The Ministry of Education and Science of the Russian Federation

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”

Introduction to GPU programming

Lecture 2. CUDA C

Nizhni Novgorod

2014

Author: S.I. Bastrakov

OBJECTIVES

The objective of this lecture is to introduce basics of CUDA C programming language: function qualifiers, inline variables, as well as general concepts of CUDA such as thread, block, kernel, that allow creating simple CUDA programs.

ABSTRACT

This lecture considers basics of CUDA C programming language. We introduce general CUDA concepts of threads, blocks and kernels and present the corresponding syntax in CUDA C. Materials of this lecture are enough to develop simple CUDA applications.

BRIEF OVERVIEW

CUDA stands for Compute Unified Device Architecture, hardware and software platform for parallel general purpose computing on NVIDIA GPUs. It supports many programming interfaces and traditionally did not support other vendors, which is possible in the latest versions. CUDA development kit includes CUDA driver, CUDA Toolkit (compiler, profiler, optimized libraries, documentation), and GPU Computing SDK.

A program written 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.

Here are some 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 simplest case, but not the only possible way. All these examples are data parallel which is very well suited for all parallel programming technologies, including CUDA: each subset of data can be processed independently. In data parallelism situation all cores perform same processing on different data. In sequential C++ implementation there is a loop with independent iterations.

On GPUs 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. 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). For

example, if 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.

CUDA C programming language is extension of C/C++ that consists of function qualifiers, memory qualifiers, and inline variables. In this lecture we cover basic subset of CUDA C. We will use the following terminology: host is CPU, device is GPU, kernel is a function that is executed on GPU in parallel. There are three function qualifiers in CUDA C. Qualifier __host__ (by default) is a function called from host and executed on host. All usual C++ functions are host functions. Qualifier __global__ is a function called from host and executed in parallel on GPU (kernel). Qualifier __device__ is a function called inside kernel or other device function and executed on GPU.

From code on GPU side the following inline 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. The following relation holds: (0, 0, 0) ≤ blockIdx < gridDim, (0, 0, 0) ≤ threadIdx < blockDim. Inline variables are read only. Variable threadIdx gives local thread index inside block. There is no inline variable fro global thread index (among all blocks), but it can be computed. For 1D indexes global thread index is idx = blockIdx.x * blockDim.x + threadIdx.x.

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. 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.

We demonstrate a simple kernel to add two vectors and the code to invoke this 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];
}

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);
}
```

FOR STUDENTS

CUDA C programming language is described in [1, 2].

REFERENCES

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

INDIVIDUAL WORK

1. What are three main components of CUDA C extensions?
2. Enlist CUDA C keywords learned from this lecture and explain their meaning.
3. Write a kernel for matrix-vector multiplication and code to invoke it.