Introduction to CUDA Programming

Oguz Kaya

Maˆıtre de Conf'erences
Universit'e Paris-Saclay and the LRI ParSys team, Orsay, France

Objectives

- Get to know the architecture of a GPU (vs. CPU) Understand the
- execution model of a CUDA program Use multiple blocks and
- threads in a CUDA kernel Learn the basic syntax for a CUDA
- program
- Maˆıtriser l'allocation et le transfert de donnèes entre CPU et GPU
- Élaborer ces concepts sur un exemple (multiplication d'un tableau)

Outline

1 GPU vs. CPU architecture

2 Running a CUDA program CUDA

3 syntax

4 Memory allocation and data transfer Example

5

Outline

1 GPU vs. CPU architecture Running

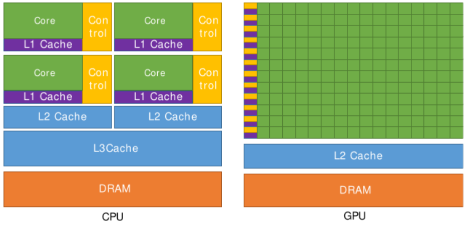
2 a CUDA program CUDA syntax

3 Memory allocation and data transfer Example

5

CPU vs GPU architecture

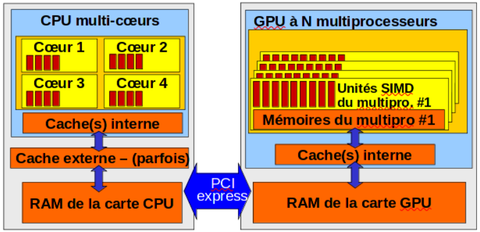
A comparison of the CPU and GPU architecture



- L1 cache potentially usable explicitly (shared memory)
- L2 cache exists
- No L3 cache
- Few complex control circuits, including
  - Ex'ecution out-of-order
  - Branch prediction
  - Instruction level parallelism (ILP)
  - Complex instruction decoder
- 10x (or more) more computing power for a mˆeme circuit area

CPU-GPU Overview

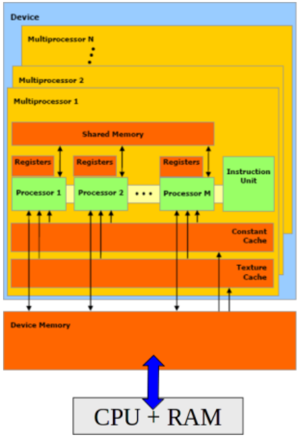
An overview of a CPU with a GPU



- The CPU uses the GPU as a scientific coprocessor for some calculations adapted to the SIMD paradigm.
- The CPU and GPU are both **multi-core** and **vector** with a particular memory hierarchy.
- The data transfer is done on the PCI express bus (32GB/s of ebit each direction for PCIe4).
- They do not have direct access (!) to each other's RAM.

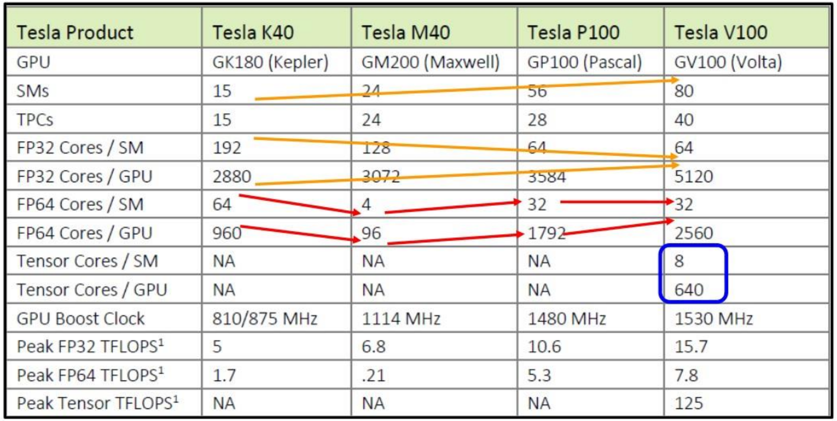
Focus on the GPU architecture

A GPU is a set of *N* independent SIMD processors sharing a global memory



- *N* multiprocessor streaming (SM)
- Each SM is a SIMD processor with *k*
  - synchronized processors ($k = 32$), in other words, **GPU cores**
  - 1 shared instruction ecoder
  - 3 types of memories shared between all *k* processors.
  - 32k - 128K distributed registers between the processors (63-255 for each **thread**) To be able to
  - exploit each SM, it is necessary to launch **at least 32 threads (per block)**

Some figures for the GPU architecture



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

Outline

1 GPU vs. CPU architecture Running

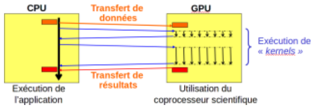
2 a CUDA program CUDA syntax

3 Memory allocation and data transfer Example

5

Principle d'execution

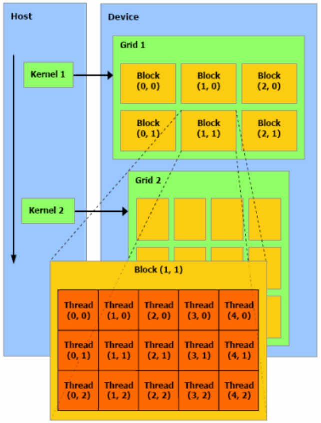
The program runs mainly on the CPU with GPU function calls.



- Before/after launching a kernel, it is necessary to transfer the data
- Minimize data transfers for efficiency
- Each kernel call is non-blocking (i.e., CPU continues execution), but it can be made blocking if you want

Execution of a grid of thread blocks

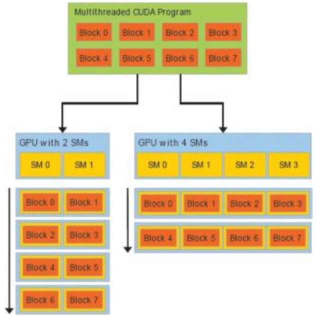
The CPU launches the execution of a kernel with a set of GPU threads.



- Threads idèntiques (run the same code) **Threads**
- organized in **blocks** (of size 32-1024) Each
- **idèntic** block runs on a SM.
- **Blocks** organized in **grids** and distributed on all the SMs
- It is necessary to launch **enough** blocks and threads so that **the whole iteration domain of the problem** is covered.

Execution of a grid of thread blocks (cont.)

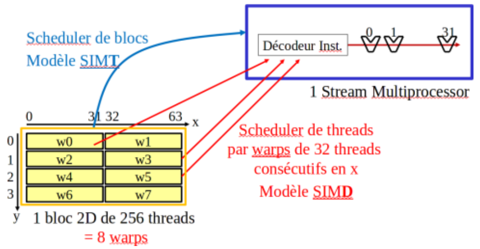
The CPU launches the execution of a kernel with a set of GPU threads.



- The block scheduler distributes the blocks to the different SMs with dynamic scheduling.
- GPUs with different architectures will be able to execute the same grid of thread blocks without any problem (with a distribution specific to its architecture, managed by the scheduler).

Granularity of the grid and blocks

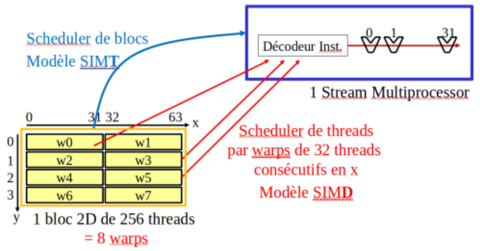
Create blocks with an integer number of **warps** The CPU starts the execution of a kernel with a set of GPU threads.



- One instruction encoder drives **32 hardware threads** (32 GPU cores)
- Each group of 32 consecutive threads in a block is called a **warp**
- The scheduler executes each **warp** of an **active** block in a SM

Granularity of the grid and blocks (cont.)

Masking of the memory access time of warps



- The GPU switches from one warp to another very quickly (because they physically **coexist** in the SM)
- The GPU masks the latency of memory accesses by multi-threading.
- So dont hesitate to create a **large number of small** GPU **threads** per block and **a large number of blocks** (ie, few works by each thread

"eger").

The choice of the number of blocks and threads

How many blocks/grid and threads/block?

- The **thread** scheduler wants to have **a lot of thread warps** in reserve to cover the memory access time
- The **block** scheduler wants to have **a lot of blocks that are not too big** to
  - have blocks in reserve to use all the SMs
  - recovery of memory access times between blocks (a SM **can** host several blocks depending on the availability of resources (register, shared memory, etc.)
- In general, **128-256 threads/block** works well (min=1, max=1024).
- Choice by experimentation or a tool from Nvidia

Running a kernel with a number of threads and blocks

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

Outline

1 GPU vs. CPU architecture Running

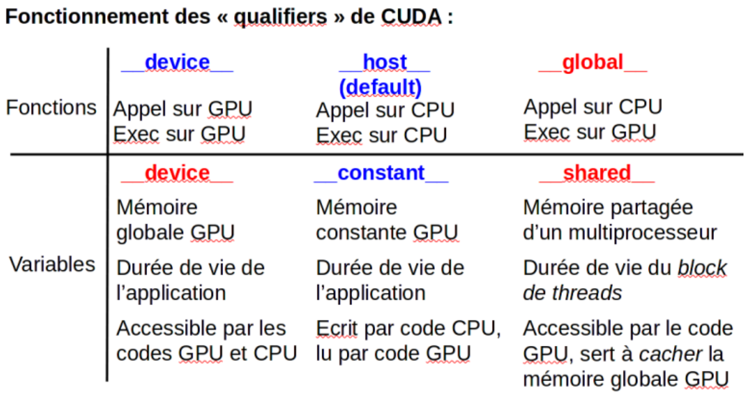
2 a CUDA program CUDA syntax

3 Memory allocation and data transfer Example

5

"CUDA Qualifiers

A qualifier is a `keyword` that differentiates the CPU/GPU functions and variables in a CUDA program.



Fonctionnement des « qualifiers » de CUDA :

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

Outline

1 GPU vs. CPU architecture Running

2 a CUDA program CUDA syntax

3 Memory allocation and data transfer Example

5

Allocation of an array on GPU

```
# define N 1024

// Static array on the CPU float TabCPU [
N ] ;

// Global static table on the GPU
--device-- float TabGPU [ N ] ;

// Dynamic table on the CPU
float * TabCPU = ( float * ) malloc (N * sizeof
( float ) ) ;

// Dynamic array on the GPU float
* TabGPU ;
cudaError_t cuStat ;
cuStat = cuda Malloc ( ( void * * ) &TabGPU , N * sizeof ( float ) );
```

- The prefix __device__ differs from the declaration of a static GPU and CPU array.
- The GPU static array must ˆetre d'eclaré outside the functions (as global variables)
- Dynamic array on GPU is allé using the function **cudaMalloc**.

Copy a static array between CPU and GPU

```
# define N 1024

// Static array on the CPU float TabCPU [
N ] ;

// Global static table on the GPU
 device float TabGPU [ N ] ;

cudaError_t cuStat ;

// Copy a CPU array into a GPU static array
cuStat = cudaMemcpyToSymbol (TabGPU , TabCPU ,
    sizeof ( float )  *   N, 0 , cudaMemcpyHostToDevice ) ;

// Copy a GPU static array into a CPU array
cuStat = cudaMemcpyFromSymbol ( TabCPU , TabGPU ,
    sizeof ( float )  *   N, 0 , cudaMemcpyDeviceToHost ) ;
```

Copy a dynamic array between CPU and GPU

```
// Transfer of a dynamic array between CPU and GPU float
*TabCPU ;
float *TabGPU ;
TabCPU = (float *)malloc(N * sizeof(float));
cudaError_t cuStat;
cuStat = cudaMalloc((void **)&TabGPU, N * sizeof(float));

// Copy a dynamic CPU array into a dynamic GPU array
cudaStat = cudaMemcpy(TabGPU, TabCPU, sizeof(float)*N,
     cudaMemcpyHostToDevice);

// Copy a dynamic GPU array into a dynamic CPU array
cudaStat = cudaMemcpy(TabCPU, TabGPU, sizeof(float)*N,
     cudaMemcpyDeviceToHost);
```

Outline

1 GPU vs. CPU architecture Running

2 a CUDA program CUDA syntax

3 Memory allocation and data transfer Example

5

Multiplying an array by blocks in CUDA

Multiply each element of an array **A[N]** by a scalar **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)
{
  // Initialization
  for (int i = 0; i < N; i++) { A[i] = i; }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * sizeof(float), 0,
       cudaMemcpyHostToDevice);
  multiplyArray<<<4, 1>>>(N, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * sizeof(float), 0,
       cudaMemcpyDeviceToHost);
  printf("%lf\n", A[2]);
  return 0;
}
```

- **device** defini the table on the GPU.
- function on the GPU.
  - This allows you to use **blockIdx.x** and **gridDim.x** for example
- We have to copy the data in the GPU before and after the calculation with **cudaMemcpy...** (to come).
- Each block always executes the mˆeme code.
- The execution is differentiated by the **blockIdx.x**.
- With *P* blocks, each block runs through *N/P* 'cons'ecutive elements of the array **A[N]**.
- Be careful with the last block if *P* does not divide *N*.

Multiplying an array by blocks in CUDA (amèliorè)

Multiply each element of an array **A[N]** by a scalar **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)
{
  // Initialization
  for (int i = 0; i < N; i++) { A[i] = i; }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  multiplyArray<<<N, 1>>>(n, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%lf\n", A[2]);
  return 0;
}
```

- function on the GPU.
  - This allows to use **blockIdx.x**
- Each block always executes the m^eme code.
- Each block performs 1 operation, so you have to run *N* blocks to cover the whole table/area of the calculation.
- The execution is differentiated by the **blockIdx.x**.

Multiplying an array by blocks and threads in CUDA

Multiply each element of an array **A[N]** by a scalar **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)
{
  // Initialization
  for (int i = 0; i < N; i++) { A[i] = i; }
  // Copy the table to the 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
  );
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%lf\n", A[2]);
  return 0;
}
```

- function on the GPU.
  - This allows to use **blockIdx.x**, **blockDim.x** and **threadIdx.x**
- Each thread and block always executes the mˆeme code.
- We use **blockSize** threads per block.
- Each thread performs 1 operation, so you have to launch **N / blockSize** blocks to cover the entire table/calculation area.
- The execution is differentiated by the **blockIdx.x** and **threadIdx.x**.
- Beware of overflowing the array (if *N* is not divisible by blockSize)

Contact

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