

Computational Science on Many-Core Architectures:

Exercise 6:

Viktor Beck, 11713110

November 2022

Task 1: Performance Modeling: Parameter Identification

a)

- **scan_kernel_1:** As parameters the kernel gets the input array X, the array Y we will write to, N the length of X and an array "carries". At first we tell each thread how much work it has to do which depends on the dimension of the number of blocks and number of threads. After that each block performs an exclusive scan (actually an inclusive scan and then saving it on positions shifted by 1) on the one part of the input array and saves it to the output array. So e.g., we have an array X consisting only of one and blocks of 4 threads. This will lead to a repeating pattern of numbers 0 to 3 in the output array (like: 0, 1, 2, 3, 0, 1, 2, ...). The other thing that is computed is the block offset that is saved to carries. If we look at above's example and with say 15 threads per block carries would look like this: 4, 4, 4, 3, 0, 0,
- **scan_kernel_2:** In this kernel we only transform the carries array. We simply perform an exclusive scan to get the "absolute" offsets from the "relative" offsets. So, again with above's example, carries now holds 0, 4, 8, 12, 15, 0, 0,
- **scan_kernel_3:** Here, we add the carries to the respective part of the arrays. So, for the arrays first 4 (= threads per block) entries we add 0, for the next 4 we add 4, then 8 and so on which leads to the final result.

b)

The following function shifts the exclusive scan by one to the left and adds the last value (on the right) to the shifted array such that we obtain the inclusive scan (full code in Appendix).

```
1 double* inclusive_scan(double const * x,  
2                        double const * cuda_x,  
3                        double      * cuda_y, int N)  
4 {  
5     exclusive_scan(cuda_x, cuda_y, N);  
6     // create arrays to store exclusive and inclusive scan result  
7     double *z_ex = (double *)malloc(sizeof(double) * N);  
8     double *z_in = (double *)malloc(sizeof(double) * N);  
9  
10    cudaMemcpy(z_ex, cuda_y, sizeof(double) * N, cudaMemcpyDeviceToHost);  
11  
12    // shift by one to get inclusive scan (probably faster with a GPU kernel)  
13    for (int i=0; i<N; i++){
```

```

14     z_in[i] = z_ex[i+1];
15 }
16 z_in[N-1] = z_ex[N-1] + x[N-1];
17 free(z_ex);
18 return z_in;
19 }

```

c)

Here we simply delete the following part of `scan_kernel_1()`:

```

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

```

d)

One can see in the figure below that the inclusive scan implementation of b) is the worst. The reason is quite obvious because we perform additionally to the exclusive scan the shifting of the array. But since the CPU is quite fast with that it does not make a huge difference. Also, the inclusive scan from c) is basically the same as the exclusive scan (just without that one if-statement).

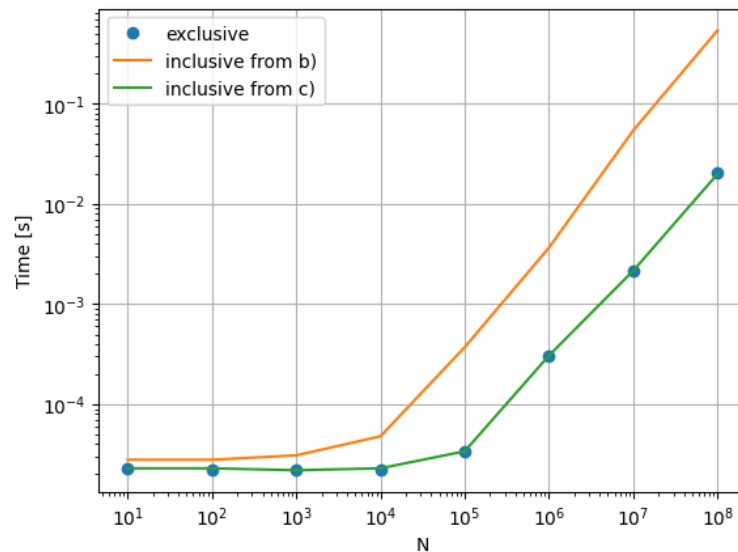


Figure 1: Comparison of execution times of exclusive and inclusive scan implementations for different N

Task 2: Finite Differences on the GPU

There is not too much to explain here - see Appendix for the whole code.

a)

This is basically the same function as the one from the lecture.

```
1 __global__ void count_nnz(int *row_offsets, int N)
2 {
3     for (int row = blockDim.x * blockIdx.x + threadIdx.x;
4         row < N * N;
5         row += gridDim.x * blockDim.x)
6     {
7         int nnz_for_this_node = 1;
8         int i = row / N;
9         int j = row % N;
10        if (i > 0)
11            nnz_for_this_node += 1;
12        if (j > 0)
13            nnz_for_this_node += 1;
14        if (i < N - 1)
15            nnz_for_this_node += 1;
16        if (j < N - 1)
17            nnz_for_this_node += 1;
18        row_offsets[row] = nnz_for_this_node;
19    }
20 }
```

b)

The exclusive scan is called in the main function as:

```
1 exclusive_scan(cuda_nnz, cuda_row_offsets, N * N+1);
```

c)

Here we use again a function from the lecture notes. The left/right/above neighbors were implemented similarly to the bottom neighbors.

```
1 __global__ void assembleA(int *row_offsets,
2                           int *col_indices,
3                           double *values,
4                           int N)
5 {
6     for (int row = blockDim.x * blockIdx.x + threadIdx.x;
7         row < N * N;
8         row += gridDim.x * blockDim.x)
9     {
10        int i = row / N;
11        int j = row % N;
12        int this_row_offset = row_offsets[row];
13        // diagonal entry
14        col_indices[this_row_offset] = i * N + j;
15        values[this_row_offset] = 4;
16        this_row_offset += 1;
17        if (i > 0)
18        { // bottom neighbor
```

```

19     col_indices[this_row_offset] = (i - 1) * N + j;
20     values[this_row_offset] = -1;
21     this_row_offset += 1;
22 }
23 if (j > 0)
24 {
25     col_indices[this_row_offset] = i * N + (j - 1);
26     values[this_row_offset] = -1;
27     this_row_offset += 1;
28 }
29 if (i < N - 1)
30 {
31     col_indices[this_row_offset] = (i + 1) * N + j;
32     values[this_row_offset] = -1;
33     this_row_offset += 1;
34 }
35 if (j < N - 1)
36 {
37     col_indices[this_row_offset] = i * N + (j + 1);
38     values[this_row_offset] = -1;
39     this_row_offset += 1;
40 }
41 }
42 }

```

d)

Most of the magic happens in the modified `solve_system()` function. There, I adjusted the allocations of `csr_colindices` and `csr_values` such that they have space corresponding to the total number of offsets. Also, the `cuda_csr_offsets` are the input for the function. For the rest, see code below:

```

1 void solve_system(int *cuda_csr_rowoffsets, int n)
2 {
3
4     int N = n * n; // number of unknowns to solve for
5
6     std::cout << "Solving Ax=b with " << N << " unknowns." << std::endl;
7
8     //
9     // Allocate CSR arrays.
10    //
11
12    int *csr_rowoffsets = (int *)malloc(sizeof(int) * n * n);
13    cudaMemcpy(csr_rowoffsets, cuda_csr_rowoffsets, sizeof(int) * n * n, cudaMemcpyDeviceToHost);
14    //print_array(csr_rowoffsets, N);
15    // get total number of non-zero entries
16    int nnz_total = csr_rowoffsets[N-1];
17    //std::cout << nnz_total << std::endl;
18
19    int *csr_colindices = (int *)malloc(sizeof(int) * nnz_total);
20    double *csr_values = (double *)malloc(sizeof(double) * nnz_total);
21
22    int *cuda_csr_colindices; cudaMalloc(&cuda_csr_colindices, sizeof(int) * nnz_total);
23    double *cuda_csr_values; cudaMalloc(&cuda_csr_values, sizeof(double) * nnz_total);
24
25
26    //
27    // fill CSR matrix with values
28    //

```

```

29     assembleA<<<16,16>>>(cuda_csr_rowoffsets,
30                           cuda_csr_colindices,
31                           cuda_csr_values,
32                           n);
33
34
35     cudaMemcpy(csr_values, cuda_csr_values, sizeof(double) * nnz_total, cudaMemcpyDeviceToHost);
36     cudaMemcpy(csr_colindices, cuda_csr_colindices, sizeof(int) * nnz_total, cudaMemcpyDeviceToHost);
37
38     //print_array(csr_values, nnz_total);
39     //print_array(csr_colindices, nnz_total);
40
41
42     //
43     // Allocate solution vector and right hand side:
44     //
45     double *solution = (double *)malloc(sizeof(double) * N);
46     double *rhs = (double *)malloc(sizeof(double) * N);
47     std::fill(rhs, rhs + N, 1);
48
49     //
50     // Call Conjugate Gradient implementation with GPU arrays
51     //
52     conjugate_gradient(N, cuda_csr_rowoffsets, cuda_csr_colindices, cuda_csr_values, rhs, solution);
53
54     //
55     // Check for convergence:
56     //
57
58     double residual_norm = relative_residual(N, csr_rowoffsets, csr_colindices, csr_values, rhs, solution);
59     std::cout << "Relative residual norm: " << residual_norm
60               << " (should be smaller than 1e-6)" << std::endl;
61
62     cudaFree(cuda_csr_rowoffsets);
63     cudaFree(cuda_csr_colindices);
64     cudaFree(cuda_csr_values);
65     free(solution);
66     free(rhs);
67     free(csr_rowoffsets);
68     free(csr_colindices);
69     free(csr_values);
70
71 }

```

Appendix

Task 1.b)

```
1 #include "poisson2d.hpp"
2 #include "timer.hpp"
3 #include <algorithm>
4 #include <iostream>
5 #include <stdio.h>
6 #include <cstring>
7
8
9 __global__ void scan_kernel_1(double const *X,
10                             double *Y,
11                             int N,
12                             double *carries)
13 {
14     __shared__ double shared_buffer[256];
15     double my_value;
16
17     unsigned int work_per_thread = (N - 1) / (gridDim.x * blockDim.x) + 1;
18     unsigned int block_start = work_per_thread * blockDim.x * blockIdx.x;
19     unsigned int block_stop = work_per_thread * blockDim.x * (blockIdx.x + 1);
20     unsigned int block_offset = 0;
21
22     // run scan on each section
23     for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
24     {
25         // load data:
26         my_value = (i < N) ? X[i] : 0;
27
28         // inclusive scan in shared buffer:
29         for(unsigned int stride = 1; stride < blockDim.x; stride *= 2)
30         {
31             __syncthreads();
32             shared_buffer[threadIdx.x] = my_value;
33             __syncthreads();
34             if (threadIdx.x >= stride)
35                 my_value += shared_buffer[threadIdx.x - stride];
36         }
37         __syncthreads();
38         shared_buffer[threadIdx.x] = my_value;
39         __syncthreads();
40
41         // exclusive scan requires us to write a zero value at the beginning of each block
42         my_value = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
43
44         // write to output array
45         if (i < N)
46             Y[i] = block_offset + my_value;
47
48         block_offset += shared_buffer[blockDim.x-1];
49     }
50
51     // write carry:
52     if (threadIdx.x == 0)
53         carries[blockIdx.x] = block_offset;
54
55 }
56
57 // exclusive-scan of carries
```

```

58 __global__ void scan_kernel_2(double *carries)
59 {
60     __shared__ double shared_buffer[256];
61
62     // load data:
63     double my_carry = carries[threadIdx.x];
64
65     // exclusive scan in shared buffer:
66
67     for(unsigned int stride = 1; stride < blockDim.x; stride *= 2)
68     {
69         __syncthreads();
70         shared_buffer[threadIdx.x] = my_carry;
71         __syncthreads();
72         if (threadIdx.x >= stride)
73             my_carry += shared_buffer[threadIdx.x - stride];
74     }
75     __syncthreads();
76     shared_buffer[threadIdx.x] = my_carry;
77     __syncthreads();
78
79     // write to output array
80     carries[threadIdx.x] = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
81 }
82
83 __global__ void scan_kernel_3(double *Y, int N,
84                             double const *carries)
85 {
86     unsigned int work_per_thread = (N - 1) / (gridDim.x * blockDim.x) + 1;
87     unsigned int block_start = work_per_thread * blockDim.x * blockIdx.x;
88     unsigned int block_stop = work_per_thread * blockDim.x * (blockIdx.x + 1);
89
90     __shared__ double shared_offset;
91
92     if (threadIdx.x == 0)
93         shared_offset = carries[blockIdx.x];
94
95     __syncthreads();
96
97     // add offset to each element in the block:
98     for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
99         if (i < N)
100             Y[i] += shared_offset;
101 }
102
103
104
105
106 void exclusive_scan(double const * input,
107                   double * output, int N)
108 {
109     int num_blocks = 256;
110     int threads_per_block = 256;
111
112     double *carries;
113     cudaMalloc(&carries, sizeof(double) * num_blocks);
114
115
116     // First step: Scan within each thread group and write carries
117     scan_kernel_1<<<num_blocks, threads_per_block>>>(input, output, N, carries);
118
119     // for printing intermediate results

```

```

120  /*
121  double *z = (double *)malloc(sizeof(double) * N);
122  cudaMemcpy(z, output, sizeof(double) * N, cudaMemcpyDeviceToHost);
123  printf("output:\n");
124  for (int i = 0; i<N; i++){
125      std::cout << z[i] << "," << std::endl;
126  }
127  double *carries_cpu = (double *)malloc(sizeof(double) * num_blocks);
128  cudaMemcpy(carries_cpu, carries, sizeof(double) * num_blocks, cudaMemcpyDeviceToHost);
129  printf("carries:\n");
130  for (int i = 0; i<num_blocks; i++){
131      std::cout << carries_cpu[i] << "," << std::endl;
132  }*/
133
134
135  // Second step: Compute offset for each thread group (exclusive scan for each thread group)
136  scan_kernel_2<<<1, num_blocks>>>(carries);
137
138  // for printing intermediate results
139  /*
140  cudaMemcpy(carries_cpu, carries, sizeof(double) * num_blocks, cudaMemcpyDeviceToHost);
141  printf("carries after scan 2:\n");
142  for (int i = 0; i<num_blocks; i++){
143      std::cout << carries_cpu[i] << "," << std::endl;
144  }*/
145
146  // Third step: Offset each thread group accordingly
147  scan_kernel_3<<<num_blocks, threads_per_block>>>(output, N, carries);
148
149  cudaFree(carries);
150 }
151
152 double* inclusive_scan(double const * x,
153                       double const * cuda_x,
154                       double      * cuda_y, int N)
155 {
156     exclusive_scan(cuda_x, cuda_y, N);
157     // create arrays to store exclusive and inclusive scan result
158     double *z_ex = (double *)malloc(sizeof(double) * N);
159     double *z_in = (double *)malloc(sizeof(double) * N);
160
161     cudaMemcpy(z_ex, cuda_y, sizeof(double) * N, cudaMemcpyDeviceToHost);
162
163     // shift by one to get inclusive scan (probably faster with a GPU kernel)
164     for (int i=0; i<N; i++){
165         z_in[i] = z_ex[i+1];
166     }
167     z_in[N-1] = z_ex[N-1] + x[N-1];
168     free(z_ex);
169     return z_in;
170 }
171
172
173
174
175 int main() {
176
177     int N = 200;
178
179     //
180     // Allocate host arrays for reference
181     //

```



```

12     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
13     {
14         double sum = 0;
15         for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++)
16         {
17             sum += csr_values[k] * x[csr_colindices[k]];
18         }
19         y[i] = sum;
20     }
21 }
22
23 // x <- x + alpha * y
24 __global__ void cuda_vecadd(int N, double *x, double *y, double alpha)
25 {
26     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
27         x[i] += alpha * y[i];
28 }
29
30 // x <- y + alpha * x
31 __global__ void cuda_vecadd2(int N, double *x, double *y, double alpha)
32 {
33     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
34         x[i] = y[i] + alpha * x[i];
35 }
36
37 // result = (x, y)
38 __global__ void cuda_dot_product(int N, double *x, double *y, double *result)
39 {
40     __shared__ double shared_mem[512];
41
42     double dot = 0;
43     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
44     {
45         dot += x[i] * y[i];
46     }
47
48     shared_mem[threadIdx.x] = dot;
49     for (int k = blockDim.x / 2; k > 0; k /= 2)
50     {
51         __syncthreads();
52         if (threadIdx.x < k)
53         {
54             shared_mem[threadIdx.x] += shared_mem[threadIdx.x + k];
55         }
56     }
57
58     if (threadIdx.x == 0)
59         atomicAdd(result, shared_mem[0]);
60 }
61
62 /** Implementation of the conjugate gradient algorithm.
63  *
64  * The control flow is handled by the CPU.
65  * Only the individual operations (vector updates, dot products, sparse
66  * matrix-vector product) are transferred to CUDA kernels.
67  *
68  * The temporary arrays p, r, and Ap need to be allocated on the GPU for use
69  * with CUDA. Modify as you see fit.
70  */
71 void conjugate_gradient(int N, // number of unknowns
72                        int *csr_rowoffsets, int *csr_colindices,
73                        double *csr_values, double *rhs, double *solution)

```

```

74 //, double *init_guess)    // feel free to add a nonzero initial guess as needed
75 {
76     // initialize timer
77     Timer timer;
78
79     // clear solution vector (it may contain garbage values):
80     std::fill(solution, solution + N, 0);
81
82     // initialize work vectors:
83     double alpha, beta;
84     double *cuda_solution, *cuda_p, *cuda_r, *cuda_Ap, *cuda_scalar;
85     cudaMalloc(&cuda_p, sizeof(double) * N);
86     cudaMalloc(&cuda_r, sizeof(double) * N);
87     cudaMalloc(&cuda_Ap, sizeof(double) * N);
88     cudaMalloc(&cuda_solution, sizeof(double) * N);
89     cudaMalloc(&cuda_scalar, sizeof(double));
90
91     cudaMemcpy(cuda_p, rhs, sizeof(double) * N, cudaMemcpyHostToDevice);
92     cudaMemcpy(cuda_r, rhs, sizeof(double) * N, cudaMemcpyHostToDevice);
93     cudaMemcpy(cuda_solution, solution, sizeof(double) * N, cudaMemcpyHostToDevice);
94
95     const double zero = 0;
96     double residual_norm_squared = 0;
97     cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
98     cuda_dot_product<<<512, 512>>>(N, cuda_r, cuda_r, cuda_scalar);
99     cudaMemcpy(&residual_norm_squared, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
100
101     double initial_residual_squared = residual_norm_squared;
102
103     int iters = 0;
104     cudaDeviceSynchronize();
105     timer.reset();
106     while (1)
107     {
108
109         // line 4: A*p:
110         cuda_csr_matvec_product<<<512, 512>>>(N, csr_rowoffsets, csr_colindices, csr_values, cuda_p, cuda_r);
111
112         // lines 5,6:
113         cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
114         cuda_dot_product<<<512, 512>>>(N, cuda_p, cuda_Ap, cuda_scalar);
115         cudaMemcpy(&alpha, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
116         alpha = residual_norm_squared / alpha;
117
118         // line 7:
119         cuda_vecadd<<<512, 512>>>(N, cuda_solution, cuda_p, alpha);
120
121         // line 8:
122         cuda_vecadd<<<512, 512>>>(N, cuda_r, cuda_Ap, -alpha);
123
124         // line 9:
125         beta = residual_norm_squared;
126         cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
127         cuda_dot_product<<<512, 512>>>(N, cuda_r, cuda_r, cuda_scalar);
128         cudaMemcpy(&residual_norm_squared, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
129
130         // line 10:
131         if (std::sqrt(residual_norm_squared / initial_residual_squared) < 1e-6)
132         {
133             break;
134         }
135     }

```

```

136 // line 11:
137 beta = residual_norm_squared / beta;
138
139 // line 12:
140 cuda_vecadd2<<<512, 512>>>(N, cuda_p, cuda_r, beta);
141
142 if (iters > 10000)
143     break; // solver didn't converge
144 ++iters;
145 }
146 cudaMemcpy(solution, cuda_solution, sizeof(double) * N, cudaMemcpyDeviceToHost);
147
148 cudaDeviceSynchronize();
149 std::cout << "Time_elapsed:" << timer.get() << " " << timer.get() / iters << "per_iteration" <<
150
151 if (iters > 10000)
152     std::cout << "Conjugate_Gradient_did_NOT_converge_within_10000_iterations"
153     << std::endl;
154 else
155     std::cout << "Conjugate_Gradient_converged_in" << iters << "iterations."
156     << std::endl;
157
158 cudaFree(cuda_p);
159 cudaFree(cuda_r);
160 cudaFree(cuda_Ap);
161 cudaFree(cuda_solution);
162 cudaFree(cuda_scalar);
163 }
164
165
166
167 // ##### MODIFIED IMLEMENTATIONS HERE #####
168
169 void print_array(int *a, int N)
170 {
171     int counter = 0;
172     std::cout << "array:\n"
173     << std::endl;
174     for (int i = 0; i < N; i++)
175     {
176         std::cout << a[i] << std::endl;
177         counter++;
178     }
179     std::cout << "Length_of_array:" << counter << std::endl;
180 }
181
182 void print_array(double *a, int N)
183 {
184     int counter = 0;
185     std::cout << "array:\n"
186     << std::endl;
187     for (int i = 0; i < N; i++)
188     {
189         std::cout << a[i] << std::endl;
190         counter++;
191     }
192     std::cout << "Length_of_array:" << counter << std::endl;
193 }
194
195 __global__ void count_nnz(int *row_offsets, int N)
196 {
197     for (int row = blockDim.x * blockIdx.x + threadIdx.x;

```

```

198         row < N * N;
199         row += gridDim.x * blockDim.x)
200     {
201         int nnz_for_this_node = 1;
202         int i = row / N;
203         int j = row % N;
204         if (i > 0)
205             nnz_for_this_node += 1;
206         if (j > 0)
207             nnz_for_this_node += 1;
208         if (i < N - 1)
209             nnz_for_this_node += 1;
210         if (j < N - 1)
211             nnz_for_this_node += 1;
212         row_offsets[row] = nnz_for_this_node;
213     }
214 }
215
216 __global__ void assembleA(int *row_offsets,
217                           int *col_indices,
218                           double *values,
219                           int N)
220 {
221     for (int row = blockDim.x * blockIdx.x + threadIdx.x;
222          row < N * N;
223          row += gridDim.x * blockDim.x)
224     {
225         int i = row / N;
226         int j = row % N;
227         int this_row_offset = row_offsets[row];
228         // diagonal entry
229         col_indices[this_row_offset] = i * N + j;
230         values[this_row_offset] = 4;
231         this_row_offset += 1;
232         if (i > 0)
233         { // bottom neighbor
234             col_indices[this_row_offset] = (i - 1) * N + j;
235             values[this_row_offset] = -1;
236             this_row_offset += 1;
237         }
238         if (j > 0)
239         {
240             col_indices[this_row_offset] = i * N + (j - 1);
241             values[this_row_offset] = -1;
242             this_row_offset += 1;
243         }
244         if (i < N - 1)
245         {
246             col_indices[this_row_offset] = (i + 1) * N + j;
247             values[this_row_offset] = -1;
248             this_row_offset += 1;
249         }
250         if (j < N - 1)
251         {
252             col_indices[this_row_offset] = i * N + (j + 1);
253             values[this_row_offset] = -1;
254             this_row_offset += 1;
255         }
256     }
257 }
258
259 /** Solve a system with 'n * n' unknowns

```

```

260 */
261 void solve_system(int *cuda_csr_rowoffsets, int n)
262 {
263
264     int N = n * n; // 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
272     int *csr_rowoffsets = (int *)malloc(sizeof(int) * n * n);
273     cudaMemcpy(csr_rowoffsets, cuda_csr_rowoffsets, sizeof(int) * n * n, cudaMemcpyDeviceToHost);
274     //print_array(csr_rowoffsets, N);
275     // get total number of non-zero entries
276     int nnz_total = csr_rowoffsets[N-1];
277     //std::cout << nnz_total << std::endl;
278
279     int *csr_colindices = (int *)malloc(sizeof(int) * nnz_total);
280     double *csr_values = (double *)malloc(sizeof(double) * nnz_total);
281
282     int *cuda_csr_colindices; cudaMalloc(&cuda_csr_colindices, sizeof(int) * nnz_total);
283     double *cuda_csr_values; cudaMalloc(&cuda_csr_values, sizeof(double) * nnz_total);
284
285
286     //
287     // fill CSR matrix with values
288     //
289     assembleA<<<16,16>>>(cuda_csr_rowoffsets,
290                          cuda_csr_colindices,
291                          cuda_csr_values,
292                          n);
293
294
295     cudaMemcpy(csr_values, cuda_csr_values, sizeof(double) * nnz_total, cudaMemcpyDeviceToHost);
296     cudaMemcpy(csr_colindices, cuda_csr_colindices, sizeof(int) * nnz_total, cudaMemcpyDeviceToHost);
297
298     //print_array(csr_values, nnz_total);
299     //print_array(csr_colindices, nnz_total);
300
301
302     //
303     // Allocate solution vector and right hand side:
304     //
305     double *solution = (double *)malloc(sizeof(double) * N);
306     double *rhs = (double *)malloc(sizeof(double) * N);
307     std::fill(rhs, rhs + N, 1);
308
309     //
310     // Call Conjugate Gradient implementation with GPU arrays
311     //
312     conjugate_gradient(N, cuda_csr_rowoffsets, cuda_csr_colindices, cuda_csr_values, rhs, solution);
313
314     //
315     // Check for convergence:
316     //
317
318     double residual_norm = relative_residual(N, csr_rowoffsets, csr_colindices, csr_values, rhs, solution);
319     std::cout << "Relative residual norm: " << residual_norm
320               << " (should be smaller than 1e-6)" << std::endl;
321

```

```

322     cudaFree(cuda_csr_rowoffsets);
323     cudaFree(cuda_csr_colindices);
324     cudaFree(cuda_csr_values);
325     free(solution);
326     free(rhs);
327     free(csr_rowoffsets);
328     free(csr_colindices);
329     free(csr_values);
330
331 }
332
333 __global__ void scan_kernel_1(int const *X,
334                               int *Y,
335                               int N,
336                               int *carries)
337 {
338     __shared__ double shared_buffer[256];
339     double my_value;
340
341     unsigned int work_per_thread = (N - 1) / (gridDim.x * blockDim.x) + 1;
342     unsigned int block_start = work_per_thread * blockDim.x * blockIdx.x;
343     unsigned int block_stop = work_per_thread * blockDim.x * (blockIdx.x + 1);
344     unsigned int block_offset = 0;
345
346     // run scan on each section
347     for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
348     {
349         // load data:
350         my_value = (i < N) ? X[i] : 0;
351
352         // inclusive scan in shared buffer:
353         for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)
354         {
355             __syncthreads();
356             shared_buffer[threadIdx.x] = my_value;
357             __syncthreads();
358             if (threadIdx.x >= stride)
359                 my_value += shared_buffer[threadIdx.x - stride];
360         }
361         __syncthreads();
362         shared_buffer[threadIdx.x] = my_value;
363         __syncthreads();
364
365         // exclusive scan requires us to write a zero value at the beginning of each block
366         my_value = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
367
368         // write to output array
369         if (i < N)
370             Y[i] = block_offset + my_value;
371
372         block_offset += shared_buffer[blockDim.x - 1];
373     }
374
375     // write carry:
376     if (threadIdx.x == 0)
377         carries[blockIdx.x] = block_offset;
378 }
379
380 // exclusive-scan of carries
381 __global__ void scan_kernel_2(int *carries)
382 {
383     __shared__ double shared_buffer[256];

```

```

384
385 // load data:
386 double my_carry = carries[threadIdx.x];
387
388 // exclusive scan in shared buffer:
389
390 for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)
391 {
392     __syncthreads();
393     shared_buffer[threadIdx.x] = my_carry;
394     __syncthreads();
395     if (threadIdx.x >= stride)
396         my_carry += shared_buffer[threadIdx.x - stride];
397 }
398 __syncthreads();
399 shared_buffer[threadIdx.x] = my_carry;
400 __syncthreads();
401
402 // write to output array
403 carries[threadIdx.x] = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
404 }
405
406 __global__ void scan_kernel_3(int *Y, int N,
407                             int const *carries)
408 {
409     unsigned int work_per_thread = (N - 1) / (gridDim.x * blockDim.x) + 1;
410     unsigned int block_start = work_per_thread * blockDim.x * blockIdx.x;
411     unsigned int block_stop = work_per_thread * blockDim.x * (blockIdx.x + 1);
412
413     __shared__ double shared_offset;
414
415     if (threadIdx.x == 0)
416         shared_offset = carries[blockIdx.x];
417
418     __syncthreads();
419
420     // add offset to each element in the block:
421     for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
422         if (i < N)
423             Y[i] += shared_offset;
424 }
425
426 void exclusive_scan(int const *input,
427                   int *output, int N)
428 {
429     int num_blocks = 256;
430     int threads_per_block = 256;
431
432     int *carries;
433     cudaMalloc(&carries, sizeof(int) * num_blocks);
434
435     // First step: Scan within each thread group and write carries
436     scan_kernel_1<<<num_blocks, threads_per_block>>>(input, output, N, carries);
437
438     // Second step: Compute offset for each thread group (exclusive scan for each thread group)
439     scan_kernel_2<<<1, num_blocks>>>(carries);
440
441     // Third step: Offset each thread group accordingly
442     scan_kernel_3<<<num_blocks, threads_per_block>>>(output, N, carries);
443
444     cudaFree(carries);
445 }

```



```

446
447
448 // #####
449
450 int main()
451 {
452     size_t N = 10;
453
454     // allocate stuff
455     int *nnz = (int *)malloc(sizeof(int) * N * N);
456     int *cuda_nnz;
457     cudaMalloc(&cuda_nnz, sizeof(int) * N * N);
458
459     int *row_offsets = (int *)malloc(sizeof(int) * N * N);
460     int *cuda_row_offsets;
461     cudaMalloc(&cuda_row_offsets, sizeof(int) * N * N);
462
463     // count non-zero entries - task a)
464     count_nnz<<<256, 256>>>(cuda_nnz, N);
465     cudaMemcpy(nnz, cuda_nnz, sizeof(int) * N * N, cudaMemcpyDeviceToHost);
466
467     // print_array(nnz, N*N);
468
469     // perform exclusive scan - task b)
470     exclusive_scan(cuda_nnz, cuda_row_offsets, N * N+1);
471
472     //cudaMemcpy(row_offsets, cuda_row_offsets, sizeof(int) * N * N, cudaMemcpyDeviceToHost);
473     // print_array(row_offsets, N * N);
474
475     solve_system(cuda_row_offsets, N); // solves a system with N*N unknowns
476
477     cudaFree(cuda_nnz);
478     free(nnz);
479     cudaFree(cuda_row_offsets);
480     free(row_offsets);
481
482     return EXIT_SUCCESS;
483 }

```