Introduction to CUDA and OpenCL

Ilona Tomkowicz, Zofia Pieńkowska

December 21, 2019

Contents

Contents 1

1 Goal of this exercise 2

2 Exercises 2&3: comparing computation time for C and C++ 2

3 Exercise 4: OpenCL elements and modifications made to the source code 3

4 Conclusions 8

1 Goal of this exercise

The subject of this exercise was getting familiar with another framework allowing parallel computing, OpenCL. The goal was to inspect and modify two simple programs with the functionality to add two vectors one to the other. Computations times for C and C++ have been compared. In the second part, a program was modified to not only add two random vectors to each other, but also add some values to those resultative vectors. The purpose of this operation was to make use of queuing the commands and chaining vectors in a simple, beginner-friendly way.

2 Exercises 2&3: comparing computation time for C and C++

A simple source code has been saved and executed. Its only functionality was adding two vectors and saving the result to the third using parallel computing. Number of vector elements has been increasing gradually and computing times have been noted. A similar comparison for Python could not be made due to lack of package `pyopencl`. The ratio of time to number of elements is illustrated on fig. 1.

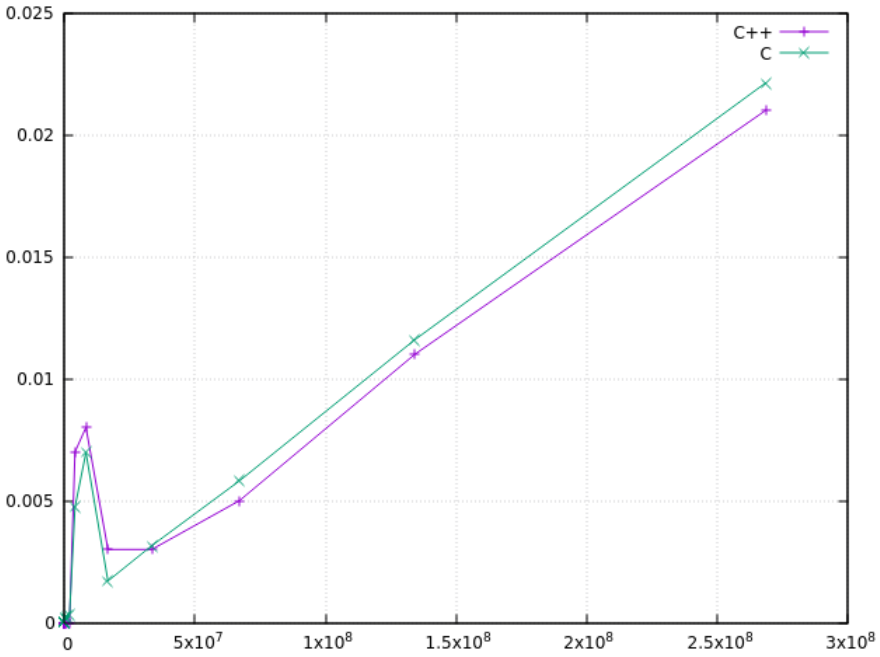

Figure 1: Time of computations for various number of elements using C and C++. The kernel did not start with the number of elements equal to 2^{29}.

3 Exercise 4: OpenCL elements and modifications made to the source code

In function `main` the first thing to do is to allocate proper amount of memory on host device. Earlier, the program was adding two vectors, A and B, and saving the resulting vector C. Now, we want to compute the following vectors as well: D=C+E and F=D+G, where E and G are filled with random floats. For this reason, the final code looks as follows:

```
    int    err;
float*    h_a = (float*) calloc(LENGTH, sizeof(float));
float*    h_b = (float*) calloc(LENGTH, sizeof(float));
float*    h_c = (float*) calloc(LENGTH, sizeof(float));
float*    h_d = (float*) calloc(LENGTH, sizeof(float));
float*    h_e = (float*) calloc(LENGTH, sizeof(float));
```

```
float*    h_f = (float*) calloc(LENGTH, sizeof(float));
float*    h_g = (float*) calloc(LENGTH, sizeof(float));
```

In the next step, device memory has been allocated for data and some of the vectors have been filled with random float numbers:

```
cl_mem d_a;
cl_mem d_b;
cl_mem d_c;
cl_mem d_d;
cl_mem d_e;
cl_mem d_f;
cl_mem d_g;

// Fill vectors a, b, e and g with random float values
int i = 0;
int count = LENGTH;
for(i = 0; i < count; i++){
    h_a[i] = rand() / (float)RAND_MAX;
    h_b[i] = rand() / (float)RAND_MAX;
    h_e[i] = rand() / (float)RAND_MAX;
    h_g[i] = rand() / (float)RAND_MAX;
```

Because in OpenCL a *device* can be any unit that is able to compute, a platform to work on needs to be defined, which is done in the next fragment of code. This fragment has been shipped with the exercise source code and is omitted here. The next step is to define **context**, **queue** and **program**. The context is the widest concept and can be understood as an environment in which kernels are executed and synchronisation is defined. The context contains all the devices we use and their memory. A queue is an ordered sequence of commands for a device. Although a device can execute many queues, one queue can only point to one device. The queue is also part of the context. Finally, the program contains the context, the kernel and optional list of target devices and build options.

```
// Create a compute context
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
checkError(err, "Creating context");
```

```
// Create a command queue
commands = clCreateCommandQueue(context, device_id, 0, &err);
checkError(err, "Creating command queue");

// Create the compute program from the source buffer
program = clCreateProgramWithSource(context, 1, (const char **)
& KernelSource, NULL, &err);
checkError(err, "Creating program");
```

Then, the kernel is created arrays corresponding with vectors are created in device memory. Some of the buffers are read-only, as the are filled with random numbers, while others are possible to read and write to, because they are a result of computation and will also be used in later computations. Only the last buffer, corresponding to vector F, is write-only, because it's the final result of the computations. In case of the read-only vectors, the pointers also have to be copied.

```
ko_vadd = clCreateKernel(program, "vadd", &err);
checkError(err, "Creating kernel");

d_a  = clCreateBuffer(context,  CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(float) * count, h_a, &err);
checkError(err, "Creating buffer d_a");

d_b  = clCreateBuffer(context,  CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(float) * count, h_b, &err);
checkError(err, "Creating buffer d_b");

d_c  = clCreateBuffer(context,  CL_MEM_READ_WRITE,
sizeof(float) * count, NULL, &err);
checkError(err, "Creating buffer d_c");

d_d  = clCreateBuffer(context,  CL_MEM_READ_WRITE,
sizeof(float) * count, NULL, &err);
checkError(err, "Creating buffer d_d");

d_e  = clCreateBuffer(context,  CL_MEM_READ_ONLY |
```

```
CL_MEM_COPY_HOST_PTR, sizeof(float) * count, h_e, &err);
checkError(err, "Creating buffer d_e");

d_f = clCreateBuffer(context, CL_MEM_READ_WRITE,
sizeof(float) * count, NULL, &err);
checkError(err, "Creating buffer d_f");

d_g = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(float) * count, h_g, &err);
checkError(err, "Creating buffer d_g");
```

Then, the kernel is enqueued and executed three times in order to complete computations. Finally, the code is tested.

```
// Enqueue the kernel 1/3
err = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_a);
err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_b);
err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_c);
err |= clSetKernelArg(ko_vadd, 3, sizeof(unsigned int), count);
checkError(err, "Setting kernel arguments");

global = count;
err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global,
NULL, 0, NULL, NULL);
checkError(err, "Enqueueing kernel (1/3)");

// Enqueue the kernel 2/3
err = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_e);
err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_c);
err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_d);
checkError(err, "Setting kernel arguments");

err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global,
NULL, 0, NULL, NULL);
checkError(err, "Enqueueing kernel (2/3)");

// Enqueue the kernel 3/3
err = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_g);
```

```
err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_d);
err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_f);
checkError(err, "Setting kernel arguments");

err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global,
NULL, 0, NULL, NULL);
checkError(err, "Enqueueing kernel (3/3)");

err = clEnqueueReadBuffer( commands, d_f, CL_TRUE, 0,
sizeof(float) * count, h_f, 0, NULL, NULL );
if (err!=CL_SUCCESS) {
    printf("Error: Failed to read output array!\n%s\n", err_code(err));
    exit(1);
}

// Test the results
correct = 0;
float tmp;

for(i = 0; i < count; i++)
{
        tmp = h_a[i] + h_b[i] + h_e[i] + h_g[i];
    tmp -= h_f[i];
    if(tmp*tmp < TOL*TOL)
        correct++;
    else {
        printf("Error occured!");
    }
}

// summarize results
printf("C = A+B+E+G:  %d out of %d results were correct.\n",
correct, count);
```

With listed modifications, the code computes all the vectors properly, resulting in following message:

```
Device is GeForce GTX 1060 6GB GPU from NVIDIA Corporation
with a max of 10 compute units
C = A+B+E+G: 128 out of 128 results were correct.
```

4 Conclusions

As for exercises 2 and 3, it can be observed that the computation time is rather similar for C and C++. There's a little advantage for C++ for number of elements smaller than 2^{25}, but for a larger number of elements C seems to be faster.

As for exercise 4, changes to the code included queuing the commands and executing the kernel multiply. Those changes have been implemented successfully, allowing the user to add the vectors which have been a sum of other vectors themselves.