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

Practice 3. Vector addition

Nizhni Novgorod

2014

Author: S.I. Bastrakov

OBJECTIVES

The objective of this practice is to develop a simple implementation of vector addition and saxpy BLAS operation using CUDA, and address data alignment and workload distribution problems.

ABSTRACT

This practice is devoted to development a simple implementation of vector addition and saxpy BLAS operation using CUDA. We address data alignment and workload distribution problems.

BRIEF OVERVIEW

We consider a problem of addition of two single-precision floating-point vectors. This problem is obviously data parallel and thus allows simple implementation using CUDA. A CPU implementation is trivial:

```
void vecAdd(int n, float * x, float * y)
{
    for (int i = 0; i < n; ++i)
        y[i] += x[i];
}
```

In CUDA a loop with independent iterations is replaced by kernel where each thread computes its global index and computes the corresponding element of the result:

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

Note that we have to check that global index of a thread is inside vector, because in general case total number of threads can be greater than vector size due to requirement that all thread blocks have the same size. A function to invoke the kernel is as follows:

```
void vecAdd_gpu(int n, float * x, float * y)
{
    const int threads_per_block = 256;
    int num_blocks = (n + threads_per_block - 1) / threads_per_block;
    dim3 grid(num_blocks, 1, 1);
    dim3 blocks(threads_per_block, 1, 1);
    vecAdd_kernel<<<grid, blocks>>>(n, x, y);
}
```

We fix block size and compute number of blocks as upper integer part of ratio of vector size and block size. This explains, why did we have to check if (i < n) in the kernel. This problem is typical for GPU programming.

A more general implementation of vector addition is function axpy from BLAS level 1. It is defined as follows: $\forall i \in 0..n$: y[i * incy] ← y[i * incy] + a * x[i * incx], where incx and incy are constant parameters. For a = incx = incy = 1 axpy is a common vector addition. Implementations on CPU and GPU are generally similar to vector addition.

Implementation on CPU:

```
void saxpy(int n, float a, float * x, int incx, float * y, int incy)
{
    for (int i = 0; i < n; ++i)
        y[i * incy] += a * x[i * incx];
}
```

Again, the loop has independent iterations, thus can be easily transformed into a kernel:

```
__global__ void saxpy_kernel(int n, float a, float * x, int incx, float * y,
int incy)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        y[i * incy] += a * x[i * incx];
}
```

A function to invoke the kernel is similar to vector addition function.

FOR STUDENTS

CUDA C 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. Implement axpy operation for complex numbers.
2. Modify axpy kernel so that it will work for any amount of block and threads per block.