INSTITUTE OF MICROELECTRONICS

360.252 Computational Science on Many-Core Architectures

Exercise 7

Author:
Peter Holzner, 01426733

Submission: December 8, 2020

Contents

---

1 Task 1: Dot Product With OpenCL

Code listings for this task:

- OpenCL dot product kernel: Listing 1
- OpenCL benchmark: Listing 2
- CUDA benchmark + kernel: Listing 3

1.1 Implementation

I've already aired my frustrations with this exercise in the chat. Considering Nvidias soft lock on OpenCL (still stuck on OCL v1.2), I thought it only to be fair to compare my OpenCL implementation to a similar CUDA kernel - also that's really the fairest benchmark anyway (same conditions for everyone). That means, that I had to rely on not using advanced synchronization constructs such warp shuffles and work group reduction (OCL >2.0) respectively. For OpenCL, I was also unable to use atomic reductions for the global reduction due to the Intel OpenCL platform not supporting the necessary 64bit atomic extensions. Remnants of my attempts (atomic_add, different dot-kernel versions) can still be found in my code. That did turn into a bit of a rant anyway, sorry!

I decided to only do the first part of the dot product on the OCL device. The results of each workgroup are written to one entry of the result array. The final results is then calculated on the CPU via summing across all workgroup results. Since we have already written an identical benchmark for CUDA in Exercise 2, I simply reused that code (and fixed it, since mine did not work properly back then).

1.2 Runtime comparisons

In Fig. 1, the total runtimes (dot product on device + reduction across work groups/blocks on host) for the 3 different implementations are shown on the left side. The OpenCL implementations completely outperform the CUDA implementation for vectors of size $N < 1e7$ - by multiple orders of magnitude. Ironically, the OCL-GPU kernel performs the best across the board up until the largest vector tested at least. I tried with larger N but the program simply exited without any messages - I assume that I simply wasn't able to allocate enough memory and did not want to cause any further problems with the device.

OpenCL does not seem to suffer from as much overhead as CUDA when starting a kernel, so OpenCL seems better to me than CUDA for smaller problem sizes and/or programs where many "smaller" kernels need to be called (for whatever reason). Apart from that, OpenCL performed better than CUDA for larger vectors as well, although I'm hesitant to make a definitive statement based on these limited benchmarks. OpenCL is much more complicated, unintuitive and annoying to

program for though - especially when one is bound to OCL v1.2 and needs to write code for multiple devices (though you can of course make your code more modular by using precompiler directives and enabling/disabling certain code parts). At least some of the aforementioned overhead that CUDA suffers from is probably hidden in some of the additional setup steps needed for the OpenCL kernel execution - both in terms of consideration needed from the programmer and in terms of the observed runtime.

The right side of Fig. 1 shows the breakdown of the two stages of the dot-product implementation. The final reduction on the host (CPU in both cases in plain C++) is clearly not the bottleneck - or at least not more than the kernel execution.

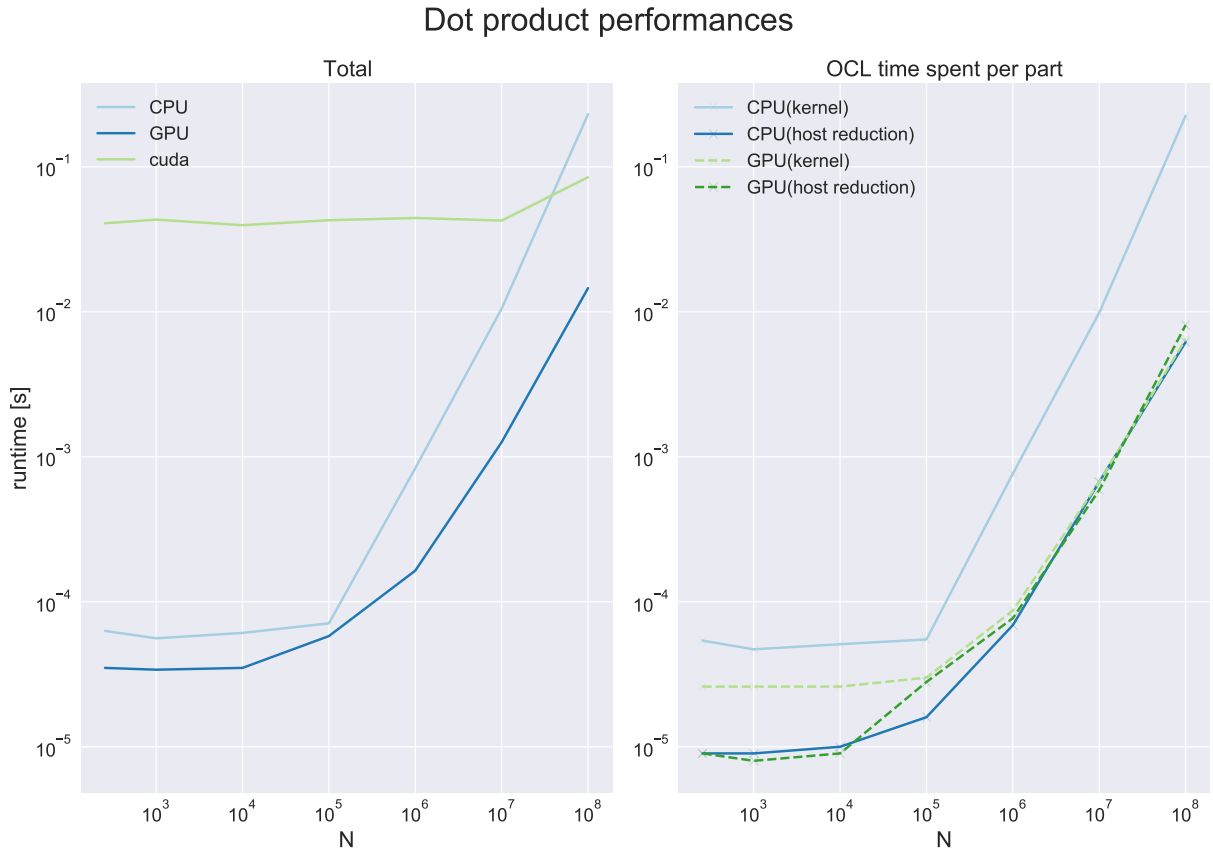

Fig. 1. Runtimes of the different vector dot product based kernels.

2 Task 2: OpenCL Sparse Matrix-Vector Product Kernels

Code listings for this task:

- OpenCL Matrix-Vector product kernel: Listing 4
- OpenCL Matrix-Vector product benchmark: Listing 5

Both kernels work and deliver the correct results as can be checked by running the benchmark code. The output looks as follows for both kernels:

Run output

```
# Benchmarking finite difference matrix
Using the following device: GeForce GTX 1080
From OCL benchmark: 0.000331
Reference:
2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | ... | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2
OpenCL:
2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | ... | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2
Difference between the two: 0 (check: 1
Time for ref product: 1.6e-05
Time for OCL product: 0.000331
# Benchmarking special matrix
Using the following device: GeForce GTX 1080
From OCL benchmark: 0.001752
Reference:
-256 | -242 | -256 | -256 | -255 | -178 | -253 | -252 | -256 | -149 | ... | -256 | -256 | -229 | -233 | -255 | -255 | -251 | -228 | -252 | -247
OpenCL:
-256 | -242 | -256 | -256 | -255 | -178 | -253 | -252 | -256 | -149 | ... | -256 | -256 | -229 | -233 | -255 | -255 | -251 | -228 | -252 | -247
Difference between the two: 0 (check: 1
Time for ref product: 0.000219
Time for OCL product: 0.001752
Data:
Runtimes in csv form can be found here
https://gtx1080.360252.org/2020/ex7/ph_data_GPU.csv
```

Fig. 2. Sample output of the benchmark running the "slow" kernel.

My kernels did however not perform too well in terms of runtime, as can (unfortunately) be seen in Fig. 3. The GPU did not perform well for me in this case - it never beat the host computation. The OpenCL-CPU kernel did however outperform the reference host computation atleast for larger problem sizes or the denser matrix (with more nonzero entries per row).

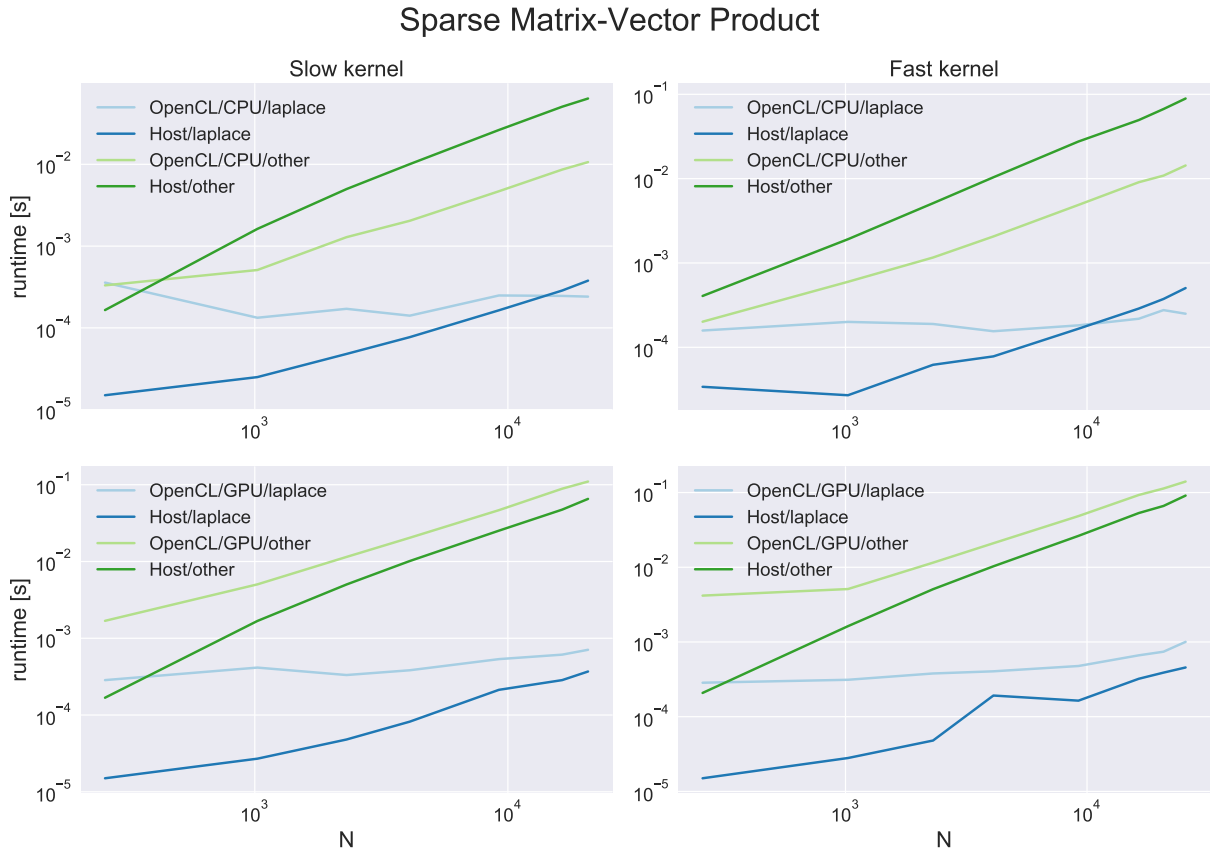

Fig. 3. Runtimes of the Sparse Matrix-Vector products. The left side shows the "slow" kernel, the right side the "faster" version. The matrix type is denoted by either "laplace" (blue lines) or "other" (green lines). The top two show the OpenCL-CPU runtimes, whereas the bottom shows the GPU runtimes.

In Fig. 4, the speedup of the "faster"[1] kernel version is plotted as compared to recorded runtime of the "slower" kernel. On the CPU, the faster kernel did actually improve the performance a little bit. One the GPU, I would classify it to be a tie.

[1]The slower kernel is called "ocl_csr_matvec". The faster kernel is called "ocl_csr_matvec_fast". Very descriptive, I know.

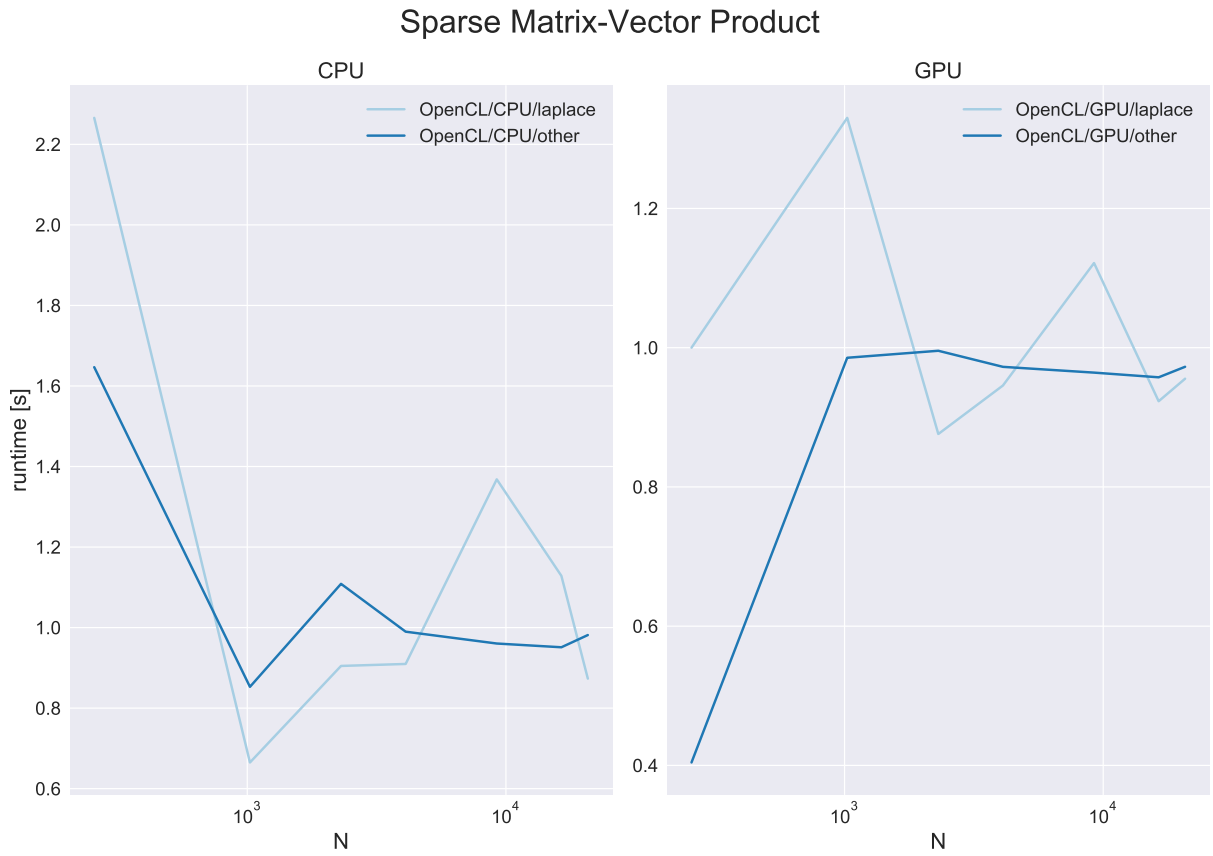

Fig. 4. Speedup of the "faster" kernel compared to the "slower" kernel on the CPU and GPU respectively.

3 Code and Kernels

Listings

Listing 1: Ex7.1: Dot products with OpenCL - Kernels

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#pragma OPENCL EXTENSION cl_khr_int64_base_atomics : enable
#pragma OPENCL EXTENSION cl_khr_int64_extended_atomics : enable

void my_atomic_add(volatile __global double *p, double val) {
  volatile __global ulong* address_as_ul = (volatile __global ulong *) p;
  volatile ulong old = *address_as_ul, assumed;
  ulong val_as_ul =  (ulong) val;
  do  {
    assumed = old;
    old = atomic_add(address_as_ul, val_as_ul)
  } while (assumed != old);
};

__kernel void vec_add(__global double *x, __global double *y, unsigned int N) {
  for (unsigned int i = get_global_id(0); i < N; i += get_global_size(0))
    x[i] += y[i];
};

__kernel void xDOTy(__global double *result,
                    __global double *x,
                    __global double *y,
                    __local cache, uint N) {
  uint gid = get_global_id(0);
  uint lid = get_local_id(0);
  uint stride = get_global_size(0);
  __local double cache[128];
  double tmp = 0.0;
  for (uint i = gid; i < N; i += stride)
    tmp += x[i] * y[i];
  cache[lid] = tmp;

  for (int i = get_local_size(0) / 2; i > 0; i /= 2) {
    barrier(CLK_LOCAL_MEM_FENCE);
    if (lid < i)
      cache[lid] += cache[lid + i];
  }
  if (lid==0)
    result[get_group_id(0)] = cache[lid];
};



// __kernel void xDOTy(__global atomic_double *result, __global double *x,
//                     __global double *y, uint N) {
//   uint gid = get_global_id(0);
//   uint lid = get_local_id(0);
//   uint stride = get_global_size(0);
//   double dot = 0.0;
//   for (uint i = tid; i < N; i += stride)
//     dot += x[i] * y[i];

//   // need to add to signature: __local double *cache,
//   // for (int i = get_local_size(0) / 2; i > 0; i /= 2) {
//   //   barrier(CLK_LOCAL_MEM_FENCE);
//   //   if (lid < i)
//   //     cache[lid] += cache[lid + i];
//   // }
//   double val = work_group_reduce_add(gid < N ? dot : 0.);

//   if (lid == 0) {
//     atomic_add(result, val);
//   }
//   // // atomic flag version
//   // // need to add: __global volatile atomic_flag *lock,
//   // if (lid == 0) {
//   //   while (!atomic_flag_test_and_set(lock)){}
//   //   *result += cache[0];
//   //   atomic_flag_clear(lock;)
//   // }
```

```
// };

// __kernel void xDOTy(__global atomic_double *result, __global double *x,
//                     __global double *y, __local double *cache,
//                     __global bool *lock, uint N) {
//   uint gid = get_global_id(0);
//   uint lid = get_local_id(0);
//   uint stride = get_global_size(0);
//   double dot = 0.0;
//   for (uint i = tid; i < N; i += stride)
//     dot += x[i] * y[i];
//   #cache[lid] = dot;

//   // for (int i = get_local_size(0) / 2; i > 0; i /= 2) {
//   //   barrier(CLK_LOCAL_MEM_FENCE);
//   //   if (lid < i)
//   //     cache[lid] += cache[lid + i];
//   // }
//   double val = work_group_reduce_max( gid < length ? input[globalId] : -INFINITY);

//   if (lid == 0) {
//     while (*lock) {}

//     *result += cache[0];
//   }
// };
```

Listing 2: Ex7.1: Dot products with OpenCL - Benchmark

```
//
// Tutorial for demonstrating a simple OpenCL vector addition kernel
//
// Author: Karl Rupp    rupp@iue.tuwien.ac.at
//
typedef double ScalarType;
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <cmath>
#include <algorithm>
#include <numeric>
#include <stdexcept>

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

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

#define LOCAL_SIZE 128
#define GLOBAL_SIZE 128
#define NUM_TESTS 5

template <typename T>
void printContainer(T container, const int size) {
  std::cout << container[0];
  for (int i = 1; i < size; ++i)
    std::cout << " | " << container[i] ;
  std::cout << std::endl;
}

template <typename T>
void printContainer(T container, const int size, const int only) {
  std::cout << container[0];
  for (int i = 1; i < only; ++i)
      std::cout  << " | " << container[i];
  std::cout << " | ...";
```

```
44    for (int i = size - only; i < size; ++i)
45      std::cout  << " | " << container[i];
46    std::cout << std::endl;
47  }
48
49  double median(std::vector<double>& vec)
50  {
51    size_t size = vec.size();
52    if (size == 0)
53            return 0.;
54    sort(vec.begin(), vec.end());
55    size_t mid = size/2;
56
57    return size % 2 == 0 ? (vec[mid] + vec[mid-1]) / 2 : vec[mid];
58  }
59
60  // const char *my_opencl_program =  "\n"
61  // "#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n"
62  // " \n\n"
63  // "__kernel void xDOTy(__global double *result, \n"
64  // "                    __global double *x,\n"
65  // "                    __global double *y, \n"
66  // "                    __local cache, uint N) {\n"
67  // "  uint gid = get_global_id(0);\n"
68  // "  uint lid = get_local_id(0);\n"
69  // "  uint stride = get_global_size(0);\n"
70  // "  double tmp = 0.0;\n"
71  // "  for (uint i = gid; i < N; i += stride)\n"
72  // "    tmp = x[i] * y[i];\n"
73  // "  cache[lid] = tmp;\n"
74  // "  \n"
75  // "  for (int i = get_local_size(0) / 2; i > 0; i /= 2) {\n"
76  // "    barrier(CLK_LOCAL_MEM_FENCE);\n"
77  // "    if (lid < i)\n"
78  // "      cache[lid] += cache[lid + i];\n"
79  // "  }\n"
80  // "  if (lid==0)\n"
81  // "    result[get_group_id()] = cache[lid]\n"
82  // "};\n";
83
84
85  // "                    __local cache, uint N) {\n"
86
87  std::string my_opencl_program_c =  "\n"
88  "#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n"
89  " \n\n"
90  "__kernel void xDOTy(__global double *result, \n"
91  "                    __global double *x,\n"
92  "                    __global double *y, \n"
93  "                    uint N) {\n"
94  "  uint gid = get_global_id(0);\n"
95  "  uint lid = get_local_id(0);\n"
96  "  uint stride = get_global_size(0);\n"
97  "  __local double cache[128];\n"
98  "  double tmp = 0.0;\n"
99  "  for (uint i = gid; i < N; i += stride)\n"
100 "    tmp += x[i] * y[i];\n"
101 "  cache[lid] = tmp;\n"
102 "  \n"
103 "  for (int i = get_local_size(0) / 2; i > 0; i /= 2) {\n"
104 "    barrier(CLK_LOCAL_MEM_FENCE);\n"
105 "    if (lid < i)\n"
106 "      cache[lid] += cache[lid + i];\n"
107 "  }\n"
108 "  if (lid==0)\n"
109 "    result[get_group_id(0)] = cache[lid];\n"
110 "};\n";
111
112 std::string header = "\n"
113 "#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n"
114 " \n\n";
115 std::string kernel_signature = "__kernel void xDOTy";
```

```
116 std::string kernel_code1 = "(__global double *result, \n"
117 "                    __global double *x,\n"
118 "                    __global double *y, \n"
119 "                    uint N) {\n";
120 std::string kernel_code2 = "  uint gid = get_global_id(0);\n"
121 "  uint lid = get_local_id(0);\n"
122 "  uint stride = get_global_size(0);\n"
123 "   __local double cache[128];\n"
124 "  double tmp = 0.0;\n"
125 "  for (uint i = gid; i < N; i += stride)\n"
126 "    tmp += x[i] * y[i];\n"
127 "  cache[lid] = tmp;\n"
128 "  \n"
129 "  for (int i = get_local_size(0) / 2; i > 0; i /= 2) {\n"
130 "    barrier(CLK_LOCAL_MEM_FENCE);\n"
131 "    if (lid < i)\n"
132 "      cache[lid] += cache[lid + i];\n"
133 "  }\n"
134 "  if (lid==0)\n"
135 "    result[get_group_id(0)] = cache[lid];\n"
136 "};\n";
137
138
139 int main()
140 {
141   cl_int err;
142
143   bool compute = false;
144   bool compile_M = true;
145   std::vector<uint> M_vec{1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
146   if (!compile_M) {
147     M_vec.clear();
148     M_vec.push_back(1);
149   }
150   // std::string target = "GeForce GTX 1080";
151   std::string target = "GPU";
152   std::vector<uint> N_vec{256, 1000, 10000, 100000, 1000000, 10000000, 100000000};
153   uint N_min = N_vec.front();
154   uint N_max = N_vec.back();
155   uint cnt = N_vec.size();
156
157
158   Timer timer;
159   timer.reset();
160
161   //
162   ////////////////////////// Part 1: Set up an OpenCL context with one device
          //////////////////////////////////
163   //
164
165   //
166   // Query platform:
167   //
168   cl_uint num_platforms;
169   cl_platform_id platform_ids[42];   //no more than 42 platforms supported...
170   err = clGetPlatformIDs(42, platform_ids, &num_platforms); OPENCL_ERR_CHECK(err);
171   std::cout << "# Platforms found: " << num_platforms << std::endl;
172
173
174   //
175   // Query devices:
176   //
177   cl_device_id device_ids[42];
178   cl_uint num_devices;
179   char device_name[64];
180   cl_device_id my_device_id;
181   cl_platform_id my_platform;
182   for (int i = 0; i < num_platforms; ++i)
183   {
184     my_platform = platform_ids[i];
185     if (target == "GPU") {
186       err = clGetDeviceIDs(my_platform, CL_DEVICE_TYPE_GPU, 42, device_ids, &num_devices);
```

```
187       }
188       else {
189         err = clGetDeviceIDs(my_platform, CL_DEVICE_TYPE_CPU, 42, device_ids, &num_devices);
190       }
191       if (err == CL_SUCCESS)
192         break;
193     }
194     OPENCL_ERR_CHECK(err);
195     std::cout << "# Devices found: " << num_devices << std::endl;
196     my_device_id = device_ids[0];
197
198     size_t device_name_len = 0;
199     err = clGetDeviceInfo(my_device_id, CL_DEVICE_NAME, sizeof(char)*63, device_name, &
          device_name_len); OPENCL_ERR_CHECK(err);
200
201     std::cout << "Using the following device: " << device_name << std::endl;
202
203     //
204     // Create context:
205     //
206     cl_context my_context = clCreateContext(0, 1, &my_device_id, NULL, NULL, &err);
          OPENCL_ERR_CHECK(err);
207
208
209     //
210     // create a command queue for the device:
211     //
212     cl_command_queue my_queue = clCreateCommandQueueWithProperties(my_context, my_device_id,
          0, &err); OPENCL_ERR_CHECK(err);
213
214
215     //
216     /////////////////////////// Part 2: Create a program and extract kernels
          ////////////////////////////////////
217     //
218     //
219     // Build the program:
220     //
221     cl_kernel my_kernel;
222     cl_program prog;
223
224
225     std::fstream csv_compile;
226     std::string csv_compile_name = "ph_data_compile.csv";
227
228     if (compile_M) {
229       csv_compile.open(csv_compile_name, std::fstream::out | std::fstream::trunc);
230       csv_compile << "M;compile_time;create_time" << std::endl;
231     }
232     int m = 1;
233
234     std::string ocl_prog = header + kernel_signature + kernel_code1 + kernel_code2;
235     for (auto& M : M_vec){
236
237       // // To gernerate the M kernels
238       for (; m < M; ++m) {
239         ocl_prog += kernel_signature + std::to_string(m) + kernel_code1 + "uint insert" + std
              ::to_string(m) + "=1;\n" + kernel_code2;
240       }
241       const char * my_opencl_program = ocl_prog.c_str();
242
243       std::vector<double> tmp(NUM_TESTS, 0);
244       for (uint iter = 0; iter < NUM_TESTS; iter++){
245         timer.reset();
246         size_t source_len = std::string(my_opencl_program).length();
247         prog = clCreateProgramWithSource(my_context, 1, &my_opencl_program, &source_len, &err)
              ;OPENCL_ERR_CHECK(err);
248         err = clBuildProgram(prog, 0, NULL, NULL, NULL, NULL);
249         tmp[iter] = timer.get();
250       }
251       double compile_time = median(tmp);
252       //
```

```
253       // Print compiler errors if there was a problem:
254       //
255       if (err != CL_SUCCESS) {
256
257         char *build_log;
258         size_t ret_val_size;
259         err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, 0, NULL, &
              ret_val_size);
260         build_log = (char *)malloc(sizeof(char) * (ret_val_size+1));
261         err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, ret_val_size,
              build_log, NULL);
262         build_log[ret_val_size] = '\0'; // terminate string
263         std::cout << "Log: " << build_log << std::endl;
264         free(build_log);
265         std::cout << "OpenCL program sources: " << std::endl << my_opencl_program << std::endl
              ;
266         return EXIT_FAILURE;
267       }
268       //
269       // Extract the only kernel in the program:
270       //
271       for (uint iter = 0; iter < NUM_TESTS; iter++){
272         timer.reset();
273         my_kernel = clCreateKernel(prog, "xDOTy", &err); OPENCL_ERR_CHECK(err);
274         tmp[iter] = timer.get();
275       }
276       double create_time = median(tmp);
277       std::cout << "Time to compile and create kernel: " << compile_time << std::endl;
278       csv_compile << M << ";"
279                   << compile_time << ";"
280                   << create_time << std::endl;
281     }
282
283
284     // Plan to reuse all these vectors and buffers
285     double y_val = 2., x_val=1.;
286     std::vector<ScalarType> x(N_max, x_val);
287     std::vector<ScalarType> y(N_max, y_val);
288     std::vector<ScalarType> dot_vec(N_max, 0);
289
290     cl_mem ocl_x = clCreateBuffer(my_context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, N_max
          * sizeof(ScalarType), &(x[0]), &err); OPENCL_ERR_CHECK(err);
291     cl_mem ocl_y = clCreateBuffer(my_context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, N_max
          * sizeof(ScalarType), &(y[0]), &err); OPENCL_ERR_CHECK(err);
292     cl_mem ocl_dot = clCreateBuffer(my_context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
          sizeof(ScalarType)*N_max, dot_vec.data(), &err); OPENCL_ERR_CHECK(err);
293
294     //
295     /////////////////////////// Part 3: Create memory buffers
          ///////////////////////////////////
296     //
297
298     std::string csv_name = "ph_data_" + target + ".csv";
299     std::fstream csv;
300     if (compute){
301       std::vector<double> times_cpu(N_vec.size(), 0.);
302       std::vector<double> times_gpu(N_vec.size(), 0.);
303       std::vector<double> times_total(N_vec.size(), 0.);
304       csv.open(csv_name, std::fstream::out | std::fstream::trunc);
305       csv << "N;target;local_size;global_size;ocl_time;cpu_time;total_time;dot" << std::endl;
306       for (auto& N: N_vec){
307
308         cl_uint vector_size = N;
309         size_t  local_size = LOCAL_SIZE;
310         size_t global_size = GLOBAL_SIZE*GLOBAL_SIZE;
311         size_t groups = 1 + int(N/LOCAL_SIZE);
312
313         //
314         /////////////////////////// Part 4: Run kernel ///////////////////////////////////
315         //
316
317
```

```
318         //
319         // Set kernel arguments:
320         //
321         // xDOTy(__global double *result,
322         //                   __global double *x,
323         //                   __global double *y,
324         //                   __local cache, uint N)
325         err = clSetKernelArg(my_kernel, 0, sizeof(cl_mem),  (double*)&ocl_dot);
                OPENCL_ERR_CHECK(err);
326         err = clSetKernelArg(my_kernel, 1, sizeof(cl_mem),  (double*)&ocl_x); OPENCL_ERR_CHECK
                (err);
327         err = clSetKernelArg(my_kernel, 2, sizeof(cl_mem),  (double*)&ocl_y); OPENCL_ERR_CHECK
                (err);
328         // err = clSetKernelArg(my_kernel, 3, sizeof(cl_float) * local_work_size[0], NULL);
                OPENCL_ERR_CHECK(err);
329         err = clSetKernelArg(my_kernel, 3, sizeof(cl_uint), (void*)&vector_size);
                OPENCL_ERR_CHECK(err);
330
331         //
332         // Enqueue kernel in command queue:
333         //
334         std::vector<double> tmp(NUM_TESTS, 0);
335         for (uint iter = 0; iter < NUM_TESTS; iter++){
336           timer.reset();
337           err = clEnqueueNDRangeKernel(my_queue, my_kernel, 1, NULL, &global_size, &local_size
                    , 0, NULL, NULL); OPENCL_ERR_CHECK(err);
338
339           // wait for all operations in queue to finish:
340           err = clFinish(my_queue); OPENCL_ERR_CHECK(err);
341           tmp[iter] = timer.get();
342         }
343         double ocl_time = median(tmp);
344
345         //
346         /////////////////////////// Part 5: Get data from OpenCL buffer
                ///////////////////////////////////
347         //
348
349         // err = clEnqueueReadBuffer(my_queue, ocl_x, CL_TRUE, 0, sizeof(ScalarType) * x.size
                (), &(x[0]), 0, NULL, NULL); OPENCL_ERR_CHECK(err);
350         err = clEnqueueReadBuffer(my_queue, ocl_dot, CL_TRUE, 0, sizeof(ScalarType)*
                vector_size, dot_vec.data(), 0, NULL, NULL); OPENCL_ERR_CHECK(err);
351
352         timer.reset();
353         double dot = std::accumulate(dot_vec.begin(), dot_vec.begin()+groups, 0.);
354         double cpu_time = timer.get();
355         double total_time = ocl_time + cpu_time;
356
357
358         std::cout << std::endl;
359         std::cout << "Result of kernel execution: " << dot;
360         std::cout << " =? " << vector_size*y_val*x_val << " : " << (dot == y_val*x_val*
                vector_size) << std::endl;
361         std::cout << "Runtime: kernel + cpu = total_time" << std::endl;
362         std::cout << ocl_time << " + " << cpu_time << " = " << total_time << std::endl;
363
364         csv << N<< ";"
365             << target << ";"
366             << local_size<< ";"
367             << global_size<< ";"
368             << ocl_time<< ";"
369             << cpu_time<< ";"
370             << total_time << ";"
371             << dot << std::endl;
372
373
374         // std::cout << "Result container:" << std::endl;
375         // printContainer(dot_vec, dot_vec.size(), 10);
376       }
377       csv.close();
378     }
379     if (compile_M) csv_compile.close();
```

```
380     //
381     // cleanup
382     //
383     clReleaseMemObject(ocl_x);
384     clReleaseMemObject(ocl_y);
385     clReleaseMemObject(ocl_dot);
386     clReleaseProgram(prog);
387     clReleaseCommandQueue(my_queue);
388     clReleaseContext(my_context);
389
390     std::cout << "Data: \nRuntimes in csv form can be found here\nhttps://gtx1080.360252.org
            /2020/ex7/" << csv_name << std::endl;
391     std::cout << "Data: \nCompile times in csv form can be found here\nhttps://gtx1080.360252.
            org/2020/ex7/" << csv_compile_name << std::endl;
392
393     std::cout << std::endl;
394     std::cout << "#" << std::endl;
395     std::cout << "# My first OpenCL application finished successfully!" << std::endl;
396     std::cout << "#" << std::endl;
397
398     std::cout << "And here it is:" << std::endl;
399     std::cout << ocl_prog;
400     return EXIT_SUCCESS;
401 }
```

Listing 3: Ex7.1: Dot products with CUDA

```
1  #include <iostream>
2  #include <string>
3  #include <vector>
4  #include "timer.hpp"
5  #include <cmath>
6
7  #define BLOCK_SIZE 256
8
9  __global__ void initKernel1(double* arr, const double value, const size_t N)
10 {
11   const int stride = blockDim.x * gridDim.x;
12   int tid = threadIdx.x +  blockIdx.x * blockDim.x;
13
14   for(; tid < N; tid += stride)
15   {
16     arr[tid] = value;
17   }
18 }
19
20 __global__ void initKernel2(double* arr, double* arr2, const size_t N)
21 {
22   const int stride = blockDim.x * gridDim.x;
23   int tid = threadIdx.x +  blockIdx.x * blockDim.x;
24
25   for(; tid < N; tid += stride)
26   {
27     arr[tid] = tid;
28     arr2[tid] = N - 1 - tid;
29   }
30 }
31
32 __global__ void initKernel3(double* arr, double* arr2, double* arr3, const size_t N)
33 {
34   const int stride = blockDim.x * gridDim.x;
35   int tid = threadIdx.x +  blockIdx.x * blockDim.x;
36
37   for(; tid < N; tid += stride)
38   {
39     arr[tid] = tid;
40     arr2[tid] = N - 1 - tid;
41     arr3[tid] = 0;
42   }
43 }
44
45 __global__ void addKernel(double* x, double* y, double* res, const size_t N)
```

```
46  {
47    const int stride = blockDim.x * gridDim.x;
48    int tid = threadIdx.x +  blockIdx.x * blockDim.x;
49
50    for(; tid < N; tid += stride)
51    {
52      res[tid] = x[tid] + y[tid];
53      //res[tid] += 1;
54    }
55  }
56
57  __global__ void dot_A_1(double* x, double* y, double* block_sums, const size_t N)
58  {
59    uint tid = threadIdx.x + blockDim.x* blockIdx.x;
60    uint stride = blockDim.x* gridDim.x;
61
62    __shared__ double cache[BLOCK_SIZE];
63
64    double tid_sum = 0.0;
65    for (; tid < N; tid += stride)
66    {
67      tid_sum += x[tid] * y[tid];
68    }
69    cache[threadIdx.x] = tid_sum;
70
71    __syncthreads();
72    for (uint i = blockDim.x/2; i != 0; i /=2)
73    {
74      if (threadIdx.x < i) //lower half does smth, rest idles
75      {
76        cache[threadIdx.x] += cache[threadIdx.x + i]; //lower looks up by stride and sums up
77      }
78      __syncthreads();
79    }
80
81    if(threadIdx.x == 0) // cache[0] now contains block_sum
82    {
83      block_sums[blockIdx.x] = cache[0];
84    }
85    __syncthreads();
86  }
87
88  __global__ void dot_A_2(double* block_sums)
89  {
90    int tid = threadIdx.x; // only one block, so this is fine!
91    for (int i = blockDim.x / 2; i >= 1; i /=2) // same principal as above
92    {
93      if ( tid < i )
94      {
95        block_sums[tid] += block_sums[tid + i];
96      }
97      __syncthreads();
98    }
99  }
100
101 __global__ void dot_Atomic(double* x, double* y, double* result, const size_t N)
102 {
103   uint tid = threadIdx.x + blockDim.x* blockIdx.x ;
104   uint stride = blockDim.x* gridDim.x ;
105
106   __shared__ double cache[BLOCK_SIZE];
107
108   double tid_sum = 0.0;
109   for (; tid < N; tid += stride)
110   {
111     tid_sum += x[tid] * y[tid];
112   }
113   tid = threadIdx.x;
114   cache[tid] = tid_sum;
115
116   __syncthreads();
117   for (uint i = blockDim.x/2; i != 0; i /=2)
```

```
118     {
119       __syncthreads();
120       if (tid < i) //lower half does smth, rest idles
121       {
122         cache[tid] += cache[tid + i ]; //lower looks up by stride and sums up
123       }
124     }
125
126     if(tid == 0) // cache[0] now contains block_sum
127     {
128       atomicAdd(result, cache[0]);
129     }
130   }
131
132   int main(void)
133   {
134     const int num_tests = 10;
135     int tests_done = num_tests;
136     const int block_size = BLOCK_SIZE;
137     const int mode = 1;
138     const int option = 2;
139
140     std::cout << "N;blocks;block_size;tests_done;total_time;time_per_test;check" << std::endl;
141     for (size_t N = 256; N <= 1000000000; N*=10)
142     {
143       std::cout << N << std::endl;
144       //int blocks = (int)(N+block_size-1)/block_size;
145       int blocks = block_size;
146       double result = 0.0;
147       double result_true = N;
148       double* h_block_sums = (double *)malloc(sizeof(double) * blocks);
149       double* presult = (double *)malloc(sizeof(double));
150       presult = &result;
151       double* pnull = (double *)malloc(sizeof(double));
152       *pnull = 0.0;
153       double *d_x, *d_y, *d_block_sums, *d_result;
154
155       cudaMalloc(&d_result, sizeof(double));
156       cudaMalloc(&d_x, N*sizeof(double));
157       cudaMalloc(&d_y, N*sizeof(double));
158       cudaMalloc(&d_block_sums, blocks*sizeof(double));
159       cudaDeviceSynchronize();
160
161       int i = 0;
162       Timer timer;
163       double total_time = 0.0;
164
165       initKernel1<<<blocks, block_size>>>(d_block_sums, 0.0, block_size);
166       initKernel1<<<blocks, block_size>>>(d_x, 1.0, N);
167       initKernel1<<<blocks, block_size>>>(d_y, 1.0, N);
168       //initKernel2<<<blocks, block_size>>>(d_x, d_y, N);
169       cudaDeviceSynchronize();
170
171       timer.reset();
172       for (i = 0; i<num_tests; i++)
173       {
174         tests_done = i+1;
175         if (option == 1)
176         {
177           dot_A_1<<<blocks, block_size>>>(d_x, d_y, d_block_sums,  N);
178           dot_A_2<<<1, block_size>>>(d_block_sums);
179           cudaDeviceSynchronize();
180         }
181         if (option == 2)
182         {
183           dot_A_1<<<blocks, block_size>>>(d_x, d_y, d_block_sums,  N);
184           cudaMemcpy(h_block_sums, d_block_sums, blocks*sizeof(double), cudaMemcpyDeviceToHost
                ;
185           //std::cout << h_block_sums[0] << " =? " << h_block_sums[blocks-1] << std::endl;
186           cudaDeviceSynchronize();
187           result = 0.0;
188           for (int j = 0; j < blocks; j+=1)
```

```
189           {
190             result += h_block_sums[j];
191           }
192         }
193         if (option == 3)
194         {
195           cudaMemcpy(d_result, pnull, sizeof(double), cudaMemcpyHostToDevice);
196           dot_Atomic<<<blocks, block_size>>>(d_x, d_y, d_result, N);
197         }
198
199         //std::cout << "(" << i+1 << ") Elapsed: " << runtime << " s" << std::endl;
200       }
201       total_time = timer.get();
202       size_t check;
203       if (option == 1)
204       {
205         cudaMemcpy(presult, d_block_sums, sizeof(double), cudaMemcpyDeviceToHost);
206         check = result - result_true;
207       }
208       if (option == 2)
209       {
210         check = result - result_true;
211       }
212       if (option == 3)
213       {
214         cudaMemcpy(presult, d_result, sizeof(double), cudaMemcpyDeviceToHost);
215         check = result - result_true;
216       }
217
218       if (mode == 0)
219       {
220         std::cout << std::endl << "Results after " << tests_done << " tests:" << std::endl;
221         std::cout << "Total runtime: " << total_time << std::endl;
222         std::cout << "Average runtime; " << total_time/tests_done << std::endl;
223         std::cout << "Check: " << result << " ?= " << result_true << std::endl;
224         std::cout << "\n\n";
225       }
226       if (mode == 1)
227       {
228         std::cout << N << ";"
229         << blocks << ";"
230         << block_size << ";"
231         << tests_done << ";"
232         << total_time << ";"
233         << total_time/tests_done << ";"
234         << check << std::endl;
235       }
236       cudaFree(d_x);
237       cudaFree(d_y);
238       cudaFree(d_block_sums);
239       cudaFree(d_result);
240       // free(presult);
241       // free(pnull);
242       free(h_block_sums);
243       if (N==256)
244       {
245         N=100;
246       }
247     }
248     cudaDeviceSynchronize();
249     return EXIT_SUCCESS;
250   }
```

Listing 4: Ex7.2: Sparse Matrix-Vector Product OpenCL - Kernels

```
1  #pragma OPENCL EXTENSION cl_khr_fp64 : enable
2
3  __kernel void ocl_csr_matvec(uint N,
4                          __global int *csr_rowoffsets,
5                          __global int *csr_colindices,
6                          __global double *csr_values,
7                          __global double const *x, __global double *y)
```

```
{
  uint gid = get_global_id(0);
  uint stride = get_global_size(0);
  for (size_t i=gid; i<N; i+=stride) {
    double value = 0;
    for (size_t j=csr_rowoffsets[i]; j<csr_rowoffsets[i+1]; ++j)
      value += csr_values[j] * x[csr_colindices[j]];
    y[i] = value;
  }
};

__kernel void ocl_csr_matvec_fast(uint N,
                        __global int *csr_rowoffsets,
                        __global int *csr_colindices,
                        __global double *csr_values,
                        __global double const *x, __global double *y)
{
  __local double cache[128];
  uint gid = get_group_id(0);
  uint lid = get_local_id(0);
  uint stride = get_num_groups(0);
  uint j_stride = get_global_size(0);
  for (size_t i=gid; i<N; i+=stride) {
    double value = 0;
    for (size_t j=csr_rowoffsets[i]; j<csr_rowoffsets[i+1]; j+=j_stride)
      value += csr_values[j] * x[csr_colindices[j]];
    cache[lid] = value;
    for (int i = get_local_size(0) / 2; i > 0; i /= 2) {
      barrier(CLK_LOCAL_MEM_FENCE);
      if (lid < i)
        cache[lid] += cache[lid + i];
    }
    if (lid==0)
      y[gid] = cache[0];
  }
};;
```

Listing 5: Ex7.2: Sparse Matrix-Vector Product OpenCL - Benchmark

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <cmath>
#include <algorithm>
#include <numeric>
#include <stdexcept>
#include "generate.hpp"
#include "timer.hpp"

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

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

// #ifndef uint
// // not defined for me...it's annoying
// using uint = uint32_t;
// #endif
typedef double ScalarType;

#define LOCAL_SIZE 128
#define GLOBAL_SIZE 128
#define NUM_TESTS 5
#define TARGET "GPU"
#define TRUNC_CSV
#define PP 16

```

```
34  #define SLOW_KERNEL
35  #ifndef SLOW_KERNEL
36  #define FAST_KERNEL
37  #endif
38  std::string target = TARGET;
39  #ifdef SLOW_KERNEL
40  std::string csv_name = "ph_data_" + target + ".csv";
41  #endif
42  #ifdef FAST_KERNEL
43  std::string csv_name = "ph_data2_" + target + ".csv";
44  #endif
45
46  std::string my_opencl_program =  "\n"
47  "#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n"
48  " \n\n"
49  "__kernel void ocl_csr_matvec(uint N,\n"
50  "                        __global int *csr_rowoffsets,\n"
51  "                        __global int *csr_colindices,\n"
52  "                        __global double *csr_values,\n"
53  "                        __global double const *x, __global double *y)\n"
54  "{\n"
55  "  uint gid = get_global_id(0);\n"
56  "  uint stride = get_global_size(0);\n"
57  "  for (size_t i=gid; i<N; i+=stride) {\n"
58  "    double value = 0;\n"
59  "    for (size_t j=csr_rowoffsets[i]; j<csr_rowoffsets[i+1]; ++j)\n"
60  "      value += csr_values[j] * x[csr_colindices[j]];\n"
61  "    y[i] = value;\n"
62  "  }\n"
63  "};"
64  " \n\n"
65  "__kernel void ocl_csr_matvec_fast(uint N,\n"
66  "                        __global int *csr_rowoffsets,\n"
67  "                        __global int *csr_colindices,\n"
68  "                        __global double *csr_values,\n"
69  "                        __global double const *x, __global double *y)\n"
70  "{\n"
71  "  __local double cache[128];\n"
72  "  uint gid = get_group_id(0);\n"
73  "  uint lid = get_local_id(0);\n"
74  "  uint stride = get_num_groups(0);\n"
75  "  uint j_stride = get_global_size(0);\n"
76  "  for (size_t i=gid; i<N; i+=stride) {\n"
77  "    double value = 0;\n"
78  "    for (size_t j=csr_rowoffsets[i]; j<csr_rowoffsets[i+1]; j+=j_stride)\n"
79  "      value += csr_values[j] * x[csr_colindices[j]];\n"
80  "    cache[lid] = value;"
81  "    for (int i = get_local_size(0) / 2; i > 0; i /= 2) {\n"
82  "      barrier(CLK_LOCAL_MEM_FENCE);\n"
83  "      if (lid < i)\n"
84  "        cache[lid] += cache[lid + i];\n"
85  "    }\n"
86  "    if (lid==0)\n"
87  "      y[gid] = cache[0];\n"
88  "  }\n"
89  "};";
90
91  template <typename T>
92  void printContainer(T container, const int size) {
93    std::cout << container[0];
94    for (int i = 1; i < size; ++i)
95      std::cout << " | " << container[i] ;
96    std::cout << std::endl;
97  }
98
99  template <typename T>
100 void printContainer(T container, const int size, const int only) {
101   std::cout << container[0];
102   for (int i = 1; i < only; ++i)
103       std::cout  << " | " << container[i];
104   std::cout << " | ...";
105   for (int i = size - only; i < size; ++i)
```

```
106       std::cout  << " | " << container[i];
107     std::cout << std::endl;
108   }
109
110   double median(std::vector<double>& vec)
111   {
112     size_t size = vec.size();
113     if (size == 0)
114             return 0.;
115     sort(vec.begin(), vec.end());
116     size_t mid = size/2;
117
118     return size % 2 == 0 ? (vec[mid] + vec[mid-1]) / 2 : vec[mid];
119   };
120
121   bool check(const double* test, const double* ref, const uint N) {
122     for (uint i = 0; i < N; ++i){
123       if (test[i] != ref[i])
124         return false;
125     }
126     return true;
127   }
128
129   double diff_norm(const double* test, const double* ref, const uint N) {
130     double norm = 0.0;
131     for (uint i = 0; i < N; ++i){
132       norm += test[i] != ref[i];
133     }
134     return sqrt(norm);
135   }
136
137
138   /** Computes y = A*x for a sparse matrix A in CSR format and vector x,y. CPU implementation.
          */
139   void csr_matvec_product(size_t N,
140                           int *csr_rowoffsets, int *csr_colindices, double *csr_values,
141                           double const *x, double *y)
142   {
143     for (size_t i=0; i<N; ++i) {
144       double value = 0;
145       for (size_t j=csr_rowoffsets[i]; j<csr_rowoffsets[i+1]; ++j)
146         value += csr_values[j] * x[csr_colindices[j]];
147
148       y[i] = value;
149     }
150   }
151
152
153   double benchmark_ocl(size_t N, size_t max_nonzeros_per_row,
154                        int* csr_rowoffsets, int *csr_colindices,
155                        double* csr_values,
156                        double* x, double* y)
157   {
158     cl_int err;
159     Timer timer1;
160     //
161     ///////////////////////////// Part 1: Set up an OpenCL context with one device
          //////////////////////////////////////
162     //
163
164     //
165     // Query platform:
166     //
167     cl_uint num_platforms;
168     cl_platform_id platform_ids[42];   //no more than 42 platforms supported...
169     err = clGetPlatformIDs(42, platform_ids, &num_platforms); OPENCL_ERR_CHECK(err);
170     //std::cout << "# Platforms found: " << num_platforms << std::endl;
171
172     //
173     // Query devices:
174     //
175     cl_device_id device_ids[42];
```

```
176     cl_uint num_devices;
177     char device_name[64];
178     cl_device_id my_device_id;
179     cl_platform_id my_platform;
180     for (int i = 0; i < num_platforms; ++i)
181     {
182       my_platform = platform_ids[i];
183       if (target == "GPU") {
184         err = clGetDeviceIDs(my_platform, CL_DEVICE_TYPE_GPU, 42, device_ids, &num_devices);
185       }
186       else {
187         err = clGetDeviceIDs(my_platform, CL_DEVICE_TYPE_CPU, 42, device_ids, &num_devices);
188       }
189       if (err == CL_SUCCESS)
190         break;
191     }
192     OPENCL_ERR_CHECK(err);
193     //std::cout << "# Devices found: " << num_devices << std::endl;
194     my_device_id = device_ids[0];
195
196     size_t device_name_len = 0;
197     err = clGetDeviceInfo(my_device_id, CL_DEVICE_NAME, sizeof(char)*63, device_name, &
          device_name_len); OPENCL_ERR_CHECK(err);
198
199     std::cout << "Using the following device: " << device_name << std::endl;
200
201     //
202     // Create context:
203     //
204     cl_context my_context = clCreateContext(0, 1, &my_device_id, NULL, NULL, &err);
          OPENCL_ERR_CHECK(err);
205
206     //
207     // create a command queue for the device:
208     //
209     cl_command_queue my_queue = clCreateCommandQueueWithProperties(my_context, my_device_id,
          0, &err); OPENCL_ERR_CHECK(err);
210
211     //
212     /////////////////////////// Part 2: Create a program and extract kernels
          ////////////////////////////////////
213     //
214     //
215     // Build the program:
216     //
217     const char* ocl_prog = my_opencl_program.c_str();
218     size_t source_len = my_opencl_program.length();
219
220     cl_program prog = clCreateProgramWithSource(my_context, 1, &ocl_prog, &source_len, &err);
          OPENCL_ERR_CHECK(err);
221     err = clBuildProgram(prog, 0, NULL, NULL, NULL, NULL);
222
223     //
224     // Print compiler errors if there was a problem:
225     //
226     if (err != CL_SUCCESS) {
227
228       char *build_log;
229       size_t ret_val_size;
230       err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, 0, NULL, &
            ret_val_size);
231       build_log = (char *)malloc(sizeof(char) * (ret_val_size+1));
232       err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, ret_val_size,
            build_log, NULL);
233       build_log[ret_val_size] = '\0'; // terminate string
234       std::cout << "Log: " << build_log << std::endl;
235       free(build_log);
236       std::cout << "OpenCL program sources: " << std::endl << my_opencl_program << std::endl;
237       return EXIT_FAILURE;
238     }
239     //
240     // Extract the only kernel in the program:
```

```
241    //
242
243  #ifdef SLOW_KERNEL
244    cl_kernel my_kernel = clCreateKernel(prog, "ocl_csr_matvec", &err); OPENCL_ERR_CHECK(err);
245  #endif
246  #ifdef FAST_KERNEL
247    cl_kernel my_kernel = clCreateKernel(prog, "ocl_csr_matvec", &err); OPENCL_ERR_CHECK(err);
248  #endif
249
250    //
251    /////////////////////////// Part 3: Create memory buffers
           ///////////////////////////////////
252    //
253
254    cl_uint vector_size = N;
255    size_t  local_size = LOCAL_SIZE;
256    size_t global_size = GLOBAL_SIZE*GLOBAL_SIZE;
257    size_t groups = 1 + int(N/LOCAL_SIZE);
258
259    cl_mem ocl_x = clCreateBuffer(my_context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, sizeof
           (ScalarType)*N, x, &err); OPENCL_ERR_CHECK(err);
260
261    cl_mem ocl_y = clCreateBuffer(my_context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, sizeof
           (ScalarType)*N, y, &err); OPENCL_ERR_CHECK(err);
262
263    cl_mem ocl_csr_rowoffsets = clCreateBuffer(my_context, CL_MEM_READ_WRITE |
           CL_MEM_COPY_HOST_PTR, sizeof(int)*(N+1), csr_rowoffsets, &err); OPENCL_ERR_CHECK(err);
264
265    cl_mem ocl_csr_colindices = clCreateBuffer(my_context, CL_MEM_READ_WRITE |
           CL_MEM_COPY_HOST_PTR, sizeof(int)*(max_nonzeros_per_row*N), csr_colindices, &err);
           OPENCL_ERR_CHECK(err);
266
267    cl_mem ocl_csr_values = clCreateBuffer(my_context, CL_MEM_READ_WRITE |
           CL_MEM_COPY_HOST_PTR, sizeof(double)*(max_nonzeros_per_row*N), csr_values, &err);
           OPENCL_ERR_CHECK(err);
268
269    //
270    /////////////////////////// Part 4: Run kernel ////////////////////////////////////
271    //
272    //
273    // Set kernel arguments:
274    //
275
276  // "__kernel void ocl_csr_matvec(size_t N,\n"
277  // "                         __global int *csr_rowoffsets,\n"
278  // "                         __global int *csr_colindices,\n"
279  // "                         __global double *csr_values,\n"
280  // "                         __global double const *x, __global double *y)\n"
281    err = clSetKernelArg(my_kernel, 0, sizeof(cl_uint), (void*)&vector_size); OPENCL_ERR_CHECK
           (err);
282    err = clSetKernelArg(my_kernel, 1, sizeof(cl_mem),  (double*)&ocl_csr_rowoffsets);
           OPENCL_ERR_CHECK(err);
283    err = clSetKernelArg(my_kernel, 2, sizeof(cl_mem),  (double*)&ocl_csr_colindices);
           OPENCL_ERR_CHECK(err);
284    err = clSetKernelArg(my_kernel, 3, sizeof(cl_mem),  (double*)&ocl_csr_values);
           OPENCL_ERR_CHECK(err);
285    err = clSetKernelArg(my_kernel, 4, sizeof(cl_mem),  (double*)&ocl_x); OPENCL_ERR_CHECK(err
           );
286    err = clSetKernelArg(my_kernel, 5, sizeof(cl_mem),  (double*)&ocl_y); OPENCL_ERR_CHECK(err
           );
287
288    //
289    // Enqueue kernel in command queue:
290    //
291    timer1.reset();
292    err = clEnqueueNDRangeKernel(my_queue, my_kernel, 1, NULL, &global_size, &local_size, 0,
           NULL, NULL); OPENCL_ERR_CHECK(err);
293
294    // wait for all operations in queue to finish:
295    err = clFinish(my_queue); OPENCL_ERR_CHECK(err);
296    double ocl_time = timer1.get();
297
```

```
298     //
299     /////////////////////////// Part 5: Get data from OpenCL buffer
            ///////////////////////////////////
300     //
301     err = clEnqueueReadBuffer(my_queue, ocl_y, CL_TRUE, 0, sizeof(ScalarType)*vector_size, y,
            0, NULL, NULL); OPENCL_ERR_CHECK(err);
302
303     // wait for all operations in queue to finish:
304     err = clFinish(my_queue); OPENCL_ERR_CHECK(err);
305     //
306     // cleanup
307     //
308     clReleaseMemObject(ocl_x);
309     clReleaseMemObject(ocl_y);
310     clReleaseMemObject(ocl_csr_rowoffsets);
311     clReleaseMemObject(ocl_csr_colindices);
312     clReleaseMemObject(ocl_csr_values);
313     clReleaseProgram(prog);
314     clReleaseCommandQueue(my_queue);
315     clReleaseContext(my_context);
316
317     std::cout << "From OCL benchmark: " << ocl_time << std::endl;
318
319     return ocl_time;
320   }
321
322
323   /** Solve a system with `points_per_direction * points_per_direction` unknowns */
324   void benchmark_matvec(size_t points_per_direction, size_t max_nonzeros_per_row,
325                         void (*generate_matrix)(size_t, int*, int*, double*),
326                         std::string gen_type) // function pointer parameter
327   {
328
329     size_t N = points_per_direction * points_per_direction; // number of rows and columns
330     std::fstream csv;
331     csv.open(csv_name, std::fstream::out | std::fstream::app);
332
333     //
334     // Allocate CSR arrays.
335     //
336     // Note: Usually one does not know the number of nonzeros in the system matrix a-priori.
337     //       For this exercise, however, we know that there are at most 5 nonzeros per row in
            the system matrix, so we can allocate accordingly.
338     //
339     int *csr_rowoffsets =    (int*)malloc(sizeof(double) * (N+1));
340     int *csr_colindices =    (int*)malloc(sizeof(double) * max_nonzeros_per_row * N);
341     double *csr_values  = (double*)malloc(sizeof(double) * max_nonzeros_per_row * N);
342
343     //
344     // fill CSR matrix with values
345     //
346     generate_matrix(points_per_direction, csr_rowoffsets, csr_colindices, csr_values);
347
348     //
349     // Allocate vectors:
350     //
351     double *x = (double*)malloc(sizeof(double) * N); std::fill(x, x + N, 1);
352     double *y = (double*)malloc(sizeof(double) * N); std::fill(y, y + N, 0);
353     double *y_ocl = (double*)malloc(sizeof(double) * N); std::fill(y, y + N, 0);
354
355     //
356     // Call matrix-vector product reference
357     //
358     Timer timer;
359     timer.reset();
360     csr_matvec_product(N, csr_rowoffsets, csr_colindices, csr_values, x, y);
361     double cpu_time = timer.get();
362
363     //
364     // Call matrix-vector product kernel
365     //
366     double ocl_time = benchmark_ocl(N, max_nonzeros_per_row, csr_rowoffsets, csr_colindices,
```

```
        csr_values, x, y_ocl);
367     std::cout << "Reference: " << std::endl;
368     printContainer(y, N, 10);
369     std::cout << "OpenCL: " << std::endl;
370     printContainer(y_ocl, N, 10);
371     double difference = diff_norm(y_ocl, y, N);
372     bool check_res = check(y_ocl, y, N);
373     std::cout << "Difference between the two: " << difference << " (check: " << check_res <<
            std::endl;
374     std::cout << "Time for ref product: " << cpu_time << std::endl;
375     std::cout << "Time for OCL product: " << ocl_time << std::endl;
376
377     csv << N<< ";"
378         << points_per_direction << ";"
379         << target << ";"
380         << gen_type << ";"
381         << ocl_time << ";"
382         << cpu_time << ";"
383         << difference << ";"
384         << check_res << std::endl;
385
386     csv.close();
387     free(x);
388     free(y);
389     free(y_ocl);
390     free(csr_rowoffsets);
391     free(csr_colindices);
392     free(csr_values);
393   }
394
395
396   int main() {
397     std::fstream csv;
398   #ifdef TRUNC_CSV
399     csv.open(csv_name, std::fstream::out | std::fstream::trunc);
400     csv << "N;points_per_direction;target;gen_type;ocl_time;cpu_time;check;diff_norm" << std::
            endl;
401   #endif
402   #ifndef TRUNC_CSV
403     csv.open(csv_name, std::fstream::out | std::fstream::app);
404   #endif
405
406     uint pp = PP;
407     std::cout << "# Benchmarking finite difference matrix" << std::endl;
408     benchmark_matvec(pp, 5, generate_fdm_laplace, "1"); // 100*100 unknowns, finite difference
            matrix
409     // the last string is just so that I know which matrix was used
410
411     std::cout << "# Benchmarking special matrix" << std::endl;
412     benchmark_matvec(pp, 2000, generate_matrix2, "2");     // 100*100 unknowns, special matrix
            with 200-2000 nonzeros per row
413
414     std::cout << "Data: \nRuntimes in csv form can be found here\nhttps://gtx1080.360252.org
            /2020/ex7/" << csv_name << std::endl;
415
416     return EXIT_SUCCESS;
417   }
```