



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 code 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](#)

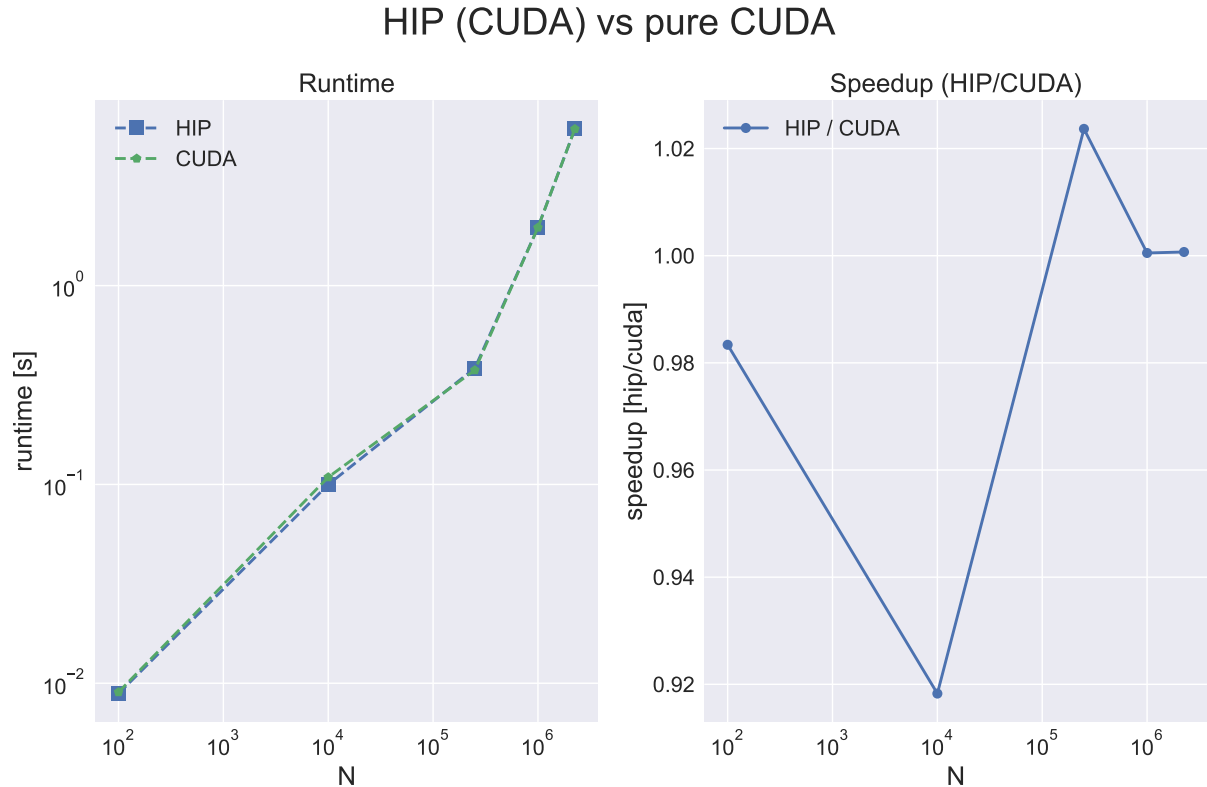


Fig. 2. HIP vs CUDA: efficiency.

HIP (CUDA) vs pure CUDA

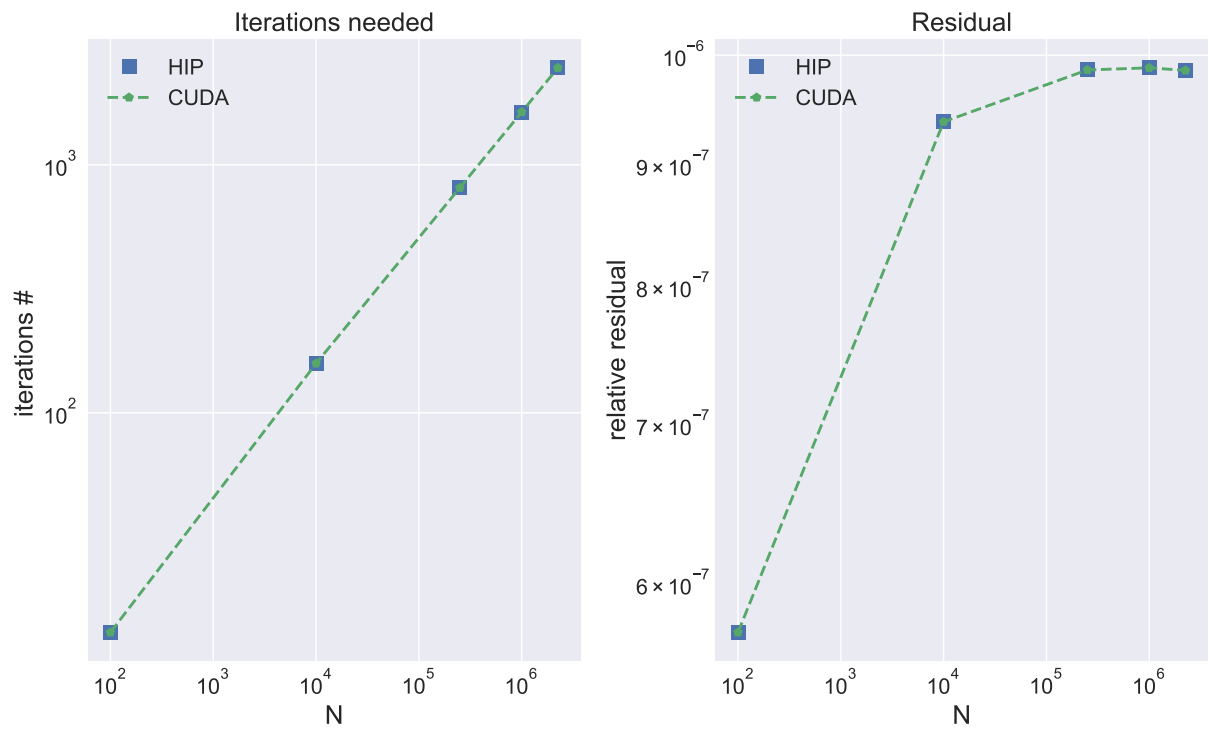


Fig. 3. HIP vs CUDA: precision.

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!

¹I 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.

HIP (CUDA) vs pure CUDA

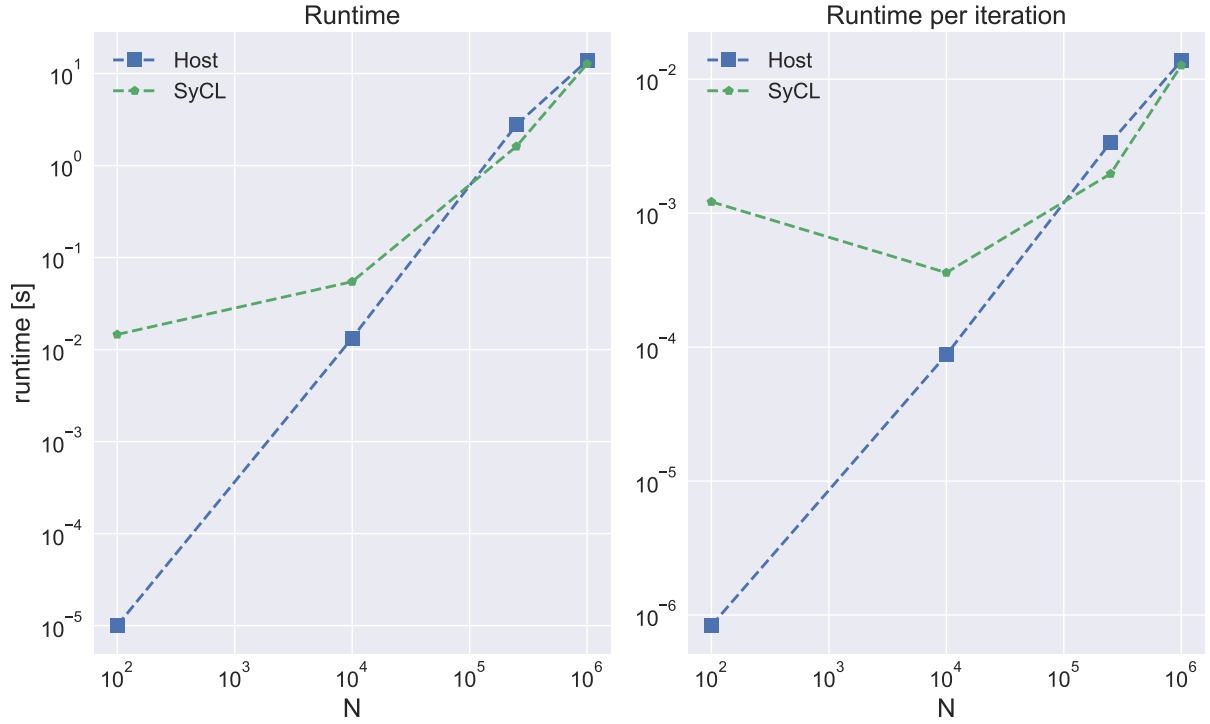


Fig. 4. Host vs Sycl: efficiency.

	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

```

1  #include "poisson2d.hpp"
2  #include "timer.hpp"
3  #include <algorithm>
4  #include <iostream>
5  #include <fstream>
6  #include <vector>
7  // #include <stdio.h>
8  #include "hip/hip_runtime.h"
9
10 // DEFINES
11 #define EX "ex10"
12 #define CSV_NAME "ph_data_hip.csv"
13 #define N_MAX_PRINT 32
14 #define PRINT_ONLY 10
15 #define NUM_TESTS 10 // should be uneven so we dont have to copy after each iteration
16
17 #define GRID_SIZE 512
18 #define BLOCK_SIZE 512
19 #define USE_MY_ATOMIC_ADD
20 #define HIP_ASSERT(x) (assert((x)==hipSuccess)) // only used it once
21
22 /** atomicAdd for doubles for hip for nvcc for many cores exercise 10 for me
23  * by: Peter HOLZNER feat. NVIDIA
24  *
25  * - Ref: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions
26  *
27  * 'Don't let your memes be dreams!'
28  * - Probably Ghandi, idk
29  */
30 __device__ double
31 my_atomic_Add(double* address, double val)
32 {
33     using ulli = unsigned long long int;
34     ulli* address_as_ull =
35         (ulli*)address;
36     ulli old = *address_as_ull, assumed;
37     do {
38         assumed = old;
39         old = atomicCAS(address_as_ull, assumed,
40             __double_as_longlong(val +
41                 __longlong_as_double(assumed)));
42     } while (assumed != old);
43     return __longlong_as_double(old);
44 };
45
46 // y = A * x
47 __global__ void
48 hip_csr_matvec_product(int N,
49     int *csr_rowoffsets, int *csr_colindices,
50     double *csr_values,
51     double *x, double *y)
52 {
53     {
54         for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i += hipBlockDim_x *
55             hipGridDim_x) {
56             double sum = 0;
57             for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++) {
58                 sum += csr_values[k] * x[csr_colindices[k]];
59             }
60             y[i] = sum;
61         }
62     }
63     // x <- x + alpha * y
64     __global__ void
65     hip_vecadd(int N, double *x, double *y, double alpha)
66     {
67         for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i += hipBlockDim_x *
68             hipGridDim_x)
69             x[i] += alpha * y[i];

```



```

69 }
70
71 // x <- y + alpha * x
72 __global__ void
73 hip_vecadd2(int N, double *x, double *y, double alpha)
74 {
75     for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i += hipBlockDim_x *
76         hipGridDim_x)
77         x[i] = y[i] + alpha * x[i];
78 }
79 /**result = (x, y)
80 */
81 __global__ void
82 hip_dot_product(int N, double *x, double *y, double *result)
83 {
84     __shared__ double shared_mem[BLOCK_SIZE];
85
86     double dot = 0;
87     for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i += hipBlockDim_x *
88         hipGridDim_x) {
89         dot += x[i] * y[i];
90     }
91     shared_mem[hipThreadIdx_x] = dot;
92     for (int k = hipBlockDim_x / 2; k > 0; k /= 2) {
93         __syncthreads();
94         if (hipThreadIdx_x < k) {
95             shared_mem[hipThreadIdx_x] += shared_mem[hipThreadIdx_x + k];
96         }
97     }
98     if (hipThreadIdx_x == 0)
99     {
100 #ifdef USE_MY_ATOMIC_ADD
101     my_atomic_Add(result, shared_mem[0]);
102 #else
103     atomicAdd(result, shared_mem[0]);
104 #endif
105     }
106 }
107 }
108
109
110
111 /** Implementation of the conjugate gradient algorithm.
112 *
113 * The control flow is handled by the CPU.
114 * Only the individual operations (vector updates, dot products, sparse
115 * matrix-vector product) are transferred to hip kernels.
116 *
117 * The temporary arrays p, r, and Ap need to be allocated on the GPU for use
118 * with hip. Modify as you see fit.
119 *
120 * Modifications:
121 * - returns runtime as double
122 * - iteration counter (iters) is passed as reference for logging to csv-file
123 * - replaced cuda with hip (literally search-and-replaced the word...)
124 * - implemented the hip-style kernel launches (although unnecessary for this
125 * exercise since we pass it to nvcc anyway :D)
126 */
127 double conjugate_gradient(int N, // number of unknowns
128     int *csr_rowoffsets, int *csr_colindices,
129     double *csr_values, double *rhs, double *solution,
130     int& iters)
131 //, double *init_guess) // feel free to add a nonzero initial guess as needed
132 {
133     // initialize timer
134     Timer timer;
135
136     // clear solution vector (it may contain garbage values):
137     std::fill(solution, solution + N, 0);
138

```

```

139 // initialize work vectors:
140 double alpha, beta;
141 double *hip_solution, *hip_p, *hip_r, *hip_Ap, *hip_scalar;
142 hipMalloc(&hip_p, sizeof(double) * N);
143 hipMalloc(&hip_r, sizeof(double) * N);
144 hipMalloc(&hip_Ap, sizeof(double) * N);
145 hipMalloc(&hip_solution, sizeof(double) * N);
146 hipMalloc(&hip_scalar, sizeof(double));
147
148 hipMemcpy(hip_p, rhs, sizeof(double) * N, hipMemcpyHostToDevice);
149 hipMemcpy(hip_r, rhs, sizeof(double) * N, hipMemcpyHostToDevice);
150 hipMemcpy(hip_solution, solution, sizeof(double) * N, hipMemcpyHostToDevice);
151
152 const double zero = 0;
153 double residual_norm_squared = 0;
154
155 hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice);
156
157 // hip_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, hip_r, hip_r, hip_scalar);
158 hipLaunchKernelGGL(hip_dot_product, // kernel
159     GRID_SIZE, BLOCK_SIZE,          // device params
160     0, 0,                          // shared mem, default stream
161     N, hip_r, hip_r, hip_scalar     // kernel arguments
162 );
163
164 hipMemcpy(&residual_norm_squared, hip_scalar, sizeof(double), hipMemcpyDeviceToHost);
165
166 double initial_residual_squared = residual_norm_squared;
167
168 iters = 0;
169 hipDeviceSynchronize();
170 timer.reset();
171 while (1) {
172
173     // line 4: A*p:
174     // hip_csr_matvec_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, csr_rowoffsets, csr_colindices,
175     //     csr_values, hip_p, hip_Ap);
176     hipLaunchKernelGGL(hip_csr_matvec_product, // kernel
177         GRID_SIZE, BLOCK_SIZE,                // device params
178         0, 0,                                // shared mem, default stream
179         N, csr_rowoffsets, csr_colindices, csr_values, hip_p, hip_Ap // kernel arguments
180     );
181
182     // lines 5,6:
183     hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice);
184     // hip_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, hip_p, hip_Ap, hip_scalar);
185     hipLaunchKernelGGL(hip_dot_product, // kernel
186         GRID_SIZE, BLOCK_SIZE,          // device params
187         0, 0,                          // shared mem, default stream
188         N, hip_p, hip_Ap, hip_scalar    // kernel arguments
189     );
190     hipMemcpy(&alpha, hip_scalar, sizeof(double), hipMemcpyDeviceToHost);
191     alpha = residual_norm_squared / alpha;
192
193     // line 7:
194     // hip_vecadd<<<GRID_SIZE, BLOCK_SIZE>>>(N, hip_solution, hip_p, alpha);
195     hipLaunchKernelGGL(hip_vecadd, // kernel
196         GRID_SIZE, BLOCK_SIZE,     // device params
197         0, 0,                     // shared mem, default stream
198         N, hip_solution, hip_p, alpha // kernel arguments
199     );
200
201     // line 8:
202     // hip_vecadd<<<GRID_SIZE, BLOCK_SIZE>>>(N, hip_r, hip_Ap, -alpha);
203     hipLaunchKernelGGL(hip_vecadd, // kernel
204         GRID_SIZE, BLOCK_SIZE,     // device params
205         0, 0,                     // shared mem, default stream
206         N, hip_r, hip_Ap, -alpha   // kernel arguments
207     );
208
209     // line 9:
210     beta = residual_norm_squared;

```

```

210     HIP_ASSERT(hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice)); // just
        checking if this works properly
211     // hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice);
212     // hip_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, hip_r, hip_r, hip_scalar);
213     hipLaunchKernelGGL(hip_dot_product, // kernel
214         GRID_SIZE, BLOCK_SIZE,          // device params
215         0, 0,                          // shared mem, default stream
216         N, hip_r, hip_r, hip_scalar     // kernel arguments
217     );
218     hipMemcpy(&residual_norm_squared, hip_scalar, sizeof(double), hipMemcpyDeviceToHost);
219
220     // line 10:
221     if (std::sqrt(residual_norm_squared / initial_residual_squared) < 1e-6) {
222         break;
223     }
224
225     // line 11:
226     beta = residual_norm_squared / beta;
227
228     // line 12:
229     // hip_vecadd2<<<GRID_SIZE, BLOCK_SIZE>>>(N, hip_p, hip_r, beta);
230     hipLaunchKernelGGL(hip_vecadd2, // kernel
231         GRID_SIZE, BLOCK_SIZE,          // device params
232         0, 0,                          // shared mem, default stream
233         N, hip_p, hip_r, beta           // kernel arguments
234     );
235
236     if (iters > 10000)
237         break; // solver didn't converge
238     ++iters;
239 }
240 hipMemcpy(solution, hip_solution, sizeof(double) * N, hipMemcpyDeviceToHost);
241
242 hipDeviceSynchronize();
243 double runtime = timer.get();
244 std::cout << "Time elapsed: " << runtime << " (" << runtime / iters << " per iteration)"
    << std::endl;
245
246 if (iters > 10000)
247     std::cout << "Conjugate Gradient did NOT converge within 10000 iterations"
    << std::endl;
248 else
249     std::cout << "Conjugate Gradient converged in " << iters << " iterations."
    << std::endl;
250
251 hipFree(hip_p);
252 hipFree(hip_r);
253 hipFree(hip_Ap);
254 hipFree(hip_solution);
255 hipFree(hip_scalar);
256
257 return runtime;
258 }
259
260 /** Solve a system with 'points_per_direction * points_per_direction' unknowns
261 */
262 void solve_system(int points_per_direction) {
263
264     int N = points_per_direction *
        points_per_direction; // number of unknowns to solve for
265
266     std::cout << "Solving Ax=b with " << N << " unknowns." << std::endl;
267
268     //
269     // Allocate CSR arrays.
270     //
271     // Note: Usually one does not know the number of nonzeros in the system matrix
272     // a-priori.
273     // For this exercise, however, we know that there are at most 5 nonzeros
274     // per row in the system matrix, so we can allocate accordingly.
275     //
276     int *csr_rowoffsets = (int *)malloc(sizeof(double) * (N + 1));

```

```

280 int *csr_colindices = (int *)malloc(sizeof(double) * 5 * N);
281 double *csr_values = (double *)malloc(sizeof(double) * 5 * N);
282
283 int *hip_csr_rowoffsets, *hip_csr_colindices;
284 double *hip_csr_values;
285 //
286 // fill CSR matrix with values
287 //
288 generate_fdm_laplace(points_per_direction, csr_rowoffsets, csr_colindices,
289                     csr_values);
290
291 //
292 // Allocate solution vector and right hand side:
293 //
294 double *solution = (double *)malloc(sizeof(double) * N);
295 double *rhs = (double *)malloc(sizeof(double) * N);
296 std::fill(rhs, rhs + N, 1);
297
298 //
299 // Allocate hip-arrays //
300 //
301 hipMalloc(&hip_csr_rowoffsets, sizeof(double) * (N + 1));
302 hipMalloc(&hip_csr_colindices, sizeof(double) * 5 * N);
303 hipMalloc(&hip_csr_values, sizeof(double) * 5 * N);
304 hipMemcpy(hip_csr_rowoffsets, csr_rowoffsets, sizeof(double) * (N + 1),
305           hipMemcpyHostToDevice);
306 hipMemcpy(hip_csr_colindices, csr_colindices, sizeof(double) * 5 * N,
307           hipMemcpyHostToDevice);
308 hipMemcpy(hip_csr_values, csr_values, sizeof(double) * 5 * N,
309           hipMemcpyHostToDevice);
310
311 //
312 // Call Conjugate Gradient implementation with GPU arrays
313 //
314 int iters = 0; // pass into the CG so we can track it
315 double runtime = conjugate_gradient(N, hip_csr_rowoffsets, hip_csr_colindices,
316                                     hip_csr_values, rhs, solution, iters);
317
318 //
319 // Check for convergence:
320 //
321 double residual_norm = relative_residual(N, csr_rowoffsets, csr_colindices, csr_values,
322                                          rhs, solution);
323 std::cout << "Relative residual norm: " << residual_norm
324           << " (should be smaller than 1e-6)" << std::endl;
325
326 // not optimal (efficient), but minimally invasive --> easy to copy
327 std::ofstream csv;
328 csv.open(CSV_NAME, std::fstream::out | std::fstream::app);
329 csv << points_per_direction << ";";
330     << N << ";";
331     << runtime << ";";
332     << residual_norm << ";";
333     << iters << std::endl;
334 csv.close();
335
336 for (int i = 0; i < N; i++)
337     std::cout << solution[i] << std::endl;
338
339 hipFree(hip_csr_rowoffsets);
340 hipFree(hip_csr_colindices);
341 hipFree(hip_csr_values);
342 free(solution);
343 free(rhs);
344 free(csr_rowoffsets);
345 free(csr_colindices);
346 free(csr_values);
347 }
348
349 int main() {
350     std::ofstream csv;

```

```

347 csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
348 csv << "p;N;runtime;residual;iterations" << std::endl;
349 csv.close();
350
351 hipDeviceProp_t devProp;
352 hipGetDeviceProperties(&devProp, 0);
353 std::cout << " System minor " << devProp.minor << std::endl;
354 std::cout << " System major " << devProp.major << std::endl;
355 std::cout << " agent prop name " << devProp.name << std::endl;
356 std::cout << "hip Device prop succeeded " << std::endl ;
357
358 // std::vector<int> p_per_dir{ 10, 100, 500,1000, 1500};
359
360 std::vector<int> p_per_dir{ 10};
361
362 for (auto& p : p_per_dir)
363 {
364     std::cout << "-----" << std::endl;
365     solve_system(p); // solves a system with p*p unknowns
366 }
367 std::cout << "\nData: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;
368
369 return EXIT_SUCCESS;
370 }

```

Listing 2: Ex10: Classical CG - CUDA version

```

1  #include "poisson2d.hpp"
2  #include "timer.hpp"
3  #include <algorithm>
4  #include <iostream>
5  #include <fstream>
6  #include <vector>
7
8  // DEFINES
9  #define EX "ex10"
10 #define CSV_NAME "ph_data_cuda.csv"
11 #define N_MAX_PRINT 32
12 #define PRINT_ONLY 10
13 #define NUM_TESTS 10 // should be uneven so we dont have to copy after each iteration
14
15 #define GRID_SIZE 512
16 #define BLOCK_SIZE 512
17 #define USE_MY_ATOMIC_ADD
18
19 /** atomicAdd for doubles for hip for nvcc for many cores exercise 10 for me
20 * by: Peter HOLZNER feat. NVIDIA
21 *
22 * - Ref: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions
23 *
24 * 'Don't let your memes be dreams!'
25 * - Probably Ghandi, idk
26 */
27 __device__ double
28 my_atomic_Add(double* address, double val)
29 {
30     using ulli = unsigned long long int;
31     ulli* address_as_ull =
32         (ulli*)address;
33     ulli old = *address_as_ull, assumed;
34     do {
35         assumed = old;
36         old = atomicCAS(address_as_ull, assumed,
37             __double_as_longlong(val +
38                 __longlong_as_double(assumed)));
39     } while (assumed != old);
40     return __longlong_as_double(old);
41 };
42
43
44 /** y = A * x
45 */

```

```

46 __global__ void cuda_csr_matvec_product(int N, int *csr_rowoffsets,
47                                         int *csr_colindices, double *csr_values,
48                                         double *x, double *y)
49 {
50     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x) {
51         double sum = 0;
52         for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++) {
53             sum += csr_values[k] * x[csr_colindices[k]];
54         }
55         y[i] = sum;
56     }
57 }
58
59 /** x <- x + alpha * y
60 */
61 __global__ void cuda_vecadd(int N, double *x, double *y, double alpha)
62 {
63     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
64         x[i] += alpha * y[i];
65 }
66
67 /** x <- y + alpha * x
68 */
69 __global__ void cuda_vecadd2(int N, double *x, double *y, double alpha)
70 {
71     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
72         x[i] = y[i] + alpha * x[i];
73 }
74
75 /** result = (x, y)
76 */
77 __global__ void cuda_dot_product(int N, double *x, double *y, double *result)
78 {
79     __shared__ double shared_mem[BLOCK_SIZE];
80
81     double dot = 0;
82     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x) {
83         dot += x[i] * y[i];
84     }
85
86     shared_mem[threadIdx.x] = dot;
87     for (int k = blockDim.x / 2; k > 0; k /= 2) {
88         __syncthreads();
89         if (threadIdx.x < k) {
90             shared_mem[threadIdx.x] += shared_mem[threadIdx.x + k];
91         }
92     }
93
94     if (threadIdx.x == 0)
95     {
96         #ifdef USE_MY_ATOMIC_ADD
97             my_atomic_Add(result, shared_mem[0]);
98         #else
99             atomicAdd(result, shared_mem[0]);
100         #endif
101     }
102 }
103
104
105
106 /** Implementation of the conjugate gradient algorithm.
107 *
108 * The control flow is handled by the CPU.
109 * Only the individual operations (vector updates, dot products, sparse
110 * matrix-vector product) are transferred to CUDA kernels.
111 *
112 * The temporary arrays p, r, and Ap need to be allocated on the GPU for use
113 * with CUDA. Modify as you see fit.
114 *
115 * Modifications:
116 * returns runtime as double
117 * iteration counter (iters) is passed as reference for logging to csv-file

```

```

118  */
119  double conjugate_gradient(int N, // number of unknowns
120                          int *csr_rowoffsets, int *csr_colindices,
121                          double *csr_values, double *rhs, double *solution,
122                          int& iters)
123  //, double *init_guess) // feel free to add a nonzero initial guess as needed
124  {
125      // initialize timer
126      Timer timer;
127
128      // clear solution vector (it may contain garbage values):
129      std::fill(solution, solution + N, 0);
130
131      // initialize work vectors:
132      double alpha, beta;
133      double *cuda_solution, *cuda_p, *cuda_r, *cuda_Ap, *cuda_scalar;
134      cudaMalloc(&cuda_p, sizeof(double) * N);
135      cudaMalloc(&cuda_r, sizeof(double) * N);
136      cudaMalloc(&cuda_Ap, sizeof(double) * N);
137      cudaMalloc(&cuda_solution, sizeof(double) * N);
138      cudaMalloc(&cuda_scalar, sizeof(double));
139
140      cudaMemcpy(cuda_p, rhs, sizeof(double) * N, cudaMemcpyHostToDevice);
141      cudaMemcpy(cuda_r, rhs, sizeof(double) * N, cudaMemcpyHostToDevice);
142      cudaMemcpy(cuda_solution, solution, sizeof(double) * N, cudaMemcpyHostToDevice);
143
144      const double zero = 0;
145      double residual_norm_squared = 0;
146      cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
147      cuda_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_r, cuda_r, cuda_scalar);
148      cudaMemcpy(&residual_norm_squared, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
149
150      double initial_residual_squared = residual_norm_squared;
151
152      iters = 0; // it's passed in from the outside
153      cudaDeviceSynchronize();
154      timer.reset();
155      while (1) {
156
157          // line 4: A*p:
158          cuda_csr_matvec_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, csr_rowoffsets, csr_colindices,
159                  csr_values, cuda_p, cuda_Ap);
160
161          // lines 5,6:
162          cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
163          cuda_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_p, cuda_Ap, cuda_scalar);
164          cudaMemcpy(&alpha, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
165          alpha = residual_norm_squared / alpha;
166
167          // line 7:
168          cuda_vecadd<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_solution, cuda_p, alpha);
169
170          // line 8:
171          cuda_vecadd<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_r, cuda_Ap, -alpha);
172
173          // line 9:
174          beta = residual_norm_squared;
175          cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
176          cuda_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_r, cuda_r, cuda_scalar);
177          cudaMemcpy(&residual_norm_squared, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
178
179          // line 10:
180          if (std::sqrt(residual_norm_squared / initial_residual_squared) < 1e-6) {
181              break;
182          }
183
184          // line 11:
185          beta = residual_norm_squared / beta;
186
187          // line 12:
188          cuda_vecadd2<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_p, cuda_r, beta);

```

```

189     if (iters > 10000)
190         break; // solver didn't converge
191     ++iters;
192 }
193 cudaMemcpy(solution, cuda_solution, sizeof(double) * N, cudaMemcpyDeviceToHost);
194
195 cudaDeviceSynchronize();
196 double runtime = timer.get();
197 std::cout << "Time elapsed: " << runtime << " (" << runtime / iters << " per iteration)"
198     << std::endl;
199
200 if (iters > 10000)
201     std::cout << "Conjugate Gradient did NOT converge within 10000 iterations"
202     << std::endl;
203 else
204     std::cout << "Conjugate Gradient converged in " << iters << " iterations."
205     << std::endl;
206
207 cudaFree(cuda_p);
208 cudaFree(cuda_r);
209 cudaFree(cuda_Ap);
210 cudaFree(cuda_solution);
211 cudaFree(cuda_scalar);
212
213 return runtime;
214 }
215
216 /** Solve a system with 'points_per_direction * points_per_direction' unknowns
217 */
218 void solve_system(int points_per_direction) {
219     int N = points_per_direction *
220         points_per_direction; // number of unknowns to solve for
221
222     std::cout << "Solving Ax=b with " << N << " unknowns." << std::endl;
223
224     //
225     // Allocate CSR arrays.
226     //
227     // Note: Usually one does not know the number of nonzeros in the system matrix
228     // a-priori.
229     // For this exercise, however, we know that there are at most 5 nonzeros
230     // per row in the system matrix, so we can allocate accordingly.
231     //
232     int *csr_rowoffsets = (int *)malloc(sizeof(double) * (N + 1));
233     int *csr_colindices = (int *)malloc(sizeof(double) * 5 * N);
234     double *csr_values = (double *)malloc(sizeof(double) * 5 * N);
235
236     int *cuda_csr_rowoffsets, *cuda_csr_colindices;
237     double *cuda_csr_values;
238     //
239     // fill CSR matrix with values
240     //
241     generate_fdm_laplace(points_per_direction, csr_rowoffsets, csr_colindices,
242         csr_values);
243
244     //
245     // Allocate solution vector and right hand side:
246     //
247     double *solution = (double *)malloc(sizeof(double) * N);
248     double *rhs = (double *)malloc(sizeof(double) * N);
249     std::fill(rhs, rhs + N, 1);
250
251     //
252     // Allocate CUDA-arrays //
253     //
254     cudaMalloc(&cuda_csr_rowoffsets, sizeof(double) * (N + 1));
255     cudaMalloc(&cuda_csr_colindices, sizeof(double) * 5 * N);
256     cudaMalloc(&cuda_csr_values, sizeof(double) * 5 * N);
257     cudaMemcpy(cuda_csr_rowoffsets, csr_rowoffsets, sizeof(double) * (N + 1),
258         cudaMemcpyHostToDevice);
259     cudaMemcpy(cuda_csr_colindices, csr_colindices, sizeof(double) * 5 * N,

```



```

        cudaMemcpyHostToDevice);
259   cudaMemcpy(cuda_csr_values,      csr_values,      sizeof(double) * 5 * N,
        cudaMemcpyHostToDevice);
260
261   //
262   // Call Conjugate Gradient implementation with GPU arrays
263   //
264   int iters = 0; // pass into the CG so we can track it
265   double runtime = conjugate_gradient(N, cuda_csr_rowoffsets, cuda_csr_colindices,
        cuda_csr_values, rhs, solution, iters);
266
267   //
268   // Check for convergence:
269   //
270   double residual_norm = relative_residual(N, csr_rowoffsets, csr_colindices, csr_values,
        rhs, solution);
271   std::cout << "Relative residual norm: " << residual_norm
272             << " (should be smaller than 1e-6)" << std::endl;
273
274   // not optimal (efficient), but minimally invasive --> easy to copy
275   std::ofstream csv;
276   csv.open(CSV_NAME, std::fstream::out | std::fstream::app);
277   csv << points_per_direction << "; "
278       << N << "; "
279       << runtime << "; "
280       << residual_norm << "; "
281       << iters << std::endl;
282   csv.close();
283
284   cudaFree(cuda_csr_rowoffsets);
285   cudaFree(cuda_csr_colindices);
286   cudaFree(cuda_csr_values);
287   free(solution);
288   free(rhs);
289   free(csr_rowoffsets);
290   free(csr_colindices);
291   free(csr_values);
292 }
293
294 int main() {
295   std::ofstream csv;
296   csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
297   csv << "p;N;runtime;residual;iterations" << std::endl;
298   csv.close();
299
300   std::vector<int> p_per_dir{ 10, 100, 500, 1000, 1500};
301
302   for (auto& p : p_per_dir)
303   {
304     std::cout << "-----" << std::endl;
305     solve_system(p); // solves a system with p*p unknowns
306   }
307   std::cout << "\nData: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;
308
309   return EXIT_SUCCESS;
310 }

```

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

```

1
2 #include <stdio.h>
3 #include <iostream>
4 #include <algorithm>
5 #include <numeric>
6 #include <vector>
7 #include <fstream>
8 #include "poisson2d.hpp"
9 #include "timer.hpp"
10 #include <CL/sycl.hpp>
11
12 // DEFINES
13 #define EX "ex10"

```

```

14 #define CSV_NAME "ph_data_sycl.csv"
15 #define N_MAX_PRINT 32
16 #define PRINT_ONLY 10
17 #define MAX_ITERS 100
18
19 #define GRID_SIZE 4
20 #define BLOCK_SIZE 4
21
22 int max_iters = 11;
23
24 /**by_sycl
25 *
26 * namespace for my sycl functions to distinguish them from normal C++ (host) functions.
27 *
28 * Design is inspired by the HIP/CUDA version.
29 */
30 namespace by_sycl{
31     namespace sycl = cl::sycl; // want to have shorter code.
32
33     const int grid_size = GRID_SIZE;
34     const int block_size = BLOCK_SIZE;
35
36     /** ret = A*x
37     * A... sparse matrix given in CSR format (3 arrays) of dense size NxN
38     * x... vector of size A
39     */
40     void csr_Ax(sycl::queue& q,
41                 size_t N,
42                 sycl::buffer<int>& buff_csr_rowoffsets,
43                 sycl::buffer<int>& buff_csr_colindices,
44                 sycl::buffer<double>& buff_csr_values,
45                 sycl::buffer<double>& buff_x,
46                 sycl::buffer<double>& buff_y)
47     {
48         // std::cout << "Hello from by_sycl::csr_Ax!" << std::endl;
49         // std::vector<double> y(a.size());
50
51         // assert(("x should be length N", x.size() == N));
52         // assert(("csr_rowoffsets should be length N+1", csr_rowoffsets.size() == (N+1)));
53
54
55         // auto global_size = grid_size * block_size;
56         // auto local_size = block_size;
57
58         // sycl::nd_range<1> work_items(global_size, local_size);
59         sycl::range<1> work_items{N};
60         // Submitting a lambda to the queue that carries out the work:
61         q.submit( [&](sycl::handler& cgh)
62                 {
63             auto rowoffsets = buff_csr_rowoffsets.get_access<sycl::access::mode::read>(cgh);
64             auto colidx = buff_csr_colindices.get_access<sycl::access::mode::read>(cgh);
65             auto values = buff_csr_values.get_access<sycl::access::mode::read>(cgh);
66             auto x = buff_x.get_access<sycl::access::mode::read>(cgh);
67             auto y = buff_y.get_access<sycl::access::mode::write>(cgh);
68
69             // The parallel section
70             cgh.parallel_for<class csr_Ax>(work_items,
71                                             [=] (sycl::id<1> row)
72             {
73                 // for (int idx = row;)
74                 double tmp = 0;
75                 for (int jj = rowoffsets[row]; jj < rowoffsets[row+1]; ++jj)
76                     tmp += values[jj] * x[colidx[jj]];
77                 y[row] = tmp;
78             }
79             //
80         );
81     });
82 }
83
84 /** x += alpha * y
85 *

```

```

86  * x...      vector of size N
87  * y...      vector of size N
88  * alpha...  scalar
89  */
90  void inc_ay(sycl::queue& q,
91             const size_t N,
92             sycl::buffer<double>& buff_x,
93             sycl::buffer<double>& buff_y,
94             const double alpha)
95  {
96      // std::cout << "Hello from by_sycl::inc_ay!" << std::endl;
97
98      auto global_size = grid_size * block_size;
99      auto local_size = block_size;
100
101      sycl::nd_range<1> work_items(global_size, local_size);
102      // Submitting a lambda to the queue that carries out the work:
103      q.submit( [&](sycl::handler& cgh)
104      {
105          auto x = buff_x.get_access<sycl::access::mode::read_write>(cgh);
106          auto y = buff_y.get_access<sycl::access::mode::read>(cgh);
107
108          // The parallel section
109          cgh.parallel_for<class inc_ay_work>(work_items,
110                                             [=] (sycl::nd_item<1> item)
111          {
112              size_t tid = item.get_global_linear_id();
113              if (tid < N)
114                  x[tid] = x[tid] + alpha * y[tid];
115          });
116      });
117  }
118
119  /** x = y + alpha * x
120  *
121  * x...      vector of size N
122  * y...      vector of size N
123  * alpha...  scalar
124  */
125  void update(sycl::queue& q,
126             const size_t N,
127             sycl::buffer<double>& buff_x,
128             sycl::buffer<double>& buff_y,
129             const double alpha)
130  {
131      // std::cout << "Hello from by_sycl::update!" << std::endl;
132
133      auto global_size = grid_size * block_size;
134      auto local_size = block_size;
135
136      sycl::nd_range<1> work_items(global_size, local_size);
137      // Submitting a lambda to the queue that carries out the work:
138      q.submit( [&](sycl::handler& cgh)
139      {
140          auto x = buff_x.get_access<sycl::access::mode::read_write>(cgh);
141          auto y = buff_y.get_access<sycl::access::mode::read>(cgh);
142
143          // The parallel section
144          cgh.parallel_for<class update_work>(work_items,
145                                              [=] (sycl::nd_item<1> item)
146          {
147              size_t tid = item.get_global_linear_id();
148              if (tid < N)
149                  x[tid] = y[tid] + alpha * x[tid];
150          });
151      });
152  }
153
154  /** dot = (x, y)
155  *
156  * x...      vector of size N

```

```

158     * y...      vector of size N
159     */
160     double dot(sycl::queue& q,
161               const size_t N,
162               sycl::buffer<double>& buff_x,
163               sycl::buffer<double>& buff_y)
164     {
165         // std::cout << "Hello from by_sycl::dot!" << std::endl;
166         double dot_result = 0;
167
168         auto global_size = grid_size * block_size;
169         auto local_size = block_size;
170
171         sycl::nd_range<1> work_items(global_size, local_size);
172         std::vector<double> result_host(grid_size, 0);
173         { // this scope is for the result buffer to go out of scope "==" write to host
174             sycl::buffer<double> buff_result(result_host.data(), result_host.size());
175             q.submit( [&](sycl::handler& cgh)
176             {
177                 auto x = buff_x.get_access<sycl::access::mode::read_write>(cgh);
178                 auto y = buff_y.get_access<sycl::access::mode::read>(cgh);
179                 auto result = buff_result.get_access<sycl::access::mode::write>(cgh);
180
181                 sycl::accessor<double, 1, sycl::access::mode::read_write, sycl::access::target::
182                     local> local_mem(sycl::range<1>(local_size), cgh);
183
184                 // The parallel section
185                 cgh.parallel_for<class dot_work>(
186                     work_items,
187                     [=] (sycl::nd_item<1> item)
188                     {
189                         size_t local_id = item.get_local_linear_id();
190                         size_t global_id = item.get_global_linear_id();
191
192                         local_mem[local_id] = 0;
193                         for (int idx = global_id; idx < N; idx += global_size)
194                         {
195                             local_mem[local_id] = x[idx] * y[idx];
196                         }
197                         item.barrier(sycl::access::fence_space::local_space);
198                         for (int k = local_size / 2; k > 0; k /= 2)
199                         {
200                             item.barrier(sycl::access::fence_space::local_space);
201                             if (local_id < k)
202                             {
203                                 local_mem[local_id] += local_mem[local_id + k];
204                             }
205                         }
206
207                         if (local_id == 0)
208                         {
209                             result[item.get_group_linear_id()] = local_mem[0];
210                             // item.barrier(sycl::access::fence_space::local_space);
211                             // std::cout << "[WG#: " << item.get_group_linear_id() << "]Number of work
212                                 groups:" << item.get_group_range(0) << std::endl;
213                             // std::cout << "[WG#: " << item.get_group_linear_id() << "local_mem[0] = " <<
214                                 local_mem[0] << std::endl;
215                         }
216                     });
217             });
218         }
219     }
220
221     //
222     //// FUNCTIONS
223     //
224
225     namespace sycl = cl::sycl; // want to have shorter code.
226

```

```

227 /** Implementation of the conjugate gradient algorithm.
228  *
229  * The control flow is handled by the CPU.
230  * Only the individual operations (vector updates, dot products, sparse matrix-vector
    product) are transferred to CUDA kernels.
231  */
232 double conjugate_gradient(size_t N, // number of unknowns
233     std::vector<int>& csr_rowoffsets,
234     std::vector<int>& csr_colindices,
235     std::vector<double>& csr_values,
236     std::vector<double>& rhs,
237     std::vector<double>& solution,
238     int& iters)
239     //, double *init_guess) // feel free to add a nonzero initial
    guess as needed
240 {
241     Timer timer;
242     sycl::device device = sycl::default_selector{}.select_device();
243     sycl::queue queue(device);
244
245     // Check for potential local memory size
246     auto has_local_mem = device.is_host()
247         || (device.get_info<sycl::info::device::local_mem_type>()
248             != sycl::info::local_mem_type::none);
249     auto local_mem_size = device.get_info<sycl::info::device::local_mem_size>();
250     if (!has_local_mem || local_mem_size < (by_sycl::block_size * sizeof(size_t)))
251     {
252         throw "Device doesn't have enough local memory!";
253     }
254
255     // clear solution vector (it may contain garbage values):
256     std::fill(solution.begin(), solution.end(), 1.);
257
258     // initialize work vectors:
259     std::vector<double> Ap(N, 0);
260     // line 2: initialize r and p:
261     std::vector<double> p(rhs);
262     std::vector<double> r(rhs);
263
264     // Initialize buffers for matrix and solution
265     sycl::buffer<int> rowoffsets_buff(csr_rowoffsets.data(), csr_rowoffsets.size());
266     sycl::buffer<int> colidx_buff(csr_colindices.data(), csr_colindices.size());
267     sycl::buffer<double> values_buff(csr_values.data(), csr_values.size());
268
269     sycl::buffer<double> solution_buff(solution.data(), solution.size());
270
271     sycl::buffer<double> r_buff(r.data(), r.size());
272     sycl::buffer<double> Ap_buff(Ap.data(), Ap.size());
273     sycl::buffer<double> p_buff(p.data(), p.size());
274
275     iters = 0;
276
277     double initial_residual_squared = by_sycl::dot(queue, N, r_buff, r_buff);
278     double res_norm = 0;
279     timer.reset();
280     while (1) {
281         // line 4: A*p:
282         // csr_matvec_product(N, csr_rowoffsets, csr_colindices, csr_values, p, Ap);
283         by_sycl::csr_Ax(queue,
284             N, rowoffsets_buff, colidx_buff, values_buff,
285             p_buff, Ap_buff);
286
287         // lines 5,6:
288         // double res_norm = 0;
289         // for (size_t i=0; i<N; ++i) res_norm += r[i] * r[i];
290         res_norm = by_sycl::dot(queue, N, r_buff, r_buff);
291
292         // queue.wait_and_throw(); // necessary for Ap?
293
294         // double alpha = 0;
295         // for (size_t i=0; i<N; ++i) alpha += p[i] * Ap[i];
296         double alpha = by_sycl::dot(queue, N, p_buff, Ap_buff);

```

```

297     alpha = res_norm / alpha;
298     // std::cout << "alpha(" << iters << ") = " << alpha << std::endl;
299
300     // queue.wait_and_throw();
301
302     // line 7,8:
303     // for (size_t i=0; i<N; ++i) {
304     //     solution[i] += alpha * p[i];
305     //     r[i]         -= alpha * Ap[i];
306     // }
307     by_sycl::inc_ay(queue, N, solution_buff, p_buff, alpha);
308     by_sycl::inc_ay(queue, N, r_buff, Ap_buff, -alpha);
309
310     double beta = res_norm;
311
312     // lines 9, 10:
313     // res_norm = 0;
314     // for (size_t i=0; i<N; ++i) res_norm += r[i] * r[i];
315     res_norm = by_sycl::dot(queue, N, r_buff, r_buff);
316     if (std::sqrt( res_norm / initial_residual_squared ) < 1e-7)
317         break;
318
319     // line 11: compute beta
320     beta = res_norm / beta;
321
322     // line 12: update p
323     // for (size_t i=0; i<N; ++i) p[i] = r[i] + beta * p[i];
324     by_sycl::update(queue, N, p_buff, r_buff, beta);
325
326     queue.wait_and_throw();
327     if (iters > max_iters) break; // solver didn't converge
328     ++iters;
329
330 }
331 double runtime = timer.get();
332 std::cout << "Time elapsed: " << runtime << " (" << runtime / iters << " per iteration)"
333     << std::endl;
334 std::cout << "Norm in CG: " << res_norm << std::endl;
335
336 if (iters > MAX_ITERS)
337     std::cout << "Conjugate Gradient did NOT converge within " << MAX_ITERS << " iterations"
338     << std::endl;
339 else
340     std::cout << "Conjugate Gradient converged in " << iters << " iterations."
341     << std::endl;
342 return runtime;
343 }
344
345
346
347 /** Solve a system with 'points_per_direction * points_per_direction' unknowns */
348 void solve_system(size_t points_per_direction) {
349
350     size_t N = points_per_direction * points_per_direction; // number of unknowns to solve for
351
352     //
353     // Allocate CSR arrays.
354     //
355     // Note: Usually one does not know the number of nonzeros in the system matrix a-priori.
356     //       For this exercise, however, we know that there are at most 5 nonzeros per row in
357     //       the system matrix, so we can allocate accordingly.
358     //
359     // int *csr_rowoffsets = (int*)malloc(sizeof(double) * (N+1));
360     // int *csr_colindices = (int*)malloc(sizeof(double) * 5 * N);
361     // double *csr_values = (double*)malloc(sizeof(double) * 5 * N);
362     std::vector<int> csr_rowoffsets(N+1);
363     std::vector<int> csr_colindices(5*N);
364     std::vector<double> csr_values(5*N);
365
366     //
367     // fill CSR matrix with values

```

```

367 //
368 std::cout << "Generating FDM..." << std::endl;
369 generate_fdm_laplace(points_per_direction, csr_rowoffsets.data(), csr_colindices.data(),
    csr_values.data());
370
371 //
372 // Allocate solution vector and right hand side:
373 //
374 // double *solution = (double*)malloc(sizeof(double) * N);
375 // double *rhs      = (double*)malloc(sizeof(double) * N);
376 std::cout << "Initializing vectors..." << std::endl;
377 std::vector<double> solution(N, 0);
378 std::vector<double> rhs(N, 1.);
379 // std::fill(rhs, rhs + N, 1);
380
381 //
382 // Call Conjugate Gradient implementation
383 //
384 int iters = 0;
385 std::cout << "Starting CG" << std::endl;
386 double runtime = conjugate_gradient(N, csr_rowoffsets, csr_colindices, csr_values, rhs,
    solution, iters);
387
388 //
389 // Check for convergence:
390 //
391 std::cout << "Calculating residual" << std::endl;
392 double residual_norm = relative_residual(N, csr_rowoffsets.data(), csr_colindices.data(),
    csr_values.data(), rhs.data(), solution.data());
393 std::cout << "Relative residual norm: " << residual_norm
394     << " (should be smaller than 1e-6)" << std::endl;
395
396 // for (auto & x : solution)
397 //     std::cout << x << std::endl;
398
399 // not optimal (efficient), but minimally invasive --> easy to copy
400 std::ofstream csv;
401 csv.open(CSV_NAME, std::fstream::out | std::fstream::app);
402 csv << points_per_direction << ";"
403     << N << ";"
404     << runtime << ";"
405     << residual_norm << ";"
406     << iters << std::endl;
407 csv.close();
408
409 // There were a bunch of frees missing anyway --> replaced with std::vectors
410 }
411
412
413 int main() {
414     std::ofstream csv;
415     csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
416     csv << "p;N;runtime;residual;iterations" << std::endl;
417     csv.close();
418
419     std::vector<int> p_per_dir{ 10, 100, 500, 1000};
420
421     max_iters = 11;
422     for (auto& p : p_per_dir)
423     {
424         std::cout << "-----" << std::endl;
425         solve_system(p); // solves a system with p*p unknowns
426         if (p == 10)
427             max_iters = 151;
428         if (p == 100)
429             max_iters = 817;
430         if (p == 500)
431             max_iters = 1001;
432     }
433     std::cout << "\nData: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;
434
435     return EXIT_SUCCESS;

```

436 }

Listing 4: Ex10: Classical CG - HOST version

```
1
2 #include <stdio.h>
3 #include <iostream>
4 #include <algorithm>
5 #include <fstream>
6 #include "poisson2d.hpp"
7 #include "timer.hpp"
8
9 // DEFINES
10 #define EX "ex10"
11 #define CSV_NAME "ph_data_host.csv"
12
13 /** Computes  $y = Ax$  for a sparse matrix A in CSR format and vector x,y */
14 void csr_matvec_product(size_t N,
15                         int *csr_rowoffsets, int *csr_colindices, double *csr_values,
16                         double *x, double *y)
17 {
18     for (size_t row=0; row < N; ++row) {
19         double val = 0; //  $y = Ax$  for this row
20         for (int jj = csr_rowoffsets[row]; jj < csr_rowoffsets[row+1]; ++jj) {
21             val += csr_values[jj] * x[csr_colindices[jj]];
22         }
23         y[row] = val;
24     }
25 }
26
27 }
28
29
30 /** Implementation of the conjugate gradient algorithm.
31  *
32  * The control flow is handled by the CPU.
33  * Only the individual operations (vector updates, dot products, sparse matrix-vector
34  * product) are transferred to CUDA kernels.
35  */
36 double conjugate_gradient(size_t N, // number of unknowns
37                           int *csr_rowoffsets, int *csr_colindices, double *csr_values,
38                           double *rhs,
39                           double *solution,
40                           int& iters)
41 //, double *init_guess) // feel free to add a nonzero initial
42 // guess as needed
43 {
44     Timer timer;
45     // clear solution vector (it may contain garbage values):
46     std::fill(solution, solution + N, 0);
47
48     // initialize work vectors:
49     double *p = (double*)malloc(sizeof(double) * N);
50     double *r = (double*)malloc(sizeof(double) * N);
51     double *Ap = (double*)malloc(sizeof(double) * N);
52
53     // line 2: initialize r and p:
54     std::copy(rhs, rhs+N, p);
55     std::copy(rhs, rhs+N, r);
56
57     iters = 0;
58     timer.reset();
59     while (1) {
60         // line 4:  $A \cdot p$ :
61         csr_matvec_product(N, csr_rowoffsets, csr_colindices, csr_values, p, Ap);
62
63         // similarly for the other operations
64
65         // lines 5,6:
66         double res_norm = 0;
67         for (size_t i=0; i<N; ++i) res_norm += r[i] * r[i];
```



```

67     double alpha = 0;
68     for (size_t i=0; i<N; ++i) alpha += p[i] * Ap[i];
69     alpha = res_norm / alpha;
70
71     // line 7,8:
72     for (size_t i=0; i<N; ++i) {
73         solution[i] += alpha * p[i];
74         r[i]        -= alpha * Ap[i];
75     }
76
77     double beta = res_norm;
78
79     // lines 9, 10:
80     res_norm = 0;
81     for (size_t i=0; i<N; ++i) res_norm += r[i] * r[i];
82     if (res_norm < 1e-7) break;
83
84     // line 11: compute beta
85     beta = res_norm / beta;
86
87     // line 12: update p
88     for (size_t i=0; i<N; ++i) p[i] = r[i] + beta * p[i];
89
90     if (iters > 1000) break; // solver didn't converge
91     ++iters;
92
93 }
94 double runtime = timer.get();
95
96 std::cout << "Conjugate Gradients converged in " << iters << " iterations." << std::endl;
97 return runtime;
98 }
99
100
101
102 /** Solve a system with 'points_per_direction * points_per_direction' unknowns */
103 void solve_system(size_t points_per_direction) {
104
105     size_t N = points_per_direction * points_per_direction; // number of unknowns to solve for
106
107     //
108     // Allocate CSR arrays.
109     //
110     // Note: Usually one does not know the number of nonzeros in the system matrix a-priori.
111     //       For this exercise, however, we know that there are at most 5 nonzeros per row in
112     //       the system matrix, so we can allocate accordingly.
113     //
114     int *csr_rowoffsets = (int*)malloc(sizeof(double) * (N+1));
115     int *csr_colindices = (int*)malloc(sizeof(double) * 5 * N);
116     double *csr_values = (double*)malloc(sizeof(double) * 5 * N);
117
118     //
119     // fill CSR matrix with values
120     //
121     generate_fdm_laplace(points_per_direction, csr_rowoffsets, csr_colindices, csr_values);
122
123     //
124     // Allocate solution vector and right hand side:
125     //
126     double *solution = (double*)malloc(sizeof(double) * N);
127     double *rhs       = (double*)malloc(sizeof(double) * N);
128     std::fill(rhs, rhs + N, 1);
129
130     //
131     // Call Conjugate Gradient implementation
132     //
133     int iters = 0;
134     double runtime = conjugate_gradient(N, csr_rowoffsets, csr_colindices, csr_values, rhs,
135                                         solution, iters);
136
137     //
138     // Check for convergence:

```

```

137 //
138 double residual_norm = relative_residual(N, csr_rowoffsets, csr_colindices, csr_values,
    rhs, solution);
139 std::cout << "Relative residual norm: " << residual_norm << " (should be smaller than 1e
    -6)" << std::endl;

140
141 // not optimal (efficient), but minimally invasive --> easy to copy
142 std::ofstream csv;
143 csv.open(CSV_NAME, std::fstream::out | std::fstream::app);
144 csv << points_per_direction << ";";
145     << N << ";";
146     << runtime << ";";
147     << residual_norm << ";";
148     << iters << std::endl;
149 csv.close();
150
151 free(solution);
152 free(rhs);
153 free(csr_rowoffsets);
154 free(csr_colindices);
155 free(csr_values);
156 }
157
158
159 int main() {
160     std::ofstream csv;
161     csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
162     csv << "p;N;runtime;residual;iterations" << std::endl;
163     csv.close();
164
165     // std::vector<int> p_per_dir{10}; // { 10, 100, 500, 1000, 1500};
166     std::vector<int> p_per_dir{ 10, 100, 500, 1000}; // ;
167
168     for (auto& p : p_per_dir)
169     {
170         std::cout << "-----" << std::endl;
171         solve_system(p); // solves a system with p*p unknowns
172     }
173     std::cout << "\nData: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;
174
175     return EXIT_SUCCESS;
176 }

```