Introduction to GPU programming

Oguz Kaya

Assistant professor
Université Paris-Saclay, ParSys group at LISN, Gif-sur-Yvette, France

Objectifs

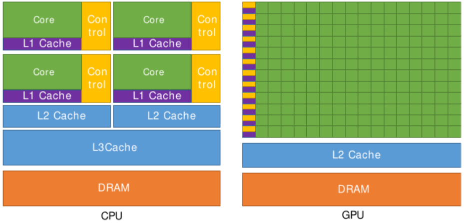
- Getting to know the architecture of a GPU (vs. CPU)
- Understanding the execution model of a CUDA program
- Using multiple blocks and threads in a CUDA kernel
- Learning the basic syntax for a CUDA program
- Mastering data allocation and transfer between CPU and GPU
- Elaborating these concepts with an example (array multiplication)

Outline

Outline

CPU vs GPU architecture

A comparison of CPU and GPU architecture

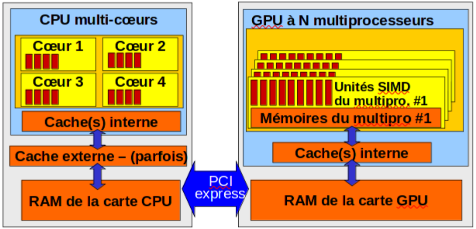


- L1 cache potentially usable explicitly (shared memory)
- L2 cache exists
- No L3 cache
- Few complex control circuits, notably
  - Out-of-order execution
  - Branch prediction
  - Instruction-level parallelism (ILP)
  - Complex instruction decoder
- 10x (or more) more computational power for the same chip area

universite PARIS-SACLAY

CPU-GPU Overview

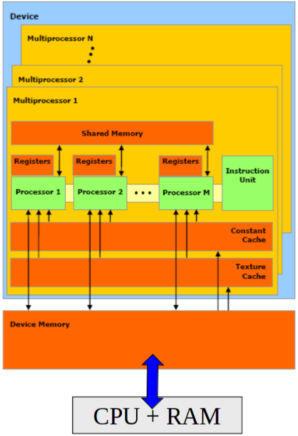
Overview of a CPU with a GPU



- The CPU uses the GPU as a scientific coprocessor for certain calculations suited to the SIMD paradigm.
- Both the CPU and GPU are **multi-core** and **vector processors** with a specific memory hierarchy.
- Data transfer occurs over the PCI Express bus (32 GB/s bandwidth in each direction for PCIe4).
- They do not have direct access to each other's RAM.

Zoom on GPU architecture

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



- N streaming "multiprocessors" (SM)
- Each SM is a SIMD processor having
  - k synchronized processors ($k = 32$), in other words, **GPU cores**
  - 1 shared instruction decoder
  - 3 types of memories shared among all k processors
  - $32k - 128K$ registers distributed among the processors (63-255 specific to each **thread**)
  - To fully utilize each SM, you need to launch **at least 32 threads (per block)**

Some numbers for the capacity of GPU architectures

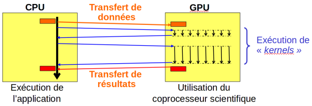
Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	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

Execution principle

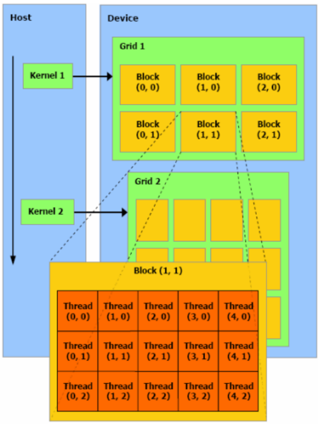
The program mainly runs on the CPU with calls to GPU functions.



- Before/after launching a kernel, data must be transferred
- Data transfers should be minimized for efficiency
- Each kernel call is non-blocking (i.e., the CPU continues execution), but it can be made blocking if desired

Execution of a grid of thread blocks

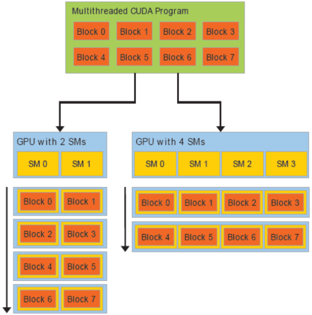
The CPU launches the execution of a kernel with a set of GPU threads.



- Identical threads (executing **the same code!**)
- **Threads** organized into **blocks** (of size 32-1024)
- Each **identical** block runs on an SM
- **Blocks** organized into **grids** and distributed across all SMs
- You need to launch **sufficient** number of blocks and threads so that **the entire iteration domain of the problem** is covered
- **Threads** within the **same block** can **share resources** and **communicate/synchronize**.
  - **Threads** in **different blocks cannot!**

Execution of a grid of thread blocks (cont.)

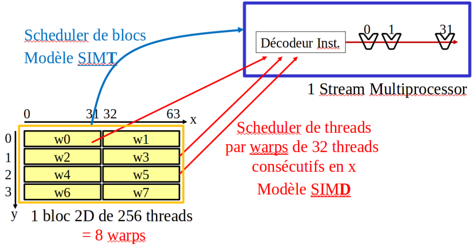
The CPU launches the execution of a kernel with a set of GPU threads.



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

Grid and block granularity

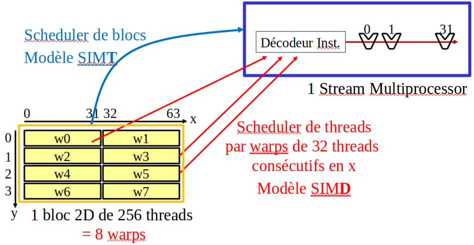
Create blocks with a certain number of **warps** The CPU launches the execution of a kernel with a set of GPU threads.



- An instruction decoder drives **32 threads hardware** (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 an SM

Grid and block granularity (cont.)

Masking of memory access time of warps



- The GPU switches from one warp to another very quickly (because they **coexist** physically in the SM)
- The GPU masks memory access latency through multi-threading.
- So do not hesitate to create a **large number of small GPU threads** per block and a **large number of blocks** (that is, little work per each "light" thread).

The choice of the number of blocks and threads

How many blocks/grid and threads/block should I use?

- The **thread** scheduler wants to have **many warps of threads** in reserve to hide the memory access latency
- The **block** scheduler wants to have **many not-too-large blocks** in order to
  - have blocks in reserve to use all SMs
  - overlap memory access times between blocks (an SM **can**
  - host multiple blocks depending on resource availability (registers, shared memory, etc.) for better SM utilization
- In general, **128-256 threads/block** works well (min=1, max=1024).
- Optimal choice by experimentation or through an Nvidia tool

universite PARIS-SACLAY

Execution of a kernel with a certain number of threads and blocks

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

Outline

CUDA Qualifiers

A qualifier is a keyword that differentiates 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

Allocation of an array on GPU

```
#define N 1024

// Static global array on the CPU
float ArrCPU[N];

// Static global array on the GPU
__device__ float ArrGPU[N];

// Dynamic array on the CPU
float *ArrCPU = (float *) malloc(N * sizeof(float));

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

- The prefix **__device__** differentiates the declaration of a static GPU array from a CPU array.
- The static GPU array must be declared outside of functions (as global variables)
- Dynamic arrays on GPU are allocated using the **cudaMalloc** function.

université PARIS-SACLAY

Copying a static array between a CPU and GPU

```
#define N 1024

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

// Static array on the GPU
__device__ float ArrGPU[N];

cudaError_t cuStat;

// Copy a static CPU array onto a static GPU array
custat = cudaMemcpyToSymbol(ArrGPU, ArrCPU,
    sizeof(float) * N, 0, cudaMemcpyHostToDevice);

// Copy a static GPU array onto a static CPU array
custat = cudaMemcpyFromSymbol(ArrCPU, ArrGPU,
    sizeof(float) * N, 0, cudaMemcpyDeviceToHost);
```

Copying a dynamic array between a CPU and GPU

```
// Copying a dynamic array between CPU and GPU
float *ArrCPU;
float *ArrGPU;
ArrCPU = (float *) malloc (N * sizeof(float));
cudaError_t cuStat;
cuStat = cudaMalloc((void **) &ArrGPU, N * sizeof(float));

// Copy a dynamic CPU array onto a dynamic GPU array
cudaStat = cudaMemcpy(ArrGPU, ArrCPU, sizeof(float)*N,
    cudaMemcpyHostToDevice);

// Copy a dynamic GPU array onto a dynamic CPU array
cudaStat = cudaMemcpy(ArrCPU, ArrGPU, sizeof(float)*N,
    cudaMemcpyDeviceToHost);
```

Outline

Multiply 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)
{
  // 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__** defines the array on the GPU.
- **__global__** defines the function on the GPU.
  - This allows using **blockIdx.x** and **gridDim.x**, for example
- We must copy the data to the GPU before and after computation with **cudaMemcpy...** (coming up).
- Each block always executes the same code.
- Execution is differentiated by **blockIdx.x**.
- With P blocks, each block processes N/P consecutive elements of the array **A[N]**.
- Be careful with the last block if P does not divide N.

universite PARIS-SACLAY

Multiply an array by blocks in CUDA (improved)

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)
{
  // 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__** defines the function on the GPU.
  - This allows using **blockIdx.x**
- Each block always executes the same code.
- Each block performs 1 operation, so N blocks must be launched to cover the entire array/computation domain.
- Execution is differentiated by **blockIdx.x**.

Multiply 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)
{
  // 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__** defines the function on the GPU.
  - This allows using **blockIdx.x**, **blockDim.x**, and **threadIdx.x**
- Each thread and block always executes the same code.
- We use **blockSize** threads per block.
- Each thread performs 1 operation, so **N / blockSize** blocks must be launched to cover the entire array/computation domain.
- Execution is differentiated by **blockIdx.x** and **threadIdx.x**.
- Be careful of array overflow (if *N* is not divisible by blockSize)

Contact

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