INSTITUTE OF MICROELECTRONICS

360.252 Computational Science on Many-Core Architectures

Exercise 8

Author:
Peter Holzner, 01426733

Submission: December 15, 2020

Contents

---

1 Libraries

Code listings for this task:

- Boost.compute: Listing 1
- Thrust: Listing 2
- VexCL: Listing 3
- ViennaCL: Listing 5
- My Kernel - CUDA: Listing 6
- My Kernel - OpenCL: Listing 7

As you promised at the start of the lecture: It's quite easy to write a kernel that can do a specific task faster than an implementation that relies on library functions - see Fig. 1 for reference. The libraries using the CUDA backend clearly outperform the OpenCL backed versions - although it's likely not an entirely fair comparison since these tests are performed on an Nvidia device only. As you mentioned, Nvidia does enforce a soft-lock on its' devices.

A big advantage of the specialized one kernel approach is likely that one only needs to schedule one kernel - provided the task can even be done within one kernel. In this case, one also does not need any temporaries to store the result of the intermediate computations $X+Y$ and $X-Y$, which is unfortunately unavoidable with some of the libraries (e.g. Boost.compute). Although it is not always immediately clear how each library deals with these temporaries, if they do not have to be created by hand (compare thrust and boost.compute to say VexCL), and what's really happening in the background.

VexCl seems to deal with these problems very well by using their Vector expressions approach where a specialized kernel is created for each expression - VexCl performs very well for me and is basically on par with my custom OpenCL kernel across the board. VexCL generates only one kernel for the task, as can be seen when setting the `VEXCL_SHOW_KERNELS` macro. The generated kernel is shown in Listing 4. As far as I understood, these expressions work similar to generators in python - they don't actually compute anything until needed.

On the other side, other libraries use a more traditional, functional approach in the style of BLAS and LAPACK (ViennaCL) or the C++ standard library algorithms (thrust and boost.compute). The speedup plot in Fig. 1 shows that CUDA outperforms OpenCL even when using the same library ("frontend") - compare ViennaCL/CUDA with ViennaCL/OpenCL. See also Fig. 2, where the runtimes are shown seperated by the backend (CUDA, OpenCL). Even for the custom kernel, the CUDA version outperforms my OpenCL version - which aren't entirely equal as I had to resort to a secondary stage (summation of work group results on the CPU) for OpenCL (was unable to

implement atomic_add for doubles). Again, there might be some vendor shenanigans going on that I can't know or account for here, so these results have to be viewed with a grain of salt.[1]

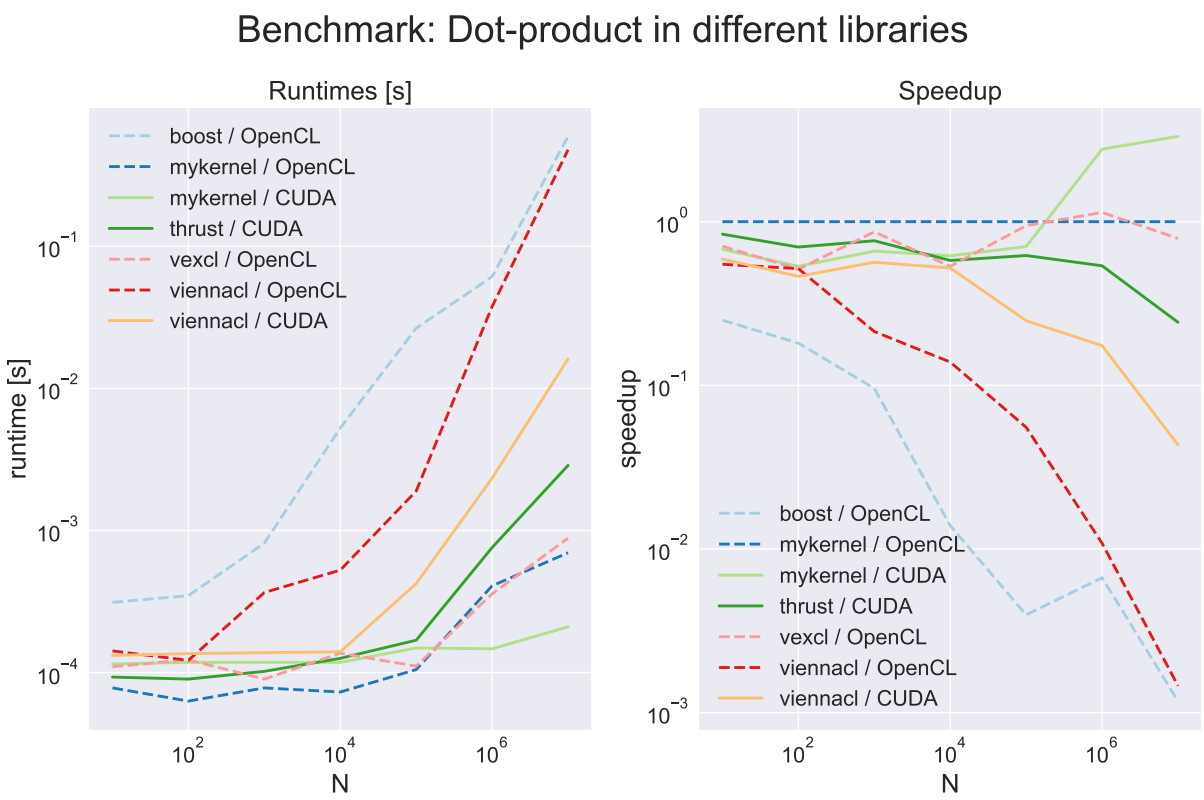

Fig. 1. The right side shows the runtimes to compute the final result. The right side shows the speedup (=relative runtime) with my OpenCL kernel as the base line.

Take away: If you "have" to use an Nvidia device then use the CUDA backend (unsurprisingly).

I hope I could make some of my thoughts and learnings understandable. This exercise was very fun and interesting for me, but it's also just a dip into a topic that can be very hard to really discuss in detail due to various facets and the difficulties of actually designing fair benchmarks. That fact also makes it even harder to accurately and meaningful describe the results of these benchmarks.

1.1 Some notes on my implementations and code

I tried to implement an atomicAdd() function for double precision floats in OpenCL based on the code snippet given in the CUDA documentation, but was unable to get it to work. You can find it in Listing 7 and easily test it out by unsetting the `#define KERNEL_ARRAY` and adjusting the kernel code appropriately, as shown in Fig. 3. The benchmark below is then adjusted based on the the define via `#ifdef` precompiler directives. Also, if there are better ways to formulate the task for a specific library (e.g. ViennaCL), please do tell me!

[1] Also, performance isn't always everything. Ease-of-use, portability and personal preferences are also important factors to consider when choosing a library for a specific project.

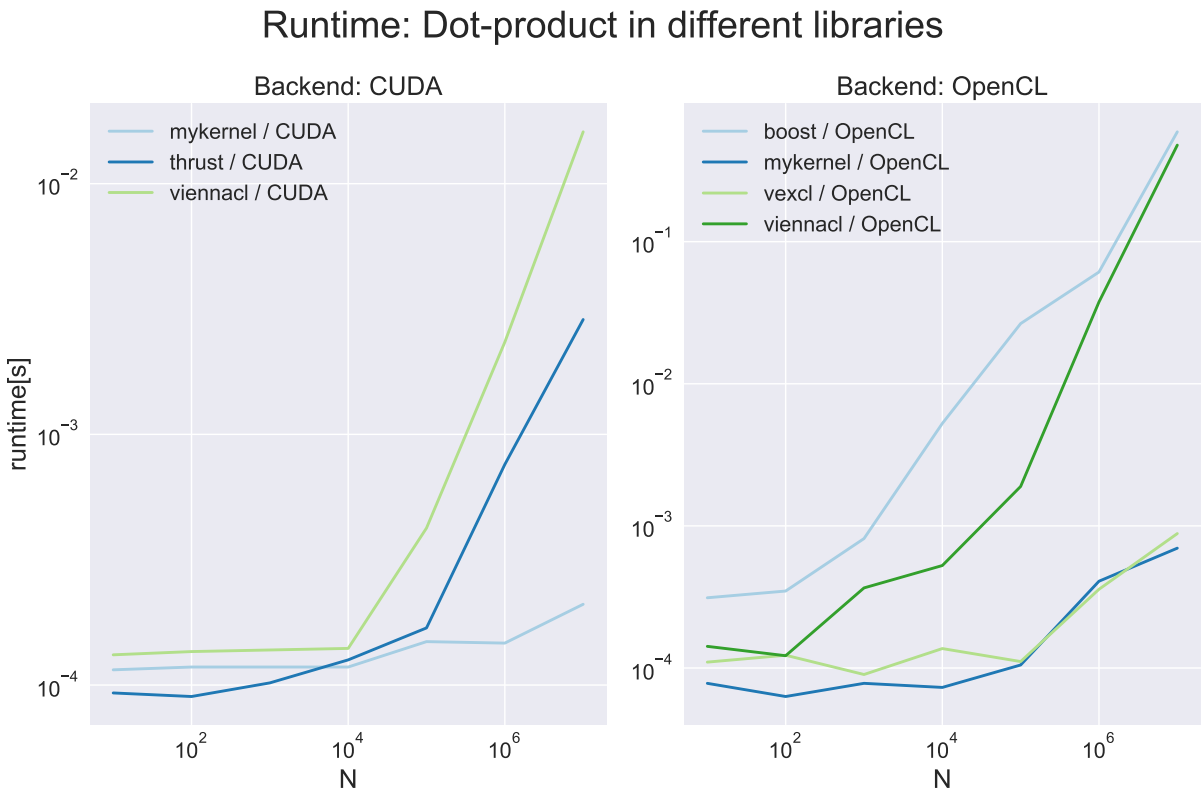

Fig. 2. Runtimes seperated by backend.

```
121     for (size_t i = CUCL_LOCALSIZE0 / 2; i != 0; i /= 2)
122     {
123         CUCL_LOCBARRIER;
124         if (lid < i)
125             cache[lid] += cache[lid + i];
126     }
127
128     if (lid == 0)
129         result[group_id] = cache[lid]; // KERNEL_ARRAY
130         // ATOMIC_ADD_FUNC(result, cache[0]); // KERNEL_ATOMIC
131 })
132 "\n\n";
133
134 // To test the atomic kernel version, simply switch the lines above
135 // and comment out the define below
136 #define KERNEL_ARRAY
137 #ifndef KERNEL_ARRAY
138     #define KERNEL_ATOMIC
139 #endif
140
141
142 //
```

Fig. 3. How to switch the versions of my OpenCL kernel.

2 BONUS ROUND

My code for the bonus exercise is here: Listing 8.

I have already tried to get this to work for the first task. I encountered some issues when having the kernel be defined in the same file as the rest of the benchmark code. The precompiler would actually not insert the proper statements based on the previously used defines before applying the `STRINGIFY` macro. I figured out (much thanks to stackoverflow, of course) the following workaround to trick the precompiler into doing that:

```
39  // entry point, but need to account for multiple arguments AND need to actually
    force replacement before applying the macro
40  #define STRINGIFY(...) mFn2(__VA_ARGS__)
41  #define mFn2(ANS) #ANS
```

Fig. 4. Modifed Macros for stringification$^{(TM)}$.

3 Sidenote

I tried to write my reports in a more informal fashion, so they're a bit more fun for you to read (hopefully). I hope my attempt is at least appreciated and if you want me to switch to a more formal/neutral/scientific writing style please tell me!

Happy holidays and "einen Guten Rutsch"!

4 Code and Kernels

Listings

Listing 1: Ex8: boost.compute

```
#include <vector>
#include <string>
#include <algorithm>
#include <numeric>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include "timer.hpp"
// boost
#include <boost/compute/algorithm/transform.hpp>
#include <boost/compute/algorithm/inner_product.hpp>
#include <boost/compute/container/vector.hpp>
#include <boost/compute/functional/math.hpp>
namespace compute = boost::compute;
// functions used:
// inner_prod(): https://www.boost.org/doc/libs/1_65_1/libs/compute/doc/html/boost/compute/inner_product.html

// DEFINES
#define EX "ex8"
#define CSV_NAME "ph_data_boost_ocl.csv"

#define COUT
#define NUM_TEST 5
#define N_MIN 10
#define N_MAX 10000000 //1e8
//
//----------- Helper functions
//
template <template <typename, typename> class Container,
          typename ValueType,
          typename Allocator = std::allocator<ValueType>>
double median(Container<ValueType, Allocator> data)
{
    size_t size = data.size();
    if (size == 0)
        return 0.;
    sort(data.begin(), data.end());
    size_t mid = size / 2;

    return size % 2 == 0 ? (data[mid] + data[mid - 1]) / 2 : data[mid];
};

template <typename T>
double median(T *array, size_t size)
{
    if (size == 0)
        return 0.;
    sort(array, array + size);
    size_t mid = size / 2;

    return size % 2 == 0 ? (array[mid] + array[mid - 1]) / 2 : array[mid];
};
//
//----------- functions for this program
//

double benchmark(compute::context& context, compute::command_queue& queue, size_t N, double x_init, double y_init, std::vector<double>& results)
{

    Timer timer;
    timer.reset();
    std::vector<double> x(N, x_init);
    std::vector<double> y(N, y_init);

    compute::vector<double> X(N, context);
    compute::vector<double> Y(N, context);
    compute::vector<double> TMP(N, context);
    compute::vector<double> TMP2(N, context);
```

```
69
70      compute::copy(x.begin(), x.end(), X.begin(), queue);
71      compute::copy(y.begin(), y.end(), Y.begin(), queue);
72      results[0] = timer.get();
73
74      double dot;
75
76      std::vector<double> tmp(NUM_TEST, 0.0);
77      for (int iter = 0; iter < NUM_TEST; iter++) {
78          timer.reset();
79          compute::transform(X.begin(), X.end(),
80              Y.begin(), TMP.begin(), compute::plus<double>{}, queue);
81          // I tried to reuse the vector X for the result of the last transform,
82          // but it did not work properly. I assume, that the reason is that these
83          // are asynchronous calls that can happen in parallel,
84          // so it might happen that parts of X
85          // are overwritten before the first is finished.
86          // That seems weird...
87          compute::transform(X.begin(), X.end(),
88              Y.begin(), TMP2.begin(), compute::minus<double>{}, queue);
89
90          dot = compute::inner_product(TMP.begin(), TMP.end(),
91                       TMP2.begin(), 0.0, queue);
92          tmp[iter] = timer.get();
93      }
94      results[1] = median(tmp);
95
96      double true_dot = (x_init + y_init) * (x_init - y_init) * N;
97
98  #ifdef COUT
99      std::cout << "(x+y, x-y) = " << dot << " ?= " << true_dot << std::endl;
100     std::cout << "Computation took " << results[1] << "s" << std::endl;
101 #endif
102     timer.reset();
103     results[3] = dot;
104     results[2] = timer.get();
105
106     return dot;
107 }
108
109 int main(int argc, char const *argv[])
110 {
111     // get default device and setup context
112     compute::device device = compute::system::default_device();
113     compute::context context(device);
114     std::cout << device.name() << std::endl; // print list of selected devices
115     compute::command_queue queue(context, device);
116
117     double x_init = 1., y_init = 2.;
118     std::vector<double> results(4, 0.0);
119
120     std::ofstream csv;
121     std::string sep = ";";
122     std::string header = "N;vec_init_time;dot_time;memcpy_time;dot_result";
123     auto to_csv = [&csv, &sep] (double x) { csv << sep << x;};
124
125     csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
126     csv << header << std::endl;
127     for (size_t N = N_MIN; N < 1+N_MAX; N*=10){
128 #ifdef COUT
129         std::cout << "N: " <<  N << std::endl;
130 #endif
131         benchmark(context, queue, N, x_init, y_init, results);
132         csv << N;
133         std::for_each(results.begin(), results.end(), to_csv);
134         csv << std::endl;
135     }
136
137     std::cout << "Data: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;
138
139     return EXIT_SUCCESS;
140 }
```

Listing 2: Ex8: thrust

```cpp
#include <vector>
#include <string>
#include <algorithm>
#include <numeric>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include "timer.hpp"
// thrust
#include <thrust/host_vector.h>
#include <thrust/inner_product.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

// DEFINES
#define EX "ex8"
#define CSV_NAME "ph_data_thrust_cuda.csv"

#define COUT
#define NUM_TEST 5
#define N_MIN 10
#define N_MAX 10000000 //1e8
//
//----------- Helper functions
//
template <template <typename, typename> class Container,
          typename ValueType,
          typename Allocator = std::allocator<ValueType>>
double median(Container<ValueType, Allocator> data)
{
    size_t size = data.size();
    if (size == 0)
        return 0.;
    sort(data.begin(), data.end());
    size_t mid = size / 2;

    return size % 2 == 0 ? (data[mid] + data[mid - 1]) / 2 : data[mid];
};

template <typename T>
double median(T *array, size_t size)
{
    if (size == 0)
        return 0.;
    sort(array, array + size);
    size_t mid = size / 2;

    return size % 2 == 0 ? (array[mid] + array[mid - 1]) / 2 : array[mid];
};
//
//----------- functions for this program
//

double benchmark(size_t N, double x_init, double y_init, std::vector<double>& results)
{

    Timer timer;
    timer.reset();
    thrust::host_vector<double> x(N, x_init);
    thrust::host_vector<double> y(N, y_init);

    thrust::device_vector<double> X = x;
    thrust::device_vector<double> Y = y;
    thrust::device_vector<double> TMP(N);
    thrust::device_vector<double> TMP2(N);
    results[0] = timer.get();

    double dot;

    std::vector<double> tmp(NUM_TEST, 0.0);
```

```
71      for (int iter = 0; iter < NUM_TEST; iter++) {
72          timer.reset();
73          thrust::transform(X.begin(), X.end(),
74              Y.begin(), TMP.begin(), thrust::plus<double>{});
75          // I tried to reuse the vector X for the result of the last transform,
76          // but it did not work properly. I assume, that the reason is that these
77          // are asynchronous calls that can happen in parallel,
78          // so it might happen that parts of X
79          // are overwritten before the first is finished.
80          // That seems weird...
81          thrust::transform(X.begin(), X.end(),
82              Y.begin(), TMP2.begin(), thrust::minus<double>{});
83
84          dot = thrust::inner_product(TMP.begin(), TMP.end(),
85                      TMP2.begin(), 0.0);
86          tmp[iter] = timer.get();
87          std::cout << "Took " << tmp[iter] << "s for this iteration" << std::endl;
88      }
89      results[1] = median(tmp);
90
91      double true_dot = (x_init + y_init) * (x_init - y_init) * N;
92
93  #ifdef COUT
94      std::cout << "(x+y, x-y) = " << dot << " ?= " << true_dot << std::endl;
95      std::cout << "Computation took " << results[1] << "s" << std::endl;
96  #endif
97      timer.reset();
98      results[3] = dot;
99      results[2] = timer.get();
100
101     return dot;
102 }
103
104 int main(int argc, char const *argv[])
105 {
106     double x_init = 1., y_init = 2.;
107     std::vector<double> results(4, 0.0);
108
109     std::ofstream csv;
110     std::string sep = ";";
111     std::string header = "N;vec_init_time;dot_time;memcpy_time;dot_result";
112     auto to_csv = [&csv, &sep] (double x) { csv << sep << x;};
113
114     csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
115     csv << header << std::endl;
116     for (size_t N = N_MIN; N < 1+N_MAX; N*=10){
117 #ifdef COUT
118         std::cout << "N: " <<  N << std::endl;
119 #endif
120         benchmark(N, x_init, y_init, results);
121         csv << N;
122         std::for_each(results.begin(), results.end(), to_csv);
123         csv << std::endl;
124     }
125
126     std::cout << "Data: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;
127
128     return EXIT_SUCCESS;
129 }
```

Listing 3: Ex8: VexCL

```
1  #include <vector>
2  #include <string>
3  #include <algorithm>
4  #include <numeric>
5  #include <iostream>
6  #include <fstream>
7  #include <cstdlib>
8  #include "timer.hpp"
9  // VexCL
10 #include <stdexcept>
```

```
11 // #define VEXCL_SHOW_KERNELS
12 #define VEXCL_BACKEND_OPENCL // default
13 //#define VEXCL_BACKEND_COMPUTE
14 //#define VEXCL_BACKEND_CUDA
15 #include <vexcl/vexcl.hpp>
16
17 // DEFINES
18 #define EX "ex8"
19 #ifdef VEXCL_BACKEND_OPENCL
20     #define CSV_NAME "ph_data_vexcl_ocl.csv"
21 #endif
22 #ifdef VEXCL_BACKEND_COMPUTE
23     #define CSV_NAME "ph_data_vexcl_ocl2.csv"
24 #endif
25 #ifdef VEXCL_BACKEND_CUDA
26     #define CSV_NAME "ph_data_vexcl_ocl3.csv"
27 #endif
28
29 #define COUT
30 #define NUM_TEST 5
31 #define N_MIN 10
32 #define N_MAX 10000000 //1e8
33 //
34 //----------- Helper functions
35 //
36 template <template <typename, typename> class Container,
37           typename ValueType,
38           typename Allocator = std::allocator<ValueType>>
39 double median(Container<ValueType, Allocator> data)
40 {
41     size_t size = data.size();
42     if (size == 0)
43         return 0.;
44     sort(data.begin(), data.end());
45     size_t mid = size / 2;
46
47     return size % 2 == 0 ? (data[mid] + data[mid - 1]) / 2 : data[mid];
48 };
49
50 template <typename T>
51 double median(T *array, size_t size)
52 {
53     if (size == 0)
54         return 0.;
55     sort(array, array + size);
56     size_t mid = size / 2;
57
58     return size % 2 == 0 ? (array[mid] + array[mid - 1]) / 2 : array[mid];
59 };
60 //
61 //----------- functions for this program
62 //
63
64 double benchmark(vex::Context ctx, size_t N, double x_init, double y_init, std::vector<
    double>& results)
65 {
66     Timer timer;
67     timer.reset();
68     std::vector<double> x(N, x_init);
69     std::vector<double> y(N, y_init);
70     vex::vector<double> X(ctx, x);
71     vex::vector<double> Y(ctx, y);
72
73     vex::Reductor<double, vex::SUM> DOT(ctx);
74     results[0] = timer.get();
75
76     double dot;
77
78     std::vector<double> tmp(NUM_TEST, 0.0);
79     for (int iter = 0; iter < NUM_TEST; iter++) {
80         timer.reset();
81         dot = DOT( (X+Y)*(X-Y) );
```

```
82          tmp[iter] = timer.get();
83      }
84      results[1] = median(tmp);
85
86      double true_dot = (x_init + y_init) * (x_init - y_init) * N;
87
88
89  #ifdef COUT
90      std::cout << "(x+y, x-y) = " << dot << " ?= " << true_dot << std::endl;
91      std::cout << "Computation took " << results[1] << "s" << std::endl;
92  #endif
93      timer.reset();
94      results[3] = dot;
95      results[2] = timer.get();
96
97      return dot;
98  }
99
100 int main(int argc, char const *argv[])
101 {
102      vex::Context ctx(vex::Filter::GPU&&vex::Filter::DoublePrecision);
103     std::cout << ctx << std::endl; // print list of selected devices
104
105     double x_init = 1., y_init = 2.;
106     std::vector<double> results(4, 0.0);
107
108     std::ofstream csv;
109     std::string sep = ";";
110     std::string header = "N;vec_init_time;dot_time;memcpy_time;dot_result";
111     auto to_csv = [&csv, &sep] (double x) { csv << sep << x;};
112
113     csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
114     csv << header << std::endl;
115     for (size_t N = N_MIN; N < 1+N_MAX; N*=10){
116 #ifdef COUT
117         std::cout << "N: " <<  N << std::endl;
118 #endif
119         benchmark(ctx, N, x_init, y_init, results);
120         csv << N;
121         std::for_each(results.begin(), results.end(), to_csv);
122         csv << std::endl;
123     }
124
125     std::cout << "Data: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;
126
127     return EXIT_SUCCESS;
128 }
```

Listing 4: Ex8: Generated VexCL kernel.

```
1  kernel void vexcl_reductor_kernel(
2      ulong n,
3      global double *prm_1,
4      global double *prm_2,
5      global double *prm_3,
6      global double *prm_4,
7      global double *g_odata,
8      local double *smem)
9  {
10     double mySum = 0;
11     for (ulong idx = get_global_id(0); idx < n; idx += get_global_size(0))
12     {
13         mySum = SUM_double(mySum, ((prm_1[idx] + prm_2[idx]) * (prm_3[idx] - prm_4[idx])));
14     }
15     local double *sdata = smem;
16     size_t tid = get_local_id(0);
17     size_t block_size = get_local_size(0);
18     sdata[tid] = mySum;
19     barrier(CLK_LOCAL_MEM_FENCE);
20     if (block_size >= 1024)
21     {
22         if (tid < 512)
```

```
23          {
24              sdata[tid] = mySum = SUM_double(mySum, sdata[tid + 512]);
25          }
26          barrier(CLK_LOCAL_MEM_FENCE);
27      }
28      if (block_size >= 512)
29      {
30          if (tid < 256)
31          {
32              sdata[tid] = mySum = SUM_double(mySum, sdata[tid + 256]);
33          }
34          barrier(CLK_LOCAL_MEM_FENCE);
35      }
36      if (block_size >= 256)
37      {
38          if (tid < 128)
39          {
40              sdata[tid] = mySum = SUM_double(mySum, sdata[tid + 128]);
41          }
42          barrier(CLK_LOCAL_MEM_FENCE);
43      }
44      if (block_size >= 128)
45      {
46          if (tid < 64)
47          {
48              sdata[tid] = mySum = SUM_double(mySum, sdata[tid + 64]);
49          }
50          barrier(CLK_LOCAL_MEM_FENCE);
51      }
52      if (block_size >= 64)
53      {
54          if (tid < 32)
55          {
56              sdata[tid] = mySum = SUM_double(mySum, sdata[tid + 32]);
57          }
58          barrier(CLK_LOCAL_MEM_FENCE);
59      }
60      if (block_size >= 32)
61      {
62          if (tid < 16)
63          {
64              sdata[tid] = mySum = SUM_double(mySum, sdata[tid + 16]);
65          }
66          barrier(CLK_LOCAL_MEM_FENCE);
67      }
68      if (block_size >= 16)
69      {
70          if (tid < 8)
71          {
72              sdata[tid] = mySum = SUM_double(mySum, sdata[tid + 8]);
73          }
74          barrier(CLK_LOCAL_MEM_FENCE);
75      }
76      if (block_size >= 8)
77      {
78          if (tid < 4)
79          {
80              sdata[tid] = mySum = SUM_double(mySum, sdata[tid + 4]);
81          }
82          barrier(CLK_LOCAL_MEM_FENCE);
83      }
84      if (block_size >= 4)
85      {
86          if (tid < 2)
87          {
88              sdata[tid] = mySum = SUM_double(mySum, sdata[tid + 2]);
89          }
90          barrier(CLK_LOCAL_MEM_FENCE);
91      }
92      if (block_size >= 2)
93      {
94          if (tid < 1)
```

```
95          {
96              sdata[tid] = mySum = SUM_double(mySum, sdata[tid + 1]);
97          }
98          barrier(CLK_LOCAL_MEM_FENCE);
99      }
100     if (tid == 0)
101         g_odata[get_group_id(0)] = sdata[0];
102 }
```

Listing 5: Ex8: ViennaCL

```
1   #include <vector>
2   #include <string>
3   #include <algorithm>
4   #include <numeric>
5   #include <iostream>
6   #include <fstream>
7   #include <cstdlib>
8   #include "timer.hpp"
9   // ViennaCL
10  #define VIENNACL_WITH_CUDA
11  // #define VIENNACL_WITH_OPENCL
12  #include "viennacl/vector.hpp"
13  #include "viennacl/linalg/inner_prod.hpp"
14
15  // DEFINES
16  #define EX "ex8"
17  #define HOST_DOT
18  #ifdef VIENNACL_WITH_CUDA
19      #define CSV_NAME "ph_data_viennacl_cuda.csv"
20  #endif
21  #ifdef VIENNACL_WITH_OPENCL
22      #define CSV_NAME "ph_data_viennacl_ocl.csv"
23  #endif
24  #define COUT
25  #define NUM_TEST 5
26  #define N_MIN 10
27  #define N_MAX 10000000 //1e8
28  //
29  //----------- Helper functions
30  //
31  template <template <typename, typename> class Container,
32            typename ValueType,
33            typename Allocator = std::allocator<ValueType>>
34  double median(Container<ValueType, Allocator> data)
35  {
36      size_t size = data.size();
37      if (size == 0)
38          return 0.;
39      sort(data.begin(), data.end());
40      size_t mid = size / 2;
41
42      return size % 2 == 0 ? (data[mid] + data[mid - 1]) / 2 : data[mid];
43  };
44
45  template <typename T>
46  double median(T *array, size_t size)
47  {
48      if (size == 0)
49          return 0.;
50      sort(array, array + size);
51      size_t mid = size / 2;
52
53      return size % 2 == 0 ? (array[mid] + array[mid - 1]) / 2 : array[mid];
54  };
55  //
56  //----------- functions for this program
57  //
58
59  double viennacl_benchmark(size_t N, double x_init, double y_init, std::vector<double>&
        results)
60  {
```

```
61      Timer timer;
62      timer.reset();
63      viennacl::vector<double> x = viennacl::scalar_vector<double>(N, x_init);
64      viennacl::vector<double> y = viennacl::scalar_vector<double>(N, y_init);
65      results[0] = timer.get();
66
67  #ifndef HOST_DOT
68      viennacl::scalar<double> dot;
69  #endif
70  #ifdef HOST_DOT
71      double dot ;
72  #endif
73
74      std::vector<double> tmp(NUM_TEST, 0.0);
75      for (int iter = 0; iter < NUM_TEST; iter++) {
76          timer.reset();
77          dot = viennacl::linalg::inner_prod(x+y, x-y);
78          tmp[iter] = timer.get();
79      }
80      results[1] = median(tmp);
81
82      double true_dot = (x_init + y_init) * (x_init - y_init) * N;
83
84
85  #ifdef COUT
86      std::cout << "(x+y, x-y) = " << dot << " ?= " << true_dot << std::endl;
87      std::cout << "Computation took " << results[1] << "s" << std::endl;
88  #endif
89      timer.reset();
90      results[3] = dot;
91      results[2] = timer.get();
92
93      return dot;
94  }
95
96  int main(int argc, char const *argv[])
97  {
98      double x_init = 1., y_init = 2.;
99      std::vector<double> results(4, 0.0);
100
101     std::ofstream csv;
102     std::string sep = ";";
103     std::string header = "N;vec_init_time;dot_time;memcpy_time;dot_result";
104     auto to_csv = [&csv, &sep] (double x) { csv << sep << x;};
105
106     csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
107     csv << header << std::endl;
108     for (size_t N = N_MIN; N < 1+N_MAX; N*=10){
109 #ifdef COUT
110         std::cout << "N: " <<  N << std::endl;
111 #endif
112         viennacl_benchmark(N, x_init, y_init, results);
113         csv << N;
114         std::for_each(results.begin(), results.end(), to_csv);
115         csv << std::endl;
116     }
117
118     std::cout << "Data: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;
119
120     return EXIT_SUCCESS;
121 }
```

Listing 6: Ex8: mykernel - CUDA

```
1  #include <vector>
2  #include <string>
3  #include <algorithm>
4  #include <numeric>
5  #include <iostream>
6  #include <fstream>
7  #include <cstdlib>
8  #include "timer.hpp"
```

```
 9
10  // DEFINES
11  #define EX "ex8"
12  #define CSV_NAME "ph_data_mykernel_cuda.csv"
13
14  #define COUT
15  #define NUM_TEST 5
16  #define N_MIN 10
17  #define N_MAX 10000000 //1e7
18
19  //
20  //----------- Helper functions
21  //
22  template <template <typename, typename> class Container,
23            typename ValueType,
24            typename Allocator = std::allocator<ValueType>>
25  double median(Container<ValueType, Allocator> data)
26  {
27      size_t size = data.size();
28      if (size == 0)
29          return 0.;
30      sort(data.begin(), data.end());
31      size_t mid = size / 2;
32
33      return size % 2 == 0 ? (data[mid] + data[mid - 1]) / 2 : data[mid];
34  };
35
36  template <typename T>
37  double median(T *array, size_t size)
38  {
39      if (size == 0)
40          return 0.;
41      sort(array, array + size);
42      size_t mid = size / 2;
43
44      return size % 2 == 0 ? (array[mid] + array[mid - 1]) / 2 : array[mid];
45  };
46  //
47  // my kernel
48  //
49  #define STRINGIFY(ARG) ARG
50  #define CUCL_KERNEL __global__                                  // __kernel
51  #define CUCL_GLOBMEM                                            // __global
52  #define CUCL_LOCMEM __shared__                                  // __local
53  #define CUCL_GLOBALID0 blockDim.x *blockIdx.x + threadIdx.x // get_global_id(0)
54  #define CUCL_GLOBALSIZE0 gridDim.x *blockDim.x                  // get_global_size(0)
55  #define CUCL_LOCALSIZE0 blockDim.x                              // get_local_size(0)
56  #define CUCL_LOCALID0 threadIdx.x                               // get_local_id(0)
57  #define ATOMIC_ADD_FUNC atomicAdd                               // my_atomic_add
58
59  #define LOCAL_SIZE 256
60  #define BLOCK_SIZE LOCAL_SIZE
61  #define GRID_SIZE 256
62  #define GLOBAL_SIZE (BLOCK_SIZE*LOCAL_SIZE)
63
64  // // atomicAdd for OpenCL
65  // #ifndef ulong
66  //     #define ulong unsigned long
67  // #endif
68  // void my_atomic_add(volatile CUCL_GLOBMEM double *p, double val) {
69  //   volatile CUCL_GLOBMEM ulong* address_as_ul = (volatile CUCL_GLOBMEM ulong *) p;
70  //   volatile ulong old = *address_as_ul, assumed;
71  //   ulong val_as_ul =  (ulong) val;
72  //   do  {
73  //     assumed = old;
74  //     old = atomic_add(address_as_ul, val_as_ul);
75  //   } while (assumed != old);
76  // };
77
78  CUCL_KERNEL void initKernel(CUCL_GLOBMEM double *x, const uint N, const double val)
79  {
80      const uint stride = CUCL_GLOBALSIZE0;
```

```
81      uint gid = CUCL_GLOBALID0;
82
83      for (; gid < N; gid += stride)
84          x[gid] = val;
85  };
86
87  CUCL_KERNEL void some_asymmetry_relation(uint N, CUCL_GLOBMEM const double *x, CUCL_GLOBMEM
        const double *y, CUCL_GLOBMEM double *result)
88  {
89      const uint stride = CUCL_GLOBALSIZE0;
90      uint gid = CUCL_GLOBALID0;
91      uint tid = threadIdx.x;
92      CUCL_LOCMEM double cache[LOCAL_SIZE];
93
94      double val = 0.0;
95      for (; gid < N; gid += stride)
96          val = (x[gid] + y[gid]) * (x[gid] - y[gid]);
97      cache[tid] = val;
98
99      __syncthreads();
100     for (size_t i = CUCL_LOCALSIZE0 / 2; i != 0; i /= 2)
101     {
102         __syncthreads();
103         if (tid < i)
104             cache[tid] += cache[tid + i];
105     }
106
107     if (tid == 0)
108         atomicAdd(result, cache[0]);
109 };
110
111 //
112 //------------ functions for this program
113 //
114 double benchmark(size_t N, double x_init, double y_init, std::vector<double> &results)
115 {
116
117     Timer timer;
118     timer.reset();
119
120     std::vector<double> x(N, x_init);
121     std::vector<double> y(N, y_init);
122
123     double *X;
124     double *Y;
125     cudaMalloc(&X, N * sizeof(double));
126     cudaMemcpy(X, x.data(), N * sizeof(double), cudaMemcpyHostToDevice);
127     cudaMalloc(&Y, N * sizeof(double));
128     cudaMemcpy(Y, y.data(), N * sizeof(double), cudaMemcpyHostToDevice);
129     results[0] = timer.get();
130
131     double dot = 0.0;
132     double *DOT;
133     cudaMalloc(&DOT, sizeof(double));
134     cudaMemcpy(DOT, &dot, sizeof(double), cudaMemcpyHostToDevice);
135
136     std::vector<double> tmp(NUM_TEST, 0.0);
137     for (int iter = 0; iter < NUM_TEST; iter++)
138     {
139         timer.reset();
140         some_asymmetry_relation<<<GRID_SIZE, BLOCK_SIZE>>>(N, X, Y, DOT);
141     cudaMemcpy(&dot, DOT, sizeof(double), cudaMemcpyDeviceToHost);
142         tmp[iter] = timer.get();
143     }
144     results[1] = median(tmp);
145
146     double true_dot = (x_init + y_init) * (x_init - y_init) * N;
147
148 #ifdef COUT
149     std::cout << "(x+y, x-y) = " << dot << " ?= " << true_dot << std::endl;
150     std::cout << "Computation took " << results[1] << "s" << std::endl;
151 #endif
```

```
152     timer.reset();
153     results[3] = dot;
154     results[2] = timer.get();
155 
156     cudaFree(X);
157     cudaFree(Y);
158     cudaFree(DOT);
159 
160     return dot;
161 }
162 
163 int main(int argc, char const *argv[])
164 {
165     double x_init = 1., y_init = 2.;
166     std::vector<double> results(4, 0.0);
167 
168     std::ofstream csv;
169     std::string sep = ";";
170     std::string header = "N;vec_init_time;dot_time;memcpy_time;dot_result";
171     auto to_csv = [&csv, &sep](double x) { csv << sep << x; };
172 
173     csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
174     csv << header << std::endl;
175     for (size_t N = N_MIN; N < 1 + N_MAX; N *= 10)
176     {
177 #ifdef COUT
178         std::cout << "N: " << N << std::endl;
179 #endif
180         benchmark(N, x_init, y_init, results);
181         csv << N;
182         std::for_each(results.begin(), results.end(), to_csv);
183         csv << std::endl;
184     }
185 
186     std::cout << "Data: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;
187 
188     return EXIT_SUCCESS;
189 }
```

Listing 7: Ex8: mykernel - OpenCL

```
1  typedef double ScalarType;
2  #include <vector>
3  #include <string>
4  #include <algorithm>
5  #include <numeric>
6  #include <iostream>
7  #include <fstream>
8  #include <cstdlib>
9  #include "timer.hpp"
10 
11 #ifdef __APPLE__
12 #include <OpenCL/cl.h>
13 #else
14 #include <CL/cl.h>
15 #endif
16 
17 // Helper include file for error checking
18 #include "ocl-error.hpp"
19 
20 // DEFINES
21 #define EX "ex8"
22 #define CSV_NAME "ph_data_mykernel_ocl.csv"
23 
24 #define COUT
25 #define NUM_TEST 5
26 #define N_MIN 10
27 #define N_MAX 10000000 //1e7
28 
29 //
30 //----------- Helper functions
31 //
```

```
32  template <template <typename, typename> class Container,
33            typename ValueType,
34            typename Allocator = std::allocator<ValueType>>
35  double median(Container<ValueType, Allocator> data)
36  {
37      size_t size = data.size();
38      if (size == 0)
39          return 0.;
40      sort(data.begin(), data.end());
41      size_t mid = size / 2;
42
43      return size % 2 == 0 ? (data[mid] + data[mid - 1]) / 2 : data[mid];
44  };
45
46  template <typename T>
47  double median(T *array, size_t size)
48  {
49      if (size == 0)
50          return 0.;
51      sort(array, array + size);
52      size_t mid = size / 2;
53
54      return size % 2 == 0 ? (array[mid] + array[mid - 1]) / 2 : array[mid];
55  };
56  //
57  // my kernel
58  //
59  // #define STRINGIFY(ARG) ARG
60  #define CUCL_KERNEL __kernel
61  #define CUCL_GLOBMEM __global
62  #define CUCL_LOCMEM __local
63  #define CUCL_GLOBALID0 get_global_id(0)
64  #define CUCL_GLOBALSIZE0 get_global_size(0)
65  #define CUCL_LOCALSIZE0 get_local_size(0)
66  #define CUCL_LOCALID0 get_local_id(0)
67  #define CUCL_LOCBARRIER barrier(CLK_LOCAL_MEM_FENCE) //__syncthreads()
68  #define ATOMIC_ADD_FUNC my_atomic_add
69
70  // entry point, but need to account for multiple arguments AND need to actually force
        replacement before applying the macro
71  #define STRINGIFY(...) mFn2(__VA_ARGS__)
72  #define mFn2(ANS) #ANS
73
74  #define LOCAL_SIZE 128
75  #define BLOCK_SIZE LOCAL_SIZE
76  #define GRID_SIZE 128
77  #define GLOBAL_SIZE (GRID_SIZE * LOCAL_SIZE)
78
79  // atomicAdd for OpenCL
80  #ifndef ulong
81      #define ulong unsigned long
82  #endif
83
84  std::string ocl_prog = "#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n"
85  "#pragma OPENCL EXTENSION cl_khr_int64_base_atomics : enable\n"
86  "#pragma OPENCL EXTENSION cl_khr_int64_extended_atomics : enable\n"
87  STRINGIFY(void my_atomic_add(volatile CUCL_GLOBMEM double *p, double val)
88  {
89    volatile CUCL_GLOBMEM ulong* address_as_ul = (volatile CUCL_GLOBMEM ulong *) p;
90    volatile ulong old = *address_as_ul;
91    volatile ulong assumed;
92    ulong val_as_ul =  (ulong) val;
93    do  {
94      assumed = old;
95      old = atomic_add(address_as_ul, val_as_ul);
96    } while (assumed != old);
97  })""
98  "\n\n"
99  STRINGIFY(CUCL_KERNEL void initKernel(CUCL_GLOBMEM double *x, const uint N, const double val
        )
100 {
101     const uint stride = CUCL_GLOBALSIZE0;
```

```
102     uint gid = CUCL_GLOBALID0;
103
104     for (; gid < N; gid += stride)
105         x[gid] = val;
106 })
107 "\n\n"
108 STRINGIFY(CUCL_KERNEL void some_asymmetry_relation(uint N, CUCL_GLOBMEM const double *x,
        CUCL_GLOBMEM const double *y, CUCL_GLOBMEM double *result)
109 {
110     const uint stride = CUCL_GLOBALSIZE0;
111     uint gid = CUCL_GLOBALID0;
112     uint lid = CUCL_LOCALID0;
113     uint group_id = get_group_id(0);
114     CUCL_LOCMEM double cache[LOCAL_SIZE];
115
116     double val = 0.0;
117     for (uint i = gid; i < N; i += stride)
118         val += (x[i] + y[i]) * (x[i] - y[i]);
119     cache[lid] = val;
120
121     for (size_t i = CUCL_LOCALSIZE0 / 2; i != 0; i /= 2)
122     {
123         CUCL_LOCBARRIER;
124         if (lid < i)
125             cache[lid] += cache[lid + i];
126     }
127
128     if (lid == 0)
129         result[group_id] = cache[lid]; // KERNEL_ARRAY
130         // ATOMIC_ADD_FUNC(result, cache[0]); // KERNEL_ATOMIC
131 })
132 "\n\n";
133
134 // To test the atomic kernel version, simply switch the lines above
135 // and comment out the define below
136 #define KERNEL_ARRAY
137 #ifndef KERNEL_ARRAY
138     #define KERNEL_ATOMIC
139 #endif
140
141
142 //
143 //----------- functions for this program
144 //
145 double benchmark(
146     cl_context& context, cl_command_queue& queue, cl_kernel& kernel,
147     size_t N, double x_init, double y_init, std::vector<double> &results)
148 {
149     cl_int err;
150     Timer timer;
151
152     size_t local_size = LOCAL_SIZE;
153     size_t global_size = GLOBAL_SIZE;
154     size_t groups = 1 + int(N/LOCAL_SIZE);
155     // std::cout << "LOCAL_SIZE: " << LOCAL_SIZE << std::endl;
156     // std::cout << "GLOBAL_SIZE: " << GLOBAL_SIZE << std::endl;
157     // std::cout << "groups: " << groups << std::endl;
158
159     double dot = 0.0;
160 #ifdef KERNEL_ARRAY
161     size_t dot_group_size = GRID_SIZE;
162 #endif
163
164 #ifdef KERNEL_ATOMIC
165     size_t dot_group_size = 1;
166 #endif
167     // std::vector<double> dot_group_results(N, dot);
168     std::vector<double> dot_group_results(dot_group_size, dot);
169
170     timer.reset();
171
172     std::vector<double> x(N, x_init);
```

```
173     std::vector<double> y(N, y_init);
174
175
176     cl_mem X = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, N * sizeof(
            double), x.data(), &err);
177     OPENCL_ERR_CHECK(err);
178     cl_mem Y = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, N * sizeof(
            double), y.data(), &err);
179     OPENCL_ERR_CHECK(err);
180
181     results[0] = timer.get();
182
183 // #ifdef KERNEL_ARRAY
184     cl_mem DOT  = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, sizeof(
            double)*dot_group_results.size(), (double*)dot_group_results.data(), &err);
            OPENCL_ERR_CHECK(err);
185 // #endif
186     // DOT = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, sizeof(double
            ), &dot, &err); OPENCL_ERR_CHECK(err);
187
188     cl_uint vector_size = N;
189 // #ifdef KERNEL_ATOMIC
190 //     // cl_double DOT = dot;
191 //     cl_mem DOT = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, sizeof
    (double), &dot, &err); OPENCL_ERR_CHECK(err);
192 // #endif
193
194     err = clSetKernelArg(kernel, 0, sizeof(cl_uint), (void*)&vector_size); OPENCL_ERR_CHECK(
            err);
195     err = clSetKernelArg(kernel, 1, sizeof(cl_mem),  (double*)&X);OPENCL_ERR_CHECK(err);
196     err = clSetKernelArg(kernel, 2, sizeof(cl_mem),  (double*)&Y); OPENCL_ERR_CHECK(err);
197 // #ifdef KERNEL_ATOMIC
198 //     err = clSetKernelArg(kernel, 3, sizeof(cl_mem),  (double*)&DOT); OPENCL_ERR_CHECK(err
    );
199 // #endif
200 // #ifdef KERNEL_ARRAY
201     err = clSetKernelArg(kernel, 3, sizeof(cl_mem),  (double*)&DOT); OPENCL_ERR_CHECK(err);
202 // #endif
203
204     std::vector<double> tmp(NUM_TEST, 0.0);
205     for (int iter = 0; iter < NUM_TEST; iter++)
206     {
207         dot = 0.0;
208         timer.reset();
209         err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size, &local_size, 0,
                NULL, NULL); OPENCL_ERR_CHECK(err);
210
211         err = clFinish(queue);
212         OPENCL_ERR_CHECK(err);
213 #ifdef KERNEL_ARRAY
214         err = clEnqueueReadBuffer(queue, DOT, CL_TRUE, 0, dot_group_results.size()*sizeof(
                double), dot_group_results.data(), 0, NULL, NULL);
215         OPENCL_ERR_CHECK(err);
216
217         // wait for all operations in queue to finish:
218         err = clFinish(queue);
219         OPENCL_ERR_CHECK(err);
220
221         for(auto& g: dot_group_results)
222             dot += g;
223 #endif
224
225         tmp[iter] = timer.get();
226     }
227     results[1] = median(tmp);
228
229     double true_dot = (x_init + y_init) * (x_init - y_init) * N;
230
231     timer.reset();
232 #ifdef KERNEL_ATOMIC
233     err = clEnqueueReadBuffer(queue, DOT, CL_TRUE, 0, dot_group_results.size()*sizeof(double
            ), dot_group_results.data(), 0, NULL, NULL);
```

```
234     OPENCL_ERR_CHECK(err);
235     err = clFinish(queue);
236     OPENCL_ERR_CHECK(err);
237     dot = dot_group_results[0];
238 #endif
239     results[3] = dot;
240     results[2] = timer.get();
241 
242 #ifdef COUT
243     std::cout << "(x+y, x-y) = " << dot << " ?= " << true_dot << std::endl;
244     std::cout << "Computation took " << results[1] << "s" << std::endl;
245 #endif
246 
247     clReleaseMemObject(X);
248     clReleaseMemObject(Y);
249     clReleaseMemObject(DOT);
250 
251     return dot;
252 }
253 
254 int main(int argc, char const *argv[])
255 {
256     cl_int err;
257     std::string target = "GPU";
258 
259     //
260     ////////////////////////// Part 1: Set up an OpenCL context with one device
        //////////////////////////////////
261     //
262 
263     //
264     // Query platform:
265     //
266     cl_uint num_platforms;
267     cl_platform_id platform_ids[42];   //no more than 42 platforms supported...
268     err = clGetPlatformIDs(42, platform_ids, &num_platforms); OPENCL_ERR_CHECK(err);
269     std::cout << "# Platforms found: " << num_platforms << std::endl;
270 
271 
272     //
273     // Query devices:
274     //
275     cl_device_id device_ids[42];
276     cl_uint num_devices;
277     char device_name[64];
278     cl_device_id my_device_id;
279     cl_platform_id my_platform;
280     for (int i = 0; i < num_platforms; ++i)
281     {
282         my_platform = platform_ids[i];
283         if (target == "GPU") {
284         err = clGetDeviceIDs(my_platform, CL_DEVICE_TYPE_GPU, 42, device_ids, &num_devices);
285         }
286         else {
287         err = clGetDeviceIDs(my_platform, CL_DEVICE_TYPE_CPU, 42, device_ids, &num_devices);
288         }
289         if (err == CL_SUCCESS)
290         break;
291     }
292     OPENCL_ERR_CHECK(err);
293     std::cout << "# Devices found: " << num_devices << std::endl;
294     my_device_id = device_ids[0];
295 
296     size_t device_name_len = 0;
297     err = clGetDeviceInfo(my_device_id, CL_DEVICE_NAME, sizeof(char)*63, device_name, &
        device_name_len); OPENCL_ERR_CHECK(err);
298 
299     std::cout << "Using the following device: " << device_name << std::endl;
300 
301     //
302     // Create context:
303     //
```

```
304     cl_context my_context = clCreateContext(0, 1, &my_device_id, NULL, NULL, &err);
            OPENCL_ERR_CHECK(err);
305
306
307     //
308     // create a command queue for the device:
309     //
310     cl_command_queue my_queue = clCreateCommandQueueWithProperties(my_context, my_device_id,
             0, &err); OPENCL_ERR_CHECK(err);
311
312
313     //
314     /////////////////////////// Part 2: Create a program and extract kernels
            ///////////////////////////////////
315     //
316     //
317     // Build the program:
318     //
319     cl_kernel my_kernel;
320     cl_program prog;
321
322     const char * my_opencl_program = ocl_prog.c_str();
323
324 #ifdef COUT
325     std::cout << "OpenCL program sources: " << std::endl << my_opencl_program << std::endl;
326 #endif
327     size_t source_len = std::string(my_opencl_program).length();
328     prog = clCreateProgramWithSource(my_context, 1, &my_opencl_program, &source_len, &err);
            OPENCL_ERR_CHECK(err);
329     err = clBuildProgram(prog, 0, NULL, NULL, NULL, NULL);
330
331     //
332     // Print compiler errors if there was a problem:
333     //
334     if (err != CL_SUCCESS) {
335
336         char *build_log;
337         size_t ret_val_size;
338         err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, 0, NULL, &
                ret_val_size);
339         build_log = (char *)malloc(sizeof(char) * (ret_val_size+1));
340         err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, ret_val_size,
                build_log, NULL);
341         build_log[ret_val_size] = '\0'; // terminate string
342         std::cout << "Log: " << build_log << std::endl;
343         free(build_log);
344         return EXIT_FAILURE;
345     }
346
347     my_kernel = clCreateKernel(prog, "some_asymmetry_relation", &err); OPENCL_ERR_CHECK(err)
            ;
348
349
350     //
351     // ------------ Benchmark setup ------------------
352     //
353     double x_init = 1., y_init = 2.;
354     std::vector<double> results(4, 0.0);
355
356     std::ofstream csv;
357     std::string sep = ";";
358     std::string header = "N;vec_init_time;dot_time;memcpy_time;dot_result";
359     auto to_csv = [&csv, &sep](double x) { csv << sep << x; };
360
361     csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
362     csv << header << std::endl;
363     for (size_t N = N_MIN; N < 1 + N_MAX; N *= 10)
364     {
365 #ifdef COUT
366         std::cout << "N: " << N << std::endl;
367 #endif
368         benchmark(my_context, my_queue, my_kernel, N, x_init, y_init, results);
```

```
369         csv << N;
370         std::for_each(results.begin(), results.end(), to_csv);
371         csv << std::endl;
372     }
373
374     std::cout << "Data: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;
375
376
377     clReleaseProgram(prog);
378     clReleaseCommandQueue(my_queue);
379     clReleaseContext(my_context);
380
381     return EXIT_SUCCESS;
382 }
```

Listing 8: Ex8: BONUS

```
1  //
2  // Tutorial for demonstrating a simple OpenCL vector addition kernel
3  //
4  // Author: Karl Rupp    rupp@iue.tuwien.ac.at
5  //
6
7  typedef double       ScalarType;
8
9
10 #include <iostream>
11 #include <string>
12 #include <vector>
13 #include <cmath>
14 #include <stdexcept>
15
16 #ifdef __APPLE__
17 #include <OpenCL/cl.h>
18 #else
19 #include <CL/cl.h>
20 #endif
21
22 // Helper include file for error checking
23 #include "ocl-error.hpp"
24 #include "timer.hpp"
25
26 //
27 // Transformation to OpenCL
28 //
29 #define CUCL_KERNEL __kernel
30 #define CUCL_GLOBALMEM __global
31 #define CUCL_LOCALMEM __local
32 #define CUCL_GLOBALID0 get_global_id(0)
33 #define CUCL_GLOBALSIZE0 get_global_size(0)
34 #define CUCL_GROUPID0 get_group_id(0)
35 #define CUCL_LOCALSIZE0 get_local_size(0)
36 #define CUCL_LOCALID0 get_local_id(0)
37 #define CUCL_BARRIER barrier(CLK_LOCAL_MEM_FENCE) //__syncthreads()
38
39 // entry point, but need to account for multiple arguments AND need to actually force
       replacement before applying the macro
40 #define STRINGIFY(...) mFn2(__VA_ARGS__)
41 #define mFn2(ANS) #ANS
42
43 // ##define STRINGIFY(ARG)   #ARG
44
45 const char *my_opencl_program = "#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n"
46 #include "dot.cucl"
47 ;
48
49 // undefine STRINGIFY after use to avoid global havoc:
50 #undef STRINGIFY
51 #undef mFn2
52
53
54 int dot_opencl(unsigned int N)
```

```
55  {
56    cl_int err;
57
58    //
59    /////////////////////////// Part 1: Set up an OpenCL context with one device ////////////////////////////////////
60    //
61
62    //
63    // Query platform:
64    //
65    cl_uint num_platforms;
66    cl_platform_id platform_ids[42];   //no more than 42 platforms supported...
67    err = clGetPlatformIDs(42, platform_ids, &num_platforms); OPENCL_ERR_CHECK(err);
68    std::cout << "# Platforms found: " << num_platforms << std::endl;
69    cl_platform_id my_platform = platform_ids[0];
70
71
72    //
73    // Query devices:
74    //
75    cl_device_id device_ids[42];
76    cl_uint num_devices;
77    err = clGetDeviceIDs(my_platform, CL_DEVICE_TYPE_ALL, 42, device_ids, &num_devices); OPENCL_ERR_CHECK(err);
78    std::cout << "# Devices found: " << num_devices << std::endl;
79    cl_device_id my_device_id = device_ids[0];
80
81    char device_name[64];
82    size_t device_name_len = 0;
83    err = clGetDeviceInfo(my_device_id, CL_DEVICE_NAME, sizeof(char)*63, device_name, &device_name_len); OPENCL_ERR_CHECK(err);
84    std::cout << "Using the following device: " << device_name << std::endl;
85
86    //
87    // Create context:
88    //
89    cl_context my_context = clCreateContext(0, 1, &my_device_id, NULL, NULL, &err); OPENCL_ERR_CHECK(err);
90
91
92    //
93    // create a command queue for the device:
94    //
95    cl_command_queue my_queue = clCreateCommandQueueWithProperties(my_context, my_device_id, 0, &err); OPENCL_ERR_CHECK(err);
96
97
98
99    //
100   /////////////////////////// Part 2: Create a program and extract kernels ////////////////////////////////////
101   //
102
103   Timer timer;
104   timer.reset();
105
106   //
107   // Build the program:
108   //
109   size_t source_len = std::string(my_opencl_program).length();
110   cl_program prog = clCreateProgramWithSource(my_context, 1, &my_opencl_program, &source_len, &err);OPENCL_ERR_CHECK(err);
111   err = clBuildProgram(prog, 0, NULL, NULL, NULL, NULL);
112
113   //
114   // Print compiler errors if there was a problem:
115   //
116   if (err != CL_SUCCESS) {
117
118     char *build_log;
119     size_t ret_val_size;
```

```
120       err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, 0, NULL, &
              ret_val_size);
121       build_log = (char *)malloc(sizeof(char) * (ret_val_size+1));
122       err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, ret_val_size,
              build_log, NULL);
123       build_log[ret_val_size] = '\0'; // terminate string
124       std::cout << "Log: " << build_log << std::endl;
125       free(build_log);
126       std::cout << "OpenCL program sources: " << std::endl << my_opencl_program << std::endl;
127       return EXIT_FAILURE;
128     }
129
130     //
131     // Extract the only kernel in the program:
132     //
133     cl_kernel my_kernel = clCreateKernel(prog, "dotProduct", &err); OPENCL_ERR_CHECK(err);
134
135     std::cout << "Time to compile and create kernel: " << timer.get() << std::endl;
136
137
138     //
139     /////////////////////////// Part 3: Create memory buffers
            ////////////////////////////////////
140     //
141
142     //
143     // Set up buffers on host:
144     //
145     cl_uint vector_size = N;
146     std::vector<ScalarType> x(vector_size, 1.0);
147     std::vector<ScalarType> y(vector_size, 2.0);
148     std::vector<ScalarType> partial(vector_size, 0.0);
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,
            vector_size * sizeof(ScalarType), &(x[0]), &err); OPENCL_ERR_CHECK(err);
154     cl_mem ocl_y       = clCreateBuffer(my_context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
            vector_size * sizeof(ScalarType), &(y[0]), &err); OPENCL_ERR_CHECK(err);
155     cl_mem ocl_partial = clCreateBuffer(my_context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
            256 * sizeof(ScalarType), &(y[0]), &err); OPENCL_ERR_CHECK(err);
156
157
158
159     //
160     /////////////////////////// Part 4: Run kernel ////////////////////////////////////
161     //
162     size_t  local_size = 256;
163     size_t global_size = 256*256;
164
165     //
166     // Set kernel arguments:
167     //
168     err = clSetKernelArg(my_kernel, 0, sizeof(cl_mem),  (void*)&ocl_x); OPENCL_ERR_CHECK(err);
169     err = clSetKernelArg(my_kernel, 1, sizeof(cl_mem),  (void*)&ocl_y); OPENCL_ERR_CHECK(err);
170     err = clSetKernelArg(my_kernel, 2, sizeof(cl_mem),  (void*)&ocl_partial); OPENCL_ERR_CHECK
            (err);
171     err = clSetKernelArg(my_kernel, 3, sizeof(cl_uint), (void*)&vector_size); OPENCL_ERR_CHECK
            (err);
172
173     //
174     // Enqueue kernel in command queue:
175     //
176     err = clEnqueueNDRangeKernel(my_queue, my_kernel, 1, NULL, &global_size, &local_size, 0,
            NULL, NULL); OPENCL_ERR_CHECK(err);
177
178     // wait for all operations in queue to finish:
179     err = clFinish(my_queue); OPENCL_ERR_CHECK(err);
180
181
182     //
```

```
183   ///////////////////////////// Part 5: Get data from OpenCL buffer
          ///////////////////////////////////
184   //
185
186   err = clEnqueueReadBuffer(my_queue, ocl_partial, CL_TRUE, 0, sizeof(ScalarType) * 256, &(
          partial[0]), 0, NULL, NULL); OPENCL_ERR_CHECK(err);
187
188   ScalarType dot = 0;
189   for (size_t i=0; i<256; ++i) dot += partial[i];
190
191   std::cout << "Result of OpenCL: " << dot << std::endl;
192
193   //
194   // cleanup
195   //
196   clReleaseMemObject(ocl_x);
197   clReleaseMemObject(ocl_y);
198   clReleaseMemObject(ocl_partial);
199   clReleaseProgram(prog);
200   clReleaseCommandQueue(my_queue);
201   clReleaseContext(my_context);
202
203   return 0;
204 }
205
206
207
208 //
209 // CUDA version
210 //
211 #define CUCL_KERNEL __global__                                  // __kernel
212 #define CUCL_GLOBALMEM                                             // __global
213 #define CUCL_LOCALMEM __shared__                                  // __local
214 #define CUCL_GLOBALID0 blockDim.x *blockIdx.x + threadIdx.x // get_global_id(0)
215 #define CUCL_GLOBALSIZE0 gridDim.x *blockDim.x                  // get_global_size(0)
216 #define CUCL_GROUPID0 blockIdx.x
217 #define CUCL_LOCALSIZE0 blockDim.x                              // get_local_size(0)
218 #define CUCL_LOCALID0 threadIdx.x                               // get_local_id(0)
219 #define CUCL_BARRIER __syncthreads()
220
221 #define STRINGIFY(...) mFn2(__VA_ARGS__)
222 #define mFn2(ANS) ANS
223
224 #include "dot.cucl"
225 // STRINGIFY( CUCL_KERNEL void dotProduct(
226 //   CUCL_GLOBALMEM double *x,
227 //   CUCL_GLOBALMEM double *y,
228 //   CUCL_GLOBALMEM double * partial_result,
229 //   unsigned int N)
230 // {
231 //   CUCL_LOCALMEM double shared_buf[512]; double thread_sum = 0;
232 //   for (int i = CUCL_GLOBALID0; i < N; i += CUCL_GLOBALSIZE0)
233 //     thread_sum += x[i]* y[i];
234
235 //   shared_buf[CUCL_LOCALID0] = thread_sum;
236 //   for (int stride = CUCL_LOCALSIZE0 / 2; stride > 0; stride /= 2) {
237 //     CUCL_BARRIER;
238 //     if (CUCL_LOCALID0 < stride)
239 //       shared_buf[CUCL_LOCALID0] += shared_buf[CUCL_LOCALID0+stride];
240 //   }
241
242 //   CUCL_BARRIER;
243 //   if (CUCL_LOCALID0 == 0)
244 //     partial_result[CUCL_GROUPID0] = shared_buf[0];
245 // } )
246 // TODO for you: Define CUCL preprocessor directives to import proper kernel from dot.cucl
247
248 int dot_cuda(unsigned int N) {
249
250   double *cuda_x, *cuda_y, *cuda_partial;
251   cudaMalloc(&cuda_x, N * sizeof(double));
252   cudaMalloc(&cuda_y, N * sizeof(double));
```

```
253     cudaMalloc(&cuda_partial, 256*sizeof(double));
254
255     // create x, y, and temporary vectors
256     std::vector<double> x(N, 1.0), y(N, 2.0), partial(256, 0.0);
257     cudaMemcpy(cuda_x, &x[0], N*sizeof(double), cudaMemcpyHostToDevice);
258     cudaMemcpy(cuda_y, &y[0], N*sizeof(double), cudaMemcpyHostToDevice);
259
260     dotProduct<<<256, 256>>>(cuda_x, cuda_y, cuda_partial, N);
261
262     // get partial results back and sum on CPU
263     cudaMemcpy(&partial[0], cuda_partial, 256*sizeof(double), cudaMemcpyDeviceToHost);
264     double dot = 0;
265     for (size_t i=0; i<256; ++i) dot += partial[i];
266
267     std::cout << "Result of CUDA dot: " << dot << std::endl;
268
269     return 0;
270   }
271
272
273   //
274   // Main execution flow:
275   //
276   int main() {
277
278     unsigned int N = 1000;
279
280     // runtime switches:
281     int use_opencl = 1;
282     int use_cuda = 1;
283
284     if (use_opencl) {
285       std::cout << "Running OpenCL version" << std::endl;
286       dot_opencl(N);
287     }
288
289     if (use_cuda) {
290       std::cout << std::endl << "Running CUDA version" << std::endl;
291       dot_cuda(N);
292     }
293
294     return EXIT_SUCCESS;
295   }
```