360.252 - Computational Science on Many-Core Architectures

WS 2020 - Exercise 4

Christian Gollmann, 01435044

Last update: November 15, 2020

Contents

1 Multiple Dot Products 1

The following kernel computes 8 dot products simultaneously. I first wanted to go for some kind of 2D-array which I pass to the kernel but somehow I wasn't able to make that work, I always had segmentation failures. So I decided to go with this rather clumsy implementation which gets the job done as well in this case. However, I will reflect more on what one could to make this more efficient at the end of this exercise point.

Listing 1: Kernel to compute 8 dot products simultaneously

```
__global__ void cuda_many_dot_product(int N, double *x, double *y0, double *y1,
double *y2, double *y3, double *y4, double *y5, double *y6, double *y7,
double *result)
{
  __shared__ double shared_mem_0[512];
  __shared__ double shared_mem_1[512];
  __shared__ double shared_mem_2[512];
  __shared__ double shared_mem_3[512];
  __shared__ double shared_mem_4[512];
  __shared__ double shared_mem_5[512];
  __shared__ double shared_mem_6[512];
  __shared__ double shared_mem_7[512];

  double dot_0 = 0;
  double dot_1 = 0;
  double dot_2 = 0;
  double dot_3 = 0;
  double dot_4 = 0;
  double dot_5 = 0;
  double dot_6 = 0;
  double dot_7 = 0;

  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
  i < N; i += blockDim.x * gridDim.x) {
    double val = x[i];
    dot_0 += val * y0[i];
    dot_1 += val * y1[i];;
    dot_2 += val * y2[i];;
    dot_3 += val * y3[i];;
    dot_4 += val * y4[i];;
    dot_5 += val * y5[i];;
    dot_6 += val * y6[i];;
    dot_7 += val * y7[i];;
  }

  shared_mem_0[threadIdx.x] = dot_0;
  shared_mem_1[threadIdx.x] = dot_1;
  shared_mem_2[threadIdx.x] = dot_2;
  shared_mem_3[threadIdx.x] = dot_3;
  shared_mem_4[threadIdx.x] = dot_4;
  shared_mem_5[threadIdx.x] = dot_5;
  shared_mem_6[threadIdx.x] = dot_6;
  shared_mem_7[threadIdx.x] = dot_7;

  for (int k = blockDim.x / 2; k > 0; k /= 2) {
    __syncthreads();
```

```
    if (threadIdx.x < k) {
      shared_mem_0[threadIdx.x] += shared_mem_0[threadIdx.x + k];
      shared_mem_1[threadIdx.x] += shared_mem_1[threadIdx.x + k];
      shared_mem_2[threadIdx.x] += shared_mem_2[threadIdx.x + k];
      shared_mem_3[threadIdx.x] += shared_mem_3[threadIdx.x + k];
      shared_mem_4[threadIdx.x] += shared_mem_4[threadIdx.x + k];
      shared_mem_5[threadIdx.x] += shared_mem_5[threadIdx.x + k];
      shared_mem_6[threadIdx.x] += shared_mem_6[threadIdx.x + k];
      shared_mem_7[threadIdx.x] += shared_mem_7[threadIdx.x + k];
    }
  }

  if (threadIdx.x == 0){
       atomicAdd(result+0, shared_mem_0[0]);
       atomicAdd(result+1, shared_mem_1[0]);
       atomicAdd(result+2, shared_mem_2[0]);
       atomicAdd(result+3, shared_mem_3[0]);
       atomicAdd(result+4, shared_mem_4[0]);
       atomicAdd(result+5, shared_mem_5[0]);
       atomicAdd(result+6, shared_mem_6[0]);
       atomicAdd(result+7, shared_mem_7[0]);
  }
}
```

2 Multiple Dot Products 2 + 3

It can be seen that the runtimes are very, let's call it regular, and all follow the same trend. (K=32) takes approximately four times as long as (K=8) which I kind of expected. Also do they grow as N grows, so no really big surprises here.

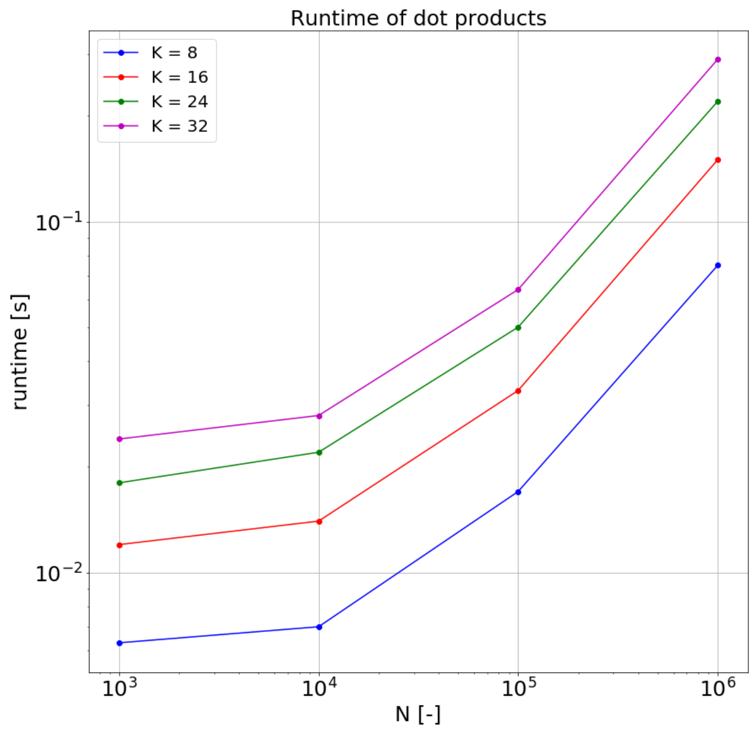

Figure 1: Runtime for dot product of vectors with N entries

Listing 2: Code for points 2 and 3

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

 __global__ void cuda_many_dot_product(int N, double *x, double *y0, double *y1,
 double *y2, double *y3, double *y4, double *y5, double *y6,
 double *y7, double *result)
{
  __shared__ double shared_mem_0[512];
  __shared__ double shared_mem_1[512];
  __shared__ double shared_mem_2[512];
  __shared__ double shared_mem_3[512];
  __shared__ double shared_mem_4[512];
  __shared__ double shared_mem_5[512];
  __shared__ double shared_mem_6[512];
  __shared__ double shared_mem_7[512];

  double dot_0 = 0;
```

```
  double dot_1 = 0;
  double dot_2 = 0;
  double dot_3 = 0;
  double dot_4 = 0;
  double dot_5 = 0;
  double dot_6 = 0;
  double dot_7 = 0;

  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N;
  i += blockDim.x * gridDim.x) {
    double val = x[i];
    dot_0 += val * y0[i];
    dot_1 += val * y1[i];;
    dot_2 += val * y2[i];;
    dot_3 += val * y3[i];;
    dot_4 += val * y4[i];;
    dot_5 += val * y5[i];;
    dot_6 += val * y6[i];;
    dot_7 += val * y7[i];;
  }

  shared_mem_0[threadIdx.x] = dot_0;
  shared_mem_1[threadIdx.x] = dot_1;
  shared_mem_2[threadIdx.x] = dot_2;
  shared_mem_3[threadIdx.x] = dot_3;
  shared_mem_4[threadIdx.x] = dot_4;
  shared_mem_5[threadIdx.x] = dot_5;
  shared_mem_6[threadIdx.x] = dot_6;
  shared_mem_7[threadIdx.x] = dot_7;

  for (int k = blockDim.x / 2; k > 0; k /= 2) {
    __syncthreads();
    if (threadIdx.x < k) {
      shared_mem_0[threadIdx.x] += shared_mem_0[threadIdx.x + k];
      shared_mem_1[threadIdx.x] += shared_mem_1[threadIdx.x + k];
      shared_mem_2[threadIdx.x] += shared_mem_2[threadIdx.x + k];
      shared_mem_3[threadIdx.x] += shared_mem_3[threadIdx.x + k];
      shared_mem_4[threadIdx.x] += shared_mem_4[threadIdx.x + k];
      shared_mem_5[threadIdx.x] += shared_mem_5[threadIdx.x + k];
      shared_mem_6[threadIdx.x] += shared_mem_6[threadIdx.x + k];
      shared_mem_7[threadIdx.x] += shared_mem_7[threadIdx.x + k];
    }
  }

  if (threadIdx.x == 0){
      atomicAdd(result+0, shared_mem_0[0]);
      atomicAdd(result+1, shared_mem_1[0]);
      atomicAdd(result+2, shared_mem_2[0]);
      atomicAdd(result+3, shared_mem_3[0]);
      atomicAdd(result+4, shared_mem_4[0]);
      atomicAdd(result+5, shared_mem_5[0]);
      atomicAdd(result+6, shared_mem_6[0]);
      atomicAdd(result+7, shared_mem_7[0]);
  }
}

int main(void)
```

```
{

  Timer timer;
  const size_t N = 100000;
  const size_t K = 16;


  //
  // 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) * 8);


  //
  // allocate device memory
  //
  std::cout << "Allocating CUDA arrays..." << std::endl;
  double *cuda_x; cudaMalloc( (void **)(&cuda_x), sizeof(double)*N);
  double *cuda_y0; cudaMalloc( (void **)(&cuda_y0), sizeof(double)*N);
  double *cuda_y1; cudaMalloc( (void **)(&cuda_y1), sizeof(double)*N);
  double *cuda_y2; cudaMalloc( (void **)(&cuda_y2), sizeof(double)*N);
  double *cuda_y3; cudaMalloc( (void **)(&cuda_y3), sizeof(double)*N);
  double *cuda_y4; cudaMalloc( (void **)(&cuda_y4), sizeof(double)*N);
  double *cuda_y5; cudaMalloc( (void **)(&cuda_y5), sizeof(double)*N);
  double *cuda_y6; cudaMalloc( (void **)(&cuda_y6), sizeof(double)*N);
  double *cuda_y7; cudaMalloc( (void **)(&cuda_y7), sizeof(double)*N);
  double *cuda_results2; cudaMalloc( (void **)(&cuda_results2), sizeof(double)*8);



  //
  // 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);
    }
  }

  //
  // Reference calculation on CPU:
  //
  for (size_t i=0; i<K; ++i) {
    results[i] = 0;
    results2[i] = 0;
    for (size_t j=0; j<N; ++j) {
      results[i] += x[j] * y[i][j];
```

```
    }
  }

  //
  // Copy data to GPU
  //
  std::cout << "Copying data to GPU..." << std::endl;
  cudaMemcpy(cuda_x, x, sizeof(double)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_y0, y[0], sizeof(double)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_y1, y[1], sizeof(double)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_y2, y[2], sizeof(double)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_y3, y[3], sizeof(double)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_y4, y[4], sizeof(double)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_y5, y[5], sizeof(double)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_y6, y[6], sizeof(double)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_y7, y[7], sizeof(double)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_results2, results2, sizeof(double)*8, cudaMemcpyHostToDevice);


  double *resultslarge  = (double*)malloc(sizeof(double) * K);

  std::vector<double> timings;
  timer.reset();

  for(int reps=0; reps < 10; ++reps)
  {

    std::cout << "Running dot products simultaneously..." << std::endl;
    for (size_t i=0; i<K/8; ++i) {
      cudaDeviceSynchronize();
      cudaMemcpy(cuda_y0, y[i*8+0], sizeof(double)*N, cudaMemcpyHostToDevice);
      cudaMemcpy(cuda_y1, y[i*8+1], sizeof(double)*N, cudaMemcpyHostToDevice);
      cudaMemcpy(cuda_y2, y[i*8+2], sizeof(double)*N, cudaMemcpyHostToDevice);
      cudaMemcpy(cuda_y3, y[i*8+3], sizeof(double)*N, cudaMemcpyHostToDevice);
      cudaMemcpy(cuda_y4, y[i*8+4], sizeof(double)*N, cudaMemcpyHostToDevice);
      cudaMemcpy(cuda_y5, y[i*8+5], sizeof(double)*N, cudaMemcpyHostToDevice);
      cudaMemcpy(cuda_y6, y[i*8+6], sizeof(double)*N, cudaMemcpyHostToDevice);
      cudaMemcpy(cuda_y7, y[i*8+7], sizeof(double)*N, cudaMemcpyHostToDevice);
      cuda_many_dot_product<<<512, 512>>>(N, cuda_x, cuda_y0, cuda_y1, cuda_y2,
      cuda_y3, cuda_y4, cuda_y5, cuda_y6,
      cuda_y7, cuda_results2);
      cudaMemcpy(resultslarge+i*8, cuda_results2, sizeof(double)*8,
      cudaMemcpyDeviceToHost);
      for (int j = 0; j < 8; j++) {
        results2[j] = 0;
      }
      cudaMemcpy(cuda_results2, results2, sizeof(double)*8, cudaMemcpyHostToDevice);
    }

    //
    // Compare results
    //
    std::cout << "Copying results back to host..." << std::endl;
    for (size_t i=0; i<K; ++i) {
      std::cout << results[i] << " on CPU, " << resultslarge[i] << " on GPU.
      Relative difference: " << fabs(results[i] - resultslarge[i]) / results[i]
      << std::endl;
```

```
    }

    timings.push_back(timer.get());
  }

  std::sort(timings.begin(), timings.end());
  double time_elapsed = timings[10/2];


  printf("Dot product took %g seconds", time_elapsed);


  //
  // Clean up:
  //
  std::cout << std::endl << std::endl << "Cleaning up..." << std::endl;
  free(x);
  cudaFree(cuda_x);
  cudaFree(cuda_y0);
  cudaFree(cuda_y1);
  cudaFree(cuda_y2);
  cudaFree(cuda_y3);
  cudaFree(cuda_y4);
  cudaFree(cuda_y5);
  cudaFree(cuda_y6);
  cudaFree(cuda_y7);

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


  free(results);
  free(results2);
  free(resultslarge);

  return 0;
}
```

3 Multiple Dot Products 4

So, how could one make the multiple dot product applicable to general values of K? I think the best approach would be to pass the vectors y to the kernel in form of a matrix and then let the kernel automatically deduce how many columns the matrix has and therefore how many dot products should be computed. The kernel's variables then wouldn't be replicated but dynamically allocated and indexed, eg. dot[0] instead of dot_0. So generally speaking, one would have to shift from static copies to dynamically allocated indexing of variables.

4 Pipelined Conjugate Gradients

It can be seen that the Pipelined CG is faster the Standard CG, but not much. I expected the difference to be bigger. It's also noticable that the difference grows bigger with increasing N. According to the lecture it should be different, at least that's what I expected. The highest system resolution I could go to was 2100, the Standard didn't even finish for that in the GPU given time. The number of iterations for the CG to converge is exactly the same for both methods. This was some kind of reassurance for me that my implementation worked properly.

As a performance benefit I would say the Pipelined version works better for large N, yet I would have expected the contrary according to the lecture.

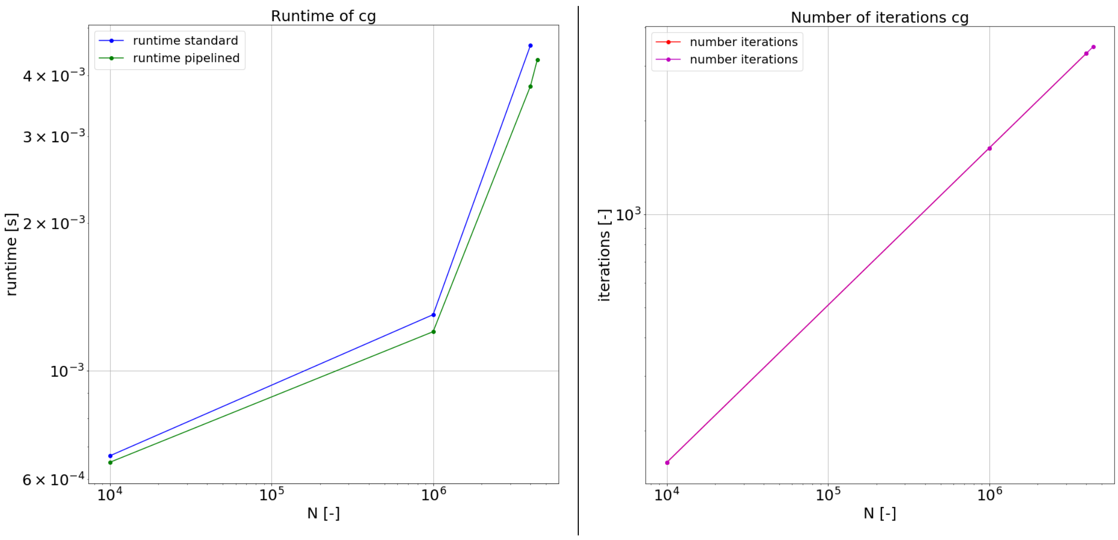

Figure 2: Runtime and iteration numbers for normal and pipelined CG

Listing 3: Pipelinded CG

```
#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;
  }
}


```

```
// 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]);
}

 __global__ void lines_7_to_9(int N, double *x, double *p, double *r, double *Ap,
 double *rr, double alpha, double beta)
 {
   __shared__ double shared_mem_2[512];
   double dot_2 = 0;

   for (int i = blockIdx.x * blockDim.x + threadIdx.x;
   i < N; i += blockDim.x * gridDim.x)
   {
     x[i] = x[i] + alpha * p[i];
     r[i] = r[i] - alpha * Ap[i];
     p[i] = r[i] + beta * p[i];
     dot_2 += r[i] * r[i];
   }

   shared_mem_2[threadIdx.x] = dot_2;

   for (int k = blockDim.x / 2; k > 0; k /= 2) {
    __syncthreads();
    if (threadIdx.x < k) {
      shared_mem_2[threadIdx.x] += shared_mem_2[threadIdx.x + k];
    }
   }


  if (threadIdx.x == 0) {
    atomicAdd(rr, shared_mem_2[0]);
 }
}

 __global__ void lines_10_to_11(int N, double *Ap,
 double *p, double *r, double *ApAp, double *pAp,
 int *csr_rowoffsets, int *csr_colindices, double *csr_values)
 {
   __shared__ double shared_mem_0[512];
```

```
    __shared__ double shared_mem_1[512];

  double dot_0 = 0;
  double dot_1 = 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_0 += Ap[i] * Ap[i];
    dot_1 += p[i] * Ap[i];
  }

  shared_mem_0[threadIdx.x] = dot_0;
  shared_mem_1[threadIdx.x] = dot_1;

  for (int k = blockDim.x / 2; k > 0; k /= 2) {
    __syncthreads();
    if (threadIdx.x < k) {
      shared_mem_0[threadIdx.x] += shared_mem_0[threadIdx.x + k];
      shared_mem_1[threadIdx.x] += shared_mem_1[threadIdx.x + k];
    }
  }

  if (threadIdx.x == 0) {
    atomicAdd(ApAp, shared_mem_0[0]);
    atomicAdd(pAp, shared_mem_1[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_ApAp, *cuda_rr, *cuda_pAp;
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_ApAp, sizeof(double));
cudaMalloc(&cuda_pAp, 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_solution, solution, sizeof(double) * N, cudaMemcpyHostToDevice);

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

double initial_residual_squared = residual_norm_squared;

// line 3
cuda_csr_matvec_product<<<512, 512>>>(N, csr_rowoffsets, csr_colindices,
csr_values, cuda_p, cuda_Ap);

// line 4
cudaMemcpy(cuda_pAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
cuda_dot_product<<<512, 512>>>(N, cuda_p, cuda_Ap, cuda_pAp);
cudaMemcpy(&alpha, cuda_pAp, sizeof(double), cudaMemcpyDeviceToHost);
alpha = residual_norm_squared / alpha;

// line 5
cudaMemcpy(cuda_ApAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
cuda_dot_product<<<512,512>>>(N, cuda_Ap, cuda_Ap, cuda_ApAp);
cudaMemcpy(&beta, cuda_ApAp, sizeof(double), cudaMemcpyDeviceToHost);
beta = alpha * alpha * beta / residual_norm_squared - 1;

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

  // lines 7 to 9:
  cudaMemcpy(cuda_rr, &zero, sizeof(double), cudaMemcpyHostToDevice);
  lines_7_to_9<<<512,512>>>(N, cuda_solution, cuda_p, cuda_r, cuda_Ap,
  cuda_rr, alpha, beta);
  cudaMemcpy(&residual_norm_squared, cuda_rr,
  sizeof(double), cudaMemcpyDeviceToHost);


  //lines 11 to 12:
  cudaMemcpy(cuda_ApAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_pAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
  lines_10_to_11<<<512,512>>>(N, cuda_Ap, cuda_p, cuda_r, cuda_ApAp,
  cuda_pAp, csr_rowoffsets, csr_colindices, csr_values);
```

```
    cudaMemcpy(&beta, cuda_ApAp, sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(&alpha, cuda_pAp, sizeof(double), cudaMemcpyDeviceToHost);


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

    // line 13:
    alpha = residual_norm_squared / alpha;

    // line 14:
    beta = alpha * alpha * beta / residual_norm_squared - 1;

    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)" << 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;

  cudaFree(cuda_p);
  cudaFree(cuda_r);
  cudaFree(cuda_Ap);
  cudaFree(cuda_solution);
  cudaFree(cuda_ApAp);
  cudaFree(cuda_rr);
  cudaFree(cuda_pAp);
}

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

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

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

  //
  // 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
    //
    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, solution);
    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);
  }
```

```
int main() {

  solve_system(1000); // solves a system with 100*100 unknowns

  return EXIT_SUCCESS;
}
```