

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:

```

1 //
2 // Name:      vadd_cpp.cpp
3 //
4 // Purpose:    Elementwise addition of  vectors (c = a + b) (d=c+e) (f=d+g)
5 //
6 // HISTORY:    Written by Tim Mattson, June 2011
7 //            Ported to C++ Wrapper API by Benedict Gaster, September 2011
8 //            Updated to C++ Wrapper API v1.2 by Tom Deakin and Simon McIntosh-Smith,
9 //            October 2012
10 //            Updated to C++ Wrapper v1.2.6 by Tom Deakin, August 2013
11 //
12 //-----
13
14 #define __CL_ENABLE_EXCEPTIONS
15
16 #include "cl.hpp"
17
18 #include "util.hpp" // utility library
19
20 #include <vector>
21 #include <cstdio>
22 #include <cstdlib>
23 #include <string>
24
25 #include <iostream>
26 #include <fstream>
27
28 // pick up device type from compiler command line or from the default type
29 #ifndef DEVICE
30 #define DEVICE CL_DEVICE_TYPE_DEFAULT
31 #endif
32
33 #include <err_code.h>
34
35 //-----
36
37 #define TOL      (0.001) // tolerance used in floating point comparisons
38 #define LENGTH  (1024)  // length of vectors a, b, c, d, e, f and g
39
40 int main(void)
41 {
42     std::vector<float> h_a(LENGTH); // a vector
43     std::vector<float> h_b(LENGTH); // b vector
44     std::vector<float> h_c(LENGTH, 0xdeadbeef); // c = a + b, from compute device
45     std::vector<float> h_e(LENGTH); // a vector
46     std::vector<float> h_g(LENGTH); // b vector
47     std::vector<float> h_d(LENGTH); // d = c + e, from compute device
48     std::vector<float> h_f(LENGTH); // f = d + g, from compute device
49
50     cl::Buffer d_a; // device memory used for the input  a vector
51     cl::Buffer d_b; // device memory used for the input  b vector
52     cl::Buffer d_c; // device memory used for the output c vector
53     cl::Buffer d_d; // device memory used for the input  d vector
54     cl::Buffer d_e; // device memory used for the input  e vector
55     cl::Buffer d_f; // device memory used for the output f vector
56     cl::Buffer d_g; // device memory used for the output g vector
57
58     // Fill vectors a and b with random float values
59     int count = LENGTH;
60     for(int i = 0; i < count; i++)
61     {
62         h_a[i] = rand() / (float)RAND_MAX;
63         h_b[i] = rand() / (float)RAND_MAX;
64         h_e[i] = rand() / (float)RAND_MAX;
65         h_g[i] = rand() / (float)RAND_MAX;
66     }
67
68     try
69     {
70         // Create a context
71         cl::Context context(DEVICE);
72
73         // Load in kernel source, creating a program object for the context
74
75         cl::Program program(context, util::loadProgram("vadd.cl"), true);
76
77         // Get the command queue
78         cl::CommandQueue queue(context);
79
80         // Create the kernel functor

```

```

81
82     cl::make_kernel<cl::Buffer, cl::Buffer, cl::Buffer, int> vadd(program, "vadd");
83
84     d_a = cl::Buffer(context, h_a.begin(), h_a.end(), true);
85     d_b = cl::Buffer(context, h_b.begin(), h_b.end(), true);
86     d_e = cl::Buffer(context, h_e.begin(), h_e.end(), true);
87     d_g = cl::Buffer(context, h_g.begin(), h_g.end(), true);
88
89     d_c = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * LENGTH);
90     d_d = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * LENGTH);
91     d_f = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * LENGTH);
92
93     util::Timer timer;
94
95     vadd(
96         cl::EnqueueArgs(
97             queue,
98             cl::NDRange(count)),
99         d_a,
100        d_b,
101        d_c,
102        d_d,
103        d_e,
104        d_f,
105        d_g,
106
107        count);
108
109     queue.finish();
110
111     double rtime = static_cast<double>(timer.getTimeMilliseconds()) / 1000.0;
112     printf("\nThe kernels ran in %lf seconds\n", rtime);
113
114     cl::copy(queue, d_c, h_c.begin(), h_c.end());
115     cl::copy(queue, d_d, h_d.begin(), h_d.end());
116     cl::copy(queue, d_f, h_f.begin(), h_f.end());
117
118     // Test the results
119     int correct = 0;
120     float tmp;
121     for(int i = 0; i < count; i++) {
122         tmp = h_a[i] + h_b[i]; // expected value for d_c[i]
123         tmp -= h_c[i];         // compute errors
124         if(tmp*tmp < TOL*TOL) { // correct if square deviation is less
125             correct++;          // than tolerance squared
126         }
127         else {
128             printf(
129                 " tmp %f h_a %f h_b %f h_c %f \n",
130                 tmp,
131                 h_a[i],
132                 h_b[i],
133                 h_c[i]);
134         }
135     }
136
137     int correct1 = 0;
138     float tmp1;
139     for(int i = 0; i < count; i++) {
140         tmp1 = h_c[i] + h_e[i]; // expected value for d_c[i]
141         tmp1 -= h_d[i];         // compute errors
142         if(tmp1*tmp1 < TOL*TOL) { // correct if square deviation is less
143             correct1++;          // than tolerance squared
144         }
145         else {
146             printf(
147                 " tmp1 %f h_c %f h_e %f h_d %f \n",
148                 tmp1,
149                 h_c[i],
150                 h_e[i],
151                 h_d[i]);
152         }
153     }
154
155 }
156

```

```

157     int correct2 = 0;
158     float tmp2;
159     for(int i = 0; i < count; i++) {
160         tmp2 = h_d[i] + h_g[i]; // expected value for d_c[i]
161         tmp2 -= h_f[i];         // compute errors
162         if(tmp2*tmp2 < TOL*TOL) { // correct if square deviation is less
163             correct2++;          // than tolerance squared
164         }
165         else {
166             printf(
167                 " tmp2 %f h_d %f h_g %f h_f %f \n",
168                 tmp2,
169                 h_d[i],
170                 h_g[i],
171                 h_f[i]);
172             }
173     }
174
175     // summarize results vector C
176     printf(
177         "vector add to find C = A+B:  %d out of %d results were correct.\n",
178         correct,
179         count);
180
181     // summarize results vector D
182     printf(
183         "vector add to find D = C+E:  %d out of %d results were correct.\n",
184         correct1,
185         count);
186
187     // summarize results vector F
188     printf(
189         "vector add to find F = D+G:  %d out of %d results were correct.\n",
190         correct2,
191         count);
192
193 }
194 catch (cl::Error err) {
195     std::cout << "Exception\n";
196     std::cerr
197         << "ERROR: "
198         << err.what()
199         << "("
200         << err_code(err.err())
201         << ")"
202         << std::endl;
203 }
204 }

```

And this is C version:

```

1  //-----
2  //
3  // Name:      vadd.c
4  //
5  // Purpose:   Elementwise addition of  vectors (c = a + b) (d=c+e) (f=d+g)
6  //
7  // HISTORY:   Written by Tim Mattson, December 2009
8  //           Updated by Tom Deakin and Simon McIntosh-Smith, October 2012
9  //           Updated by Tom Deakin, July 2013
10 //           Updated by Tom Deakin, October 2014
11 //
12 //-----
13

```

```

14
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <sys/types.h>
18 #ifdef __APPLE__
19 #include <OpenCL/opencl.h>
20 #include <unistd.h>
21 #else
22 #include <CL/cl.h>
23 #endif
24
25 #include "err_code.h"
26
27 //pick up device type from compiler command line or from
28 //the default type
29 #ifndef DEVICE
30 #define DEVICE CL_DEVICE_TYPE_DEFAULT
31 #endif
32
33 extern double wtime(); // returns time since some fixed past point (wtime.c)
34 extern int output_device_info(cl_device_id );
35
36 //-----
37
38 #define TOL (0.001) // tolerance used in floating point comparisons
39 #define LENGTH (1024) // length of vectors a, b, c, d, f and g
40
41 //-----
42
43 const char *KernelSource = "\n" \
44 "    kernel void vadd(                                \n" \
45 "        __global float* a,                          \n" \
46 "        __global float* b,                          \n" \
47 "        __global float* c,                          \n" \
48 "        const unsigned int count)                  \n" \
49 "{                                                    \n" \
50 "    int i = get_global_id(0);                      \n" \
51 "    if(i < count)                                  \n" \
52 "        c[i] = a[i] + b[i];                        \n" \
53 "    }                                                \n" \
54 "\n";
55
56 //-----
57
58
59 int main(int argc, char** argv)
60 {
61     int err; // error code returned from OpenCL calls
62
63     float* h_a = (float*) calloc(LENGTH, sizeof(float)); // a vector
64     float* h_b = (float*) calloc(LENGTH, sizeof(float)); // b vector
65     float* h_c = (float*) calloc(LENGTH, sizeof(float)); // c vector (a+b)
66     float* h_d = (float*) calloc(LENGTH, sizeof(float)); // d vector (c+e)
67     float* h_e = (float*) calloc(LENGTH, sizeof(float)); // e vector
68     float* h_f = (float*) calloc(LENGTH, sizeof(float)); // f vector (d+g)
69     float* h_g = (float*) calloc(LENGTH, sizeof(float)); // g vector
70
71     unsigned int correct; // number of correct results
72
73     size_t global; // global domain size
74
75     cl_device_id device_id; // compute device id
76     cl_context context; // compute context
77     cl_command_queue commands; // compute command queue
78     cl_program program; // compute program
79     cl_kernel ko_vadd; // compute kernel
80
81     cl_mem d_a; // device memory used for the input a vector
82     cl_mem d_b; // device memory used for the input b vector
83     cl_mem d_c; // device memory used for the output c vector
84     cl_mem d_d; // device memory used for the input d vector
85     cl_mem d_e; // device memory used for the input e vector
86     cl_mem d_f; // device memory used for the output f vector
87     cl_mem d_g; // device memory used for the output g vector
88
89

```

```

90 // Fill vectors a and b with random float values
91 int i = 0;
92 int count = LENGTH;
93 for(i = 0; i < count; i++){
94     h_a[i] = rand() / (float)RAND_MAX;
95     h_b[i] = rand() / (float)RAND_MAX;
96     h_e[i] = rand() / (float)RAND_MAX;
97     h_g[i] = rand() / (float)RAND_MAX;
98 }
99
100 // Set up platform and GPU device
101
102 cl_uint numPlatforms;
103
104 // Find number of platforms
105 err = clGetPlatformIDs(0, NULL, &numPlatforms);
106 checkError(err, "Finding platforms");
107 if (numPlatforms == 0)
108 {
109     printf("Found 0 platforms!\n");
110     return EXIT_FAILURE;
111 }
112
113 // Get all platforms
114 cl_platform_id Platform[numPlatforms];
115 err = clGetPlatformIDs(numPlatforms, Platform, NULL);
116 checkError(err, "Getting platforms");
117
118 // Secure a GPU
119 for (i = 0; i < numPlatforms; i++)
120 {
121     err = clGetDeviceIDs(Platform[i], DEVICE, 1, &device_id, NULL);
122     if (err == CL_SUCCESS)
123     {
124         break;
125     }
126 }
127
128 if (device_id == NULL)
129     checkError(err, "Finding a device");
130
131 err = output_device_info(device_id);
132 checkError(err, "Printing device output");
133
134 // Create a compute context
135 context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
136 checkError(err, "Creating context");
137
138 // Create a command queue
139 commands = clCreateCommandQueue(context, device_id, 0, &err);
140 checkError(err, "Creating command queue");
141
142 // Create the compute program from the source buffer
143 program = clCreateProgramWithSource(context, 1, (const char **) & KernelSource, NULL, &err);
144 checkError(err, "Creating program");
145
146 // Build the program
147 err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
148 if (err != CL_SUCCESS)
149 {
150     size_t len;
151     char buffer[2048];
152
153     printf("Error: Failed to build program executable!\n%s\n", err_code(err));
154     clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
155     printf("%s\n", buffer);
156     return EXIT_FAILURE;
157 }
158
159 // Create the compute kernel from the program
160 ko_vadd = clCreateKernel(program, "vadd", &err);
161 checkError(err, "Creating kernel");

```

```

162
163 // Create the input (a, b) and output (c) arrays in device memory
164 d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * count, NULL, &err);
165 checkError(err, "Creating buffer d_a");
166
167 d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * count, NULL, &err);
168 checkError(err, "Creating buffer d_b");
169
170 d_c = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * count, NULL, &err);
171 checkError(err, "Creating buffer d_c");
172
173 d_d = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * count, NULL, &err);
174 checkError(err, "Creating buffer d_d");
175
176 d_e = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * count, NULL, &err);
177 checkError(err, "Creating buffer d_e");
178
179 d_f = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * count, NULL, &err);
180 checkError(err, "Creating buffer d_f");
181
182 d_g = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * count, NULL, &err);
183 checkError(err, "Creating buffer d_g");
184
185
186 // Write a and b vectors into compute device memory
187 err = clEnqueueWriteBuffer(commands, d_a, CL_TRUE, 0, sizeof(float) * count, h_a, 0, NULL, NULL);
188 checkError(err, "Copying h_a to device at d_a");
189
190 err = clEnqueueWriteBuffer(commands, d_b, CL_TRUE, 0, sizeof(float) * count, h_b, 0, NULL, NULL);
191 checkError(err, "Copying h_b to device at d_b");
192
193 err = clEnqueueWriteBuffer(commands, d_e, CL_TRUE, 0, sizeof(float) * count, h_e, 0, NULL, NULL);
194 checkError(err, "Copying h_e to device at d_g");
195
196 err = clEnqueueWriteBuffer(commands, d_g, CL_TRUE, 0, sizeof(float) * count, h_g, 0, NULL, NULL);
197 checkError(err, "Copying h_g to device at d_g");
198
199 // Set the arguments to our compute kernel (a,b,c)
200 err = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_a);
201 err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_b);
202 err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_c);
203 err |= clSetKernelArg(ko_vadd, 3, sizeof(unsigned int), &count);
204 checkError(err, "Setting kernel arguments");
205
206 double rtime = wtime();
207
208 // Execute the kernel over the entire range of our 1d input data set
209 // letting the OpenCL runtime choose the work-group size
210 global = count;
211 err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global, NULL, 0, NULL, NULL);
212 checkError(err, "Enqueueing kernel");
213
214 // Wait for the commands to complete before stopping the timer
215 err = clFinish(commands);
216 checkError(err, "Waiting for kernel to finish");
217
218 rtime = wtime() - rtime;
219 printf("\nThe addition's C=A+B kernel ran in %lf seconds\n", rtime);
220
221 // Read back the results from the compute device
222 err = clEnqueueReadBuffer(commands, d_c, CL_TRUE, 0, sizeof(float) * count, h_c, 0, NULL, NULL);
223 if (err != CL_SUCCESS)
224 {
225     printf("Error: Failed to read output array!\n%s\n", err_code(err));
226     exit(1);
227 }
228
229
230 // Set the arguments to our compute kernel (c,e,d)
231 err = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_c);
232 err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_e);
233 err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_d);
234 err |= clSetKernelArg(ko_vadd, 3, sizeof(unsigned int), &count);
235 checkError(err, "Setting kernel arguments");
236
237 double rtime = wtime();
238
239 // Execute the kernel over the entire range of our 1d input data set
240 // letting the OpenCL runtime choose the work-group size
241 global = count;
242 err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global, NULL, 0, NULL, NULL);
243 checkError(err, "Enqueueing kernel");
244
245 // Wait for the commands to complete before stopping the timer
246 err = clFinish(commands);
247 checkError(err, "Waiting for kernel to finish");
248

```

```

249     rtime = wtime() - rtime;
250     printf("\nThe addition's D=C+E kernel ran in %lf seconds\n",rtime);
251
252     // Read back the results from the compute device
253     err = clEnqueueReadBuffer( commands, d_d, CL_TRUE, 0, sizeof(float) * count, h_d, 0, NULL, NULL );
254     if (err != CL_SUCCESS)
255     {
256         printf("Error: Failed to read output array!\n%s\n", err_code(err));
257         exit(1);
258     }
259
260     // Set the arguments to our compute kernel (d,f,g)
261     err = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_d);
262     err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_g);
263     err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_f);
264     err |= clSetKernelArg(ko_vadd, 3, sizeof(unsigned int), &count);
265     checkError(err, "Setting kernel arguments");
266
267     double rtime = wtime();
268
269     // Execute the kernel over the entire range of our 1d input data set
270     // letting the OpenCL runtime choose the work-group size
271     global = count;
272     err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global, NULL, 0, NULL, NULL);
273     checkError(err, "Enqueueing kernel");
274
275     // Wait for the commands to complete before stopping the timer
276     err = clFinish(commands);
277     checkError(err, "Waiting for kernel to finish");
278
279     rtime = wtime() - rtime;
280     printf("\nThe addition's F=D+G kernel ran in %lf seconds\n",rtime);
281
282     // Read back the results from the compute device
283     err = clEnqueueReadBuffer( commands, d_f, CL_TRUE, 0, sizeof(float) * count, h_f, 0, NULL, NULL );
284     if (err != CL_SUCCESS)
285     {
286         printf("Error: Failed to read output array!\n%s\n", err_code(err));
287         exit(1);
288     }
289
290
291     // Test the results (a,b,c)
292     correct = 0;
293     float tmp;
294
295     for(i = 0; i < count; i++)
296     {
297         tmp = h_a[i] + h_b[i]; // assign element i of a+b to tmp
298         tmp -= h_c[i]; // compute deviation of expected and output result
299         if(tmp*tmp < TOL*TOL) // correct if square deviation is less than tolerance squared
300             correct++;
301         else {
302             printf(" tmp %f h_a %f h_b %f h_c %f \n",tmp, h_a[i], h_b[i], h_c[i]);
303         }
304     }
305
306     // summarise results
307     printf("C = A+B: %d out of %d results were correct.\n", correct, count);
308
309     // Test the results (c,e,d)
310     correct1 = 0;
311     float tmp1;
312
313     for(i = 0; i < count; i++)
314     {
315         tmp1 = h_c[i] + h_e[i]; // assign element i of a+b to tmp
316         tmp1 -= h_d[i]; // compute deviation of expected and output result
317         if(tmp1*tmp1 < TOL*TOL) // correct if square deviation is less than tolerance squared
318             correct1++;
319         else {
320             printf(" tmp1 %f h_c %f h_e %f h_d %f \n",tmp1, h_c[i], h_e[i], h_d[i]);
321         }
322     }
323
324     // summarise results
325     printf("D = C+E: %d out of %d results were correct.\n", correct, count);
326
327     // Test the results (d,g,f)
328     correct2 = 0;
329     float tmp2;
330

```



```

331     for(i = 0; i < count; i++)
332     {
333         tmp2 = h_d[i] + h_g[i];    // assign element i of a+b to tmp
334         tmp2 -= h_f[i];           // compute deviation of expected and output result
335         if(tmp2*tmp2 < TOL*TOL)    // correct if square deviation is less than tolerance squared
336             correct2++;
337         else {
338             printf(" tmp2 %f h_d %f h_g %f h_f %f \n",tmp2, h_d[i], h_g[i], h_f[i]);
339         }
340     }
341
342     // summarise results
343     printf("F = D+G:  %d out of %d results were correct.\n", correct, count);
344
345     // cleanup then shutdown
346     clReleaseMemObject(d_a);
347     clReleaseMemObject(d_b);
348     clReleaseMemObject(d_c);
349     clReleaseMemObject(d_d);
350     clReleaseMemObject(d_e);
351     clReleaseMemObject(d_f);
352     clReleaseMemObject(d_g);
353
341
342     // summarise results
343     printf("F = D+G:  %d out of %d results were correct.\n", correct, count);
344
345     // cleanup then shutdown
346     clReleaseMemObject(d_a);
347     clReleaseMemObject(d_b);
348     clReleaseMemObject(d_c);
349     clReleaseMemObject(d_d);
350     clReleaseMemObject(d_e);
351     clReleaseMemObject(d_f);
352     clReleaseMemObject(d_g);
353
354
355     clReleaseProgram(program);
356     clReleaseKernel(ko_vadd);
357     clReleaseCommandQueue(commands);
358     clReleaseContext(context);
359
354
355     clReleaseProgram(program);
356     clReleaseKernel(ko_vadd);
357     clReleaseCommandQueue(commands);
358     clReleaseContext(context);
359
360     free(h_a);
361     free(h_b);
362     free(h_c);
363     free(h_d);
364     free(h_e);
365     free(h_f);
366     free(h_g);
367
368     return 0;
369 }

```

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.