Computational Science on Many-Core Architectures
Exercise 5

Leon Schwarzäugl

November 27, 2023

The code for all tasks can be found at: `https://github.com/Swarsel/CSE_TUWIEN/tree/main/WS2023/Many-Core%20Architectures/e5`

1 Performance Modeling: Parameter Identification

1.1 a)

For measuring PCIE latency of cudaMemcpy, I copied a very small package of data (1 double). I repeated the measurement 100 times to average a bit better - I would have done this way more often, but I did not want to clutter the online environment with that many requests. Also from the initial times I omitted the lowest and highest 10 times each before averaging - I think because the current load on the online environment was high, there were some "exceptional" runs in the data that were possibly caused by the server lagging.
Also, this measurement is possibly not all too accurate; this is because I am not sure how exactly cudaMemcpy() works here - surely for each call we have a latency that should be constant, and then a term dependant on the data. I could have assumed here that that latter term stays constant for increasing N, in which case I could have made a second measurement run with a bigger amount of data, then solving the resulting system for this dependant term and as such calculate the latency. But since I have no idea if that dependency is actually linear (I have the feeling it is not) I just used this more simple approach.
My findings concluded in the PCIE latency for cudaMemcpy being $\approx$ **12**μs. This is roughly in line with the data from the lecture slides, with the actual value possibly being a bit smaller due to the above.

1.2 b)

For measuring the latency of a CUDA kernel call, I tried to model the launch of an "empty kernel", such as

```
__global__ void empty() {}
```

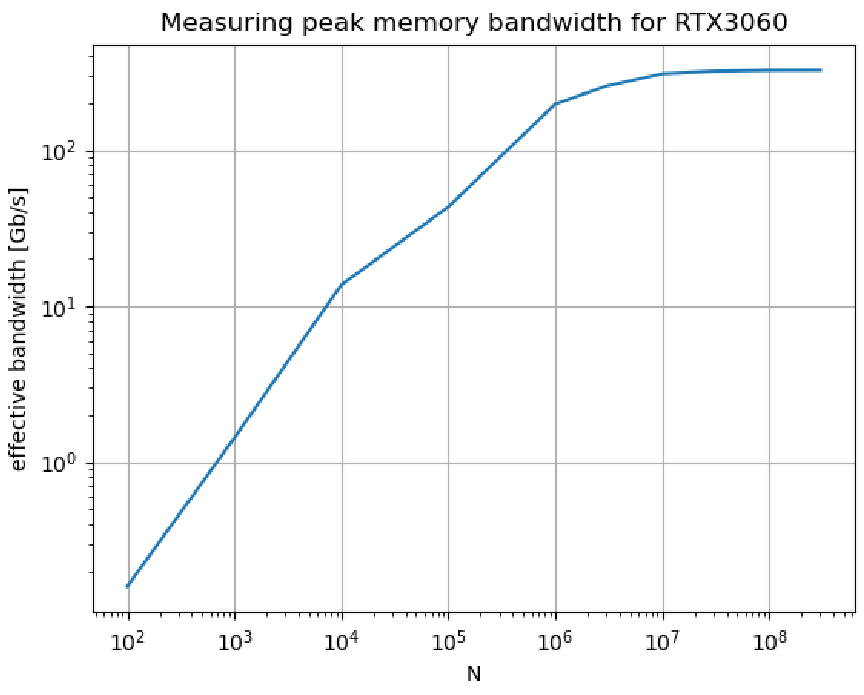
This has one problem in that the compiler will optimize the calling of this kernel away. The way that I decided to tackle this with was by initializing a volatile variable within the kernel:

```
__global__ void empty() {
volatile int a = 0;
}
```

This should result in the call not being optimized because the volatile flag tells the compiler that it might be accessed from placed that are out of scope on compile level. I received a launch latency of $\approx$ **0,01**s. This is way beyond the expected $\approx 10\mu$s. I am not completely sure why. I ran the using my "anti-optimized" version as well as with a completely empty kernel for checking, both resulted in these times.

1.3 c)

For measuring the practical peak bandwidth, I basically repeated what we already did in earlier exercises - saturating the kernel with a high enough system size for a kernel call - I used here a simple copy kernel. For added accuracy, I also ran a reference kernel that runs through the same instructions apart from the copying step and subtracted that from the actual kernel run time, to minimize the amount of overhead that got included in the data.



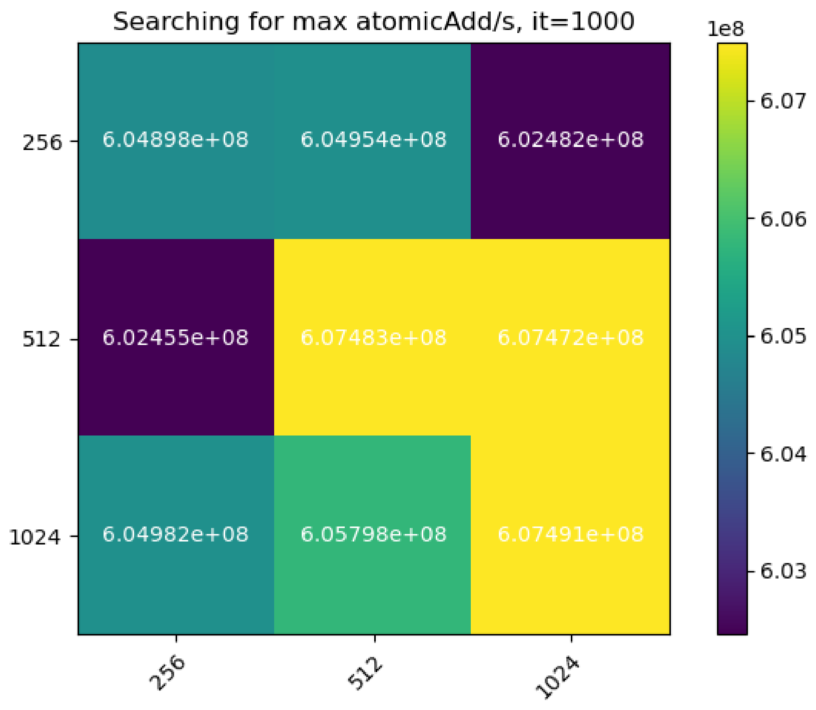
We can see that we reach the peak bandwidth at about $N = 10^7$, where we receive $\approx \mathbf{330}\frac{\mathrm{Gb}}{\mathrm{s}}$.

1.4 d)

For measuring the maximum amount of atomicAdds to a single address, I used a simple kernel that lets each thread call an atomicAdd to that address multiple times:

```
__global__
void atomicAddKernel(double *dest, int it){
    for (unsigned int i = 0; i<it; i ++) {
        atomicAdd(dest,1);
    }
}
```

The total amount of atomic adds was then devided by the time taken, which results in a max amount of about $\mathbf{6 \cdot 10^8}$ atomicAdds per second. I checked this for several constellations of grid- and blocksizes because I was interested if there would be any significant change between them. The results are quite uniform however.



1.5 e)

Lastly I measured the peak floating point rate using multiply-add for testing. The kernel used was

```
    __global__
void copyKernel(int N, double src1, double src2){
  volatile double res = 0;
  for(int i = 0; i < 1000; i ++){
    res += src1 * src2;
  }
}
```

To this kernel I passed two random constants and added to a volatile double - again in hopes of avoiding optimizations, as that result is never used. I tried to avoid unnecessary memory accesses here as much as possible, also again I subtracted the runtime of a reference kernel. The peak rates were obtained for a high grid size of 4096 and the max block size of 1024, and resulted in about 170GFlOPs/s

2 Conjugate Gradients

2.1 a)

The matrix-vector product was implemented akin to the lecture slides as:

```
    __global__
void csr_matvec_product(int N, int *rowoffsets, int *colindices, const double *values, doubl
    for (int row = blockDim.x * blockIdx.x + threadIdx.x; row < N; row += gridDim.x * blockD
        double val = 0;
        for (int jj = rowoffsets[row]; jj < rowoffsets[row+1]; ++jj) {
            val += values[jj] * x[colindices[jj]];
        }
        y[row] = val;
    }
}
```

2.2 b)

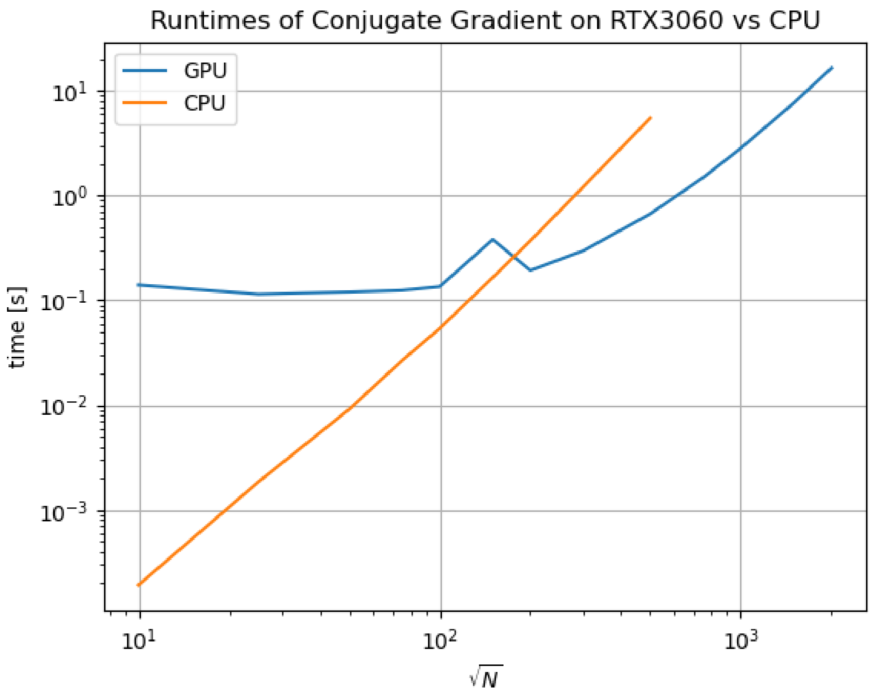
These kernels were conglomerated into two kernels, with the dot kernel basically just reduced from last exercise:

```
    __global__ void dot(int N, double *x, double *y, double *results) {
    double alpha1{0};
    for(int j = blockIdx.x * blockDim.x + threadIdx.x; j < N; j += blockDim.x*gridDim.x) {
        alpha1 += x[j] * y[j];
    }

    for (int j=warpSize/2; j>0; j=j/2) {
        alpha1 += __shfl_xor_sync(0xffffffff, alpha1, j);
    }

    if (threadIdx.x % warpSize == 0) {
        atomicAdd(results, alpha1);
    }
}

__global__
void vecIterate(int N, double *out, double *in1, double *in2, double mod) {
        for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim
            out[i] = in1[i] + mod * in2[i];
        }
}
```

dot (...) takes care of the dot products, while vecIterate (...) takes care of the kernel calls in lines 7, 8, and 12.

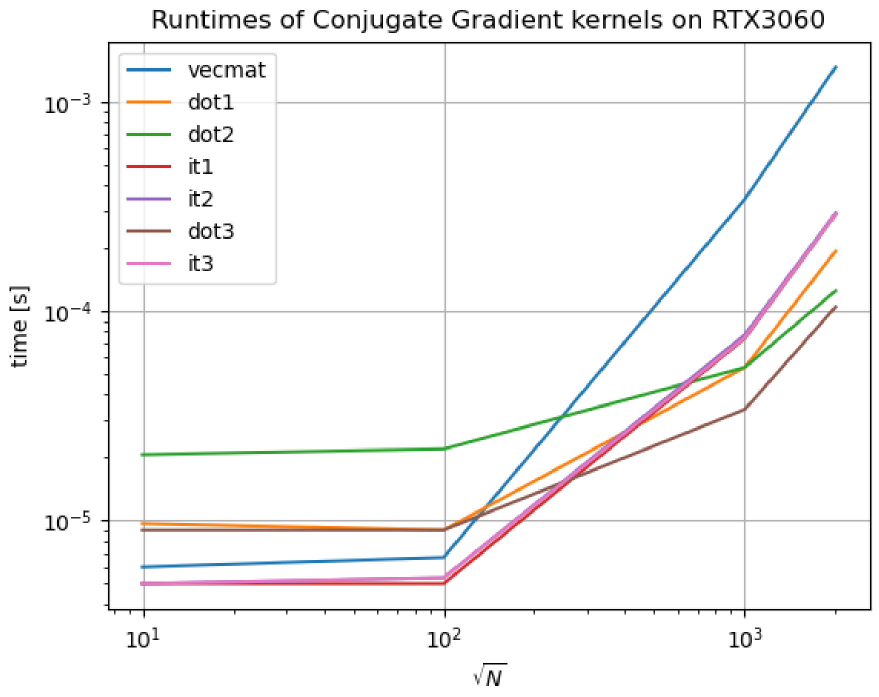
2.3 c)

I compared the time of my implementation to the pure-CPU implementation - the GPU becomes faster than the CPU version quite swiftly. There is one weird jump in the plot, where my approach of discarding some of the lowest and highest times combined with averaging still had one outlier in. Because there were so many calls to the online environment, I only did 7 runs for each N here - of course more would have been better, but I wanted to get this done more quickly. The bump should be interpreted as a measuring error.



2.4 d)

Here are the runtimes of each individual kernel:



We can see the dot products are a bit slower in comparison for small systems, but are then the fastest for bigger systems. The matrix-vector multiplication has the steepest ascent for big system sizes, and should be further optimized when mainly working on those. The vecIterate kernels start the fastest but also become more expensive quickly with rising system sice and should then also be further optimized.

3 Bonus Point: The Need for Speed

I optimized the kernel by omitting all the settings of CPU variables for the dot product; instead I set it to 0 within the kernels such as

```
__global__ void dot(int N, double *x, double *y, double *results) {
double alpha1{0};
unsigned int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
if (thread_id == 0) *results = 0;
[...]
```

I checked different grid/block-sizes, but was not really able to find combination that was throughout superior to the standard $<<< 256, 256 >>>$. As a last

optimization, I computed lines 7 and 8 in the same kernel.
Sadly these optimizations did not lead to a significant decrease in runtime as compared to the "unoptimized" version (both versions however save on one calculation of pAp (the dot product of p with Ap) per iteration already, by calculating it initially once before the while-loop, and then using the updated value next, so that is one optimization that is not tracked here).