



360.252 - COMPUTATIONAL SCIENCE ON MANY-CORE ARCHITECTURES

WS 2020 - EXERCISE 5

Christian GOLLMANN, 01435044

Last update: November 23, 2020

Contents

1	Inclusive and Exclusive Scan 1	1
2	Inclusive and Exclusive Scan 2	6
3	Inclusive and Exclusive Scan 3	7
4	Inclusive and Exclusive Scan 4	8
5	Finite Differences on the GPU 1	9
6	Finite Differences on the GPU 2 and 3	10

1 Inclusive and Exclusive Scan 1

Let's start with `scan_kernel_1`. In my sketch I show how the situation looks like for a grid with 4 blocks and a blocksize of 6 threads per block. The for loop in line 24 goes over the values of X which belong to the respective block, in my case block 2. At the end of the for loop, we can write the 6 temporary scan results into the result vector Y. The current offset gets saved in `block_offset` and gets added to the values of the next iteration. Please refer to the sketch for further details. At the end of `scan_kernel_1`, every block holds its exclusively scanned respective values and the carry which is needed in the next steps.

In `scan_kernel.2` the respective carries get summed up so they can be added to the already exclusively scanned values in `scan_kernel.3`. Since a picture says more than thousand words, please also refer to the sketch in figure 2 for further details.

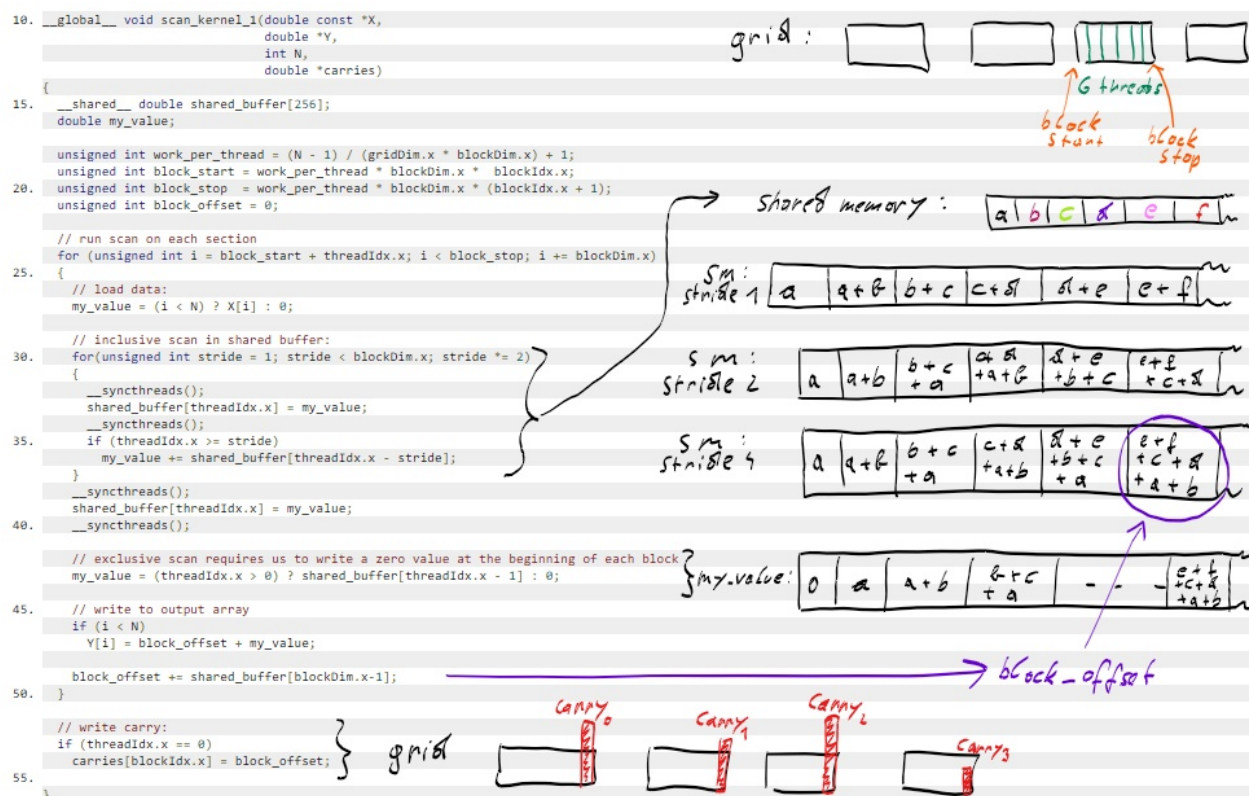


Figure 1: scan_kernel_1

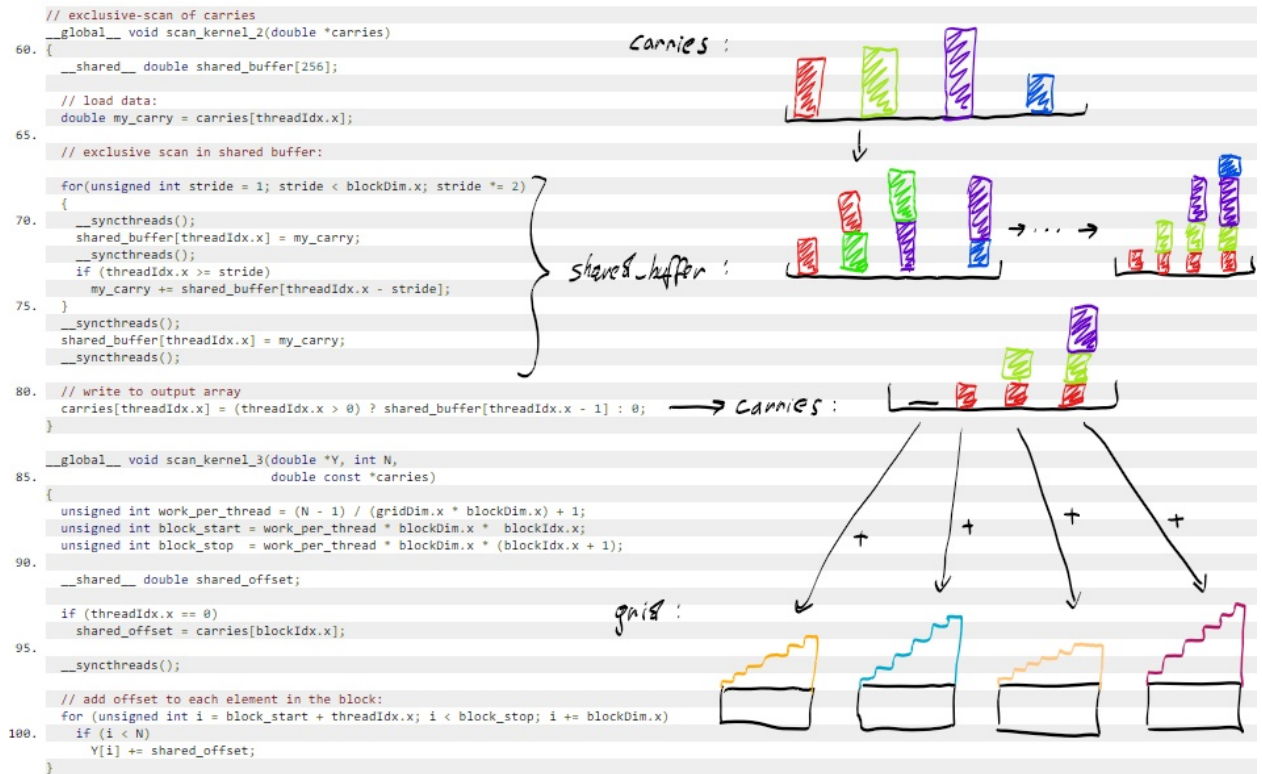


Figure 2: scan_kernel_2 and scan_kernel_3

For sake of completeness, I would also like to add the whole provided code here.

Listing 1: Provided Code for exclusive scan

```

1  #include "poisson2d.hpp"
2  #include "timer.hpp"
3  #include <algorithm>
4  #include <iostream>
5  #include <stdio.h>
6
7
8
9
10 __global__ void scan_kernel_1(double const *X,
11                             double *Y,
12                             int N,
13                             double *carries)
14 {
15     __shared__ double shared_buffer[256];
16     double my_value;
17
18     unsigned int work_per_thread = (N - 1) / (gridDim.x * blockDim.x) + 1;
19     unsigned int block_start = work_per_thread * blockDim.x * blockIdx.x;
20     unsigned int block_stop = work_per_thread * blockDim.x * (blockIdx.x + 1);
21     unsigned int block_offset = 0;
22
23     // run scan on each section
24     for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
25     {

```

```

26 // load data:
27 my_value = (i < N) ? X[i] : 0;
28
29 // inclusive scan in shared buffer:
30 for(unsigned int stride = 1; stride < blockDim.x; stride *= 2)
31 {
32     __syncthreads();
33     shared_buffer[threadIdx.x] = my_value;
34     __syncthreads();
35     if (threadIdx.x >= stride)
36         my_value += shared_buffer[threadIdx.x - stride];
37 }
38 __syncthreads();
39 shared_buffer[threadIdx.x] = my_value;
40 __syncthreads();
41
42 // exclusive scan requires us to write a zero value at the beginning of each block
43 my_value = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
44
45 // write to output array
46 if (i < N)
47     Y[i] = block_offset + my_value;
48
49     block_offset += shared_buffer[blockDim.x-1];
50 }
51
52 // write carry:
53 if (threadIdx.x == 0)
54     carries[blockIdx.x] = block_offset;
55
56 }
57
58 // exclusive-scan of carries
59 __global__ void scan_kernel_2(double *carries)
60 {
61     __shared__ double shared_buffer[256];
62
63     // load data:
64     double my_carry = carries[threadIdx.x];
65
66     // exclusive scan in shared buffer:
67
68     for(unsigned int stride = 1; stride < blockDim.x; stride *= 2)
69     {
70         __syncthreads();
71         shared_buffer[threadIdx.x] = my_carry;
72         __syncthreads();
73         if (threadIdx.x >= stride)
74             my_carry += shared_buffer[threadIdx.x - stride];
75     }
76     __syncthreads();
77     shared_buffer[threadIdx.x] = my_carry;
78     __syncthreads();
79
80     // write to output array
81     carries[threadIdx.x] = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
82 }

```

```

83
84 --global-- void scan_kernel_3(double *Y, int N,
85                               double const *carries)
86 {
87     unsigned int work_per_thread = (N - 1) / (gridDim.x * blockDim.x) + 1;
88     unsigned int block_start = work_per_thread * blockDim.x * blockIdx.x;
89     unsigned int block_stop = work_per_thread * blockDim.x * (blockIdx.x + 1);
90
91     --shared-- double shared_offset;
92
93     if (threadIdx.x == 0)
94         shared_offset = carries[blockIdx.x];
95
96     __syncthreads();
97
98     // add offset to each element in the block:
99     for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
100         if (i < N)
101             Y[i] += shared_offset;
102 }
103
104
105
106
107 void exclusive_scan(double const * input,
108                    double * output, int N)
109 {
110     int num_blocks = 256;
111     int threads_per_block = 256;
112
113     double *carries;
114     cudaMalloc(&carries, sizeof(double) * num_blocks);
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     // Second step: Compute offset for
120     each thread group (exclusive scan for each thread group)
121     scan_kernel_2<<<1, num_blocks>>>(carries);
122
123     // Third step: Offset each thread group accordingly
124     scan_kernel_3<<<num_blocks, threads_per_block>>>(output, N, carries);
125
126     cudaFree(carries);
127 }
128
129
130
131
132
133
134 int main() {
135
136     int N = 200;
137
138     //
139     // Allocate host arrays for reference

```

```

140 //
141 double *x = (double *)malloc(sizeof(double) * N);
142 double *y = (double *)malloc(sizeof(double) * N);
143 double *z = (double *)malloc(sizeof(double) * N);
144 std::fill(x, x + N, 1);
145
146 // reference calculation:
147 y[0] = 0;
148 for (std::size_t i=1; i<N; ++i) y[i] = y[i-1] + x[i-1];
149
150 //
151 // Allocate CUDA-arrays
152 //
153 double *cuda_x, *cuda_y;
154 cudaMalloc(&cuda_x, sizeof(double) * N);
155 cudaMalloc(&cuda_y, sizeof(double) * N);
156 cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice);
157
158
159 // Perform the exclusive scan and obtain results
160 exclusive_scan(cuda_x, cuda_y, N);
161 cudaMemcpy(z, cuda_y, sizeof(double) * N, cudaMemcpyDeviceToHost);
162
163 //
164 // Print first few entries for reference
165 //
166 std::cout << "CPU y: ";
167 for (int i=0; i<10; ++i) std::cout << y[i] << " ";
168 std::cout << " ... ";
169 for (int i=N-10; i<N; ++i) std::cout << y[i] << " ";
170 std::cout << std::endl;
171
172 std::cout << "GPU y: ";
173 for (int i=0; i<10; ++i) std::cout << z[i] << " ";
174 std::cout << " ... ";
175 for (int i=N-10; i<N; ++i) std::cout << z[i] << " ";
176 std::cout << std::endl;
177
178 //
179 // Clean up:
180 //
181 free(x);
182 free(y);
183 free(z);
184 cudaFree(cuda_x);
185 cudaFree(cuda_y);
186 return EXIT_SUCCESS;
187 }

```

2 Inclusive and Exclusive Scan 2

In order to make the scan exclusive, using what was already provided, I simply shift the end result.

Listing 2: Provided Code for exclusive scan

```
1  __global__ void makeInclusive(double *Y, int N, const double *X)
2  {
3      for (int i = blockDim.x * blockIdx.x + threadIdx.x; i < N-1;
4          i += gridDim.x * blockDim.x) {
5          Y[i] = Y[i+1];
6      }
7      if (blockDim.x * blockIdx.x + threadIdx.x == 0)
8          Y[N-1] += X[N-1];
9  }
10
11
12 void inclusive_scan(double const * input ,
13                   double          * output , int N)
14 {
15     int num_blocks = 256;
16     int threads_per_block = 256;
17
18     double *carries;
19     cudaMalloc(&carries , sizeof(double) * num_blocks);
20
21     // First step: Scan within each thread group and write carries
22     scan_kernel_1<<<num_blocks , threads_per_block>>>(input , output , N, carries);
23
24     // Second step: Compute offset for each
25     thread group (exclusive scan for each thread group)
26     scan_kernel_2<<<1, num_blocks>>>(carries);
27
28     // Third step: Offset each thread group accordingly
29     scan_kernel_3<<<num_blocks , threads_per_block>>>(output , N, carries);
30
31     // Make inclusive
32     makeInclusive<<<num_blocks , threads_per_block>>>(output , N, input);
33
34     cudaFree( carries );
35 }
```

3 Inclusive and Exclusive Scan 3

In order to make the scan exclusive, modifying the existing code, it is enough to remove line 43 from the provided code.

4 Inclusive and Exclusive Scan 4

The different implementations perform pretty similar what was no big surprise I think because they are almost the same. Just for higher values of N , the inclusive Scan that reuses the existing code and therefore has an additional kernel, is slower. This can be explained by the fact that there is one more kernel to be executed.

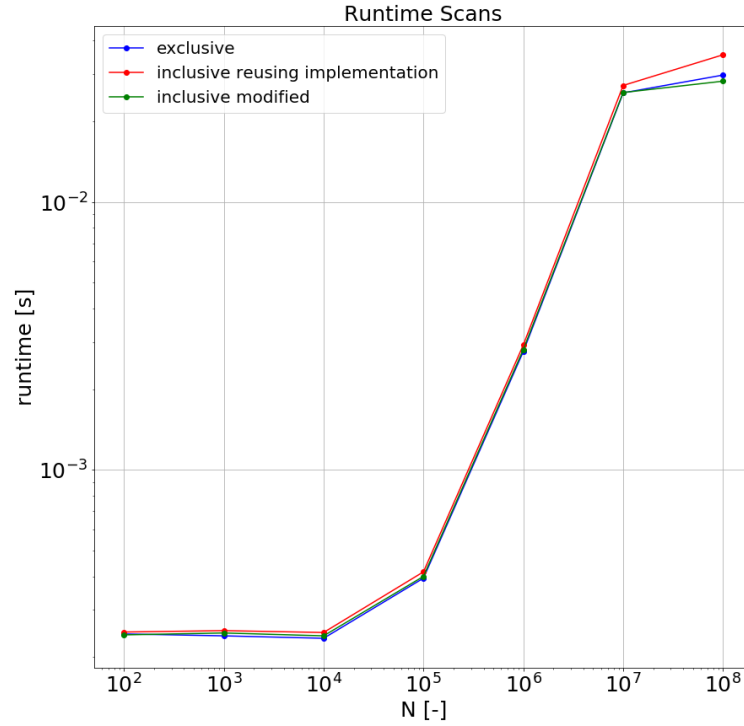


Figure 3: runtimes for different scan implementations

5 Finite Differences on the GPU 1

In order to write the kernel I took what was already provided at the lecture and modified it very slightly.

Listing 3: Kernel that counts and stores the number of nonzero entries

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

6 Finite Differences on the GPU 2 and 3

My code for this exercise point I tested with the Finite Difference matrix from NSSC I, lecture 3, see figure 4. If I run the code, I get the output you can see in figure 5.

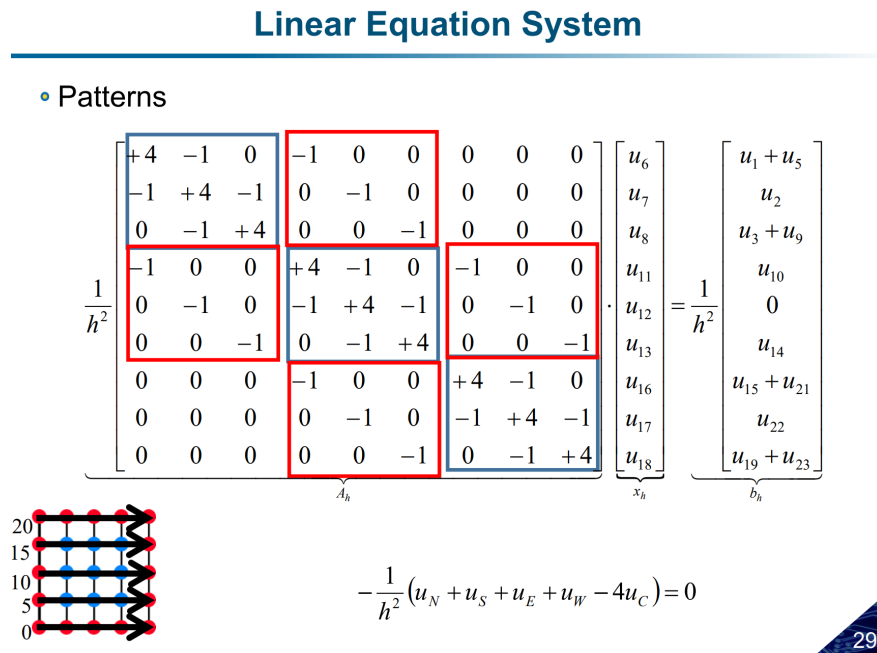


Figure 4: Test matrix, source: NSSC I (360.242), lecture 3

Compile output

[Compilation successful]

Run output

```
0 | 3 | 7 | 10 | 14 | 19 | 23 | 26 | 30 | 33 |
4 | -1 | -1 | -1 | 4 | -1 | -1 | -1 | 4 | -1 | -1 | -1 | -1 | 4 | -1 | -1 | -1 | 4 | -1 | -1 | -1 | 4 | -1 | -1 | -1 | 4 |
0 | 1 | 3 | 0 | 1 | 2 | 4 | 1 | 2 | 5 | 0 | 3 | 4 | 6 | 1 | 3 | 4 | 5 | 7 | 2 | 4 | 5 | 8 | 3 | 6 | 7 | 4 | 6 | 7 | 8 | 5 | 7 | 8 |
```

Figure 5: first row: exclusively scanned row offsets of system matrix, second row: nonzero matrix values of system matrix, last row: column indices of system matrix

Listing 4: Code to assemble a system matrix in CRS format

```

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

```

```

57
58 // exclusive-scan of carries
59 __global__ void scan_kernel_2(double *carries)
60 {
61     __shared__ double shared_buffer[256];
62
63     // load data:
64     double my_carry = carries[threadIdx.x];
65
66     // exclusive scan in shared buffer:
67
68     for(unsigned int stride = 1; stride < blockDim.x; stride *= 2)
69     {
70         __syncthreads();
71         shared_buffer[threadIdx.x] = my_carry;
72         __syncthreads();
73         if (threadIdx.x >= stride)
74             my_carry += shared_buffer[threadIdx.x - stride];
75     }
76     __syncthreads();
77     shared_buffer[threadIdx.x] = my_carry;
78     __syncthreads();
79
80     // write to output array
81     carries[threadIdx.x] = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
82 }
83
84 __global__ void scan_kernel_3(double *Y, int N,
85                             double const *carries)
86 {
87     unsigned int work_per_thread = (N - 1) / (gridDim.x * blockDim.x) + 1;
88     unsigned int block_start = work_per_thread * blockDim.x * blockIdx.x;
89     unsigned int block_stop = work_per_thread * blockDim.x * (blockIdx.x + 1);
90
91     __shared__ double shared_offset;
92
93     if (threadIdx.x == 0)
94         shared_offset = carries[blockIdx.x];
95
96     __syncthreads();
97
98     // add offset to each element in the block:
99     for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
100         if (i < N)
101             Y[i] += shared_offset;
102 }
103
104 __global__ void count_nnz(double* row_offsets, int N, int M) {
105     for(int row = blockDim.x * blockIdx.x + threadIdx.x;
106         row < N * M; row += gridDim.x * blockDim.x) {
107         int nnz_for_this_node = 1;
108         int i = row / N;
109         int j = row % N;
110
111         if(i > 0) nnz_for_this_node += 1;
112         if(j > 0) nnz_for_this_node += 1;
113         if(i < N-1) nnz_for_this_node += 1;

```

```

114         if(j < M-1) nnz_for_this_node += 1;
115
116         row_offsets[row] = nnz_for_this_node;
117     }
118 }
119
120
121 __global__ void populate_values(double* values, int* columns, double* row_offsets,
122 int N, int M) {
123     for(int row = blockDim.x * blockIdx.x + threadIdx.x; row < N*M;
124 row += gridDim.x * blockDim.x) {
125         int i = row / N;
126         int j = row % N;
127         int counter = 0;
128
129         if ( i > 0) {
130             values[(int)row_offsets[row] + counter] = -1;
131             columns[(int)row_offsets[row] + counter] = (i-1)*N+j;
132             counter++;
133         }
134
135         if ( j > 0) {
136             values[(int)row_offsets[row] + counter] = -1;
137             columns[(int)row_offsets[row] + counter] = i*N+(j-1);
138             counter++;
139         }
140
141         values[(int)row_offsets[row] + counter] = 4;
142         columns[(int)row_offsets[row] + counter] = i*N+j;
143
144         counter++;
145
146         if ( j < M-1) {
147             values[(int)row_offsets[row] + counter] = -1;
148             columns[(int)row_offsets[row] + counter] = i*N+(j+1);
149             counter++;
150         }
151         if ( i < N-1) {
152             values[(int)row_offsets[row] + counter] = -1;
153             columns[(int)row_offsets[row] + counter] = (i+1)*N+j;
154             counter++;
155         }
156     }
157 }
158
159
160 void exclusive_scan(double const * input,
161                    double * output, int N)
162 {
163     int num_blocks = 256;
164     int threads_per_block = 256;
165
166     double *carries;
167     cudaMalloc(&carries, sizeof(double) * num_blocks);
168
169     // First step: Scan within each thread group and write carries
170     scan_kernel_1<<<<num_blocks, threads_per_block>>>>(input, output, N, carries);

```

```

171
172 // Second step: Compute offset for each thread group (exclusive scan for
173 // each thread group)
174 scan_kernel_2<<<1, num_blocks>>>(carries);
175
176 // Third step: Offset each thread group accordingly
177 scan_kernel_3<<<num_blocks, threads_per_block>>>(output, N, carries);
178
179 cudaFree(carries);
180 }
181
182
183
184 template <typename T>
185 void printContainer(T container, int N) {
186     for (int i = 0; i < N; i++) {
187         std::cout << container[i] << " | ";
188     }
189 }
190
191
192 int main() {
193
194     int N = 3;
195     int M = 3;
196
197     //
198     // Allocate host arrays for reference
199     //
200     double *row_offsets = (double *)malloc(sizeof(double) * (N*M+1));
201
202
203     //
204     // Allocate CUDA-arrays
205     //
206     double *cuda_row_offsets;
207     double *cuda_row_offsets_2;
208     double *cuda_values;
209     int *cuda_columns;
210
211     cudaMalloc(&cuda_row_offsets, sizeof(double) * (N*M+1));
212     cudaMalloc(&cuda_row_offsets_2, sizeof(double) * (N*M+1));
213
214
215     // Perform the calculations
216     count_nnz<<<256, 256>>>(cuda_row_offsets, N, M);
217     exclusive_scan(cuda_row_offsets, cuda_row_offsets_2, N*M+1);
218     cudaMemcpy(row_offsets, cuda_row_offsets_2, sizeof(double) * (N*M+1),
219     cudaMemcpyDeviceToHost);
220
221     printContainer(row_offsets, N*M+1);
222     std::cout << std::endl;
223
224
225     int numberOfValues = (int)row_offsets[N*M];
226     double *values = (double *)malloc(sizeof(double) * numberOfValues);
227     int *columns = (int *)malloc(sizeof(int) * numberOfValues);

```



```

228     cudaMalloc(&cuda_values , sizeof(double) * numberOfValues);
229     cudaMalloc(&cuda_columns , sizeof(int) * numberOfValues);
230
231     populate_values<<<256, 256>>>(cuda_values , cuda_columns , cuda_row_offsets_2 , N, M);
232     cudaMemcpy(values , cuda_values , sizeof(double) * numberOfValues ,
233     cudaMemcpyDeviceToHost);
234     cudaMemcpy(columns , cuda_columns , sizeof(int) * numberOfValues ,
235     cudaMemcpyDeviceToHost);
236
237
238
239     printContainer(values , numberOfValues);
240     std::cout << std::endl;
241     printContainer(columns , numberOfValues);
242
243
244     //
245     // Clean up:
246     //
247     free(row_offsets);
248     free(values);
249     free(columns);
250     cudaFree(cuda_row_offsets);
251     cudaFree(cuda_row_offsets_2);
252     cudaFree(cuda_values);
253     cudaFree(cuda_columns);
254     return EXIT_SUCCESS;
255 }

```
