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.

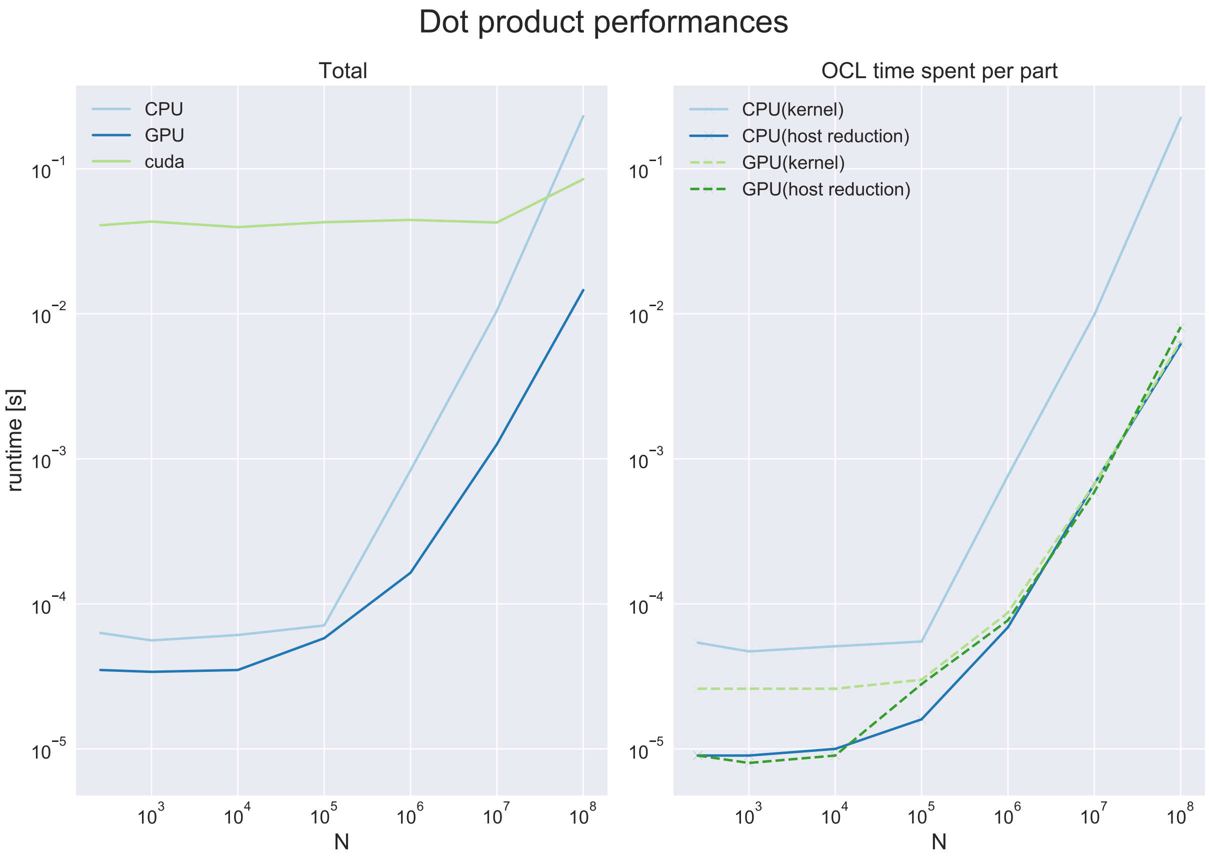


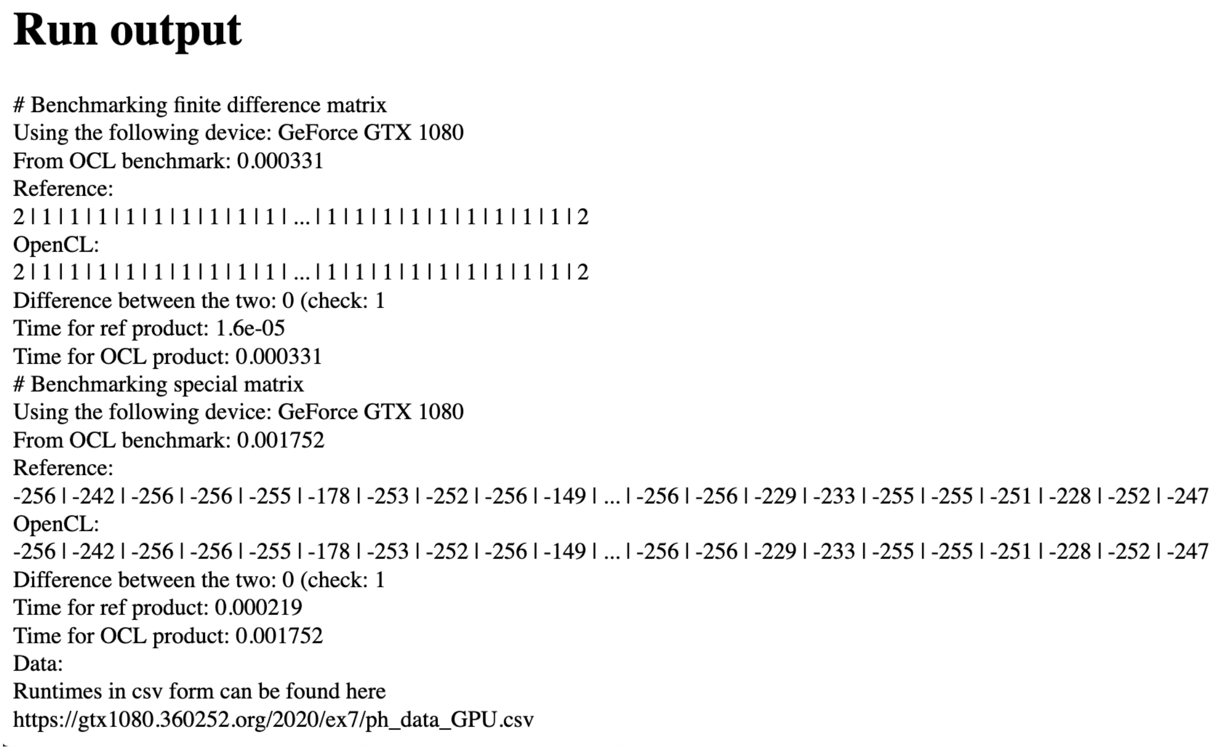
Fig. 1. Runtimes of the different vector dot product based kernels.

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 | 2
OpenCL:
2 | 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).

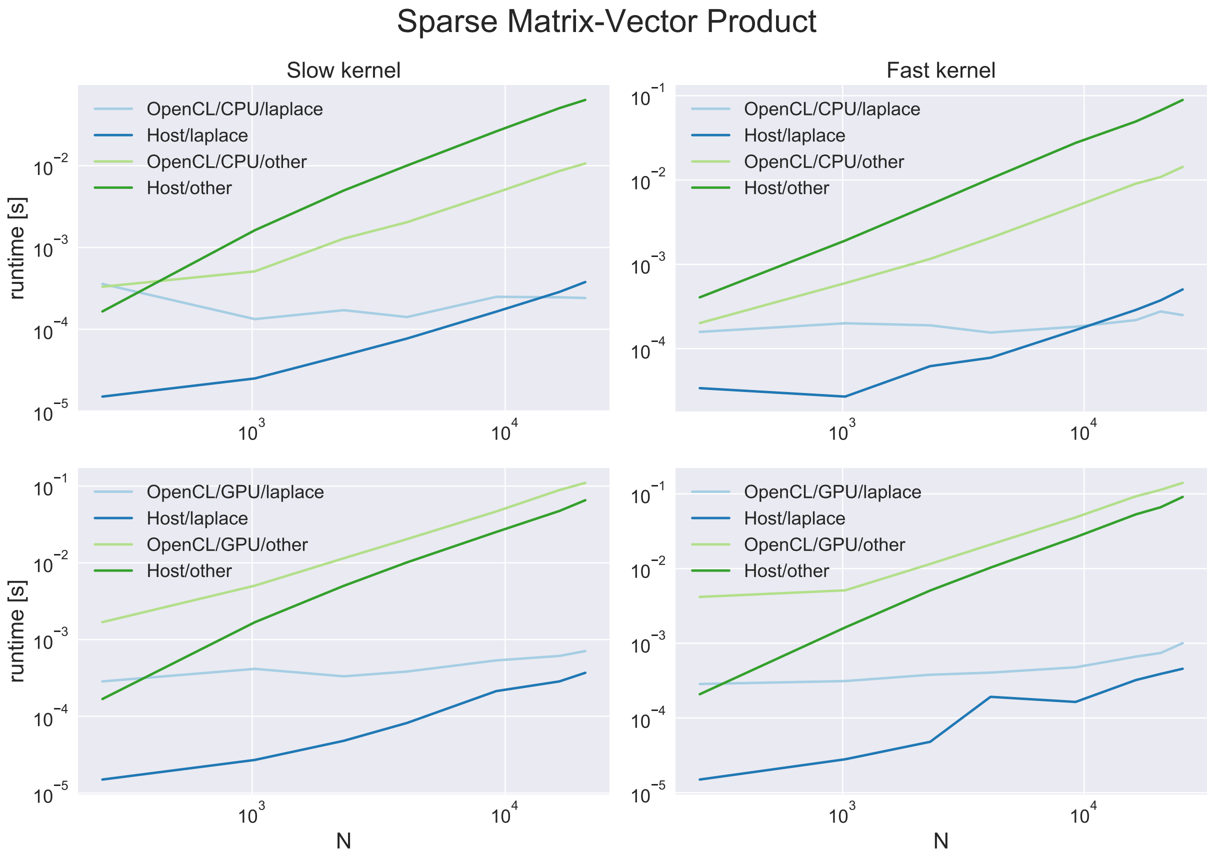


Fig. 3. Runtimes of the Sparse Matrix-Vector products. The left side shows the "slow" kernel, the right side the "faster" version. The matrix type is denoted by either "laplace" (blue lines) or "other" (green lines). The top two show the OpenCL-CPU runtimes, whereas the bottom shows the GPU runtimes.

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.

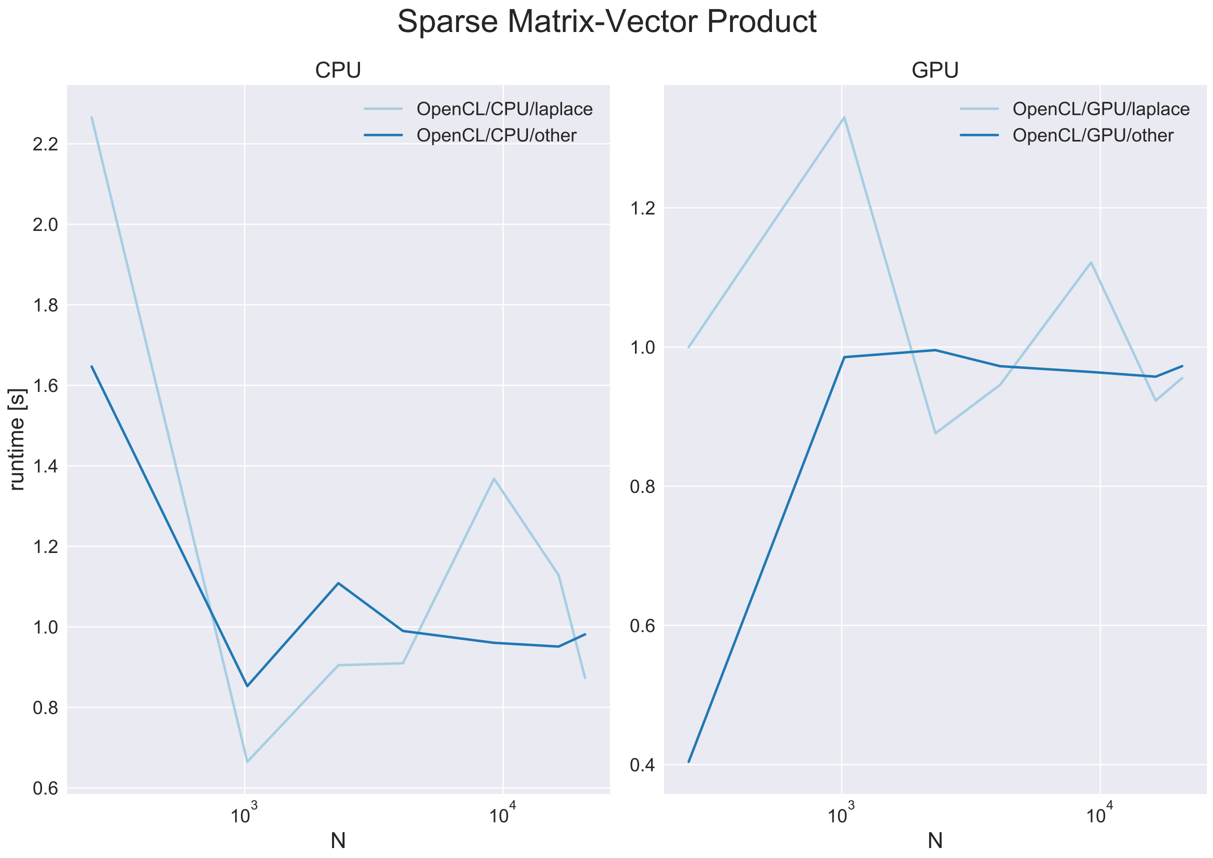


Fig. 4. Speedup of the "faster" kernel compared to the "slower" kernel on the CPU and GPU respectively.

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 << " | ...";
  for (int i = size - only; i < size; ++i)
    std::cout  << " | " << container[i];
  std::cout << std::endl;
}

double median(std::vector<double>& vec)
{
  size_t size = vec.size();
  if (size == 0)
          return 0.;
  sort(vec.begin(), vec.end());
  size_t mid = size/2;

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

// const char *my_opencl_program =  "\n"
// "#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n"
// " \n\n"
// "__kernel void xDOTy(__global double *result, \n"
// "                    __global double *x,\n"
// "                    __global double *y, \n"
// "                    __local cache, uint N) {\n"
// "  uint gid = get_global_id(0);\n"
// "  uint lid = get_local_id(0);\n"
// "  uint stride = get_global_size(0);\n"
// "  double tmp = 0.0;\n"
// "  for (uint i = gid; i < N; i += stride)\n"
// "    tmp = x[i] * y[i];\n"
// "  cache[lid] = tmp;\n"
// "  \n"
// "  for (int i = get_local_size(0) / 2; i > 0; i /= 2) {\n"
// "    barrier(CLK_LOCAL_MEM_FENCE);\n"
// "    if (lid < i)\n"
// "      cache[lid] += cache[lid + i];\n"
// "  }\n"
// "  if (lid==0)\n"
// "    result[get_group_id()] = cache[lid]\n"
// "};\n";


// "                    __local cache, uint N) {\n"

std::string my_opencl_program_c =  "\n"
"#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n"
" \n\n"
"__kernel void xDOTy(__global double *result, \n"
"                    __global double *x,\n"
"                    __global double *y, \n"
"                    uint N) {\n"
"  uint gid = get_global_id(0);\n"
"  uint lid = get_local_id(0);\n"
"  uint stride = get_global_size(0);\n"
"  __local double cache[128];\n"
"  double tmp = 0.0;\n"
"  for (uint i = gid; i < N; i += stride)\n"
"    tmp += x[i] * y[i];\n"
"  cache[lid] = tmp;\n"
"  \n"
"  for (int i = get_local_size(0) / 2; i > 0; i /= 2) {\n"
"    barrier(CLK_LOCAL_MEM_FENCE);\n"
"    if (lid < i)\n"
"      cache[lid] += cache[lid + i];\n"
"  }\n"
"  if (lid==0)\n"
"    result[get_group_id(0)] = cache[lid];\n"
"};\n";

std::string header = "\n"
"#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n"
" \n\n";
std::string kernel_signature = "__kernel void xDOTy";
```

```
std::string kernel_code1 = "(__global double *result, \n"
"                    __global double *x,\n"
"                    __global double *y, \n"
"                    uint N) {\n";
std::string kernel_code2 = "  uint gid = get_global_id(0);\n"
"  uint lid = get_local_id(0);\n"
"  uint stride = get_global_size(0);\n"
"   __local double cache[128];\n"
"  double tmp = 0.0;\n"
"  for (uint i = gid; i < N; i += stride)\n"
"    tmp += x[i] * y[i];\n"
"  cache[lid] = tmp;\n"
"  \n"
"  for (int i = get_local_size(0) / 2; i > 0; i /= 2) {\n"
"    barrier(CLK_LOCAL_MEM_FENCE);\n"
"    if (lid < i)\n"
"      cache[lid] += cache[lid + i];\n"
"  }\n"
"  if (lid==0)\n"
"    result[get_group_id(0)] = cache[lid];\n"
"};\n";


int main()
{
  cl_int err;

  bool compute = false;
  bool compile_M = true;
  std::vector<uint> M_vec{1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
  if (!compile_M) {
    M_vec.clear();
    M_vec.push_back(1);
  }
  // std::string target = "GeForce GTX 1080";
  std::string target = "GPU";
  std::vector<uint> N_vec{256, 1000, 10000, 100000, 1000000, 10000000, 100000000};
  uint N_min = N_vec.front();
  uint N_max = N_vec.back();
  uint cnt = N_vec.size();


  Timer timer;
  timer.reset();

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

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


  //
  // Query devices:
  //
  cl_device_id device_ids[42];
  cl_uint num_devices;
  char device_name[64];
  cl_device_id my_device_id;
  cl_platform_id my_platform;
  for (int i = 0; i < num_platforms; ++i)
  {
    my_platform = platform_ids[i];
    if (target == "GPU") {
      err = clGetDeviceIDs(my_platform, CL_DEVICE_TYPE_GPU, 42, device_ids, &num_devices);
    }
    else {
      err = clGetDeviceIDs(my_platform, CL_DEVICE_TYPE_CPU, 42, device_ids, &num_devices);
    }
    if (err == CL_SUCCESS)
      break;
  }
  OPENCL_ERR_CHECK(err);
  std::cout << "# Devices found: " << num_devices << std::endl;
  my_device_id = device_ids[0];

  size_t device_name_len = 0;
  err = clGetDeviceInfo(my_device_id, CL_DEVICE_NAME, sizeof(char)*63, device_name, &device_name_len); OPENCL_ERR_CHECK(err);

  std::cout << "Using the following device: " << device_name << std::endl;

  //
  // Create context:
  //
  cl_context my_context = clCreateContext(0, 1, &my_device_id, NULL, NULL, &err); OPENCL_ERR_CHECK(err);


  //
  // create a command queue for the device:
  //
  cl_command_queue my_queue = clCreateCommandQueueWithProperties(my_context, my_device_id, 0, &err); OPENCL_ERR_CHECK(err);


  //
  /////////////////////////// Part 2: Create a program and extract kernels ///////////////////////////////////
  //
  //
  // Build the program:
  //
  cl_kernel my_kernel;
  cl_program prog;


  std::fstream csv_compile;
  std::string csv_compile_name = "ph_data_compile.csv";

  if (compile_M) {
    csv_compile.open(csv_compile_name, std::fstream::out | std::fstream::trunc);
    csv_compile << "M;compile_time;create_time" << std::endl;
  }
  int m = 1;

  std::string ocl_prog = header + kernel_signature + kernel_code1 + kernel_code2;
  for (auto& M : M_vec){

    // // To gernerate the M kernels
    for (; m < M; ++m) {
      ocl_prog += kernel_signature + std::to_string(m) + kernel_code1 + "uint insert" + std::to_string(m) + "=1;\n" + kernel_code2;
    }
    const char * my_opencl_program = ocl_prog.c_str();

    std::vector<double> tmp(NUM_TESTS, 0);
    for (uint iter = 0; iter < NUM_TESTS; iter++){
      timer.reset();
      size_t source_len = std::string(my_opencl_program).length();
      prog = clCreateProgramWithSource(my_context, 1, &my_opencl_program, &source_len, &err);OPENCL_ERR_CHECK(err);
      err = clBuildProgram(prog, 0, NULL, NULL, NULL, NULL);
      tmp[iter] = timer.get();
    }
    double compile_time = median(tmp);
    //
```

```
    // Print compiler errors if there was a problem:
    //
    if (err != CL_SUCCESS) {

      char *build_log;
      size_t ret_val_size;
      err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, 0, NULL, &ret_val_size);
      build_log = (char *)malloc(sizeof(char) * (ret_val_size+1));
      err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, ret_val_size, build_log, NULL);
      build_log[ret_val_size] = '\0'; // terminate string
      std::cout << "Log: " << build_log << std::endl;
      free(build_log);
      std::cout << "OpenCL program sources: " << std::endl << my_opencl_program << std::endl;
      return EXIT_FAILURE;
    }
    //
    // Extract the only kernel in the program:
    //
    for (uint iter = 0; iter < NUM_TESTS; iter++){
      timer.reset();
      my_kernel = clCreateKernel(prog, "xDOTy", &err); OPENCL_ERR_CHECK(err);
      tmp[iter] = timer.get();
    }
    double create_time = median(tmp);
    std::cout << "Time to compile and create kernel: " << compile_time << std::endl;
    csv_compile << M << ";"
                << compile_time << ";"
                << create_time << std::endl;
  }


  // Plan to reuse all these vectors and buffers
  double y_val = 2., x_val=1.;
  std::vector<ScalarType> x(N_max, x_val);
  std::vector<ScalarType> y(N_max, y_val);
  std::vector<ScalarType> dot_vec(N_max, 0);

  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);
  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);
  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);

  //
  ///////////////////////// Part 3: Create memory buffers ///////////////////////////////////
  //

  std::string csv_name = "ph_data_" + target + ".csv";
  std::fstream csv;
  if (compute){
    std::vector<double> times_cpu(N_vec.size(), 0.);
    std::vector<double> times_gpu(N_vec.size(), 0.);
    std::vector<double> times_total(N_vec.size(), 0.);
    csv.open(csv_name, std::fstream::out | std::fstream::trunc);
    csv << "N;target;local_size;global_size;ocl_time;cpu_time;total_time;dot" << std::endl;
    for (auto& N: N_vec){

      cl_uint vector_size = N;
      size_t  local_size = LOCAL_SIZE;
      size_t global_size = GLOBAL_SIZE*GLOBAL_SIZE;
      size_t groups = 1 + int(N/LOCAL_SIZE);

      //
      ///////////////////////// Part 4: Run kernel ///////////////////////////////////
      //


      //
      // Set kernel arguments:
      //
      // xDOTy(__global double *result,
      //       __global double *x,
      //       __global double *y,
      //       __local cache, uint N)
      err = clSetKernelArg(my_kernel, 0, sizeof(cl_mem),  (double*)&ocl_dot); OPENCL_ERR_CHECK(err);
      err = clSetKernelArg(my_kernel, 1, sizeof(cl_mem),  (double*)&ocl_x); OPENCL_ERR_CHECK(err);
      err = clSetKernelArg(my_kernel, 2, sizeof(cl_mem),  (double*)&ocl_y); OPENCL_ERR_CHECK(err);
      // err = clSetKernelArg(my_kernel, 3, sizeof(cl_float) * local_work_size[0], NULL); OPENCL_ERR_CHECK(err);
      err = clSetKernelArg(my_kernel, 3, sizeof(cl_uint), (void*)&vector_size); OPENCL_ERR_CHECK(err);

      //
      // Enqueue kernel in command queue:
      //
      std::vector<double> tmp(NUM_TESTS, 0);
      for (uint iter = 0; iter < NUM_TESTS; iter++){
        timer.reset();
        err = clEnqueueNDRangeKernel(my_queue, my_kernel, 1, NULL, &global_size, &local_size, 0, NULL, NULL); OPENCL_ERR_CHECK(err);

        // wait for all operations in queue to finish:
        err = clFinish(my_queue); OPENCL_ERR_CHECK(err);
        tmp[iter] = timer.get();
      }
      double ocl_time = median(tmp);

      //
      ///////////////////////// Part 5: Get data from OpenCL buffer ///////////////////////////////////
      //

      // err = clEnqueueReadBuffer(my_queue, ocl_x, CL_TRUE, 0, sizeof(ScalarType) * x.size(), &(x[0]), 0, NULL, NULL); OPENCL_ERR_CHECK(err);
      err = clEnqueueReadBuffer(my_queue, ocl_dot, CL_TRUE, 0, sizeof(ScalarType)*vector_size, dot_vec.data(), 0, NULL, NULL); OPENCL_ERR_CHECK(err);

      timer.reset();
      double dot = std::accumulate(dot_vec.begin(), dot_vec.begin()+groups, 0.);
      double cpu_time = timer.get();
      double total_time = ocl_time + cpu_time;


      std::cout << std::endl;
      std::cout << "Result of kernel execution: " << dot;
      std::cout << " =? " << vector_size*y_val*x_val << " : " << (dot == y_val*x_val*vector_size) << std::endl;
      std::cout << "Runtime: kernel + cpu = total_time" << std::endl;
      std::cout << ocl_time << " + " << cpu_time << " = " << total_time << std::endl;

      csv << N<< ";"
          << target << ";"
          << local_size<< ";"
          << global_size<< ";"
          << ocl_time<< ";"
          << cpu_time<< ";"
          << total_time << ";"
          << dot << std::endl;


      // std::cout << "Result container:" << std::endl;
      // printContainer(dot_vec, dot_vec.size(), 10);
    }
    csv.close();
  }
  if (compile_M) csv_compile.close();
  //
  // cleanup
  //
  clReleaseMemObject(ocl_x);
  clReleaseMemObject(ocl_y);
  clReleaseMemObject(ocl_dot);
  clReleaseProgram(prog);
  clReleaseCommandQueue(my_queue);
  clReleaseContext(my_context);

  std::cout << "Data: \nRuntimes in csv form can be found here\nhttps://gtx1080.360252.org/2020/ex7/" << csv_name << std::endl;
  std::cout << "Data: \nCompile times in csv form can be found here\nhttps://gtx1080.360252.org/2020/ex7/" << csv_compile_name << std::endl;

  std::cout << std::endl;
  std::cout << "#" << std::endl;
  std::cout << "# My first OpenCL application finished successfully!" << std::endl;
  std::cout << "#" << std::endl;

  std::cout << "And here it is:" << std::endl;
  std::cout << ocl_prog;
  return EXIT_SUCCESS;
}
```

Listing 3: Ex7.1: Dot products with CUDA

```
#include <iostream>
#include <string>
#include <vector>
#include "timer.hpp"
#include <cmath>

#define BLOCK_SIZE 256

__global__ void initKernel1(double* arr, const double value, const size_t N)
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x +  blockIdx.x * blockDim.x;

  for(; tid < N; tid += stride)
  {
    arr[tid] = value;
  }
}

__global__ void initKernel2(double* arr, double* arr2, const size_t N)
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x +  blockIdx.x * blockDim.x;

  for(; tid < N; tid += stride)
  {
    arr[tid] = tid;
    arr2[tid] = N - 1 - tid;
  }
}

__global__ void initKernel3(double* arr, double* arr2, double* arr3, const size_t N)
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x +  blockIdx.x * blockDim.x;

  for(; tid < N; tid += stride)
  {
    arr[tid] = tid;
    arr2[tid] = N - 1 - tid;
    arr3[tid] = 0;
  }
}

__global__ void addKernel(double* x, double* y, double* res, const size_t N)
```

```
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x +  blockIdx.x * blockDim.x;

  for(; tid < N; tid += stride)
  {
    res[tid] = x[tid] + y[tid];
    //res[tid] += 1;
  }
}

__global__ void dot_A_1(double* x, double* y, double* block_sums, const size_t N)
{
  uint tid = threadIdx.x + blockDim.x* blockIdx.x;
  uint stride = blockDim.x* gridDim.x;

  __shared__ double cache[BLOCK_SIZE];

  double tid_sum = 0.0;
  for (; tid < N; tid += stride)
  {
    tid_sum += x[tid] * y[tid];
  }
  cache[threadIdx.x] = tid_sum;

  __syncthreads();
  for (uint i = blockDim.x/2; i != 0; i /=2)
  {
    if (threadIdx.x < i) //lower half does smth, rest idles
    {
      cache[threadIdx.x] += cache[threadIdx.x + i]; //lower looks up by stride and sums up
    }
    __syncthreads();
  }

  if(threadIdx.x == 0) // cache[0] now contains block_sum
  {
    block_sums[blockIdx.x] = cache[0];
  }
  __syncthreads();
}

__global__ void dot_A_2(double* block_sums)
{
  int tid = threadIdx.x; // only one block, so this is fine!
  for (int i = blockDim.x / 2; i >= 1; i /=2) // same principal as above
  {
    if ( tid < i )
    {
      block_sums[tid] += block_sums[tid + i];
    }
    __syncthreads();
  }
}

__global__ void dot_Atomic(double* x, double* y, double* result, const size_t N)
{
  uint tid = threadIdx.x + blockDim.x* blockIdx.x ;
  uint stride = blockDim.x* gridDim.x ;

  __shared__ double cache[BLOCK_SIZE];

  double tid_sum = 0.0;
  for (; tid < N; tid += stride)
  {
    tid_sum += x[tid] * y[tid];
  }
  tid = threadIdx.x;
  cache[tid] = tid_sum;

  __syncthreads();
  for (uint i = blockDim.x/2; i != 0; i /=2)
  {
    __syncthreads();
    if (tid < i) //lower half does smth, rest idles
    {
      cache[tid] += cache[tid + i ]; //lower looks up by stride and sums up
    }
  }

  if(tid == 0) // cache[0] now contains block_sum
  {
    atomicAdd(result, cache[0]);
  }
}

int main(void)
{
  const int num_tests = 10;
  int tests_done = num_tests;
  const int block_size = BLOCK_SIZE;
  const int mode = 1;
  const int option = 2;

  std::cout << "N;blocks;block_size;tests_done;total_time;time_per_test;check" << std::endl;
  for (size_t N = 256; N <= 1000000000; N*=10)
  {
    std::cout << N << std::endl;
    //int blocks = (int)(N+block_size-1)/block_size;
    int blocks = block_size;
    double result = 0.0;
    double result_true = N;
    double* h_block_sums = (double *)malloc(sizeof(double) * blocks);
    double* presult = (double *)malloc(sizeof(double));
    presult = &result;
    double* pnull = (double *)malloc(sizeof(double));
    *pnull = 0.0;
    double *d_x, *d_y, *d_block_sums, *d_result;

    cudaMalloc(&d_result, sizeof(double));
    cudaMalloc(&d_x, N*sizeof(double));
    cudaMalloc(&d_y, N*sizeof(double));
    cudaMalloc(&d_block_sums, blocks*sizeof(double));
    cudaDeviceSynchronize();

    int i = 0;
    Timer timer;
    double total_time = 0.0;

    initKernel1<<<blocks, block_size>>>(d_block_sums, 0.0, block_size);
    initKernel1<<<blocks, block_size>>>(d_x, 1.0, N);
    initKernel1<<<blocks, block_size>>>(d_y, 1.0, N);
    //initKernel2<<<blocks, block_size>>>(d_x, d_y, N);
    cudaDeviceSynchronize();

    timer.reset();
    for (i = 0; i<num_tests; i++)
    {
      tests_done = i+1;
      if (option == 1)
      {
        dot_A_1<<<blocks, block_size>>>(d_x, d_y, d_block_sums,  N);
        dot_A_2<<<1, block_size>>>(d_block_sums);
        cudaDeviceSynchronize();
      }
      if (option == 2)
      {
        dot_A_1<<<blocks, block_size>>>(d_x, d_y, d_block_sums,  N);
        cudaMemcpy(h_block_sums, d_block_sums, blocks*sizeof(double), cudaMemcpyDeviceToHost);
        //std::cout << h_block_sums[0] << " =? " << h_block_sums[blocks-1] << std::endl;
        cudaDeviceSynchronize();
        result = 0.0;
        for (int j = 0; j < blocks; j+=1)
        {
          result += h_block_sums[j];
        }
      }
      if (option == 3)
      {
        cudaMemcpy(d_result, pnull, sizeof(double), cudaMemcpyHostToDevice);
        dot_Atomic<<<blocks, block_size>>>(d_x, d_y, d_result, N);
      }

      //std::cout << "(" << i+1 << ") Elapsed: " << runtime << " s" << std::endl;
    }
    total_time = timer.get();
    size_t check;
    if (option == 1)
    {
      cudaMemcpy(presult, d_block_sums, sizeof(double), cudaMemcpyDeviceToHost);
      check = result - result_true;
    }
    if (option == 2)
    {
      check = result - result_true;
    }
    if (option == 3)
    {
      cudaMemcpy(presult, d_result, sizeof(double), cudaMemcpyDeviceToHost);
      check = result - result_true;
    }

    if (mode == 0)
    {
      std::cout << std::endl << "Results after " << tests_done << " tests:" << std::endl;
      std::cout << "Total runtime: " << total_time << std::endl;
      std::cout << "Average runtime; " << total_time/tests_done << std::endl;
      std::cout << "Check: " << result << " ?= " << result_true << std::endl;
      std::cout << "\n\n";
    }
    if (mode == 1)
    {
      std::cout << N << ";"
      << blocks << ";"
      << block_size << ";"
      << tests_done << ";"
      << total_time << ";"
      << total_time/tests_done << ";"
      << check << std::endl;
    }
    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_block_sums);
    cudaFree(d_result);
    // free(presult);
    // free(pnull);
    free(h_block_sums);
    if (N==256)
    {
      N=100;
    }
  }
  cudaDeviceSynchronize();
  return EXIT_SUCCESS;
}
```

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

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

__kernel void ocl_csr_matvec(uint N,
                        __global int *csr_rowoffsets,
                        __global int *csr_colindices,
                        __global double *csr_values,
                        __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

```

```
#define SLOW_KERNEL
#ifndef SLOW_KERNEL
#define FAST_KERNEL
#endif
std::string target = TARGET;
#ifdef SLOW_KERNEL
std::string csv_name = "ph_data_" + target + ".csv";
#endif
#ifdef FAST_KERNEL
std::string csv_name = "ph_data2_" + target + ".csv";
#endif

std::string my_opencl_program =  "\n"
"#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n"
" \n\n"
"__kernel void ocl_csr_matvec(uint N,\n"
"                        __global int *csr_rowoffsets,\n"
"                        __global int *csr_colindices,\n"
"                        __global double *csr_values,\n"
"                        __global double const *x, __global double *y)\n"
"{\n"
"  uint gid = get_global_id(0);\n"
"  uint stride = get_global_size(0);\n"
"  for (size_t i=gid; i<N; i+=stride) {\n"
"    double value = 0;\n"
"    for (size_t j=csr_rowoffsets[i]; j<csr_rowoffsets[i+1]; ++j)\n"
"      value += csr_values[j] * x[csr_colindices[j]];\n"
"    y[i] = value;\n"
"  }\n"
"};"
" \n\n"
"__kernel void ocl_csr_matvec_fast(uint N,\n"
"                        __global int *csr_rowoffsets,\n"
"                        __global int *csr_colindices,\n"
"                        __global double *csr_values,\n"
"                        __global double const *x, __global double *y)\n"
"{\n"
"  __local double cache[128];\n"
"  uint gid = get_group_id(0);\n"
"  uint lid = get_local_id(0);\n"
"  uint stride = get_num_groups(0);\n"
"  uint j_stride = get_global_size(0);\n"
"  for (size_t i=gid; i<N; i+=stride) {\n"
"    double value = 0;\n"
"    for (size_t j=csr_rowoffsets[i]; j<csr_rowoffsets[i+1]; j+=j_stride)\n"
"      value += csr_values[j] * x[csr_colindices[j]];\n"
"    cache[lid] = value;"
"    for (int i = get_local_size(0) / 2; i > 0; i /= 2) {\n"
"      barrier(CLK_LOCAL_MEM_FENCE);\n"
"      if (lid < i)\n"
"        cache[lid] += cache[lid + i];\n"
"    }\n"
"    if (lid==0)\n"
"      y[gid] = cache[0];\n"
"  }\n"
"};";

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 << " | ...";
  for (int i = size - only; i < size; ++i)
```

```
    std::cout  << " | " << container[i];
  std::cout << std::endl;
}

double median(std::vector<double>& vec)
{
  size_t size = vec.size();
  if (size == 0)
          return 0.;
  sort(vec.begin(), vec.end());
  size_t mid = size/2;

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

bool check(const double* test, const double* ref, const uint N) {
  for (uint i = 0; i < N; ++i){
    if (test[i] != ref[i])
      return false;
  }
  return true;
}

double diff_norm(const double* test, const double* ref, const uint N) {
  double norm = 0.0;
  for (uint i = 0; i < N; ++i){
    norm += test[i] != ref[i];
  }
  return sqrt(norm);
}


/** Computes y = A*x for a sparse matrix A in CSR format and vector x,y. CPU implementation. */
void csr_matvec_product(size_t N,
                        int *csr_rowoffsets, int *csr_colindices, double *csr_values,
                        double const *x, double *y)
{
  for (size_t i=0; i<N; ++i) {
    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;
  }
}


double benchmark_ocl(size_t N, size_t max_nonzeros_per_row,
                  int* csr_rowoffsets, int *csr_colindices,
                  double* csr_values,
                  double* x, double* y)
{
  cl_int err;
  Timer timer1;
  //
  /////////////////////////// Part 1: Set up an OpenCL context with one device ////////////////////////////////////
  //

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

  //
  // Query devices:
  //
  cl_device_id device_ids[42];
  cl_uint num_devices;
  char device_name[64];
  cl_device_id my_device_id;
  cl_platform_id my_platform;
  for (int i = 0; i < num_platforms; ++i)
  {
    my_platform = platform_ids[i];
    if (target == "GPU") {
      err = clGetDeviceIDs(my_platform, CL_DEVICE_TYPE_GPU, 42, device_ids, &num_devices);
    }
    else {
      err = clGetDeviceIDs(my_platform, CL_DEVICE_TYPE_CPU, 42, device_ids, &num_devices);
    }
    if (err == CL_SUCCESS)
      break;
  }
  OPENCL_ERR_CHECK(err);
  //std::cout << "# Devices found: " << num_devices << std::endl;
  my_device_id = device_ids[0];

  size_t device_name_len = 0;
  err = clGetDeviceInfo(my_device_id, CL_DEVICE_NAME, sizeof(char)*63, device_name, &device_name_len); OPENCL_ERR_CHECK(err);

  std::cout << "Using the following device: " << device_name << std::endl;

  //
  // Create context:
  //
  cl_context my_context = clCreateContext(0, 1, &my_device_id, NULL, NULL, &err); OPENCL_ERR_CHECK(err);

  //
  // create a command queue for the device:
  //
  cl_command_queue my_queue = clCreateCommandQueueWithProperties(my_context, my_device_id, 0, &err); OPENCL_ERR_CHECK(err);

  //
  ////////////////////////// Part 2: Create a program and extract kernels ///////////////////////////////////
  //
  //
  // Build the program:
  //
  const char* ocl_prog = my_opencl_program.c_str();
  size_t source_len = my_opencl_program.length();

  cl_program prog = clCreateProgramWithSource(my_context, 1, &ocl_prog, &source_len, &err); OPENCL_ERR_CHECK(err);
  err = clBuildProgram(prog, 0, NULL, NULL, NULL, NULL);

  //
  // Print compiler errors if there was a problem:
  //
  if (err != CL_SUCCESS) {

    char *build_log;
    size_t ret_val_size;
    err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, 0, NULL, &ret_val_size);
    build_log = (char *)malloc(sizeof(char) * (ret_val_size+1));
    err = clGetProgramBuildInfo(prog, my_device_id, CL_PROGRAM_BUILD_LOG, ret_val_size, build_log, NULL);
    build_log[ret_val_size] = '\0'; // terminate string
    std::cout << "Log: " << build_log << std::endl;
    free(build_log);
    std::cout << "OpenCL program sources: " << std::endl << my_opencl_program << std::endl;
    return EXIT_FAILURE;
  }
  //
  // Extract the only kernel in the program:
```

```
  //

#ifdef SLOW_KERNEL
  cl_kernel my_kernel = clCreateKernel(prog, "ocl_csr_matvec", &err); OPENCL_ERR_CHECK(err);
#endif
#ifdef FAST_KERNEL
  cl_kernel my_kernel = clCreateKernel(prog, "ocl_csr_matvec", &err); OPENCL_ERR_CHECK(err);
#endif

  //
  /////////////////////////// Part 3: Create memory buffers ///////////////////////////////////
  //

  cl_uint vector_size = N;
  size_t  local_size = LOCAL_SIZE;
  size_t global_size = GLOBAL_SIZE*GLOBAL_SIZE;
  size_t groups = 1 + int(N/LOCAL_SIZE);

  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);

  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);

  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);

  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);

  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);

  //
  /////////////////////////// Part 4: Run kernel ///////////////////////////////////
  //
  //
  // Set kernel arguments:
  //

// "__kernel void ocl_csr_matvec(size_t N,\n"
// "                        __global int *csr_rowoffsets,\n"
// "                        __global int *csr_colindices,\n"
// "                        __global double *csr_values,\n"
// "                        __global double const *x, __global double *y)\n"
  err = clSetKernelArg(my_kernel, 0, sizeof(cl_uint), (void*)&vector_size); OPENCL_ERR_CHECK(err);
  err = clSetKernelArg(my_kernel, 1, sizeof(cl_mem),  (double*)&ocl_csr_rowoffsets); OPENCL_ERR_CHECK(err);
  err = clSetKernelArg(my_kernel, 2, sizeof(cl_mem),  (double*)&ocl_csr_colindices); OPENCL_ERR_CHECK(err);
  err = clSetKernelArg(my_kernel, 3, sizeof(cl_mem),  (double*)&ocl_csr_values); OPENCL_ERR_CHECK(err);
  err = clSetKernelArg(my_kernel, 4, sizeof(cl_mem),  (double*)&ocl_x); OPENCL_ERR_CHECK(err);
  err = clSetKernelArg(my_kernel, 5, sizeof(cl_mem),  (double*)&ocl_y); OPENCL_ERR_CHECK(err);

  //
  // Enqueue kernel in command queue:
  //
  timer1.reset();
  err = clEnqueueNDRangeKernel(my_queue, my_kernel, 1, NULL, &global_size, &local_size, 0, NULL, NULL); OPENCL_ERR_CHECK(err);

  // wait for all operations in queue to finish:
  err = clFinish(my_queue); OPENCL_ERR_CHECK(err);
  double ocl_time = timer1.get();

```

```cpp
  //
  ////////////////////////// Part 5: Get data from OpenCL buffer //////////////////////////////////
  //
  err = clEnqueueReadBuffer(my_queue, ocl_y, CL_TRUE, 0, sizeof(ScalarType)*vector_size, y, 0, NULL, NULL); OPENCL_ERR_CHECK(err);

  // wait for all operations in queue to finish:
  err = clFinish(my_queue); OPENCL_ERR_CHECK(err);
  //
  // cleanup
  //
  clReleaseMemObject(ocl_x);
  clReleaseMemObject(ocl_y);
  clReleaseMemObject(ocl_csr_rowoffsets);
  clReleaseMemObject(ocl_csr_colindices);
  clReleaseMemObject(ocl_csr_values);
  clReleaseProgram(prog);
  clReleaseCommandQueue(my_queue);
  clReleaseContext(my_context);

  std::cout << "From OCL benchmark: " << ocl_time << std::endl;

  return ocl_time;
}


/** Solve a system with `points_per_direction * points_per_direction` unknowns */
void benchmark_matvec(size_t points_per_direction, size_t max_nonzeros_per_row,
                      void (*generate_matrix)(size_t, int*, int*, double*),
                      std::string gen_type) // function pointer parameter
{

  size_t N = points_per_direction * points_per_direction; // number of rows and columns
  std::fstream csv;
  csv.open(csv_name, std::fstream::out | std::fstream::app);

  //
  // Allocate CSR arrays.
  //
  // Note: Usually one does not know the number of nonzeros in the system matrix a-priori.
  //       For this exercise, however, we know that there are at most 5 nonzeros per row in the system matrix, so we can allocate accordingly.
  //
  int *csr_rowoffsets =    (int*)malloc(sizeof(double) * (N+1));
  int *csr_colindices =    (int*)malloc(sizeof(double) * max_nonzeros_per_row * N);
  double *csr_values  = (double*)malloc(sizeof(double) * max_nonzeros_per_row * N);

  //
  // fill CSR matrix with values
  //
  generate_matrix(points_per_direction, csr_rowoffsets, csr_colindices, csr_values);

  //
  // Allocate vectors:
  //
  double *x = (double*)malloc(sizeof(double) * N); std::fill(x, x + N, 1);
  double *y = (double*)malloc(sizeof(double) * N); std::fill(y, y + N, 0);
  double *y_ocl = (double*)malloc(sizeof(double) * N); std::fill(y, y + N, 0);

  //
  // Call matrix-vector product reference
  //
  Timer timer;
  timer.reset();
  csr_matvec_product(N, csr_rowoffsets, csr_colindices, csr_values, x, y);
  double cpu_time = timer.get();

  //
  // Call matrix-vector product kernel
  //
  double ocl_time = benchmark_ocl(N, max_nonzeros_per_row, csr_rowoffsets, csr_colindices,
    csr_values, x, y_ocl);
  std::cout << "Reference: " << std::endl;
  printContainer(y, N, 10);
  std::cout << "OpenCL: " << std::endl;
  printContainer(y_ocl, N, 10);
  double difference = diff_norm(y_ocl, y, N);
  bool check_res = check(y_ocl, y, N);
  std::cout << "Difference between the two: " << difference << " (check: " << check_res << std::endl;
  std::cout << "Time for ref product: " << cpu_time << std::endl;
  std::cout << "Time for OCL product: " << ocl_time << std::endl;

  csv << N<< ";"
    << points_per_direction << ";"
    << target << ";"
    << gen_type << ";"
    << ocl_time << ";"
    << cpu_time << ";"
    << difference << ";"
    << check_res << std::endl;

  csv.close();
  free(x);
  free(y);
  free(y_ocl);
  free(csr_rowoffsets);
  free(csr_colindices);
  free(csr_values);
}


int main() {
  std::fstream csv;
#ifdef TRUNC_CSV
  csv.open(csv_name, std::fstream::out | std::fstream::trunc);
  csv << "N;points_per_direction;target;gen_type;ocl_time;cpu_time;check;diff_norm" << std::endl;
#endif
#ifndef TRUNC_CSV
  csv.open(csv_name, std::fstream::out | std::fstream::app);
#endif

  uint pp = PP;
  std::cout << "# Benchmarking finite difference matrix" << std::endl;
  benchmark_matvec(pp, 5, generate_fdm_laplace, "1"); // 100*100 unknowns, finite difference matrix
  // the last string is just so that I know which matrix was used

  std::cout << "# Benchmarking special matrix" << std::endl;
  benchmark_matvec(pp, 2000, generate_matrix2, "2");     // 100*100 unknowns, special matrix with 200-2000 nonzeros per row

  std::cout << "Data: \nRuntimes in csv form can be found here\nhttps://gtx1080.360252.org/2020/ex7/" << csv_name << std::endl;

  return EXIT_SUCCESS;
}
```