INSTITUTE OF MICROELECTRONICS

360.252 Computational Science on Many-Core Architectures

Exercise 5

Author:
Peter Holzner, 01426733

Submission: November 24, 2020

Contents

---

1 Task 1: Inclusive and Exclusive Scans

Code listings for this task:

- main: Listing 3

1.1 (1) Explanation

In short, `scan\_kernel\_1` calculates exclusive sums of separate parts of the vector `X` and stores them in `Y`. The last element of each of these parts are then stored in the `carries` array.

In `scan\_kernel\_2`, the `carries` array is turned into the exclusive sum of the `carries` array.

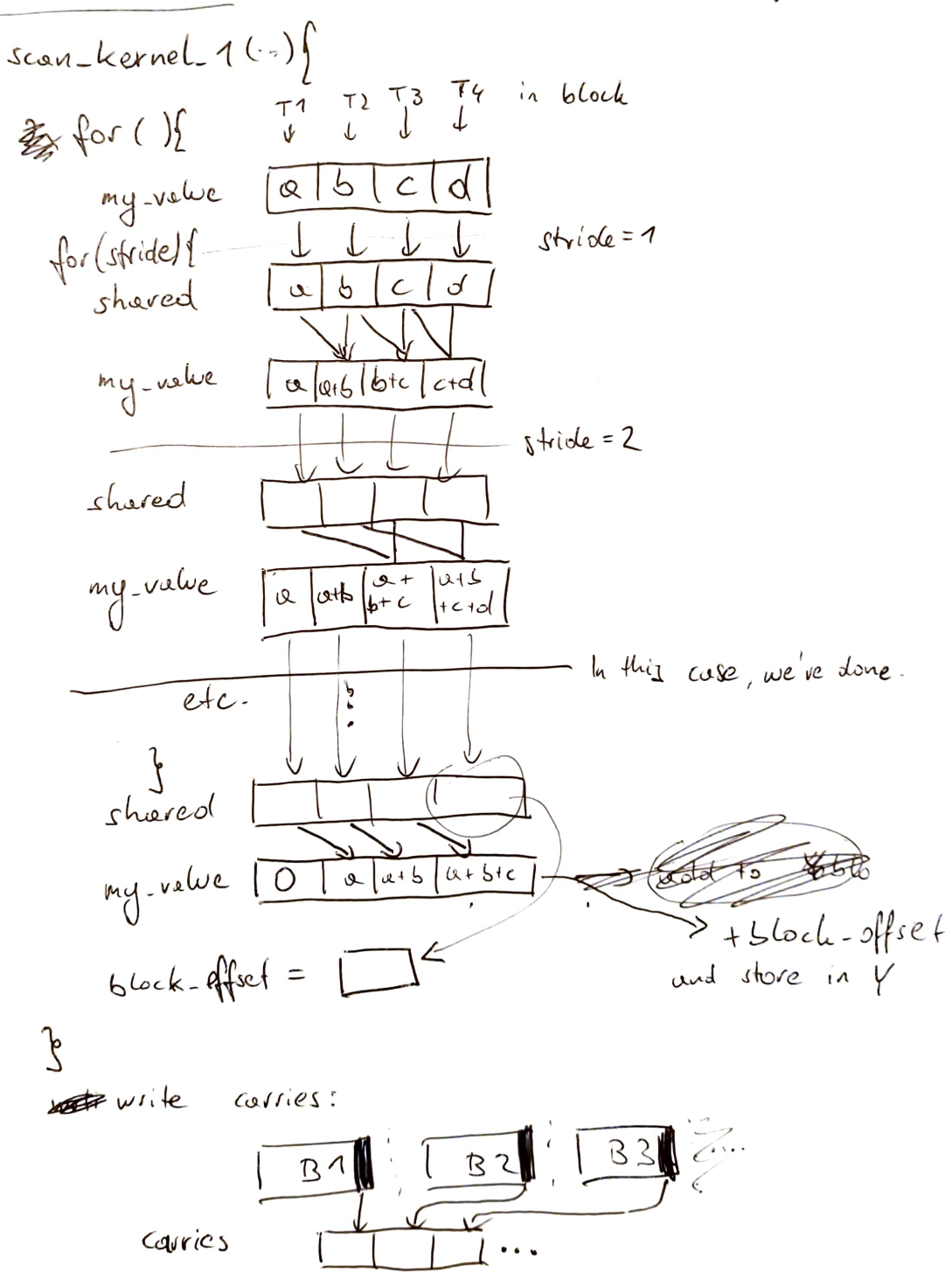
In `scan\_kernel\_3`, the `carries` array and the partial results in `Y` are combined to get the final result (stored in `Y`).

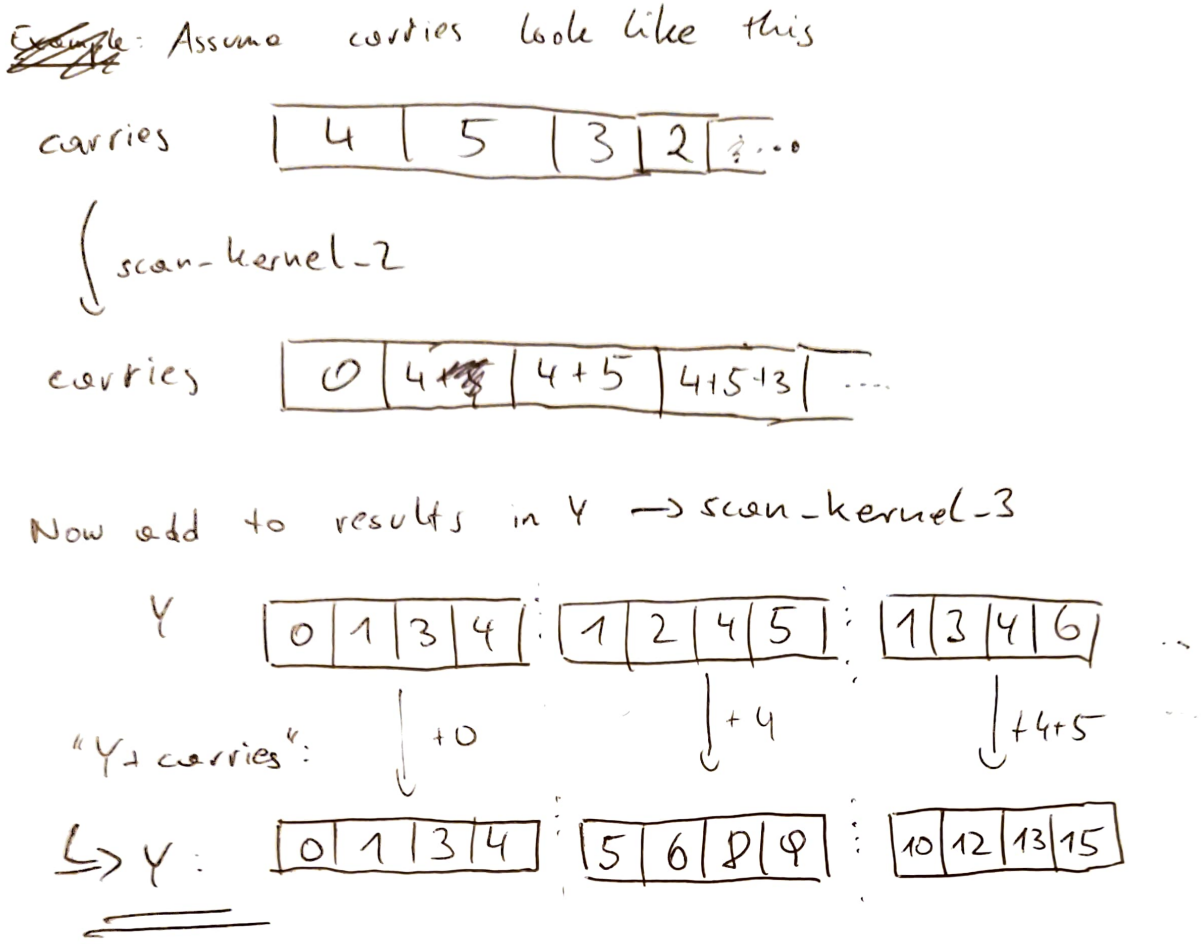
I have tried to explain the code further in the following two handwritten notes:

Example:

3 blocks à 4 threads → B1, B2, B3

Vector X → Result Y (length arbitrary, but assume longer than 12)





1.2 (2) Exclusive -> inclusive scan

There are two different ways to achieve this. One can either shift the result of the exclusive scan to the left and compute the last entry in the scan result by adding to the last element of the exclusive scan.

The other option is the one I chose: simply add the original vector to the exclusive scan vector. There shouldn't really be a much of a difference between the two versions, but I have not tested that.

```
1  __global__ void vectorAddInPlace(double* x, const double* y, const size_t N) {
2    for (size_t tid = threadIdx.x + blockIdx.x * blockDim.x; tid < N; tid += blockDim.x *
         gridDim.x)
3      x[tid] += y[tid];
4  }
5  void inclusive_scan(double const * input,
6                      double * output, int N)
7  {
8    exclusive_scan(input, output, N);
9
10   // To make inclusive
11   vectorAddInPlace<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(output, input, N);
12 }
```

Listing 1: Ex5.1: Addition to exclusive scan –> inclusive scan.

1.3 (3) Proper inclusive scan

Simply replace line 43 in the original implementation in the following way.

```
1  // my_value = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
2  my_value = shared_buffer[threadIdx.x];
```

Listing 2: Ex5.1: Replacement in scan kernels.

The inclusive scan starts with the value of the first entry.

1.4 (4) Performance

In Figure 1, we can see the performance of the GPU and CPU implementation. Overall, the GPU implementations outperform the CPU ones for large N - starting somewhere between $N = 10^5$ and $N = 10^6$ - as expected. In Figure 2, we can see the speedup achieved by the different GPU implementations. 'gpu_in' is the algorithm from the second subtask, 'gpu_in2' the one from the third subtask (proper implementation). The extra kernel call to make the sum inclusive is the reason for the worse performance of 'gpu_in'. 'gpu_in2' and 'gpu_ex' are equal in speedup and runtime as expected.

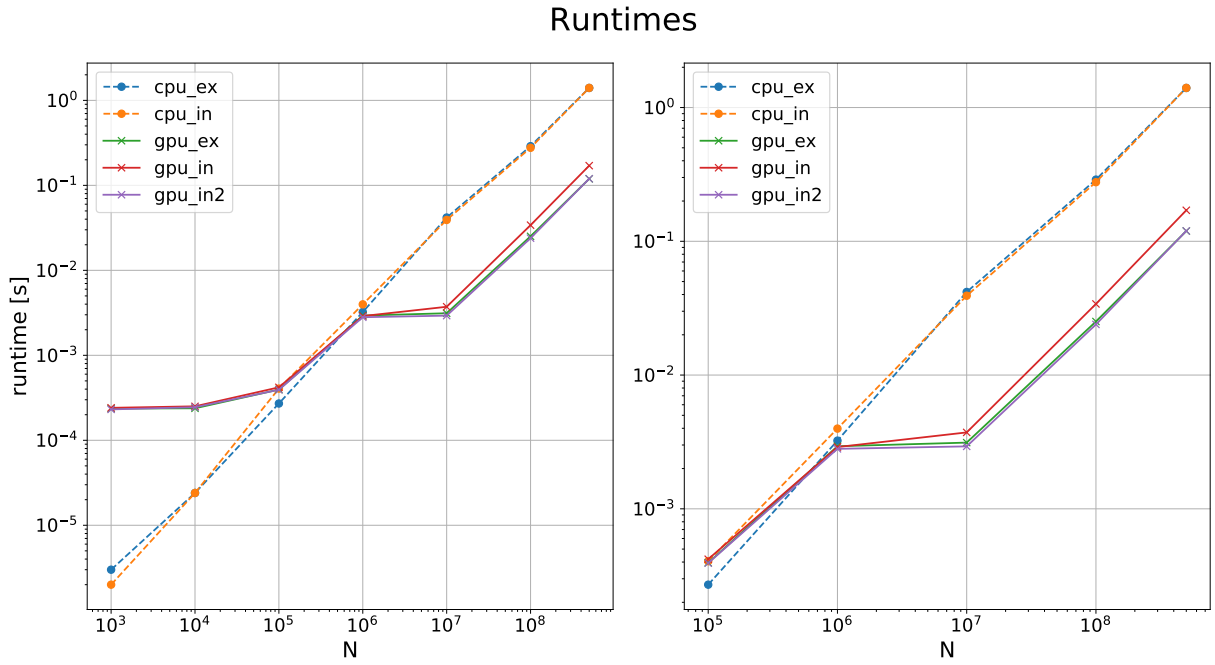


Fig. 1. Runtimes for exclusive and inclusive scan on CPU and GPU. Right side shows runtimes for larger N.

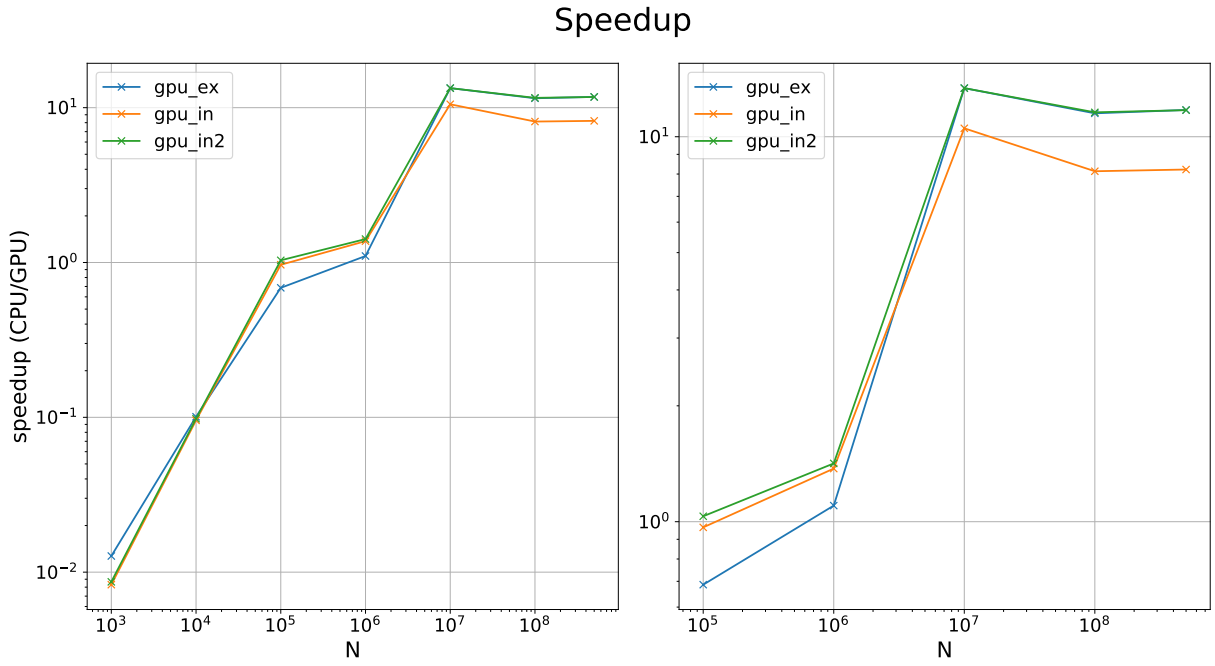


Fig. 2. Speedup for exclusive and inclusive scan on GPU relative to the respective CPU implementation. (Speedup > 1 means faster than respective CPU implementation.)

2 Task 2: FD on GPU

Code listings for this task:

- Finite Difference with CG on GPU: Listing 4

Task	kernel name (Code Line)
Counts and stores the number of nonzero entries for each row	nz_for_this_row (Line 239)
Form the row offset array of the CSR format	in main (Line 474 - 476)
Write the column indices + the nonzero matrix values to the CSR arrays	assembleA (Line 255)

Table 1: Kernels for this task and where to find them in the code.

In Figure 3, we can see the absolute (left) and relative (right) runtimes for assembly on the CPU and GPU as well as the runtime of the CG implementation over different N. One can clearly see that assembling the system matrix on the GPU is much faster (by a factor of $10^2 - 10^4$ depending on N).

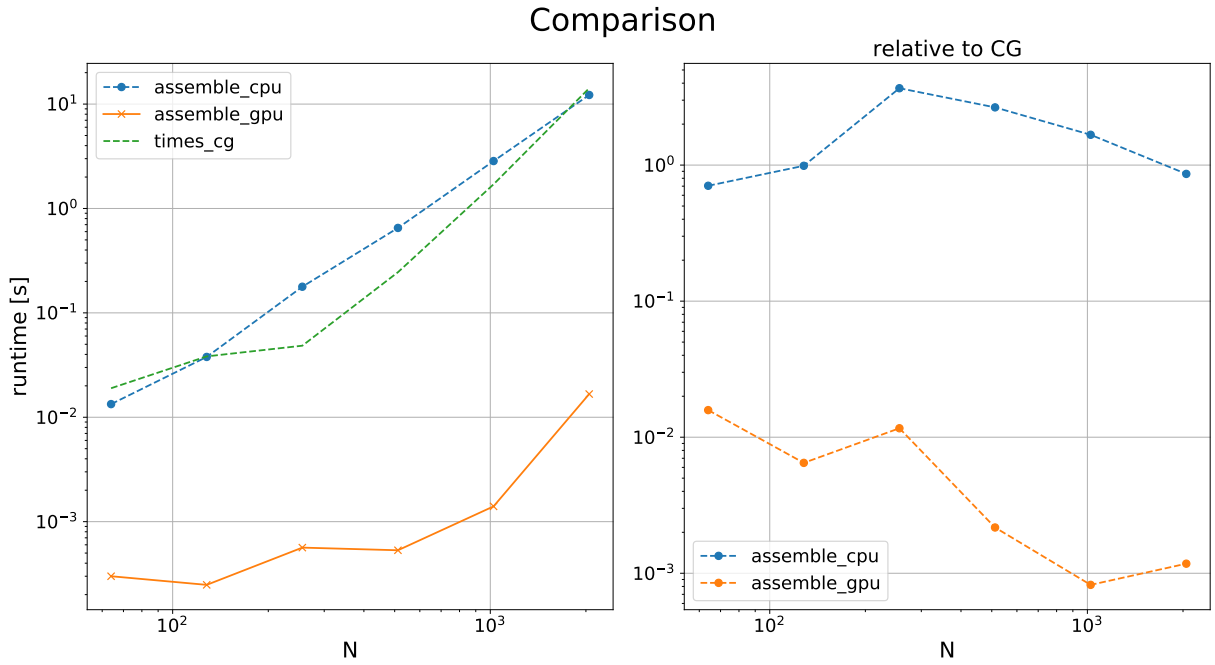


Fig. 3. Absolute (left) and relative (right) runtimes of the two assembly implementations and the CG runtime. The relative runtimes are calculated in relation to the CG runtime. (Grid and Block size = 256)

3 Code and Kernels

Listings

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

# define NUM_BLOCKS 256
# define THREADS_PER_BLOCK 256
# define MAX_POW 9
#define NUM_TESTS 3


__global__ void scan_kernel_1(double const *X,
                              double *Y,
                              int N,
                              double *carries)
{
  __shared__ double shared_buffer[NUM_BLOCKS];
  double my_value;

  unsigned int work_per_thread = (N - 1) / (gridDim.x * blockDim.x) + 1;
  unsigned int block_start = work_per_thread * blockDim.x *  blockIdx.x;
  unsigned int block_stop  = work_per_thread * blockDim.x * (blockIdx.x + 1);
  unsigned int block_offset = 0;

  // run scan on each section
  for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
  {
    // load data:
    my_value = (i < N) ? X[i] : 0;

    // inclusive scan in shared buffer:
    for(unsigned int stride = 1; stride < blockDim.x; stride *= 2)
    {
      __syncthreads();
      shared_buffer[threadIdx.x] = my_value;
      __syncthreads();
      if (threadIdx.x >= stride)
        my_value += shared_buffer[threadIdx.x - stride];
    }
    __syncthreads();
    shared_buffer[threadIdx.x] = my_value;
    __syncthreads();

    // exclusive scan requires us to write a zero value at the beginning of each block
    my_value = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;

    // write to output array
    if (i < N)
      Y[i] = block_offset + my_value;

    block_offset += shared_buffer[blockDim.x-1];
  }

  // write carry:
  if (threadIdx.x == 0)
    carries[blockIdx.x] = block_offset;

}

__global__ void scan_kernel_1_inc(double const *X,
  double *Y,
  int N,
  double *carries)
{
__shared__ double shared_buffer[NUM_BLOCKS];
double my_value;

unsigned int work_per_thread = (N - 1) / (gridDim.x * blockDim.x) + 1;
```

```
73  unsigned int block_start = work_per_thread * blockDim.x *  blockIdx.x;
74  unsigned int block_stop  = work_per_thread * blockDim.x * (blockIdx.x + 1);
75  unsigned int block_offset = 0;
76
77  // run scan on each section
78  for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
79  {
80  // load data:
81  my_value = (i < N) ? X[i] : 0;
82
83  // inclusive scan in shared buffer:
84  for(unsigned int stride = 1; stride < blockDim.x; stride *= 2)
85  {
86  __syncthreads();
87  shared_buffer[threadIdx.x] = my_value;
88  __syncthreads();
89  if (threadIdx.x >= stride)
90  my_value += shared_buffer[threadIdx.x - stride];
91  }
92  __syncthreads();
93  shared_buffer[threadIdx.x] = my_value;
94  __syncthreads();
95
96  // my_value = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
97  my_value = shared_buffer[threadIdx.x];
98
99  // write to output array
100 if (i < N)
101 Y[i] = block_offset + my_value;
102
103 block_offset += shared_buffer[blockDim.x-1];
104 }
105
106 // write carry:
107 if (threadIdx.x == 0)
108 carries[blockIdx.x] = block_offset;
109
110 }
111
112 // exclusive-scan of carries
113 __global__ void scan_kernel_2(double *carries)
114 {
115   __shared__ double shared_buffer[NUM_BLOCKS];
116
117   // load data:
118   double my_carry = carries[threadIdx.x];
119
120   // exclusive scan in shared buffer:
121
122   for(unsigned int stride = 1; stride < blockDim.x; stride *= 2)
123   {
124     __syncthreads();
125     shared_buffer[threadIdx.x] = my_carry;
126     __syncthreads();
127     if (threadIdx.x >= stride)
128       my_carry += shared_buffer[threadIdx.x - stride];
129   }
130   __syncthreads();
131   shared_buffer[threadIdx.x] = my_carry;
132   __syncthreads();
133
134   // // write to output array
135   carries[threadIdx.x] = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
136
137   // // write to output array
138   // carries[threadIdx.x] = shared_buffer[threadIdx.x];
139 }
140
141 __global__ void scan_kernel_3(double *Y, int N,
142                               double const *carries)
143 {
144   unsigned int work_per_thread = (N - 1) / (gridDim.x * blockDim.x) + 1;
```

```
145   unsigned int block_start = work_per_thread * blockDim.x *  blockIdx.x;
146   unsigned int block_stop  = work_per_thread * blockDim.x * (blockIdx.x + 1);
147
148   __shared__ double shared_offset;
149
150   if (threadIdx.x == 0)
151     shared_offset = carries[blockIdx.x];
152
153   __syncthreads();
154
155   // add offset to each element in the block:
156   for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
157     if (i < N)
158       Y[i] += shared_offset;
159 }
160
161 // // Shitfts array to left (to make it inclusive scan)
162 // __global__ void shiftToLeft(double *Y, int N, const double *X)
163 //  {
164 //   tid = blockDim.x * blockIdx.x + threadIdx.x;
165 //   for (int i = tid; i < N-1; i += gridDim.x * blockDim.x) {
166 //     Y[i] = Y[i+1];
167 //   }
168 //   // 0,1 ,2 ,3 ,4 , ...,N-2 ,N-1
169 //   //  /  /  /  /  /          /
170 //   // 0, 1, 2, 3, 4, ..., N-1
171 //   if (tid == 0)
172 //       Y[N-1] += X[N-1];
173 //  }
174 /* Task 1 a) */
175 __global__ void vectorAddInPlace(double* x, const double* y, const size_t N) {
176   for (size_t tid = threadIdx.x + blockIdx.x * blockDim.x; tid < N; tid += blockDim.x *
        gridDim.x)
177     x[tid] += y[tid];
178 }
179
180
181 void exclusive_scan(double const * input,
182                     double * output, int N)
183 {
184   double *carries;
185   cudaMalloc(&carries, sizeof(double) * NUM_BLOCKS);
186
187   // First step: Scan within each thread group and write carries
188   scan_kernel_1<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(input, output, N, carries);
189
190   // Second step: Compute offset for each thread group (exclusive scan for each thread group
        )
191   scan_kernel_2<<<1, NUM_BLOCKS>>>(carries);
192
193   // Third step: Offset each thread group accordingly
194   scan_kernel_3<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(output, N, carries);
195
196   cudaFree(carries);
197 }
198
199 void inclusive_scan(double const * input,
200                     double * output, int N)
201 {
202   exclusive_scan(input, output, N);
203
204   // To make inclusive
205   vectorAddInPlace<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(output, input, N);
206 }
207
208 void inclusive_scan2(double const * input,
209                     double * output, int N)
210 {
211   double *carries;
212   cudaMalloc(&carries, sizeof(double) * NUM_BLOCKS);
213
214   // First step: Scan within each thread group and write carries
```

```
215     scan_kernel_1_inc<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(input, output, N, carries);
216
217     // Second step: Compute offset for each thread group (exclusive scan for each thread group
            )
218     scan_kernel_2<<<1, NUM_BLOCKS>>>(carries);
219
220     // Third step: Offset each thread group accordingly
221     scan_kernel_3<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(output, N, carries);
222
223     cudaFree(carries);
224   }
225
226   double median(std::vector<double>& vec)
227   {
228     // modified taken from here: https://stackoverflow.com/questions/2114797/compute-median-of
            -values-stored-in-vector-c
229
230     size_t size = vec.size();
231
232     if (size == 0)
233             return 0.;
234
235     sort(vec.begin(), vec.end());
236
237     size_t mid = size/2;
238
239     return size % 2 == 0 ? (vec[mid] + vec[mid-1]) / 2 : vec[mid];
240   }
241
242   int main() {
243     size_t max_N = (size_t)std::pow(10,MAX_POW);
244     std::vector<size_t> vec_Ns;
245     std::cout << "Check if correct Ns are calculated: (10^3- 10^" << MAX_POW << ")\n";
246     int cnt = 2;
247     for (size_t x = 1000; x <= max_N; x*=10) {
248       if (x==max_N) x/=2;
249       vec_Ns.push_back(x);
250       std::cout << cnt++ << " : " << x << "\n";
251     }
252     std::cout << std::endl;
253
254     Timer timer;
255     // Container for runtimes
256     std::vector<double> times_cpu_ex;
257     std::vector<double> times_cpu_in;
258
259     std::vector<double> times_gpu_ex;
260     std::vector<double> times_gpu_in;
261     std::vector<double> times_gpu_in2;
262     std::vector<double> tmp(NUM_TESTS);
263
264     for (size_t& N: vec_Ns) {
265       //
266       // Allocate host arrays for reference
267       //
268       double *x = (double *)malloc(sizeof(double) * N);
269       double *y = (double *)malloc(sizeof(double) * N);
270       double *z = (double *)malloc(sizeof(double) * N);
271       double *z2 = (double *)malloc(sizeof(double) * N);
272       std::fill(x, x + N, 1);
273
274       // reference calculation EXCLUSIVE:
275       for (int tests = 0; tests < NUM_TESTS; ++tests){
276         timer.reset();
277         y[0] = 0;
278         for (std::size_t i=1; i<N; ++i) y[i] = y[i-1] + x[i-1];
279         tmp[tests]= timer.get();
280       }
281       times_cpu_ex.push_back(median(tmp));
282
283       //
284       // Allocate CUDA-arrays
```

```
285       //
286       double *cuda_x, *cuda_y;
287       cudaMalloc(&cuda_x, sizeof(double) * N);
288       cudaMalloc(&cuda_y, sizeof(double) * N);
289       cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice);
290
291
292       // Perform the exclusive scan and obtain results
293       for (int tests = 0; tests < NUM_TESTS; ++tests){
294       timer.reset();
295       exclusive_scan(cuda_x, cuda_y, N);
296         tmp[tests]= timer.get();
297       }
298       times_gpu_ex.push_back(median(tmp));
299       cudaMemcpy(z, cuda_y, sizeof(double) * N, cudaMemcpyDeviceToHost);
300
301       //
302       // Print first few entries for reference
303       //
304       if (N ==100) {
305         std::cout << "Exclusive scan:\n";
306         std::cout << "CPU y: ";
307         for (int i=0; i<10; ++i) std::cout << y[i] << " ";
308         std::cout << " ... ";
309         for (int i=N-10; i<N; ++i) std::cout << y[i] << " ";
310         std::cout << std::endl;
311
312         std::cout << "GPU y: ";
313         for (int i=0; i<10; ++i) std::cout << z[i] << " ";
314         std::cout << " ... ";
315         for (int i=N-10; i<N; ++i) std::cout << z[i] << " ";
316         std::cout << std::endl;
317       }
318
319       //--------- INCLUSIVE SCAN ---------- //
320       // reference calculation INCLUSIVE:
321       for (int tests = 0; tests < NUM_TESTS; ++tests){
322         timer.reset();
323         y[0] = x[0];
324         for (std::size_t i=1; i<N; ++i) y[i] = y[i-1] + x[i];
325         tmp[tests]= timer.get();
326       }
327       times_cpu_in.push_back(median(tmp));
328
329       // Perform the inclusive scans and obtain results
330       for (int tests = 0; tests < NUM_TESTS; ++tests){
331         timer.reset();
332         inclusive_scan(cuda_x, cuda_y, N);
333         tmp[tests]= timer.get();
334       }
335       times_gpu_in.push_back(median(tmp));
336       cudaMemcpy(z, cuda_y, sizeof(double) * N, cudaMemcpyDeviceToHost);
337
338       for (int tests = 0; tests < NUM_TESTS; ++tests){
339         timer.reset();
340         inclusive_scan2(cuda_x, cuda_y, N);
341         tmp[tests]= timer.get();
342       }
343       times_gpu_in2.push_back(median(tmp));
344       cudaMemcpy(z2, cuda_y, sizeof(double) * N, cudaMemcpyDeviceToHost);
345
346       if (N == 100) {
347         std::cout << "Inclusive scan:\n";
348         std::cout << "CPU y_: ";
349         for (int i=0; i<10; ++i) std::cout << y[i] << " ";
350         std::cout << " ... ";
351         for (int i=N-10; i<N; ++i) std::cout << y[i] << " ";
352         std::cout << std::endl;
353
354         std::cout << "GPU y1: ";
355         for (int i=0; i<10; ++i) std::cout << z[i] << " ";
356         std::cout << " ... ";
```

```
357         for (int i=N-10; i<N; ++i) std::cout << z[i] << " ";
358         std::cout << std::endl;
359         std::cout << "GPU y2: ";
360         for (int i=0; i<10; ++i) std::cout << z2[i] << " ";
361         std::cout << " ... ";
362         for (int i=N-10; i<N; ++i) std::cout << z2[i] << " ";
363         std::cout << std::endl;
364       }
365
366
367       //
368       // Clean up:
369       //
370       free(x);
371       free(y);
372       free(z);
373       cudaFree(cuda_x);
374       cudaFree(cuda_y);
375     }
376     std::cout << "\n\nResults in csv form can be found here\nhttps://gtx1080.360252.org/2020/
          ex5/in_ex_data_ph.csv" << std::endl;
377
378     // OUT PUT TO CSV FILE
379     std::string header = "N;times_cpu_ex;times_gpu_ex;times_cpu_in;times_gpu_in;times_gpu_in2"
          ;
380     std::string sep = ";";
381
382     std::fstream csv;
383     csv.open ("in_ex_data_ph.csv", std::fstream::out | std::fstream::trunc);
384     csv << header << std::endl;
385     for (int i = 0; i < vec_Ns.size(); ++i ) {
386       csv << std::scientific << vec_Ns[i] << sep
387         << times_cpu_ex[i] << sep
388         << times_gpu_ex[i] << sep
389         << times_cpu_in[i] << sep
390         << times_gpu_in[i] << sep
391         << times_gpu_in2[i] << "\n";
392     }
393     csv << std::endl;
394     csv.close();
395
396     return EXIT_SUCCESS;
397 }
```

Listing 3: Ex5.1: Scans

```
1  #include "poisson2d.hpp"
2  #include "timer.hpp"
3  #include <algorithm>
4  #include <cmath>
5  #include <vector>
6  #include <iostream>
7  #include <fstream>
8  #include <stdio.h>
9
10  // Block and grid size defines.
11 // Seperate defines are really just for future convenience...
12 #define CSV_NAME "fd_data_ph.csv"
13 #define BLOCK_SIZE 256
14 #define GRID_SIZE 256
15 #define SEP ";"
16 // #define DEBUG
17
18 template <typename T>
19 void printContainer(T container, const int size, const int only) {
20   if (only){
21     for (int i = 0; i < only; ++i)
22         std::cout << container[i] << " | ";
23     std::cout << " ... ";
24     for (int i = size - only; i < size; ++i)
25       std::cout << container[i] << " | ";
26   }
```

```
27    else {
28      for (int i = 0; i < size; ++i)
29          std::cout << container[i] << " | ";
30    }
31    std::cout << std::endl;
32   }
33
34  // y = A * x
35  __global__ void cuda_csr_matvec_product(int N, int *csr_rowoffsets,
36      int *csr_colindices, double *csr_values,
37      double *x, double *y)
38    {
39    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x) {
40    double sum = 0;
41    for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++) {
42    sum += csr_values[k] * x[csr_colindices[k]];
43    }
44    y[i] = sum;
45    }
46    }
47
48    // x <- x + alpha * y
49    __global__ void cuda_vecadd(int N, double *x, double *y, double alpha)
50    {
51    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
52    x[i] += alpha * y[i];
53    }
54
55    // x <- y + alpha * x
56    __global__ void cuda_vecadd2(int N, double *x, double *y, double alpha)
57    {
58    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
59    x[i] = y[i] + alpha * x[i];
60    }
61
62    // result = (x, y)
63    __global__ void cuda_dot_product(int N, double *x, double *y, double *result)
64    {
65    __shared__ double shared_mem[BLOCK_SIZE];
66
67    double dot = 0;
68    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x) {
69    dot += x[i] * y[i];
70    }
71
72    shared_mem[threadIdx.x] = dot;
73    for (int k = blockDim.x / 2; k > 0; k /= 2) {
74    __syncthreads();
75    if (threadIdx.x < k) {
76    shared_mem[threadIdx.x] += shared_mem[threadIdx.x + k];
77    }
78    }
79
80    if (threadIdx.x == 0) atomicAdd(result, shared_mem[0]);
81    }
82
83    __global__ void part1(int N,
84      double* x, double* r, double *p, double *Ap,
85      double alpha, double beta)
86    {
87      // lines 2 , 3 + 4
88      for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
            {
89        x[i] = x[i] + alpha * p[i];
90        double r_tmp = r[i] - alpha * Ap[i];
91        r[i] = r_tmp;
92      //}
93      // Merge these two?
94      //for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x
            ) {
95        p[i] = r_tmp + beta * p[i];
96      }
```

```
97    }
98
99    __global__ void part2(int N,
100     int *csr_rowoffsets, int *csr_colindices, double *csr_values,
101     double* r, double *p, double *Ap,
102     double* ApAp, double* pAp, double* rr
103     )
104   {
105     __shared__ double shared_mem_ApAp[BLOCK_SIZE];
106     __shared__ double shared_mem_pAp[BLOCK_SIZE];
107     __shared__ double shared_mem_rr[BLOCK_SIZE];
108     // Mat-vec product
109     double dot_ApAp = 0., dot_pAp = 0., dot_rr = 0.;
110     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
          {
111       double sum = 0;
112       for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++) {
113         sum += csr_values[k] * p[csr_colindices[k]];
114       }
115       Ap[i] = sum;
116       dot_ApAp += sum*sum;
117       dot_pAp += p[i]*sum;
118       dot_rr += r[i]*r[i];
119     }
120     // now :
121     // Ap = Ap_i --> Line 5
122     // we are ready for reductions
123
124     shared_mem_ApAp[threadIdx.x] = dot_ApAp;
125     shared_mem_pAp[threadIdx.x] = dot_pAp;
126     shared_mem_rr[threadIdx.x]  = dot_rr;
127     for (int k = blockDim.x / 2; k > 0; k /= 2) {
128       __syncthreads();
129       if (threadIdx.x < k) {
130         shared_mem_ApAp[threadIdx.x] += shared_mem_ApAp[threadIdx.x + k];
131         shared_mem_pAp[threadIdx.x] += shared_mem_pAp[threadIdx.x + k];
132         shared_mem_rr[threadIdx.x] += shared_mem_rr[threadIdx.x + k];
133       }
134     }
135
136     if (threadIdx.x == 0) {
137       atomicAdd(ApAp, shared_mem_ApAp[0]);
138       atomicAdd(pAp, shared_mem_pAp[0]);
139       atomicAdd(rr, shared_mem_rr[0]);
140     }
141     // now:
142     // ApAp, pAp, rr --> Line 6
143   }
144
145 __global__ void scan_kernel_1(int const *X,
146                               int *Y,
147                               int N,
148                               int *carries)
149 {
150   __shared__ int shared_buffer[BLOCK_SIZE];
151   int my_value;
152
153   unsigned int work_per_thread = (N - 1) / (gridDim.x * blockDim.x) + 1;
154   unsigned int block_start = work_per_thread * blockDim.x *  blockIdx.x;
155   unsigned int block_stop  = work_per_thread * blockDim.x * (blockIdx.x + 1);
156   unsigned int block_offset = 0;
157
158   // run scan on each section
159   for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
160   {
161     // load data:
162     my_value = (i < N) ? X[i] : 0;
163
164     // inclusive scan in shared buffer:
165     for(unsigned int stride = 1; stride < blockDim.x; stride *= 2)
166     {
167       __syncthreads();
```

```
168         shared_buffer[threadIdx.x] = my_value;
169         __syncthreads();
170         if (threadIdx.x >= stride)
171           my_value += shared_buffer[threadIdx.x - stride];
172       }
173       __syncthreads();
174       shared_buffer[threadIdx.x] = my_value;
175       __syncthreads();
176
177       // exclusive scan requires us to write a zero value at the beginning of each block
178       my_value = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
179
180       // write to output array
181       if (i < N)
182         Y[i] = block_offset + my_value;
183
184       block_offset += shared_buffer[blockDim.x-1];
185     }
186
187     // write carry:
188     if (threadIdx.x == 0)
189       carries[blockIdx.x] = block_offset;
190
191   }
192
193   // exclusive-scan of carries
194   __global__ void scan_kernel_2(int *carries)
195   {
196     __shared__ int shared_buffer[BLOCK_SIZE];
197
198     // load data:
199     int my_carry = carries[threadIdx.x];
200
201     // exclusive scan in shared buffer:
202
203     for(unsigned int stride = 1; stride < blockDim.x; stride *= 2)
204     {
205       __syncthreads();
206       shared_buffer[threadIdx.x] = my_carry;
207       __syncthreads();
208       if (threadIdx.x >= stride)
209         my_carry += shared_buffer[threadIdx.x - stride];
210     }
211     __syncthreads();
212     shared_buffer[threadIdx.x] = my_carry;
213     __syncthreads();
214
215     // write to output array
216     carries[threadIdx.x] = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
217   }
218
219   __global__ void scan_kernel_3(int *Y, int N,
220                                 int const *carries)
221   {
222     unsigned int work_per_thread = (N - 1) / (gridDim.x * blockDim.x) + 1;
223     unsigned int block_start = work_per_thread * blockDim.x *  blockIdx.x;
224     unsigned int block_stop  = work_per_thread * blockDim.x * (blockIdx.x + 1);
225
226     __shared__ int shared_offset;
227
228     if (threadIdx.x == 0)
229       shared_offset = carries[blockIdx.x];
230
231     __syncthreads();
232
233     // add offset to each element in the block:
234     for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
235       if (i < N)
236         Y[i] += shared_offset;
237   }
238
239   __global__ void count_nnz(int* row_offsets, int N, int M) {
```

```
240     for(int row = blockDim.x * blockIdx.x + threadIdx.x; row < N * M; row += gridDim.x *
            blockDim.x) {
241         int nnz_for_this_node = 1;
242         int i = row / N;
243         int j = row % N;
244 
245         if(i > 0) nnz_for_this_node += 1;
246         if(j > 0) nnz_for_this_node += 1;
247         if(i < N-1) nnz_for_this_node += 1;
248         if(j < M-1) nnz_for_this_node += 1;
249 
250         row_offsets[row] = nnz_for_this_node;
251     }
252 }
253 
254 
255 __global__ void assembleA(double* values, int* columns, int* row_offsets, int N, int M) {
256     for(int row = blockDim.x * blockIdx.x + threadIdx.x; row < N*M; row += gridDim.x *
            blockDim.x) {
257         int i = row / N;
258         int j = row % N;
259         int counter = 0;
260 
261         if ( i > 0) {
262             values[(int)row_offsets[row] + counter] = -1;
263             columns[(int)row_offsets[row] + counter] = (i-1)*N+j;
264             counter++;
265         }
266 
267         if ( j > 0) {
268             values[(int)row_offsets[row] + counter] = -1;
269             columns[(int)row_offsets[row] + counter] = i*N+(j-1);
270             counter++;
271         }
272 
273         values[(int)row_offsets[row] + counter] = 4;
274         columns[(int)row_offsets[row] + counter] = i*N+j;
275 
276         counter++;
277 
278         if ( j < M-1) {
279             values[(int)row_offsets[row] + counter] = -1;
280             columns[(int)row_offsets[row] + counter] = i*N+(j+1);
281             counter++;
282         }
283         if ( i < N-1) {
284             values[(int)row_offsets[row] + counter] = -1;
285             columns[(int)row_offsets[row] + counter] = (i+1)*N+j;
286             counter++;
287         }
288     }
289 }
290 
291 
292 void exclusive_scan(int const * input,
293                     int * output, int N)
294 {
295 int *carries;
296 cudaMalloc(&carries, sizeof(int) * GRID_SIZE);
297 
298 // First step: Scan within each thread group and write carries
299 scan_kernel_1<<<GRID_SIZE, BLOCK_SIZE>>>(input, output, N, carries);
300 
301 // Second step: Compute offset for each thread group (exclusive scan for each thread group)
302 scan_kernel_2<<<1, GRID_SIZE>>>(carries);
303 
304 // Third step: Offset each thread group accordingly
305 scan_kernel_3<<<GRID_SIZE, BLOCK_SIZE>>>(output, N, carries);
306 
307 cudaFree(carries);
308 }
309 
```

```
310 int conjugate_gradient(int N, // number of unknows
311     int *csr_rowoffsets, int *csr_colindices,
312     double *csr_values, double *rhs, double *solution)
313 //, double *init_guess)   // feel free to add a nonzero initial guess as needed
314 {
315   // initialize timer
316   Timer timer;
317
318   // clear solution vector (it may contain garbage values):
319   std::fill(solution, solution + N, 0);
320
321   // initialize work vectors:
322   double alpha, beta, pAp, ApAp, rr;
323   double* cuda_pAp, *cuda_ApAp, *cuda_rr;
324   double* cuda_x, *cuda_p, *cuda_r, *cuda_Ap;
325   cudaMalloc(&cuda_p, sizeof(double) * N);
326   cudaMalloc(&cuda_r, sizeof(double) * N);
327   cudaMalloc(&cuda_Ap, sizeof(double) * N);
328   cudaMalloc(&cuda_x, sizeof(double) * N);
329   cudaMalloc(&cuda_pAp, sizeof(double));
330   cudaMalloc(&cuda_ApAp, sizeof(double));
331   cudaMalloc(&cuda_rr, sizeof(double));
332
333   cudaMemcpy(cuda_p, rhs, sizeof(double) * N, cudaMemcpyHostToDevice);
334   cudaMemcpy(cuda_r, rhs, sizeof(double) * N, cudaMemcpyHostToDevice);
335   cudaMemcpy(cuda_x, solution, sizeof(double) * N, cudaMemcpyHostToDevice);
336
337   const double zero = 0;
338   cudaMemcpy(cuda_pAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
339   cudaMemcpy(cuda_ApAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
340   cudaMemcpy(cuda_rr, &zero, sizeof(double), cudaMemcpyHostToDevice);
341
342   // Initial values: i = 0
343   // device
344   cuda_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_r, cuda_r, cuda_rr);
345   cuda_csr_matvec_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, csr_rowoffsets, csr_colindices,
          csr_values, cuda_p, cuda_Ap);
346   cuda_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_p, cuda_Ap, cuda_pAp);
347   cuda_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_Ap, cuda_Ap, cuda_ApAp);
348   cudaMemcpy(&rr, cuda_rr, sizeof(double), cudaMemcpyDeviceToHost);
349   cudaMemcpy(&pAp, cuda_pAp, sizeof(double), cudaMemcpyDeviceToHost);
350   cudaMemcpy(&ApAp, cuda_ApAp, sizeof(double), cudaMemcpyDeviceToHost);
351   cudaDeviceSynchronize();
352
353   // host side of things
354   double initial_residual_squared = rr;
355
356   #ifdef DEBUG
357   std::cout << "Initial residual norm: " << initial_residual_squared << std::endl;
358   #endif
359   alpha = rr / pAp;
360   //beta = (alpha*alpha * ApAp - rr) / rr;
361   beta = alpha * alpha * ApAp / rr - 1;
362
363   int iters = 1;
364   cudaDeviceSynchronize();
365   timer.reset();
366   while (1) {
367   part1<<<BLOCK_SIZE, GRID_SIZE>>>(N,
368   cuda_x, cuda_r, cuda_p, cuda_Ap,
369   alpha, beta);
370
371   cudaMemcpy(cuda_pAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
372   cudaMemcpy(cuda_ApAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
373   cudaMemcpy(cuda_rr, &zero, sizeof(double), cudaMemcpyHostToDevice);
374   part2<<<BLOCK_SIZE, GRID_SIZE>>>(N,
375   csr_rowoffsets, csr_colindices, csr_values,
376   cuda_r, cuda_p, cuda_Ap,
377   cuda_ApAp, cuda_pAp, cuda_rr);
378
379   cudaDeviceSynchronize();
380   cudaMemcpy(&rr, cuda_rr, sizeof(double), cudaMemcpyDeviceToHost);
```

```
381     cudaMemcpy(&pAp, cuda_pAp, sizeof(double), cudaMemcpyDeviceToHost);
382     cudaMemcpy(&ApAp, cuda_ApAp, sizeof(double), cudaMemcpyDeviceToHost);
383     cudaDeviceSynchronize();
384     // line 10:
385     double rel_norm = std::sqrt(rr / initial_residual_squared);
386     if (rel_norm < 1e-6) {
387     break;
388     }
389     alpha = rr / pAp;
390     //beta = (alpha*alpha * ApAp - rr) / rr;
391     beta = alpha * alpha * ApAp / rr - 1;
392
393   #ifdef DEBUG
394     if (iters%100==0) {
395     std::cout << "Norm after " << iters << " iterations:\n"
396     << "rel. norm: " << rel_norm << "\n"
397     << "abs. norm: " << std::sqrt(beta) << std::endl;
398     }
399   #endif
400     if (iters > 10000)
401     break; // solver didn't converge
402     ++iters;
403     }
404     cudaMemcpy(solution, cuda_x, sizeof(double) * N, cudaMemcpyDeviceToHost);
405
406     cudaDeviceSynchronize();
407   #ifdef DEBUG
408     std::cout << "Time elapsed: " << timer.get() << " (" << timer.get() / iters << " per
          iteration)" << std::endl;
409   #endif
410     if (iters > 10000)
411       std::cout << "Conjugate Gradient did NOT converge within 10000 iterations"
412       << std::endl;
413       else
414       std::cout << "Conjugate Gradient converged in " << iters << " iterations."
415       << std::endl;
416
417     // Vectors
418     cudaFree(cuda_x);
419     cudaFree(cuda_p);
420     cudaFree(cuda_r);
421     cudaFree(cuda_Ap);
422     // Scalers
423     cudaFree(cuda_pAp);
424     cudaFree(cuda_ApAp);
425     cudaFree(cuda_rr);
426     return iters;
427   }
428
429
430   int main() {
431
432     std::string csv_name = CSV_NAME;
433     std::cout << "\n\nResults in csv form can be found here\nhttps://gtx1080.360252.org/2020/
          ex5/" + csv_name << std::endl;
434     std::string header = "N;M;unknowns;nz_found;times_assemble_cpu;times_assemble_gpu;times_cg
          ;iters;norm_after";
435
436     std::fstream csv;
437     csv.open(csv_name, std::fstream::out | std::fstream::trunc);
438     csv << header << std::endl;
439     csv.close();
440
441     Timer timer;
442
443
444     std::vector<int> N_vec;
445     for (int i = 64; i <= 10000; i *= 2) {
446       N_vec.push_back(i);
447     }
448
449     for (int& N: N_vec) {
```

```
450       std::cout << "N = M = " << N << std::endl;
451       int M = N;
452       //
453       // Allocate host arrays for reference
454       //
455       int *row_offsets = (int *)malloc(sizeof(int) * (N*M+1));
456
457
458       //
459       // Allocate CUDA-arrays
460       //
461       int *cuda_row_offsets;
462       int *cuda_row_offsets_2;
463       double *cuda_values;
464       int *cuda_columns;
465
466       cudaMalloc(&cuda_row_offsets, sizeof(int) * (N*M+1));
467       cudaMalloc(&cuda_row_offsets_2, sizeof(int) * (N*M+1));
468
469
470       // Perform the calculations
471       int numberOfValues;
472       timer.reset();
473       count_nnz<<<GRID_SIZE, BLOCK_SIZE>>>(cuda_row_offsets_2, N, M);
474       exclusive_scan(cuda_row_offsets_2, cuda_row_offsets, N*M+1);
475       cudaMemcpy(row_offsets, cuda_row_offsets, sizeof(int) * (N*M+1), cudaMemcpyDeviceToHost)
              ;
476       numberOfValues = row_offsets[N*M];
477
478     #ifdef DEBUG
479       printContainer(row_offsets, N*M+1, 4);
480       std::cout << std::endl;
481     #endif
482
483       double *values = (double *)malloc(sizeof(double) * numberOfValues);
484       int *columns = (int *)malloc(sizeof(int) * numberOfValues);
485       cudaMalloc(&cuda_columns, sizeof(int) * numberOfValues);
486       cudaMalloc(&cuda_values, sizeof(double) * numberOfValues);
487
488       assembleA<<<GRID_SIZE, BLOCK_SIZE>>>(cuda_values, cuda_columns, cuda_row_offsets, N, M);
489       double time_assemble_gpu = timer.get();
490
491       cudaMemcpy(values, cuda_values, sizeof(double) * numberOfValues, cudaMemcpyDeviceToHost)
              ;
492       cudaMemcpy(columns, cuda_columns, sizeof(int) * numberOfValues, cudaMemcpyDeviceToHost);
493
494   #ifdef DEBUG
495       printContainer(values, numberOfValues, 4);
496       std::cout << std::endl;
497       printContainer(columns, numberOfValues, 4);
498   #endif
499   /* -------- CPU -----------*/
500       int *csr_rowoffsets = (int *)malloc(sizeof(int) * (N*M+1));
501       int *csr_colindices = (int *)malloc(sizeof(int) * (N*M+1)*5);
502       double *csr_values = (double *)malloc(sizeof(double) * (N*M+1)*5);
503
504     #ifdef DEBUG
505       std::cout << "generate CPU "<<std::endl;
506     #endif
507       timer.reset();
508       generate_fdm_laplace(N, csr_rowoffsets, csr_colindices, csr_values);
509       double time_assemble_cpu = timer.get();
510
511   /* -------- CPU -----------*/
512
513         //
514       // Allocate solution vector and right hand side:
515       //
516       double *solution = (double *)malloc(sizeof(double) * N*M);
517       double *rhs = (double *)malloc(sizeof(double) * N*M);
518       std::fill(rhs, rhs + N*M, 1.);
519
```

```
520       timer.reset();
521       int iters = conjugate_gradient(N*M, cuda_row_offsets, cuda_columns, cuda_values, rhs,
              solution);
522       double runtime = timer.get();
523     #ifdef DEBUG
524       std::cout << "runtime: " << runtime << std::endl;
525     #endif
526       double residual_norm = relative_residual(N*M, row_offsets, columns, values, rhs,
              solution);
527
528     #ifndef DEBUG
529
530       csv.open (csv_name, std::fstream::out | std::fstream::app);
531       csv << header << std::endl;
532       csv << N << SEP << M << SEP
533         << N*M << SEP
534         << numberOfValues << SEP
535         << time_assemble_cpu << SEP
536         << time_assemble_gpu << SEP
537         << runtime << SEP
538         << iters << SEP
539         << residual_norm << std::endl;
540       csv.close();
541     #endif
542
543       //
544       // Clean up:
545       //
546       free(row_offsets);
547       free(values);
548       free(columns);
549       free(csr_rowoffsets);
550       free(csr_colindices);
551       free(csr_values);
552       cudaFree(cuda_row_offsets);
553       cudaFree(cuda_row_offsets_2);
554       cudaFree(cuda_values);
555       cudaFree(cuda_columns);
556     }
557
558     std::cout << "\n\nResults in csv form can be found here\nhttps://gtx1080.360252.org/2020/
          ex5/" + csv_name << std::endl;
559     return EXIT_SUCCESS;
560   }
```

Listing 4: Ex5.2: FD on GPU