

Computational Science on Many-Core Architectures:

Exercise 7:

Viktor Beck, 11713110

December 2022

Pipelined Conjugate Gradients

In the `conjugate_gradient` function only two kernels are called. The variables are computed like in the pseudo code from lecture 7a. In the figure below one can see that the pipelined implementation is significantly faster due to less kernel calls per iteration especially for smaller numbers of N . For higher numbers the difference becomes smaller.

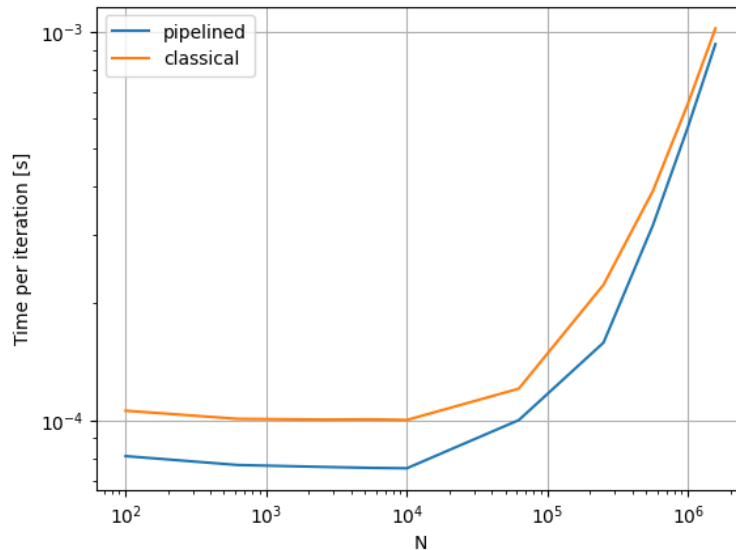


Figure 1: Times per iteration for N between 10^3 and 10^7 for the RTX GPU

Kernel 1:

Here the given functions (like `cuda_vecAdd`, etc.) were simply copied into a new kernel function and adjusted. x , r , p and the dot product can be computed in single for-loop. For the dot product a sum reduction is performed afterwards.

```
1 //////////////// CG KERNEL 1: ///////////////////
2
3 __global__ void cuda_cg_1(int N,
4                           double alpha,
```

```

5         double beta,
6         double *x,
7         double *r,
8         double *p,
9         double *Ap,
10        double *result_rr)
11 {
12
13     __shared__ double shared_mem[512];
14     double dot = 0;
15
16     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
17     {
18         // line 2,3,4: get x, r, p
19         x[i] = x[i] + alpha * p[i];
20         r[i] = r[i] - alpha * Ap[i];
21         p[i] = r[i] + beta * p[i];
22
23         // line 6: get dot(r,r)
24         dot += r[i] * r[i];
25     }
26
27     __syncthreads();
28     shared_mem[threadIdx.x] = dot;
29
30     for (int k = blockDim.x / 2; k > 0; k /= 2)
31     {
32         __syncthreads();
33         if (threadIdx.x < k)
34         {
35             shared_mem[threadIdx.x] += shared_mem[threadIdx.x + k];
36         }
37     }
38
39     if (threadIdx.x == 0)
40     {
41         atomicAdd(result_rr, shared_mem[0]);
42         // printf("%g", r[0]);
43     }
44 }

```

Kernel 2:

Here the `cuda_csr_matvec_product` function was copied into the kernel for the computation of `Ap` and the norms are computed like in kernel 1 as dot products.

```

1  //////////// CG KERNEL 2: ////////////
2  __global__ void cuda_cg_2(int N,
3                          int *csr_rowoffsets,
4                          int *csr_colindices,
5                          double *csr_values,
6                          double *p,
7                          double *Ap,
8                          double *result1,
9                          double *result2)
10 {
11
12     __shared__ double shared_mem1[512];
13     __shared__ double shared_mem2[512];
14
15     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)

```

```

16 {
17     // line 5: get Ap
18     double sum = 0;
19     for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++)
20     {
21         sum += csr_values[k] * p[csr_colindices[k]];
22     }
23     Ap[i] = sum;
24 }
25
26 // line 6: get dot(Ap,Ap) and dot(p,Ap)
27 double dot1 = 0;
28 double dot2 = 0;
29 for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
30 {
31     dot1 += Ap[i] * Ap[i];
32     dot2 += p[i] * Ap[i];
33 }
34
35 shared_mem1[threadIdx.x] = dot1;
36 shared_mem2[threadIdx.x] = dot2;
37 for (int k = blockDim.x / 2; k > 0; k /= 2)
38 {
39     __syncthreads();
40     if (threadIdx.x < k)
41     {
42         shared_mem1[threadIdx.x] += shared_mem1[threadIdx.x + k];
43         shared_mem2[threadIdx.x] += shared_mem2[threadIdx.x + k];
44     }
45 }
46
47 if (threadIdx.x == 0)
48 {
49     atomicAdd(result1, shared_mem1[0]);
50     atomicAdd(result2, shared_mem2[0]);
51 }
52 }

```

Appendix

Task 1.b)

```
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; i += blockDim.x * gridDim.x)
13     {
14         double sum = 0;
15         for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++)
16         {
17             sum += csr_values[k] * x[csr_colindices[k]];
18         }
19         y[i] = sum;
20     }
21 }
22
23 // x <- x + alpha * y
24 __global__ void cuda_vecadd(int N, double *x, double *y, double alpha)
25 {
26     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
27         x[i] += alpha * y[i];
28 }
29
30 // x <- y + alpha * x
31 __global__ void cuda_vecadd2(int N, double *x, double *y, double alpha)
32 {
33     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
34         x[i] = y[i] + alpha * x[i];
35 }
36
37 // result = (x, y)
38 __global__ void cuda_dot_product(int N, double *x, double *y, double *result)
39 {
40     __shared__ double shared_mem[512];
41
42     double dot = 0;
43     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
44     {
45         dot += x[i] * y[i];
46     }
47
48     shared_mem[threadIdx.x] = dot;
49     for (int k = blockDim.x / 2; k > 0; k /= 2)
50     {
51         __syncthreads();
52         if (threadIdx.x < k)
53         {
54             shared_mem[threadIdx.x] += shared_mem[threadIdx.x + k];
55         }
56     }
57 }
```

```

58     if (threadIdx.x == 0)
59         atomicAdd(result, shared_mem[0]);
60 }
61
62 //////////////// CG KERNEL 1: ////////////////
63
64 __global__ void cuda_cg_1(int N,
65                          double alpha,
66                          double beta,
67                          double *x,
68                          double *r,
69                          double *p,
70                          double *Ap,
71                          double *result_rr)
72 {
73
74     __shared__ double shared_mem[512];
75     double dot = 0;
76
77     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
78     {
79         // line 2,3,4: get x, r, p
80         x[i] = x[i] + alpha * p[i];
81         r[i] = r[i] - alpha * Ap[i];
82         p[i] = r[i] + beta * p[i];
83
84         // line 6: get dot(r,r)
85         dot += r[i] * r[i];
86     }
87
88     __syncthreads();
89     shared_mem[threadIdx.x] = dot;
90
91     for (int k = blockDim.x / 2; k > 0; k /= 2)
92     {
93         __syncthreads();
94         if (threadIdx.x < k)
95         {
96             shared_mem[threadIdx.x] += shared_mem[threadIdx.x + k];
97         }
98     }
99
100     if (threadIdx.x == 0)
101     {
102         atomicAdd(result_rr, shared_mem[0]);
103         // printf("%g", r[0]);
104     }
105 }
106
107 //////////////// CG KERNEL 2: ////////////////
108 __global__ void cuda_cg_2(int N,
109                          int *csr_rowoffsets,
110                          int *csr_colindices,
111                          double *csr_values,
112                          double *p,
113                          double *Ap,
114                          double *result1,
115                          double *result2)
116 {
117
118     __shared__ double shared_mem1[512];
119     __shared__ double shared_mem2[512];

```

```

120
121 for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
122 {
123     // line 5: get Ap
124     double sum = 0;
125     for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++)
126     {
127         sum += csr_values[k] * p[csr_colindices[k]];
128     }
129     Ap[i] = sum;
130 }
131
132 // line 6: get dot(Ap,Ap) and dot(p,Ap)
133 double dot1 = 0;
134 double dot2 = 0;
135 for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
136 {
137     dot1 += Ap[i] * Ap[i];
138     dot2 += p[i] * Ap[i];
139 }
140
141 shared_mem1[threadIdx.x] = dot1;
142 shared_mem2[threadIdx.x] = dot2;
143 for (int k = blockDim.x / 2; k > 0; k /= 2)
144 {
145     __syncthreads();
146     if (threadIdx.x < k)
147     {
148         shared_mem1[threadIdx.x] += shared_mem1[threadIdx.x + k];
149         shared_mem2[threadIdx.x] += shared_mem2[threadIdx.x + k];
150     }
151 }
152
153 if (threadIdx.x == 0)
154 {
155     atomicAdd(result1, shared_mem1[0]);
156     atomicAdd(result2, shared_mem2[0]);
157 }
158 }
159
160 /** Implementation of the conjugate gradient algorithm.
161  *
162  * The control flow is handled by the CPU.
163  * Only the individual operations (vector updates, dot products, sparse
164  * matrix-vector product) are transferred to CUDA kernels.
165  *
166  * The temporary arrays p, r, and Ap need to be allocated on the GPU for use
167  * with CUDA. Modify as you see fit.
168  */
169 void conjugate_gradient(int N, // number of unknowns
170                        int *csr_rowoffsets, int *csr_colindices,
171                        double *csr_values, double *rhs, double *solution)
172 //, double *init_guess) // feel free to add a nonzero initial guess as needed
173 {
174     // initialize timer
175     Timer timer;
176
177     // clear solution vector (it may contain garbage values):
178     std::fill(solution, solution + N, 0);
179
180     // initialize work vectors:
181     double alpha, beta, residual_norm_squared, dot_pAp, dot_ApAp;

```

```

182 double *cuda_solution, *cuda_p, *cuda_r, *cuda_Ap, *cuda_scalar, *cuda_dot_pAp, *cuda_dot_ApAp;
183 cudaMalloc(&cuda_p, sizeof(double) * N);
184 cudaMalloc(&cuda_r, sizeof(double) * N);
185 cudaMalloc(&cuda_Ap, sizeof(double) * N);
186 cudaMalloc(&cuda_solution, sizeof(double) * N);
187 cudaMalloc(&cuda_scalar, sizeof(double));
188
189 cudaMalloc(&cuda_dot_ApAp, sizeof(double));
190 cudaMalloc(&cuda_dot_pAp, sizeof(double));
191
192 cudaMemcpy(cuda_p, rhs, sizeof(double) * N, cudaMemcpyHostToDevice);
193 cudaMemcpy(cuda_r, rhs, sizeof(double) * N, cudaMemcpyHostToDevice);
194 cudaMemcpy(cuda_solution, solution, sizeof(double) * N, cudaMemcpyHostToDevice);
195
196 // get residual_norm_squared
197 const double zero = 0;
198 cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
199 cuda_dot_product<<<512, 512>>>(N, cuda_r, cuda_r, cuda_scalar);
200 cudaMemcpy(&residual_norm_squared, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
201
202 double initial_residual_squared = residual_norm_squared;
203
204 // line 1: get alpha0, beta0, Ap0
205 cuda_csr_matvec_product<<<512, 512>>>(N, csr_rowoffsets, csr_colindices, csr_values, cuda_p, cuda_Ap,
206
207 cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
208 cuda_dot_product<<<512, 512>>>(N, cuda_p, cuda_Ap, cuda_scalar);
209 cudaMemcpy(&dot_pAp, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
210 alpha = residual_norm_squared / dot_pAp;
211
212 cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
213 cuda_dot_product<<<512, 512>>>(N, cuda_Ap, cuda_Ap, cuda_scalar);
214 cudaMemcpy(&dot_ApAp, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
215
216 beta = (alpha * alpha * dot_ApAp - residual_norm_squared) / residual_norm_squared;
217
218
219
220 int iters = 0;
221 cudaDeviceSynchronize();
222 timer.reset();
223 while (1)
224 {
225
226     cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
227
228     // std::cout << alpha << ", " << residual_norm_squared << std::endl;
229     cuda_cg_1<<<512, 512>>>(N, alpha, beta, cuda_solution, cuda_r, cuda_p, cuda_Ap, cuda_scalar);
230
231     cudaMemcpy(&residual_norm_squared, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
232
233     // std::cout << residual_norm_squared << std::endl;
234
235     cudaMemcpy(cuda_dot_ApAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
236     cudaMemcpy(cuda_dot_pAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
237
238     // std::cout << dot_ApAp << ", " << dot_pAp << ", " << residual_norm_squared << std::endl;
239
240     cuda_cg_2<<<512, 512>>>(N, csr_rowoffsets, csr_colindices, csr_values,
241                             cuda_p, cuda_Ap, cuda_dot_ApAp, cuda_dot_pAp);
242
243     cudaMemcpy(&dot_ApAp, cuda_dot_ApAp, sizeof(double), cudaMemcpyDeviceToHost);

```

```

244     cudaMemcpy(&dot_pAp, cuda_dot_pAp, sizeof(double), cudaMemcpyDeviceToHost);
245     // std::cout << alpha << ", " << residual_norm_squared << std::endl;
246
247     // line 7:
248     alpha = residual_norm_squared / dot_pAp;
249
250     // line 8:
251     beta = (alpha * alpha * dot_ApAp - residual_norm_squared) / residual_norm_squared;
252
253     // std::cout << dot_ApAp << ", " << dot_pAp << std::endl;
254
255     // check for convergence
256     if (std::sqrt(residual_norm_squared / initial_residual_squared) < 1e-6)
257     {
258         break;
259     }
260
261     if (iters > 10000)
262         break; // solver didn't converge
263     ++iters;
264 }
265 cudaMemcpy(solution, cuda_solution, sizeof(double) * N, cudaMemcpyDeviceToHost);
266
267 cudaDeviceSynchronize();
268 /*
269 std::cout << "Time elapsed: " << timer.get() << " (" << timer.get() / iters << " per iteration)" <<
270
271 if (iters > 10000)
272     std::cout << "Conjugate Gradient did NOT converge within 10000 iterations"
273     << std::endl;
274 else
275     std::cout << "Conjugate Gradient converged in " << iters << " iterations."
276     << std::endl;
277 */
278
279 std::cout << timer.get() / iters << ", " << std::endl;
280
281 cudaFree(cuda_p);
282 cudaFree(cuda_r);
283 cudaFree(cuda_Ap);
284 cudaFree(cuda_solution);
285 cudaFree(cuda_scalar);
286 }
287
288 /** Solve a system with 'points_per_direction * points_per_direction' unknowns
289 */
290 void solve_system(int points_per_direction)
291 {
292
293     int N = points_per_direction *
294             points_per_direction; // number of unknowns to solve for
295
296     // std::cout << "Solving Ax=b with " << N << " unknowns." << std::endl;
297
298     //
299     // Allocate CSR arrays.
300     //
301     // Note: Usually one does not know the number of nonzeros in the system matrix
302     // a-priori.
303     // For this exercise, however, we know that there are at most 5 nonzeros
304     // per row in the system matrix, so we can allocate accordingly.
305     //

```



```

306     int *csr_rowoffsets = (int *)malloc(sizeof(double) * (N + 1));
307     int *csr_colindices = (int *)malloc(sizeof(double) * 5 * N);
308     double *csr_values = (double *)malloc(sizeof(double) * 5 * N);
309
310     int *cuda_csr_rowoffsets, *cuda_csr_colindices;
311     double *cuda_csr_values;
312     //
313     // fill CSR matrix with values
314     //
315     generate_fdm_laplace(points_per_direction, csr_rowoffsets, csr_colindices,
316                         csr_values);
317
318     //
319     // Allocate solution vector and right hand side:
320     //
321     double *solution = (double *)malloc(sizeof(double) * N);
322     double *rhs = (double *)malloc(sizeof(double) * N);
323     std::fill(rhs, rhs + N, 1);
324
325     //
326     // Allocate CUDA-arrays //
327     //
328     cudaMalloc(&cuda_csr_rowoffsets, sizeof(double) * (N + 1));
329     cudaMalloc(&cuda_csr_colindices, sizeof(double) * 5 * N);
330     cudaMalloc(&cuda_csr_values, sizeof(double) * 5 * N);
331     cudaMemcpy(cuda_csr_rowoffsets, csr_rowoffsets, sizeof(double) * (N + 1), cudaMemcpyHostToDevice);
332     cudaMemcpy(cuda_csr_colindices, csr_colindices, sizeof(double) * 5 * N, cudaMemcpyHostToDevice);
333     cudaMemcpy(cuda_csr_values, csr_values, sizeof(double) * 5 * N, cudaMemcpyHostToDevice);
334
335     //
336     // Call Conjugate Gradient implementation with GPU arrays
337     //
338     conjugate_gradient(N, cuda_csr_rowoffsets, cuda_csr_colindices, cuda_csr_values, rhs, solution);
339
340     //
341     // Check for convergence:
342     //
343     /*
344     double residual_norm = relative_residual(N, csr_rowoffsets, csr_colindices, csr_values, rhs, solution);
345     std::cout << "Relative residual norm: " << residual_norm
346               << " (should be smaller than 1e-6)" << std::endl;
347     */
348     cudaFree(cuda_csr_rowoffsets);
349     cudaFree(cuda_csr_colindices);
350     cudaFree(cuda_csr_values);
351     free(solution);
352     free(rhs);
353     free(csr_rowoffsets);
354     free(csr_colindices);
355     free(csr_values);
356 }
357
358 int main()
359 {
360     std::vector<int> N_vec = {10, 25, 50, 75, 100, 250, 500, 750, 1000, 1250};
361     for (const auto &N : N_vec)
362     {
363         solve_system(N); // solves a system with 100*100 unknowns
364     }
365
366     return EXIT_SUCCESS;
367 }

```