Introduction, CUDA Basics

Jiří Filipovič

Fall 2021

About the class

The class is focused on algorithm design and programming of *general purpose* computing applications on *many-core vector processors*

About the class

The class is focused on algorithm design and programming of *general purpose* computing applications on *many-core vector processors*

We will focus to CUDA GPUs first:

- C for CUDA is good for teaching (easy API, a lot of examples available, mature compilers and tools)
- restricted to NVIDIA GPUs and x86 CPUs (with PGI)

About the class

The class is focused on algorithm design and programming of *general purpose* computing applications on *many-core vector processors*

We will focus to CUDA GPUs first:

- C for CUDA is good for teaching (easy API, a lot of examples available, mature compilers and tools)
- restricted to NVIDIA GPUs and x86 CPUs (with PGI)

After learning CUDA, we focus to OpenCL

- programming model very similar to CUDA, easy to learn when you already know CUDA
- can be used with various HW devices
- we will focus on code optimizations for x86, Intel MIC (Xeon Phi) and AMD GPUs

The class is practically oriented – besides efficient parallelization, we will focus on writing efficient code.

What is offered

You will learn:

- architecture of NVIDIA and AMD GPUs, Xeon Phi
- architecture-aware design of data-parallel algorithms
- programming in C for CUDA and OpenCL
- performance tuning and profiling
- basic tools and libraries for CUDA GPUs
- use cases

What is expected from you

During the semester, you will work on a practically oriented project

- important part of your total score in the class
- the same task for everybody, we will compare speed of your implementation
- 50 + 20 points of total score
    - working code: 25 points
    - efficient implementation: 25 points
    - speed of your code relative to your class mates: at most 20 points (only to improve your final grading)

Exam (oral or written, depending on the number of students)

- 50 points

Grading

For those finishing by exam:

- A: 92–100
- B: 86–91
- C: 78–85
- D: 72–77
- E: 66–71
- F: 0–65 pts

For those finishing by colloquium:

- 50 pts

Materials – CUDA

CUDA documentation (installed as a part of CUDA Toolkit, downloadable from *developer.nvidia.com*)

- CUDA C Programming Guide (most important properties of CUDA)
- CUDA C Best Practices Guide (more detailed document focusing on optimizations)
- CUDA Reference Manual (complete description of C for CUDA API)
- other useful documents (nvcc guide, PTX language description, library manuals, . . .)

CUDA article series, Supercomputing for the Masses

- http://www.ddj.com/cpp/207200659

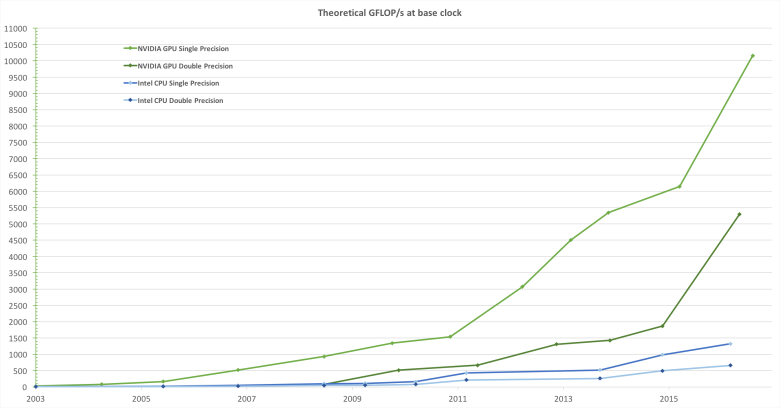
Materials – OpenCL

- OpenCL 1.1 Specification
- AMD Accelerated Parallel Processing Programming Guide
- Intel OpenCL SDK Programming Guide
- Writing Optimal OpenCL Code with Intel OpenCL SDK

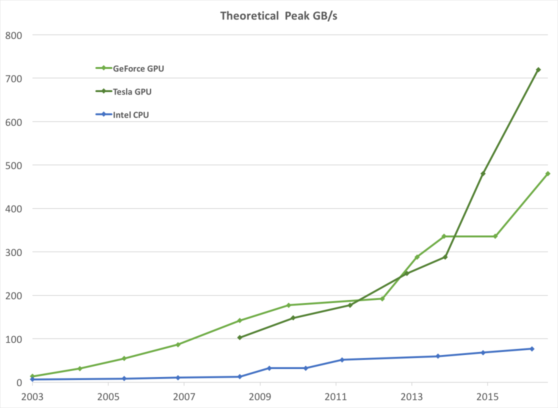
Materials – Parallel Programming

- Ben-Ari M., Principles of Concurrent and Distributed Programming, 2nd Ed. Addison-Wesley, 2006
- Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill, Patterns for Parallel Programming, Addison-Wesley, 2004

Motivation – GPU arithmetic performance



Motivation – GPU memory bandwidth



Motivation – programming complexity

OK, GPUs are more powerful, but GPU programming is substantially more difficult, right?

- well, it is more difficult comparing to writing serial C/C++ code...
- but can we compare it to serial code?

Motivation – programming complexity

OK, GPUs are more powerful, but GPU programming is substantially more difficult, right?

- well, it is more difficult comparing to writing serial C/C++ code...
- but can we compare it to serial code?

Moore's Law

Number of transistors on a single chip doubles every 18 months

Motivation – programming complexity

OK, GPUs are more powerful, but GPU programming is substantially more difficult, right?

- well, it is more difficult comparing to writing serial C/C++ code...
- but can we compare it to serial code?

Moore's Law

Number of transistors on a single chip doubles every 18 months

Corresponding growth of performance comes from

- **in the past:** frequency increase, instruction parallelism, out-of-order instruction processing, caches, etc.
- **today:** vector instructions, increase in number of cores

Motivation – paradigm change

Moore's Law consequences:

- **in the past:**changes were important for compiler developers; application developers didn't need to worry
- **today**: in order to utilize state-of-the-art processors, it is necessary to write parallel and vectorized code
  - it is necessary to find parallelism in the problem being solved, which is a task for a programmer, not for a compiler (at least for now)
  - writing efficient code for modern CPUs is similarly difficult as writing for GPUs

Electrostatic Potential Map

Important problem from computational chemistry

- we have a molecule defined by position and charges of its atoms
- the goal is to compute charges at a 3D spatial grid around the molecule

In a given point of the grid, we have

$$V_i = \sum_j \frac{w_j}{4\pi\epsilon_0 r_{ij}}$$

Where w_j is charge of the j-th atom, r_{ij} is Euclidean distance between atom j and the grid point i and ϵ_0 is vacuum permittivity.

Electrostatic Potential Map

Initial implementation

- suppose we know nothing about HW, just know C++
- algorithm needs to process 3D grid such that it sums potential of all atoms for each grid point
- we will iterate over atoms in outer loop, as it allows to precompute positions of grid points and minimizes number of accesses into input/output array

Electrostatic Potential Map

```
void coulomb(const sAtom* atoms, const int nAtoms,
    const float gs, const int gSize, float *grid) {
  for (int a = 0; a < nAtoms; a++) {
    sAtom myAtom = atoms[a];
    for (int x = 0; x < gSize; x++) {
      float dx2 = powf((float)x * gs - myAtom.x, 2.0f);
      for (int y = 0; y < gSize; y++) {
        float dy2 = powf((float)y * gs - myAtom.y);
        for (int z = 0; z < gSize; z++) {
          float dz = (float)z * gs - myAtom.z;
          float e = myAtom.w / sqrtf(dx2 + dy2 + dz*dz);
          grid[z*gSize*gSize + y*gSize + x] += e;
        }
      }
    }
  }
}
```

Electrostatic Potential Map

Execution on 4-core CPU at 3.6GHz (Sandy Bridge) + GeForce GTX 1070 (Pascal)

- naive implementation **164.7** millions of atoms evaluated per second (MEvals/s)

Electrostatic Potential Map

Execution on 4-core CPU at 3.6GHz (Sandy Bridge) + GeForce GTX 1070 (Pascal)

- naive implementation **164.7** millions of atoms evaluated per second (MEvals/s)
- **476.9** Mevals/s when optimized cache: **2.9**× speedup

Electrostatic Potential Map

Execution on 4-core CPU at 3.6GHz (Sandy Bridge) + GeForce GTX 1070 (Pascal)

- naive implementation **164.7** millions of atoms evaluated per second (MEvals/s)
- **476.9** Mevals/s when optimized cache: **2.9**× speedup
- **2,577** Mevals/s when vectorized: **15.6**× speedup

Electrostatic Potential Map

Execution on 4-core CPU at 3.6GHz (Sandy Bridge) + GeForce GTX 1070 (Pascal)

- naive implementation **164.7** millions of atoms evaluated per second (MEvals/s)
- **476.9** Mevals/s when optimized cache: **2.9**× speedup
- **2,577** Mevals/s when vectorized: **15.6**× speedup
- **9,914** Mevals/s when parallelized: **60.2**× speedup

Electrostatic Potential Map

Execution on 4-core CPU at 3.6GHz (Sandy Bridge) + GeForce GTX 1070 (Pascal)

- naive implementation **164.7** millions of atoms evaluated per second (MEvals/s)
- **476.9** Mevals/s when optimized cache: **2.9**× speedup
- **2,577** Mevals/s when vectorized: **15.6**× speedup
- **9,914** Mevals/s when parallelized: **60.2**× speedup
- **537,900** Mevals/s GPU version: **3266**× speedup

GPU speedup over already tuned CPU code is 54×, but the optimization effort is similar for CPU and GPU. In this class, you will learn how to optimize the code.

Why are GPUs so powerful?

Types of Parallelism

- Task parallelism
  - decomposition of a task into the problems that may be processed in parallel
  - usually more complex tasks performing different actions
  - usually more frequent (and complex) synchronization
  - ideal for small number of high-performance processors
- Data parallelism
  - parallelism on the level of data structures
  - usually the same operations on many items of a data structure
  - finer-grained parallelism allows for simple construction of individual processors

Why are GPUs so powerful?

From programmer's perspective

- some problems are rather data-parallel, some task-parallel (graph traversal vs. matrix multiplication)

From hardware perspective

- processors for data-parallel tasks may be simpler
- it is possible to achieve higher arithmetic performance with the same size of a processor
- simpler memory access patterns allow for high-throughput memory designs

GPU Architecture

GPU Architecture

Main differences compared to CPU

- high parallelism: hundreds thousands threads needed to utilize high-end GPUs
- SIMT model: subsets of threads runs in lock-step mode
- distributed on-chip memory: subsets of threads shares their private memory
- restricted caching capabilities: small cache, often read-only

Algorithms usually need to be redesigned to be efficient on GPU.

GPU Architecture

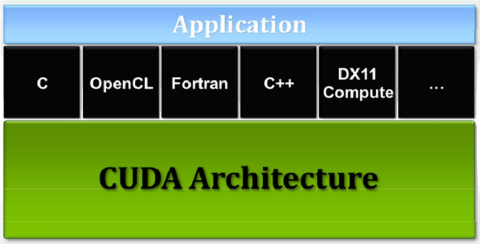
Within the system:

- co-processor with dedicated memory (discrete GPU)
- asynchronous processing of instructions
- attached using PCI-E to the rest of the system (discrete GPU)

CUDA

CUDA (Compute Unified Device Architecture)

- architecture for parallel computations developed by NVIDIA
- provides a new programming model, allows efficient implementation of general GPU computations
- may be used in multiple programming languages



G80 Processor

G80

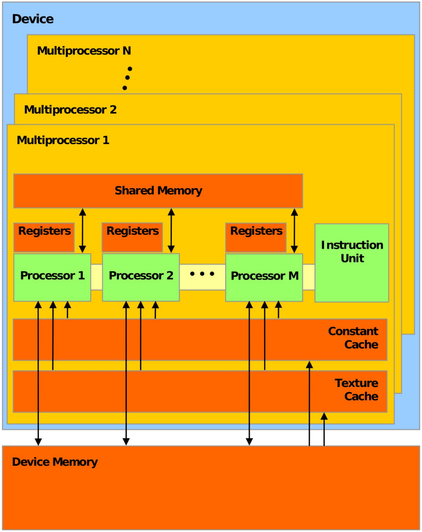
- the first CUDA processor
- 16 multiprocessors
- each multiprocessor
  - 8 scalar processors
  - 2 units for special functions
  - up to 768 threads
    - HW for thread switching and scheduling
  - threads are grouped into warps by 32
    - SIMT
  - native synchronization within the multiprocessor

G80 Memory Model

Memory model

- 8192 registers shared among all threads of a multiprocessor
- 16 kB of shared memory
  - local within the multiprocessor
  - as fast as registry (under certain constraints)
- constant memory
  - cached, read-only
- texture memory
  - cached with 2D locality, read-only
- global memory
  - non cached, read-write
- data transfers between global memory and system memory through PCI-E

G80 Processor



C for CUDA

C for CUDA is an extension of C for parallel computations

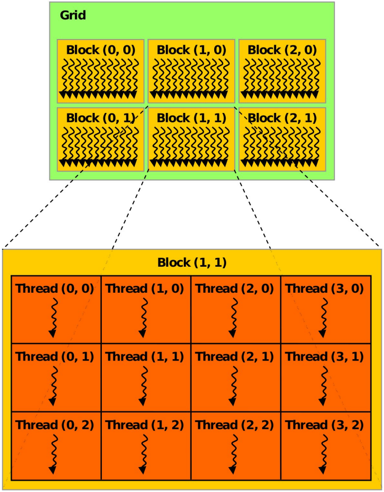
- explicit separation of host (CPU) and device (GPU) code
- thread hierarchy
- memory hierarchy
- synchronization mechanisms
- API

Thread Hierarchy

Thread hierarchy

- threads are organized into blocks
- blocks form a grid
- problem is decomposed into sub-problems that can be run independently in parallel (blocks)
- individual sub-problems are divided into small pieces that can be run cooperatively in parallel (threads)
- all threads from a block run on the same multiprocessor
- scales well

Thread Hierarchy

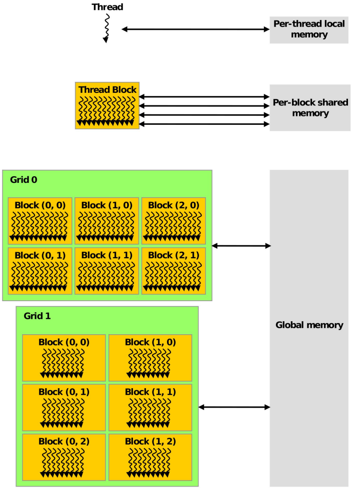


Memory Hierarchy

More memory types:

- different visibility
- different lifetime
- different speed and behavior
- brings good scalability

Memory Hierarchy



An Example – Sum of Vectors

We want to sum vectors a and b and store the result in vector c

An Example – Sum of Vectors

We want to sum vectors a and b and store the result in vector c
We need to find parallelism in the problem.

An Example – Sum of Vectors

We want to sum vectors *a* and *b* and store the result in vector *c*

We need to find parallelism in the problem.

Serial sum of vectors:

```
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

An Example – Sum of Vectors

We want to sum vectors *a* and *b* and store the result in vector *c*

We need to find parallelism in the problem.

Serial sum of vectors:

```
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

Individual iterations are independent – it is possible to parallelize, scales with the size of the vector.

An Example – Sum of Vectors

We want to sum vectors *a* and *b* and store the result in vector *c*

We need to find parallelism in the problem.

Serial sum of vectors:

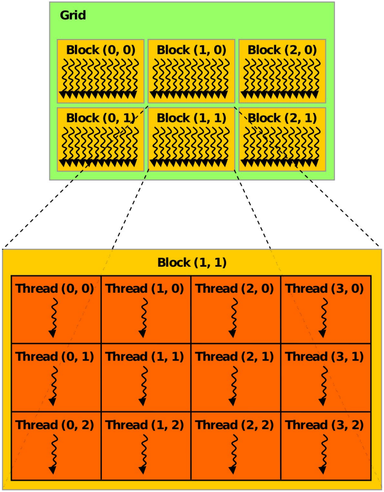
```
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

Individual iterations are independent – it is possible to parallelize, scales with the size of the vector.
i-th thread sums i-th component of the vector:

```
c[i] = a[i] + b[i];
```

How do we find id of the thread?

Thread Hierarchy



Thread and Block Identification

C for CUDA has built-in variables:

- **threadIdx.**{**x, y, z**} tells position of a thread in a block
- **blockDim.**{**x, y, z**} tells size of the block
- **blockIdx.**{**x, y, z**} tells position of the block in grid (z always equals 1)
- **gridDim.**{**x, y, z**} tells grid size (z always equals 1)

An Example – Sum of Vectors

Thus we calculate the position of the thread (grid and block are one-dimensional):

An Example – Sum of Vectors

Thus we calculate the position of the thread (grid and block are one-dimensional):

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

An Example – Sum of Vectors

Thus we calculate the position of the thread (grid and block are one-dimensional):

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

Whole function for parallel summation of vectors:

```
__global__ void addvec(float *a, float *b, float *c){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}
```

An Example – Sum of Vectors

Thus we calculate the position of the thread (grid and block are one-dimensional):

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

Whole function for parallel summation of vectors:

```
__global__ void addvec(float *a, float *b, float *c){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}
```

The function defines so called kernel; we specify how meny threads and what structure will be run when calling.

Function Type Quantifiers

C syntax enhanced by quantifiers defining where the code is executed and from where it can be called:

- __**device**__ function is run on device (GPU) only and can be called from the device code only
- __**global**__ function is run on device (GPU) only and can be called from the host (CPU) code only
- __**host**__ function is run on host only and can be called from the host only
- __host__ and __device__ may be combined – function is compiled for both then

The following steps are needed for the full computation:

The following steps are needed for the full computation:

- allocate memory for vectors and fill it with data

The following steps are needed for the full computation:

- allocate memory for vectors and fill it with data
- *allocate memory on GPU*

The following steps are needed for the full computation:

- allocate memory for vectors and fill it with data
- *allocate memory on GPU*
- *copy vectors a a b to GPU*

The following steps are needed for the full computation:

- allocate memory for vectors and fill it with data
- *allocate memory on GPU*
- *copy vectors a a b to GPU*
- **compute the sum on GPU**

The following steps are needed for the full computation:

- allocate memory for vectors and fill it with data
- *allocate memory on GPU*
- *copy vectors a a b to GPU*
- **compute the sum on GPU**
- *store the result from GPU into c*

The following steps are needed for the full computation:

- allocate memory for vectors and fill it with data
- *allocate memory on GPU*
- *copy vectors* a *a* b *to GPU*
- **compute the sum on GPU**
- *store the result from GPU into* c
- use the result in c :-)

When managed memory is used (requires GPU with computing capability 3.0 and CUDA 6.0 or better), steps written in italics are not required.

An Example – Sum of Vectors

CPU code that fills *a* and *b* and computes *c*

```
#include <stdio.h>
#define N 64
int main(){
  float *a, *b, *c;
  cudaMallocManaged(&a, N*sizeof(*a));
  cudaMallocManaged(&b, N*sizeof(*b));
  cudaMallocManaged(&c, N*sizeof(*c));
  for (int i = 0; i < N; i++) {
    a[i] = i;
    b[i] = i*3;
  }

// GPU code will be here

  for (int i = 0; i < N; i++)
    printf("%f, ", c[i]);

  cudaFree(a); cudaFree(b); cudaFree(c);

  return 0;
}
```

GPU Memory Management

Using managed memory, CUDA maintains memory transfers between CPU and GPU automatically.

- memory coherency is guaranteed
- GPU memory cannot be used when any GPU kernel is running

Memory operations can be programmed explicitly

```
cudaMalloc(void** devPtr, size_t count);
cudaFree(void* devPtr);
cudaMemcpy(void* dst, const void* src, size_t count,
  enum cudaMemcpyKind kind);
```

An Example – Sum of Vectors

Running the kernel:

- kernel is called as a function; between the name and the arguments, there are triple angle brackets with specification of grid and block size
- we need to know block size and their count
- we will use 1D block and grid with fixed block size
- the size of the grid is determined in a way to compute the whole problem of vector sum

For vector size divisible by 32:

```
#define BLOCK 32
addvec<<<N/BLOCK, BLOCK>>>(a, b, c);
```

How to solve a general vector size?

An Example – Sum of Vectors

We will modify the kernel source:

```
__global__ void addvec(float *a, float *b, float *c, int n){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) c[i] = a[i] + b[i];
}
```

And call the kernel with sufficient number of threads:

```
addvec<<<N/BLOCK + 1, BLOCK>>>(a, b, c, N);
```

An Example – Running It

Now we just need to compile it :-)

```
nvcc -o vecadd vecadd.cu
```

Where to work with CUDA?

- on a remote computer: airacuda.fi.muni.cz, barracuda.fi.muni.cz, accounts will be made
- your own machine: download and install CUDA toolkit and SDK from developer.nvidia.com