


Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program of the Lobachevsky State University of Nizhni Novgorod among the world's research and education centers

Strategic initiative “Achieving leading positions in the field of supercomputer technology and high-performance computing”

Lobachevsky State University of Nizhni Novgorod

Faculty of Computational mathematics and cybernetics

Introduction to GPU programming

02 Lecture
CUDA C

Bastrakov S.I.
Software department

Contents

- Introduction to CUDA
- CUDA compute model
- CUDA C

Introduction to CUDA

NVIDIA CUDA

- **CUDA –** *Compute Unified Device Architecture*.
- Hardware and software platform for parallel general purpose computing on NVIDIA GPUs.
- Supports many programming interfaces
- Traditionally did not support other vendors. In the latest versions it is possible.
  - E.g. PGI CUDA-x86 http://www.pgroup.com/resources/cuda-x86.htm

Many programming interfaces

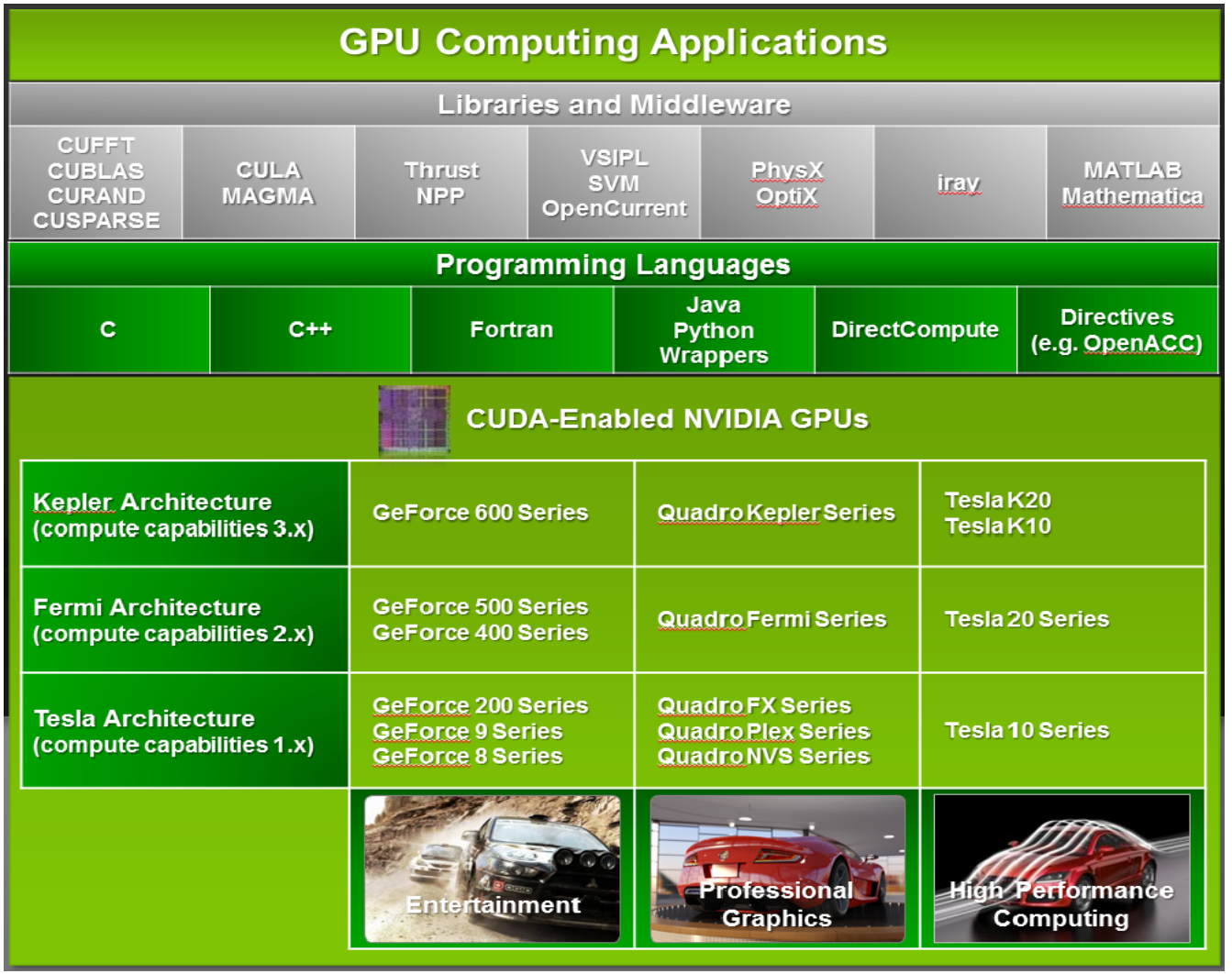

Image source: NVIDIA CUDA C Programming Guide v. 6.5

CUDA Kit

- CUDA driver.
- CUDA Toolkit.
  - compiler;
  - profiler;
  - optimized libraries;
  - documentation.
- GPU Computing SDK.

CUDA compute model

Main concepts

- Program with CUDA C consists of 2 parts:
  - CPU part written on C/C++ (including function main).
  - GPU executes special functions – **kernels** and functions called within kernels.
- Many **threads** execute kernel body in parallel.
- Kernel is called from CPU with specified number of threads.
- GPU programming mainly follows stream compute model.

Simple examples of kernels

- Vector addition: kernel computes one element of the result.
- Matrix multiplication (by definition): kernel computes one element of the result.
- Integrating PDE with explicit scheme: kernel computes one grid value.
- *The common property of these examples is that number of threads is equal to number of work elements. This is the most simple case, but not the only possible way.*

Data parallelism

- All examples given on the previous slide are *data parallel.*
- This is the most simple case for parallel programming technologies, including CUDA: each subset of data can be processed independently.
- **Data parallelism** – all cores perform same processing on different data.
- In sequential C++ implementation there is a loop with independent iterations.

Thread hierarchy

- Thread hierarchy is used to make a correspondence of thread model to hardware architecture.
- Threads are grouped into **thread blocks**.
- All thread blocks have the same size.
- Blocks are grouped into **grid**.
- Kernel is executed on a grid. Kernel call sets number of blocks and block size.

Identifiers

- Each thread and block have identifiers.
- They are used to find out which data to process in each thread.
- Identifiers are 3D (some components may be constant, thus giving 1D or 2D).
- Example: kernel performs matrix multiplication by definition, each thread computes one element. It is convenient to use 2D indexes so that x-components is row index and y-component is column index.

Identifiers

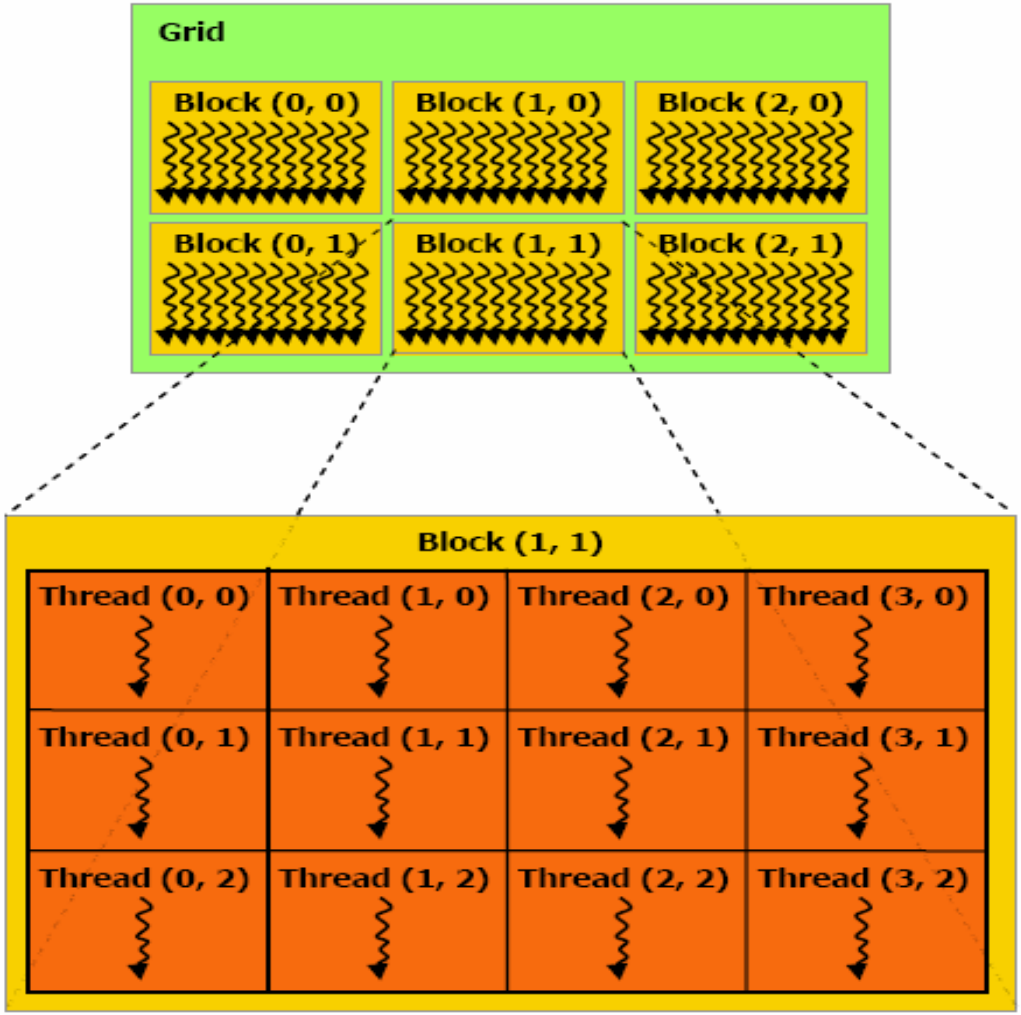

Image source: NVIDIA CUDA C Programming Guide v. 6.5

CUDA C

CUDA C

- CUDA C is extensions of C/C++ that consists of
  - function qualifiers;
  - memory qualifiers;
  - inline variables.
- In this lecture we cover basic subset of CUDA C.
- Terminology:
  - **host** = CPU;
  - **device** = GPU;
  - **kernel** = function that is executed on GPU in parallel.

Function qualifiers

Qualifier	*Executed on*	*Called from*
__host__	host	host
__global__	device	host
__device__	device	device

- **__host__** (by default) is a function called from host and executed on host. All usual C++ functions are host functions.
- **__global__** is a function called from host and executed in parallel on GPU (kernel).
- **__device__** is a function called inside kernel or other device function and executed on GPU.

Syntax examples

```
__host__ float hostSquare(float a) {
    return a * a;
}

__device__ float deviceSquare(float a) {
    return a * a;
}

__global__ void kernel(float a) {
    float a2 = deviceSquare(a);
}
```

Inline variables

- From code on GPU side the following variables are accessible:
  - **gridDim** – size of grid;
  - **blockIdx** – index of current block;
  - **blockDim** – size of block;
  - **threadIdx** – index of current thread inside current block.
- All indexes are 3D, access to components via .x, .y, .z.
- (0, 0, 0) ≤ blockIdx < gridDim, (0, 0, 0) ≤ threadIdx < blockDim.
- Read only.

Unique index of a thread

- threadIdx is local thread index inside block. There is no inline variable fro global thread index (among all blocks).
- But it can be computed.
- We assume indexes are 1D, i.e. only x-component is used.

idx = blockIdx.x * blockDim.x + threadIdx.x;

Global thread index

Shift of thread 0 of current block to thread 0 of block 0

Shift of current thread to thread 0 of current block

Example: vector addition

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

Example: matrix addition

```
__global__ void matAdd_kernel(const float * a, const
      float * b, float * result, int m, int n)

{

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

  int j = blockIdx.y * blockDim.y + threadIdx.y;

  int idx = i * n + j;

  result[idx] = a[idx] + b[idx];

}
```

Kernel calls

- Kernels are called with execution configuration.
- It is set by expression **<<< Dg, Db >>>** between kernel name and arguments.
- **Dg** is number of blocks, total number of blocks is Dg.x * Dg.y * Dg.z.
- **Db** is block size (all blocks have the same size), total number of threads per block is Db.x * Db.y * Db.z.
- Besides, there are two optional parameters with default values, we will consider those in the next lectures.

Kernel calls

- Grid and block sizes are variables of CUDA type dim3 that is vector of 3 integers.
- By default all components are initialized with 1. For 1D indexes one can use int values.
- Example:

```
some_kernel <<<  201, 500 >>> (some_args);
```

Calling kernel <some_kernel> with arguments <some_args> on a grid of 201 blocks, 500 threads per block. In total

201 * 500 = 100500 threads.

Example: calling vector addition kernel

- We use the kernel from previous example.
- Assume block size is 256.
- We need to compute number of blocks.

```
void vecAdd(const float * a, const float * b,
  float * result, int n)
{
  const int block_size = 256;
  int num_blocks = ?;
  vecAdd_kernel <<< num_blocks, block_size
     >>> (a, b, result);
}
```

Example: calling vector addition kernel

```
void vecAdd(const float * a, const float * b,
  float * result, int n)
{
  const int block_size = 256;
  int num_blocks =
   (n + block_size - 1) / block_size;
  vecAdd_kernel <<< num_blocks, block_size   >>>
  (a, b, result);
}
```

- Is it correct?

Fixed vector addition kernel

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

Treating alignment

- Choose number of blocks as upper integer part and add condition in kernels.
- Ensure alignment.
- Make workload of thread not fixed.
- Optimal choice depends on a situation.

“CUDA Hello, World!”…

```
#include <cstdlib>
#include <iostream>
#include <cuda_runtime.h>
__global__ void vecAdd_kernel(
          const float * a, const float * b,
          float * result, int n)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n)
    result[i] = a[i] + b[i];
}
```

“CUDA Hello, World!” …

```
int main() {
  int n = 1000;
  float * a = new float[n], * a_gpu;
  cudaMalloc((void**)&a_gpu, n *
     sizeof(float));
  float * b = new float[n], * b_gpu;
  cudaMalloc((void**)&b_gpu, n *
     sizeof(float));
  float * result = new float[n], * result_gpu;
  cudaMalloc((void**)&result_gpu, n *
     sizeof(float));
```

“CUDA Hello, World!” …

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

cudaMemcpy(a_gpu, a, n * sizeof(float),
   cudaMemcpyHostToDevice);
cudaMemcpy(b_gpu, b, n * sizeof(float),
   cudaMemcpyHostToDevice);
```

“CUDA Hello, World!”

```
const int block_size = 256;
int num_blocks =
    (n + block_size - 1) / block_size;
vecAdd_kernel <<< num_blocks, block_size
      >>> (a_gpu, b_gpu, result_gpu, n);
cudaMemcpy(result, result_gpu, n *
  sizeof(float), cudaMemcpyDeviceToHost);
delete [] a; delete [] b; delete [] result;
cudaFree(a_gpu); cudaFree(b_gpu);
  cudaFree(result_gpu);
return 0;
}
```

Building

- nvcc.
- Build rules for Microsoft Visual Studio 2008.
- From Visual Studio 2010 there is a special project type for CUDA.

References

- Sanders J., Kandrot E. CUDA by Example: An Introduction to General-Purpose GPU Programming. – Addison-Wesley Professional, 2010. – 312 p.
- NVIDIA CUDA C Programming Guide. [http://docs.nvidia.com/cuda/cuda-c-programming-guide/].

Authors

- Bastrakov S.I.,
  Assistant of the Software department of CMC faculty.
  bastrakov@vmk.unn.ru