

Computational Science on Many-Core Architectures:

Exercise 9

Viktor Beck, 11713110

December 2022

Task 1: Libraries

Below we can see the comparison of the performances of the different libraries and the custom CUDA and OpenCL implementations. The time was measured such that only the computation of the dot product and the addition / subtraction of the vectors x and y is taken into account.

It seems that the custom CUDA implementation (from exercise 2) is the fastest but the amount of code written for such a small task is quite a lot in comparison to the few lines which were necessary for ViennaCL. The custom OpenCL implementation has the most lines of code and still performs second worst. In general one could say that the ViennaCL library has the best "effort to performance" ratio - at least in this case.

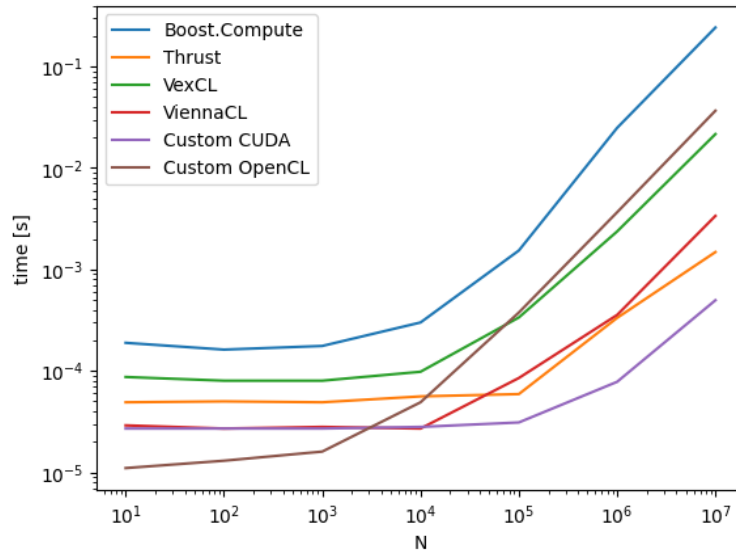


Figure 1: Comparing execution times for different libraries and custom implementations of a dot product for different vector sizes N on the RTX GPU

Task 2: HIP

Somehow, I was not able to install and use HIP therefore, and converted it manually as described in lecture 9 (with good old "ctrl+F"). Also, I could not test it... You can find the respective code snippet in the appendix.

Appendix

Task 1:

Boost.Compute

```
1 // specify use of OpenCL 1.2:
2 #define CL_TARGET_OPENCL_VERSION 120
3 #define CL_MINIMUM_OPENCL_VERSION 120
4
5
6 #include <vector>
7 #include <algorithm>
8 #include <iostream>
9
10 #include "timer.hpp"
11
12 #include <boost/compute/algorithm/transform.hpp>
13 #include <boost/compute/algorithm/inner_product.hpp>
14 #include <boost/compute/container/vector.hpp>
15 #include <boost/compute/functional/math.hpp>
16
17 namespace compute = boost::compute;
18
19 int main()
20 {
21
22     // get default device and setup context
23     compute::device device = compute::system::default_device();
24     compute::context context(device);
25     compute::command_queue queue(context, device);
26
27     Timer timer;
28
29     for (int k = 0; k <=7; k++){
30
31         int N = pow(10, k);
32
33
34
35         // generate data on the host
36         std::vector<float> h_x(N);
37         std::fill(h_x.begin(), h_x.end(), 1);
38         std::vector<float> h_y(N);
39         std::fill(h_y.begin(), h_y.end(), 2);
40
41         // create a vector on the device
42         compute::vector<float> d_x (N, context);
43         compute::vector<float> d_y (N, context);
44
45         // transfer data from the host to the device
46         compute::copy(h_x.begin(), h_x.end(), d_x.begin(), queue);
47         compute::copy(h_y.begin(), h_y.end(), d_y.begin(), queue);
```

```

48         timer.reset();
49
50         // x+y
51         compute::vector<float> x_plus_y(N, context);
52
53         compute::transform(
54             d_x.begin(),
55             d_x.end(),
56             d_y.begin(),
57             x_plus_y.begin(),
58             compute::plus<float>(),
59             queue
60         );
61
62         // x-y
63         compute::vector<float> x_minus_y(N, context);
64         compute::transform(
65             d_x.begin(),
66             d_x.end(),
67             d_y.begin(),
68             x_minus_y.begin(),
69             compute::minus<float>(),
70             queue
71         );
72
73         float res = compute::inner_product(x_plus_y.begin(), x_plus_y.end(),
74             x_minus_y.begin(), 0.f, queue);
75
76         //std::cout << res << std::endl;
77
78         std::cout << timer.get() << std::endl;
79     }
80
81     return 0;
82 }

```

Thrust

```

1  #include <thrust/host_vector.h>
2  #include <thrust/device_vector.h>
3  #include <thrust/sort.h>
4  #include <thrust/transform.h>
5  #include <thrust/inner_product.h>
6  #include <cstdlib>
7
8  #include <vector>
9  #include "timer.hpp"
10
11 int main(void)
12 {
13     Timer timer;
14
15     for (int k = 0; k <= 7; k++)
16     {
17         int N = pow(10, k);
18
19         // x and y on host
20         thrust::host_vector<int> h_x(N);
21         thrust::fill(h_x.begin(), h_x.end(), 1);

```

```

22     thrust::host_vector<int> h_y(N);
23     thrust::fill(h_y.begin(), h_y.end(), 2);
24
25     // transfer data to the device
26     thrust::device_vector<int> d_x = h_x;
27     thrust::device_vector<int> d_y = h_y;
28
29     timer.reset();
30
31     // x+y
32     thrust::device_vector<float> x_plus_y(N);
33
34     thrust::transform(
35         d_x.begin(),
36         d_x.end(),
37         d_y.begin(),
38         x_plus_y.begin(),
39         thrust::plus<float>());
40
41     // x-y
42     thrust::device_vector<float> x_minus_y(N);
43
44     thrust::transform(
45         d_x.begin(),
46         d_x.end(),
47         d_y.begin(),
48         x_minus_y.begin(),
49         thrust::minus<float>());
50
51     float res = thrust::inner_product(x_plus_y.begin(), x_plus_y.end(), x_minus_y
        .begin(), 0.f);
52
53     // std::cout << res << std::endl;
54
55     std::cout << timer.get() << std::endl;
56 }
57
58 return 0;
59 }

```

VexCL

```

1
2 // The following three defines are necessary to pick the correct OpenCL version
   on the machine:
3 #define VEXCL_HAVE_OPENCL_HPP
4 #define CL_HPP_TARGET_OPENCL_VERSION 120
5 #define CL_HPP_MINIMUM_OPENCL_VERSION 120
6
7 #include <iostream>
8 #include <stdexcept>
9 #include <vexcl/vexcl.hpp>
10
11 #include "timer.hpp"
12 #include <vector>
13
14 int main()
15 {
16     vex::Context ctx(vex::Filter::GPU && vex::Filter::DoublePrecision);
17

```

```

18     std::cout << ctx << std::endl; // print list of selected devices
19
20     Timer timer;
21
22     for (int k = 0; k <= 7; k++)
23     {
24
25         int N = pow(10, k);
26
27         std::vector<double> h_x(N, 1.0), h_y(N, 2.0);
28
29         vex::vector<double> d_x(ctx, h_x);
30         vex::vector<double> d_y(ctx, h_y);
31
32
33         timer.reset();
34
35         vex::vector<double> x_plus_y = d_x + d_y;
36         vex::vector<double> x_minus_y = d_x - d_y;
37
38         vex::Reductor<double, vex::SUM> sum(ctx);
39         double res = sum(x_plus_y * x_minus_y);
40
41         // std::cout << res << std::endl;
42
43         std::cout << timer.get() << std::endl;
44     }
45     return 0;
46 }

```

ViennaCL

```

1
2 #include <vector>
3 #include <iostream>
4 #include "timer.hpp"
5
6 #define VIENNAACL_WITH_CUDA
7
8 #include "viennacl/vector.hpp"
9 #include "viennacl/linalg/inner_prod.hpp"
10
11 int main()
12 {
13     Timer timer;
14
15     for (int k = 0; k <= 7; k++)
16     {
17
18         int N = pow(10, k);
19
20         viennacl::vector<double> x = viennacl::scalar_vector<double>(N, 1.0);
21         viennacl::vector<double> y = viennacl::scalar_vector<double>(N, 2.0);
22
23         timer.reset();
24
25         viennacl::vector<double> x_plus_y = x + y;
26         viennacl::vector<double> x_minus_y = x - y;
27
28         double res = viennacl::linalg::inner_prod(x_plus_y, x_minus_y);

```

```

29         // std::cout << res << std::endl;
30
31         std::cout << timer.get() << std::endl;
32     }
33
34     return EXIT_SUCCESS;
35 }

```

Task 2:

```

1  #include "poisson2d.hpp"
2  #include "timer.hpp"
3  #include <algorithm>
4  #include <iostream>
5  #include <stdio.h>
6
7  // y = A * x
8  __global__ void hip_csr_matvec_product(int N, int *csr_rowoffsets,
9                                          int *csr_colindices, double *csr_values,
10                                         double *x, double *y)
11  {
12      for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i +=
13            hipBlockDim_x * hipGridDim_x)
14      {
15          double sum = 0;
16          for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++)
17          {
18              sum += csr_values[k] * x[csr_colindices[k]];
19          }
20          y[i] = sum;
21      }
22
23  // x <- x + alpha * y
24  __global__ void hip_vecadd(int N, double *x, double *y, double alpha)
25  {
26      for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i +=
27            hipBlockDim_x * hipGridDim_x)
28          x[i] += alpha * y[i];
29  }
30
31  // x <- y + alpha * x
32  __global__ void hip_vecadd2(int N, double *x, double *y, double alpha)
33  {
34      for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i +=
35            hipBlockDim_x * hipGridDim_x)
36          x[i] = y[i] + alpha * x[i];
37  }
38
39  // result = (x, y)
40  __global__ void hip_dot_product(int N, double *x, double *y, double *result)
41  {
42      __shared__ double shared_mem[512];
43
44      double dot = 0;
45      for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i +=
46            hipBlockDim_x * hipGridDim_x)
47      {
48          dot += x[i] * y[i];
49      }

```

```

46     }
47
48     shared_mem[hipThreadIdx_x] = dot;
49     for (int k = hipBlockDim_x / 2; k > 0; k /= 2)
50     {
51         __syncthreads();
52         if (hipThreadIdx_x < k)
53         {
54             shared_mem[hipThreadIdx_x] += shared_mem[hipThreadIdx_x + k];
55         }
56     }
57
58     if (hipThreadIdx_x == 0)
59         atomicAdd(result, shared_mem[0]);
60 }
61
62 //////////////// CG KERNEL 1: ///////////////////
63
64 __global__ void hip_cg_1(int N,
65                          double alpha,
66                          double beta,
67                          double *x,
68                          double *r,
69                          double *p,
70                          double *Ap,
71                          double *result_rr)
72 {
73
74     __shared__ double shared_mem[512];
75     double dot = 0;
76
77     for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i +=
78         hipBlockDim_x * hipGridDim_x)
79     {
80         // line 2,3,4: get x, r, p
81         x[i] = x[i] + alpha * p[i];
82         r[i] = r[i] - alpha * Ap[i];
83         p[i] = r[i] + beta * p[i];
84
85         // line 6: get dot(r,r)
86         dot += r[i] * r[i];
87     }
88
89     __syncthreads();
90     shared_mem[hipThreadIdx_x] = dot;
91
92     for (int k = hipBlockDim_x / 2; k > 0; k /= 2)
93     {
94         __syncthreads();
95         if (hipThreadIdx_x < k)
96         {
97             shared_mem[hipThreadIdx_x] += shared_mem[hipThreadIdx_x + k];
98         }
99     }
100
101     if (hipThreadIdx_x == 0)
102     {
103         atomicAdd(result_rr, shared_mem[0]);
104         // printf("%g", r[0]);
105     }
106 }

```

```

107 //////////////// CG KERNEL 2: ////////////////
108 __global__ void hip_cg_2(int N,
109                          int *csr_rowoffsets,
110                          int *csr_colindices,
111                          double *csr_values,
112                          double *p,
113                          double *Ap,
114                          double *result1,
115                          double *result2)
116 {
117
118     __shared__ double shared_mem1[512];
119     __shared__ double shared_mem2[512];
120
121     for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i +=
         hipBlockDim_x * hipGridDim_x)
122     {
123         // line 5: get Ap
124         double sum = 0;
125         for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++)
126         {
127             sum += csr_values[k] * p[csr_colindices[k]];
128         }
129         Ap[i] = sum;
130     }
131
132     // line 6: get dot(Ap,Ap) and dot(p,Ap)
133     double dot1 = 0;
134     double dot2 = 0;
135     for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i +=
         hipBlockDim_x * hipGridDim_x)
136     {
137         dot1 += Ap[i] * Ap[i];
138         dot2 += p[i] * Ap[i];
139     }
140
141     shared_mem1[hipThreadIdx_x] = dot1;
142     shared_mem2[hipThreadIdx_x] = dot2;
143     for (int k = hipBlockDim_x / 2; k > 0; k /= 2)
144     {
145         __syncthreads();
146         if (hipThreadIdx_x < k)
147         {
148             shared_mem1[hipThreadIdx_x] += shared_mem1[hipThreadIdx_x + k];
149             shared_mem2[hipThreadIdx_x] += shared_mem2[hipThreadIdx_x + k];
150         }
151     }
152
153     if (hipThreadIdx_x == 0)
154     {
155         atomicAdd(result1, shared_mem1[0]);
156         atomicAdd(result2, shared_mem2[0]);
157     }
158 }
159
160 /** Implementation of the conjugate gradient algorithm.
161  *
162  * The control flow is handled by the CPU.
163  * Only the individual operations (vector updates, dot products, sparse
164  * matrix-vector product) are transferred to hip kernels.
165  *
166  * The temporary arrays p, r, and Ap need to be allocated on the GPU for use

```



```

167 * with hip. Modify as you see fit.
168 */
169 void conjugate_gradient(int N, // number of unknowns
170                        int *csr_rowoffsets, int *csr_colindices,
171                        double *csr_values, double *rhs, double *solution)
172 //, double *init_guess) // feel free to add a nonzero initial guess as needed
173 {
174     // initialize timer
175     Timer timer;
176
177     // clear solution vector (it may contain garbage values):
178     std::fill(solution, solution + N, 0);
179
180     // initialize work vectors:
181     double alpha, beta, residual_norm_squared, dot_pAp, dot_ApAp;
182     double *hip_solution, *hip_p, *hip_r, *hip_Ap, *hip_scalar, *hip_dot_pAp, *
        hip_dot_ApAp;
183     hipMalloc(&hip_p, sizeof(double) * N);
184     hipMalloc(&hip_r, sizeof(double) * N);
185     hipMalloc(&hip_Ap, sizeof(double) * N);
186     hipMalloc(&hip_solution, sizeof(double) * N);
187     hipMalloc(&hip_scalar, sizeof(double));
188
189     hipMalloc(&hip_dot_ApAp, sizeof(double));
190     hipMalloc(&hip_dot_pAp, sizeof(double));
191
192     hipMemcpy(hip_p, rhs, sizeof(double) * N, hipMemcpyHostToDevice);
193     hipMemcpy(hip_r, rhs, sizeof(double) * N, hipMemcpyHostToDevice);
194     hipMemcpy(hip_solution, solution, sizeof(double) * N, hipMemcpyHostToDevice);
195
196     // get residual_norm_squared
197     const double zero = 0;
198     hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice);
199     hipLaunchKernelGGL(hip_dot_product, 512, 512, 0,0, N, hip_r, hip_r, hip_scalar)
200     ;
201     hipMemcpy(&residual_norm_squared, hip_scalar, sizeof(double),
        hipMemcpyDeviceToHost);
202
203     double initial_residual_squared = residual_norm_squared;
204
205     // line 1: get alpha0, beta0, Ap0
206     hipLaunchKernelGGL(hip_csr_matvec_product, 512, 512, 0,0, N, csr_rowoffsets,
        csr_colindices, csr_values, hip_p, hip_Ap);
207
208     hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice);
209     hipLaunchKernelGGL(hip_dot_product, 512, 512, 0,0, N, hip_p, hip_Ap, hip_scalar)
210     ;
211     hipMemcpy(&dot_pAp, hip_scalar, sizeof(double), hipMemcpyDeviceToHost);
212     alpha = residual_norm_squared / dot_pAp;
213
214     hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice);
215     hipLaunchKernelGGL(hip_dot_product, 512, 512, 0,0, N, hip_Ap, hip_Ap,
        hip_scalar);
216     hipMemcpy(&dot_ApAp, hip_scalar, sizeof(double), hipMemcpyDeviceToHost);
217
218     beta = (alpha * alpha * dot_ApAp - residual_norm_squared) /
        residual_norm_squared;
219
220     int iters = 0;
221     hipDeviceSynchronize();

```

```

222 timer.reset();
223 while (1)
224 {
225
226     hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice);
227
228     // std::cout << alpha << ", " << residual_norm_squared << std::endl;
229     hipLaunchKernelGGL(hip_cg_1, 512, 512, 0,0, N, alpha, beta, hip_solution,
230         hip_r, hip_p, hip_Ap, hip_scalar);
231
232     hipMemcpy(&residual_norm_squared, hip_scalar, sizeof(double),
233         hipMemcpyDeviceToHost);
234
235     // std::cout << residual_norm_squared << std::endl;
236
237     hipMemcpy(hip_dot_ApAp, &zero, sizeof(double), hipMemcpyHostToDevice);
238     hipMemcpy(hip_dot_pAp, &zero, sizeof(double), hipMemcpyHostToDevice);
239
240     // std::cout << dot_ApAp << ", " << dot_pAp << ", " << residual_norm_squared
241         << std::endl;
242
243     hipLaunchKernelGGL(hip_cg_2, 512, 512, 0,0, N, csr_rowoffsets, csr_colindices
244         , csr_values,
245         hip_p, hip_Ap, hip_dot_ApAp, hip_dot_pAp);
246
247     hipMemcpy(&dot_ApAp, hip_dot_ApAp, sizeof(double), hipMemcpyDeviceToHost);
248     hipMemcpy(&dot_pAp, hip_dot_pAp, sizeof(double), hipMemcpyDeviceToHost);
249     // std::cout << alpha << ", " << residual_norm_squared << std::endl;
250
251     // line 7:
252     alpha = residual_norm_squared / dot_pAp;
253
254     // line 8:
255     beta = (alpha * alpha * dot_ApAp - residual_norm_squared) /
256         residual_norm_squared;
257
258     // std::cout << dot_ApAp << ", " << dot_pAp << std::endl;
259
260     // check for convergence
261     if (std::sqrt(residual_norm_squared / initial_residual_squared) < 1e-6)
262     {
263         break;
264     }
265
266     if (iters > 10000)
267         break; // solver didn't converge
268     ++iters;
269 }
270 hipMemcpy(solution, hip_solution, sizeof(double) * N, hipMemcpyDeviceToHost);
271
272 hipDeviceSynchronize();
273 /*
274 std::cout << "Time elapsed: " << timer.get() << " (" << timer.get() / iters <<
275     " per iteration)" << std::endl;
276
277 if (iters > 10000)
278     std::cout << "Conjugate Gradient did NOT converge within 10000 iterations"
279         << std::endl;
280 else
281     std::cout << "Conjugate Gradient converged in " << iters << " iterations."
282         << std::endl;
283 */

```

```

278
279     std::cout << timer.get() / iters << ", " << std::endl;
280
281     hipFree(hip_p);
282     hipFree(hip_r);
283     hipFree(hip_Ap);
284     hipFree(hip_solution);
285     hipFree(hip_scalar);
286 }
287
288 /** Solve a system with 'points_per_direction * points_per_direction' unknowns
289  */
290 void solve_system(int points_per_direction)
291 {
292
293     int N = points_per_direction *
294           points_per_direction; // number of unknowns to solve for
295
296     // std::cout << "Solving Ax=b with " << N << " unknowns." << std::endl;
297
298     //
299     // Allocate CSR arrays.
300     //
301     // Note: Usually one does not know the number of nonzeros in the system matrix
302     // a-priori.
303     //      For this exercise, however, we know that there are at most 5 nonzeros
304     //      per row in the system matrix, so we can allocate accordingly.
305     //
306     int *csr_rowoffsets = (int *)malloc(sizeof(double) * (N + 1));
307     int *csr_colindices = (int *)malloc(sizeof(double) * 5 * N);
308     double *csr_values = (double *)malloc(sizeof(double) * 5 * N);
309
310     int *hip_csr_rowoffsets, *hip_csr_colindices;
311     double *hip_csr_values;
312     //
313     // fill CSR matrix with values
314     //
315     generate_fdm_laplace(points_per_direction, csr_rowoffsets, csr_colindices,
316                          csr_values);
317
318     //
319     // Allocate solution vector and right hand side:
320     //
321     double *solution = (double *)malloc(sizeof(double) * N);
322     double *rhs = (double *)malloc(sizeof(double) * N);
323     std::fill(rhs, rhs + N, 1);
324
325     //
326     // Allocate hip-arrays //
327     //
328     hipMalloc(&hip_csr_rowoffsets, sizeof(double) * (N + 1));
329     hipMalloc(&hip_csr_colindices, sizeof(double) * 5 * N);
330     hipMalloc(&hip_csr_values, sizeof(double) * 5 * N);
331     hipMemcpy(hip_csr_rowoffsets, csr_rowoffsets, sizeof(double) * (N + 1),
332              hipMemcpyHostToDevice);
332     hipMemcpy(hip_csr_colindices, csr_colindices, sizeof(double) * 5 * N,
333              hipMemcpyHostToDevice);
333     hipMemcpy(hip_csr_values, csr_values, sizeof(double) * 5 * N,
334              hipMemcpyHostToDevice);
334
335     //
336     // Call Conjugate Gradient implementation with GPU arrays

```

```

337 //
338 conjugate_gradient(N, hip_csr_rowoffsets, hip_csr_colindices, hip_csr_values,
    rhs, solution);
339
340 //
341 // Check for convergence:
342 //
343 /*
344 double residual_norm = relative_residual(N, csr_rowoffsets, csr_colindices,
    csr_values, rhs, solution);
345 std::cout << "Relative residual norm: " << residual_norm
346     << " (should be smaller than 1e-6)" << std::endl;
347 */
348 hipFree(hip_csr_rowoffsets);
349 hipFree(hip_csr_colindices);
350 hipFree(hip_csr_values);
351 free(solution);
352 free(rhs);
353 free(csr_rowoffsets);
354 free(csr_colindices);
355 free(csr_values);
356 }
357
358 int main()
359 {
360     std::vector<int> N_vec = {10, 25, 50, 75, 100, 250, 500, 750, 1000};
361     for (const auto &N : N_vec)
362     {
363         solve_system(N); // solves a system with 100*100 unknowns
364     }
365
366     return EXIT_SUCCESS;
367 }

```