# REPORT 4

## Introduction to CUDA and OpenCL

AGH-Faculty of Physics and Applied Computer Science

authors:

**Kinga Pyrek**

**Aleksandra Rolka**

On our last lab classes we were debating about OpenCL which is a framework for writing programs that execute across heterogeneous platforms consisting of GPU, CPU and others processors of hardware accelerators. We can spot a lot of similarities between CUDA and OpenCL like: both are based on C language, same type of parallelization and hierarchical structure of program. OpenCL can be very profitable for us and decrease the execution time.

You can easily notice that OpenCL has an approachable environment. Next thing we did was looking over different version of “vadd” code in C and C++, which were adding two vectors and we discussed the differences between them. Afterwards we had a task to edit the “vadd” code. The purpose of program was to create a vector C by adding vectors A and B, then create vectors D = C + E and F = D + G. We did it in C and C++ version. Here is the C++ code:

```cpp
//
// Name:        vadd_cpp.cpp
//
// Purpose:     Elementwise addition of  vectors (c = a + b) (d=c+e) (f=d+g)
//
// HISTORY:     Written by Tim Mattson, June 2011
//              Ported to C++ Wrapper API by Benedict Gaster, September 2011
//              Updated to C++ Wrapper API v1.2 by Tom Deakin and Simon McIntosh-Smith,
//              October 2012
//              Updated to C++ Wrapper v1.2.6 by Tom Deakin, August 2013
//
//------------------------------------------------------------------------------

#define __CL_ENABLE_EXCEPTIONS

#include "cl.hpp"

#include "util.hpp" // utility library

#include <vector>
#include <cstdio>
#include <cstdlib>
#include <string>

#include <iostream>
#include <fstream>

// pick up device type from compiler command line or from the default type
#ifndef DEVICE
#define DEVICE CL_DEVICE_TYPE_DEFAULT
#endif

#include <err_code.h>

//------------------------------------------------------------------------------

#define TOL    (0.001)   // tolerance used in floating point comparisons
#define LENGTH (1024)    // length of vectors a, b, c, d, e, f and g

int main(void)
{
    std::vector<float> h_a(LENGTH);                // a vector
    std::vector<float> h_b(LENGTH);                // b vector
    std::vector<float> h_c(LENGTH, 0xdeadbeef);    // c = a + b, from compute device
    std::vector<float> h_e(LENGTH);                // a vector
    std::vector<float> h_g(LENGTH);                // b vector
    std::vector<float> h_d(LENGTH);                // d = c + e, from compute device
    std::vector<float> h_f(LENGTH);                // f = d + g, from compute device

    cl::Buffer d_a;                        // device memory used for the input  a vector
    cl::Buffer d_b;                        // device memory used for the input  b vector
    cl::Buffer d_c;                       // device memory used for the output c vector
    cl::Buffer d_d;                        // device memory used for the input  d vector
    cl::Buffer d_e;                        // device memory used for the input  e vector
    cl::Buffer d_f;                       // device memory used for the output f vector
    cl::Buffer d_g;                       // device memory used for the output g vector

    // Fill vectors a and b with random float values
    int count = LENGTH;
    for(int 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;
    }

    try
    {
        // Create a context
        cl::Context context(DEVICE);

        // Load in kernel source, creating a program object for the context

        cl::Program program(context, util::loadProgram("vadd.cl"), true);

        // Get the command queue
        cl::CommandQueue queue(context);

        // Create the kernel functor
```

```
    cl::make_kernel<cl::Buffer, cl::Buffer, cl::Buffer, int> vadd(program, "vadd");

    d_a   = cl::Buffer(context, h_a.begin(), h_a.end(), true);
    d_b   = cl::Buffer(context, h_b.begin(), h_b.end(), true);
    d_e   = cl::Buffer(context, h_e.begin(), h_e.end(), true);
    d_g   = cl::Buffer(context, h_g.begin(), h_g.end(), true);

    d_c  = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * LENGTH);
    d_d  = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * LENGTH);
    d_f  = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * LENGTH);

    util::Timer timer;

    vadd(
        cl::EnqueueArgs(
            queue,
            cl::NDRange(count)),
        d_a,
        d_b,
        d_c,
        d_d,
        d_e,
        d_f,
        d_g,

        count);

    queue.finish();

    double rtime = static_cast<double>(timer.getTimeMilliseconds()) / 1000.0;
    printf("\nThe kernels ran in %lf seconds\n", rtime);

    cl::copy(queue, d_c, h_c.begin(), h_c.end());
    cl::copy(queue, d_d, h_d.begin(), h_d.end());
    cl::copy(queue, d_f, h_f.begin(), h_f.end());

    // Test the results
    int correct = 0;
    float tmp;
    for(int i = 0; i < count; i++) {
        tmp = h_a[i] + h_b[i]; // expected value for d_c[i]
        tmp -= h_c[i];                     // compute errors
        if(tmp*tmp < TOL*TOL) {      // correct if square deviation is less
            correct++;                        //  than tolerance squared
        }
        else {

            printf(
                " tmp %f h_a %f h_b %f  h_c %f \n",
                tmp,
                h_a[i],
                h_b[i],
                h_c[i]);
        }
    }

    int correct1 = 0;
    float tmp1;
    for(int i = 0; i < count; i++) {
        tmp1 = h_c[i] + h_e[i]; // expected value for d_c[i]
        tmp1 -= h_d[i];                     // compute errors
        if(tmp1*tmp1 < TOL*TOL) {      // correct if square deviation is less
            correct1++;                        //  than tolerance squared
        }
        else {

            printf(
                " tmp1 %f h_c %f h_e %f  h_d %f \n",
                tmp1,
                h_c[i],
                h_e[i],
                h_d[i]);
        }
    }
```

```
        int correct2 = 0;
        float tmp2;
        for(int i = 0; i < count; i++) {
            tmp2 = h_d[i] + h_g[i]; // expected value for d_c[i]
            tmp2 -= h_f[i];                       // compute errors
            if(tmp2*tmp2 < TOL*TOL) {       // correct if square deviation is less
                correct2++;                          //  than tolerance squared
            }
            else {

                printf(
                    " tmp2 %f h_d %f h_g %f  h_f %f \n",
                    tmp2,
                    h_d[i],
                    h_g[i],
                    h_f[i]);
            }
        }
        // summarize results vector C
        printf(
            "vector add to find C = A+B:  %d out of %d results were correct.\n",
            correct,
            count);

        // summarize results vector D
        printf(
            "vector add to find D = C+E:  %d out of %d results were correct.\n",
            correct1,
            count);

        // summarize results vector F
        printf(
            "vector add to find F = D+G:  %d out of %d results were correct.\n",
            correct2,
            count);

    }
    catch (cl::Error err) {
        std::cout << "Exception\n";
        std::cerr
            << "ERROR: "
            << err.what()
            << "("
            << err_code(err.err())
           << ")"
           << std::endl;
    }
}
```

## And this is C version:

```
//------------------------------------------------------------------------------
//
// Name:       vadd.c
//
// Purpose:    Elementwise addition of  vectors (c = a + b) (d=c+e) (f=d+g)
//
// HISTORY:    Written by Tim Mattson, December 2009
//             Updated by Tom Deakin and Simon McIntosh-Smith, October 2012
//             Updated by Tom Deakin, July 2013
//             Updated by Tom Deakin, October 2014
//
//------------------------------------------------------------------------------

```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#ifdef __APPLE__
#include <OpenCL/opencl.h>
#include <unistd.h>
#else
#include <CL/cl.h>
#endif

#include "err_code.h"

//pick up device type from compiler command line or from
//the default type
#ifndef DEVICE
#define DEVICE CL_DEVICE_TYPE_DEFAULT
#endif

extern double wtime();       // returns time since some fixed past point (wtime.c)
extern int output_device_info(cl_device_id );

//------------------------------------------------------------------------------

#define TOL    (0.001)   // tolerance used in floating point comparisons
#define LENGTH (1024)    // length of vectors a, b, c, d, f and g

//------------------------------------------------------------------------------

const char *KernelSource = "\n" \
"__kernel void vadd(                                                 \n" \
"   __global float* a,                                                  \n" \
"   __global float* b,                                                  \n" \
"   __global float* c,                                                  \n" \
"   const unsigned int count)                                           \n" \
"{                                                                      \n" \
"   int i = get_global_id(0);                                           \n" \
"   if(i < count)                                                       \n" \
"       c[i] = a[i] + b[i];                                             \n" \
"}                                                                      \n" \
"\n";

//------------------------------------------------------------------------------

int main(int argc, char** argv)
{
    int          err;               // error code returned from OpenCL calls

    float*       h_a = (float*) calloc(LENGTH, sizeof(float));       // a vector
    float*       h_b = (float*) calloc(LENGTH, sizeof(float));       // b vector
    float*       h_c = (float*) calloc(LENGTH, sizeof(float));       // c vector (a+b)
    float*       h_d = (float*) calloc(LENGTH, sizeof(float));       // d vector (c+e)
    float*       h_e = (float*) calloc(LENGTH, sizeof(float));       // e vector
    float*       h_f = (float*) calloc(LENGTH, sizeof(float));       // f vector (d+g)
    float*       h_g = (float*) calloc(LENGTH, sizeof(float));       // g vector

    unsigned int correct;           // number of correct results

    size_t global;                  // global domain size

    cl_device_id     device_id;     // compute device id
    cl_context       context;       // compute context
    cl_command_queue commands;      // compute command queue
    cl_program       program;       // compute program
    cl_kernel        ko_vadd;       // compute kernel

    cl_mem d_a;                     // device memory used for the input  a vector
    cl_mem d_b;                     // device memory used for the input  b vector
    cl_mem d_c;                     // device memory used for the output c vector
    cl_mem d_d;                     // device memory used for the input  d vector
    cl_mem d_e;                     // device memory used for the input  e vector
    cl_mem d_f;                     // device memory used for the output f vector
    cl_mem d_g;                     // device memory used for the output g vector

```

```c
// Fill vectors a and b 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;
}

// Set up platform and GPU device

cl_uint numPlatforms;

// Find number of platforms
err = clGetPlatformIDs(0, NULL, &numPlatforms);
checkError(err, "Finding platforms");
if (numPlatforms == 0)
{
    printf("Found 0 platforms!\n");
    return EXIT_FAILURE;
}

// Get all platforms
cl_platform_id Platform[numPlatforms];
err = clGetPlatformIDs(numPlatforms, Platform, NULL);
checkError(err, "Getting platforms");

// Secure a GPU
for (i = 0; i < numPlatforms; i++)
{
    err = clGetDeviceIDs(Platform[i], DEVICE, 1, &device_id, NULL);
    if (err == CL_SUCCESS)
    {
        break;
    }
}

if (device_id == NULL)
    checkError(err, "Finding a device");

err = output_device_info(device_id);
checkError(err, "Printing device output");

// 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");

// Build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS)
{
    size_t len;
    char buffer[2048];

    printf("Error: Failed to build program executable!\n%s\n", err_code(err));
    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
    printf("%s\n", buffer);
    return EXIT_FAILURE;
}

// Create the compute kernel from the program
ko_vadd = clCreateKernel(program, "vadd", &err);
checkError(err, "Creating kernel");
```

```
    // Create the input (a, b) and output (c) arrays in device memory
    d_a  = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_a");

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

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

    d_g  = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_b");


    // Write a and b vectors into compute device memory
    err = clEnqueueWriteBuffer(commands, d_a, CL_TRUE, 0, sizeof(float) * count, h_a, 0, NULL, NULL);
    checkError(err, "Copying h_a to device at d_a");

    err = clEnqueueWriteBuffer(commands, d_b, CL_TRUE, 0, sizeof(float) * count, h_b, 0, NULL, NULL);
    checkError(err, "Copying h_b to device at d_b");

    err = clEnqueueWriteBuffer(commands, d_e, CL_TRUE, 0, sizeof(float) * count, h_e, 0, NULL, NULL);
    checkError(err, "Copying h_e to device at d_g");

    err = clEnqueueWriteBuffer(commands, d_g, CL_TRUE, 0, sizeof(float) * count, h_g, 0, NULL, NULL);
    checkError(err, "Copying h_g to device at d_g");

    // Set the arguments to our compute kernel   (a,b,c)
    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");

    double rtime = wtime();

    // Execute the kernel over the entire range of our 1d input data set
    // letting the OpenCL runtime choose the work-group size
    global = count;
    err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global, NULL, 0, NULL, NULL);
    checkError(err, "Enqueueing kernel");

    // Wait for the commands to complete before stopping the timer
    err = clFinish(commands);
    checkError(err, "Waiting for kernel to finish");

    rtime = wtime() - rtime;
    printf("\nThe addition's C=A+B kernel ran in %lf seconds\n",rtime);

    // Read back the results from the compute device
    err = clEnqueueReadBuffer( commands, d_c, CL_TRUE, 0, sizeof(float) * count, h_c, 0, NULL, NULL );
    if (err != CL_SUCCESS)
    {
        printf("Error: Failed to read output array!\n%s\n", err_code(err));
        exit(1);
    }


    // Set the arguments to our compute kernel    (c,e,d)
    err  = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_c);
    err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_e);
    err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_d);
    err |= clSetKernelArg(ko_vadd, 3, sizeof(unsigned int), &count);
    checkError(err, "Setting kernel arguments");

    double rtime = wtime();

    // Execute the kernel over the entire range of our 1d input data set
    // letting the OpenCL runtime choose the work-group size
    global = count;
    err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global, NULL, 0, NULL, NULL);
    checkError(err, "Enqueueing kernel");

    // Wait for the commands to complete before stopping the timer
    err = clFinish(commands);
    checkError(err, "Waiting for kernel to finish");

```

```c
    rtime = wtime() - rtime;
    printf("\nThe addition's D=C+E kernel ran in %lf seconds\n",rtime);

    // Read back the results from the compute device
    err = clEnqueueReadBuffer( commands, d_d, CL_TRUE, 0, sizeof(float) * count, h_d, 0, NULL, NULL );
    if (err != CL_SUCCESS)
    {
        printf("Error: Failed to read output array!\n%s\n", err_code(err));
        exit(1);
    }

    // Set the arguments to our compute kernel    (d,f,g)
    err  = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_d);
    err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_g);
    err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_f);
    err |= clSetKernelArg(ko_vadd, 3, sizeof(unsigned int), &count);
    checkError(err, "Setting kernel arguments");

    double rtime = wtime();

    // Execute the kernel over the entire range of our 1d input data set
    // letting the OpenCL runtime choose the work-group size
    global = count;
    err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global, NULL, 0, NULL, NULL);
    checkError(err, "Enqueueing kernel");

    // Wait for the commands to complete before stopping the timer
    err = clFinish(commands);
    checkError(err, "Waiting for kernel to finish");

    rtime = wtime() - rtime;
    printf("\nThe addition's F=D+G kernel ran in %lf seconds\n",rtime);

    // Read back the results from the compute device
    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 (a,b,c)
    correct = 0;
    float tmp;

    for(i = 0; i < count; i++)
    {
        tmp = h_a[i] + h_b[i];      // assign element i of a+b to tmp
        tmp -= h_c[i];              // compute deviation of expected and output result
        if(tmp*tmp < TOL*TOL)         // correct if square deviation is less than tolerance squared
            correct++;
        else {
            printf(" tmp %f h_a %f h_b %f h_c %f \n",tmp, h_a[i], h_b[i], h_c[i]);
        }
    }

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

    // Test the results (c,e,d)
    correct1 = 0;
    float tmp1;

    for(i = 0; i < count; i++)
    {
        tmp1 = h_c[i] + h_e[i];     // assign element i of a+b to tmp
        tmp1 -= h_d[i];             // compute deviation of expected and output result
        if(tmp1*tmp1 < TOL*TOL)        // correct if square deviation is less than tolerance squared
            correct1++;
        else {
            printf(" tmp1 %f h_c %f h_e %f h_d %f \n",tmp1, h_c[i], h_e[i], h_d[i]);
        }
    }

    // summarise results
    printf("D = C+E:  %d out of %d results were correct.\n", correct, count);

    // Test the results (d,g,f)
    correct2 = 0;
    float tmp2;
```

```
    for(i = 0; i < count; i++)
    {
        tmp2 = h_d[i] + h_g[i];     // assign element i of a+b to tmp
        tmp2 -= h_f[i];             // compute deviation of expected and output result
        if(tmp2*tmp2 < TOL*TOL)        // correct if square deviation is less than tolerance squared
            correct2++;
        else {
            printf(" tmp2 %f h_d %f h_g %f h_f %f \n",tmp2, h_d[i], h_g[i], h_f[i]);
        }
    }

    // summarise results
    printf("F = D+G:  %d out of %d results were correct.\n", correct, count);

    // cleanup then shutdown
    clReleaseMemObject(d_a);
    clReleaseMemObject(d_b);
    clReleaseMemObject(d_c);
    clReleaseMemObject(d_d);
    clReleaseMemObject(d_e);
    clReleaseMemObject(d_f);
    clReleaseMemObject(d_g);
```


```
    // summarise results
    printf("F = D+G:  %d out of %d results were correct.\n", correct, count);

    // cleanup then shutdown
    clReleaseMemObject(d_a);
    clReleaseMemObject(d_b);
    clReleaseMemObject(d_c);
    clReleaseMemObject(d_d);
    clReleaseMemObject(d_e);
    clReleaseMemObject(d_f);
    clReleaseMemObject(d_g);
```


```

    clReleaseProgram(program);
    clReleaseKernel(ko_vadd);
    clReleaseCommandQueue(commands);
    clReleaseContext(context);
```


```

    clReleaseProgram(program);
    clReleaseKernel(ko_vadd);
    clReleaseCommandQueue(commands);
    clReleaseContext(context);
```


```
    free(h_a);
    free(h_b);
    free(h_c);
    free(h_d);
    free(h_e);
    free(h_f);
    free(h_g);

    return 0;
}
```

Now let’s take a look at differences. First of all, there is a significant dissimilarity in the length of code. We can see that C++ code is much more compact. Also we don’t have control errors functions in C++ version, because of “cl.hpp” library, which contains really helpful functions. Thanks to that our code is clearer and more friendly for users. It’s also worth to mention that we didn’t notice much difference in time performance between C and C++ code.