Vienna University of Technology

360.252 Computational Science on Many Core Architectures

Institute for Microelectronics

Exercise 7

Authors:
Camilo Tello Fachin
12127084

Supervisor:
Dipl.-Ing. Dr.techn. Karl Rupp

December 11, 2022

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Abstract

Here documented the results of Exercise 7.

Contents

1 Pipelined Conjugate Gradients (5/5 Points)

1.1 Implement the Algorithm (3/3 Points)

In the listings below the Kernels for the blue and red colored font from lecture slides 7a. The task was to only execute 2 Kernel call's per iteration, with these two Kernels, this can be achieved.

1.1.1 Blue Font Algorithm Part

Kernel for Blue Font Algorithm Part

```
__global__ void cuda_blue(int N, double *x, double *p, double *Ap, double *r, double *r_ip, double Alpha,
    double Beta)
{
  __shared__ double shared_memory[512];
  double partial_dot_product = 0;

  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x){
    double p_thread = p[i];
    double Ap_thread = Ap[i];
    double r_thread = r[i] - Alpha * Ap_thread;
    x[i] += Alpha * p_thread;
    r[i] = r_thread;
    p[i] = r_thread + Beta * p_thread;
     partial_dot_product += r_thread *r_thread;
  }
  shared_memory[threadIdx.x] = partial_dot_product;
  for (int j = blockDim.x / 2; j > 0; j /= 2) {
    __syncthreads();
    if (threadIdx.x < j) {
      shared_memory[threadIdx.x] += shared_memory[threadIdx.x + j];
  }

  if (threadIdx.x == 0) r_ip[blockIdx.x] = shared_memory[0];
  }
}
```

1.1.2 Red Font Algorithm Part

Kernels for Red Algorithm Part

```
__global__ void cuda_blue(int N, double *x, double *p, double *Ap, double *r, double *r_ip , double Alpha, double Beta)
{
  __shared__ double shared_memory[512];
  double partial_dot_product = 0;
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x){
    double p_thread = p[i];
    double Ap_thread = Ap[i];
    double r_thread = r[i] - Alpha * Ap_thread;
    x[i] += Alpha * p_thread;
    r[i] = r_thread;
    p[i] = r_thread + Beta * p_thread;
     partial_dot_product += r_thread *r_thread;
  }
  shared_memory[threadIdx.x] = partial_dot_product ;
  for (int j = blockDim.x / 2; j > 0; j /= 2) {
    __syncthreads();
    if (threadIdx.x < j) {
      shared_memory[threadIdx.x] += shared_memory[threadIdx.x + j];
  }

  if (threadIdx.x == 0) r_ip[blockIdx.x] = shared_memory[0];
  }
}
```

1.1.3 Bringing Everything together

In the fuction `conjugate_gradient_pipelined()` we use both the Kernels from before and iterate till convergence. In order for this algorithm to work, it needs initial α_0, β_0 and also an Ap_0, this is done before the actual iterations in the `while()` loop. One can use the old functions from the last exercise like the dot product and the matrix vector product to obtain the two values and the vector. In the listing below shown the `while()` loop that iterates till convergence.

CG `while()` loop

```
int iters = 0;
cudaDeviceSynchronize();
timer.reset();
while (1) {
  // Line 2-4 and partial of line 6, The blue colored part in the algorithm:
  cuda_blue<<<blocks_Inch,thrds_block>>>(N, cuda_solution, cuda_p, cuda_Ap, cuda_r, cuda_ip_rr, alpha,
      beta);
  // cudaMemcopy for blockwise inner product <r_i, r_i> for CPU calculations of alpha and beta
  cudaMemcpy(bwip_rr, cuda_ip_rr, sizeof(double) * blocks_Inch, cudaMemcpyDeviceToHost);
  // Line 5 and 6, the red colored part in the algorithm
  cuda_red<<<blocks_Inch,thrds_block>>>(N, csr_rowoffsets, csr_colindices, csr_values, cuda_p, cuda_Ap,
      cuda_ip_ApAp, cuda_ip_pAp);
  // cudaMemcopy for blockwise inner product <Ap_i, Ap_i> and <p_i, Ap_i> for CPU calculations of alpha
      and beta
  cudaMemcpy(bwip_pAp, cuda_ip_pAp, sizeof(double) * blocks_Inch, cudaMemcpyDeviceToHost);
  cudaMemcpy(bwip_ApAp, cuda_ip_ApAp, sizeof(double) * blocks_Inch, cudaMemcpyDeviceToHost);
  // CPU summation of blockwise inner products.
  ip_rr = bwip_rr[0];
  ip_ApAp = bwip_ApAp[0];
  ip_pAp = bwip_pAp[0];
  for(size_t i=1; i<blocks_Inch; ++i)
  {
    ip_rr += bwip_rr[i];
    ip_ApAp += bwip_ApAp[i];
    ip_pAp += bwip_pAp[i];
  }
  //Check if convergence criterion is fulfilled .
  if (std::sqrt(ip_rr / initial_residual_squared) < 1e-6) {
    break;
  }
  // Computation of alpha and beta for next while iteration .
  alpha = ip_rr / ip_pAp;
  beta = (alpha*alpha*ip_ApAp - ip_rr) / ip_rr;

  if (iters > 10000)
    break; // solver didn't converge
  ++iters;
}
```

1.2 Comparison of Pipelined CG and Classical CG on both GPU's (2/2 Points)

1.2.1 CG Algorithm Performance on RTX3060

For this specific implementation, the runtime for the pipelined CG algorithm on the RTX3060 outperforms the classic one only for $\sqrt{N}$ larger than approximately 300. After this point, the imporvements of the pipelined algorithm appear to be only minor.

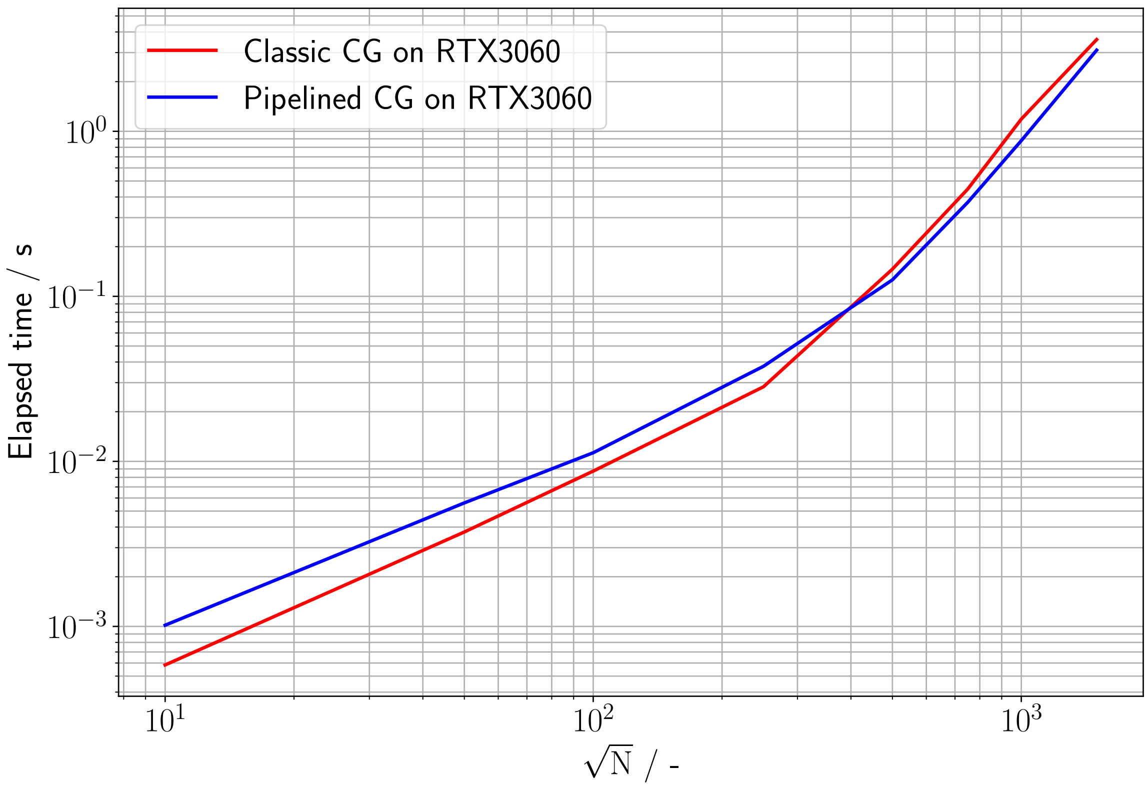

Figure 1: Elapsed time for Classical and Pipelined CG Algorithm on the RTX3060 GPU

1.2.2 CG Algorithm Performance on Tesla K40

For this specific implementation, the runtime for the pipelined CG algorithm on the Tesla K40 outperforms the classic one only for $\sqrt{N}$ larger than approximately 350. After this point, the imporvements of the pipelined algorithm appear to be only minor.

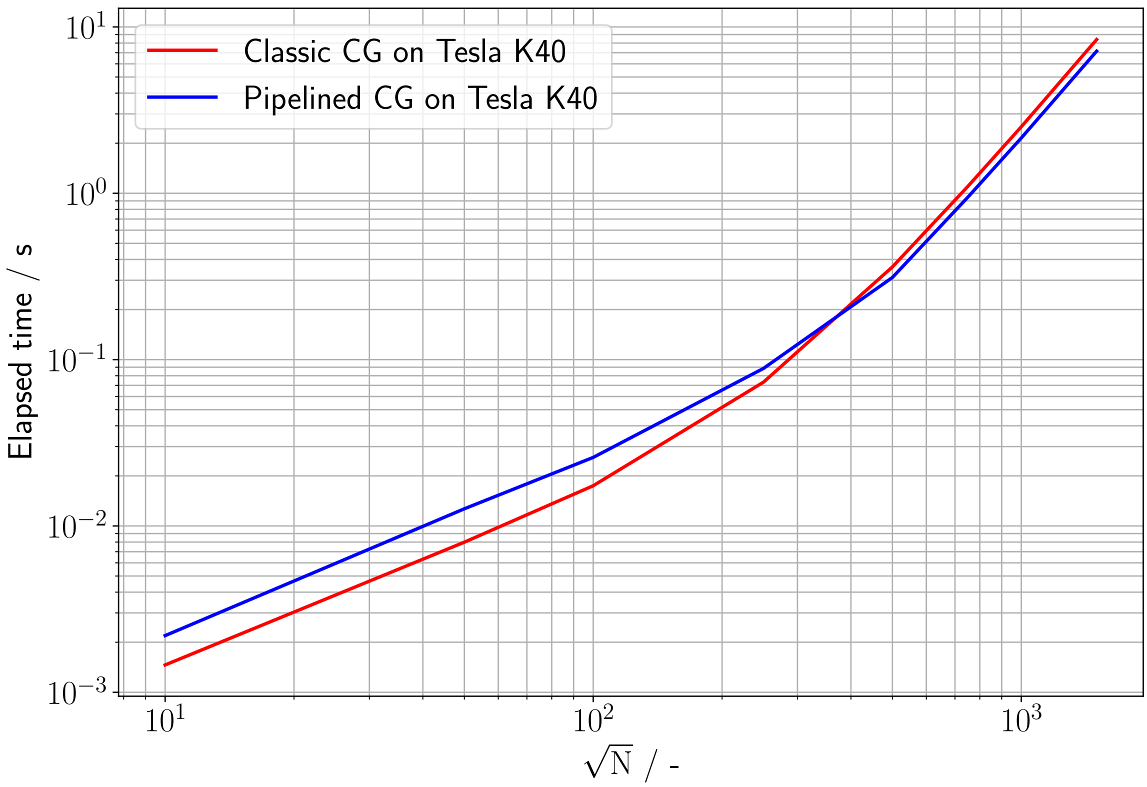

Figure 2: Elapsed time for Classical and Pipelined CG Algorithm on the Tesla K40 GPU