360.252 - Computational Science on Many-Core Architectures

WS 2020 - Exercise 8

Christian Gollmann, 01435044

Last update: December 12, 2020

Contents

1 Libraries

When comparing the runtimes of the dot product for different vector sizes N, some interesting results were found.
First of all it is nice to see, that the self made CUDA kernel performs very well compared to the other implementations.
Secondly, the vex implementation seems to be the most efficient for large N.
Thirdly, thrust and boost are structured much alike, they both calculate temporary vectors to hold x+y and x-y. Yet boost is performing really bad. First I thought I made some mistake with the timing but I couldn't find any. Probably I chose a very inefficient implementation.
The openCL kernel can not really be taken into account here because I'm still summing up the values on the CPU which may falsivy the result compared to the other implementations.

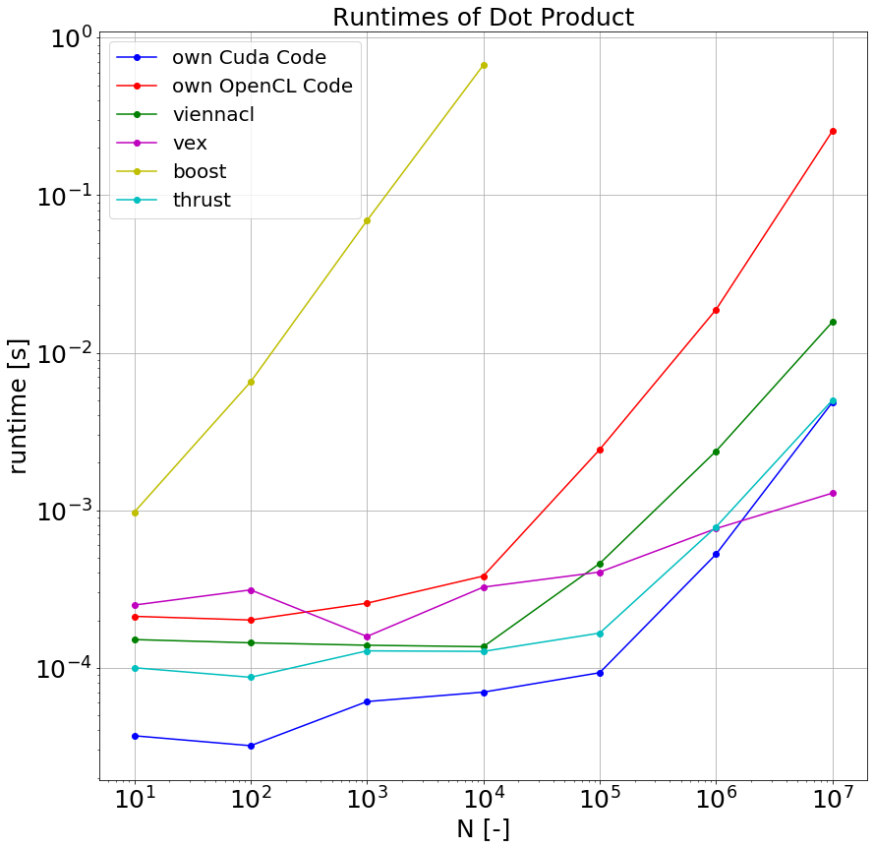

Figure 1: runtimes of dot product

Listing 1: Boost

```
1  #include <vector>
2  #include <algorithm>
3  #include <iostream>
4
5  #include <boost/compute/algorithm/transform.hpp>
6  #include <boost/compute/container/vector.hpp>
7  #include <boost/compute/functional/math.hpp>
8  #include <boost/geometry/arithmetic/dot_product.hpp>
9  #include <boost/range/numeric.hpp>
10 #include "timer.hpp"
```

```
11
12
13  namespace compute = boost::compute;
14
15  int main()
16  {
17      int N = 1000;
18      Timer timer;
19
20      // get default device and setup context
21      compute::device device = compute::system::default_device();
22      compute::context context(device);
23      compute::command_queue queue(context, device);
24
25      // generate random data on the host
26      std::vector<double> x(N, 1);
27      std::vector<double> y(N, 2);
28
29     // create a vector on the device
30     compute::vector<double> d_x(x.size(), context);
31     compute::vector<double> d_y(y.size(), context);
32     compute::vector<double> d_arg1(N, context);
33     compute::vector<double> d_arg2(N, context);
34
35     // transfer data from the host to the device
36     compute::copy(x.begin(), x.end(), d_x.begin(), queue);
37     compute::copy(y.begin(), y.end(), d_y.begin(), queue);
38      // calculate inner product
39
40  std::vector<double> timings;
41    double z;
42    for(int reps=0; reps < 10; ++reps) {
43        timer.reset();
44      compute::transform(d_x.begin(), d_x.end(), d_y.begin(), d_arg1.begin(),
45      compute::plus<double>{}, queue);
46      compute::transform(d_x.begin(), d_x.end(), d_y.begin(), d_arg2.begin(),
47      compute::minus<double>{}, queue);
48       z = boost::inner_product(d_arg1, d_arg2, 0);
49       timings.push_back(timer.get());
50      }
51    std::sort(timings.begin(), timings.end());
52    double time_elapsed = timings[10/2];
53
54    std::cout << "Time elapsed: " << time_elapsed << std::endl << std::endl;
55
56      std::cout << "Dot Product = " << z << std::endl;
57
58      return 0;
59  }
```

Listing 2: Thrust

```
1  #include <thrust/host_vector.h>
2  #include <thrust/inner_product.h>
3  #include <thrust/device_vector.h>
4  #include <thrust/sort.h>
5  #include <cstdlib>
```

```cpp
#include <algorithm>

#include "timer.hpp"


int main(void) {

    int N = 1000;
    Timer timer;

  // initialize x and y
  thrust::host_vector<double> h_x(N, 1.0);
  thrust::host_vector<double> h_y(N, 2.0);


  // transfer data to the device
  thrust::device_vector<double> d_x = h_x;
  thrust::device_vector<double> d_y = h_y;
  thrust::device_vector<double> d_arg1(N);
  thrust::device_vector<double> d_arg2(N);

  std::vector<double> timings;
  double z;
  for(int reps=0; reps < 10; ++reps) {
      timer.reset();
    thrust::transform(d_x.begin(), d_x.end(), d_y.begin(), d_arg1.begin(),
    thrust::plus<double>());
    thrust::transform(d_x.begin(), d_x.end(), d_y.begin(), d_arg2.begin(),
    thrust::minus<double>());

    z = thrust::inner_product(d_arg1.begin(), d_arg1.end(), d_arg2.begin(), 0.0);
    timings.push_back(timer.get());
  }
std::sort(timings.begin(), timings.end());
double time_elapsed = timings[10/2];

std::cout << "Time elapsed: " << time_elapsed << std::endl << std::endl;

  std::cout << "Inner Product = " << z << std::endl;

  return 0;
}
```

Listing 3: VexCL

```cpp
#include <iostream>
#include <stdexcept>
#include <vexcl/vexcl.hpp>
#include "timer.hpp"


int main() {
 vex::Context ctx(vex::Filter::GPU&&vex::Filter::DoublePrecision);

 std::cout << ctx << std::endl; // print list of selected devices

 size_t N = 1000;
```

```
13   Timer timer;
14   std::vector<double> a(N, 1.0), b(N, 2.0);
15
16   vex::vector<double> A(ctx, a);
17   vex::vector<double> B(ctx, b);
18
19  std::vector<double> timings;
20    double z;
21    for(int reps=0; reps < 10; ++reps) {
22        timer.reset();
23      vex::Reductor<double, vex::SUM> sum(ctx);
24      z = sum((A+B) * (A-B));
25      timings.push_back(timer.get());
26      }
27    std::sort(timings.begin(), timings.end());
28    double time_elapsed = timings[10/2];
29
30    std::cout << "Time elapsed: " << time_elapsed << std::endl << std::endl;
31
32   std::cout << "Dot Product = " << z << std::endl;
33
34   return 0;
35  }
```

Listing 4: ViennaCL

```
1  #include "timer.hpp"
2  #include <iostream>
3
4  #define VIENNACL_WITH_CUDA
5
6  #include "viennacl/vector.hpp"
7  #include "viennacl/linalg/inner_prod.hpp"
8
9
10 int main() {
11
12   Timer timer;
13   size_t N = 1000;
14   viennacl::vector<double> x = viennacl::scalar_vector<double>(N, 1.0);
15   viennacl::vector<double> y = viennacl::scalar_vector<double>(N, 2.0);
16
17   std::vector<double> timings;
18   double z;
19   for(int reps=0; reps < 10; ++reps) {
20       timer.reset();
21       z = viennacl::linalg::inner_prod(x+y, x-y);
22       timings.push_back(timer.get());
23     }
24   std::sort(timings.begin(), timings.end());
25   double time_elapsed = timings[10/2];
26
27   std::cout << "Time elapsed: " << time_elapsed << std::endl << std::endl;
28
29   std::cout << "Inner Product = " << z << std::endl;
30
31   return EXIT_SUCCESS;
```

```
}
```

Listing 5: CUDA

```
#include <algorithm>
#include <iostream>
#include <stdio.h>
#include <vector>
#include <iostream>

__global__ void dot_product(int* x, int* y, int* dot, int N) {

    int index = threadIdx.x + blockDim.x * blockIdx.x;
    int stride = blockDim.x * gridDim.x;

    __shared__ int cache[128];

    int temp = 0;
    while (index < N) {
        temp += (x[index] + y[index]) * (x[index] - y[index]);
        index += stride;
    }

    cache[threadIdx.x] = temp;

    __syncthreads();

    for (int i = blockDim.x/2; i > 0; i/= 2) {
        __syncthreads();
        if (threadIdx.x < i)
            cache[threadIdx.x] += cache[threadIdx.x + i];
    }

    if (threadIdx.x == 0)
        atomicAdd(dot, cache[0]);

}


int main() {

    Timer timer;
    int N = 1000;

    int *x = (int *)malloc(sizeof(int) * N);
    int *y = (int *)malloc(sizeof(int) * N);
    int *dot = (int *)malloc(sizeof(int));

    for (int i = 0; i < N; i++) {
        x[i] = 1;
        y[i] = 2;
    }
    *dot = 0;

    int *cuda_x;
    int *cuda_y;
    int *cuda_dot;
```

```
54      cudaMalloc(&cuda_x, sizeof(int) * N);
55      cudaMalloc(&cuda_y, sizeof(int) * N);
56      cudaMalloc(&cuda_dot, sizeof(int));
57
58      cudaMemcpy(cuda_x, x, sizeof(int) * N, cudaMemcpyHostToDevice);
59      cudaMemcpy(cuda_y, y, sizeof(int) * N, cudaMemcpyHostToDevice);
60      cudaMemcpy(cuda_dot, dot, sizeof(int), cudaMemcpyHostToDevice);
61
62      std::vector<double> timings;
63      for(int reps=0; reps < 10; ++reps) {
64          timer.reset();
65          dot_product<<<N/256, 128>>>(cuda_x, cuda_y, cuda_dot, N);
66          cudaMemcpy(dot, cuda_dot, sizeof(int), cudaMemcpyDeviceToHost);
67          timings.push_back(timer.get());
68          std::cout << "Dot Product = " << *dot << std::endl;
69          *dot = 0;
70          cudaMemcpy(cuda_dot, dot, sizeof(int), cudaMemcpyHostToDevice);
71      }
72
73      std::sort(timings.begin(), timings.end());
74      double time_elapsed = timings[10/2];
75
76      std::cout << "Time elapsed: " << time_elapsed << std::endl << std::endl;
77
78
79
80      return EXIT_SUCCESS;
81  }
```

Listing 6: OpenCL

```
1   typedef double       ScalarType;
2
3
4   #include <iostream>
5   #include <numeric>
6   #include <string>
7   #include <algorithm>
8   #include <vector>
9   #include <cmath>
10  #include <stdexcept>
11
12  #ifdef __APPLE__
13  #include <OpenCL/cl.h>
14  #else
15  #include <CL/cl.h>
16  #endif
17
18  // Helper include file for error checking
19  #include "ocl-error.hpp"
20  #include "timer.hpp"
21
22
23  const char *my_opencl_program = ""
24  "#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n"    // required to enable 'double'
25  inside OpenCL programs
26  ""
```

```
27 "__kernel void dotProductFirstStep(__global double *x,\n"
28 "                      __global double *y,\n"
29 "                       __global double *dot, \n"
30 "                      unsigned int N\n)"
31 "{\n"
32 "  for (unsigned int i  = get_global_id(0);\n"
33 "                     i  < N;\n"
34 "                     i += get_global_size(0))\n"
35 "    dot[i] = (x[i]+y[i]) * (x[i]-y[i]);\n"
36 "}";  // you can have multiple kernels within a single OpenCL program.
37 For simplicity, this OpenCL program contains only a single kernel.
38
39
40 int main()
41 {
42   cl_int err;
43
44   //
45   /////////////////////////// Part 1: Set up an OpenCL context with one device ////////
46   //
47
48   //
49   // Query platform:
50   //
51   cl_uint num_platforms;
52   cl_platform_id platform_ids[42];    //no more than 42 platforms supported...
53   err = clGetPlatformIDs(42, platform_ids, &num_platforms);
54   OPENCL_ERR_CHECK(err);
55   std::cout << "# Platforms found: " << num_platforms << std::endl;
56   cl_platform_id my_platform = platform_ids[1];
57
58
59   //
60   // Query devices:
61   //
62   cl_device_id device_ids[42];
63   cl_uint num_devices;
64   err = clGetDeviceIDs(my_platform, CL_DEVICE_TYPE_ALL, 42, device_ids, &num_devices);
65   OPENCL_ERR_CHECK(err);
66   std::cout << "# Devices found: " << num_devices << std::endl;
67   cl_device_id my_device_id = device_ids[0];
68
69   char device_name[64];
70   size_t device_name_len = 0;
71   err = clGetDeviceInfo(my_device_id, CL_DEVICE_NAME, sizeof(char)*63,
72   device_name, &device_name_len);
73   OPENCL_ERR_CHECK(err);
74   std::cout << "Using the following device: " << device_name << std::endl;
75
76   //
77   // Create context:
78   //
79   cl_context my_context = clCreateContext(0, 1, &my_device_id, NULL, NULL, &err);
80   OPENCL_ERR_CHECK(err);
81
82
83   //
```

```
84    // create a command queue for the device:
85    //
86    cl_command_queue my_queue = clCreateCommandQueueWithProperties(my_context,
87    my_device_id, 0, &err);
88    OPENCL_ERR_CHECK(err);
89
90
91
92    //
93    ////////////////////////// Part 2: Create a program and extract kernels ////////////
94    //
95
96    Timer timer;
97    timer.reset();
98
99    //
100   // Build the program:
101   //
102   size_t source_len = std::string(my_opencl_program).length();
103   cl_program prog = clCreateProgramWithSource(my_context, 1, &my_opencl_program,
104   &source_len, &err);OPENCL_ERR_CHECK(err);
105   err = clBuildProgram(prog, 0, NULL, NULL, NULL, NULL);
106
107   //
108   // Print compiler errors if there was a problem:
109   //
110   if (err != CL_SUCCESS) {
111
112     char *build_log;
113     size_t ret_val_size;
114     err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, 0, NULL,
115     &ret_val_size);
116     build_log = (char *)malloc(sizeof(char) * (ret_val_size+1));
117     err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, ret_val_size,
118     build_log, NULL);
119     build_log[ret_val_size] = '\0'; // terminate string
120     std::cout << "Log: " << build_log << std::endl;
121     free(build_log);
122     std::cout << "OpenCL program sources: "
123     << std::endl << my_opencl_program << std::endl;
124     return EXIT_FAILURE;
125   }
126
127   //
128   // Extract the only kernel in the program:
129   //
130   cl_kernel my_kernel = clCreateKernel(prog, "dotProductFirstStep", &err);
131   OPENCL_ERR_CHECK(err);
132
133   std::cout << "Time to compile and create kernel: " << timer.get() << std::endl;
134
135
136   //
137   ////////////////////////// Part 3: Create memory buffers ////////////////////////////
138   //
139
140   //
```

```
141   // Set up buffers on host:
142   //
143   int N = 1000;
144   cl_uint vector_size = N;
145   std::vector<ScalarType> x(vector_size, 1.0);
146   std::vector<ScalarType> y(vector_size, 2.0);
147   std::vector<ScalarType> dot(vector_size, 0);
148
149   std::cout << std::endl;
150   std::cout << "Vectors before kernel launch:" << std::endl;
151   std::cout << "x: " << x[0] << " " << x[1] << " " << x[2] << " ..." << std::endl;
152   std::cout << "y: " << y[0] << " " << y[1] << " " << y[2] << " ..." << std::endl;
153
154   ScalarType result = 0;
155   std::vector<double> timings;
156     for(int reps=0; reps < 10; ++reps) {
157         timer.reset();
158         //
159         // Now set up OpenCL buffers:
160         //
161         cl_mem ocl_x = clCreateBuffer(my_context,
162         CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, vector_size * sizeof(ScalarType),
163         &(x[0]), &err); OPENCL_ERR_CHECK(err);
164         cl_mem ocl_y = clCreateBuffer(my_context,
165         CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, vector_size * sizeof(ScalarType),
166         &(y[0]), &err); OPENCL_ERR_CHECK(err);
167         cl_mem ocl_dot = clCreateBuffer(my_context,
168         CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, vector_size * sizeof(ScalarType),
169         &(dot[0]), &err); OPENCL_ERR_CHECK(err);
170
171
172         //
173         /////////////////////////// Part 4: Run kernel ////////////////////////////////
174         //
175         size_t  local_size = 128;
176         size_t global_size = 128*128;
177
178         //
179         // Set kernel arguments:
180         //
181
182         err = clSetKernelArg(my_kernel, 0, sizeof(cl_mem),  (void*)&ocl_x);
183         OPENCL_ERR_CHECK(err);
184         err = clSetKernelArg(my_kernel, 1, sizeof(cl_mem),  (void*)&ocl_y);
185         OPENCL_ERR_CHECK(err);
186         err = clSetKernelArg(my_kernel, 2, sizeof(cl_mem),  (void*)&ocl_dot);
187         OPENCL_ERR_CHECK(err);
188         err = clSetKernelArg(my_kernel, 3, sizeof(cl_uint), (void*)&vector_size);
189         OPENCL_ERR_CHECK(err);
190
191         //
192         // Enqueue kernel in command queue:
193         //
194         err = clEnqueueNDRangeKernel(my_queue, my_kernel, 1, NULL, &global_size,
195         &local_size, 0, NULL, NULL); OPENCL_ERR_CHECK(err);
196
197         // wait for all operations in queue to finish:
```

```
198         err = clFinish(my_queue); OPENCL_ERR_CHECK(err);
199
200
201         //
202         ///////////////////////////// Part 5: Get data from OpenCL buffer ///////////////
203         //
204
205         err = clEnqueueReadBuffer(my_queue, ocl_dot, CL_TRUE, 0,
206         sizeof(ScalarType) * dot.size(), &(dot[0]), 0, NULL, NULL);
207         OPENCL_ERR_CHECK(err);
208
209
210         // sum the elements of dot
211         result = 0;
212         result = std::accumulate(dot.begin(), dot.end(), result);
213
214         timings.push_back(timer.get());
215         clReleaseMemObject(ocl_x);
216         clReleaseMemObject(ocl_y);
217         clReleaseMemObject(ocl_dot);
218
219     }
220
221     std::sort(timings.begin(), timings.end());
222     double time_elapsed = timings[10/2];
223
224   std::cout << "Calculation of Dot Product took " << time_elapsed << std::endl
225   << std::endl;
226
227   std::cout << std::endl;
228   std::cout << "Vectors after kernel execution:" << std::endl;
229   std::cout << "x: " << x[0] << " " << x[1] << " " << x[2] << " ..." << std::endl;
230   std::cout << "y: " << y[0] << " " << y[1] << " " << y[2] << " ..." << std::endl;
231   std::cout << "Dot-Product: " << result << std::endl;
232
233   //
234   // cleanup
235   //
236   clReleaseProgram(prog);
237   clReleaseCommandQueue(my_queue);
238   clReleaseContext(my_context);
239
240   std::cout << std::endl;
241   std::cout << "#" << std::endl;
242   std::cout << "# My first OpenCL application finished successfully!" << std::endl;
243   std::cout << "#" << std::endl;
244   return EXIT_SUCCESS;
245 }
```