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.

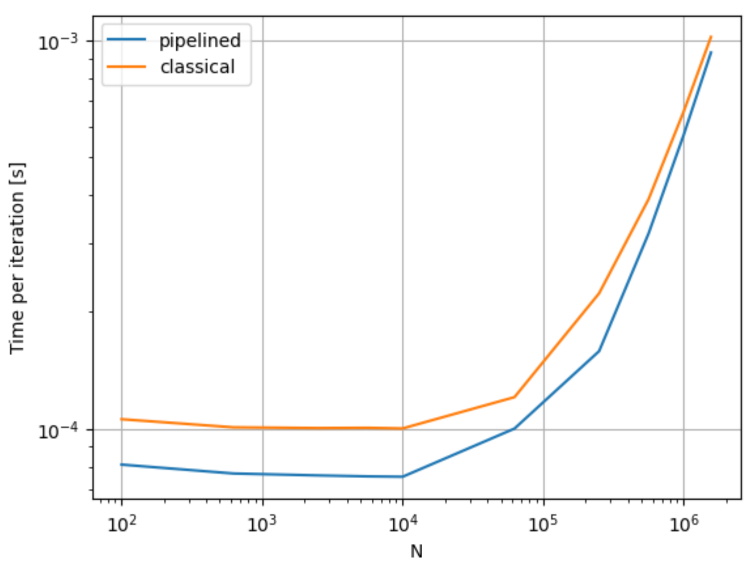

Figure 1: Times per iteration for N between 10^3 and 10^7 for the RTX GPU

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.

```
////////////// CG KERNEL 1: //////////////

__global__ void cuda_cg_1(int N,
                          double alpha,
```

```
                              double beta,
                              double *x,
                              double *r,
                              double *p,
                              double *Ap,
                              double *result_rr)
{

  __shared__ double shared_mem[512];
  double dot = 0;

  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
  {
    // line 2,3,4: get x, r, p
    x[i] = x[i] + alpha * p[i];
    r[i] = r[i] - alpha * Ap[i];
    p[i] = r[i] + beta * p[i];

    // line 6: get dot(r,r)
    dot += r[i] * r[i];
  }

  __syncthreads();
  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_rr, shared_mem[0]);
    // printf("%g", r[0]);
  }
}
```

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.

```
////////////// CG KERNEL 2: //////////////
__global__ void cuda_cg_2(int N,
                              int *csr_rowoffsets,
                              int *csr_colindices,
                              double *csr_values,
                              double *p,
                              double *Ap,
                              double *result1,
                              double *result2)
{

  __shared__ double shared_mem1[512];
  __shared__ double shared_mem2[512];

  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
```

```
  {
    // line 5: get Ap
    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;
  }

  // line 6: get dot(Ap,Ap) and dot(p,Ap)
  double dot1 = 0;
  double dot2 = 0;
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
  {
    dot1 += Ap[i] * Ap[i];
    dot2 += p[i] * Ap[i];
  }

  shared_mem1[threadIdx.x] = dot1;
  shared_mem2[threadIdx.x] = dot2;
  for (int k = blockDim.x / 2; k > 0; k /= 2)
  {
    __syncthreads();
    if (threadIdx.x < k)
    {
      shared_mem1[threadIdx.x] += shared_mem1[threadIdx.x + k];
      shared_mem2[threadIdx.x] += shared_mem2[threadIdx.x + k];
    }
  }

  if (threadIdx.x == 0)
  {
    atomicAdd(result1, shared_mem1[0]);
    atomicAdd(result2, shared_mem2[0]);
  }
}
```

Appendix

Task 1.b)

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

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

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

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

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

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

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

```

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

////////////// CG KERNEL 1: //////////////

__global__ void cuda_cg_1(int N,
                          double alpha,
                          double beta,
                          double *x,
                          double *r,
                          double *p,
                          double *Ap,
                          double *result_rr)
{

  __shared__ double shared_mem[512];
  double dot = 0;

  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
  {
    // line 2,3,4: get x, r, p
    x[i] = x[i] + alpha * p[i];
    r[i] = r[i] - alpha * Ap[i];
    p[i] = r[i] + beta * p[i];

    // line 6: get dot(r,r)
    dot += r[i] * r[i];
  }

  __syncthreads();
  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_rr, shared_mem[0]);
    // printf("%g", r[0]);
  }
}

////////////// CG KERNEL 2: //////////////
__global__ void cuda_cg_2(int N,
                          int *csr_rowoffsets,
                          int *csr_colindices,
                          double *csr_values,
                          double *p,
                          double *Ap,
                          double *result1,
                          double *result2)
{

  __shared__ double shared_mem1[512];
  __shared__ double shared_mem2[512];
```

```

  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
  {
    // line 5: get Ap
    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;
  }

  // line 6: get dot(Ap,Ap) and dot(p,Ap)
  double dot1 = 0;
  double dot2 = 0;
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
  {
    dot1 += Ap[i] * Ap[i];
    dot2 += p[i] * Ap[i];
  }

  shared_mem1[threadIdx.x] = dot1;
  shared_mem2[threadIdx.x] = dot2;
  for (int k = blockDim.x / 2; k > 0; k /= 2)
  {
    __syncthreads();
    if (threadIdx.x < k)
    {
      shared_mem1[threadIdx.x] += shared_mem1[threadIdx.x + k];
      shared_mem2[threadIdx.x] += shared_mem2[threadIdx.x + k];
    }
  }

  if (threadIdx.x == 0)
  {
    atomicAdd(result1, shared_mem1[0]);
    atomicAdd(result2, shared_mem2[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, residual_norm_squared, dot_pAp, dot_ApAp;
```

```
double *cuda_solution, *cuda_p, *cuda_r, *cuda_Ap, *cuda_scalar, *cuda_dot_pAp, *cuda_dot_ApAp;
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));

cudaMalloc(&cuda_dot_ApAp, sizeof(double));
cudaMalloc(&cuda_dot_pAp, 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);

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

double initial_residual_squared = residual_norm_squared;

// line 1: get alpha0, beta0, Ap0
cuda_csr_matvec_product<<<512, 512>>>(N, csr_rowoffsets, csr_colindices, csr_values, cuda_p, cuda_A

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

cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);
cuda_dot_product<<<512, 512>>>(N, cuda_Ap, cuda_Ap, cuda_scalar);
cudaMemcpy(&dot_ApAp, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);

beta = (alpha * alpha * dot_ApAp - residual_norm_squared) / residual_norm_squared;



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

  cudaMemcpy(cuda_scalar, &zero, sizeof(double), cudaMemcpyHostToDevice);

  // std::cout << alpha << ", " << residual_norm_squared << std::endl;
  cuda_cg_1<<<512, 512>>>(N, alpha, beta, cuda_solution, cuda_r, cuda_p, cuda_Ap, cuda_scalar);

  cudaMemcpy(&residual_norm_squared, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);

  // std::cout << residual_norm_squared << std::endl;

  cudaMemcpy(cuda_dot_ApAp, &zero, sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(cuda_dot_pAp, &zero, sizeof(double), cudaMemcpyHostToDevice);

  // std::cout << dot_ApAp << ", " << dot_pAp << ", " << residual_norm_squared << std::endl;

  cuda_cg_2<<<512, 512>>>(N, csr_rowoffsets, csr_colindices, csr_values,
                          cuda_p, cuda_Ap, cuda_dot_ApAp, cuda_dot_pAp);

  cudaMemcpy(&dot_ApAp, cuda_dot_ApAp, sizeof(double), cudaMemcpyDeviceToHost);
```

```
    cudaMemcpy(&dot_pAp, cuda_dot_pAp, sizeof(double), cudaMemcpyDeviceToHost);
    // std::cout << alpha << ", " << residual_norm_squared << std::endl;

    // line 7:
    alpha = residual_norm_squared / dot_pAp;

    // line 8:
    beta = (alpha * alpha * dot_ApAp - residual_norm_squared) / residual_norm_squared;

    // std::cout << dot_ApAp << ", " << dot_pAp << std::endl;

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

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

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

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

  std::cout << timer.get() / iters << "," << std::endl;

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

/** 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, soluti
  std::cout << "Relative residual norm: " << residual_norm
            << " (should be smaller than 1e-6)" << std::endl;
  */
  cudaFree(cuda_csr_rowoffsets);
  cudaFree(cuda_csr_colindices);
  cudaFree(cuda_csr_values);
  free(solution);
  free(rhs);
  free(csr_rowoffsets);
  free(csr_colindices);
  free(csr_values);
}

int main()
{
  std::vector<int> N_vec = {10, 25, 50, 75, 100, 250, 500, 750, 1000, 1250};
  for (const auto &N : N_vec)
  {
    solve_system(N); // solves a system with 100*100 unknowns
  }

  return EXIT_SUCCESS;
}
```