Introduction à la Programmation GPU

Oguz Kaya

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

université
PARIS-SACLAY

Outline

université PARIS-SACLAY

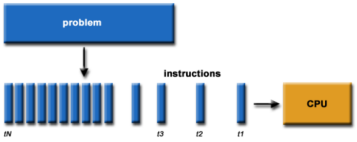
Outline

Objectifs

- Faire connaissance avec le calcul parallèle.
- Decouvrir les applications qui ont besoin de la puissance de calcul.
- Explorer l'architecture moderne d'un ordinateur parallèle.
- Faire une introduction rapide à la programmation GPU avec exemples.
- Apprendre comment compiler, exécuter, lancer un programme GPU.

Programmation séquentielle

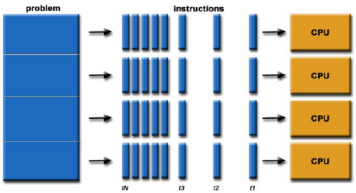
Traditionnellement, les logiciels sont basés sur le calcul **séquentiel**:



- Un problème est découpé en instructions.
- Ces instructions sont exécutées **séquentiellement** les unes après les autres.
- Elles sont exécutées par **un seul processeur**.
- À un instant donné, une seule instruction est exécutée.
- La performance est determinée principalement par **la frèquence** (Hz) du processeur.

Programmation parallèle

La **programmation parallèle** permet l'utilisation de plusieurs ressources de calcul pour résoudre un problème donné:

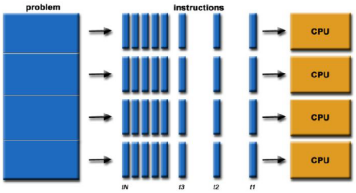


- Un problème est découpé en parties qui peuvent être lancées simultanément.
- Chaque partie est encore découpée en instructions.
- Les instructions de chaque partie sont exécutées **en parallèle** en utilisant plusieurs processeurs.
- La performance est determinée par:
  - La frèquence du processeur
  - Le nombre de processeurs
  - Le degré de parallelisation du problème

Outline

Applications du calcul parallèle

De nombreuses applications gourmandes en temps de calcul dans de diverses domaines:

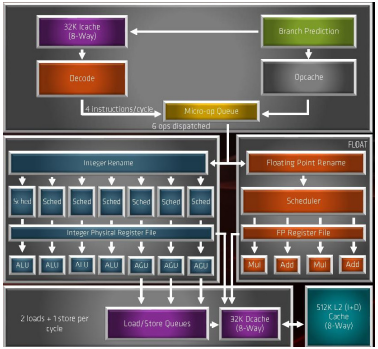


- Calcul scientifique: Simulations en physique, chimie, biologie, ...
- Traitement des reseaux neurones
- Graphiques (rendering, jeux vidéo, etc.)
- Systèmes d'exploitation (Linux, Android, etc.)
- et bien d'autres...

Outline

CPU

"Central Processing Unit", unité de calcul générale consistant à



L'architecture Zen 2

- Plusieurs unités d'exécutions (cœurs)
- Plusieurs niveaux de mémoire (registres, L1, L2, L3, RAM)
- Plusieurs ports d'exécution dans chaque cœur (ALUs, unités vectorielles)
- Unités vectorielles (AVX2, AVX512, Arm Neon, ...)
- Exécution de quelques threads (1-4) simultanément
- Capable d'exploiter le parallelisme au niveau des instructions (micro-op buffer, renumérotation d'instructions, renommer les registres, ...)
- Une partie considérable du circuit est consacrée au ILP + cache

GPU

"Graphical Processing Unit", unité de calcul spécifique véctorielle consistant à

L'architecture Nvidia Ampere

- Plusieurs unités d'exécutions (symmetric multiprocesseurs (SM))
- Plusieurs niveaux de mémoire (registres, mémoire partagée, L1, L2, RAM)
- Plusieurs unités vectorielles (2-4) larges (16-32 flottants) dans chaque SM
- Exécution de milliers de threads simultanément
- Grand tableau de registres (65K)
- Échange de threads très rapide
- La plupart du circuit est consacrée aux unités vectorielles
- Parallélisation prend l'effort

GPU (cont.)

L'architecture Nvidia Ampere

Supercalculateur / Cluster

Un ensemble de machines (CPU+GPU) connecté

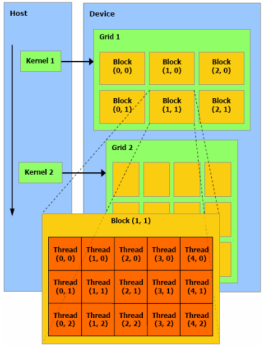
Jolio Curie supercalculateur,
300K cœurs CPU, 1024 GPUs

- Connexion par un réseau avec une topologie particulière (anneau, grille, torus, clique, etc.)
- Bibliothèques de communications adaptée à la topologie
- Capable de s'adresser à des problèmes de très grande taille
- Aujourd'hui, à l'échelle d'exaflops (10^{18} opérations flottants par seconde)

Outline

Programmation GPU

La programmation GPU est adaptée au modèle d'exécution single-instruction-multiple-thread (SIMT).



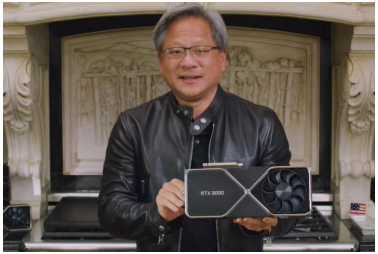
Exécution des kernels GPUs par blocs et threads

- Un GPU consiste à plusieurs processeurs identiques (comme CPU) appelés "streaming multiprocessors" (SM).
- Chaque SM a plusieurs "cœurs", et chaque cœur peut exécuter un thread simultanement (à venir ...)
- Il y a un kernel (fonction) à exécuter par tous les SMs/threads.
- Un **bloc** est l'execution du kernel dans un SM.
- Le calcul dans un bloc se différentie des autres par un identifiant (blockIdx.x).

université PARIS-SACLAY

Programmation CUDA

CUDA est le langage de programmation, developpé par NVIDIA et basé sur C/C++, consiste à



Nvidia Ampere RTX 3090 GPU avec un Nvidia Fanboy

- Un compilateur (nvcc)
- Bibliothèque de base (cuda.h)
- De nombreux bibliothèques avec des noyeaux optimisés (cuBLAS, cuDNN, cuSOLVER, cuTENSOR, RAPIDS, ...)

Hello World en OpenMP

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

int main(int argc, char **argv)
{
#pragma omp parallel num_threads(3)
  {
    int thid = omp_get_thread_num();
    int numth = omp_get_num_threads();
    printf("Hello from thread %d/%d.\n", thid, numth);
  }
  return 0;
}
```

- **omp.h** est la bibliothèque OpenMP qui fournit les fonctions nécessaires (e.g., pour obtenir thid, numth).
- **#pragma omp parallel** crée 3 threads qui exécutent **le même code** de manière **asynchrone**.
- Chaque thread a un identifiant unique entre 0 et 2 (ou $P-1$ si on crée P threads)
- Sorties possibles ???
  - 3! possibilités car threads asynchrones

Hello World en CUDA

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

__global__ void helloWorld()
{
  printf("Hello from block %d/%d\n",
      blockIdx.x, gridDim.x);
}

int main(int argc, char **argv)
{
  helloWorld<<<3, 1>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

- **cuda.h** fournit les fonctions nécessaires.
- **__global__** précise la définition d'un kernel GPU (sinon fonction CPU par défaut)
  - Définit **blockIdx.x** et **gridDim.x**
- **<<< 3, 1 >>>** crée 3 blocs (chaqun ayant 1 seul thread, à venir) qui exécutent **le même code** de manière **asynchrone**.
- Chaque bloc a un identifiant unique entre 0 et 3 (ou $P-1$ si on lance le kernel avec P blocs)
- **blockIdx.x** est prédefini et donne l'identifiant d'un bloc dans un kernel GPU.
- **gridDim.x** est prédefini et donne le nombre de blocs utilisés dans le kernel GPU en cours d'exécution.
- Sorties possibles ???

Multiplier un tableau en OpenMP

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

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

#define N 1024

int main(int argc, char **argv)
{
  float A[N];
  float c = 2.0;
  // Initialisation
  for (int i = 0; i < N; i++) { A[i] = i; }
#pragma omp parallel num_threads(4)
  {
    for (int i = 0; i < N; i++) {
      A[i] *= c;
    }
  }
  return 0;
}
```

- Est-ce correct ce programme?
  - Non! Chaque élément est multiplié 4 fois!

Multiplier un tableau en OpenMP (cont.)

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

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

#define N 1024

int main(int argc, char **argv)
{
  float A[N];
  float c = 2.0;
  // Initialisation
  for (int i = 0; i < N; i++) { A[i] = i; }
#pragma omp parallel num_threads(4)
  {
    int thid = omp_get_thread_num();
    int numth = omp_get_num_threads();
    int elemParTh = N / numth;
    int begin = thid * elemParTh;
    int end;
    if (thid < numth - 1) { // Avant le dernier thread
      end = (thid + 1) * elemParTh;
    } else { // Le dernier thread
      end = N;
    }
    for (int i = begin; i < end; i++) {
      A[i] *= c;
    }
  }
  return 0;
}
```

- Chaque thread exécute toujours le même code.
- Cette fois-ci, l'exécution est différenciée par le **thid**.
- Avec P threads, chaque thread parcours N/P éléments consécutifs du tableau **A[N]**.
- Attention au dernier thread si P ne divise pas N.

Multiplier un tableau 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.

Outline

Compilation et exécution d'un programme CUDA

- Les fichiers de source doivent avoir l'extension **.cu** (e.g. programme.cu)
- **Compilation: nvcc programme.cu -o programme**
  - Possibilité de préciser l'architecture avec **-arch sm_xx** (e.g. -arch sm_75 pour Turing)
- **Exécution: ./programme**

References

Contact

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