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).

```
double* inclusive_scan(double const * x,
                       double const * cuda_x,
                       double       * cuda_y, int N)
{
  exclusive_scan(cuda_x, cuda_y, N);
  // create arrays to store exclusive and inclusive scan result
  double *z_ex = (double *)malloc(sizeof(double) * N);
  double *z_in = (double *)malloc(sizeof(double) * N);

  cudaMemcpy(z_ex, cuda_y, sizeof(double) * N, cudaMemcpyDeviceToHost);

  // shift by one to get inclusive scan (probably faster with a GPU kernel)
  for (int i=0; i<N; i++){
```

```
    z_in[i] = z_ex[i+1];
  }
  z_in[N-1] = z_ex[N-1] + x[N-1];
  free(z_ex);
  return z_in;
}
```

c)

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

```
    // 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;
```

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).

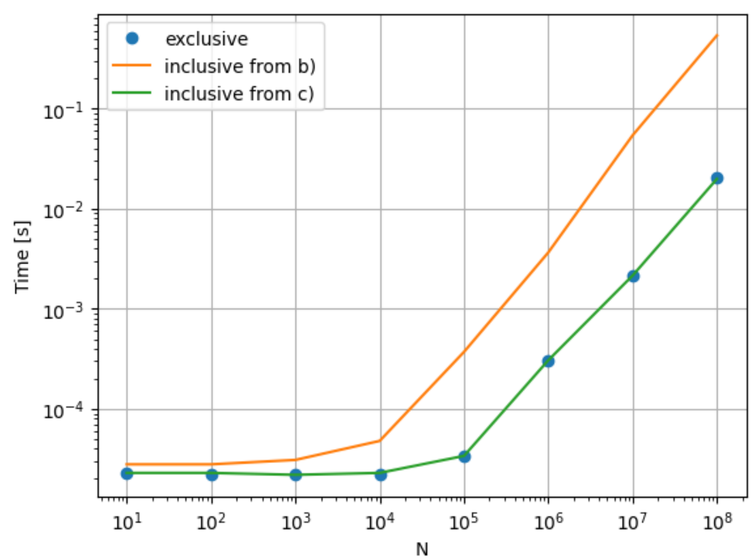

Figure 1: Comparison of execution times of exclusive and inclusive scan implementations for different N

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.

```
__global__ void count_nnz(int *row_offsets, int N)
{
  for (int row = blockDim.x * blockIdx.x + threadIdx.x;
       row < N * N;
       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 < N - 1)
      nnz_for_this_node += 1;
    row_offsets[row] = nnz_for_this_node;
  }
}
```

b)

The exclusive scan is called in the main function as:

```
  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.

```
__global__ void assembleA(int *row_offsets,
                          int *col_indices,
                          double *values,
                          int N)
{
  for (int row = blockDim.x * blockIdx.x + threadIdx.x;
       row < N * N;
       row += gridDim.x * blockDim.x)
  {
    int i = row / N;
    int j = row % N;
    int this_row_offset = row_offsets[row];
    // diagonal entry
    col_indices[this_row_offset] = i * N + j;
    values[this_row_offset] = 4;
    this_row_offset += 1;
    if (i > 0)
    { // bottom neighbor
```

```
      col_indices[this_row_offset] = (i - 1) * N + j;
      values[this_row_offset] = -1;
      this_row_offset += 1;
    }
    if (j > 0)
    {
      col_indices[this_row_offset] = i * N + (j - 1);
      values[this_row_offset] = -1;
      this_row_offset += 1;
    }
    if (i < N - 1)
    {
      col_indices[this_row_offset] = (i + 1) * N + j;
      values[this_row_offset] = -1;
      this_row_offset += 1;
    }
    if (j < N - 1)
    {
      col_indices[this_row_offset] = i * N + (j + 1);
      values[this_row_offset] = -1;
      this_row_offset += 1;
    }
  }
}
```

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:

```
void solve_system(int *cuda_csr_rowoffsets, int n)
{

  int N = n * n; // number of unknows to solve for

  std::cout << "Solving␣Ax=b␣with␣" << N << "␣unknowns." << std::endl;

  //
  // Allocate CSR arrays.
  //

  int *csr_rowoffsets = (int *)malloc(sizeof(int) * n* n);
  cudaMemcpy(csr_rowoffsets, cuda_csr_rowoffsets, sizeof(int) * n* n, cudaMemcpyDeviceToHost);
  //print_array(csr_rowoffsets, N);
  // get total number of non-zero entries
  int nnz_total = csr_rowoffsets[N-1];
  //std::cout << nnz_total << std::endl;

  int *csr_colindices = (int *)malloc(sizeof(int) * nnz_total);
  double *csr_values = (double *)malloc(sizeof(double) *nnz_total);

  int *cuda_csr_colindices; cudaMalloc(&cuda_csr_colindices, sizeof(int) * nnz_total);
  double *cuda_csr_values; cudaMalloc(&cuda_csr_values, sizeof(double) * nnz_total);


  //
  // fill CSR matrix with values
  //
```

```
  assembleA<<<16,16>>>(cuda_csr_rowoffsets,
                       cuda_csr_colindices,
                       cuda_csr_values,
                       n);


  cudaMemcpy(csr_values, cuda_csr_values, sizeof(double) * nnz_total, cudaMemcpyDeviceToHost);
  cudaMemcpy(csr_colindices, cuda_csr_colindices, sizeof(int) * nnz_total, cudaMemcpyDeviceToHost);

  //print_array(csr_values, nnz_total);
  //print_array(csr_colindices, nnz_total);


  //
  // Allocate solution vector and right hand side:
  //
  double *solution = (double *)malloc(sizeof(double) * N);
  double *rhs = (double *)malloc(sizeof(double) * N);
  std::fill(rhs, rhs + N, 1);

  //
  // Call Conjugate Gradient implementation with GPU arrays
  //
  conjugate_gradient(N, cuda_csr_rowoffsets, cuda_csr_colindices, cuda_csr_values, rhs, solution);

  //
  // Check for convergence:
  //

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

  cudaFree(cuda_csr_rowoffsets);
  cudaFree(cuda_csr_colindices);
  cudaFree(cuda_csr_values);
  free(solution);
  free(rhs);
  free(csr_rowoffsets);
  free(csr_colindices);
  free(csr_values);

}
```

Appendix

Task 1.b)

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


__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);

  // for printing intermediate results
```

```
  /*
  double *z = (double *)malloc(sizeof(double) * N);
  cudaMemcpy(z, output, sizeof(double) * N, cudaMemcpyDeviceToHost);
  printf("output:\n");
  for (int i = 0; i<N; i++){
      std::cout << z[i] << "," << std::endl;
  }
  double *carries_cpu = (double *)malloc(sizeof(double) * num_blocks);
  cudaMemcpy(carries_cpu, carries, sizeof(double) * num_blocks, cudaMemcpyDeviceToHost);
  printf("carries:\n");
  for (int i = 0; i<num_blocks; i++){
      std::cout << carries_cpu[i] << "," << std::endl;
  }*/


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

  // for printing intermediate results
  /*
  cudaMemcpy(carries_cpu, carries, sizeof(double) * num_blocks, cudaMemcpyDeviceToHost);
  printf("carries after scan 2:\n");
  for (int i = 0; i<num_blocks; i++){
      std::cout << carries_cpu[i] << "," << std::endl;
  }*/

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

  cudaFree(carries);
}

double* inclusive_scan(double const * x,
                       double const * cuda_x,
                       double       * cuda_y, int N)
{
  exclusive_scan(cuda_x, cuda_y, N);
  // create arrays to store exclusive and inclusive scan result
  double *z_ex = (double *)malloc(sizeof(double) * N);
  double *z_in = (double *)malloc(sizeof(double) * N);

  cudaMemcpy(z_ex, cuda_y, sizeof(double) * N, cudaMemcpyDeviceToHost);

  // shift by one to get inclusive scan (probably faster with a GPU kernel)
  for (int i=0; i<N; i++){
    z_in[i] = z_ex[i+1];
  }
  z_in[N-1] = z_ex[N-1] + x[N-1];
  free(z_ex);
  return z_in;
}




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=0; i<N; ++i) y[i] = y[i-1] + x[i];

  //
  // 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 INclusive scan and obtain results
  z = inclusive_scan(x, cuda_x, cuda_y, N);

  //
  // 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;
}
```

Task 2.a)b)c)d)

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

// y = A * x
__global__ void cuda_csr_matvec_product(int N, int *csr_rowoffsets,
                                         int *csr_colindices, double *csr_values,
                                         double *x, double *y)
{
```

```
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
  {
    double sum = 0;
    for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++)
    {
      sum += csr_values[k] * x[csr_colindices[k]];
    }
    y[i] = sum;
  }
}

// x <- x + alpha * y
__global__ void cuda_vecadd(int N, double *x, double *y, double alpha)
{
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
    x[i] += alpha * y[i];
}

// x <- y + alpha * x
__global__ void cuda_vecadd2(int N, double *x, double *y, double alpha)
{
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
    x[i] = y[i] + alpha * x[i];
}

// result = (x, y)
__global__ void cuda_dot_product(int N, double *x, double *y, double *result)
{
  __shared__ double shared_mem[512];

  double dot = 0;
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
  {
    dot += x[i] * y[i];
  }

  shared_mem[threadIdx.x] = dot;
  for (int k = blockDim.x / 2; k > 0; k /= 2)
  {
    __syncthreads();
    if (threadIdx.x < k)
    {
      shared_mem[threadIdx.x] += shared_mem[threadIdx.x + k];
    }
  }

  if (threadIdx.x == 0)
    atomicAdd(result, shared_mem[0]);
}

/** Implementation of the conjugate gradient algorithm.
 *
 *  The control flow is handled by the CPU.
 *  Only the individual operations (vector updates, dot products, sparse
 * matrix-vector product) are transferred to CUDA kernels.
 *
 *  The temporary arrays p, r, and Ap need to be allocated on the GPU for use
 * with CUDA. Modify as you see fit.
 */
void conjugate_gradient(int N, // number of unknows
                        int *csr_rowoffsets, int *csr_colindices,
                        double *csr_values, double *rhs, double *solution)
```

```
//, double *init_guess)   // feel free to add a nonzero initial guess as needed
{
  // initialize timer
  Timer timer;

  // clear solution vector (it may contain garbage values):
  std::fill(solution, solution + N, 0);

  // initialize work vectors:
  double alpha, beta;
  double *cuda_solution, *cuda_p, *cuda_r, *cuda_Ap, *cuda_scalar;
  cudaMalloc(&cuda_p, sizeof(double) * N);
  cudaMalloc(&cuda_r, sizeof(double) * N);
  cudaMalloc(&cuda_Ap, sizeof(double) * N);
  cudaMalloc(&cuda_solution, sizeof(double) * N);
  cudaMalloc(&cuda_scalar, sizeof(double));

  cudaMemcpy(cuda_p, rhs, sizeof(double) * N, cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_r, rhs, sizeof(double) * N, cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_solution, solution, sizeof(double) * N, cudaMemcpyHostToDevice);

  const double zero = 0;
  double residual_norm_squared = 0;
  cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
  cuda_dot_product<<<512, 512>>>(N, cuda_r, cuda_r, cuda_scalar);
  cudaMemcpy(&residual_norm_squared, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);

  double initial_residual_squared = residual_norm_squared;

  int iters = 0;
  cudaDeviceSynchronize();
  timer.reset();
  while (1)
  {

    // line 4: A*p:
    cuda_csr_matvec_product<<<512, 512>>>(N, csr_rowoffsets, csr_colindices, csr_values, cuda_p, cuda

    // lines 5,6:
    cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
    cuda_dot_product<<<512, 512>>>(N, cuda_p, cuda_Ap, cuda_scalar);
    cudaMemcpy(&alpha, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
    alpha = residual_norm_squared / alpha;

    // line 7:
    cuda_vecadd<<<512, 512>>>(N, cuda_solution, cuda_p, alpha);

    // line 8:
    cuda_vecadd<<<512, 512>>>(N, cuda_r, cuda_Ap, -alpha);

    // line 9:
    beta = residual_norm_squared;
    cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
    cuda_dot_product<<<512, 512>>>(N, cuda_r, cuda_r, cuda_scalar);
    cudaMemcpy(&residual_norm_squared, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);

    // line 10:
    if (std::sqrt(residual_norm_squared / initial_residual_squared) < 1e-6)
    {
      break;
    }

```

```
    // line 11:
    beta = residual_norm_squared / beta;

    // line 12:
    cuda_vecadd2<<<512, 512>>>(N, cuda_p, cuda_r, beta);

    if (iters > 10000)
      break; // solver didn't converge
    ++iters;
  }
  cudaMemcpy(solution, cuda_solution, sizeof(double) * N, cudaMemcpyDeviceToHost);

  cudaDeviceSynchronize();
  std::cout << "Time␣elapsed:␣" << timer.get() << "␣(" << timer.get() / iters << "␣per␣iteration)" <<

  if (iters > 10000)
    std::cout << "Conjugate␣Gradient␣did␣NOT␣converge␣within␣10000␣iterations"
              << std::endl;
  else
    std::cout << "Conjugate␣Gradient␣converged␣in␣" << iters << "␣iterations."
              << std::endl;

  cudaFree(cuda_p);
  cudaFree(cuda_r);
  cudaFree(cuda_Ap);
  cudaFree(cuda_solution);
  cudaFree(cuda_scalar);
}



// ################ MODIFIED IMLEMENTATIONS HERE ################

void print_array(int *a, int N)
{
  int counter = 0;
  std::cout << "array:\n"
            << std::endl;
  for (int i = 0; i < N; i++)
  {
    std::cout << a[i] << std::endl;
    counter++;
  }
  std::cout << "Length␣of␣array:" << counter << std::endl;
}

void print_array(double *a, int N)
{
  int counter = 0;
  std::cout << "array:\n"
            << std::endl;
  for (int i = 0; i < N; i++)
  {
    std::cout << a[i] << std::endl;
    counter++;
  }
  std::cout << "Length␣of␣array:" << counter << std::endl;
}

__global__ void count_nnz(int *row_offsets, int N)
{
  for (int row = blockDim.x * blockIdx.x + threadIdx.x;
```

```
       row < N * N;
       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 < N - 1)
      nnz_for_this_node += 1;
    row_offsets[row] = nnz_for_this_node;
  }
}

__global__ void assembleA(int *row_offsets,
                          int *col_indices,
                          double *values,
                          int N)
{
  for (int row = blockDim.x * blockIdx.x + threadIdx.x;
       row < N * N;
       row += gridDim.x * blockDim.x)
  {
    int i = row / N;
    int j = row % N;
    int this_row_offset = row_offsets[row];
    // diagonal entry
    col_indices[this_row_offset] = i * N + j;
    values[this_row_offset] = 4;
    this_row_offset += 1;
    if (i > 0)
    { // bottom neighbor
      col_indices[this_row_offset] = (i - 1) * N + j;
      values[this_row_offset] = -1;
      this_row_offset += 1;
    }
    if (j > 0)
    {
      col_indices[this_row_offset] = i * N + (j - 1);
      values[this_row_offset] = -1;
      this_row_offset += 1;
    }
    if (i < N - 1)
    {
      col_indices[this_row_offset] = (i + 1) * N + j;
      values[this_row_offset] = -1;
      this_row_offset += 1;
    }
    if (j < N - 1)
    {
      col_indices[this_row_offset] = i * N + (j + 1);
      values[this_row_offset] = -1;
      this_row_offset += 1;
    }
  }
}

/** Solve a system with `n * n` unknowns
```

```
 */
void solve_system(int *cuda_csr_rowoffsets, int n)
{

  int N = n * n; // number of unknows to solve for

  std::cout << "Solving␣Ax=b␣with␣" << N << "␣unknowns." << std::endl;

  //
  // Allocate CSR arrays.
  //

  int *csr_rowoffsets = (int *)malloc(sizeof(int) * n* n);
  cudaMemcpy(csr_rowoffsets, cuda_csr_rowoffsets, sizeof(int) * n* n, cudaMemcpyDeviceToHost);
  //print_array(csr_rowoffsets, N);
  // get total number of non-zero entries
  int nnz_total = csr_rowoffsets[N-1];
  //std::cout << nnz_total << std::endl;

  int *csr_colindices = (int *)malloc(sizeof(int) * nnz_total);
  double *csr_values = (double *)malloc(sizeof(double) *nnz_total);

  int *cuda_csr_colindices; cudaMalloc(&cuda_csr_colindices, sizeof(int) * nnz_total);
  double *cuda_csr_values; cudaMalloc(&cuda_csr_values, sizeof(double) * nnz_total);


  //
  // fill CSR matrix with values
  //
  assembleA<<<16,16>>>(cuda_csr_rowoffsets,
                       cuda_csr_colindices,
                       cuda_csr_values,
                       n);


  cudaMemcpy(csr_values, cuda_csr_values, sizeof(double) * nnz_total, cudaMemcpyDeviceToHost);
  cudaMemcpy(csr_colindices, cuda_csr_colindices, sizeof(int) * nnz_total, cudaMemcpyDeviceToHost);

  //print_array(csr_values, nnz_total);
  //print_array(csr_colindices, nnz_total);


  //
  // Allocate solution vector and right hand side:
  //
  double *solution = (double *)malloc(sizeof(double) * N);
  double *rhs = (double *)malloc(sizeof(double) * N);
  std::fill(rhs, rhs + N, 1);

  //
  // Call Conjugate Gradient implementation with GPU arrays
  //
  conjugate_gradient(N, cuda_csr_rowoffsets, cuda_csr_colindices, cuda_csr_values, rhs, solution);

  //
  // Check for convergence:
  //

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

```

```
  cudaFree(cuda_csr_rowoffsets);
  cudaFree(cuda_csr_colindices);
  cudaFree(cuda_csr_values);
  free(solution);
  free(rhs);
  free(csr_rowoffsets);
  free(csr_colindices);
  free(csr_values);

}

__global__ void scan_kernel_1(int const *X,
                              int *Y,
                              int N,
                              int *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(int *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(int *Y, int N,
                              int 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(int const *input,
                    int *output, int N)
{
  int num_blocks = 256;
  int threads_per_block = 256;

  int *carries;
  cudaMalloc(&carries, sizeof(int) * 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()
{
  size_t N = 10;

  // allocate stuff
  int *nnz = (int *)malloc(sizeof(int) * N * N);
  int *cuda_nnz;
  cudaMalloc(&cuda_nnz, sizeof(int) * N * N);

  int *row_offsets = (int *)malloc(sizeof(int) * N * N);
  int *cuda_row_offsets;
  cudaMalloc(&cuda_row_offsets, sizeof(int) * N * N);

  // count non-zero entries - task a)
  count_nnz<<<256, 256>>>(cuda_nnz, N);
  cudaMemcpy(nnz, cuda_nnz, sizeof(int) * N * N, cudaMemcpyDeviceToHost);

  // print_array(nnz, N*N);

  // perform exclusive scan - task b)
  exclusive_scan(cuda_nnz, cuda_row_offsets, N * N+1);

  //cudaMemcpy(row_offsets, cuda_row_offsets, sizeof(int) * N * N, cudaMemcpyDeviceToHost);
  // print_array(row_offsets, N * N);

  solve_system(cuda_row_offsets, N); // solves a system with N*N unknowns

  cudaFree(cuda_nnz);
  free(nnz);
  cudaFree(cuda_row_offsets);
  free(row_offsets);

  return EXIT_SUCCESS;
}
```