Introduction to GPU Programming

Oguz Kaya

Ma^ıtre de Conf'erences
Universit'e Paris–Saclay and the LRI ParSys team, Orsay, France

Outline

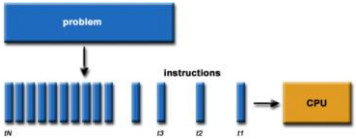
Outline

Objectives

- Get to know parallel computing.
- Discover the applications that need computing power. Explore the
- modern architecture of a parallel computer.
- Give a quick introduction to GPU programming with examples. Learn how
- to compile, run, launch a GPU program.

Equential programming

Traditionally, software is based on **sequential** calculation:

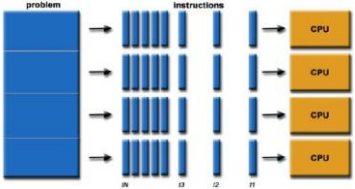


- A problem is often encountered in instructions.
- These instructions are executed **sequentially** one after the other.
- They are executed by **a single processor**. At a given moment, only one instruction is executed.
- The performance is mainly determined by **the frequency** (Hz) of the processor.

Parallel programming

Parallel programming allows the use of several computing resources to solve a problem:

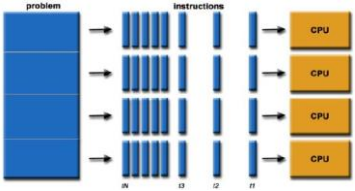
- A problem is divided into parts that can ˆetre launched in parallel
- Each part is still under instruction.
- The instructions of each part are executed **in parallel** using several processors.
- The performance is determined by:
  - The frequency of the processor
  - The number of processors
  - The degree of parallelism of the problem



Outline

Applications of parallel computing

Many time-consuming applications in various fields:

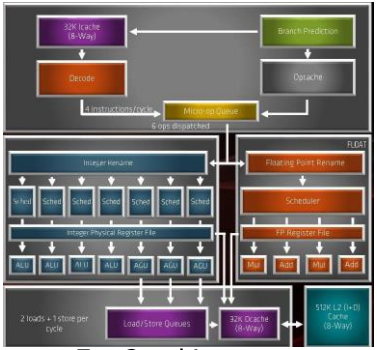


- Scientific computing: Simulations in physics, chemistry, biology, …
- Neural network processing (rendering,
- video games, etc.)
- Operating systems (Linux, Android, etc.) and
- many others...

Outline

CPU

"Central Processing Unit, a general computing unit consisting of:



Zen 2 architecture

- Several execution units (hearts)
- Multiple memory levels (registers, L1, L2, L3, RAM)
- Several execution ports in each core (ALUs, vector units)
- Vector units (AVX2, AVX512, Arm Neon, ...)
- Execution of some threads (1-4) simultaneously
- Able to exploit parallelism at the instruction level (micro-op buffer, instruction renaming, register renaming, ...)
- A considerable part of the circuit is dedicated to ILP + cache

GPU

"Graphical Processing Unit", a specific virtual computing unit consisting of

The Nvidia Ampere architecture

- Multiple execution units (symmetric multiprocessors (SM))
- Several memory levels (registers, shared memory, L1, L2, RAM)
- Several large (2-4) vector units (16-32 floats) in each DM
- Execution of thousands of simultaneous threads
- Large register array (65K)
- Very fast thread exchange
- Most of the circuit is devoted to vector units
- Parall'elisation takes the effort

GPU (cont.)

The Nvidia Ampere architecture

Supercomputer / Cluster

A set of machines (CPU+GPU) connected

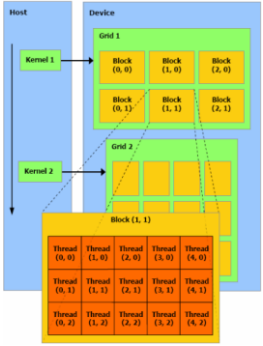
Jolio Curie supercomputer,
300K CPU cores, 1024 GPUs

- Connection by a network with a particular topology (ring, grid, torus, clique, etc.)
- Topology-adapted communications libraries
- Able to address very large problems
- Today, at the exaflops scale (^{1018} floating operations per second)

Outline

GPU Programming

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

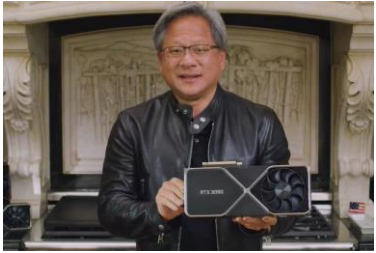


Execution of GPU kernels by blocks and threads

- A GPU consists of several identical processors (as CPUs) called streaming multiprocessors (SM).
- Each SM has several "cores", and each core can run a thread simultaneously (coming soon ...)
- There is a kernel (function) to be executed by all SMs/threads.
- A **block** is the execution of the kernel in a SM.
- The calculation in a block differs from the 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 with Nvidia Fanboy

- A compiler (nvcc) Basic
- library (cuda.h)
- Many libraries with optimized kernels (cuBLAS, cuDNN, cuSOLVER, cuTENSOR, RAPIDS, ...)

Hello World in OpenMP

```
# include < c s t d i o >
# include " omp . h"

int main ( int argc , char  * *  a r g v )
{
# pragma omp p a r a l l e l num threads ( 3 )
  {
    int t h i d = omp get thread num ( ) ;
    int numth = omp get num threads ( ) ;
    p r i n t f ( " Hello from thread u% d/% d . \n" , t h i d , numth ) ;
  }
  return 0 ;
}
```

- **omp.h** is the OpenMP library that provides the necessary functions (e.g., to get thid, numth).
- **#pragma omp parallel** cr'ee 3 threads that execute **the mˆeme code asynchronously**.
- Each thread has a unique identifier between 0 and 2 (or P - 1 if we cr'ee P threads)
- Possible exits ???
  - 3! Possibility for asynchronous threads

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 the necessary functions.
- **__global__** defines a GPU kernel (otherwise default CPU function)
  - Defines **blockIdx.x** and **gridDim.x**
- **<<< 3, 1 >>>** cr'ee 3 blocks (chaqun having 1 single thread, `a venir) which execute **the m^eme code asynchronously**.
- Each block has a unique identifier between 0 and 3 (or P - 1 if we run the kernel with P blocks)
- **blockIdx.x** is predefined and gives the identifier of a block in a GPU kernel.
- **gridDim.x** is predefined and gives the number of blocks used in the running GPU kernel.
- Possible exits ???

Multiplying an array in OpenMP

Multiply each element of an array **A[N]** by a scalar **c**.

```
# include < c s t d i o >
# include " omp . h"

# define N 1024

int main ( int argc , char * * a r g v )
{
   float A [ N ] ;
   float c = 2 . 0 ;
   // Initialization
   for ( int i = 0 ; i < N; i ++) { A [ i ] = i ; }
# pragma omp p a r a l l e l num threads ( 4 )
   {
      for ( int i = 0; i < N; i ++) {
         A [ i ] * = c ;
      }
   }
   return 0 ;
}
```

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

Multiplying an array in OpenMP (cont.)

Multiply each element of an array **A[N]** by a scalar **c**.

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

#define N 1024

int main(int argc, char **argv)
{
  float A[N];
  float c = 2.0;
  // Initialization
  for(int i = 0; i < N; i++) { A[i] = i; }
#pragma ompparallelnumthreads(4)
  {
    int thid = omp_get_thread_num();int
    numth = omp_get_num_threads();int
    elem Par Th = N / numth;
    int begin = thid * elem Par Th;
    int end;
    if(thid < numth - 1) { // Before the last thread
      end = (thid + 1) * elem Par Th;
    } else { // The last thread
      end = N;
    }
    for(int i = begin; i < end; i++) {
      A[i] *= c;
    }
  }
  return 0;
}
```

- Each thread always executes the m^eme code. This
- time, the execution is differentiated by the **thid**.
- With *P* threads, each thread runs through *N/P* 'cons'ecutive elements of the array **A[N]**.
- Be careful with the last thread if *P* does not divide *N*.

Multiplying 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)
{
  // 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*.

Compiling and running a CUDA program

- Source files must have the extension **.cu** (e.g. program.cu)
- **Compilation: nvcc program.cu -o program**
  - Possibility to specify the architecture with **-arch sm xx** (e.g. -arch sm 75 for Turing)
- **Execution: ./program**

Contact

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