INSTITUTE OF MICROELECTRONICS

360.252 Computational Science on Many-Core Architectures

Exercise 4

Author:
Peter Holzner, 01426733

Submission: November 17, 2020

Contents

1 Task 1: Multiple Dot Products

1.1 Case: K is multiple of 8

Code listings for this task:

- main: Listing 3
- kernels: Listing 2

xDOTy and xDOTy8 are my kernels, where the second is simply a specialized version of the first (general vector dot product) for this task.

Outperforming the cublas library is not a difficult task as it turns out! For reference, see the plots in Figure 1. One can even do it with a simple CPU "kernel" if the given vector is small enough ($N = 1000$ elements). The CUDA overhead is simply too big here.

The remaining three cases ($N = 1e4, 1e5, 1e6$) are more interesting. For $N = 1e4$ (top right), one needs to actually pack all $1 + 8$-vectors into one kernel, instead of calling the same $1 + 1$-kernel eight times, to achieve better performance on the GPU than on the CPU. Again, the culprit is the kernel call overhead.

The main takeaway here is that - contrary to usual software development practices - writing monolothic kernels is better, at least if your goal is maximum performance. Or rather, if you want the best possible performance, use specialized kernels and reduce the amount of kernel calls as much as possible (write kernels based on the different subtasks in a program and not on generic function such as dot-products).

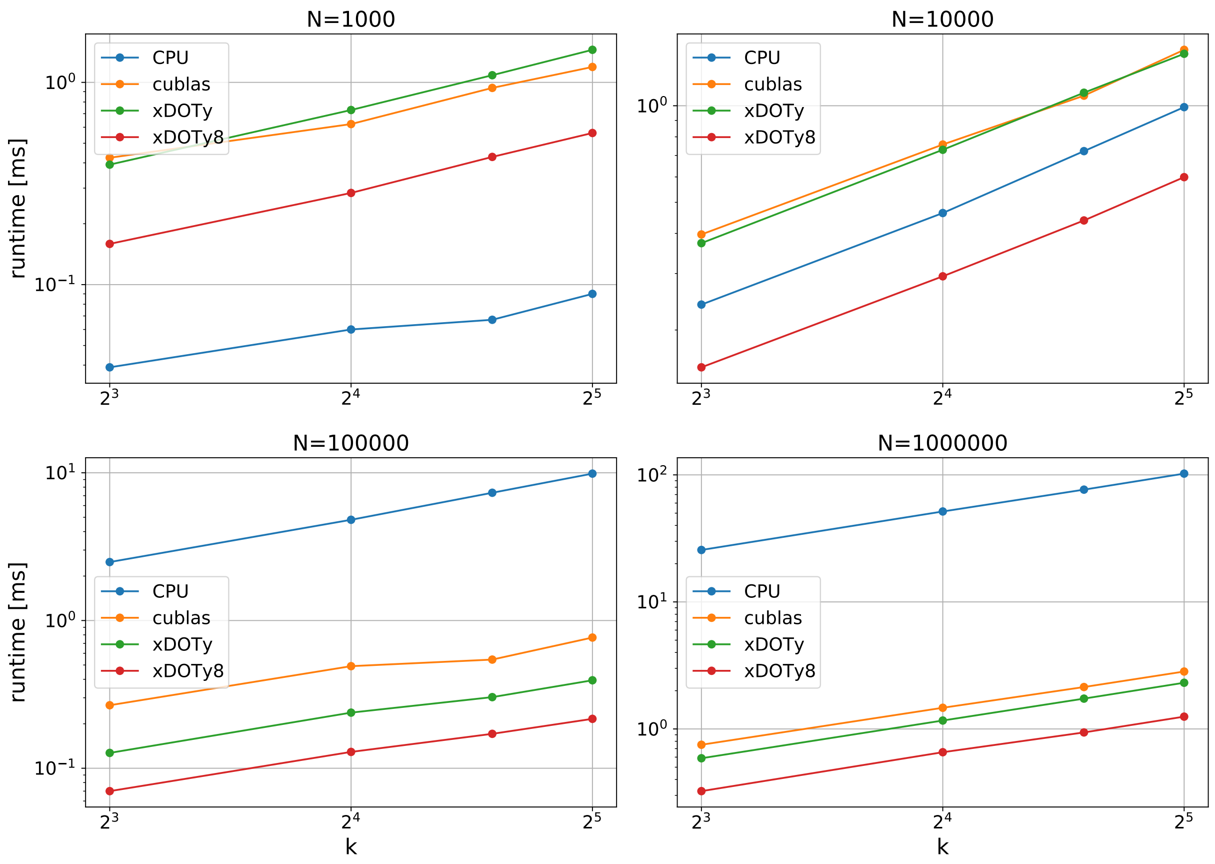

Fig. 1. LogLog plots of the kernel runtimes over the number of secondary vectors k for different vector sizes N.

1.2 Case: arbitrary (int) K

If one considers the following kernel signature:

```
__global__ void xDOTy8(const size_t N, double* x,
                       double* y1, double* y2,
                       double* y3, double* y4,
                       double* y5, double* y6,
                       double* y7, double* y8,
                       double* z)
```

Listing 1: Ex4.1: xDOTy8 call signature

then the first to generalizing the kernel becomes clear. Instead of using seperate pointers for each y_i, one instead uses either (1) a pointer of pointers $double ** \, y\_ptrs$ (a matrix where each row/column is one y_i) and specifies how many different secondary vectors y_ptrs reference or (2) passing $double * \; y$ as a flattened matrix (one big array containing every y) and also specifying how many y_i are contained (=number of dot products).

The shared cache for the partial results would then need to be either (i) dynamically allocted to allow packing all element wise operations into one for-loop (as done in xDOTy8) or (ii) again split the kernel into seperate sub groups of f.e. size 8 and allocate 8 caches. The second option "(ii)" would essentially move the for-loop we used around the $xDOTy8$-kernel call on the CPU side into the kernel. We needs to make sure that the pointer arithmetic used in the kernel is correct, which could become quite complicated in either cases.

2 Task 2: Pipelined Conjugate Gradients

Code listings for this task:

- Pipelined CG: Listing 4
- Kernels + modified CG loop (for easier reading): Listing 5
- Classical CG: Listing 6

The pipelined version is slightly faster than the classical version, as can be seen in Figure 2. I used the provided implementation of the classical version. The achieved speedup ranges from 1.05 to 1.4, depending on the number of unknowns N. My results match what we discussed in the lecture, namely that the achieved speedup (1.4) is larger for small N and smaller (1.1) for large N. The best runtime per iteration is achieved at roughly $N = 10^5$ for both version. In fact, they perform similarly from this point on. I think that this can be explained by the fact that the main benefit of the pipelined version is that less overhead is produced due to fewer kernel calls.

Unsurprisingly, the number of iterations needed did not change between the version which also is a sign that the pipelined version works correctly.

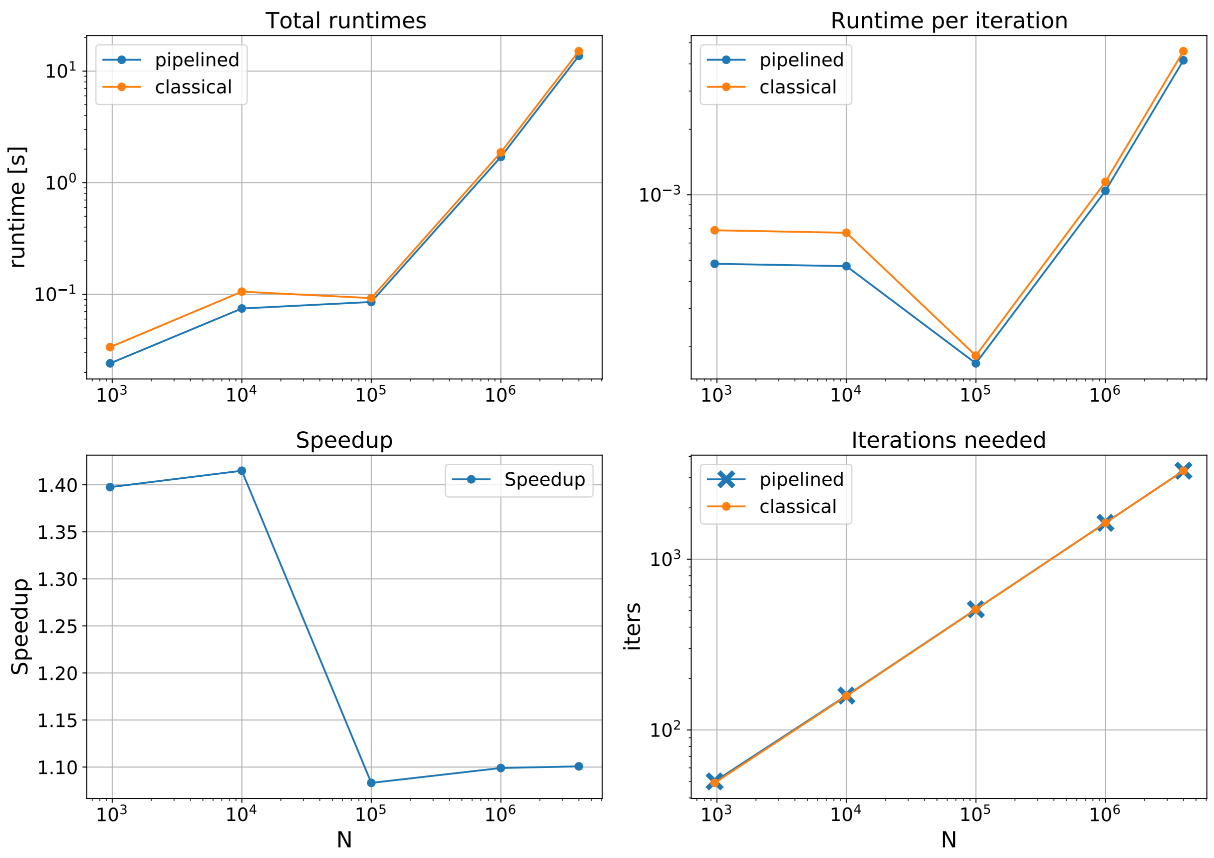

Fig. 2. Plots for runtime analysis of the different CG versions. (Grid and Block size = 256)

3 Code and Kernels

Listings

```
#pragma once
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <stdio.h>
#include <cmath>
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include "timer.hpp"

#define BLOCK_SIZE 256
#define GRID_SIZE 256

__global__ void xDOTy(const size_t N, double* x,
  double* y,
  double* z)
{
  size_t tid = threadIdx.x + blockDim.x* blockIdx.x;
  const size_t stride = blockDim.x* gridDim.x;

  __shared__ double cache[BLOCK_SIZE];

  double tid_sum = 0.0;
  for (; tid < N; tid += stride)
  {
    double tmp_x = x[tid];
    tid_sum +=  tmp_x * y[tid];
  }
  tid = threadIdx.x;
  cache[tid] = tid_sum;

  __syncthreads();
  for (size_t i = blockDim.x/2; i != 0; i /=2)
  {
    __syncthreads();
    if (tid < i) //lower half does smth, rest idles
      cache[tid] += cache[tid + i]; //lower looks up by stride and sums up
  }

  if(tid == 0) // cache[0] now contains block_sum
  {
    atomicAdd(z, cache[0]);
  }
}

/** Computes 8 vector dot products of type <x,y_i> with i=1,...,8 at once

z should be a pointer to an array of size 8 to store the results.
*/
__global__ void xDOTy8(const size_t N, double* x,
                      double* y1, double* y2,
                      double* y3, double* y4,
                      double* y5, double* y6,
                      double* y7, double* y8,
                      double* z)
{
  size_t tid = threadIdx.x + blockDim.x* blockIdx.x;
  const size_t stride = blockDim.x* gridDim.x;

  __shared__ double cache1[BLOCK_SIZE];
  __shared__ double cache2[BLOCK_SIZE];
  __shared__ double cache3[BLOCK_SIZE];
  __shared__ double cache4[BLOCK_SIZE];
  __shared__ double cache5[BLOCK_SIZE];
  __shared__ double cache6[BLOCK_SIZE];
  __shared__ double cache7[BLOCK_SIZE];
  __shared__ double cache8[BLOCK_SIZE];

  double tid_sum1 = 0.0; double tid_sum2 = 0.0;
  double tid_sum3 = 0.0; double tid_sum4 = 0.0;
  double tid_sum5 = 0.0; double tid_sum6 = 0.0;
  double tid_sum7 = 0.0; double tid_sum8 = 0.0;
  for (; tid < N; tid += stride)
  {
    double tmp_x = x[tid];
    tid_sum1 +=  tmp_x * y1[tid];
    tid_sum2 +=  tmp_x * y2[tid];
    tid_sum3 +=  tmp_x * y3[tid];
    tid_sum4 +=  tmp_x * y4[tid];
    tid_sum5 +=  tmp_x * y5[tid];
    tid_sum6 +=  tmp_x * y6[tid];
    tid_sum7 +=  tmp_x * y7[tid];
    tid_sum8 +=  tmp_x * y8[tid];
  }
  tid = threadIdx.x;
  cache1[tid] = tid_sum1;
  cache2[tid] = tid_sum2;
  cache3[tid] = tid_sum3;
  cache4[tid] = tid_sum4;
  cache5[tid] = tid_sum5;
  cache6[tid] = tid_sum6;
  cache7[tid] = tid_sum7;
  cache8[tid] = tid_sum8;

  __syncthreads();
  for (size_t i = blockDim.x/2; i != 0; i /=2)
  {
    __syncthreads();
    if (tid < i) { //lower half
      cache1[tid] += cache1[tid + i];
      cache2[tid] += cache2[tid + i];
      cache3[tid] += cache3[tid + i];
      cache4[tid] += cache4[tid + i];
    // }
    // else if (tid < i*2){
      cache5[tid] += cache5[tid + i];
      cache6[tid] += cache6[tid + i];
      cache7[tid] += cache7[tid + i];
      cache8[tid] += cache8[tid + i];
    }

  }

  if (tid==0) // cache0 now contains block_sum
  {
    atomicAdd(z, cache1[0]);
    atomicAdd(z+1, cache2[0]);
    atomicAdd(z+2, cache3[0]);
    atomicAdd(z+3, cache4[0]);
    atomicAdd(z+4, cache5[0]);
    atomicAdd(z+5, cache6[0]);
    atomicAdd(z+6, cache7[0]);
    atomicAdd(z+7, cache8[0]);
  }
}
```

Listing 2: Ex4.1: Kernels

```
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <stdio.h>
#include <cmath>
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include "timer.hpp"
#include "mdot_kernels.cuh"

int main(void)
{
    const size_t K = 32;
    Timer timer;
```

```
std::vector<double> times_CPU;
std::vector<double> times_cublas;
std::vector<double> times_xDOTy;
std::vector<double> times_xDOTy8;
std::vector<int> vec_ks;
std::vector<int> vec_Ns;

for (size_t N = 1000; N <= 1000000; N*=10) {
  //
  // Initialize CUBLAS:
  //
  std::cout << "Init CUBLAS..." << std::endl;
  cublasHandle_t h;
  cublasCreate(&h);

  //
  // allocate host memory:
  //
  std::cout << "Allocating host arrays..." << std::endl;
  double  *x = (double*)malloc(sizeof(double) * N);
  double **y = (double**)malloc(sizeof(double*) * K);
  for (size_t i=0; i<K; ++i) {
    y[i] = (double*)malloc(sizeof(double) * N);
  }
  double *results  = (double*)malloc(sizeof(double) * K);
  double *results2 = (double*)malloc(sizeof(double) * K);
  double *results_xDOTy = (double*)malloc(sizeof(double) * K);
  double *results_xDOTy8 = (double*)malloc(sizeof(double) * K);
  std::fill(results_xDOTy, results_xDOTy+K, 0.0);
  std::fill(results_xDOTy8, results_xDOTy8+K, 0.0);


  //
  // allocate device memory
  //
  std::cout << "Allocating CUDA arrays..." << std::endl;
  double *cuda_x; cudaMalloc(&cuda_x, sizeof(double)*N);
  double **cuda_y = (double**)malloc(sizeof(double*) * K);  // storing CUDA
  double *cuda_results; cudaMalloc(&cuda_results, sizeof(double)*K);
  for (size_t i=0; i<K; ++i) {
    cudaMalloc( (void **)(&cuda_y[i]), sizeof(double)*N);
  }

  //
  // fill host arrays with values
  //
  for (size_t j=0; j<N; ++j) {
    x[j] = 1 + j%K;
  }
  for (size_t i=0; i<K; ++i) {
    for (size_t j=0; j<N; ++j) {
      y[i][j] = 1 + rand() / (1.1 * RAND_MAX);
    }
    results2[i] = 0;
  }

  //
  // Copy data to GPU
  //
  std::cout << "Copying data to GPU..." << std::endl;
  cudaMemcpy(cuda_x, x, sizeof(double)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_results, results_xDOTy8, sizeof(double)*K, cudaMemcpyHostToDevice);
  for (size_t i=0; i<K; ++i) {
    cudaMemcpy(cuda_y[i], y[i], sizeof(double)*N, cudaMemcpyHostToDevice);
  }

  for (int k = 8; k <= K; k+=8) {

    //
    // Reference calculation on CPU:
    //
```

```
timer.reset();
for (size_t i=0; i<k; ++i) {
  results[i] = 0;
  for (size_t j=0; j<N; ++j) {
    results[i] += x[j] * y[i][j];
  }
}
double time_CPU = timer.get();

//
// Let CUBLAS do the work:
//
std::cout << "Running dot products with CUBLAS..." << std::endl;
timer.reset();
for (size_t i=0; i<k; ++i) {
  cublasDdot(h, N, cuda_x, 1, cuda_y[i], 1, results2 + i);
}
double time_cublas = timer.get();

//
// Let xDOTy do the work:
//
std::cout << "Running dot products with custom xDOTy8..." << std::endl;
timer.reset();
for (size_t i=0; i<k; ++i) {
  xDOTy<<<GRID_SIZE, BLOCK_SIZE>>>(N,
        cuda_x, cuda_y[i],
        cuda_results+i);
}
cudaMemcpy(results_xDOTy, cuda_results, sizeof(double)*k, cudaMemcpyDeviceToHost);
double time_xDOTy = timer.get();

//
// Let xDOTy8 do the work:
//
cudaMemcpy(cuda_results, results_xDOTy8, sizeof(double)*k, cudaMemcpyHostToDevice);
std::cout << "Running dot products with custom xDOTy8..." << std::endl;
timer.reset();
for (size_t i=0; i<(int)k/8; ++i) {
  int batch_offset = (i*8);
  xDOTy8<<<GRID_SIZE, BLOCK_SIZE>>>(N,
        cuda_x,
        cuda_y[batch_offset], cuda_y[batch_offset+1],
        cuda_y[batch_offset+2], cuda_y[batch_offset+3],
        cuda_y[batch_offset+4], cuda_y[batch_offset+5],
        cuda_y[batch_offset+6], cuda_y[batch_offset+7],
        cuda_results+batch_offset);
}
cudaMemcpy(results_xDOTy8, cuda_results, sizeof(double)*k, cudaMemcpyDeviceToHost);
double time_xDOTy8 = timer.get();

//
// Compare results
//
if (k==8) {
  std::cout << "------------------------------------------------------------" << std
      ::endl;
  std::cout << "Copying results back to host..." << std::endl;
  for (size_t i=0; i<k; ++i) {
    std::cout << results[i] << " on CPU, " << results2[i] << " on GPU. Relative
        difference: " << fabs(results[i] - results2[i]) / results[i] << std::endl;
  }std::cout << "------------------------------------------------------------" <<
      std::endl;
  std::cout << "Now to compare the custom kernel xDOTy to CPU..." << std::endl;
  for (size_t i=0; i<k; ++i) {
    std::cout << results[i] << " on CPU, " << results_xDOTy[i] << " on GPU. Relative
         difference: " << fabs(results[i] - results_xDOTy[i]) / results[i] << std::
        endl;
  }
  std::cout << "------------------------------------------------------------" << std
      ::endl;
  std::cout << "Now to compare the custom kernel xDOTy8 to CPU..." << std::endl;
```

```
      for (size_t i=0; i<k; ++i) {
        std::cout << results[i] << " on CPU, " << results_xDOTy8[i] << " on GPU.
            Relative difference: " << fabs(results[i] - results_xDOTy8[i]) / results[i]
            << std::endl;
      }
    }

    bool in_percent = false;
    auto speedup = [ref_time=time_CPU, in_percent] (double comp_time) -> double { return
         (in_percent) ? (ref_time/comp_time)*100 : ref_time/comp_time;};
    auto time_in_ms = [] (double time) -> double { return time*1e-3;};
    std::cout << "------------------------------------------------------------" << std::
        endl;
    std::cout << "And now compare the runtime of all implementations..." << std::endl;
    std::string s_unit = (in_percent) ? "%" : "";
    std::string t_unit = "ms";
    std::cout << "CPU.........." << time_in_ms(time_CPU) << t_unit << std::endl;
    std::cout << "CUBLAS..." << time_in_ms(time_cublas) << t_unit << " >> Speedup: " <<
        speedup(time_cublas) << s_unit << std::endl;
    std::cout << "xDOTy......" << time_in_ms(time_xDOTy) << t_unit << " >> Speedup: " <<
         speedup(time_xDOTy) << s_unit << std::endl;
    std::cout << "xDOTy8...." << time_in_ms(time_xDOTy8) << t_unit << " >> Speedup: " <<
         speedup(time_xDOTy8) << s_unit << std::endl;

    times_CPU.push_back(time_CPU);
    times_cublas.push_back(time_cublas);
    times_xDOTy.push_back(time_xDOTy);
    times_xDOTy8.push_back(time_xDOTy8);
    vec_ks.push_back(k);
    vec_Ns.push_back(N);
  }


  //
  // Clean up:
  //
  std::cout << "Cleaning up..." << std::endl;
  free(x);
  cudaFree(cuda_x);

  for (size_t i=0; i<K; ++i) {
    free(y[i]);
    cudaFree(cuda_y[i]);
  }
  free(y);
  free(cuda_y);

  free(results);
  free(results2);
  free(results_xDOTy8);

  cublasDestroy(h);
}

std::cout << "--------------------------CSV--------------------------------" << std::
    endl;
  std::string sep = ";";
  std::cout << "N" << sep << "k" << sep << "time_CPU" << sep << "time_cublas" << sep <<
      "time_xDOTy" << sep << "time_xDOTy8\n";
  for (int i = 0; i < vec_ks.size(); ++i ) {
    std::cout << std::setprecision("scientific") << vec_Ns[i] << sep
      << vec_ks[i] << sep
      << times_CPU[i] << sep
      << times_cublas[i] << sep
      << times_xDOTy[i] << sep
      << times_xDOTy8[i] << "\n";
  }
  std::cout << std::endl;

return 0;
```

```
}
```

Listing 3: Ex4.1: Main

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

// Block and grid size defines.
// Seperate defines are really just for future convenience...
#define BLOCK_SIZE 512
#define GRID_SIZE 512
#define SEP ";"
//#define DEBUG

// 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[BLOCK_SIZE];

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]);
}

__global__ void part1(int N,
  double* x, double* r, double *p, double *Ap,
  double alpha, double beta)
{
  // lines 2 , 3 + 4
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x) {
    x[i] = x[i] + alpha * p[i];
    double r_tmp = r[i] - alpha * Ap[i];
    r[i] = r_tmp;
  //}
  // Merge these two?
  //for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
      {
    p[i] = r_tmp + beta * p[i];
  }
}

__global__ void part2(int N,
  int *csr_rowoffsets, int *csr_colindices, double *csr_values,
  double* r, double *p, double *Ap,
  double* ApAp, double* pAp, double* rr
  )
{
  __shared__ double shared_mem_ApAp[BLOCK_SIZE];
  __shared__ double shared_mem_pAp[BLOCK_SIZE];
  __shared__ double shared_mem_rr[BLOCK_SIZE];
  // Mat-vec product
  double dot_ApAp = 0., dot_pAp = 0., dot_rr = 0.;
  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] * p[csr_colindices[k]];
    }
    Ap[i] = sum;
    dot_ApAp += sum*sum;
    dot_pAp += p[i]*sum;
    dot_rr += r[i]*r[i];
  }
  // now :
  // Ap = Ap_i --> Line 5
  // we are ready for reductions

  shared_mem_ApAp[threadIdx.x] = dot_ApAp;
  shared_mem_pAp[threadIdx.x] = dot_pAp;
  shared_mem_rr[threadIdx.x]  = dot_rr;
  for (int k = blockDim.x / 2; k > 0; k /= 2) {
    __syncthreads();
    if (threadIdx.x < k) {
      shared_mem_ApAp[threadIdx.x] += shared_mem_ApAp[threadIdx.x + k];
      shared_mem_pAp[threadIdx.x] += shared_mem_pAp[threadIdx.x + k];
      shared_mem_rr[threadIdx.x] += shared_mem_rr[threadIdx.x + k];
    }
  }

  if (threadIdx.x == 0) {
    atomicAdd(ApAp, shared_mem_ApAp[0]);
    atomicAdd(pAp, shared_mem_pAp[0]);
    atomicAdd(rr, shared_mem_rr[0]);
  }
  // now:
  // ApAp, pAp, rr --> Line 6
}



/** 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.
 */
int 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, pAp, ApAp, rr;
  double* cuda_pAp, *cuda_ApAp, *cuda_rr;
  double* cuda_x, *cuda_p, *cuda_r, *cuda_Ap;
  cudaMalloc(&cuda_p, sizeof(double) * N);
  cudaMalloc(&cuda_r, sizeof(double) * N);
  cudaMalloc(&cuda_Ap, sizeof(double) * N);
  cudaMalloc(&cuda_x, sizeof(double) * N);
  cudaMalloc(&cuda_pAp, sizeof(double));
  cudaMalloc(&cuda_ApAp, sizeof(double));
  cudaMalloc(&cuda_rr, sizeof(double));

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

  const double zero = 0;
  cudaMemcpy(cuda_pAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_ApAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_rr, &zero, sizeof(double), cudaMemcpyHostToDevice);

  // Initial values: i = 0
  // device
  cuda_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_r, cuda_r, cuda_rr);
  cuda_csr_matvec_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, csr_rowoffsets, csr_colindices,
      csr_values, cuda_p, cuda_Ap);
  cuda_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_p, cuda_Ap, cuda_pAp);
  cuda_dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_Ap, cuda_Ap, cuda_ApAp);
  cudaMemcpy(&rr, cuda_rr, sizeof(double), cudaMemcpyDeviceToHost);
  cudaMemcpy(&pAp, cuda_pAp, sizeof(double), cudaMemcpyDeviceToHost);
  cudaMemcpy(&ApAp, cuda_ApAp, sizeof(double), cudaMemcpyDeviceToHost);
  cudaDeviceSynchronize();

  // host side of things
  double initial_residual_squared = rr;

#ifdef DEBUG
  std::cout << "Initial residual norm: " << initial_residual_squared << std::endl;
#endif
  alpha = rr / pAp;
  //beta = (alpha*alpha * ApAp - rr) / rr;
  beta = alpha * alpha * ApAp / rr - 1;

  int iters = 1;
  cudaDeviceSynchronize();
  timer.reset();
  while (1) {
    part1<<<BLOCK_SIZE, GRID_SIZE>>>(N,
      cuda_x, cuda_r, cuda_p, cuda_Ap,
      alpha, beta);

    cudaMemcpy(cuda_pAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_ApAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_rr, &zero, sizeof(double), cudaMemcpyHostToDevice);
    part2<<<BLOCK_SIZE, GRID_SIZE>>>(N,
      csr_rowoffsets, csr_colindices, csr_values,
      cuda_r, cuda_p, cuda_Ap,
      cuda_ApAp, cuda_pAp, cuda_rr);

    cudaDeviceSynchronize();
    cudaMemcpy(&rr, cuda_rr, sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(&pAp, cuda_pAp, sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(&ApAp, cuda_ApAp, sizeof(double), cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();
    // line 10:
    double rel_norm = std::sqrt(rr / initial_residual_squared);
    if (rel_norm < 1e-6) {
      break;
    }
    alpha = rr / pAp;
    //beta = (alpha*alpha * ApAp - rr) / rr;
    beta = alpha * alpha * ApAp / rr - 1;

#ifdef DEBUG
    if (iters%100==0) {
      std::cout << "Norm after " << iters << " iterations:\n"
        << "rel. norm: " << rel_norm << "\n"
        << "abs. norm: " << std::sqrt(beta) << std::endl;
    }
#endif
    if (iters > 10000)
      break; // solver didn't converge
    ++iters;
  }
  cudaMemcpy(solution, cuda_x, sizeof(double) * N, cudaMemcpyDeviceToHost);

  cudaDeviceSynchronize();
#ifdef DEBUG
  std::cout << "Time elapsed: " << timer.get() << " (" << timer.get() / iters << " per
      iteration)" << std::endl;

  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;
#endif
  // Vectors
  cudaFree(cuda_x);
  cudaFree(cuda_p);
  cudaFree(cuda_r);
  cudaFree(cuda_Ap);
  // Scalers
  cudaFree(cuda_pAp);
  cudaFree(cuda_ApAp);
  cudaFree(cuda_rr);
  return iters;
}

/** Solve a system with `points_per_direction * points_per_direction` unknowns
 */
void solve_system(int points_per_direction) {

  Timer timer;
  int N = points_per_direction *
          points_per_direction; // number of unknows to solve for
#ifdef DEBUG
  std::cout << "Solving Ax=b with " << N << " unknowns." << std::endl;
#endif
  //
  // Allocate CSR arrays.
  //
  // Note: Usually one does not know the number of nonzeros in the system matrix
  // a-priori.
  //       For this exercise, however, we know that there are at most 5 nonzeros
  //       per row in the system matrix, so we can allocate accordingly.
  //
  int *csr_rowoffsets = (int *)malloc(sizeof(double) * (N + 1));
  int *csr_colindices = (int *)malloc(sizeof(double) * 5 * N);
  double *csr_values = (double *)malloc(sizeof(double) * 5 * N);

  int *cuda_csr_rowoffsets, *cuda_csr_colindices;
  double *cuda_csr_values;
  //
  // fill CSR matrix with values
  //
  generate_fdm_laplace(points_per_direction, csr_rowoffsets, csr_colindices,
                       csr_values);

  //
  // 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);

  //
  // Allocate CUDA-arrays //
  //
  cudaMalloc(&cuda_csr_rowoffsets, sizeof(double) * (N + 1));
  cudaMalloc(&cuda_csr_colindices, sizeof(double) * 5 * N);
  cudaMalloc(&cuda_csr_values, sizeof(double) * 5 * N);
  cudaMemcpy(cuda_csr_rowoffsets, csr_rowoffsets, sizeof(double) * (N + 1),
      cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_csr_colindices, csr_colindices, sizeof(double) * 5 * N,
      cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_csr_values,     csr_values,     sizeof(double) * 5 * N,
      cudaMemcpyHostToDevice);

  //
  // Call Conjugate Gradient implementation with GPU arrays
  //
  timer.reset();
  int iters = conjugate_gradient(N, cuda_csr_rowoffsets, cuda_csr_colindices,
      cuda_csr_values, rhs, solution);
  double runtime = timer.get();

    //
  // Check for convergence:
  //
  double residual_norm = relative_residual(N, csr_rowoffsets, csr_colindices, csr_values,
      rhs, solution);

#ifdef DEBUG
  std::cout << "Time elapsed: " << runtime << " (" << runtime) / iters << " per iteration)"
      << std::endl;
  std::cout << "Relative residual norm: " << residual_norm
          << " (should be smaller than 1e-6)" << std::endl;
#endif
#ifndef DEBUG
  std::cout << points_per_direction << SEP
    << N << SEP
    << runtime << SEP
    << iters << SEP
    << residual_norm << std::endl;
#endif
  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);
}

int main() {

  std::vector<size_t> p_per_dir{ (size_t)sqrt(1e3), (size_t)sqrt(1e4), (size_t)sqrt(1e5), (
      size_t)sqrt(1e6), (size_t)sqrt(4e6)};
  std::cout << "p" << SEP "N" << SEP
    << "time" << SEP << "iters" << SEP<< "norm_after" << std::endl;
  for (auto& points: p_per_dir)
    solve_system(points); // solves a system with 100*100 unknowns

  return EXIT_SUCCESS;
```

```
}
```

Listing 4: Ex4.2: Pipelined version of CG.

```
__global__ void part1(int N,
    double* x, double* r, double *p, double *Ap,
    double alpha, double beta)
  {
    // lines 2 , 3 + 4
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
        {
      x[i] = x[i] + alpha * p[i];
      double r_tmp = r[i] - alpha * Ap[i];
      r[i] = r_tmp;
    //}
    // Merge these two?
    //for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x
        ) {
      p[i] = r_tmp + beta * p[i];
    }
  }

  __global__ void part2(int N,
    int *csr_rowoffsets, int *csr_colindices, double *csr_values,
    double* r, double *p, double *Ap,
    double* ApAp, double* pAp, double* rr
    )
  {
    __shared__ double shared_mem_ApAp[BLOCK_SIZE];
    __shared__ double shared_mem_pAp[BLOCK_SIZE];
    __shared__ double shared_mem_rr[BLOCK_SIZE];
    // Mat-vec product
    double dot_ApAp = 0., dot_pAp = 0., dot_rr = 0.;
    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] * p[csr_colindices[k]];
      }
      Ap[i] = sum;
      dot_ApAp += sum*sum;
      dot_pAp += p[i]*sum;
      dot_rr += r[i]*r[i];
    }
    // now :
    // Ap = Ap_i --> Line 5
    // we are ready for reductions

    shared_mem_ApAp[threadIdx.x] = dot_ApAp;
    shared_mem_pAp[threadIdx.x] = dot_pAp;
    shared_mem_rr[threadIdx.x]  = dot_rr;
    for (int k = blockDim.x / 2; k > 0; k /= 2) {
      __syncthreads();
      if (threadIdx.x < k) {
        shared_mem_ApAp[threadIdx.x] += shared_mem_ApAp[threadIdx.x + k];
        shared_mem_pAp[threadIdx.x] += shared_mem_pAp[threadIdx.x + k];
        shared_mem_rr[threadIdx.x] += shared_mem_rr[threadIdx.x + k];
      }
    }

    if (threadIdx.x == 0) {
      atomicAdd(ApAp, shared_mem_ApAp[0]);
      atomicAdd(pAp, shared_mem_pAp[0]);
      atomicAdd(rr, shared_mem_rr[0]);
    }
    // now:
    // ApAp, pAp, rr --> Line 6
  }

  int cg(void)
  {
    ...

    int iters = 1;
  cudaDeviceSynchronize();
  timer.reset();
  while (1) {
    part1<<<BLOCK_SIZE, GRID_SIZE>>>(N,
      cuda_x, cuda_r, cuda_p, cuda_Ap,
      alpha, beta);

    cudaMemcpy(cuda_pAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_ApAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_rr, &zero, sizeof(double), cudaMemcpyHostToDevice);
    part2<<<BLOCK_SIZE, GRID_SIZE>>>(N,
      csr_rowoffsets, csr_colindices, csr_values,
      cuda_r, cuda_p, cuda_Ap,
      cuda_ApAp, cuda_pAp, cuda_rr);

    cudaDeviceSynchronize();
    cudaMemcpy(&rr, cuda_rr, sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(&pAp, cuda_pAp, sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(&ApAp, cuda_ApAp, sizeof(double), cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();
    // line 10:
    double rel_norm = std::sqrt(rr / initial_residual_squared);
    if (rel_norm < 1e-6) {
      break;
    }
    alpha = rr / pAp;
    //beta = (alpha*alpha * ApAp - rr) / rr;
    beta = alpha * alpha * ApAp / rr - 1;

#ifdef DEBUG
    if (iters%100==0) {
      std::cout << "Norm after " << iters << " iterations:\n"
        << "rel. norm: " << rel_norm << "\n"
        << "abs. norm: " << std::sqrt(beta) << std::endl;
    }
#endif
    if (iters > 10000)
      break; // solver didn't converge
    ++iters;
  }

  ....


  }
```

Listing 5: Ex4.2: Kernels and modified CG loop used for the pipelined version.

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

// Block and grid size defines.
// Seperate defines are really just for future convenience...
#define BLOCK_SIZE 512
#define GRID_SIZE 512
#define SEP ";"
//#define DEBUG

// 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[BLOCK_SIZE];

  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.
 */
int 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<<<GRID_SIZE, BLOCK_SIZE>>>(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<<<GRID_SIZE, BLOCK_SIZE>>>(N, csr_rowoffsets, csr_colindices,
        csr_values, cuda_p, cuda_Ap);

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

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

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

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

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

  cudaDeviceSynchronize();
#ifdef DEBUG
  std::cout << "Time elapsed: " << timer.get() << " (" << timer.get() / iters << " per
      iteration)" << std::endl;

  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;
#endif
  cudaFree(cuda_p);
  cudaFree(cuda_r);
  cudaFree(cuda_Ap);
  cudaFree(cuda_solution);
  cudaFree(cuda_scalar);

  return iters;
}

/** Solve a system with `points_per_direction * points_per_direction` unknowns
 */
void solve_system(int points_per_direction) {

  Timer timer;
  int N = points_per_direction *
          points_per_direction; // number of unknows to solve for
#ifdef DEBUG
  std::cout << "Solving Ax=b with " << N << " unknowns." << std::endl;
#endif
  //
  // Allocate CSR arrays.
  //
  // Note: Usually one does not know the number of nonzeros in the system matrix
  // a-priori.
  //       For this exercise, however, we know that there are at most 5 nonzeros
  //       per row in the system matrix, so we can allocate accordingly.
  //
  int *csr_rowoffsets = (int *)malloc(sizeof(double) * (N + 1));
  int *csr_colindices = (int *)malloc(sizeof(double) * 5 * N);
  double *csr_values = (double *)malloc(sizeof(double) * 5 * N);

  int *cuda_csr_rowoffsets, *cuda_csr_colindices;
  double *cuda_csr_values;
  //
  // fill CSR matrix with values
  //
  generate_fdm_laplace(points_per_direction, csr_rowoffsets, csr_colindices,
                       csr_values);

  //
  // 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);

  //
  // Allocate CUDA-arrays //
  //
  cudaMalloc(&cuda_csr_rowoffsets, sizeof(double) * (N + 1));
  cudaMalloc(&cuda_csr_colindices, sizeof(double) * 5 * N);
  cudaMalloc(&cuda_csr_values, sizeof(double) * 5 * N);
  cudaMemcpy(cuda_csr_rowoffsets, csr_rowoffsets, sizeof(double) * (N + 1),
      cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_csr_colindices, csr_colindices, sizeof(double) * 5 * N,
      cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_csr_values,     csr_values,     sizeof(double) * 5 * N,
      cudaMemcpyHostToDevice);

  //
  // Call Conjugate Gradient implementation with GPU arrays
  //
  timer.reset();
  int iters = conjugate_gradient(N, cuda_csr_rowoffsets, cuda_csr_colindices,
      cuda_csr_values, rhs, solution);
  double runtime = timer.get();

  //
  // Check for convergence:
  //
  double residual_norm = relative_residual(N, csr_rowoffsets, csr_colindices, csr_values,
      rhs, solution);
#ifdef DEBUG
  std::cout << "Relative residual norm: " << residual_norm
```

```
            << " (should be smaller than 1e-6)" << std::endl;
#endif
#ifndef DEBUG
  std::cout << points_per_direction << SEP
    << N << SEP
    << runtime << SEP
    << iters << SEP
    << residual_norm << std::endl;
#endif
  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);
}

int main() {

  std::vector<size_t> p_per_dir{ (size_t)sqrt(1e3), (size_t)sqrt(1e4), (size_t)sqrt(1e5), (
      size_t)sqrt(1e6), (size_t)sqrt(4e6)};
#ifndef DEBUG
  std::cout << "p" << SEP "N" << SEP << "time" << SEP << "iters" << SEP << "norm_after" <<
      std::endl;
#endif
  for (auto& points: p_per_dir)
    solve_system(points); // solves a system with 100*100 unknowns

  return EXIT_SUCCESS;
}
```

Listing 6: Ex4.2: Classical version of CG.