Introduction à la Programmation CUDA

Oguz Kaya

Maître de Conférences
Université Paris-Saclay et l'Équipe ParSys du LRI, Orsay, France

université
PARIS-SACLAY

Objectifs

- Faire connaissance avec l'architecture d'un GPU (vs. CPU)
- Compréhender le modèle d'exécution d'un programme CUDA
- Utiliser un multiple de blocs et threads dans un kernel CUDA
- Apprendre le syntaxe de base pour un programme CUDA
- Maîtriser l'allocation et le transfert de données entre CPU et GPU
- Élaborer ces concepts sur un exemple (multiplication d'un tableau)

Outline

1. Architecture GPU vs CPU
2. Exécution d'un programme CUDA
3. Syntaxe CUDA
4. Allocation mémoire et transfert de données
5. Exemple

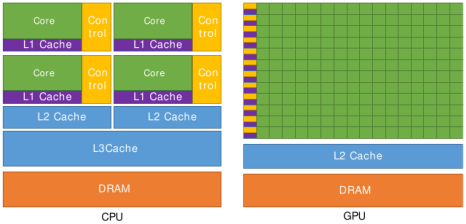
université PARIS-SACLAY

Outline

1. Architecture GPU vs CPU
2. Exécution d'un programme CUDA
3. Syntaxe CUDA
4. Allocation mémoire et transfert de données
5. Exemple

Architecture CPU vs GPU

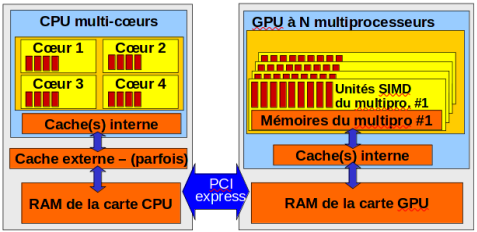
Un comparatif de l'architecture CPU et GPU



- Cache L1 potentiellement utilisable explicitement (shared memory)
- Cache L2 existe
- Pas de cache L3
- Peu de circuit de contrôle complexe, notamment
  - Exécution out-of-order
  - Branch prediction
  - Instruction level parallelism (ILP)
  - Complex instruction decoder
- 10x (ou plus) plus de puissance de calcul pour une même zone de circuit

Vue d'ensemble CPU-GPU

Une vue d'ensemble d'un CPU avec un GPU

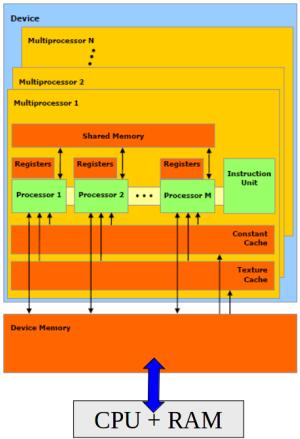


- Le CPU utilise le GPU comme un coprocesseur scientifique pour certains calculs adaptés au paradigme SIMD.
- Le CPU et le GPU sont tous les deux **multi-cœurs** et **vectorielles** avec une hiérarchie de mémoire particulière.
- Le transfert de donnés se fait sur le bus PCI express (32Go/s débit chaque direction pour PCIe4).
- Ils n'ont pas accès directe (!) à la RAM de l'un l'autre.

université PARIS-SACLAY

Zoom sur l'architecture GPU

Un GPU est un ensemble de N processeurs SIMD indépendants partageant une mémoire globale



- N streaming "multiprocesseurs" (SM)
- Chaque SM est un processeur SIMD ayant
  - k processeurs synchronisées ($k = 32$), autrement dit, **cœurs GPU**
  - 1 décodeur d'instructions partagé
  - 3 types de mémoires partagées entres toutes les k processeurs.
  - $32k - 128K$ registres distribués entres les processeurs (63-255 propre à chaque **thread**)
  - Pour pouvoir bien exploiter chaque SM, il faut lancer **au moins 32 threads (par bloc)**

université PARIS-SACLAY

Quelques chiffres pour l'architecture GPU

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS[1]	5	6.8	10.6	15.7
Peak FP64 TFLOPS[1]	1.7	.21	5.3	7.8
Peak Tensor TFLOPS[1]	NA	NA	NA	125

Certaines capacités de calculs n'évoluent pas de manière monotone

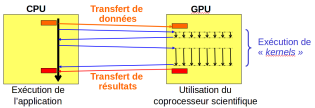
université PARIS-SACLAY

Outline

1. Architecture GPU vs CPU
2. Exécution d'un programme CUDA
3. Syntaxe CUDA
4. Allocation mémoire et transfert de données
5. Exemple

Principe d'exécution

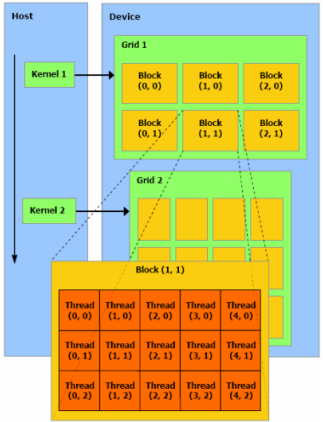
Le programe s'exécute principalement sur le CPU avec des appels de fonctions GPU.



- Avant/après de lancer un kernel, il faut transférer les données
- Il faut minimiser les transferts de données pour l'efficacité
- Chaque appel de kernel est non-bloquant (i.e., CPU continue l'exécution), mais on peut le rendre bloqant si on veut

Exécution d'une grille de blocs de threads

Le CPU lance l'exécution d'un kernel avec un ensemble de threads GPU.

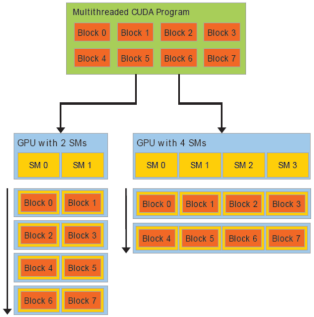


- Threads idéntiques (exécutent le même code)
- **Threads** organisés en **blocs** (de taille 32-1024)
- Chaque bloc **idéntique** s'exécute sur un SM.
- **Blocs** organisés en **grilles** et répartis sur tous les SMs
- Il faut lancer **suffisamment** de blocs et threads pour que **tout le domaine d'itération du problème** est parcouru.

université PARIS-SACLAY

Exécution d'une grille de blocs de threads (cont.)

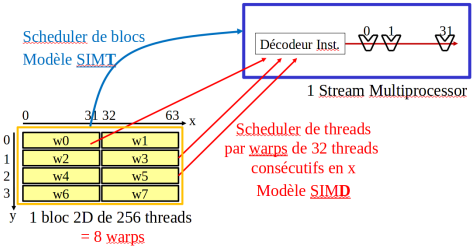
Le CPU lance l'exécution d'un kernel avec un ensemble de threads GPU.



- L'ordonnanceur de blocs répartit les blocs sur les différents SMs avec un ordonnancement dynamique.
- GPUs ayant différentes architectures pourront sans problème exécuter la même grille de blocs de threads (avec une répartition propre à son architecture, gérée par l'ordonnanceur).

Granularité de la grille et des blocs

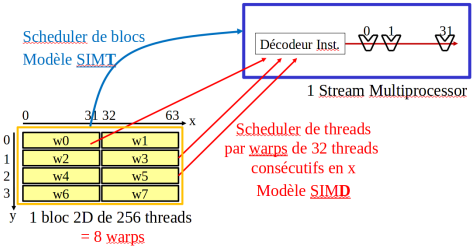
Créer des blocs un nombre entier de **warps** Le CPU lance l'exécution d'un kernel avec un ensemble de threads GPU.



- Un décodeur d'instruction pilote **32 threads hardware** (32 cœurs GPU)
- Chaque groupe de 32 threads consécutifs dans un bloc est appelé un **warp**
- L'ordonnanceur exécute chaque **warp** d'un bloc **actif** dans un SM

Granularité de la grille et des blocs (cont.)

Masquage du temps d'accès mémoires des warps



- Le GPU passe d'un warp à un autre très rapidement (car ils **coexistent** physiquemment dans le SM)
- Le GPU masque la latence des accès mémoire par multi-threading.
- Donc ne pas hésiter à créer un **grand nombre de petits threads** GPU par bloc et **un grand nombre de blocs** (c'est à dire, peu de travaille par chaque thread "léger").

université PARIS-SACLAY

Le choix du nombre de blocs et threads

Combien de blocs/grille et threads/bloc?

- L'ordonnanceur de **threads** souhaite avoir **beaucoup de warps de threads** en réserve pour recouvrir le temps d'accès à la mémoire
- L'ordonnanceur de **blocs** souhaite avoir **beaucoup de blocs pas trop gros** pour
  - avoir des blocs en réserve pour utiliser tous les SMs
  - recouvrement des temps d'accès mémoire entre blocs (un SM **peut** accueillir plusieurs blocs selon la disponibilité de ressources (registre, shared memory, etc.)
- En général, **128-256 threads/bloc** marche bien (min=1, max=1024).
- Choix par expérimentation ou un outil du Nvidia

Exécution d'un kernel avec un certain nombre de threads et blocs

```
int threadsParBloc = 256;
int numBlocs = N / threadsParBloc;
kernelGPU<<<numBlocs, threadsParBloc>>>(arg1, arg2, ...)
```

Outline

1. Architecture GPU vs CPU
2. Exécution d'un programme CUDA
3. Syntaxe CUDA
4. Allocation mémoire et transfert de données
5. Exemple

"Qualifiers" de CUDA

Un qualifier est un mot clé qui différentie les fonctions et les varibles CPU/GPU dans un programme CUDA.

Fonctionnement des « qualifiers » de CUDA :

	`__device__`	`__host__` (default)	`__global__`
Fonctions	Appel sur GPU Exec sur GPU	Appel sur CPU Exec sur CPU	Appel sur CPU Exec sur GPU
	`__device__`	**`__constant__`**	**`__shared__`**
Variables	Mémoire globale GPU	Mémoire constante GPU	Mémoire partagée d'un multiprocesseur
	Durée de vie de l'application	Durée de vie de l'application	Durée de vie du *block de threads*
	Accessible par les codes GPU et CPU	Ecrit par code CPU, lu par code GPU	Accessible par le code GPU, sert à *cacher* la mémoire globale GPU

Outline

1. Architecture GPU vs CPU
2. Exécution d'un programme CUDA
3. Syntaxe CUDA
4. Allocation mémoire et transfert de données
5. Exemple

Allocation d'un tableau sur GPU

```
#define N 1024

// Tableau statique sur le CPU
float TabCPU[N];

// Tableau statique global sur le GPU
__device__ float TabGPU[N];

// Tableau dynamique sur le CPU
float *TabCPU = (float *) malloc(N * sizeof(float));

// Tableau dynamique sur le GPU
float *TabGPU;
cudaError_t cuStat;
cuStat = cudaMalloc((void **) &TabGPU, N * sizeof(float));
```

- Le préfixe **__device__** différentie la déclaration d'un tableau statique GPU et CPU.
- Le tableau statique GPU doit être déclaré en dehors des fonctions (comme variables globales)
- Tableau dynamique sur GPU est alloué à l'aide de la fonction **cudaMalloc**.

Copier un tableau statique entre CPU et GPU

```
#define N 1024

// Tableau statique sur le CPU
float TabCPU[N];

// Tableau statique global sur le GPU
__device__ float TabGPU[N];

cudaError_t cuStat;

// Copier un tableau CPU dans un tableau statique GPU
custat = cudaMemcpyToSymbol(TabGPU, TabCPU,
    sizeof(float) * N, 0, cudaMemcpyHostToDevice);

// Copier un tableau statique GPU dans un tableau CPU
custat = cudaMemcpyFromSymbol(TabCPU, TabGPU,
    sizeof(float) * N, 0, cudaMemcpyDeviceToHost);
```

Copier un tableau dynamique entre CPU et GPU

```
// Transfert d'un tableau dynamique entre CPU et GPU
float *TabCPU;
float *TabGPU;
TabCPU = (float *) malloc (N * sizeof(float));
cudaError_t cuStat;
cuStat = cudaMalloc((void **) &TabGPU, N * sizeof(float));

// Copier un tableau CPU dynamique dans un tableau GPU dynamique
cudaStat = cudaMemcpy(TabGPU, TabCPU, sizeof(float)*N,
    cudaMemcpyHostToDevice);

// Copier un tableau GPU dynamique dans un tableau CPU dynamique
cudaStat = cudaMemcpy(TabCPU, TabGPU, sizeof(float)*N,
    cudaMemcpyDeviceToHost);
```

Outline

1. Architecture GPU vs CPU
2. Exécution d'un programme CUDA
3. Syntaxe CUDA
4. Allocation mémoire et transfert de données
5. Exemple

Multiplier un tableau par blocs en CUDA

Multiplier chaque élément d'un tableau **A[N]** par un scalaire **c**.

```
#include <cstdio>
#include "cuda.h"

#define N 1024
float A[N];
float c = 2.0;

__device__ float dA[N];

__global__ void multiplyArray(int n, float c)
{
  int elemParBlock = n / gridDim.x;
  int begin = blockIdx.x * elemParBlock;
  int end;
  if (blockIdx.x < gridDim.x - 1) {
    end = (blockIdx.x + 1) * elemParBlock;
  } else {
    end = n;
  }
  for (int i = begin; i < end; i++) { dA[i] *= c; }
}

int main(int argc, char **argv)
{
  // Initialisation
  for (int i = 0; i < N; i++) { A[i] = i; }
  // Copier le tableau vers le GPU
  cudaMemcpyToSymbol(dA, A, N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  multiplyArray<<<4, 1>>>(N, c);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%lf\n", A[2]);
  return 0;
}
```

- **__device__** defini le tableau sur le GPU.
- **__global__** defini la fonction sur le GPU.
  - Ce qui permet d'utiliser **blockIdx.x** et **gridDim.x** par exemple
- On doit copier les données dans le GPU avant et après le calcul avec **cudaMemcpy...** (à venir).
- Chaque bloc exécute toujours le même code.
- L'exécution est différenciée par le **blockIdx.x**.
- Avec P blocs, chaque bloc parcours N/P éléments consécutifs du tableau **A[N]**.
- Attention au dernier bloc si P ne divise pas N.

Multiplier un tableau par blocs en CUDA (amélioré)

Multiplier chaque élément d'un tableau **A[N]** par un scalaire **c**.

```
#include <cstdio>
#include "cuda.h"

#define N 1024
float A[N];
float c = 2.0;

__device__ float dA[N];

__global__ void multiplyArray(int n, float c)
{
  int i = blockIdx.x;
  dA[i] *= c;
}

int main(int argc, char **argv)
{
  // Initialisation
  for (int i = 0; i < N; i++) { A[i] = i; }
  // Copier le tableau vers le GPU
  cudaMemcpyToSymbol(dA, A, N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  multiplyArray<<<N, 1>>>(n, c);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%lf\n", A[2]);
  return 0;
}
```

- **__global__** defini la fonction sur le GPU.
  - Ce qui permet d'utiliser **blockIdx.x**
- Chaque bloc exécute toujours le même code.
- Chaque bloc effectue 1 opération, donc il faut lancer N blocs pour couvrir tout le tableau/domaine du calcul.
- L'exécution est différenciée par le **blockIdx.x**.

Multiplier un tableau par blocs et threads en CUDA

Multiplier chaque élément d'un tableau **A[N]** par un scalaire **c**.

```
#include <cstdio>
#include "cuda.h"

#define N 1024
float A[N];
float c = 2.0;

__device__ float dA[N];

__global__ void multiplyArray(int n, float c)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < n)
  dA[i] *= c;
}

int main(int argc, char **argv)
{
  // Initialisation
  for (int i = 0; i < N; i++) { A[i] = i; }
  // Copier le tableau vers le GPU
  cudaMemcpyToSymbol(dA, A, N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  int blockSize = 128;
  int numBlocks = N / blockSize;
  if (N % blockSize) numBlocks++;
  multiplyArray<<<numBlocks, blockSize>>>(n, c);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%lf\n", A[2]);
  return 0;
}
```

- **__global__** defini la fonction sur le GPU.
  - Ce qui permet d'utiliser **blockIdx.x**, **blockDim.x** et **threadIdx.x**
- Chaque thread et bloc exécute toujours le même code.
- On utilise **blockSize** threads par bloc.
- Chaque thread effectue 1 opération, donc il faut lancer **N / blockSize** blocs pour couvrir tout le tableau/domaine du calcul.
- L'exécution est différenciée par le **blockIdx.x** et **threadIdx.x**.
- Attention au dépassement du tableau (si N n'est pas divisible par blockSize)

université PARIS-SACLAY

Contact

Oguz Kaya
Université Paris-Saclay and LRI, Paris, France
oguz.kaya@lri.com
www.oguzkaya.com