Heidelberg University
Institute for Computer Engineering (ZITI)

Master of Science Computer Engineering

GPU Computing

Exercise 5

gpucomp03

Benjamin Maier
Daniel Barley
Laura Nell

Due date December 15th, 09:00

5.1 Reading

The GPU Coputing Era

Because of the still increasing demand for faster and higher-definition graphics, the development of increasingly parallel and programmable GPUs has not stopped yet. Using the example of Nvidia GPUs, the evolution of GPU computing and its parallel computing model as well as the benefit of CPU+GPU coprocessing are described with proof by example application performance speedups. In particular, the focus of the, at that time, state of the art architectures lies on the one hand on the CUDA scalable parallel architecture, on the other hand, regarding computing architectures, on the Fermi scalable computing architecture.

The Fermi computing architecture enables a raise in throughput by several novel features like its streaming multiprocessors introducing a memory model using fast on-chip shared memory or ECC memory protection.
The main conclusion is, that CPU+GPU coprocessing enables much more efficient execution of almost all kinds of algorithms by using the CPU, which is latency optimized, for serial parts of the code and the GPU, which is throughput optimized, for parallel parts of the code. This fits perfectly for the CUDA programming model, which is able to launch a series of parallel kernels from a single sequential control thread.

The CUDA programming model is still widely used in today's applications due to its specialization on parallel computing. It is also still state of the art to extend CPU+GPU coprocessing for time consuming applications, although with further developed computing architectures. Therefore, we accept the paper and its content.

5.2 Matrix Multiply - GPU naive version

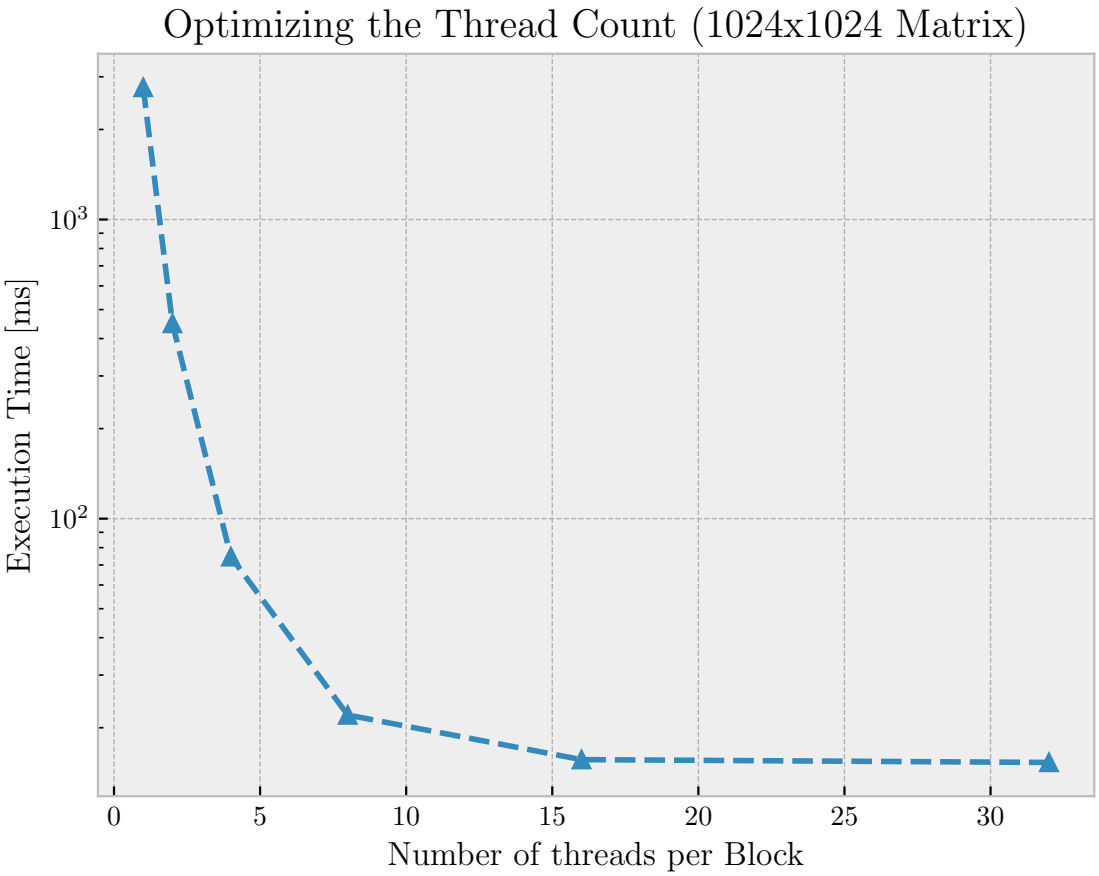

Figure 1: The thread per block count was varied between 1 and 32. The best performance is unsurprisingly achieved when using the maximum of 32 threads per block. This is in line with previous observations that less threads doing smaller tasks limits performance significantly.

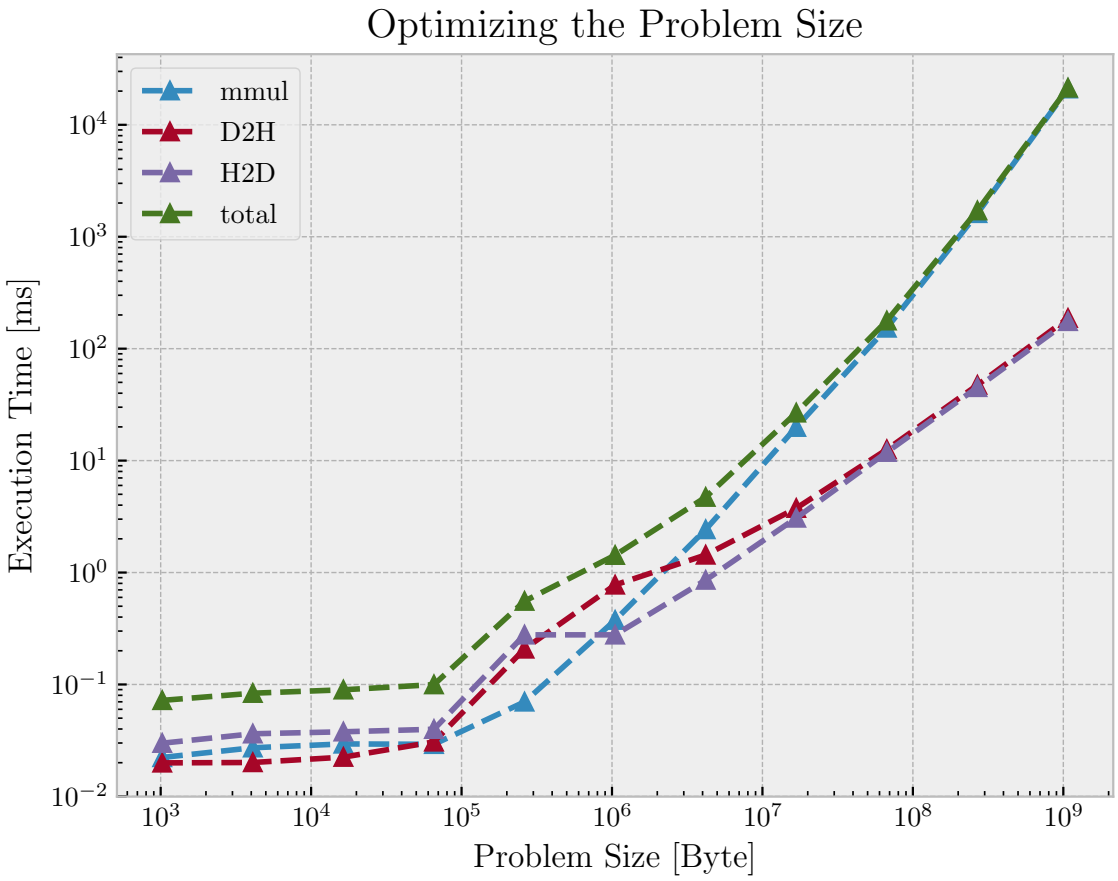

Figure 2: Using the optimal count of 32 threads the problem size was varied and the latencies due to data movement and computation compared. For smaller problem sizes data movement is far too expensive. Only when reaching a few mega bytes the computation amortizes the movement of data

5.3 Matrix Multiply - GPU version using shared memory

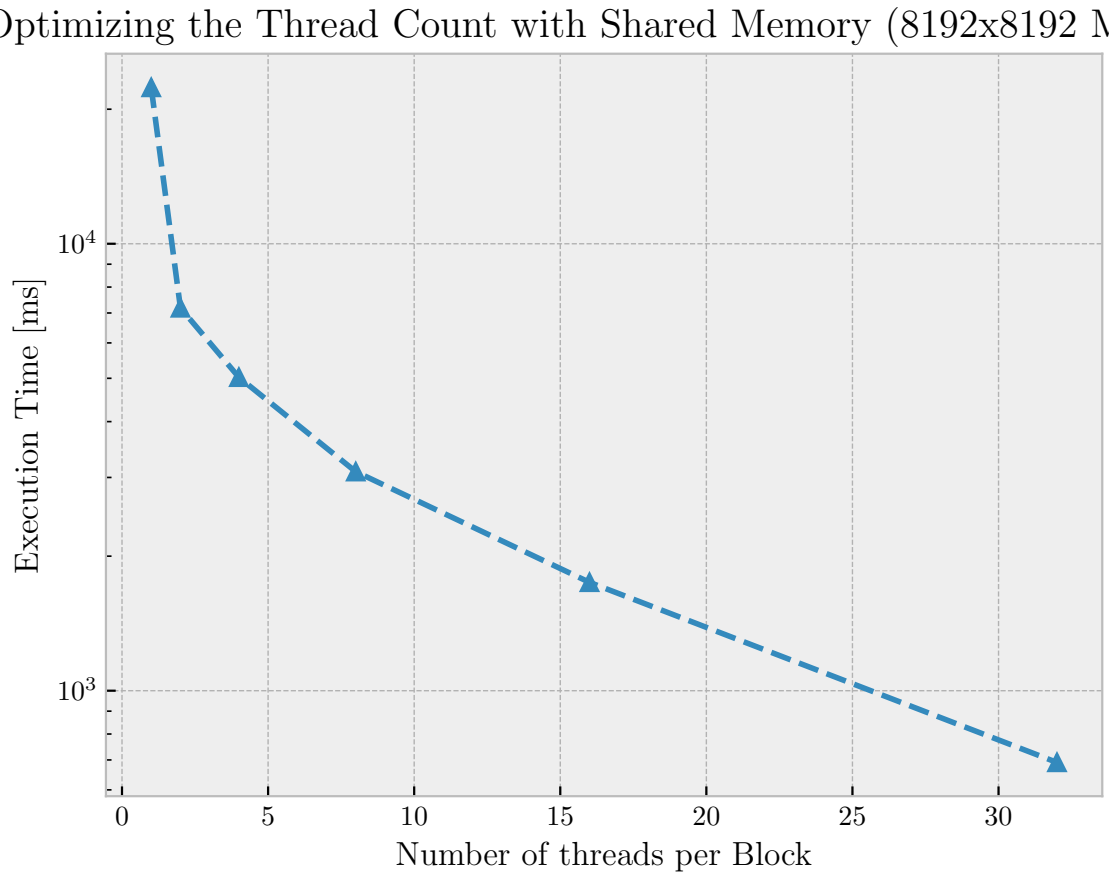

Figure 3: The same as in the previous task was done for the shared memory optimized version. Again the maximum number of threads is optimal. Additionally to the computation more threads mean a faster collaborative load into shared memory.

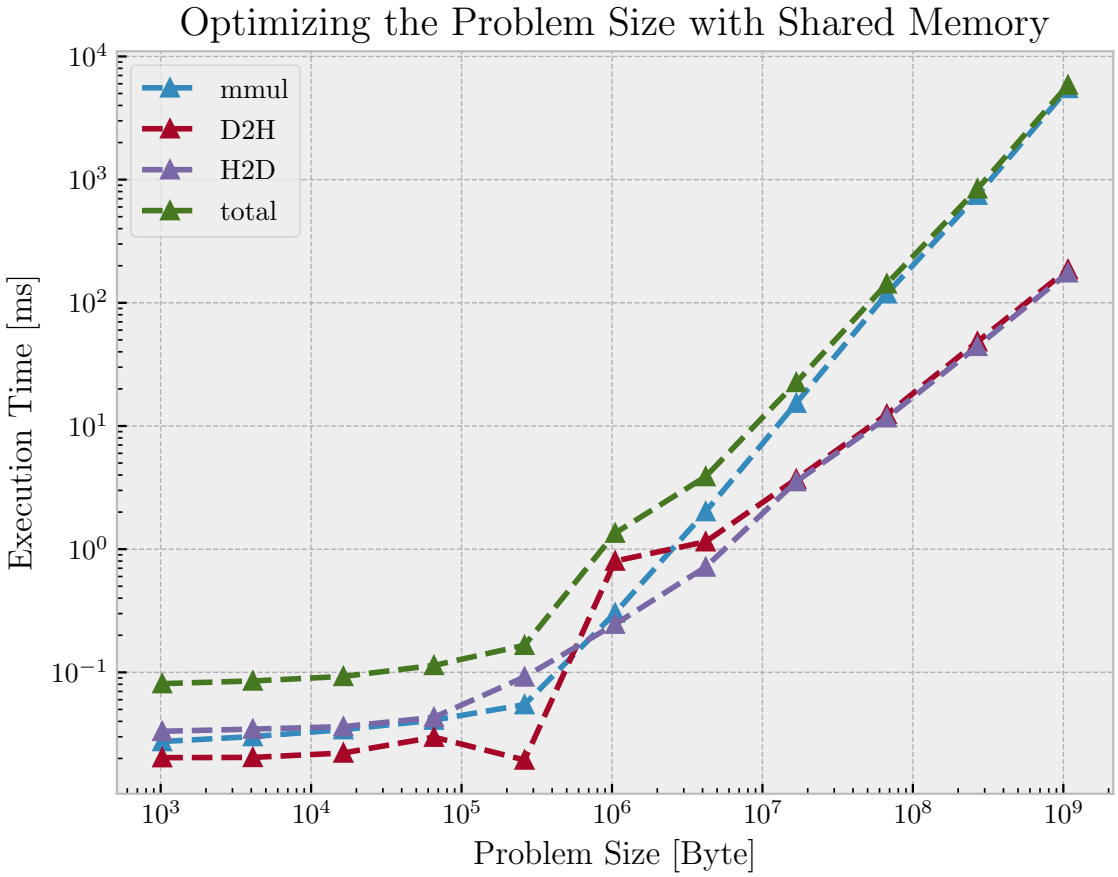

Figure 4: Same as the naïve version

Table 1: Speedups acheived for 1024x1024 (largest value tested on CPU)

Program Version	Speedup
unoptimized excl. data movement	301
unoptimized inc. data movement	154
optimized excl. data movement	361
optimized inc. data movement	188

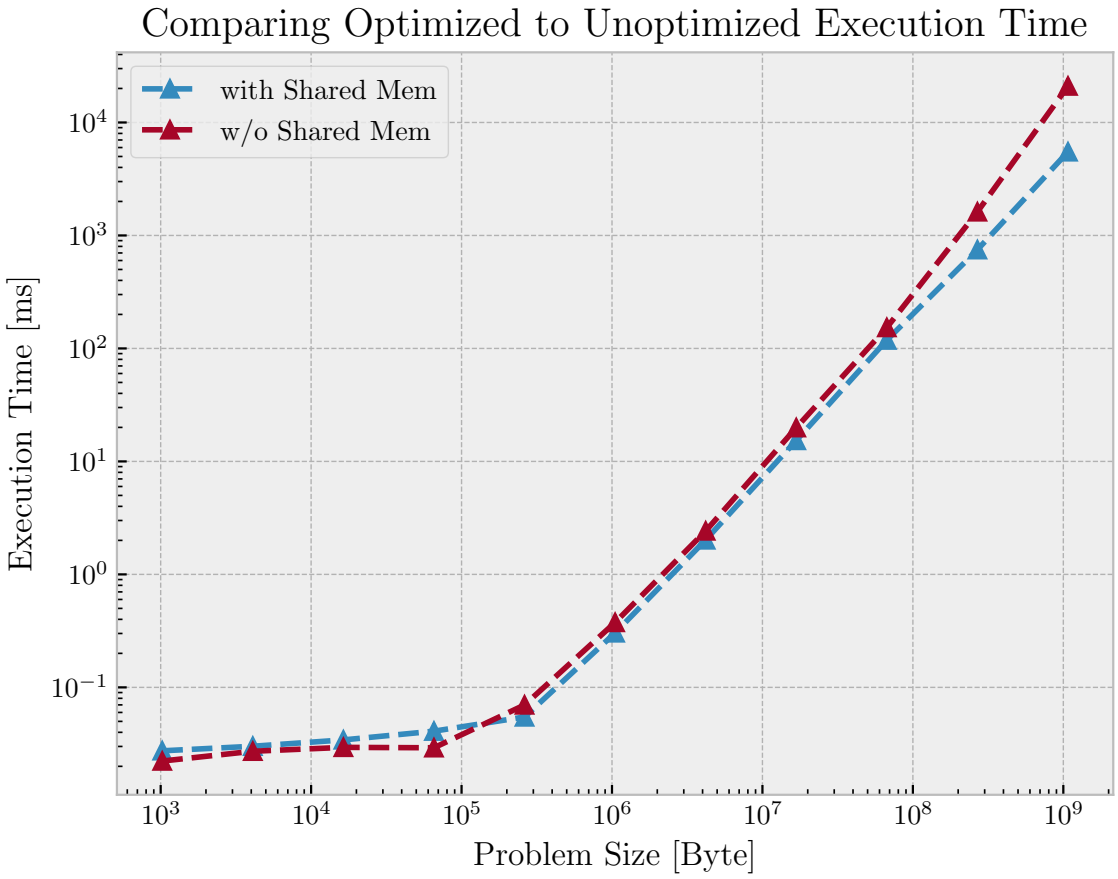

Figure 5: Comparing execution times of the naïve and optimized versions reveals that for small problem sizes there is no notable difference. The naïve version even out performs the shared memory version probably due to the data movement and synchronization overhead introduced.

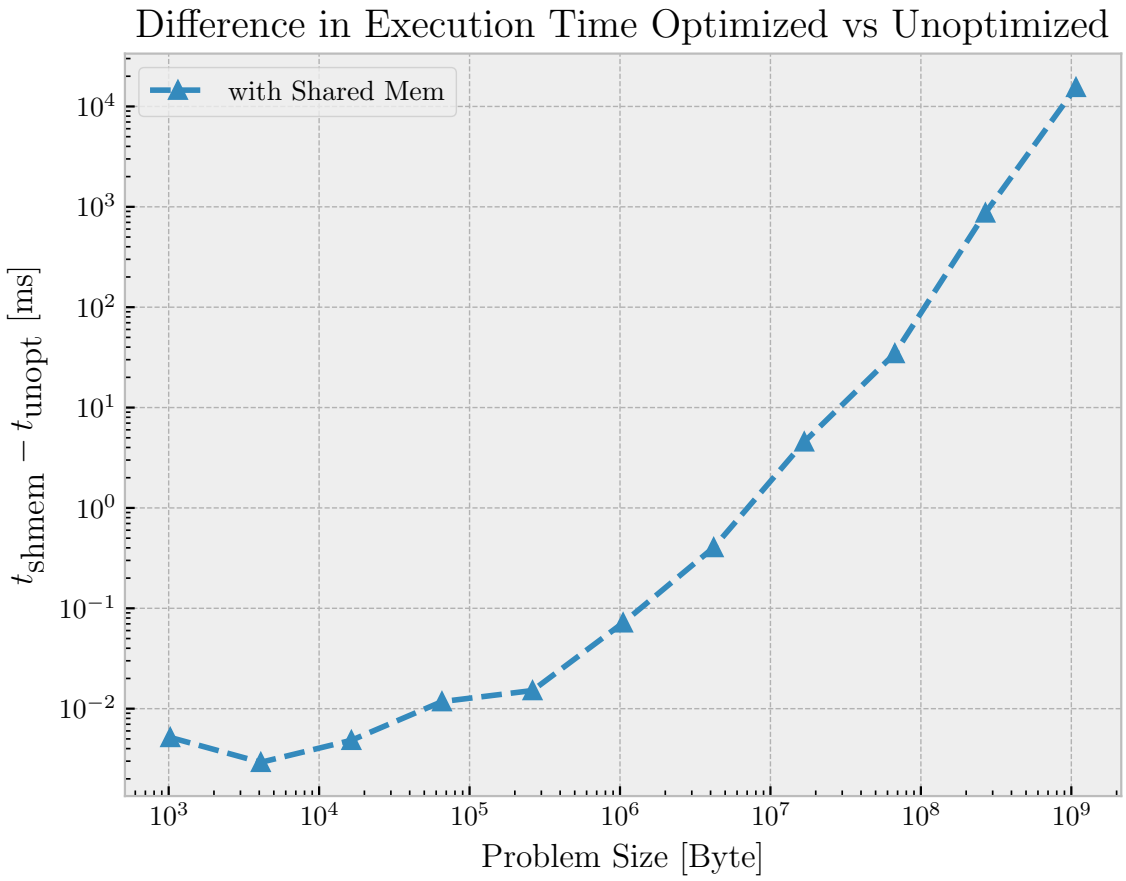

Figure 6: Difference in execution time visualized. The shared memory optimized matrix multiply starts to outperform the naïve version in the hundreds of mega bytes.

5.4 Willingness to present

Hereby, we declare our will to present the results presented in the former sections.