360.252 - Computational Science on Many-Core Architectures

WS 2020 - Exercise 7

Christian Gollmann, 01435044

Last update: December 6, 2020

Contents

1 Dot Product with OpenCL 1

For my kernel, I chose to do the summation, as suggested in the chat, on the CPU. I tested it with vectors of size 128*1024, the result can be seen in figure 1.

Compile output

[Compilation successful]

Run output

```
# Platforms found: 2
# Devices found: 1
Using the following device: GeForce GTX 1080
Time to compile and create kernel: 0.330539

Vectors before kernel launch:
x: 3 3 3 ...
y: 2 2 2 ...

Vectors after kernel execution:
x: 3 3 3 ...
y: 2 2 2 ...
Dot-Product: 786432

#
# My first OpenCL application finished successfully!
#
```

Figure 1: result of dot product

Listing 1: OpenCL kernel for Dot Product

```
const char *my_opencl_program = ""
"#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n"     // required to enable 'double'
                                                      //inside OpenCL programs
""
"__kernel void dotProductFirstStep(__global double *x,\n"
"                        __global double *y,\n"
"                         __global double *dot, \n"
"                        unsigned int N\n)"
"{\n"
"  for (unsigned int i  = get_global_id(0);\n"
"                       i  < N;\n"
"                       i += get_global_size(0))\n"
"    dot[i] = x[i] * y[i];\n"
"}";
}
```

Listing 2: Summing vector elements on CPU

```
// sum the elements of dot
  ScalarType result = 0;
  result = std::accumulate(dot.begin(), dot.end(), result);
```

For my own convenience I also list the whole code embedding of the Dot Product.

Listing 3: code embedding for Dot Product

```
typedef double       ScalarType;

#include <iostream>
#include <numeric>
#include <string>
#include <vector>
#include <cmath>
#include <stdexcept>

#ifdef __APPLE__
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif

// Helper include file for error checking
#include "ocl-error.hpp"
#include "timer.hpp"


const char *my_opencl_program = ""
"#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n"    // required to enable
// 'double' inside OpenCL programs
""
"__kernel void dotProductFirstStep(__global double *x,\n"
"                       __global double *y,\n"
"                        __global double *dot, \n"
"                       unsigned int N\n)"
"{\n"
"  for (unsigned int i  = get_global_id(0);\n"
"                      i  < N;\n"
"                      i += get_global_size(0))\n"
"    dot[i] = x[i] * y[i];\n"
"}";  // you can have multiple kernels within a single OpenCL program.
// For simplicity, this OpenCL program contains only a single kernel.


int main()
{
  cl_int err;

  //
  /////////////////////////// Part 1: Set up an OpenCL context with one device ////////
  //

  //
  // Query platform:
  //
  cl_uint num_platforms;
  cl_platform_id platform_ids[42];   //no more than 42 platforms supported...
  err = clGetPlatformIDs(42, platform_ids, &num_platforms); OPENCL_ERR_CHECK(err);
  std::cout << "# Platforms found: " << num_platforms << std::endl;
  cl_platform_id my_platform = platform_ids[0];

```

```
55
56      //
57      // Query devices:
58      //
59      cl_device_id device_ids[42];
60      cl_uint num_devices;
61      err = clGetDeviceIDs(my_platform, CL_DEVICE_TYPE_ALL, 42, device_ids, &num_devices);
62      OPENCL_ERR_CHECK(err);
63      std::cout << "# Devices found: " << num_devices << std::endl;
64      cl_device_id my_device_id = device_ids[0];
65
66      char device_name[64];
67      size_t device_name_len = 0;
68      err = clGetDeviceInfo(my_device_id, CL_DEVICE_NAME, sizeof(char)*63,
69      device_name, &device_name_len); OPENCL_ERR_CHECK(err);
70      std::cout << "Using the following device: " << device_name << std::endl;
71
72      //
73      // Create context:
74      //
75      cl_context my_context = clCreateContext(0, 1, &my_device_id, NULL, NULL, &err);
76      OPENCL_ERR_CHECK(err);
77
78
79      //
80      // create a command queue for the device:
81      //
82      cl_command_queue my_queue = clCreateCommandQueueWithProperties(my_context,
83      my_device_id, 0, &err); OPENCL_ERR_CHECK(err);
84
85
86
87      //
88      ///////////////////////////// Part 2: Create a program and extract kernels ////////////
89      //
90
91      Timer timer;
92      timer.reset();
93
94      //
95      // Build the program:
96      //
97      size_t source_len = std::string(my_opencl_program).length();
98      cl_program prog = clCreateProgramWithSource(my_context, 1, &my_opencl_program,
99      &source_len, &err);
100     OPENCL_ERR_CHECK(err);
101     err = clBuildProgram(prog, 0, NULL, NULL, NULL, NULL);
102
103     //
104     // Print compiler errors if there was a problem:
105     //
106     if (err != CL_SUCCESS) {
107
108       char *build_log;
109       size_t ret_val_size;
110       err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, 0,
111       NULL, &ret_val_size);
```

```
112       build_log = (char *)malloc(sizeof(char) * (ret_val_size+1));
113       err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG,
114       ret_val_size, build_log, NULL);
115       build_log[ret_val_size] = '\0'; // terminate string
116       std::cout << "Log: " << build_log << std::endl;
117       free(build_log);
118       std::cout << "OpenCL program sources: " << std::endl <<
119       my_opencl_program << std::endl;
120       return EXIT_FAILURE;
121     }
122
123     //
124     // Extract the only kernel in the program:
125     //
126     cl_kernel my_kernel = clCreateKernel(prog, "dotProductFirstStep", &err);
127     OPENCL_ERR_CHECK(err);
128
129     std::cout << "Time to compile and create kernel: " << timer.get() << std::endl;
130
131
132     //
133     /////////////////////////// Part 3: Create memory buffers ////////////////////////////
134     //
135
136     //
137     // Set up buffers on host:
138     //
139     int N = 128*1024;
140     cl_uint vector_size = N;
141     std::vector<ScalarType> x(vector_size, 3.0);
142     std::vector<ScalarType> y(vector_size, 2.0);
143     std::vector<ScalarType> dot(vector_size, 0);
144
145     std::cout << std::endl;
146     std::cout << "Vectors before kernel launch:" << std::endl;
147     std::cout << "x: " << x[0] << " " << x[1] << " " << x[2] << " ..." << std::endl;
148     std::cout << "y: " << y[0] << " " << y[1] << " " << y[2] << " ..." << std::endl;
149
150     //
151     // Now set up OpenCL buffers:
152     //
153     cl_mem ocl_x = clCreateBuffer(my_context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
154     vector_size * sizeof(ScalarType), &(x[0]), &err); OPENCL_ERR_CHECK(err);
155     cl_mem ocl_y = clCreateBuffer(my_context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
156     vector_size * sizeof(ScalarType), &(y[0]), &err); OPENCL_ERR_CHECK(err);
157     cl_mem ocl_dot = clCreateBuffer(my_context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
158     vector_size * sizeof(ScalarType), &(dot[0]), &err); OPENCL_ERR_CHECK(err);
159
160
161     //
162     /////////////////////////// Part 4: Run kernel ////////////////////////////////////
163     //
164     size_t  local_size = 128;
165     size_t global_size = 128*128;
166
167     //
168     // Set kernel arguments:
```

```
169     //
170     err = clSetKernelArg(my_kernel, 0, sizeof(cl_mem),  (void*)&ocl_x);
171     OPENCL_ERR_CHECK(err);
172     err = clSetKernelArg(my_kernel, 1, sizeof(cl_mem),  (void*)&ocl_y);
173     OPENCL_ERR_CHECK(err);
174     err = clSetKernelArg(my_kernel, 2, sizeof(cl_mem),  (void*)&ocl_dot);
175     OPENCL_ERR_CHECK(err);
176     err = clSetKernelArg(my_kernel, 3, sizeof(cl_uint), (void*)&vector_size);
177     OPENCL_ERR_CHECK(err);
178
179     //
180     // Enqueue kernel in command queue:
181     //
182     err = clEnqueueNDRangeKernel(my_queue, my_kernel, 1, NULL, &global_size,
183     &local_size, 0, NULL, NULL);
184     OPENCL_ERR_CHECK(err);
185
186     // wait for all operations in queue to finish:
187     err = clFinish(my_queue);
188     OPENCL_ERR_CHECK(err);
189
190
191     //
192     ///////////////////////////// Part 5: Get data from OpenCL buffer ////////////////////
193     //
194
195     err = clEnqueueReadBuffer(my_queue, ocl_dot, CL_TRUE, 0,
196     sizeof(ScalarType) * dot.size(), &(dot[0]), 0, NULL, NULL); OPENCL_ERR_CHECK(err);
197
198
199     // sum the elements of dot
200     ScalarType result = 0;
201     result = std::accumulate(dot.begin(), dot.end(), result);
202
203     std::cout << std::endl;
204     std::cout << "Vectors after kernel execution:" << std::endl;
205     std::cout << "x: " << x[0] << " " << x[1] << " " << x[2] << " ..." << std::endl;
206     std::cout << "y: " << y[0] << " " << y[1] << " " << y[2] << " ..." << std::endl;
207     std::cout << "Dot-Product: " << result << std::endl;
208
209     //
210     // cleanup
211     //
212     clReleaseMemObject(ocl_x);
213     clReleaseMemObject(ocl_y);
214     clReleaseMemObject(ocl_dot);
215     clReleaseProgram(prog);
216     clReleaseCommandQueue(my_queue);
217     clReleaseContext(my_context);
218
219     std::cout << std::endl;
220     std::cout << "#" << std::endl;
221     std::cout << "# My first OpenCL application finished successfully!" << std::endl;
222     std::cout << "#" << std::endl;
223     return EXIT_SUCCESS;
224 }
```

2 Dot Product with OpenCL 2 + 3

It can be seen that there is basically no difference in timing the OpenCL Dot Product on CPU and GPU. That probably origins from the fact that I decided to sum the vector entries of the intermediate result on the CPU. This is however not the case with the CUDA implementation. Therefore the CUDA implementation runs faster than the OpenCL one.

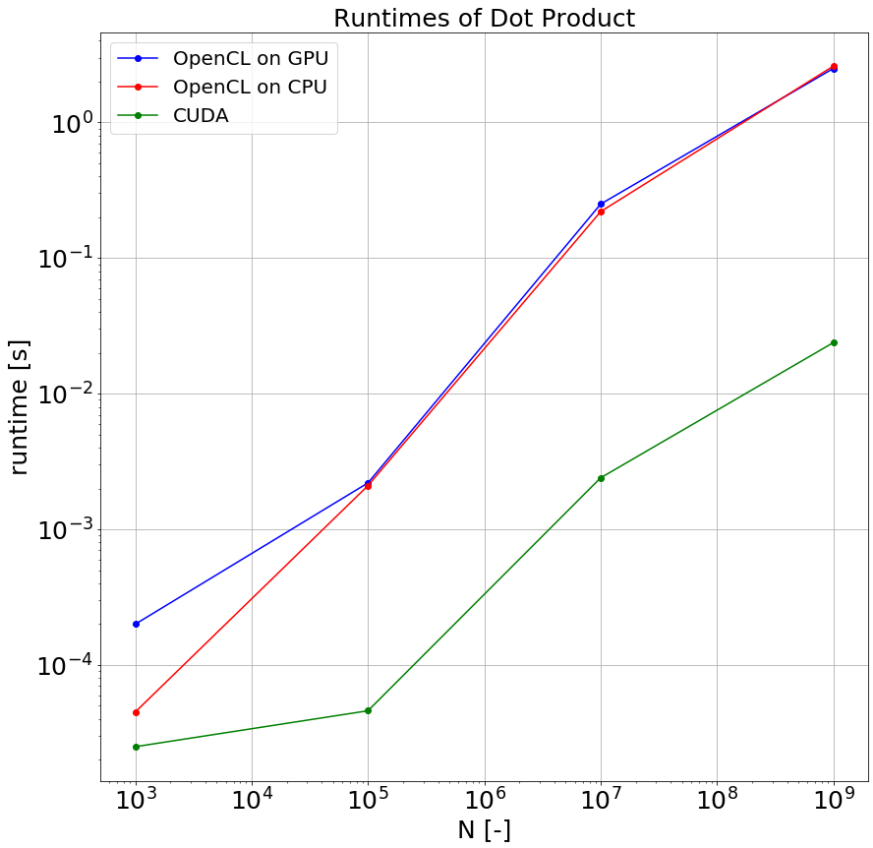

Figure 2: Runtimes of Dot Products

Listing 4: Timing for OpenCL Code

```
1  std::vector<double> timings;
2      for(int reps=0; reps < 10; ++reps) {
3          timer.reset();
4          //
5          // Now set up OpenCL buffers:
6          //
7          cl_mem ocl_x = clCreateBuffer(my_context, CL_MEM_READ_WRITE |
8          CL_MEM_COPY_HOST_PTR, vector_size * sizeof(ScalarType), &(x[0]), &err);
9          OPENCL_ERR_CHECK(err);
10         cl_mem ocl_y = clCreateBuffer(my_context, CL_MEM_READ_WRITE |
11         CL_MEM_COPY_HOST_PTR, vector_size * sizeof(ScalarType), &(y[0]), &err);
12         OPENCL_ERR_CHECK(err);
13         cl_mem ocl_dot = clCreateBuffer(my_context, CL_MEM_READ_WRITE |
14         CL_MEM_COPY_HOST_PTR, vector_size * sizeof(ScalarType), &(dot[0]), &err);
15         OPENCL_ERR_CHECK(err);
16
17
18         //
```

```
19          ///////////////////////////// Part 4: Run kernel ////////////////////////////////
20          //
21          size_t  local_size = 128;
22          size_t global_size = 128*128;
23
24          //
25          // Set kernel arguments:
26          //
27
28          err = clSetKernelArg(my_kernel, 0, sizeof(cl_mem),  (void*)&ocl_x);
29          OPENCL_ERR_CHECK(err);
30          err = clSetKernelArg(my_kernel, 1, sizeof(cl_mem),  (void*)&ocl_y);
31          OPENCL_ERR_CHECK(err);
32          err = clSetKernelArg(my_kernel, 2, sizeof(cl_mem),  (void*)&ocl_dot);
33          OPENCL_ERR_CHECK(err);
34          err = clSetKernelArg(my_kernel, 3, sizeof(cl_uint), (void*)&vector_size);
35          OPENCL_ERR_CHECK(err);
36
37          //
38          // Enqueue kernel in command queue:
39          //
40          err = clEnqueueNDRangeKernel(my_queue, my_kernel, 1, NULL, &global_size,
41          &local_size, 0, NULL, NULL); OPENCL_ERR_CHECK(err);
42
43          // wait for all operations in queue to finish:
44          err = clFinish(my_queue); OPENCL_ERR_CHECK(err);
45
46
47          //
48          ///////////////////////////// Part 5: Get data from OpenCL buffer ///////////////
49          //
50
51          err = clEnqueueReadBuffer(my_queue, ocl_dot, CL_TRUE, 0,
52          sizeof(ScalarType) * dot.size(), &(dot[0]), 0, NULL, NULL);
53          OPENCL_ERR_CHECK(err);
54
55
56          // sum the elements of dot
57          result = 0;
58          result = std::accumulate(dot.begin(), dot.end(), result);
59
60          timings.push_back(timer.get());
61          clReleaseMemObject(ocl_x);
62          clReleaseMemObject(ocl_y);
63          clReleaseMemObject(ocl_dot);
64
65      }
66
67      std::sort(timings.begin(), timings.end());
68      double time_elapsed = timings[10/2];
69
70    std::cout << "Calculation of Dot Product took " <<
71    time_elapsed << std::endl << std::endl;
```

Listing 5: Timing for CUDA Code

```
1 #include <stdio.h>
```

```
2  #include <iostream>
3  #include <algorithm>
4  #include "timer.hpp"
5  #include <random>
6
7
8  __global__ void dot_product(double *x, double *y, double *dot, unsigned int n)
9  {
10     unsigned int index = threadIdx.x + blockDim.x*blockIdx.x;
11     unsigned int stride = blockDim.x*gridDim.x;
12
13     __shared__ double cache[256];
14
15     double temp = 0.0;
16     while(index < n){
17         temp += x[index]*y[index];
18
19         index += stride;
20     }
21
22     cache[threadIdx.x] = temp;
23
24     __syncthreads();
25
26     for(int i = blockDim.x/2; i>0; i/=2)
27     {
28         __syncthreads();
29         if(threadIdx.x < i)
30             cache[threadIdx.x] += cache[threadIdx.x + i];
31     }
32
33     if(threadIdx.x == 0){
34         atomicAdd(dot, cache[0]);
35     }
36 }
37
38
39
40 int main()
41 {
42     unsigned int n = 128*1024;
43     double *h_prod;
44     double *d_prod;
45     double *h_x, *h_y;
46     double *d_x, *d_y;
47     Timer timer;
48
49     h_prod = new double[n];
50     h_x = new double[n];
51     h_y = new double[n];
52
53
54     // fill host array with data
55     for(unsigned int i=0;i<n;i++){
56         h_x[i] = 3;
57         h_y[i] = 2;
58     }
```

```
59
60      // start timer
61      std::vector<double> timings;
62      for(int reps=0; reps < 10; ++reps) {
63          timer.reset();
64
65          // allocate memory
66          cudaMalloc(&d_prod, sizeof(double));
67          cudaMalloc(&d_x, n*sizeof(double));
68          cudaMalloc(&d_y, n*sizeof(double));
69          cudaMemset(d_prod, 0.0, sizeof(double));
70
71
72          // copy data to device
73          cudaMemcpy(d_x, h_x, n*sizeof(double), cudaMemcpyHostToDevice);
74          cudaMemcpy(d_y, h_y, n*sizeof(double), cudaMemcpyHostToDevice);
75
76
77          dot_product<<<256, 256>>>(d_x, d_y, d_prod, n);
78
79          // copy data back to host
80          cudaMemcpy(h_prod, d_prod, sizeof(double), cudaMemcpyDeviceToHost);
81
82          // get runtime
83          timings.push_back(timer.get());
84      }
85
86      std::sort(timings.begin(), timings.end());
87      double time_elapsed = timings[10/2];
88
89
90      // report results
91      std::cout<<"dot product computed on GPU is: "<<*h_prod<<" and took " <<
92      time_elapsed << " s" <<std::endl;
93
94
95      // free memory
96      free(h_prod);
97      free(h_x);
98      free(h_y);
99      cudaFree(d_prod);
100     cudaFree(d_x);
101     cudaFree(d_y);
102
103 }
```