Introduction to GPU programming

Oguz Kaya

Assistant Professor
Université Paris-Saclay and LISN, Gif-sur-Yvette, France

Outline

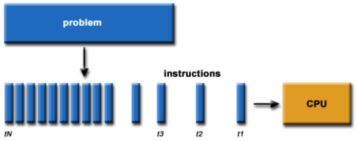
Outline

Objectives

- Some reminders on parallel computing, applicatins, and architecture.
- Rapid overview to GPU architecture and programming
- Compilation, execution, and launching mechanisms of GPU programs

Sequential program

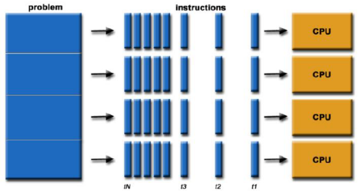
Traditionally, algorithms and software are based on a **sequential execution**:



- A problem is split into instructions.
- These instructions are executed **sequentially** one after another.
- They are executed by **only one processor**
- At a given time step, at most one instruction is executed.
- The performance is determined mostly by the **frequency** (Hz) of the processor.

Parallel programming

Parallel programming enables the use of multiple processing units to solve a given problem

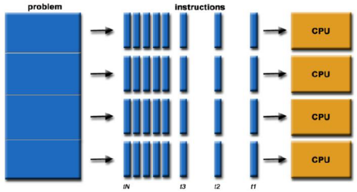


- A problem is split into independent parts that can treated simultaneously.
- Each part is split into instructions.
- Instructions of different parts are executed **in parallel** by using multiple processors.
- The performance is determined by the
  - processor frequency
  - number of of processors
  - degree of parallelization of the problem.

Outline

Applications of parallel computing

Numerous applications requiring a lot of compute power in diverse domains:

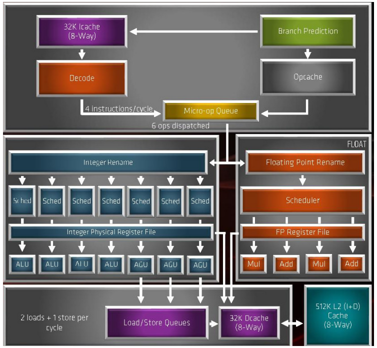


- Scientific computing: Simulations for physics, chemistry, biology, climate, quantum computing, etc.
- Neural network training and inference
- Computer graphics: Rendering, video games, animations, 3D modeling, etc.
- Operating systems: Linux, Windows, Android, etc.
- and many others

Outline

CPU

"Central Processing Unit", or CPU, generally consists of

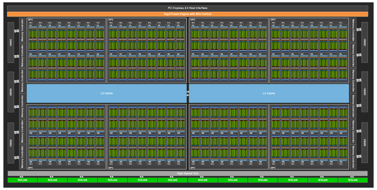


L'architecture Zen 2

- Multiple processing units (cores)
- Multiple memory levels (registers, L1, L2, L3, RAM)
- Multiple execution ports in each core (ALUs, vector units)
- Vector units for SIMD parallelism (AVX2, AVX512, Arm Neon, ...)
- Simultenous execution of multiple threads (2-4)
- Capable of exploiting the instruction level parallelism (ILP) with micro-op buffer, instruction reordering, register renaming, etc.
- A good portion of hardware is dedicated to managing ILP + cache

GPU

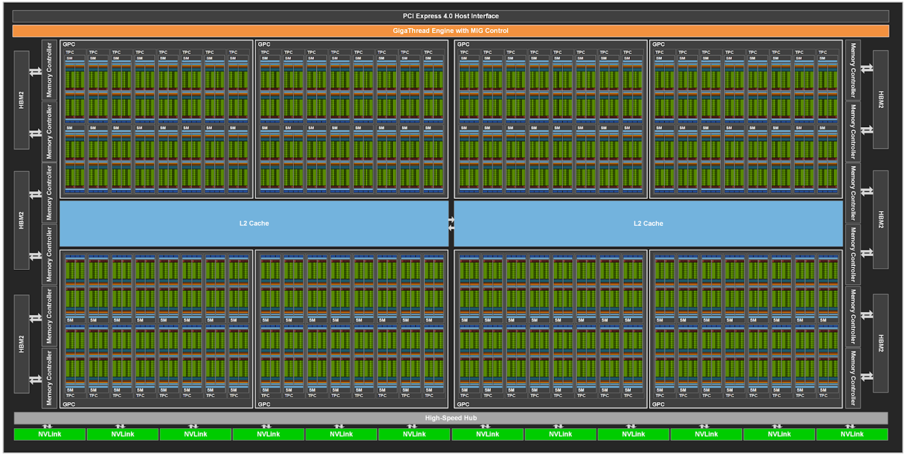
“Graphical Processing Unit”, or GPU, is a vector processing unit consisting of



L’architecture Nvidia Ampere

- Multiple execution units (streaming multiprocessors - SMs)
- Multiple memory levels (registers, shared memory, L1, L2, RAM)
- Multiple vector units (1-8) that are wide (16-32 floats) in each SM.
- Simultaneous execution of **thousands** of threads
- Giant register table (65K)
- Very fast context switching among threads, since threads reside in SMs simultenously
- Most of the hardware is dedicated to vector compute units, very high performace potential.
- Parallelization requires some effort

GPU (cont.)


PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

Nvidia Ampere architecture

Supercomputer / cluster

An ensamble of connected machines (CPU+GPU) over a network

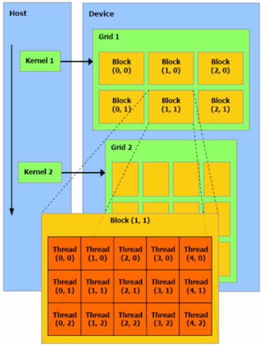
Jolio Curie supercomputer ,
300K CPU cores, 1024 GPUs

- Connection with a network having a topology (ring, grid, torus, clique, etc.)
- Optimized communication libraries adapted to the topology
- Capable of adressing very large-scale problems
- Today, we already exceeded the exaflop scale (10^{18} floating point operations per second, or flops/s)

Outline

GPU programming

GPU programming is adapted to the single-instruction-multiple-thread (SIMT) model.

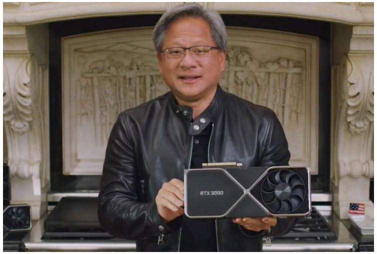


GPU kernel execution by blocks of threads

- Each GPU consists of multiple identical processors (like CPU cores) called **streaming multiprocessors** (SM).
- Each SM has multiple smaller cores, and each core can execute a thread simultenously (to come).
- There is a kernel (GPU function) to execute by all threads on SM cores.
- A **block** is the execution of a kernel on an SM.
- Computation on a block differentiates itself from others by an identifier - **blockIdx.x**

CUDA programming

CUDA is the programming language, developed by Nvidia and based on C/C++, consists of



Nvidia Ampere RTX 3090 GPU

- A compiler (nvcc)
- A base library (cuda.h)
- And many other libraries with optimized kernels (cuBLAS, cuDNN, cuSOLVER, cuTENSOR, RAPIDS, etc.)

Hello World in 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** is the OpenMP library header that provides necessary functions (e.g., to obtain thid, numth, etc.)
- **#pragma omp parallel** creates 3 threads that execute **the same code** and **simultaneously** and **asynchronously**
- Each thread has a unique identifier between 0 and 2 (or $P-1$ if we create P threads)
- Possible outputs?
  - 3! possibilities due to asynchronous execution

Hello World in 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** provides necessary functions.
- **__global__** specifies the definition of a GPU kernel (otherwise it is a CPU function by default)
  - Defines **blockIdx.x** and **gridDim.x**
- $<<<\mathbf{3,1}>>>$ creates 3 blocks (each having 1 thread, more to come) that execute **the same code asynchronously**.
- Each block has an **identifier** between 0 et 3 (or $P-1$ if we launch the kernel with P blocks)
- **blockIdx.x** is predefined and provides the block identifier in a GPU kernel
- **gridDim.x** is predefined and provides the number of blocks used in a GPU kernel.
- Possible outputs?

Multiply and array in OpenMP

Multiply each element of an array **A[N]** by a scalar **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;
}
```

- Is this program correct?
  - No! Each element is multiplied 4 times!

Multiply and array in OpenMP (cont.)

Multiply each element of an array **A[N]** by a scalar **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;
}
```

- Each thread still executes the same code
- This time, the execution is differentiated by **thid**
- Using P threads, each thread iterates over N/P consecutive elements of **A[N]**
- Be careful for the last thread if P is not divisible by N.

Multiply an array 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 an array on the GPU
- **__global__** defines a function for the GPU.
  - Which enables using **blockIdx.x** and **gridDim.x** for example
- Without these keywords, it is for the CPU by default.
- We must copy data in the GPU memory before and after the computation with **cudaMemcpy...** (to come).
- Each block still executes the same code.
- Execution is differentiated by **blockIdx.x**
- With P blocks, each block iterates over N/P consecutive elements of **A[N]**.
- Careful for the last block if N is not divisible by P!

Outline

Compilation and execution of a CUDA program

- Source files must have the extension **.cu** (e.g., program.cu)
- **Compilation: nvcc program.cu -o program**
  - Possible to specify the target architecture with **-arch sm_xx** (e.g. -arch sm_75 for Turing)
- **Execution: ./program**

References

Contact

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