INSTITUTE OF MICROELECTRONICS

360.252 Computational Science on Many-Core Architectures

Exercise 10

Author:
Peter Holzner, 01426733

Submission: January 19, 2021

Contents

1 Introduction 1

2 HIP 1

3 SyCL 3
3.1 My troubles with the CG algorithm continue... 3

4 Code and Kernels 5

1 Introduction

"Vielleicht, daher ist es seltsam, dass, wenn es irgendeine eine Phrase, die garantiert wird, um mich auf den Weg, es ist, wenn jemand zu mir sagt: 'Okay, fein. Du bist derChef!', Sagt Branson. Was michä argert ist, dass in 90 Prozent der Fälle, wie, was diese Person wirklich sagen will, ist: 'Okay, dann, glaube ich nicht mit Ihnen einverstanden,aber ich werde rollen und tun es weil sie sagen mir zu. Aber wenn es nicht klappt werdeich der Erste sein, der daran erinnern, dass es nicht meine Idee'."
- Christine Aschbacher (feat. Google Translate?), 2020

Fig. 1. God(zilla) is dead.

2 HIP

Code listings for this task:

- Classical CG - HIP: Listing 1
- Classical CG - CUDA: Listing 2

I expect nothing...to change in terms of performance. As we discussed, hip-nvcc simply converts HIP code into CUDA coda and then hands it over to nvcc.

A good test setup would include running each code multiple times and evaluating statistical measures of the recorded runtimes (mean, standard deviation, etc.). Unfortunately, to implement this would for this exercise would/will be quite the hassle (read: take a lot of time) and not be very interesting. So, I've decided to use a very simple test setup (run each benchmark once per N) and compare a couple different runs per hand, knowing that it isn't really thorough or conclusive this way. I wanted to focus my time more on the second task (SyCL) and might rework this part, if I have time (if you read this: I did not have enough time.).

With that out of the way, in Fig. 2 you can see a representative sample of my tests. As expected, performance (efficiency) between the two versions is indeed identical (within a very small margin). The speedup plot on the right side illustrates this well ($Speedup = 1$ means equal efficiency). Both implementations are also equally precise - as expected - which is illustrated in Fig. 3

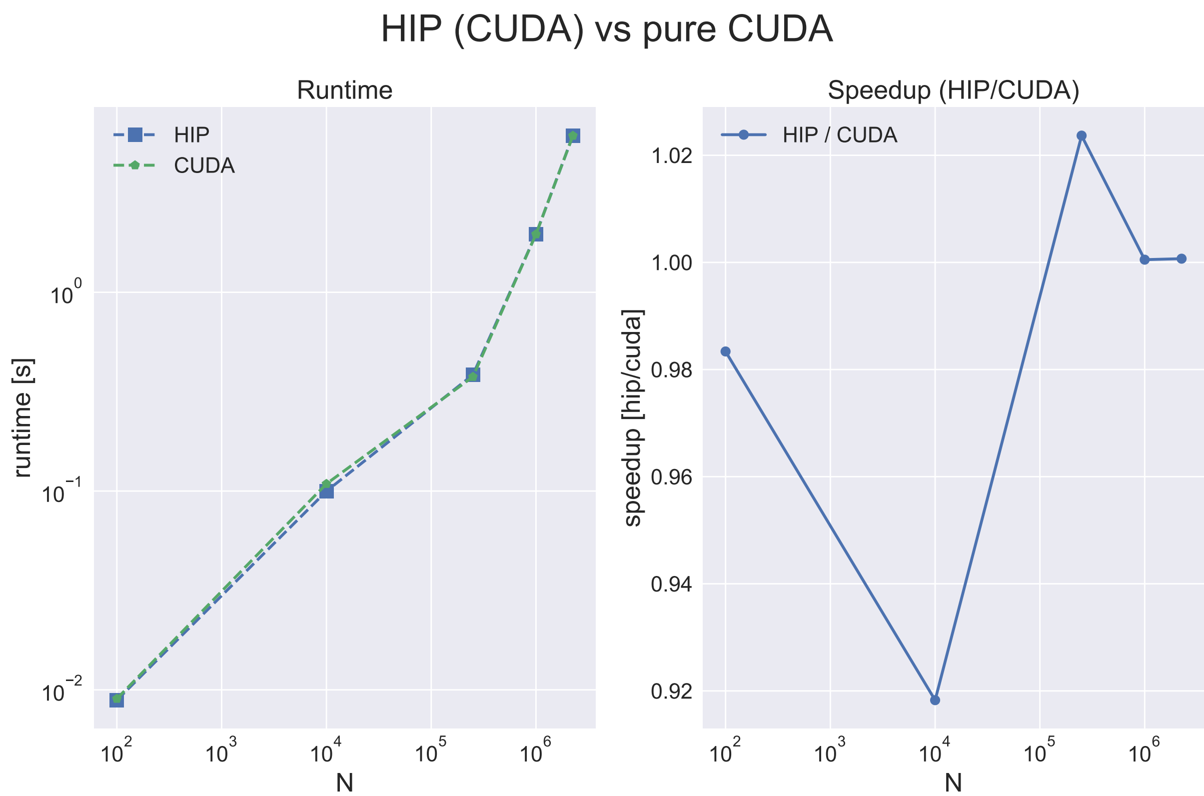

Fig. 2. HIP vs CUDA: efficiency.

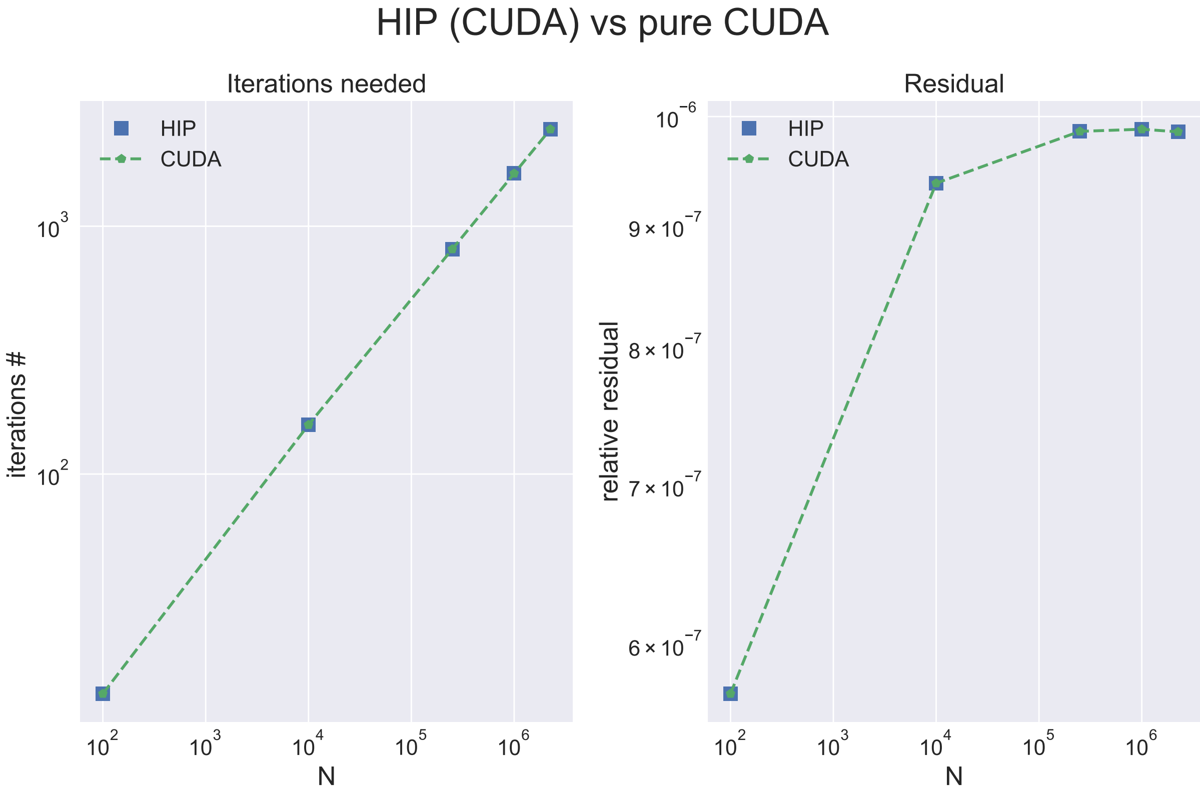

Fig. 3. HIP vs CUDA: precision.

3 SyCL

- SyCL: Listing 3
- Host: Listing 4

I'd like to lead with the Urban Dictionary definition of C++:

> "A programming language for Real Men. Most languages try to provide a simplified way to solve specific problems well. C++ makes no such concession and tries to be mediocre at everything. It lets you program at a very high level, and a very low level in the same program. It lets you write procedural code, object oriented code, generic code and mix them all up. It makes you decide everything and provides no help if you get it wrong.
>
> It is by far the biggest, most complicated, ugly, down-right dangerous language you can use. But it does run fast. It takes at least twice as long to program in C++ as any other major language (except C).
>
> The men who program in C++ are Real Men. The women who program in C++ are Real Men too. You can spot a C++ programmer from their testosterone fueled swagger, and the unbelievable amount of contempt they inject into the phrase Java "programmer". They'll probably do the air quotes and all."

Somehow, this definition resonated with me especially during the programming I did for this task.[1]

3.1 My troubles with the CG algorithm continue...

For some reason, I've always had trouble when implementing this particular algorithm. I didn't manage to get the SyCL version to work properly. I was only able to check the vector addition and dot product SyCL-based functions for correctness (limited test cases of course).

However, to still get some runtimes to compare to the host version, I first ran the host version. I noted the number of iterations needed for convergence and forced the SyCl to run for the same number of iterations. The numerical results of my benchmarks can be found in the tables Table 1 and Table 2. The runtimes (and runtime per iteration) are also visualized in Fig. 4 and the results are in... my SyCL implementation (I used 8 workgroups with 8 threads) is also terrible in terms of efficiency.

Overall, my experience with SyCL wasn't too bad though. The C++ - hurdle is real and the documentation is dense. If I'd not had some prior experience with CUDA or OpenCL, I doubt I'd have been able to understand anything (quickly). I particularly liked the concept of the buffers as I think they give nice control over the managed data and work well in conjunction with std-lib containers. Although this last exercise wasn't filled with success for me, I hope you at least had some fun with my ~~memes~~ report!

[1] I don't subscribe to this whole "Real Men and Java is inferior thing" - I just found the definition funny and stumbled on it while working on this exercise.

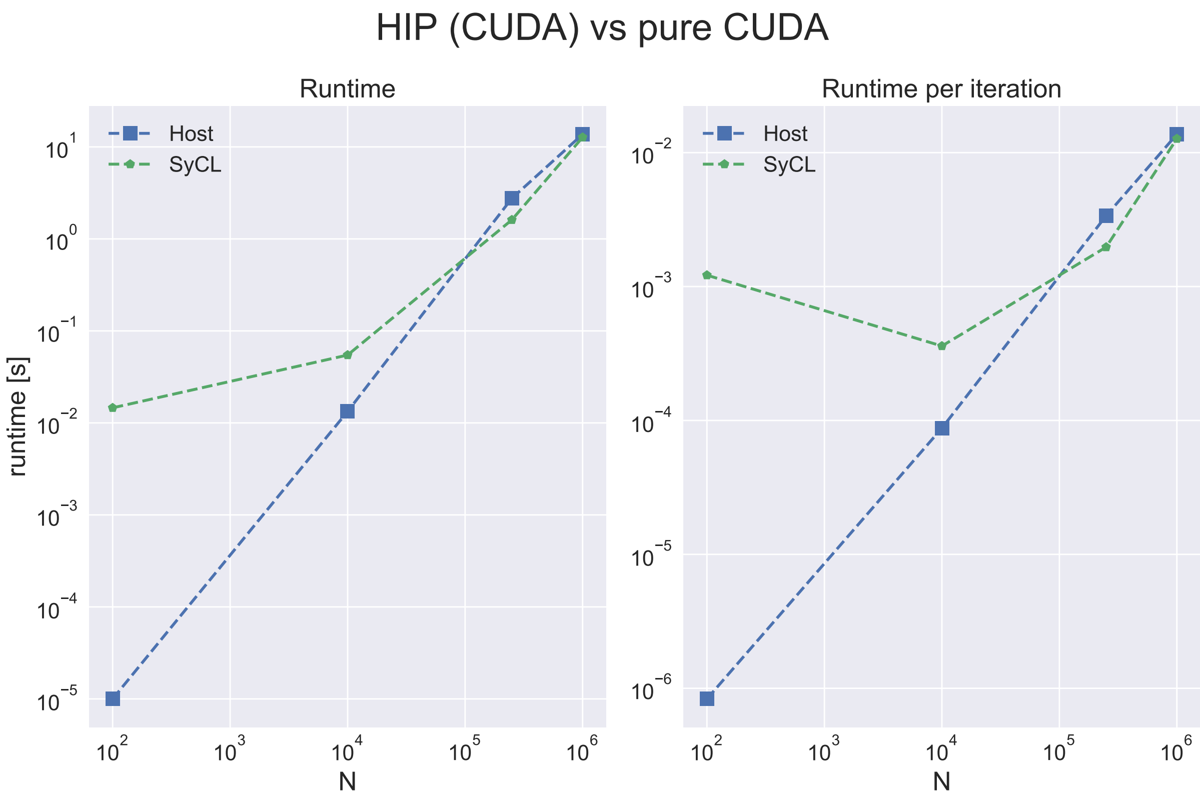

Fig. 4. Host vs Sycl: efficiency.

	p	N	runtime	residual	iterations	runtime/iter
0	10	100	0.000010	1.758420e-05	12	8.333333e-07
1	100	10000	0.013327	2.729190e-06	152	8.767763e-05
2	500	250000	2.763240	6.259210e-07	818	3.378044e-03
3	1000	1000000	13.724200	4.207000e-02	1001	1.371049e-02

Table 1: Host version results

	p	N	runtime	residual	iterations	runtime/iter
0	10	100	0.014589	2.656940e+07	12	0.001216
1	100	10000	0.054664	7.907860e+75	152	0.000360
2	500	250000	1.611170	NaN	818	0.001970
3	1000	1000000	12.724800	NaN	1002	0.012699

Table 2: SyCL version results

4 Code and Kernels

Listings

1 Ex10: Classical CG - HIP version 6
2 Ex10: Classical CG - CUDA version 11
3 Ex10: Classical CG - (HIP)SyCL version 15
4 Ex10: Classical CG - HOST version 22

Listing 1: Ex10: Classical CG - HIP version

```
#include "poisson2d.hpp"
#include "timer.hpp"
#include <algorithm>
#include <iostream>
#include <fstream>
#include <vector>
// #include <stdio.h>
#include "hip/hip_runtime.h"

// DEFINES
#define EX "ex10"
#define CSV_NAME "ph_data_hip.csv"
#define N_MAX_PRINT 32
#define PRINT_ONLY 10
#define NUM_TESTS 10 // should be uneven so we dont have to copy after each iteration

#define GRID_SIZE 512
#define BLOCK_SIZE 512
#define USE_MY_ATOMIC_ADD
#define HIP_ASSERT(x) (assert((x)==hipSuccess)) // only used it once

/** atomicAdd for doubles for hip for nvcc for many cores exercise 10 for me
 * by: Peter HOLZNER feat. NVIDIA
 *
 * - Ref: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions
 *
 * 'Don't let your memes be dreams!'
 * - Probably Ghandi, idk
 */
__device__ double
my_atomic_Add(double* address, double val)
{
  using ulli = unsigned long long int;
  ulli* address_as_ull =
                            (ulli*)address;
  ulli old = *address_as_ull, assumed;
  do {
      assumed = old;
      old = atomicCAS(address_as_ull, assumed,
                      __double_as_longlong(val +
                              __longlong_as_double(assumed)));

  } while (assumed != old);
  return __longlong_as_double(old);
};

// y = A * x
__global__ void
hip_csr_matvec_product(int N,
                       int *csr_rowoffsets, int *csr_colindices,
                       double *csr_values,
                       double *x, double *y)
{
  for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i += hipBlockDim_x *
      hipGridDim_x) {
    double sum = 0;
    for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++) {
      sum += csr_values[k] * x[csr_colindices[k]];
    }
    y[i] = sum;
  }
}

// x <- x + alpha * y
__global__ void
hip_vecadd(int N, double *x, double *y, double alpha)
{
  for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i += hipBlockDim_x *
      hipGridDim_x)
    x[i] += alpha * y[i];
```

```
}

// x <- y + alpha * x
__global__ void
hip_vecadd2(int N, double *x, double *y, double alpha)
{
  for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i += hipBlockDim_x * hipGridDim_x)
    x[i] = y[i] + alpha * x[i];
}

/**result = (x, y)
 */
__global__ void
hip_dot_product(int N, double *x, double *y, double *result)
{
  __shared__ double shared_mem[BLOCK_SIZE];

  double dot = 0;
  for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i += hipBlockDim_x * hipGridDim_x) {
    dot += x[i] * y[i];
  }

  shared_mem[hipThreadIdx_x] = dot;
  for (int k = hipBlockDim_x / 2; k > 0; k /= 2) {
    __syncthreads();
    if (hipThreadIdx_x < k) {
      shared_mem[hipThreadIdx_x] += shared_mem[hipThreadIdx_x + k];
    }
  }

  if (hipThreadIdx_x == 0)
  {
#ifdef USE_MY_ATOMIC_ADD
    my_atomic_Add(result, shared_mem[0]);
#else
    atomicAdd(result, shared_mem[0]);
#endif
  }
}



/** Implementation of the conjugate gradient algorithm.
 *
 *  The control flow is handled by the CPU.
 *  Only the individual operations (vector updates, dot products, sparse
 * matrix-vector product) are transferred to hip kernels.
 *
 *  The temporary arrays p, r, and Ap need to be allocated on the GPU for use
 * with hip. Modify as you see fit.
 *
 * Modifications:
 * - returns runtime as double
 * - iteration counter (iters) is passed as reference for logging to csv-file
 * - replaced cuda with hip (literally search-and-replaced the word...)
 * - implemented the hip-style kernel launches (although unnecessary for this
 *   exercise since we pass it to nvcc anyway :D)
 */
double conjugate_gradient(int N, // number of unknows
                          int *csr_rowoffsets, int *csr_colindices,
                          double *csr_values, double *rhs, double *solution,
                          int& iters)
//, double *init_guess)   // feel free to add a nonzero initial guess as needed
{
  // initialize timer
  Timer timer;

  // clear solution vector (it may contain garbage values):
  std::fill(solution, solution + N, 0);

```

```
  // initialize work vectors:
  double alpha, beta;
  double *hip_solution, *hip_p, *hip_r, *hip_Ap, *hip_scalar;
  hipMalloc(&hip_p, sizeof(double) * N);
  hipMalloc(&hip_r, sizeof(double) * N);
  hipMalloc(&hip_Ap, sizeof(double) * N);
  hipMalloc(&hip_solution, sizeof(double) * N);
  hipMalloc(&hip_scalar, sizeof(double));

  hipMemcpy(hip_p, rhs, sizeof(double) * N, hipMemcpyHostToDevice);
  hipMemcpy(hip_r, rhs, sizeof(double) * N, hipMemcpyHostToDevice);
  hipMemcpy(hip_solution, solution, sizeof(double) * N, hipMemcpyHostToDevice);

  const double zero = 0;
  double residual_norm_squared = 0;

  hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice);

  // hip_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, hip_r, hip_r, hip_scalar);
  hipLaunchKernelGGL(hip_dot_product, // kernel
    GRID_SIZE, BLOCK_SIZE,            // device params
    0, 0,                             // shared mem, default stream
    N, hip_r, hip_r, hip_scalar       // kernel arguments
  );

  hipMemcpy(&residual_norm_squared, hip_scalar, sizeof(double), hipMemcpyDeviceToHost);

  double initial_residual_squared = residual_norm_squared;

  iters = 0;
  hipDeviceSynchronize();
  timer.reset();
  while (1) {

    // line 4: A*p:
    // hip_csr_matvec_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, csr_rowoffsets, csr_colindices,
        csr_values, hip_p, hip_Ap);
    hipLaunchKernelGGL(hip_csr_matvec_product, // kernel
      GRID_SIZE, BLOCK_SIZE,            // device params
      0, 0,                             // shared mem, default stream
      N, csr_rowoffsets, csr_colindices, csr_values, hip_p, hip_Ap       // kernel arguments
    );

    // lines 5,6:
    hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice);
    // hip_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, hip_p, hip_Ap, hip_scalar);
    hipLaunchKernelGGL(hip_dot_product, // kernel
      GRID_SIZE, BLOCK_SIZE,            // device params
      0, 0,                             // shared mem, default stream
      N, hip_p, hip_Ap, hip_scalar      // kernel arguments
    );
    hipMemcpy(&alpha, hip_scalar, sizeof(double), hipMemcpyDeviceToHost);
    alpha = residual_norm_squared / alpha;

    // line 7:
    // hip_vecadd<<<GRID_SIZE, BLOCK_SIZE>>>(N, hip_solution, hip_p, alpha);
    hipLaunchKernelGGL(hip_vecadd,    // kernel
      GRID_SIZE, BLOCK_SIZE,          // device params
      0, 0,                           // shared mem, default stream
      N, hip_solution, hip_p, alpha   // kernel arguments
    );

    // line 8:
    // hip_vecadd<<<GRID_SIZE, BLOCK_SIZE>>>(N, hip_r, hip_Ap, -alpha);
    hipLaunchKernelGGL(hip_vecadd, // kernel
      GRID_SIZE, BLOCK_SIZE,       // device params
      0, 0,                        // shared mem, default stream
      N, hip_r, hip_Ap, -alpha     // kernel arguments
    );

    // line 9:
    beta = residual_norm_squared;
```

```
    HIP_ASSERT(hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice)); // just checking if this works properly
    // hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice);
    // hip_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, hip_r, hip_r, hip_scalar);
    hipLaunchKernelGGL(hip_dot_product, // kernel
      GRID_SIZE, BLOCK_SIZE,            // device params
      0, 0,                             // shared mem, default stream
      N, hip_r, hip_r, hip_scalar       // kernel arguments
    );
    hipMemcpy(&residual_norm_squared, hip_scalar, sizeof(double), hipMemcpyDeviceToHost);

    // line 10:
    if (std::sqrt(residual_norm_squared / initial_residual_squared) < 1e-6) {
      break;
    }

    // line 11:
    beta = residual_norm_squared / beta;

    // line 12:
    // hip_vecadd2<<<GRID_SIZE, BLOCK_SIZE>>>(N, hip_p, hip_r, beta);
    hipLaunchKernelGGL(hip_vecadd2, // kernel
      GRID_SIZE, BLOCK_SIZE,            // device params
      0, 0,                             // shared mem, default stream
      N, hip_p, hip_r, beta      // kernel arguments
    );

    if (iters > 10000)
      break; // solver didn't converge
    ++iters;
  }
  hipMemcpy(solution, hip_solution, sizeof(double) * N, hipMemcpyDeviceToHost);

  hipDeviceSynchronize();
  double runtime = timer.get();
  std::cout << "Time elapsed: " << runtime << " (" << runtime / iters << " per iteration)"
      << std::endl;

  if (iters > 10000)
    std::cout << "Conjugate Gradient did NOT converge within 10000 iterations"
              << std::endl;
  else
    std::cout << "Conjugate Gradient converged in " << iters << " iterations."
              << std::endl;

  hipFree(hip_p);
  hipFree(hip_r);
  hipFree(hip_Ap);
  hipFree(hip_solution);
  hipFree(hip_scalar);

  return runtime;
}

/** Solve a system with `points_per_direction * points_per_direction` unknowns
 */
void solve_system(int points_per_direction) {

  int N = points_per_direction *
          points_per_direction; // number of unknows to solve for

  std::cout << "Solving Ax=b with " << N << " unknowns." << std::endl;

  //
  // 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) * 5 * N);
  double *csr_values = (double *)malloc(sizeof(double) * 5 * N);

  int *hip_csr_rowoffsets, *hip_csr_colindices;
  double *hip_csr_values;
  //
  // fill CSR matrix with values
  //
  generate_fdm_laplace(points_per_direction, csr_rowoffsets, csr_colindices,
                       csr_values);

  //
  // Allocate solution vector and right hand side:
  //
  double *solution = (double *)malloc(sizeof(double) * N);
  double *rhs = (double *)malloc(sizeof(double) * N);
  std::fill(rhs, rhs + N, 1);

  //
  // Allocate hip-arrays //
  //
  hipMalloc(&hip_csr_rowoffsets, sizeof(double) * (N + 1));
  hipMalloc(&hip_csr_colindices, sizeof(double) * 5 * N);
  hipMalloc(&hip_csr_values, sizeof(double) * 5 * N);
  hipMemcpy(hip_csr_rowoffsets, csr_rowoffsets, sizeof(double) * (N + 1),
      hipMemcpyHostToDevice);
  hipMemcpy(hip_csr_colindices, csr_colindices, sizeof(double) * 5 * N,
      hipMemcpyHostToDevice);
  hipMemcpy(hip_csr_values,     csr_values,     sizeof(double) * 5 * N,
      hipMemcpyHostToDevice);

  //
  // Call Conjugate Gradient implementation with GPU arrays
  //
  int iters = 0; // pass into the CG so we can track it
  double runtime = conjugate_gradient(N, hip_csr_rowoffsets, hip_csr_colindices,
      hip_csr_values, rhs, solution, iters);

  //
  // Check for convergence:
  //
  double residual_norm = relative_residual(N, csr_rowoffsets, csr_colindices, csr_values,
      rhs, solution);
  std::cout << "Relative residual norm: " << residual_norm
            << " (should be smaller than 1e-6)" << std::endl;

  // not optimal (efficient), but minimally invasive --> easy to copy
  std::ofstream csv;
  csv.open(CSV_NAME, std::fstream::out | std::fstream::app);
  csv << points_per_direction << ";"
    << N << ";"
    << runtime << ";"
    << residual_norm << ";"
    << iters << std::endl;
  csv.close();

  for (int i = 0; i < N; i++)
    std::cout << solution[i] << std::endl;

  hipFree(hip_csr_rowoffsets);
  hipFree(hip_csr_colindices);
  hipFree(hip_csr_values);
  free(solution);
  free(rhs);
  free(csr_rowoffsets);
  free(csr_colindices);
  free(csr_values);
}

int main() {

  std::ofstream csv;
```

```
  csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
  csv << "p;N;runtime;residual;iterations" << std::endl;
  csv.close();

  hipDeviceProp_t devProp;
  hipGetDeviceProperties(&devProp, 0);
  std::cout << " System minor " << devProp.minor << std::endl;
  std::cout << " System major " << devProp.major << std::endl;
  std::cout << " agent prop name " << devProp.name << std::endl;
  std::cout << "hip Device prop succeeded " << std::endl ;

  // std::vector<int> p_per_dir{ 10, 100, 500,1000, 1500};

  std::vector<int> p_per_dir{ 10};

  for (auto& p : p_per_dir)
  {
    std::cout << "---------------------------" << std::endl;
    solve_system(p); // solves a system with p*p unknowns
  }
  std::cout << "\nData: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;

  return EXIT_SUCCESS;
}
```

Listing 2: Ex10: Classical CG - CUDA version

```
#include "poisson2d.hpp"
#include "timer.hpp"
#include <algorithm>
#include <iostream>
#include <fstream>
#include <vector>

// DEFINES
#define EX "ex10"
#define CSV_NAME "ph_data_cuda.csv"
#define N_MAX_PRINT 32
#define PRINT_ONLY 10
#define NUM_TESTS 10 // should be uneven so we dont have to copy after each iteration

#define GRID_SIZE 512
#define BLOCK_SIZE 512
#define USE_MY_ATOMIC_ADD

/** atomicAdd for doubles for hip for nvcc for many cores exercise 10 for me
 * by: Peter HOLZNER feat. NVIDIA
 *
 * - Ref: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions
 *
 * 'Don't let your memes be dreams!'
 * - Probably Ghandi, idk
 */
 __device__ double
 my_atomic_Add(double* address, double val)
 {
   using ulli = unsigned long long int;
   ulli* address_as_ull =
                             (ulli*)address;
   ulli old = *address_as_ull, assumed;
   do {
       assumed = old;
       old = atomicCAS(address_as_ull, assumed,
                       __double_as_longlong(val +
                               __longlong_as_double(assumed)));

   } while (assumed != old);
   return __longlong_as_double(old);
 };

/** y = A * x
 */
```

```
__global__ void cuda_csr_matvec_product(int N, int *csr_rowoffsets,
                                        int *csr_colindices, double *csr_values,
                                        double *x, double *y)
{
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x) {
    double sum = 0;
    for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++) {
      sum += csr_values[k] * x[csr_colindices[k]];
    }
    y[i] = sum;
  }
}

/** x <- x + alpha * y
 */
__global__ void cuda_vecadd(int N, double *x, double *y, double alpha)
{
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
    x[i] += alpha * y[i];
}

/** x <- y + alpha * x
 */
__global__ void cuda_vecadd2(int N, double *x, double *y, double alpha)
{
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
    x[i] = y[i] + alpha * x[i];
}

/**result = (x, y)
 */
__global__ void cuda_dot_product(int N, double *x, double *y, double *result)
{
  __shared__ double shared_mem[BLOCK_SIZE];

  double dot = 0;
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x) {
    dot += x[i] * y[i];
  }

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

  if (threadIdx.x == 0)
  {
    #ifdef USE_MY_ATOMIC_ADD
      my_atomic_Add(result, shared_mem[0]);
    #else
      atomicAdd(result, shared_mem[0]);
    #endif
  }
}



/** Implementation of the conjugate gradient algorithm.
 *
 *  The control flow is handled by the CPU.
 *  Only the individual operations (vector updates, dot products, sparse
 * matrix-vector product) are transferred to CUDA kernels.
 *
 *  The temporary arrays p, r, and Ap need to be allocated on the GPU for use
 * with CUDA. Modify as you see fit.
 *
 * Modifications:
 * returns runtime as double
 * iteration counter (iters) is passed as reference for logging to csv-file
```

```
 */
double conjugate_gradient(int N, // number of unknows
                          int *csr_rowoffsets, int *csr_colindices,
                          double *csr_values, double *rhs, double *solution,
                          int& iters)
//, double *init_guess)   // feel free to add a nonzero initial guess as needed
{
  // initialize timer
  Timer timer;

  // clear solution vector (it may contain garbage values):
  std::fill(solution, solution + N, 0);

  // initialize work vectors:
  double alpha, beta;
  double *cuda_solution, *cuda_p, *cuda_r, *cuda_Ap, *cuda_scalar;
  cudaMalloc(&cuda_p, sizeof(double) * N);
  cudaMalloc(&cuda_r, sizeof(double) * N);
  cudaMalloc(&cuda_Ap, sizeof(double) * N);
  cudaMalloc(&cuda_solution, sizeof(double) * N);
  cudaMalloc(&cuda_scalar, sizeof(double));

  cudaMemcpy(cuda_p, rhs, sizeof(double) * N, cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_r, rhs, sizeof(double) * N, cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_solution, solution, sizeof(double) * N, cudaMemcpyHostToDevice);

  const double zero = 0;
  double residual_norm_squared = 0;
  cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
  cuda_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_r, cuda_r, cuda_scalar);
  cudaMemcpy(&residual_norm_squared, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);

  double initial_residual_squared = residual_norm_squared;

  iters = 0; // it's passed in from the outside
  cudaDeviceSynchronize();
  timer.reset();
  while (1) {

    // line 4: A*p:
    cuda_csr_matvec_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, csr_rowoffsets, csr_colindices,
        csr_values, cuda_p, cuda_Ap);

    // lines 5,6:
    cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
    cuda_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_p, cuda_Ap, cuda_scalar);
    cudaMemcpy(&alpha, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
    alpha = residual_norm_squared / alpha;

    // line 7:
    cuda_vecadd<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_solution, cuda_p, alpha);

    // line 8:
    cuda_vecadd<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_r, cuda_Ap, -alpha);

    // line 9:
    beta = residual_norm_squared;
    cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
    cuda_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_r, cuda_r, cuda_scalar);
    cudaMemcpy(&residual_norm_squared, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);

    // line 10:
    if (std::sqrt(residual_norm_squared / initial_residual_squared) < 1e-6) {
      break;
    }

    // line 11:
    beta = residual_norm_squared / beta;

    // line 12:
    cuda_vecadd2<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_p, cuda_r, beta);

```

```
    if (iters > 10000)
      break; // solver didn't converge
    ++iters;
  }
  cudaMemcpy(solution, cuda_solution, sizeof(double) * N, cudaMemcpyDeviceToHost);

  cudaDeviceSynchronize();
  double runtime = timer.get();
  std::cout << "Time elapsed: " << runtime << " (" << runtime / iters << " per iteration)"
      << std::endl;

  if (iters > 10000)
    std::cout << "Conjugate Gradient did NOT converge within 10000 iterations"
              << std::endl;
  else
    std::cout << "Conjugate Gradient converged in " << iters << " iterations."
              << std::endl;

  cudaFree(cuda_p);
  cudaFree(cuda_r);
  cudaFree(cuda_Ap);
  cudaFree(cuda_solution);
  cudaFree(cuda_scalar);

  return runtime;
}

/** Solve a system with `points_per_direction * points_per_direction` unknowns
 */
void solve_system(int points_per_direction) {

  int N = points_per_direction *
          points_per_direction; // number of unknows to solve for

  std::cout << "Solving Ax=b with " << N << " unknowns." << std::endl;

  //
  // 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) * 5 * N);
  double *csr_values = (double *)malloc(sizeof(double) * 5 * N);

  int *cuda_csr_rowoffsets, *cuda_csr_colindices;
  double *cuda_csr_values;
  //
  // fill CSR matrix with values
  //
  generate_fdm_laplace(points_per_direction, csr_rowoffsets, csr_colindices,
                       csr_values);

  //
  // Allocate solution vector and right hand side:
  //
  double *solution = (double *)malloc(sizeof(double) * N);
  double *rhs = (double *)malloc(sizeof(double) * N);
  std::fill(rhs, rhs + N, 1);

  //
  // Allocate CUDA-arrays //
  //
  cudaMalloc(&cuda_csr_rowoffsets, sizeof(double) * (N + 1));
  cudaMalloc(&cuda_csr_colindices, sizeof(double) * 5 * N);
  cudaMalloc(&cuda_csr_values, sizeof(double) * 5 * N);
  cudaMemcpy(cuda_csr_rowoffsets, csr_rowoffsets, sizeof(double) * (N + 1),
      cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_csr_colindices, csr_colindices, sizeof(double) * 5 * N,
```

```
        cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_csr_values,     csr_values,     sizeof(double) * 5 * N,
        cudaMemcpyHostToDevice);

  //
  // Call Conjugate Gradient implementation with GPU arrays
  //
  int iters = 0; // pass into the CG so we can track it
  double runtime = conjugate_gradient(N, cuda_csr_rowoffsets, cuda_csr_colindices,
        cuda_csr_values, rhs, solution, iters);

  //
  // Check for convergence:
  //
  double residual_norm = relative_residual(N, csr_rowoffsets, csr_colindices, csr_values,
        rhs, solution);
  std::cout << "Relative residual norm: " << residual_norm
            << " (should be smaller than 1e-6)" << std::endl;

  // not optimal (efficient), but minimally invasive --> easy to copy
  std::ofstream csv;
  csv.open(CSV_NAME, std::fstream::out | std::fstream::app);
  csv << points_per_direction << ";"
    << N << ";"
    << runtime << ";"
    << residual_norm << ";"
    << iters << std::endl;
  csv.close();

  cudaFree(cuda_csr_rowoffsets);
  cudaFree(cuda_csr_colindices);
  cudaFree(cuda_csr_values);
  free(solution);
  free(rhs);
  free(csr_rowoffsets);
  free(csr_colindices);
  free(csr_values);
}

int main() {
  std::ofstream csv;
  csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
  csv << "p;N;runtime;residual;iterations" << std::endl;
  csv.close();

  std::vector<int> p_per_dir{ 10, 100, 500,1000, 1500};

  for (auto& p : p_per_dir)
  {
    std::cout << "--------------------------" << std::endl;
    solve_system(p); // solves a system with p*p unknowns
  }
  std::cout << "\nData: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;

  return EXIT_SUCCESS;
}
```

Listing 3: Ex10: Classical CG - (HIP)SyCL version

```

#include <stdio.h>
#include <iostream>
#include <algorithm>
#include <numeric>
#include <vector>
#include <fstream>
#include "poisson2d.hpp"
#include "timer.hpp"
#include <CL/sycl.hpp>

// DEFINES
#define EX "ex10"
```

```
#define CSV_NAME "ph_data_sycl.csv"
#define N_MAX_PRINT 32
#define PRINT_ONLY 10
#define MAX_ITERS 100

#define GRID_SIZE 4
#define BLOCK_SIZE 4

int max_iters = 11;

/**by_sycl
 *
 * namespace for my sycl functions to dinstinguish them from normal C++ (host) functions.
 *
 * Design is inspired by the HIP/CUDA version.
 */
namespace by_sycl{
  namespace sycl = cl::sycl; // want to have shorter code.

  const int grid_size = GRID_SIZE;
  const int block_size = BLOCK_SIZE;

  /** ret = A*x
   * A... sparse matrix given in CSR format (3 arrays) of dense size NxN
   * x... vector of size A
   */
  void csr_Ax(sycl::queue& q,
              size_t N,
              sycl::buffer<int>& buff_csr_rowoffsets,
              sycl::buffer<int>& buff_csr_colindices,
              sycl::buffer<double>& buff_csr_values,
              sycl::buffer<double>& buff_x,
              sycl::buffer<double>& buff_y)
  {
    // std::cout << "Hello from by_sycl::csr_Ax!" << std::endl;
    // std::vector<double> y(a.size());

    // assert(("x should be length N", x.size() == N));
    // assert(("csr_rowoffsets should be length N+1", csr_rowoffsets.size() == (N+1)));


    // auto global_size = grid_size * block_size;
    // auto local_size = block_size;

    // sycl::nd_range<1> work_items(global_size, local_size);
    sycl::range<1> work_items{N};
    // Submitting a lambda to the queue that carries out the work:
    q.submit( [&](sycl::handler& cgh)
    {
      auto rowoffsets = buff_csr_rowoffsets.get_access<sycl::access::mode::read>(cgh);
      auto colidx = buff_csr_colindices.get_access<sycl::access::mode::read>(cgh);
      auto values = buff_csr_values.get_access<sycl::access::mode::read>(cgh);
      auto x = buff_x.get_access<sycl::access::mode::read>(cgh);
      auto y = buff_y.get_access<sycl::access::mode::write>(cgh);

      // The parallel section
      cgh.parallel_for<class csr_Ax>(work_items,
                                     [=] (sycl::id<1> row)
      {
        // for (int idx = row;)
        double tmp = 0;
        for (int jj = rowoffsets[row]; jj < rowoffsets[row+1]; ++jj)
          tmp += values[jj] * x[colidx[jj]];
        y[row] = tmp;
      }
      //
      );
    });
  }

  /** x += alpha * y
   *
```

```
 * x...     vector of size N
 * y...     vector of size N
 * alpha... scalar
 */
void inc_ay(sycl::queue& q,
            const size_t N,
            sycl::buffer<double>& buff_x,
            sycl::buffer<double>& buff_y,
            const double alpha)
{
  // std::cout << "Hello from by_sycl::inc_ay!" << std::endl;

  auto global_size = grid_size * block_size;
  auto local_size = block_size;

  sycl::nd_range<1> work_items(global_size, local_size);
  // Submitting a lambda to the queue that carries out the work:
  q.submit( [&](sycl::handler& cgh)
  {
    auto x = buff_x.get_access<sycl::access::mode::read_write>(cgh);
    auto y = buff_y.get_access<sycl::access::mode::read>(cgh);

    // The parallel section
    cgh.parallel_for<class inc_ay_work>(work_items,
                              [=] (sycl::nd_item<1> item)
    {
      size_t tid = item.get_global_linear_id();
      if (tid < N)
        x[tid] = x[tid] + alpha * y[tid];
    });
  });
}

/** x = y + alpha * x
 *
 * x...     vector of size N
 * y...     vector of size N
 * alpha... scalar
 */
void update(sycl::queue& q,
            const size_t N,
            sycl::buffer<double>& buff_x,
            sycl::buffer<double>& buff_y,
            const double alpha)
{
  // std::cout << "Hello from by_sycl::update!" << std::endl;

  auto global_size = grid_size * block_size;
  auto local_size = block_size;

  sycl::nd_range<1> work_items(global_size, local_size);
  // Submitting a lambda to the queue that carries out the work:
  q.submit( [&](sycl::handler& cgh)
  {
    auto x = buff_x.get_access<sycl::access::mode::read_write>(cgh);
    auto y = buff_y.get_access<sycl::access::mode::read>(cgh);

    // The parallel section
    cgh.parallel_for<class update_work>(work_items,
                                [=] (sycl::nd_item<1> item)
    {
      size_t tid = item.get_global_linear_id();
      if (tid < N)
        x[tid] = y[tid] + alpha * x[tid];
    });
  });
}


/** dot = (x, y)
 *
 * x...     vector of size N
```

```
   * y...      vector of size N
   */
  double dot(sycl::queue& q,
               const size_t N,
               sycl::buffer<double>& buff_x,
               sycl::buffer<double>& buff_y)
  {
    // std::cout << "Hello from by_sycl::dot!" << std::endl;
    double dot_result = 0;

    auto global_size = grid_size * block_size;
    auto local_size = block_size;

    sycl::nd_range<1> work_items(global_size, local_size);
    std::vector<double> result_host(grid_size, 0);
    { // this scope is for the result buffer to go out of scope "==" write to host
      sycl::buffer<double> buff_result(result_host.data(), result_host.size());
      q.submit( [&](sycl::handler& cgh)
      {
        auto x = buff_x.get_access<sycl::access::mode::read_write>(cgh);
        auto y = buff_y.get_access<sycl::access::mode::read>(cgh);
        auto result = buff_result.get_access<sycl::access::mode::write>(cgh);

        sycl::accessor<double, 1, sycl::access::mode::read_write, sycl::access::target::
            local> local_mem(sycl::range<1>(local_size), cgh);

        // The parallel section
        cgh.parallel_for<class dot_work>(
          work_items,
          [=] (sycl::nd_item<1> item)
        {
          size_t local_id = item.get_local_linear_id();
          size_t global_id = item.get_global_linear_id();

          local_mem[local_id] = 0;
          for (int idx = global_id; idx < N; idx += global_size)
          {
            local_mem[local_id] = x[idx] * y[idx];
          }
          item.barrier(sycl::access::fence_space::local_space);
          for (int k = local_size / 2; k > 0; k /= 2)
          {
            item.barrier(sycl::access::fence_space::local_space);
            if (local_id < k)
            {
              local_mem[local_id] += local_mem[local_id + k];
            }
          }

          if (local_id == 0)
          {
            result[item.get_group_linear_id()] = local_mem[0];
            // item.barrier(sycl::access::fence_space::local_space);
            // std::cout << "[WG#: " << item.get_group_linear_id() << "]Number of work
                groups:" << item.get_group_range(0)  << std::endl;
            // std::cout << "[WG#: " << item.get_group_linear_id() << "local_mem[0] = " <<
                local_mem[0] << std::endl;
          }
        });
      });
    }
    dot_result = std::accumulate(result_host.begin(), result_host.end(), 0);
    return dot_result;
  }
}

//
//// FUNCTIONS
//

namespace sycl = cl::sycl; // want to have shorter code.

```

```cpp
/** Implementation of the conjugate gradient algorithm.
 *
 *  The control flow is handled by the CPU.
 *  Only the individual operations (vector updates, dot products, sparse matrix-vector product) are transferred to CUDA kernels.
 */
double conjugate_gradient(size_t N,  // number of unknows
                          std::vector<int>& csr_rowoffsets,
                          std::vector<int>& csr_colindices,
                          std::vector<double>& csr_values,
                          std::vector<double>& rhs,
                          std::vector<double>& solution,
                          int& iters)
                          //, double *init_guess)   // feel free to add a nonzero initial guess as needed
{
  Timer timer;
  sycl::device device = sycl::default_selector{}.select_device();
  sycl::queue queue(device);

  // Check for potential local memory size
  auto has_local_mem = device.is_host()
      || (device.get_info<sycl::info::device::local_mem_type>()
      != sycl::info::local_mem_type::none);
  auto local_mem_size = device.get_info<sycl::info::device::local_mem_size>();
  if (!has_local_mem || local_mem_size < (by_sycl::block_size * sizeof(size_t)))
  {
      throw "Device doesn't have enough local memory!";
  }

  // clear solution vector (it may contain garbage values):
  std::fill(solution.begin(), solution.end(), 1.);

  // initialize work vectors:
  std::vector<double> Ap(N, 0);
  // line 2: initialize r and p:
  std::vector<double> p(rhs);
  std::vector<double> r(rhs);

  // Initialize buffers for matrix and solution
  sycl::buffer<int> rowoffsets_buff(csr_rowoffsets.data(), csr_rowoffsets.size());
  sycl::buffer<int> colidx_buff(csr_colindices.data(), csr_colindices.size());
  sycl::buffer<double> values_buff(csr_values.data(), csr_values.size());

  sycl::buffer<double> solution_buff(solution.data(), solution.size());

  sycl::buffer<double> r_buff(r.data(), r.size());
  sycl::buffer<double> Ap_buff(Ap.data(), Ap.size());
  sycl::buffer<double> p_buff(p.data(), p.size());

  iters = 0;

  double initial_residual_squared = by_sycl::dot(queue, N, r_buff, r_buff);
  double res_norm = 0;
  timer.reset();
  while (1) {
    // line 4: A*p:
    // csr_matvec_product(N, csr_rowoffsets, csr_colindices, csr_values, p, Ap);
    by_sycl::csr_Ax(queue,
                    N, rowoffsets_buff, colidx_buff, values_buff,
                    p_buff, Ap_buff);

    // lines 5,6:
    // double res_norm = 0;
    // for (size_t i=0; i<N; ++i) res_norm += r[i] * r[i];
    res_norm = by_sycl::dot(queue, N, r_buff, r_buff);

    // queue.wait_and_throw(); // necessary for Ap?

    // double alpha = 0;
    // for (size_t i=0; i<N; ++i) alpha += p[i] * Ap[i];
    double alpha = by_sycl::dot(queue, N, p_buff, Ap_buff);
```

```
    alpha = res_norm / alpha;
    // std::cout << "alpha(" << iters << ") = " << alpha << std::endl;

    // queue.wait_and_throw();

    // line 7,8:
    // for (size_t i=0; i<N; ++i) {
    //   solution[i] += alpha *  p[i];
    //   r[i]        -= alpha * Ap[i];
    // }
    by_sycl::inc_ay(queue, N, solution_buff, p_buff, alpha);
    by_sycl::inc_ay(queue, N, r_buff, Ap_buff, -alpha);

    double beta = res_norm;

    // lines 9, 10:
    // res_norm = 0;
    // for (size_t i=0; i<N; ++i) res_norm += r[i] * r[i];
    res_norm = by_sycl::dot(queue, N, r_buff, r_buff);
    if (std::sqrt( res_norm / initial_residual_squared ) < 1e-7)
      break;

    // line 11: compute beta
    beta = res_norm / beta;

    // line 12: update p
    // for (size_t i=0; i<N; ++i) p[i] = r[i] + beta * p[i];
    by_sycl::update(queue, N, p_buff, r_buff, beta);

    queue.wait_and_throw();
    if (iters > max_iters) break;  // solver didn't converge
    ++iters;

  }
  double runtime = timer.get();
  std::cout << "Time elapsed: " << runtime << " (" << runtime / iters << " per iteration)"
      << std::endl;
  std::cout << "Norm in CG: " << res_norm << std::endl;

  if (iters > MAX_ITERS)
    std::cout << "Conjugate Gradient did NOT converge within " << MAX_ITERS << " iterations"
              << std::endl;
  else
    std::cout << "Conjugate Gradient converged in " << iters << " iterations."
              << std::endl;

  return runtime;
}



/** Solve a system with `points_per_direction * points_per_direction` unknowns */
void solve_system(size_t points_per_direction) {

  size_t N = points_per_direction * points_per_direction; // number of unknows to solve for

  //
  // 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) * 5 * N);
  // double *csr_values  = (double*)malloc(sizeof(double) * 5 * N);
  std::vector<int> csr_rowoffsets(N+1);
  std::vector<int> csr_colindices(5*N);
  std::vector<double> csr_values(5*N);

  //
  // fill CSR matrix with values
```

```
  //
  std::cout << "Generating FDM..." << std::endl;
  generate_fdm_laplace(points_per_direction, csr_rowoffsets.data(), csr_colindices.data(),
      csr_values.data());

  //
  // Allocate solution vector and right hand side:
  //
  // double *solution = (double*)malloc(sizeof(double) * N);
  // double *rhs      = (double*)malloc(sizeof(double) * N);
  std::cout << "Initializing vectors..." << std::endl;
  std::vector<double> solution(N, 0);
  std::vector<double> rhs(N, 1.);
  // std::fill(rhs, rhs + N, 1);

  //
  // Call Conjugate Gradient implementation
  //
  int iters = 0;
  std::cout << "Starting CG" << std::endl;
  double runtime = conjugate_gradient(N, csr_rowoffsets, csr_colindices, csr_values, rhs,
      solution, iters);

  //
  // Check for convergence:
  //
  std::cout << "Calculating residual" << std::endl;
  double residual_norm = relative_residual(N, csr_rowoffsets.data(), csr_colindices.data(),
      csr_values.data(), rhs.data(), solution.data());
  std::cout << "Relative residual norm: " << residual_norm
            << " (should be smaller than 1e-6)" << std::endl;

  // for (auto & x : solution)
  //   std::cout << x << std::endl;

  // not optimal (efficient), but minimally invasive --> easy to copy
  std::ofstream csv;
  csv.open(CSV_NAME, std::fstream::out | std::fstream::app);
  csv << points_per_direction << ";"
    << N << ";"
    << runtime << ";"
    << residual_norm << ";"
    << iters << std::endl;
  csv.close();

  // There were a bunch of frees missing anyway --> replaced with std::vectors
}


int main() {
  std::ofstream csv;
  csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
  csv << "p;N;runtime;residual;iterations" << std::endl;
  csv.close();

  std::vector<int> p_per_dir{ 10, 100, 500, 1000};

  max_iters = 11;
  for (auto& p : p_per_dir)
  {
    std::cout << "--------------------------" << std::endl;
    solve_system(p); // solves a system with p*p unknowns
    if (p == 10)
      max_iters = 151;
    if (p == 100)
      max_iters = 817;
    if (p == 500)
      max_iters = 1001;
  }
  std::cout << "\nData: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;

  return EXIT_SUCCESS;
```

```
}
```

Listing 4: Ex10: Classical CG - HOST version

```

#include <stdio.h>
#include <iostream>
#include <algorithm>
#include <fstream>
#include "poisson2d.hpp"
#include "timer.hpp"

// DEFINES
#define EX "ex10"
#define CSV_NAME "ph_data_host.csv"

/** Computes y = A*x for a sparse matrix A in CSR format and vector x,y  */
void csr_matvec_product(size_t N,
                        int *csr_rowoffsets, int *csr_colindices, double *csr_values,
                        double *x, double *y)
{

  for (size_t row=0; row < N; ++row) {
    double val = 0; // y = Ax for this row
    for (int jj = csr_rowoffsets[row]; jj < csr_rowoffsets[row+1]; ++jj) {
      val += csr_values[jj] * x[csr_colindices[jj]];
    }
    y[row] = val;
  }

}


/** Implementation of the conjugate gradient algorithm.
 *
 *  The control flow is handled by the CPU.
 *  Only the individual operations (vector updates, dot products, sparse matrix-vector product) are transferred to CUDA kernels.
 */
double conjugate_gradient(size_t N,  // number of unknows
                        int *csr_rowoffsets, int *csr_colindices, double *csr_values,
                        double *rhs,
                        double *solution,
                        int& iters)
                        //, double *init_guess)   // feel free to add a nonzero initial guess as needed
{
  Timer timer;
  // clear solution vector (it may contain garbage values):
  std::fill(solution, solution + N, 0);

  // initialize work vectors:
  double *p = (double*)malloc(sizeof(double) * N);
  double *r = (double*)malloc(sizeof(double) * N);
  double *Ap = (double*)malloc(sizeof(double) * N);

  // line 2: initialize r and p:
  std::copy(rhs, rhs+N, p);
  std::copy(rhs, rhs+N, r);

  iters = 0;
  timer.reset();
  while (1) {

    // line 4: A*p:
    csr_matvec_product(N, csr_rowoffsets, csr_colindices, csr_values, p, Ap);

    // similarly for the other operations

    // lines 5,6:
    double res_norm = 0;
    for (size_t i=0; i<N; ++i) res_norm += r[i] * r[i];
```

```
    double alpha = 0;
    for (size_t i=0; i<N; ++i) alpha += p[i] * Ap[i];
    alpha = res_norm / alpha;

    // line 7,8:
    for (size_t i=0; i<N; ++i) {
      solution[i] += alpha *  p[i];
      r[i]        -= alpha * Ap[i];
    }

    double beta = res_norm;

    // lines 9, 10:
    res_norm = 0;
    for (size_t i=0; i<N; ++i) res_norm += r[i] * r[i];
    if (res_norm < 1e-7) break;

    // line 11: compute beta
    beta = res_norm / beta;

    // line 12: update p
    for (size_t i=0; i<N; ++i) p[i] = r[i] + beta * p[i];

    if (iters > 1000) break;  // solver didn't converge
    ++iters;

  }
  double runtime = timer.get();

  std::cout << "Conjugate Gradients converged in " << iters << " iterations." << std::endl;
  return runtime;
}



/** Solve a system with `points_per_direction * points_per_direction` unknowns */
void solve_system(size_t points_per_direction) {

  size_t N = points_per_direction * points_per_direction; // number of unknows to solve for

  //
  // 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) * 5 * N);
  double *csr_values  = (double*)malloc(sizeof(double) * 5 * N);

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

  //
  // Allocate solution vector and right hand side:
  //
  double *solution = (double*)malloc(sizeof(double) * N);
  double *rhs      = (double*)malloc(sizeof(double) * N);
  std::fill(rhs, rhs + N, 1);

  //
  // Call Conjugate Gradient implementation
  //
  int iters = 0;
  double runtime = conjugate_gradient(N, csr_rowoffsets, csr_colindices, csr_values, rhs, solution, iters);

  //
  // Check for convergence:
```

```cpp
  //
  double residual_norm = relative_residual(N, csr_rowoffsets, csr_colindices, csr_values, rhs, solution);
  std::cout << "Relative residual norm: " << residual_norm << " (should be smaller than 1e-6)" << std::endl;

  // not optimal (efficient), but minimally invasive --> easy to copy
  std::ofstream csv;
  csv.open(CSV_NAME, std::fstream::out | std::fstream::app);
  csv << points_per_direction << ";"
    << N << ";"
    << runtime << ";"
    << residual_norm << ";"
    << iters << std::endl;
  csv.close();

  free(solution);
  free(rhs);
  free(csr_rowoffsets);
  free(csr_colindices);
  free(csr_values);
}


int main() {
  std::ofstream csv;
  csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
  csv << "p;N;runtime;residual;iterations" << std::endl;
  csv.close();

  // std::vector<int> p_per_dir{10}; // { 10, 100, 500,1000, 1500};
  std::vector<int> p_per_dir{ 10, 100, 500, 1000}; // ;

  for (auto& p : p_per_dir)
  {
    std::cout << "--------------------------" << std::endl;
    solve_system(p); // solves a system with p*p unknowns
  }
  std::cout << "\nData: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;

  return EXIT_SUCCESS;
}
```