie le nombre/géométrie des fils par bloc (ici un vecteur linéaire de 400 fils)
- *grid=(1,1)* spécifie le nombre/géométrie des blocs pour l'exécution (ici un seul bloc)

Exemple : réduction –1

Réduction d'un tableau en mémoire partagée

- Idée : on veut sommer tous les termes d'un tableau
- Limitations : on suppose le tableau assez petit pour tenir dans la mémoire partagée d'un bloc, la taille du tableau est une puissance de 2
- Principe : approche "dichotomique" \Rightarrow on somme en parallèle un ensemble de paires de termes, puis on itère sur les paires de résultats

1	2	3	4	5	6	7	8
X	O	X	O	X	O	X	O

Exemple : réduction -2

Réduction d'un tableau en mémoire partagée

- Idée : on veut sommer tous les termes d'un tableau
- Limitations : on suppose le tableau assez petit pour tenir dans la mémoire partagée d'un bloc, la taille du tableau est une puissance de 2
- Principe : approche "dichotomique" \Rightarrow on somme en parallèle un ensemble de paires de termes, puis on itère sur les paires de résultats

1	2	3	4	5	6	7	8
x	o	x	o	x	o	x	o
3	2	7	4	11	6	15	8

Exemple : réduction –3

Réduction d'un tableau en mémoire partagée

- Idée : on veut sommer tous les termes d'un tableau
- Limitations : on suppose le tableau assez petit pour tenir dans la mémoire partagée d'un bloc, la taille du tableau est une puissance de 2
- Principe : approche "dichotomique" \Rightarrow on somme en parallèle un ensemble de paires de termes, puis on itère sur les paires de résultats

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

3	2	7	4	11	6	15	8
---	---	---	---	----	---	----	---

x	0	0	0	x	0	0	0
---	---	---	---	---	---	---	---

10	2	7	4	26	6	15	8
----	---	---	---	----	---	----	---

Exemple : réduction -4

Réduction d'un tableau en mémoire partagée

- Idée : on veut sommer tous les termes d'un tableau
- Limitations : on suppose le tableau assez petit pour tenir dans la mémoire partagée d'un bloc, la taille du tableau est une puissance de 2
- Principe : approche "dichotomique" \Rightarrow on somme en parallèle un ensemble de paires de termes, puis on itère sur les paires de résultats

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

3	2	7	4	11	6	15	8
---	---	---	---	----	---	----	---

10	2	7	4	26	6	15	8
----	---	---	---	----	---	----	---

x	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

36	2	7	4	26	6	15	8
----	---	---	---	----	---	----	---

Exemple : réduction –5

Exemple 2–a : kernel naïf de réduction d'un vecteur

```
// a non-main function, callable only from the device
__device__ void sumAllTerms(int *vect) {
    int tid = threadIdx.x;
    for (int offset=1; offset < MAX; offset*=2) {
        if (tid % (offset*2) == 0 && tid+offset < MAX) {
            vect[tid] += vect[tid+offset];
        }
        __syncthreads();
    }
}

__global__ main(...) {
    __shared__ int vect[MAX]; // vect in shared memory

    ... // fill vector
    sumAllTerms(vect); // reduce it
    ...
}
```

Exemple : réduction –6

Critique de la réduction naïve

- La divergence est importante : un fil sur deux d'un warp ne fait rien
- L'opérateur modulo est coûteux sur GPU
- Idée : "regrouper" tous les premiers termes des paires pour faire travailler tous les fils d'un warp

1	2	3	4	5	6	7	8
X	X	X	X	0	0	0	0

Exemple : réduction -7

Critique de la réduction naïve

- La divergence est importante : un fil sur deux d'un warp ne fait rien
- L'opérateur modulo est coûteux sur GPU
- Idée : "regrouper" tous les premiers termes des paires pour faire travailler tous les fils d'un warp

1	2	3	4	5	6	7	8
x	x	x	x	0	0	0	0
6	8	10	12	5	6	7	8

Exemple : réduction -8

Critique de la réduction naïve

- La divergence est importante : un fil sur deux d'un warp ne fait rien
- L'opérateur modulo est coûteux sur GPU
- Idée : "regrouper" tous les premiers termes des paires pour faire travailler tous les fils d'un warp

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

6	8	10	12	5	6	7	8
---	---	----	----	---	---	---	---

x	x	o	o	o	o	o	o
---	---	---	---	---	---	---	---

16	20	10	12	5	6	7	8
----	----	----	----	---	---	---	---

Exemple : réduction -9

Critique de la réduction naïve

- La divergence est importante : un fil sur deux d'un warp ne fait rien
- L'opérateur modulo est coûteux sur GPU
- Idée : "regrouper" tous les premiers termes des paires pour faire travailler tous les fils d'un warp

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

6	8	10	12	5	6	7	8
---	---	----	----	---	---	---	---

16	20	10	12	5	6	7	8
----	----	----	----	---	---	---	---

x 0 0 0 0 0 0 0

36	20	10	12	5	6	7	8
----	----	----	----	---	---	---	---

Exemple : réduction –10

Exemple 2–b : kernel de réduction d'un vecteur

```
__device__ void sumAllTerms(int *vect) {  
    int tid = threadIdx.x;  
  
    for (int offset=MAX/2; offset>0; offset>>=1) {  
        if (tid < offset) {  
            vect[tid] += vect[tid+offset];  
        }  
        __syncthreads();  
    }  
  
}
```

Exemple : réduction –11

- Il y a toujours des fils sans travail, mais il sont regroupés → moins de divergence !
- Encore optimisable (voir documents Nvidia) :
- ⇒ pas besoin de synchroniser si seulement un warp
- ⇒ dérouler/séquentialiser la boucle si peu d'itérations

Exemple : réduction -12

Exemple 2-c : réduction vecteur de taille quelconque

```
__device__ void sumAllTerms(int *vect) {
    int tid = threadIdx.x;

    for (int s=((MAX-1)/2)+1, b=MAX/2; b>0; b=s/2,s=((s-1)/2)+1)
    {
        if (tid < b) {
            vect[tid] += vect[tid+s];
        }
        __syncthreads();
    }
}
```

Exemple : réduction –13

Exemple 3 : reduction Max avec instructions atomiques

```
__device__ void max(int *vect) {  
    int tid = threadIdx.x;  
  
    // vect[0] is the supposed maximum  
    if (tid > 0 && tid < MAX)  
        atomicMax(&vect[0], vect[tid]);  
  
    __syncthreads();  
}
```

- Plus besoin de boucle !
- Pour les opérations faiblement conflictuelles en écriture ! (ex : petits tableaux, ...)
- Seulement sur les cartes récentes pour la "shared memory" (*compute capability* ≥ 2.0) , en mémoire globale depuis *compute capability* ≥ 1.2

Quelques règles empiriques

- Faire simple en premier lieu, optimiser ensuite
- Eviter au maximum la divergence au sein d'un même warp
- Charger les données en registre ou mémoire partagée quand c'est possible
- Synchroniser les boucles : notamment si une partie des instructions de la boucle peut être esquivée par un *break* ou *continue*, alors faire précéder par un *threadsync()*
- Ne pas essayer de synchroniser les blocs autrement que par sortie du kernel
- Réduire un vecteur de petite taille peut se traiter par des opérations atomiques (code beaucoup plus simple)

Installation

Installer CUDA en local

- attention, il faut les sources du noyau (sous linux)
- adresse du site CUDA :
<http://developer.nvidia.com/cuda-toolkit-32-downloads>
- instructions dans : `Linux_Getting_Started_Guide`
- installer dans l'ordre :
 - ▶ driver
 - ▶ toolkit
 - ▶ `gpucomputingSDK`
- il est prudent de compiler les exemples du SDK (il faudra probablement installer des bibliothèques C supplémentaires)

Conclusions

En résumé

- Une simple introduction : il reste des choses à dire... (accès coalescents à la mémoire, ...)
- Python + PyCUDA : alternative intéressante à C, au moins pour le prototypage rapide

Références

- <http://gpgpu.org/>
- <http://developer.nvidia.com/>
- http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf
- <http://documen.tician.de/pycuda/>
- <http://mathema.tician.de/software/pycuda>