



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	1
2	Multiple Dot Products 2 + 3	3
3	Multiple Dot Products 4	8
4	Pipelined Conjugate Gradients	9

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

```
1  __global__ void cuda_many_dot_product(int N, double *x, double *y0, double *y1,
2  double *y2, double *y3, double *y4, double *y5, double *y6, double *y7,
3  double *result)
4  {
5      __shared__ double shared_mem_0[512];
6      __shared__ double shared_mem_1[512];
7      __shared__ double shared_mem_2[512];
8      __shared__ double shared_mem_3[512];
9      __shared__ double shared_mem_4[512];
10     __shared__ double shared_mem_5[512];
11     __shared__ double shared_mem_6[512];
12     __shared__ double shared_mem_7[512];
13
14     double dot_0 = 0;
15     double dot_1 = 0;
16     double dot_2 = 0;
17     double dot_3 = 0;
18     double dot_4 = 0;
19     double dot_5 = 0;
20     double dot_6 = 0;
21     double dot_7 = 0;
22
23     for (int i = blockIdx.x * blockDim.x + threadIdx.x;
24          i < N; i += blockDim.x * gridDim.x) {
25         double val = x[i];
26         dot_0 += val * y0[i];
27         dot_1 += val * y1[i];
28         dot_2 += val * y2[i];
29         dot_3 += val * y3[i];
30         dot_4 += val * y4[i];
31         dot_5 += val * y5[i];
32         dot_6 += val * y6[i];
33         dot_7 += val * y7[i];
34     }
35
36     shared_mem_0[threadIdx.x] = dot_0;
37     shared_mem_1[threadIdx.x] = dot_1;
38     shared_mem_2[threadIdx.x] = dot_2;
39     shared_mem_3[threadIdx.x] = dot_3;
40     shared_mem_4[threadIdx.x] = dot_4;
41     shared_mem_5[threadIdx.x] = dot_5;
42     shared_mem_6[threadIdx.x] = dot_6;
43     shared_mem_7[threadIdx.x] = dot_7;
44
45     for (int k = blockDim.x / 2; k > 0; k /= 2) {
46         __syncthreads();
```

```

47     if (threadIdx.x < k) {
48         shared_mem_0[threadIdx.x] += shared_mem_0[threadIdx.x + k];
49         shared_mem_1[threadIdx.x] += shared_mem_1[threadIdx.x + k];
50         shared_mem_2[threadIdx.x] += shared_mem_2[threadIdx.x + k];
51         shared_mem_3[threadIdx.x] += shared_mem_3[threadIdx.x + k];
52         shared_mem_4[threadIdx.x] += shared_mem_4[threadIdx.x + k];
53         shared_mem_5[threadIdx.x] += shared_mem_5[threadIdx.x + k];
54         shared_mem_6[threadIdx.x] += shared_mem_6[threadIdx.x + k];
55         shared_mem_7[threadIdx.x] += shared_mem_7[threadIdx.x + k];
56     }
57 }
58
59 if (threadIdx.x == 0){
60     atomicAdd(result+0, shared_mem_0[0]);
61     atomicAdd(result+1, shared_mem_1[0]);
62     atomicAdd(result+2, shared_mem_2[0]);
63     atomicAdd(result+3, shared_mem_3[0]);
64     atomicAdd(result+4, shared_mem_4[0]);
65     atomicAdd(result+5, shared_mem_5[0]);
66     atomicAdd(result+6, shared_mem_6[0]);
67     atomicAdd(result+7, shared_mem_7[0]);
68 }
69 }

```

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.

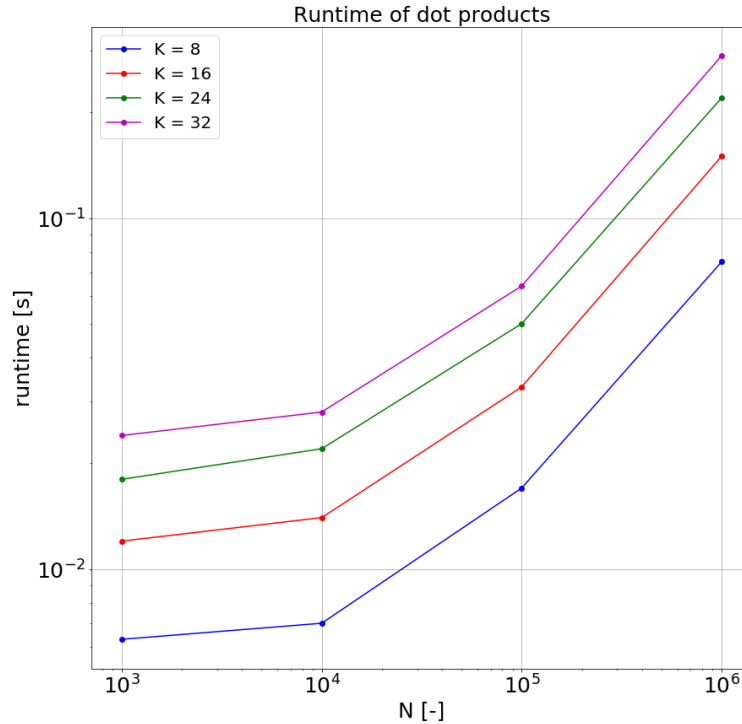


Figure 1: Runtime for dot product of vectors with N entries

Listing 2: Code for points 2 and 3

```

1  #include <cuda_runtime.h>
2  #include <cublas_v2.h>
3  #include <stdio.h>
4  #include <cmath>
5  #include <iostream>
6  #include "timer.hpp"
7  #include <algorithm>
8  #include <vector>
9
10 __global__ void cuda_many_dot_product(int N, double *x, double *y0, double *y1,
11 double *y2, double *y3, double *y4, double *y5, double *y6,
12 double *y7, double *result)
13 {
14     __shared__ double shared_mem_0[512];
15     __shared__ double shared_mem_1[512];
16     __shared__ double shared_mem_2[512];
17     __shared__ double shared_mem_3[512];
18     __shared__ double shared_mem_4[512];
19     __shared__ double shared_mem_5[512];
20     __shared__ double shared_mem_6[512];
21     __shared__ double shared_mem_7[512];
22
23     double dot_0 = 0;

```

```

24     double dot_1 = 0;
25     double dot_2 = 0;
26     double dot_3 = 0;
27     double dot_4 = 0;
28     double dot_5 = 0;
29     double dot_6 = 0;
30     double dot_7 = 0;
31
32     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N;
33          i += blockDim.x * gridDim.x) {
34         double val = x[i];
35         dot_0 += val * y0[i];
36         dot_1 += val * y1[i];
37         dot_2 += val * y2[i];
38         dot_3 += val * y3[i];
39         dot_4 += val * y4[i];
40         dot_5 += val * y5[i];
41         dot_6 += val * y6[i];
42         dot_7 += val * y7[i];
43     }
44
45     shared_mem_0[threadIdx.x] = dot_0;
46     shared_mem_1[threadIdx.x] = dot_1;
47     shared_mem_2[threadIdx.x] = dot_2;
48     shared_mem_3[threadIdx.x] = dot_3;
49     shared_mem_4[threadIdx.x] = dot_4;
50     shared_mem_5[threadIdx.x] = dot_5;
51     shared_mem_6[threadIdx.x] = dot_6;
52     shared_mem_7[threadIdx.x] = dot_7;
53
54     for (int k = blockDim.x / 2; k > 0; k /= 2) {
55         __syncthreads();
56         if (threadIdx.x < k) {
57             shared_mem_0[threadIdx.x] += shared_mem_0[threadIdx.x + k];
58             shared_mem_1[threadIdx.x] += shared_mem_1[threadIdx.x + k];
59             shared_mem_2[threadIdx.x] += shared_mem_2[threadIdx.x + k];
60             shared_mem_3[threadIdx.x] += shared_mem_3[threadIdx.x + k];
61             shared_mem_4[threadIdx.x] += shared_mem_4[threadIdx.x + k];
62             shared_mem_5[threadIdx.x] += shared_mem_5[threadIdx.x + k];
63             shared_mem_6[threadIdx.x] += shared_mem_6[threadIdx.x + k];
64             shared_mem_7[threadIdx.x] += shared_mem_7[threadIdx.x + k];
65         }
66     }
67
68     if (threadIdx.x == 0){
69         atomicAdd(result+0, shared_mem_0[0]);
70         atomicAdd(result+1, shared_mem_1[0]);
71         atomicAdd(result+2, shared_mem_2[0]);
72         atomicAdd(result+3, shared_mem_3[0]);
73         atomicAdd(result+4, shared_mem_4[0]);
74         atomicAdd(result+5, shared_mem_5[0]);
75         atomicAdd(result+6, shared_mem_6[0]);
76         atomicAdd(result+7, shared_mem_7[0]);
77     }
78 }
79
80 int main(void)

```

```

81 {
82
83     Timer timer;
84     const size_t N = 100000;
85     const size_t K = 16;
86
87
88     //
89     // allocate host memory:
90     //
91     std::cout << "Allocating host arrays..." << std::endl;
92     double *x = (double*)malloc(sizeof(double) * N);
93     double **y = (double**)malloc(sizeof(double*) * K);
94     for (size_t i=0; i<K; ++i) {
95         y[i] = (double*)malloc(sizeof(double) * N);
96     }
97     double *results = (double*)malloc(sizeof(double) * K);
98     double *results2 = (double*)malloc(sizeof(double) * 8);
99
100
101     //
102     // allocate device memory
103     //
104     std::cout << "Allocating CUDA arrays..." << std::endl;
105     double *cuda_x; cudaMalloc( (void **>(&cuda_x), sizeof(double)*N);
106     double *cuda_y0; cudaMalloc( (void **>(&cuda_y0), sizeof(double)*N);
107     double *cuda_y1; cudaMalloc( (void **>(&cuda_y1), sizeof(double)*N);
108     double *cuda_y2; cudaMalloc( (void **>(&cuda_y2), sizeof(double)*N);
109     double *cuda_y3; cudaMalloc( (void **>(&cuda_y3), sizeof(double)*N);
110     double *cuda_y4; cudaMalloc( (void **>(&cuda_y4), sizeof(double)*N);
111     double *cuda_y5; cudaMalloc( (void **>(&cuda_y5), sizeof(double)*N);
112     double *cuda_y6; cudaMalloc( (void **>(&cuda_y6), sizeof(double)*N);
113     double *cuda_y7; cudaMalloc( (void **>(&cuda_y7), sizeof(double)*N);
114     double *cuda_results2; cudaMalloc( (void **>(&cuda_results2), sizeof(double)*8);
115
116
117
118     //
119     // fill host arrays with values
120     //
121     for (size_t j=0; j<N; ++j) {
122         x[j] = 1 + j%K;
123     }
124     for (size_t i=0; i<K; ++i) {
125         for (size_t j=0; j<N; ++j) {
126             y[i][j] = 1 + rand() / (1.1 * RANDMAX);
127         }
128     }
129
130     //
131     // Reference calculation on CPU:
132     //
133     for (size_t i=0; i<K; ++i) {
134         results[i] = 0;
135         results2[i] = 0;
136         for (size_t j=0; j<N; ++j) {
137             results[i] += x[j] * y[i][j];

```

```

138     }
139 }
140
141 //
142 // Copy data to GPU
143 //
144 std::cout << "Copying data to GPU..." << std::endl;
145 cudaMemcpy(cuda_x, x, sizeof(double)*N, cudaMemcpyHostToDevice);
146 cudaMemcpy(cuda_y0, y[0], sizeof(double)*N, cudaMemcpyHostToDevice);
147 cudaMemcpy(cuda_y1, y[1], sizeof(double)*N, cudaMemcpyHostToDevice);
148 cudaMemcpy(cuda_y2, y[2], sizeof(double)*N, cudaMemcpyHostToDevice);
149 cudaMemcpy(cuda_y3, y[3], sizeof(double)*N, cudaMemcpyHostToDevice);
150 cudaMemcpy(cuda_y4, y[4], sizeof(double)*N, cudaMemcpyHostToDevice);
151 cudaMemcpy(cuda_y5, y[5], sizeof(double)*N, cudaMemcpyHostToDevice);
152 cudaMemcpy(cuda_y6, y[6], sizeof(double)*N, cudaMemcpyHostToDevice);
153 cudaMemcpy(cuda_y7, y[7], sizeof(double)*N, cudaMemcpyHostToDevice);
154 cudaMemcpy(cuda_results2, results2, sizeof(double)*8, cudaMemcpyHostToDevice);
155
156
157 double *resultslarge = (double*)malloc(sizeof(double) * K);
158
159 std::vector<double> timings;
160 timer.reset();
161
162 for(int reps=0; reps < 10; ++reps)
163 {
164
165     std::cout << "Running dot products simultaneously..." << std::endl;
166     for (size_t i=0; i<K/8; ++i) {
167         cudaDeviceSynchronize();
168         cudaMemcpy(cuda_y0, y[i*8+0], sizeof(double)*N, cudaMemcpyHostToDevice);
169         cudaMemcpy(cuda_y1, y[i*8+1], sizeof(double)*N, cudaMemcpyHostToDevice);
170         cudaMemcpy(cuda_y2, y[i*8+2], sizeof(double)*N, cudaMemcpyHostToDevice);
171         cudaMemcpy(cuda_y3, y[i*8+3], sizeof(double)*N, cudaMemcpyHostToDevice);
172         cudaMemcpy(cuda_y4, y[i*8+4], sizeof(double)*N, cudaMemcpyHostToDevice);
173         cudaMemcpy(cuda_y5, y[i*8+5], sizeof(double)*N, cudaMemcpyHostToDevice);
174         cudaMemcpy(cuda_y6, y[i*8+6], sizeof(double)*N, cudaMemcpyHostToDevice);
175         cudaMemcpy(cuda_y7, y[i*8+7], sizeof(double)*N, cudaMemcpyHostToDevice);
176         cuda_many_dot_product<<<512, 512>>>(N, cuda_x, cuda_y0, cuda_y1, cuda_y2,
177         cuda_y3, cuda_y4, cuda_y5, cuda_y6,
178         cuda_y7, cuda_results2);
179         cudaMemcpy(resultslarge+i*8, cuda_results2, sizeof(double)*8,
180         cudaMemcpyDeviceToHost);
181         for (int j = 0; j < 8; j++) {
182             results2[j] = 0;
183         }
184         cudaMemcpy(cuda_results2, results2, sizeof(double)*8, cudaMemcpyHostToDevice);
185     }
186
187 //
188 // Compare results
189 //
190 std::cout << "Copying results back to host..." << std::endl;
191 for (size_t i=0; i<K; ++i) {
192     std::cout << results[i] << " on CPU, " << resultslarge[i] << " on GPU.
193     Relative difference: " << fabs(results[i] - resultslarge[i]) / results[i]
194     << std::endl;

```



```

195     }
196
197     timings.push_back(timer.get());
198 }
199
200 std::sort(timings.begin(), timings.end());
201 double time_elapsed = timings[10/2];
202
203
204 printf("Dot product took %g seconds", time_elapsed);
205
206
207 //
208 // Clean up:
209 //
210 std::cout << std::endl << std::endl << "Cleaning up..." << std::endl;
211 free(x);
212 cudaFree(cuda_x);
213 cudaFree(cuda_y0);
214 cudaFree(cuda_y1);
215 cudaFree(cuda_y2);
216 cudaFree(cuda_y3);
217 cudaFree(cuda_y4);
218 cudaFree(cuda_y5);
219 cudaFree(cuda_y6);
220 cudaFree(cuda_y7);
221
222 for (size_t i=0; i<K; ++i) {
223     free(y[i]);
224 }
225 free(y);
226
227
228 free(results);
229 free(results2);
230 free(resultslarge);
231
232 return 0;
233 }

```

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 than the Standard CG, but not much. I expected the difference to be bigger. It's also noticeable 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.

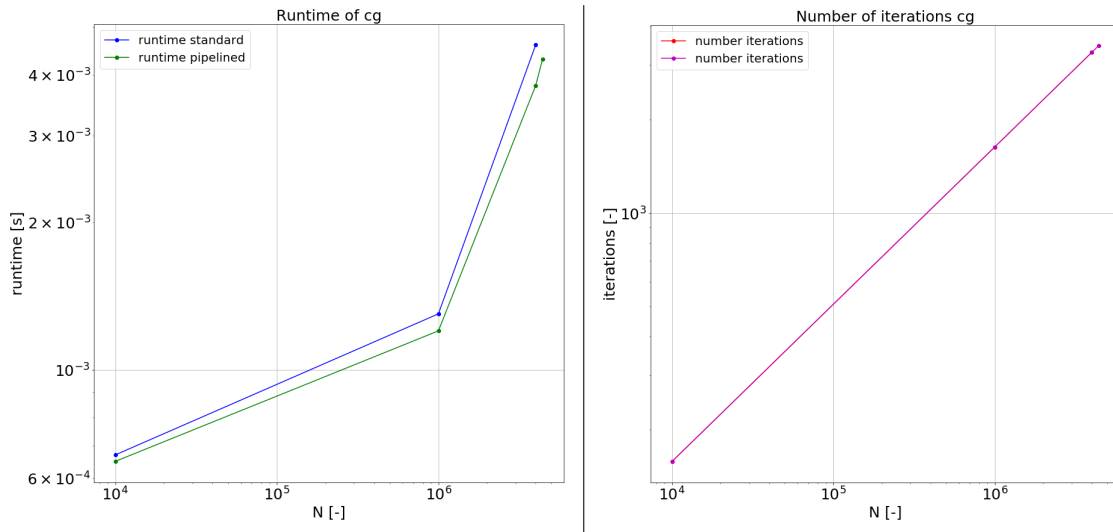


Figure 2: Runtime and iteration numbers for normal and pipelined CG

Listing 3: Pipelined CG

```

1  #include "poisson2d.hpp"
2  #include "timer.hpp"
3  #include <algorithm>
4  #include <iostream>
5  #include <stdio.h>
6
7  // y = A * x
8  --global-- void cuda_csr_matvec_product(int N, int *csr_rowoffsets,
9                                          int *csr_colindices, double *csr_values,
10                                         double *x, double *y)
11  {
12      for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N;
13           i += blockDim.x * gridDim.x) {
14          double sum = 0;
15          for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++) {
16              sum += csr_values[k] * x[csr_colindices[k]];
17          }
18          y[i] = sum;
19      }
20  }
21
22

```

```

23 // result = (x, y)
24 __global__ void cuda_dot_product(int N, double *x, double *y, double *result)
25 {
26     __shared__ double shared_mem[512];
27
28     double dot = 0;
29     for (int i = blockIdx.x * blockDim.x + threadIdx.x;
30          i < N; i += blockDim.x * gridDim.x) {
31         dot += x[i] * y[i];
32     }
33
34     shared_mem[threadIdx.x] = dot;
35     for (int k = blockDim.x / 2; k > 0; k /= 2) {
36         __syncthreads();
37         if (threadIdx.x < k) {
38             shared_mem[threadIdx.x] += shared_mem[threadIdx.x + k];
39         }
40     }
41
42     if (threadIdx.x == 0) atomicAdd(result, shared_mem[0]);
43 }
44
45 __global__ void lines_7_to_9(int N, double *x, double *p, double *r, double *Ap,
46 double *rr, double alpha, double beta)
47 {
48     __shared__ double shared_mem_2[512];
49     double dot_2 = 0;
50
51     for (int i = blockIdx.x * blockDim.x + threadIdx.x;
52          i < N; i += blockDim.x * gridDim.x)
53     {
54         x[i] = x[i] + alpha * p[i];
55         r[i] = r[i] - alpha * Ap[i];
56         p[i] = r[i] + beta * p[i];
57         dot_2 += r[i] * r[i];
58     }
59
60     shared_mem_2[threadIdx.x] = dot_2;
61
62     for (int k = blockDim.x / 2; k > 0; k /= 2) {
63         __syncthreads();
64         if (threadIdx.x < k) {
65             shared_mem_2[threadIdx.x] += shared_mem_2[threadIdx.x + k];
66         }
67     }
68
69
70     if (threadIdx.x == 0) {
71         atomicAdd(rr, shared_mem_2[0]);
72     }
73 }
74
75 __global__ void lines_10_to_11(int N, double *Ap,
76 double *p, double *r, double *ApAp, double *pAp,
77 int *csr_rowoffsets, int *csr_colindices, double *csr_values)
78 {
79     __shared__ double shared_mem_0[512];

```

```

80     __shared__ double shared_mem_1[512];
81
82     double dot_0 = 0;
83     double dot_1 = 0;
84
85     for (int i = blockIdx.x * blockDim.x + threadIdx.x;
86          i < N; i += blockDim.x * gridDim.x) {
87         double sum = 0;
88         for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++) {
89             sum += csr_values[k] * p[csr_colindices[k]];
90         }
91         Ap[i] = sum;
92
93         dot_0 += Ap[i] * Ap[i];
94         dot_1 += p[i] * Ap[i];
95     }
96
97     shared_mem_0[threadIdx.x] = dot_0;
98     shared_mem_1[threadIdx.x] = dot_1;
99
100    for (int k = blockDim.x / 2; k > 0; k /= 2) {
101        __syncthreads();
102        if (threadIdx.x < k) {
103            shared_mem_0[threadIdx.x] += shared_mem_0[threadIdx.x + k];
104            shared_mem_1[threadIdx.x] += shared_mem_1[threadIdx.x + k];
105        }
106    }
107
108    if (threadIdx.x == 0) {
109        atomicAdd(&ApAp, shared_mem_0[0]);
110        atomicAdd(&pAp, shared_mem_1[0]);
111    }
112 }
113
114
115 /** Implementation of the conjugate gradient algorithm.
116  *
117  * The control flow is handled by the CPU.
118  * Only the individual operations (vector updates, dot products, sparse
119  * matrix-vector product) are transferred to CUDA kernels.
120  *
121  * The temporary arrays p, r, and Ap need to be allocated on the GPU for use
122  * with CUDA. Modify as you see fit.
123  */
124 void conjugate_gradient(int N, // number of unknowns
125                        int *csr_rowoffsets, int *csr_colindices,
126                        double *csr_values, double *rhs, double *solution)
127 //, double *init_guess) // feel free to add a nonzero initial guess as needed
128 {
129     // initialize timer
130     Timer timer;
131
132     // clear solution vector (it may contain garbage values):
133     std::fill(solution, solution + N, 0);
134
135     // initialize work vectors:
136     double alpha, beta;

```

```

137 double *cuda_solution , *cuda_p , *cuda_r , *cuda_Ap , *cuda_ApAp , *cuda_rr , *cuda_pAp ;
138 cudaMalloc(&cuda_p , sizeof(double) * N);
139 cudaMalloc(&cuda_r , sizeof(double) * N);
140 cudaMalloc(&cuda_Ap , sizeof(double) * N);
141 cudaMalloc(&cuda_solution , sizeof(double) * N);
142 cudaMalloc(&cuda_ApAp , sizeof(double));
143 cudaMalloc(&cuda_pAp , sizeof(double));
144 cudaMalloc(&cuda_rr , sizeof(double));
145
146
147
148 cudaMemcpy(cuda_p , rhs , sizeof(double) * N , cudaMemcpyHostToDevice);
149 cudaMemcpy(cuda_r , rhs , sizeof(double) * N , cudaMemcpyHostToDevice);
150 cudaMemcpy(cuda_solution , solution , sizeof(double) * N , cudaMemcpyHostToDevice);
151
152 const double zero = 0;
153 double residual_norm_squared = 0;
154 cudaMemcpy(cuda_rr , &zero , sizeof(double) , cudaMemcpyHostToDevice);
155 cuda_dot_product <<<512, 512>>>(N , cuda_r , cuda_r , cuda_rr);
156 cudaMemcpy(&residual_norm_squared , cuda_rr , sizeof(double) , cudaMemcpyDeviceToHost);
157
158 double initial_residual_squared = residual_norm_squared;
159
160 // line 3
161 cuda_csr_matvec_product <<<512, 512>>>(N , csr_rowoffsets , csr_colindices ,
162 csr_values , cuda_p , cuda_Ap);
163
164 // line 4
165 cudaMemcpy(cuda_pAp , &zero , sizeof(double) , cudaMemcpyHostToDevice);
166 cuda_dot_product <<<512, 512>>>(N , cuda_p , cuda_Ap , cuda_pAp);
167 cudaMemcpy(&alpha , cuda_pAp , sizeof(double) , cudaMemcpyDeviceToHost);
168 alpha = residual_norm_squared / alpha;
169
170 // line 5
171 cudaMemcpy(cuda_ApAp , &zero , sizeof(double) , cudaMemcpyHostToDevice);
172 cuda_dot_product <<<512,512>>>(N , cuda_Ap , cuda_Ap , cuda_ApAp);
173 cudaMemcpy(&beta , cuda_ApAp , sizeof(double) , cudaMemcpyDeviceToHost);
174 beta = alpha * alpha * beta / residual_norm_squared - 1;
175
176 int iters = 0;
177 cudaDeviceSynchronize();
178 timer.reset();
179 while (1) {
180
181     // lines 7 to 9:
182     cudaMemcpy(cuda_rr , &zero , sizeof(double) , cudaMemcpyHostToDevice);
183     lines_7_to_9 <<<512,512>>>(N , cuda_solution , cuda_p , cuda_r , cuda_Ap ,
184     cuda_rr , alpha , beta);
185     cudaMemcpy(&residual_norm_squared , cuda_rr ,
186     sizeof(double) , cudaMemcpyDeviceToHost);
187
188
189     //lines 11 to 12:
190     cudaMemcpy(cuda_ApAp , &zero , sizeof(double) , cudaMemcpyHostToDevice);
191     cudaMemcpy(cuda_pAp , &zero , sizeof(double) , cudaMemcpyHostToDevice);
192     lines_10_to_11 <<<512,512>>>(N , cuda_Ap , cuda_p , cuda_r , cuda_ApAp ,
193     cuda_pAp , csr_rowoffsets , csr_colindices , csr_values);

```

```

194     cudaMemcpy(&beta, cuda_ApAp, sizeof(double), cudaMemcpyDeviceToHost);
195     cudaMemcpy(&alpha, cuda_pAp, sizeof(double), cudaMemcpyDeviceToHost);
196
197
198     // line convergence check:
199     if (std::sqrt(residual_norm_squared / initial_residual_squared) < 1e-6) {
200         break;
201     }
202
203     // line 13:
204     alpha = residual_norm_squared / alpha;
205
206     // line 14:
207     beta = alpha * alpha * beta / residual_norm_squared - 1;
208
209     if (iters > 10000)
210         break; // solver didn't converge
211     ++iters;
212 }
213 cudaMemcpy(solution, cuda_solution, sizeof(double) * N, cudaMemcpyDeviceToHost);
214
215 cudaDeviceSynchronize();
216 std::cout << "Time elapsed: " << timer.get() << " (" << timer.get() / iters
217 << " per iteration)" << std::endl;
218
219 if (iters > 10000)
220     std::cout << "Conjugate Gradient did NOT converge within 10000 iterations"
221     << std::endl;
222 else
223     std::cout << "Conjugate Gradient converged in " << iters << " iterations."
224     << std::endl;
225
226 cudaFree(cuda_p);
227 cudaFree(cuda_r);
228 cudaFree(cuda_Ap);
229 cudaFree(cuda_solution);
230 cudaFree(cuda_ApAp);
231 cudaFree(cuda_rr);
232 cudaFree(cuda_pAp);
233 }
234
235 /** Solve a system with 'points_per_direction * points_per_direction' unknowns
236 */
237 void solve_system(int points_per_direction) {
238
239     int N = points_per_direction *
240             points_per_direction; // number of unknowns to solve for
241
242     std::cout << "Solving Ax=b with " << N << " unknowns." << std::endl;
243
244     //
245     // Allocate CSR arrays.
246     //
247     // Note: Usually one does not know the number of nonzeros in the system matrix
248     // a-priori.
249     // For this exercise, however, we know that there are at most 5 nonzeros
250     // per row in the system matrix, so we can allocate accordingly.

```

```

251 //
252 int *csr_rowoffsets = (int *)malloc(sizeof(double) * (N + 1));
253 int *csr_colindices = (int *)malloc(sizeof(double) * 5 * N);
254 double *csr_values = (double *)malloc(sizeof(double) * 5 * N);
255
256 int *cuda_csr_rowoffsets, *cuda_csr_colindices;
257 double *cuda_csr_values;
258 //
259 // fill CSR matrix with values
260 //
261 generate_fdm_laplace(points_per_direction, csr_rowoffsets, csr_colindices,
262                     csr_values);
263
264 //
265 // Allocate solution vector and right hand side:
266 //
267 double *solution = (double *)malloc(sizeof(double) * N);
268 double *rhs = (double *)malloc(sizeof(double) * N);
269 std::fill(rhs, rhs + N, 1);
270
271 //
272 // Allocate CUDA arrays //
273 //
274 cudaMalloc(&cuda_csr_rowoffsets, sizeof(double) * (N + 1));
275 cudaMalloc(&cuda_csr_colindices, sizeof(double) * 5 * N);
276 cudaMalloc(&cuda_csr_values, sizeof(double) * 5 * N);
277 cudaMemcpy(cuda_csr_rowoffsets, csr_rowoffsets, sizeof(double) * (N + 1),
278             cudaMemcpyHostToDevice);
279 cudaMemcpy(cuda_csr_colindices, csr_colindices, sizeof(double) * 5 * N,
280             cudaMemcpyHostToDevice);
281 cudaMemcpy(cuda_csr_values, csr_values, sizeof(double) * 5 * N,
282             cudaMemcpyHostToDevice);
283
284 //
285 // Call Conjugate Gradient implementation with GPU arrays
286 //
287 conjugate_gradient(N, cuda_csr_rowoffsets, cuda_csr_colindices,
288                   cuda_csr_values, rhs, solution);
289
290 //
291 // Check for convergence:
292 //
293 double residual_norm = relative_residual(N, csr_rowoffsets, csr_colindices,
294     csr_values, rhs, solution);
295 std::cout << "Relative residual norm: " << residual_norm
296     << " (should be smaller than 1e-6)" << std::endl;
297
298 cudaFree(cuda_csr_rowoffsets);
299 cudaFree(cuda_csr_colindices);
300 cudaFree(cuda_csr_values);
301 free(solution);
302 free(rhs);
303 free(csr_rowoffsets);
304 free(csr_colindices);
305 free(csr_values);
306 }
307

```



```
308 int main() {  
309     solve_system(1000); // solves a system with 100*100 unknowns  
310  
311     return EXIT_SUCCESS;  
312 }  
313 }
```
