Introduction, CUDA Basics

Jiří Filipovič

Fall 2013

What is included

The class is focused on algorithm design and programming of *general purpose* computing applications on graphical processors

What is included

The class is focused on algorithm design and programming of *general purpose* computing applications on graphical processors

We will learn:

- design of parallel algorithms with focus on utilization of programming model available in todays GPU

What is included

The class is focused on algorithm design and programming of *general purpose* computing applications on graphical processors

We will learn:

- design of parallel algorithms with focus on utilization of programming model available in todays GPU
- CUDA-based GPU architectures

What is included

The class is focused on algorithm design and programming of *general purpose* computing applications on graphical processors

We will learn:

- design of parallel algorithms with focus on utilization of programming model available in todays GPU
- CUDA-based GPU architectures
- programming in C for CUDA

What is included

The class is focused on algorithm design and programming of *general purpose* computing applications on graphical processors

We will learn:

- design of parallel algorithms with focus on utilization of programming model available in todays GPU
- CUDA-based GPU architectures
- programming in C for CUDA
- tools and libraries

What is included

The class is focused on algorithm design and programming of *general purpose* computing applications on graphical processors

We will learn:

- design of parallel algorithms with focus on utilization of programming model available in todays GPU
- CUDA-based GPU architectures
- programming in C for CUDA
- tools and libraries
- code optimization for CUDA

What is included

The class is focused on algorithm design and programming of *general purpose* computing applications on graphical processors

We will learn:

- design of parallel algorithms with focus on utilization of programming model available in todays GPU
- CUDA-based GPU architectures
- programming in C for CUDA
- tools and libraries
- code optimization for CUDA
- case studies

What is included

The class is focused on algorithm design and programming of *general purpose* computing applications on graphical processors
We will learn:

- design of parallel algorithms with focus on utilization of programming model available in todays GPU
- CUDA-based GPU architectures
- programming in C for CUDA
- tools and libraries
- code optimization for CUDA
- case studies

The class is practically orented – GPU is constant-times faster than CPU, therefore besides time complexity, writing an optimal code is important.

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: 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, . . .)

University of Illinois textbook

- available from http://courses.ece.illinois.edu/ece498/al/Syllabus.html

CUDA article series, Supercomputing for the Masses

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

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 – Moore's Law

Moore's Law

Number of transistors on a single chip doubles every 18 months

Motivation – Moore's Law

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

Motivation – paradigm change

Moore's Law consequences:

- **in the past:** speed of a single-threaded program doubled each 18 months
  - changes were important for compiler developers; application developers didn't need to worry
- **today**: speed of prcessing of a parallel program having sufficient number of processes/threads doubles every 18 months
  - in order to utilize state-of-the-art processors, it is necessary to devleop parallel algorithms
  - 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)

Motivation – 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
  - ideal for small number of high-performance processor goals
  - more frequent (and complex) synchronization, usually
- Data parallelism
  - paralellism 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

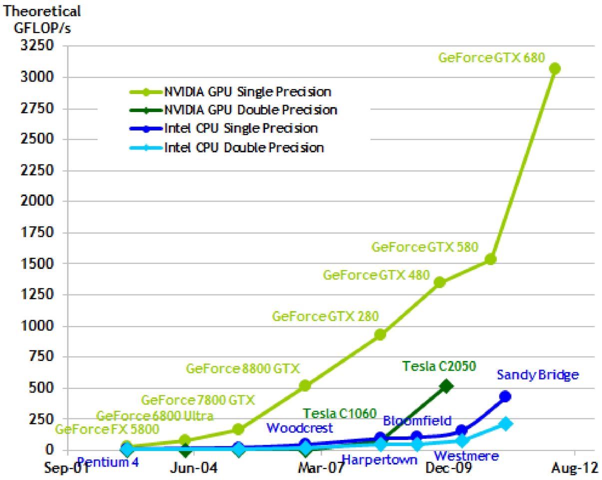
Motivation – Types of Parallelism

- from programmer's perspective
  - different paradigm requires different approach to algorithm design
  - some problems are rather data-parallel, some task-parallel
- from hardware perspective
  - processors for data-parallel tasks may be simpler
  - it si possible to achieve higher arithmetic performance with the same number of processors
  - simpler memory access patterns allow for high-throughput memory designs

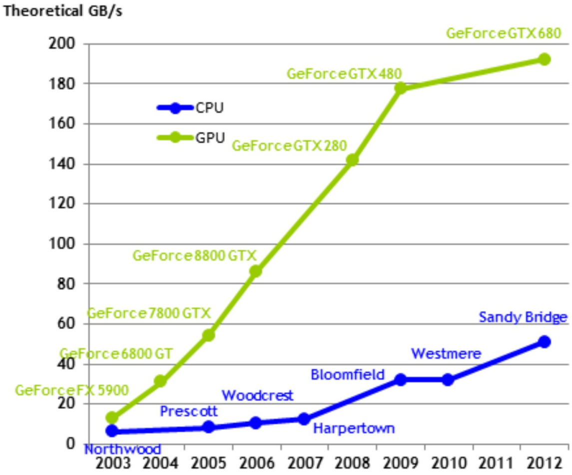
Motivace – Graphical Computations

- Data parallel
  - the same task implemented for each pixel/vertex
- Predefined functions
- Programmable functions
  - special graphics effects
  - GPU become more and more programmable
  - it is possible to implement also non-graphics tasks

Motivation – Performance



Motivation – Performance



Motivation – Summary

- GPUs are powerful
  - an order of magnitude performance increase is worth studying a new programming model
- for full utilization of modern GPUs and CPUs, parallel programming is necessary
  - parallel architecture of GPUs ceases to be an order of magnitude harder to master
- GPUs are widespread
  - cheap
  - lots of users have a desktop supercomputer

Motivation – Applications

Use of GPU for general computations is a dynamically developing field with broad applicability

Motivation – Applications

Use of GPU for general computations is a dynamically developing field with broad applicability

- high-performance scientific calculations
  - computational chemistry
  - physical simulations
  - image processing
  - and others. . .

Motivation – Applications

Use of GPU for general computations is a dynamically developing field with broad applicability

- high-performance scientific calculations
  - computational chemistry
  - physical simulations
  - image processing
  - and others. . .
- performance-hungry home and desktop applications
  - encoding/decoding of multimedia data
  - game physics
  - image editing, 3D rendering
  - etc.

Motivation – Applications

SW developers are still a sought-for scarce resource. . .

Motivation – Applications

SW developers are still a sought-for scarce resource. . .
SW developers capable of parallel SW development are extremely sought-for scarce resource

Motivation – Applications

SW developers are still a sought-for scarce resource. . .

SW developers capable of parallel SW development are extremely sought-for scarce resource

A lot of existing software is not parallel

- it is necessary to make it parallel in order to increase performance
- and somebody has to do it :-)

Historic Excursion

- SIMD model since '60s
  - Solomon project by Westinghouse company at the beginning of '60s
  - transferred to University of Illinois as ILLIAC IV
  - separate ALU for each data element – massively parallel
  - original plan: 256 ALUs, 1 GFLOPS
  - finished in 1972, 64 ALUs, 100–150 MFLOPS
- in '80s–90s: vector supercomputers, TOP500
- in todays CPUs: SSE (x86), ActiVec (PowerPC)
- Cg: programming vertex and pixel shaders in graphics grads (cca 2003)
- CUDA: general GPU programming, SIMT model (first released on 15. February 2007)
- future?
  - OpenCL
  - higher programming languages, automatic parallelization

GPU Architecture

CPU vs. GPU

- couple of cores vs. vs. tens of multiprocessors
- out of order vs. in order
- MIMD, SIMD short vectors vs. SIMT for long vectors
- large cache vs. small cache, often read-only

GPU uses more transistors for computating units then for cache and control $\implies$ higher performance, less flexibility

GPU Architecture

GPU Architecture

Within the system:

- co-processor with dedicated memory
- asychnornous processing of instructions
- attached using PCI-E to the rest of the system

G80 Processor

G80

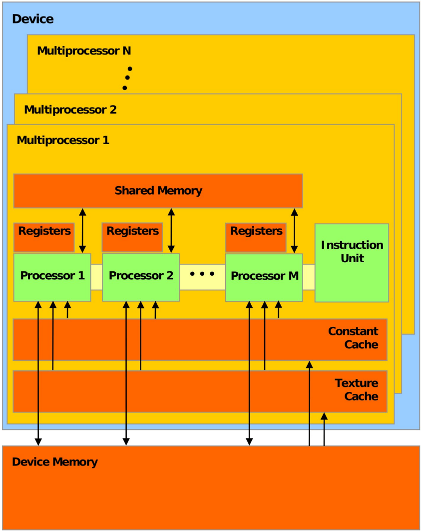
- first CUDA processor
- 16 multiprocessors
- each multiprocessor
  - 8 scalar processors
  - 2 units for special functions
  - up to 768 threads
    - HW for thread switching and scheduling
  - threads are grouped into warps by 32
    - SIMT
  - native synchronization within the multiprocessor

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



Further Development

Processors based on G80

- double-precision calculations
- relaxed rules for efficient memory access to global memory
- more of on-chip resources (more registers, more threads per MP)
- better sychronization options (atomic operations, warp voting)

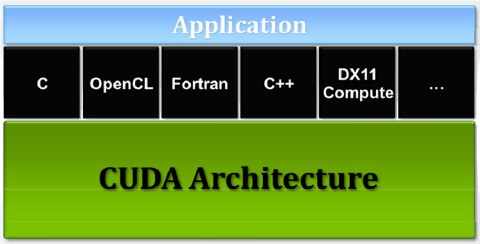
Fermi

- higher parallelization on multiprocessor level (more cores, two warp schedulers, higher double-precission performance)
- configurable L1 and shared L2 cache
- flat address space
- better floating point precision
- parallel run of kernels
- better synchronization tools
- other changes stemming from a different architecture

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



C for CUDA

C for CUDA is extension of C for parallel computations

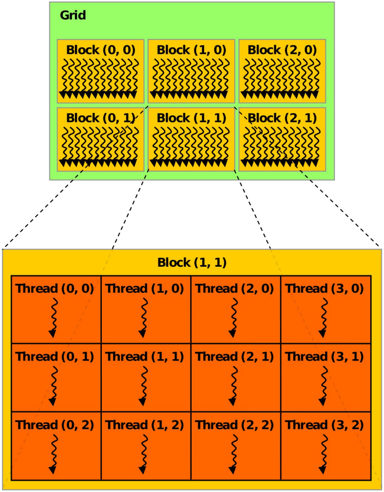
- explicit separation of host (CPU) and device (GPU) code
- thread hierarchy
- memory hierarchy
- synchronization mechanisms
- API

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)
- scales well

Thread Hierarchy

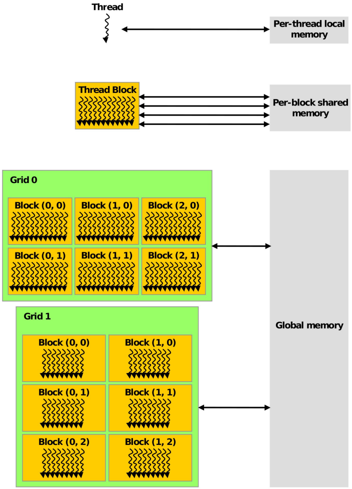


Memory Hierarchy

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 $\vec{a}$ and $\vec{b}$ and store the result in vector $\vec{c}$

An Example – Sum of Vectors

We want to sum vectors $\vec{a}$ and $\vec{b}$ and store the result in vector $\vec{c}$

We need to find parallelism in the problem.

An Example – Sum of Vectors

We want to sum vectors $\vec{a}$ and $\vec{b}$ and store the result in vector $\vec{c}$

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 $\vec{a}$ and $\vec{b}$ and store the result in vector $\vec{c}$
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 $\vec{a}$ and $\vec{b}$ and store the result in vector $\vec{c}$
We need to find parallelism in the problem.
Serial sum of vectors:

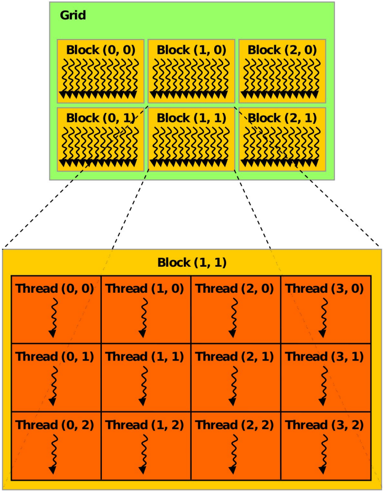
```
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

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 which thread we are?

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 run and from where it may be called:

- __**device**__ function is run on device (GPU) only and may be called from the device code only
- __**global**__ function is run on device (GPU) only and may be called from the host (CPU) code only
- __**host**__ function is run on host only and may 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** $\vec{c}$

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 $\vec{c}$**
- use the result in $\vec{c}$:-)

An Example – Sum of Vectors

CPU code that fills $\vec{a}$ and $\vec{b}$ and computes $\vec{c}$

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

// GPU code will be here

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

GPU Memory Management

It is necessary to allocate the memory dynamically.

```
cudaMalloc(void** devPtr, size_t count);
```

allocates memory of the *count* size and sets the pointer *devPtr* to it.
To release the memory:

```
cudaFree(void* devPtr);
```

To copy the memory:

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

copies *count* bytes from *src* to *dst*, *kind* determins copying direction (e.g., *cudaMemcpyHostToDevice*, or *cudaMemcpyDeviceToHost*).

An Example – Sum of Vectors

We allocate the memory and transfer the data:

```
float *d_a, *d_b, *d_c;
cudaMalloc((void**)&d_a, N*sizeof(*d_a));
cudaMalloc((void**)&d_b, N*sizeof(*d_b));
cudaMalloc((void**)&d_c, N*sizeof(*d_c));

cudaMemcpy(d_a, a, N*sizeof(*d_a), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, N*sizeof(*d_b), cudaMemcpyHostToDevice);

// the kernel will be run here

cudaMemcpy(c, d_c, N*sizeof(*c), cudaMemcpyDeviceToHost);

cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```

An Example – Sum of Vectors

Running the kernel:

- kernel is called as a function; between the name and the arguments, there are three 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 dividable by 32:

```
#define BLOCK 32
addvec<<<N/BLOCK, BLOCK>>>(d_a, d_b, d_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>>>(d_a, d_b, d_c, N);
```

An Example – Running It

Now we just need to compile it :-)

```
nvcc -I/usr/local/cuda/include -L/usr/local/cuda/lib -lcudart \
  -o vecadd vecadd.cu
```

Where to work with CUDA?

- on a remote computer: barracuda.fi.muni.cz, airacuda.fi.muni.cz, accounts will be made
- Windows stations in computer halls (will be specified later)
- your own machine: download and install CUDA toolkit and SDK from developer.nvidia.com
- source code used in lectures will be published as a part of course materials

Today we have demonstrated

- why it is good to know CUDA
- differences of GPUs
- C for CUDA basics

Today we have demonstrated

- why it is good to know CUDA
- differences of GPUs
- C for CUDA basics

Next lecture will focus on

- more detailed introduction to GPU from hardware perspective
- parallelism provided by GPU
- memory available to GPU
- more complex examples of GPU implementations

Today we have demonstrated

- why it is good to know CUDA
- differences of GPUs
- C for CUDA basics

Next lecture will focus on

- more detailed introduction to GPU from hardware perspective
- parallelism provided by GPU
- memory available to GPU
- more complex examples of GPU implementations

An assignment for you:

- try to compile your first CUDA program
- play with it if you like