# 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

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.

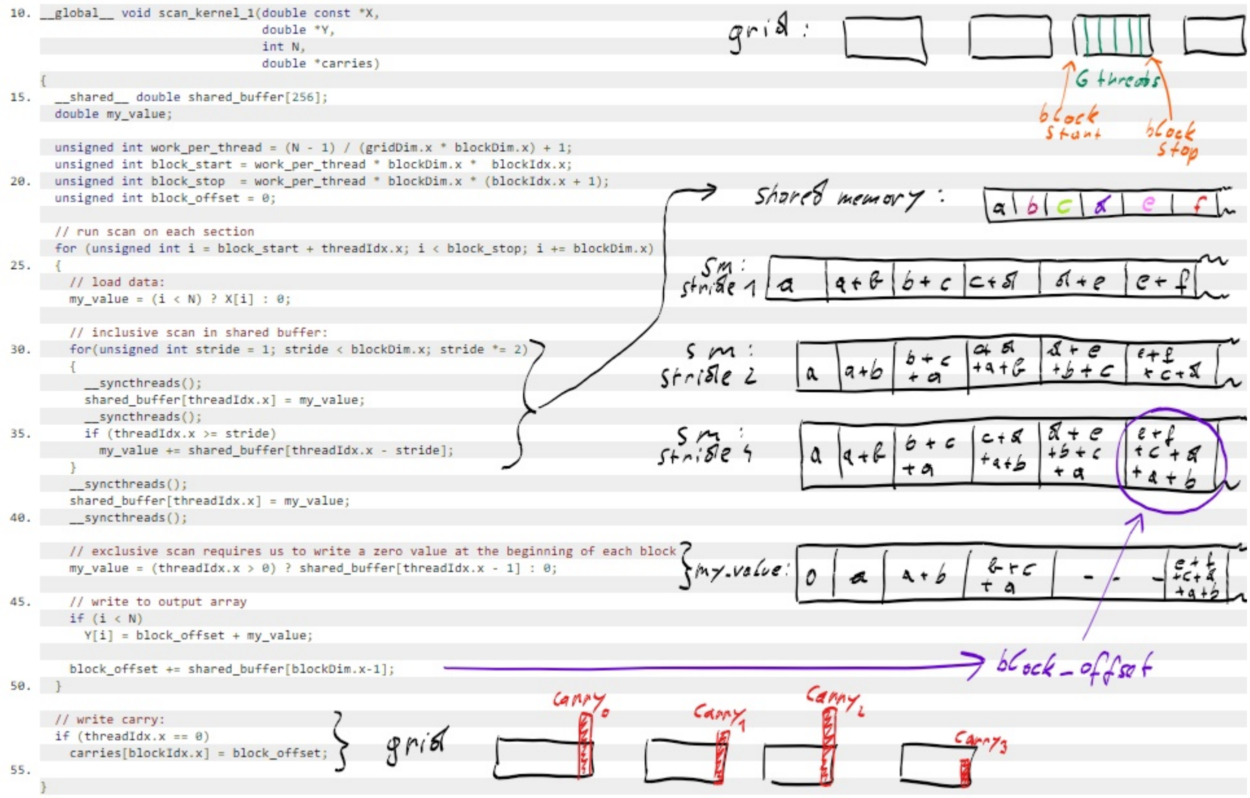



Figure 1: scan_kernel_1

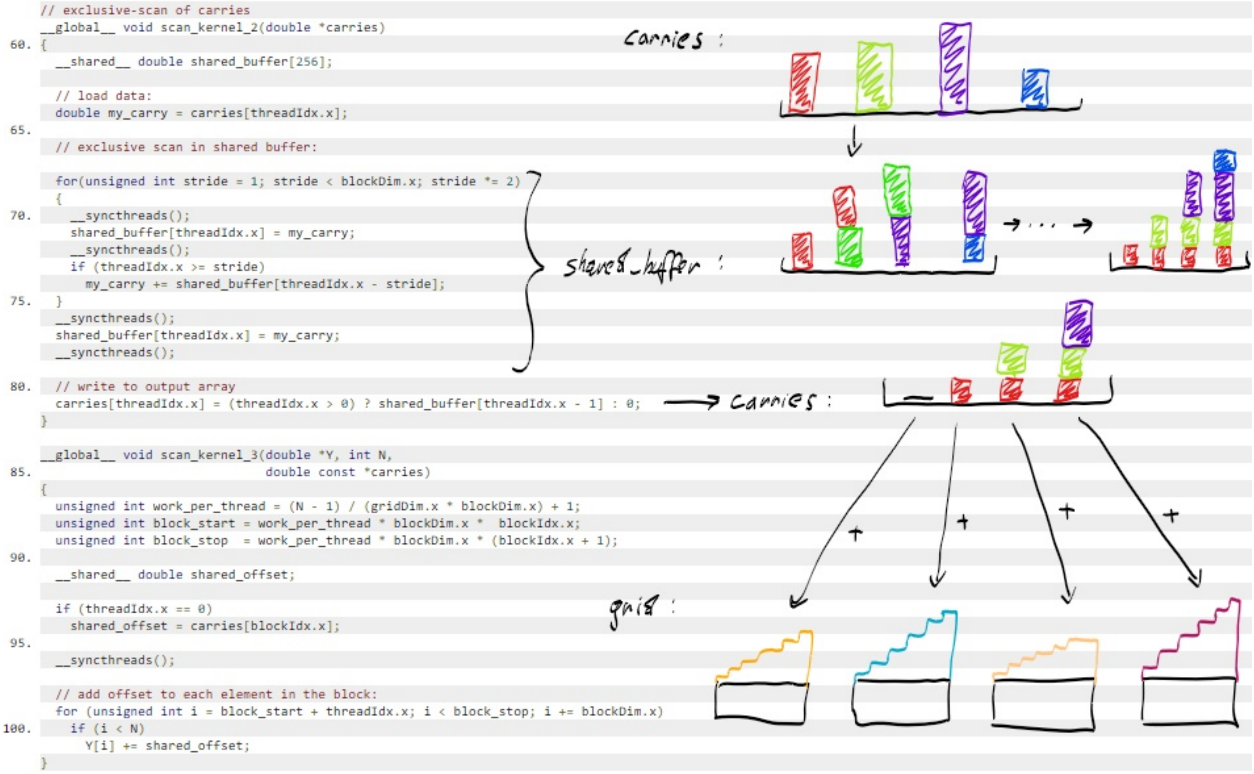



Figure 2: scan_kernel_2 and scan_kernel_3

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

Listing 1: Provided Code for exclusive scan

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




__global__ void scan_kernel_1(double const *X,
                              double *Y,
                              int N,
                              double *carries)
{
  __shared__ double shared_buffer[256];
  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;

}

// exclusive-scan of carries
__global__ void scan_kernel_2(double *carries)
{
  __shared__ double shared_buffer[256];

  // load data:
  double my_carry = carries[threadIdx.x];

  // exclusive scan in shared buffer:

  for(unsigned int stride = 1; stride < blockDim.x; stride *= 2)
  {
    __syncthreads();
    shared_buffer[threadIdx.x] = my_carry;
    __syncthreads();
    if (threadIdx.x >= stride)
      my_carry += shared_buffer[threadIdx.x - stride];
  }
  __syncthreads();
  shared_buffer[threadIdx.x] = my_carry;
  __syncthreads();

  // write to output array
  carries[threadIdx.x] = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
}
```

```

__global__ void scan_kernel_3(double *Y, int N,
                              double const *carries)
{
  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);

  __shared__ double shared_offset;

  if (threadIdx.x == 0)
    shared_offset = carries[blockIdx.x];

  __syncthreads();

  // add offset to each element in the block:
  for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
    if (i < N)
      Y[i] += shared_offset;
}




void exclusive_scan(double const * input,
                    double       * output, int N)
{
  int num_blocks = 256;
  int threads_per_block = 256;

  double *carries;
  cudaMalloc(&carries, sizeof(double) * num_blocks);

  // First step: Scan within each thread group and write carries
  scan_kernel_1<<<num_blocks, threads_per_block>>>(input, output, N, carries);

  // Second step: Compute offset for
  each thread group (exclusive scan for each thread group)
  scan_kernel_2<<<1, num_blocks>>>(carries);

  // Third step: Offset each thread group accordingly
  scan_kernel_3<<<num_blocks, threads_per_block>>>(output, N, carries);

  cudaFree(carries);
}






int main() {

  int N = 200;

  //
  // Allocate host arrays for reference
```

```
  //
  double *x = (double *)malloc(sizeof(double) * N);
  double *y = (double *)malloc(sizeof(double) * N);
  double *z = (double *)malloc(sizeof(double) * N);
  std::fill(x, x + N, 1);

  // reference calculation:
  y[0] = 0;
  for (std::size_t i=1; i<N; ++i) y[i] = y[i-1] + x[i-1];

  //
  // Allocate CUDA-arrays
  //
  double *cuda_x, *cuda_y;
  cudaMalloc(&cuda_x, sizeof(double) * N);
  cudaMalloc(&cuda_y, sizeof(double) * N);
  cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice);


  // Perform the exclusive scan and obtain results
  exclusive_scan(cuda_x, cuda_y, N);
  cudaMemcpy(z, cuda_y, sizeof(double) * N, cudaMemcpyDeviceToHost);

  //
  // Print first few entries for reference
  //
  std::cout << "CPU y: ";
  for (int i=0; i<10; ++i) std::cout << y[i] << " ";
  std::cout << " ... ";
  for (int i=N-10; i<N; ++i) std::cout << y[i] << " ";
  std::cout << std::endl;

  std::cout << "GPU y: ";
  for (int i=0; i<10; ++i) std::cout << z[i] << " ";
  std::cout << " ... ";
  for (int i=N-10; i<N; ++i) std::cout << z[i] << " ";
  std::cout << std::endl;

  //
  // Clean up:
  //
  free(x);
  free(y);
  free(z);
  cudaFree(cuda_x);
  cudaFree(cuda_y);
  return EXIT_SUCCESS;
}
```

## 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

```
 __global__ void makeInclusive(double *Y, int N, const double *X)
 {
     for (int i = blockDim.x * blockIdx.x + threadIdx.x; i < N-1;
     i += gridDim.x * blockDim.x) {
        Y[i] = Y[i+1];
    }
    if (blockDim.x * blockIdx.x + threadIdx.x == 0)
        Y[N-1] += X[N-1];
 }


void inclusive_scan(double const * input,
                    double       * output, int N)
{
  int num_blocks = 256;
  int threads_per_block = 256;

  double *carries;
  cudaMalloc(&carries, sizeof(double) * num_blocks);

  // First step: Scan within each thread group and write carries
  scan_kernel_1<<<num_blocks, threads_per_block>>>(input, output, N, carries);

  // Second step: Compute offset for each
  thread group (exclusive scan for each thread group)
  scan_kernel_2<<<1, num_blocks>>>(carries);

  // Third step: Offset each thread group accordingly
  scan_kernel_3<<<num_blocks, threads_per_block>>>(output, N, carries);

  // Make inclusive
  makeInclusive<<<num_blocks, threads_per_block>>>(output, N, input);

  cudaFree(carries);
}
```

# 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.

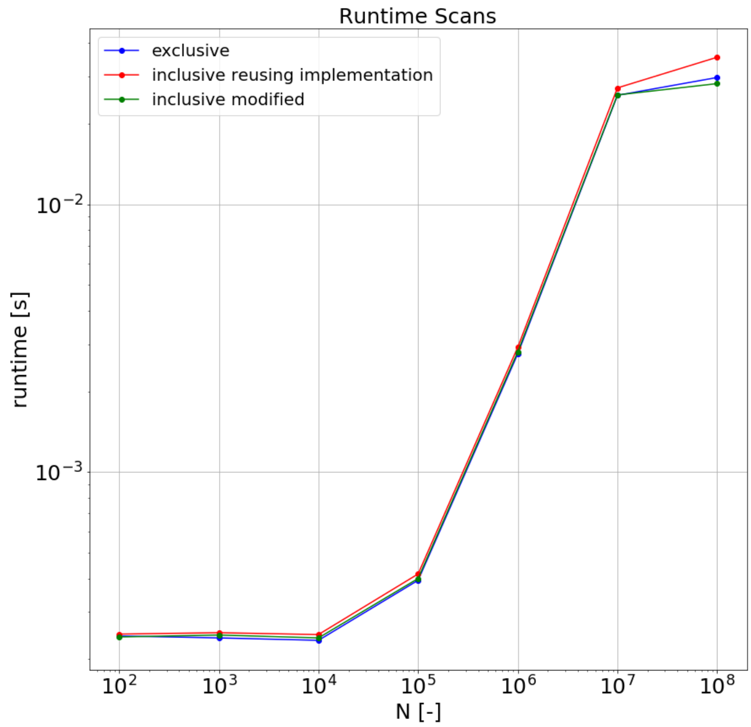



Figure 3: runtimes for different scan implementations

## 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

```
__global__ void count_nnz(double* row_offsets, int N, int M) {
    for(int row = blockDim.x * blockIdx.x + threadIdx.x;
    row < N * M; row += gridDim.x * blockDim.x) {
        int nnz_for_this_node = 1;
        int i = row / N;
        int j = row % N;

        if(i > 0) nnz_for_this_node += 1;
        if(j > 0) nnz_for_this_node += 1;
        if(i < N-1) nnz_for_this_node += 1;
        if(j < M-1) nnz_for_this_node += 1;

        row_offsets[row] = nnz_for_this_node;
    }
}
```

# 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.

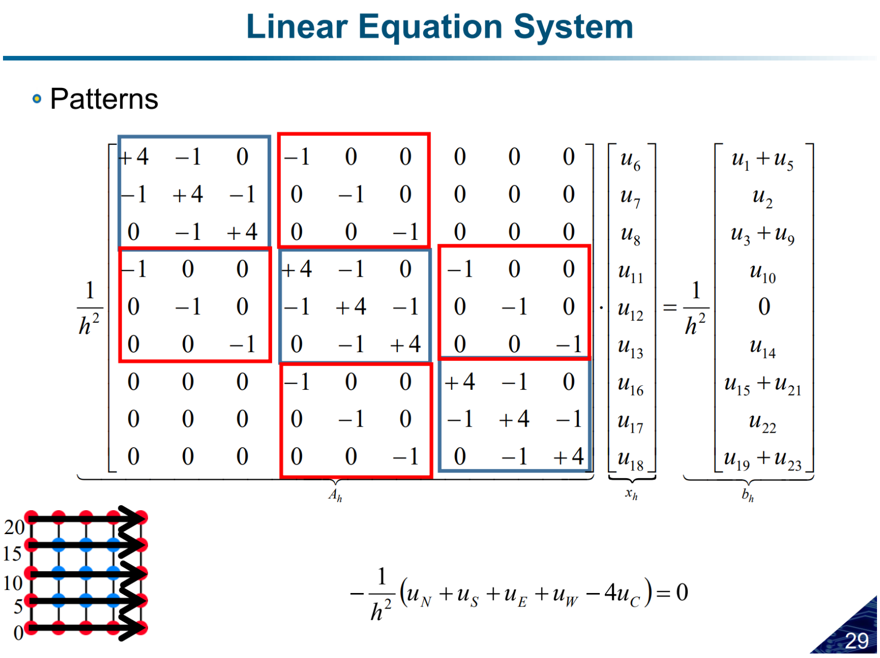
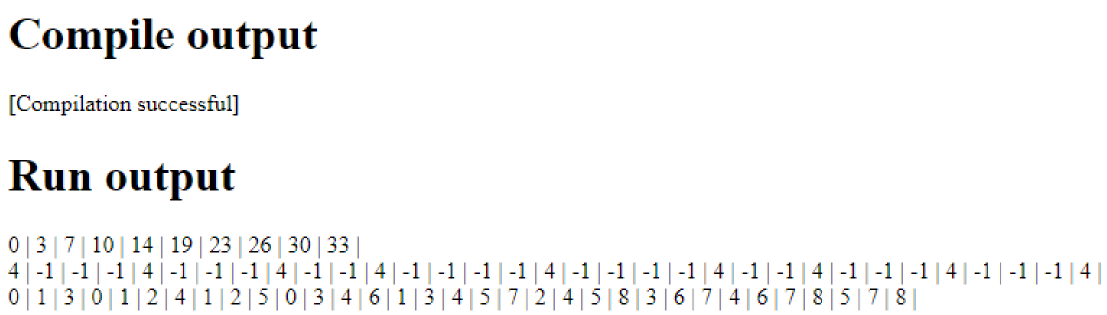



Figure 4: Test matrix, source: NSSC I (360.242), lecture 3



## 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 | 4 | -1 | -1 | -1 | -1 | 4 | -1 | -1 | -1 | -1 | 4 | -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

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




__global__ void scan_kernel_1(double const *X,
                              double *Y,
                              int N,
                              double *carries)
{
  __shared__ double shared_buffer[256];
  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;

}
```

```
// exclusive-scan of carries
__global__ void scan_kernel_2(double *carries)
{
  __shared__ double shared_buffer[256];

  // load data:
  double my_carry = carries[threadIdx.x];

  // exclusive scan in shared buffer:

  for(unsigned int stride = 1; stride < blockDim.x; stride *= 2)
  {
    __syncthreads();
    shared_buffer[threadIdx.x] = my_carry;
    __syncthreads();
    if (threadIdx.x >= stride)
      my_carry += shared_buffer[threadIdx.x - stride];
  }
  __syncthreads();
  shared_buffer[threadIdx.x] = my_carry;
  __syncthreads();

  // write to output array
  carries[threadIdx.x] = (threadIdx.x > 0) ? shared_buffer[threadIdx.x - 1] : 0;
}

__global__ void scan_kernel_3(double *Y, int N,
                              double const *carries)
{
  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);

  __shared__ double shared_offset;

  if (threadIdx.x == 0)
    shared_offset = carries[blockIdx.x];

  __syncthreads();

  // add offset to each element in the block:
  for (unsigned int i = block_start + threadIdx.x; i < block_stop; i += blockDim.x)
    if (i < N)
      Y[i] += shared_offset;
}

__global__ void count_nnz(double* row_offsets, int N, int M) {
    for(int row = blockDim.x * blockIdx.x + threadIdx.x;
    row < N * M; row += gridDim.x * blockDim.x) {
        int nnz_for_this_node = 1;
        int i = row / N;
        int j = row % N;

        if(i > 0) nnz_for_this_node += 1;
        if(j > 0) nnz_for_this_node += 1;
        if(i < N-1) nnz_for_this_node += 1;
```

```
        if(j < M-1) nnz_for_this_node += 1;

        row_offsets[row] = nnz_for_this_node;
    }
}


__global__ void populate_values(double* values, int* columns, double* row_offsets,
int N, int M) {
    for(int row = blockDim.x * blockIdx.x + threadIdx.x; row < N*M;
    row += gridDim.x * blockDim.x) {
        int i = row / N;
        int j = row % N;
        int counter = 0;

        if ( i > 0) {
            values[(int)row_offsets[row] + counter] = -1;
            columns[(int)row_offsets[row] + counter] = (i-1)*N+j;
            counter++;
        }

        if ( j > 0) {
            values[(int)row_offsets[row] + counter] = -1;
            columns[(int)row_offsets[row] + counter] = i*N+(j-1);
            counter++;
        }

        values[(int)row_offsets[row] + counter] = 4;
        columns[(int)row_offsets[row] + counter] = i*N+j;

        counter++;

        if ( j < M-1) {
            values[(int)row_offsets[row] + counter] = -1;
            columns[(int)row_offsets[row] + counter] = i*N+(j+1);
            counter++;
        }
        if ( i < N-1) {
            values[(int)row_offsets[row] + counter] = -1;
            columns[(int)row_offsets[row] + counter] = (i+1)*N+j;
            counter++;
        }
    }
}


void exclusive_scan(double const * input,
                    double       * output, int N)
{
  int num_blocks = 256;
  int threads_per_block = 256;

  double *carries;
  cudaMalloc(&carries, sizeof(double) * num_blocks);

  // First step: Scan within each thread group and write carries
  scan_kernel_1<<<num_blocks, threads_per_block>>>(input, output, N, carries);
```

```
  // Second step: Compute offset for each thread group (exclusive scan for
  // each thread group)
  scan_kernel_2<<<1, num_blocks>>>(carries);

  // Third step: Offset each thread group accordingly
  scan_kernel_3<<<num_blocks, threads_per_block>>>(output, N, carries);

  cudaFree(carries);
}



 template <typename T>
 void printContainer(T container, int N) {
     for (int i = 0; i < N; i++) {
         std::cout << container[i] << " | ";
     }
 }


int main() {

  int N = 3;
  int M = 3;

  //
  // Allocate host arrays for reference
  //
  double *row_offsets = (double *)malloc(sizeof(double) * (N*M+1));


  //
  // Allocate CUDA-arrays
  //
  double *cuda_row_offsets;
  double *cuda_row_offsets_2;
  double *cuda_values;
  int *cuda_columns;

  cudaMalloc(&cuda_row_offsets, sizeof(double) * (N*M+1));
  cudaMalloc(&cuda_row_offsets_2, sizeof(double) * (N*M+1));


  // Perform the calculations
  count_nnz<<<256, 256>>>(cuda_row_offsets, N, M);
  exclusive_scan(cuda_row_offsets, cuda_row_offsets_2, N*M+1);
  cudaMemcpy(row_offsets, cuda_row_offsets_2, sizeof(double) * (N*M+1),
  cudaMemcpyDeviceToHost);

  printContainer(row_offsets, N*M+1);
  std::cout << std::endl;


  int numberOfValues = (int)row_offsets[N*M];
  double *values = (double *)malloc(sizeof(double) * numberOfValues);
  int *columns = (int *)malloc(sizeof(int) * numberOfValues);
```

```
    cudaMalloc(&cuda_values, sizeof(double) * numberOfValues);
    cudaMalloc(&cuda_columns, sizeof(int) * numberOfValues);

    populate_values<<<256, 256>>>(cuda_values, cuda_columns, cuda_row_offsets_2, N, M);
    cudaMemcpy(values, cuda_values, sizeof(double) * numberOfValues,
    cudaMemcpyDeviceToHost);
    cudaMemcpy(columns, cuda_columns, sizeof(int) * numberOfValues,
    cudaMemcpyDeviceToHost);



    printContainer(values, numberOfValues);
    std::cout << std::endl;
    printContainer(columns, numberOfValues);


    //
    // Clean up:
    //
    free(row_offsets);
    free(values);
    free(columns);
    cudaFree(cuda_row_offsets);
    cudaFree(cuda_row_offsets_2);
    cudaFree(cuda_values);
    cudaFree(cuda_columns);
    return EXIT_SUCCESS;
}
```