Vienna University of Technology

360.252 Computational Science on Many Core Architectures

Institute for Microelectronics

Exercise 5

Authors:
Camilo Tello Fachin
12127084

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

November 22, 2022

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Abstract

Here documented the results of exercise 5, that was quite fun actually!

Contents

1 Performance Modeling: Parameter Identification (5/5 Points)

1.1 Task 1a - PCI Express latency for `cudaMemcpy()` in μs (1 Point)

In order to find the latencies for the `K40 Tesla` and the `RTX 3060` GPU's, the assumption is made that timings one is able to measure with `timer.hpp` and `<iostream>` the following quantities: a latency ℓ, a constant time for `cudaMemcpy()`'ing one `double` called $T_{\texttt{double}}$, two timings one measures T_1 and T_2 and an arbitrary positive integer N to `cudaMemcpy()` a vector of doubles of length N.

$$
\begin{aligned}
\ell + T_{\texttt{double}} &= T_1 \\
\ell + \mathrm{N} \cdot T_{\texttt{double}} &= T_2 \\
\rightarrow \ell &= \frac{T_2 - T_1 \cdot \mathrm{N}}{1 - \mathrm{N}}
\end{aligned}
$$

With this model assumption one can obtain the latency ℓ for both devices. The calculation of the above equation can be seen in the listing below in line 28.

C++ Cuda Code Obtaining the latency

```
1  int N = 1e7;
2  double *x = (double *)malloc(sizeof(double));
3  double *x_N = (double *)malloc(sizeof(double) * N);
4  *x = 1;
5  std::fill(x_N, x_N + N, 1);
6  double *cuda_x, *cuda_x_N;
7  cudaMalloc(&cuda_x, sizeof(double));
8  cudaMalloc(&cuda_x_N, sizeof(double) * N);
9
10 std::vector<double> timings_double(100, 0.0);
11 std::vector<double> timings_N_doubles(100, 0.0);
12
13 for (size_t i=0; i<=100; ++i){
14   timer.reset();
15   cudaMemcpy(cuda_x, x, sizeof(double), cudaMemcpyHostToDevice);
16   cudaDeviceSynchronize();
17   timings_double[i] = timer.get();
18 }
19 double timing_double = findMedian(timings_double, 100);
20
21 for (size_t i=0; i<=100; ++i){
22   timer.reset();
23   cudaMemcpy(cuda_x_N, x_N, sizeof(double)*N, cudaMemcpyHostToDevice);
24   timings_N_doubles[i] = timer.get();
25   cudaDeviceSynchronize();
26 }
27 double timing_N_doubles = findMedian(timings_N_doubles, 100);
28 double latency = (timing_N_doubles - timing_double*N)/(1-N);
```

`PCI Express gen3 Latency on RTX3060:  3.99926` μs

`PCI Express gen3 Latency on K40 TESLA: 8.9987` μs

1.2 Task 1b - Kernel Launch Latency in μs (1 Point)

Task 1b is relatively straight forward, just launch a high number of empty kernels with `<<<1,1>>>` and find the median time!

C++ Cuda Code for Kernel Launch Latency

```
__global__ void cuda_5_1b()
{
// Kennt's ihr eh Spiegeldondi? - Mahatma Ghandi, ca. 1940 - idea of an empty Kernel..
}
int main() {

Timer timer;
std::vector<double> timings(100, 0.0);

for (size_t i=0; i<100; ++i){
  timer.reset();
  cuda_5_1b<<<1, 1>>>();
  cudaDeviceSynchronize();
  timings[i] = timer.get();
  cudaDeviceSynchronize();
}
double latency = findMedian(timings, 100);
std::cout << "Kernel Launch Latency: " << latency << std::endl;
```

`Kernel Launch Latency on RTX3060:  10` μs

`Kernel Launch Latency on K40 TESLA: 5` μs

1.3 Task 1c - Practical Peak Memory Bandwidth in GB/s (1 Point)

The Numbers for the obtained Peak Memory Bandwidth are in the legend of Figure 1 and the relevant code chunks in the listing below.

C++ Cuda Code for Peak Memory Bandwidth

```
__global__ void cuda_5_1c(double *x, double *y, double *z, int N)
{
  unsigned int total_threads = blockDim.x * gridDim.x;
  unsigned int global_tid = blockIdx.x * blockDim.x + threadIdx.x;
  for (unsigned int i = global_tid ; i<N; i += total_threads) {
    z[i] = x[i] + y[i];
  }


}

for(int j=0; j < median_int; j++){
  cudaDeviceSynchronize();
  timer.reset();
  cuda_5_1c<<<((N_vec[i]+255)/256), 256>>>(gpu_x, gpu_y, gpu_z, N_vec[i]);
  cudaDeviceSynchronize();
  timings.push_back(timer.get());
}

peak_bw.push_back((3*N_vec[i]*sizeof(double)*pow(10,-9))/findMedian(timings, median_int));
timings.clear();
std::cout << N_vec[i] << ", " << peak_bw[i] << std::endl;
```

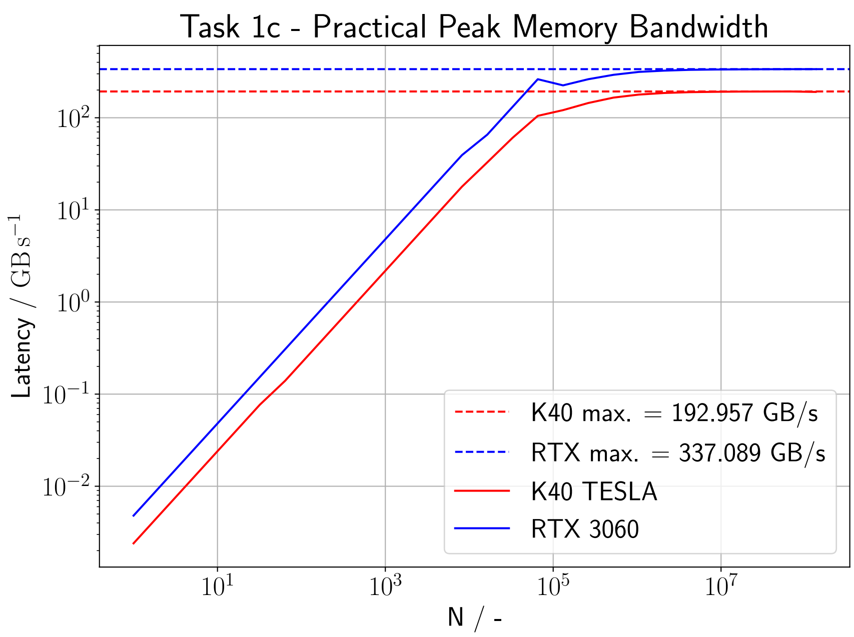

Figure 1: Results for Task 1c

1.4 Task 1d - Maximum Number of `atomicAdd()` / s

The Numbers for the obtained `atomicAdd()`'s per second are in the legend of Figure 2 and the relevant code chunks in the listing below.

C++ Cuda Code for Maximum number of `atomicAdd()`

```
__global__ void cuda_5_1d(double *atomic_add_result)
{
  atomicAdd(atomic_add_result, threadIdx.x);
}

    for (size_t i=0; i<N_max; ++i){

        for(int j=0; j < median_int; j++){
            cudaDeviceSynchronize();
            timer.reset();
            cuda_5_1d<<<(int)N_vec[i], 1>>>(x_gpu);
            cudaDeviceSynchronize();
            timings.push_back(timer.get());
        }

    median_timings.push_back(findMedian(timings, median_int));
    timings.clear();
    std::cout << N_vec[i] << "," << N_vec[i]/median_timings[i] << std::endl;
    }

    return EXIT_SUCCESS;
}
```

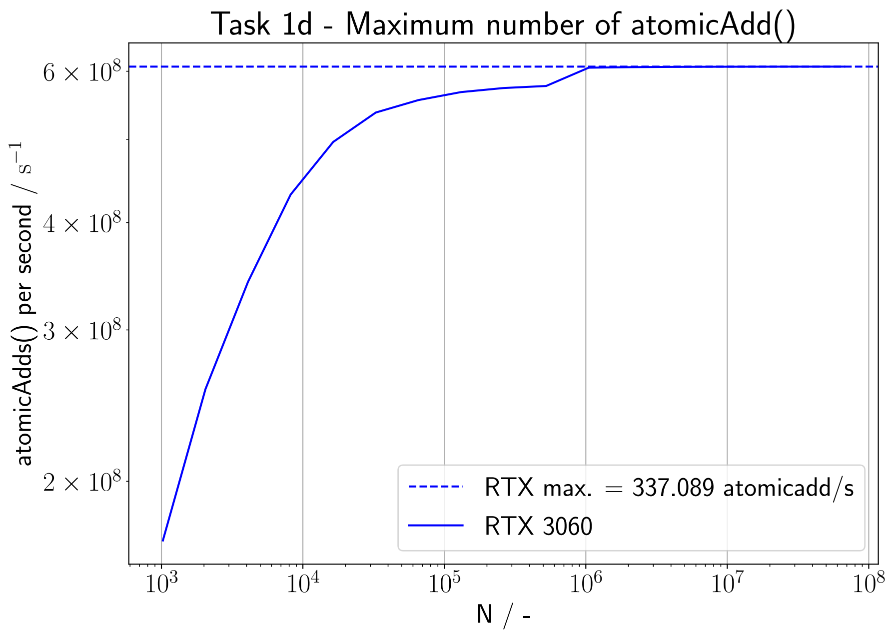

Figure 2: Results for Task 1d

1.5 Task 1e - Peak FLOP's (i.e. $\alpha + = \beta \cdot \gamma$) for `double`'s as GFLOPs/s (1 Point)

The Numbers obtained for the Peak Floating Point Rate for both GPU's are in the legend of figure 3 and the relevant code chunks in the listing below.

C++ Cuda Code Maximal Floating Point Rate

```
__global__ void cuda_5_1c(double *x, double *y, double *z, int N)
{
  float a = x[blockIdx.x*blockDim.x + threadIdx.x];
  float b = y[blockIdx.x*blockDim.x + threadIdx.x];
  float c;
  for (int i = 0; i < 8*3000; i++) {
    c += a * b;
  }
  z[blockIdx.x*blockDim.x + threadIdx.x] += c;
}

for ( size_t i=0; i<N_max-N_min; ++i){
  for(int j=0; j < median_int; j++){
    cudaDeviceSynchronize();
    timer.reset();
    cuda_5_1c<<<((N_vec[i]+255)/256), 256>>>(gpu_x, gpu_y, gpu_z, N_vec[i]);
    cudaDeviceSynchronize();
    timings.push_back(timer.get());
  }
  peak_flops.push_back((2*8*3000*N_vec[i]* pow(10, -9))/findMedian(timings, median_int));
  timings.clear();
  std::cout << N_vec[i] << "," << peak_flops[i] << std::endl;
```

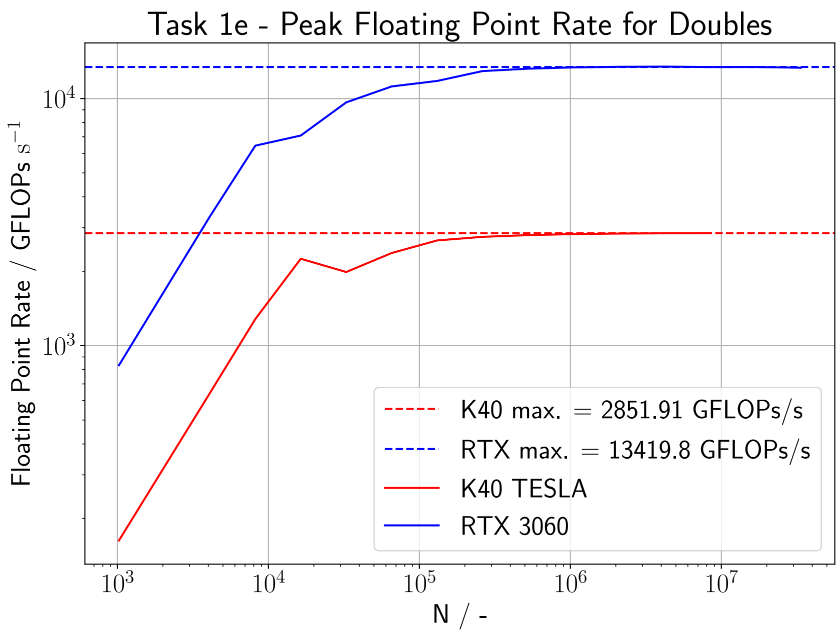

Figure 3

2 Conjugate Gradients (5/5 Points)

2.1 Task 2a - A CUDA Kernel for the Matrix-Vector Product (1 Point)

Below the listing with the relevant code chunk for Task 2a.

C++ Cuda Code for 1a Kernel

```
__global__ void CUDA_csr_matvec_product(size_t N, int * csr_rowoffsets , int * csr_colindices , double *
    csr_values , double *x, double *y)
{
  for (int row = blockDim.x * blockIdx.x + threadIdx.x; row < N; row += gridDim.x * blockDim.x)
  {
    double val = 0;
    for (int jj = csr_rowoffsets [row]; jj < csr_rowoffsets [row + 1]; ++jj)
    {
      val += csr_values[ jj ] * x[ csr_colindices [ jj ]];
    }
    y[row] = val;
  }
}
```

2.2 Task 2b - CUDA Kernels for Vector Operations (2 Points)

Below the listings with the relevant code chunks for Task 2b.

C++ Cuda Code for 1a Kernel

```
__global__ void CUDA_dot_product(int N, double *x, double *y, double *result)
{
  __shared__ double shared_mem[1024];
  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 vectorAdd_Kernel_7(int N, double *x, double *y, double a)
{
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
  {
    x[i] += a * y[i];
  }
}

__global__ void vectorAdd_Kernel_8(int N, double *x, double *y, double a)
{
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
  {
    x[i] -= a * y[i];
  }
}

__global__ void vectorAdd_Kernel_12(int N, double *x, double *y, double a)
{
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
  {
    x[i] = y[i] + a * x[i];
  }
}
```

2.3 Task 2c - Convergence Behaviour of CUDA Cojungate Gradient (1 Point)

Plots for Task 2!

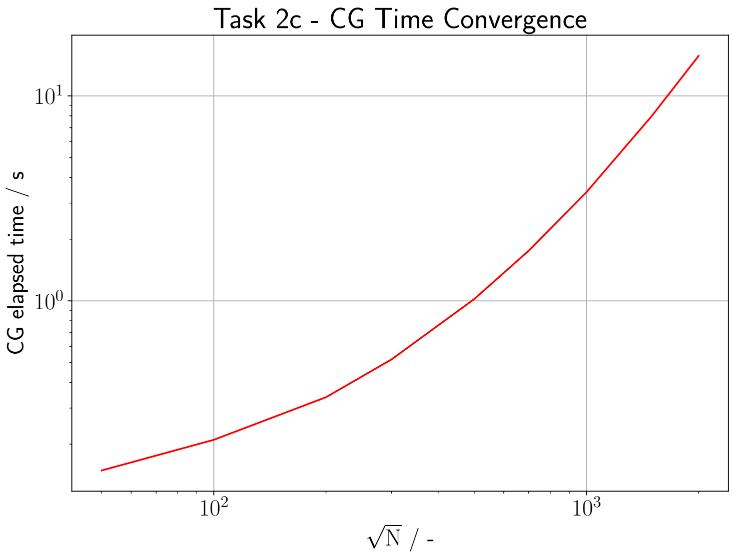

Figure 4: Partial Results for Task 2c

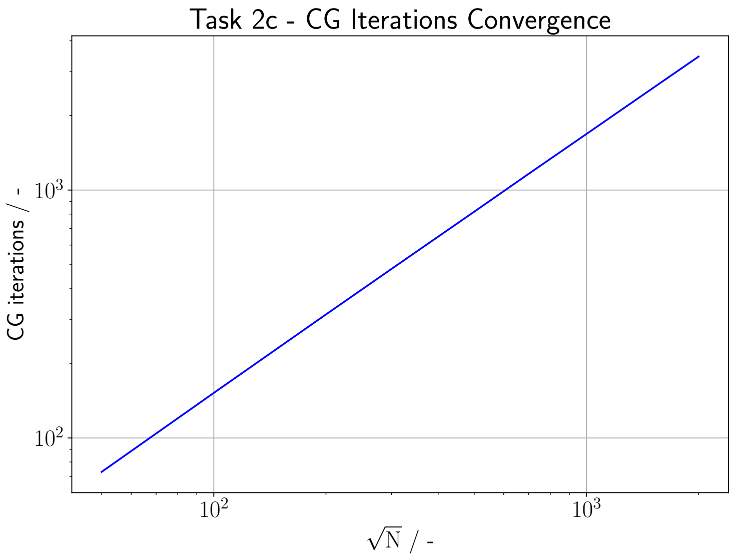

Figure 5: Partial Results for Task 2c

2.4 Task 2d - Which Parts are worthwile optimizing? (1 Point)

The addition kernels seem worthwhile optimizing!

C++ Console output for sqrt(N)

```
CG converged after 3454 iterations .
CUDA_csr_matvec_product: 0.000002
CUDA_dot_product: 0.000253
vectorAdd_Kernel_7: 0.000288
vectorAdd_Kernel_8: 0.000291
vectorAdd_Kernel_12: 0.000288
Relative residual norm: 1.56973e-07 (should be smaller than 1e-6)
t = 16.063496
```