Heidelberg University
Institute for Computer Engineering (ZITI)

Master of Science Computer Engineering

GPU Computing

Exercise 4

gpucomp03

Benjamin Maier
Daniel Barley
Laura Nell

Due date December 1st, 09:00

4.1 Reading

Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU

Because of the massive amount of generated data, it is important for throughput computing applications that handle with these data to have a reasonable amount of data parallelism. Such applications can be realized for CPUs or GPUs. Because of their different architectures, they perform different for different throughput computing workloads and have different possibilities for optimizations. Thus, 14 throughput computing kernels with different characteristics are compared based on their execution on the Intel Core i7 CPU and the Nvidia GTX280 GPU.

The performance of the several kernels depends highly on their requirements and the corresponding architecture of the devices like bandwidth limitations, compute flops, cache size or synchronization.
Typically the highest performance is achieved when a suitable number of multiple threads are used per core.
Nevertheless, the performance gap between CPU and GPU can be decreased a lot (down to an average of 2.5X) by individual optimizations of software and hardware on both devices. Main optimizations for CPUs leading to performance improvements are multithreading, cache blocking and reorganization of memory accesses for SIMDfication. Optimizations that lead to GPU performance improvements are minimizing global synchronization and using local shared buffers.
Besides that, the key hardware architecture features which lead to an imporved performance were identified.

The main characteristics of performance differences still fit for todays architectures, while the overall performance and functionality of CPUs and GPUs has actually improved. Thus, we accept the paper and its content.

NVIDIA Tesla: A Unified Graphics and Computing Architecture

To enable flexible, programmable graphics and high-performance computing, NVIDIA has developed the Tesla scalable unified graphics and parallel computing architecture, which extends the functionality of traditional graphics pipelines. The goal for that architeture was to unify vertex and pixel processors in one architecture.
The developed architecture itself along with the resulting (parallel) workflows is described in the paper before an overview of the parallel programming model in C with CUDA tools is given.

There are many innovations in the Tesla architecture, including the execution of scalar instructions or the SIMT processor architecture, which enables executing threads grouped into several warps and increases the efficiency a lot.
Regarding the programming model, the Tesla architecture introduces cooperative thread arrays or CUDA thread blocks to ensure the correct execution of a large number of concurrent threads.
The Tesla architecture was developed for scalability also with large problem sizes. Thus, GPUs using this architecture deliver high performance also for demanding parallel-computing and

graphics applications.

Nvidia was one major driver in the development of GPUs and corresponding architectures as highly parallel general purpose computers. Thus, also the Tesla architecture had, at this time, some great new improvements for graphics processing and lead to further innovations. We therefore fully accept the paper and its content.

4.2 Shared Memory Analysis - Basis

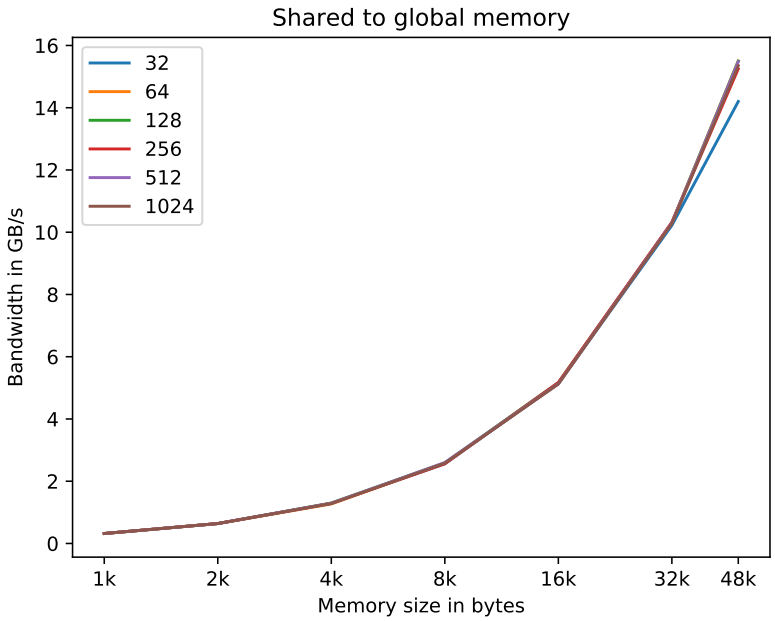

Figure 1: Global to shared memory

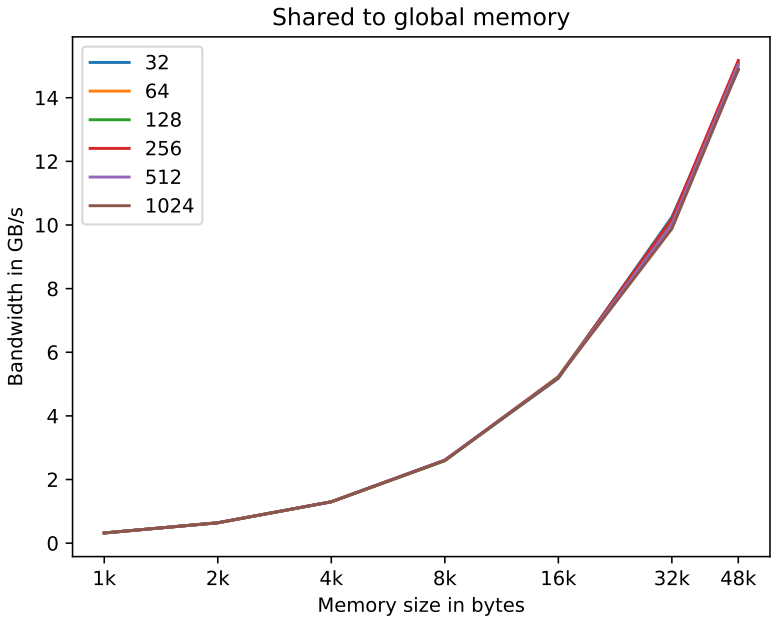

Figure 2: Shared to global memory

For the basic versions (figure 1 and 2) we could not find any influence of the amount of threads. However, we decided to start with a thread amount of 32, since below that we would not fully utilize the GPU.

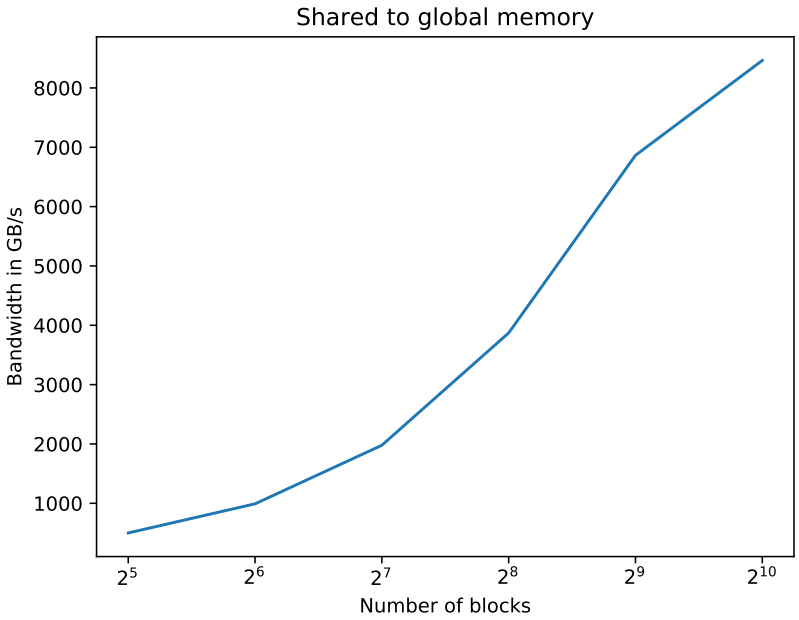

Figure 3: Global to shared memory

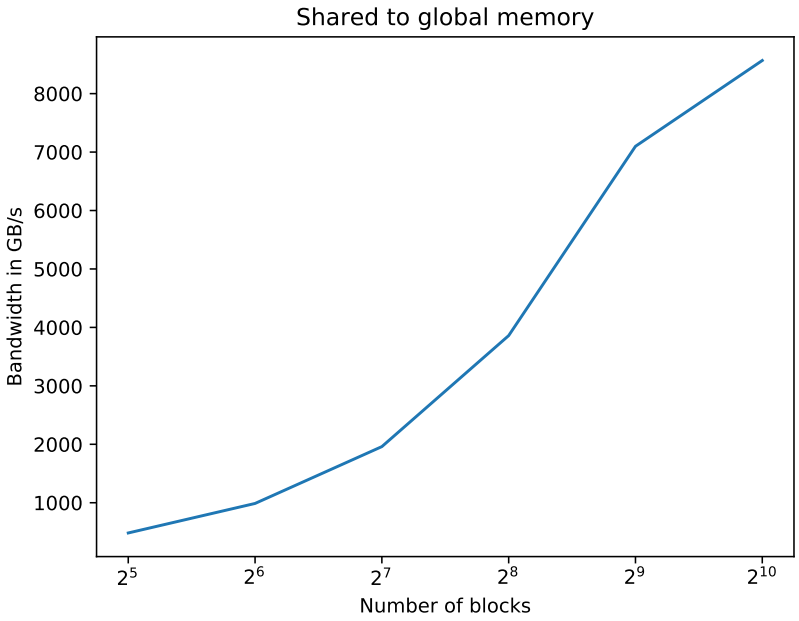

Figure 4: Shared to global memory

For the next task we should vary the number of blocks. In figure 3 and 4 one can see that the throughput scales linear with the amount of blocks. The reason for this is that every block has its own amount of shared memory and so the throughput gets added.

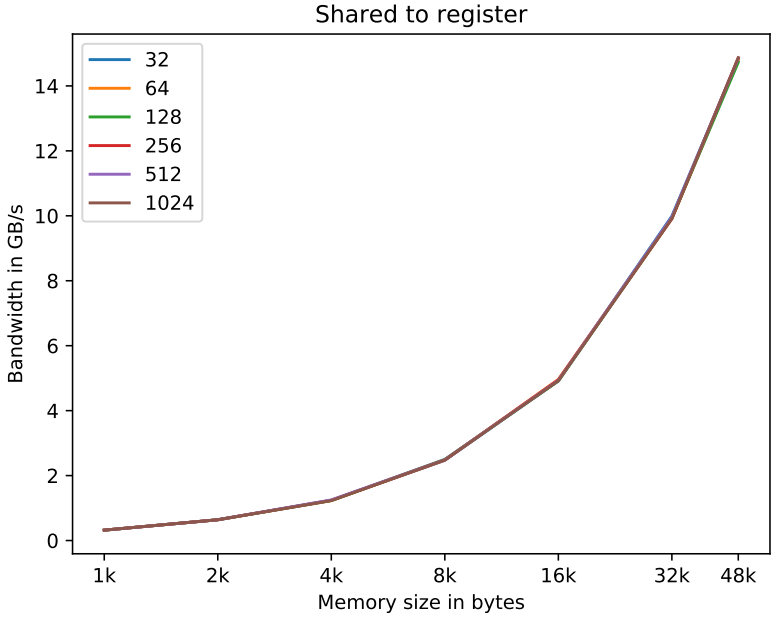

Figure 5: Global to shared memory

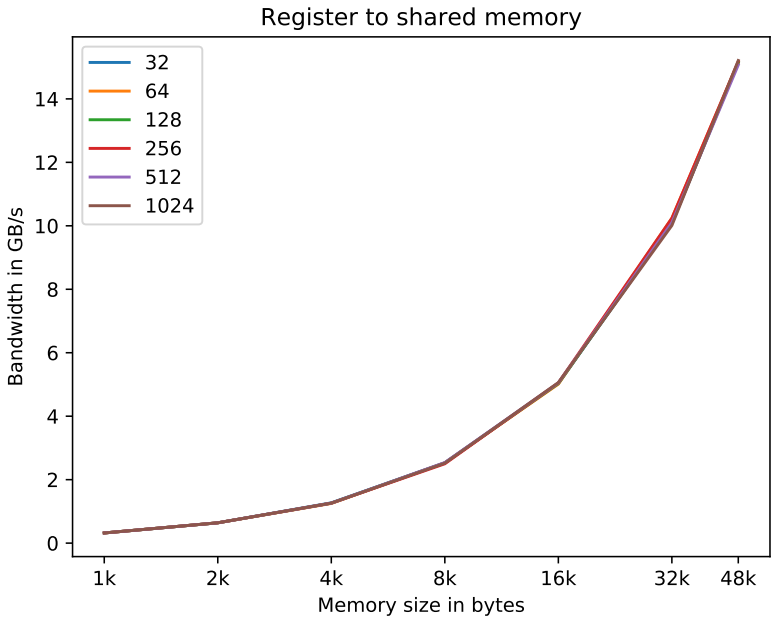

Figure 6: Shared to global memory

Finally we should measure a simple shared memory to register transfer. The results are shown in figure 5 and 6. Again we cannot see a big difference.

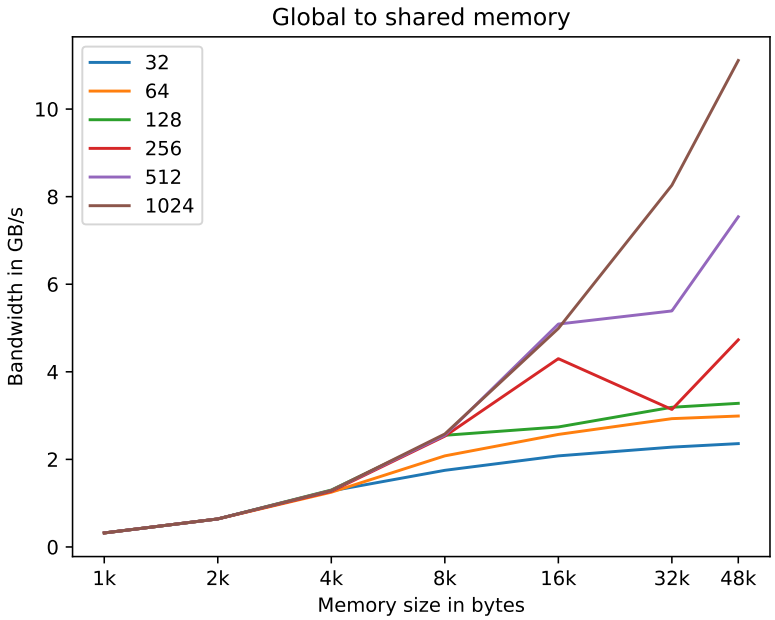

Figure 7: Global to shared memory with consecutively access pattern.

Here at this point we would like to add some remarks according to the code. General

spoken there are two main methods to access the data inside the kernel: (i) consecutively inside a thread, (ii) strided across the threads inside a block. The benefit of the second method is the better usage of the L1 cache. To verify this we repeated the first measurement with the first method. The results are shown in figure 7. According to the results, we decided to use the second access pattern for the rest of the exercise. Maybe this is the reason why our measurements show similar results.

4.3 Shared Memory Analysis - Conflicts

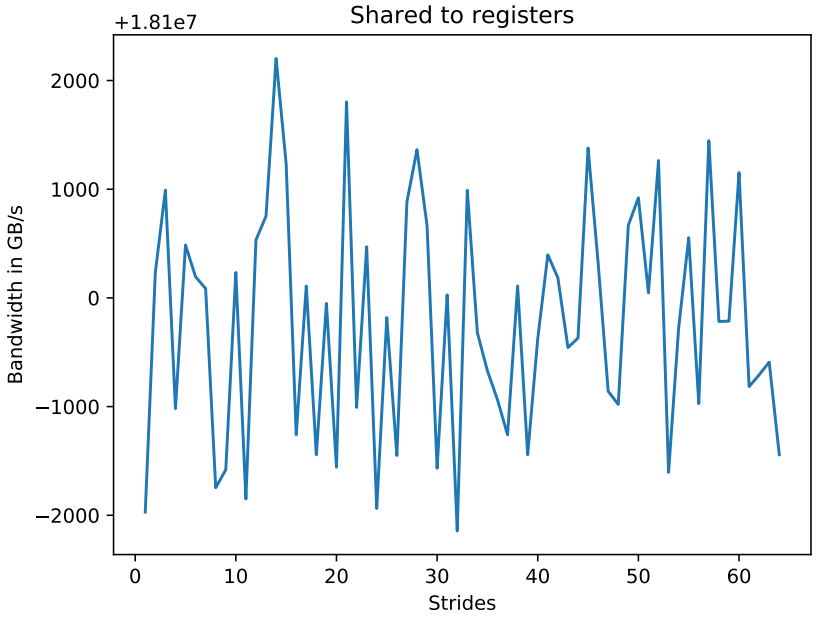

Figure 8: Bank conflicts

In the next exercise, we should try to measure the impact of bank conflicts. However, as you can see in figure 8, we cannot find a impact of the strided access pattern. Maybe there is a error in our code. Normally you would expect, that with a multiple of the bank size, you get alot of conflicts and therefore have a higher clock count.

4.4 Matrix Multiply - CPU Sequential Version

The Code for this section can be found in the `matrix_multiply` subfolder and compiled by calling `make` inside the folder. This results in the `mmul.out` binary which takes two parameters: `-s <N>` which produces a $N \times N$ matrix and `-p <0 || 1>` which dis/enables printing of the results.

4.4.1 Verification

When initialized as $A[i,j] = i + j$ and $B[i,j] = i \cdot j$ the multiplication yields:

$$\begin{bmatrix} 0 & 30 & 60 & 90 & 120 \\ 0 & 40 & 80 & 120 & 160 \\ 0 & 50 & 100 & 150 & 200 \\ 0 & 60 & 120 & 180 & 240 \\ 0 & 70 & 140 & 210 & 280 \end{bmatrix} \tag{1}$$

4.4.2 Performance

The measurements are taken on an AMD Ryzen 5 3600.

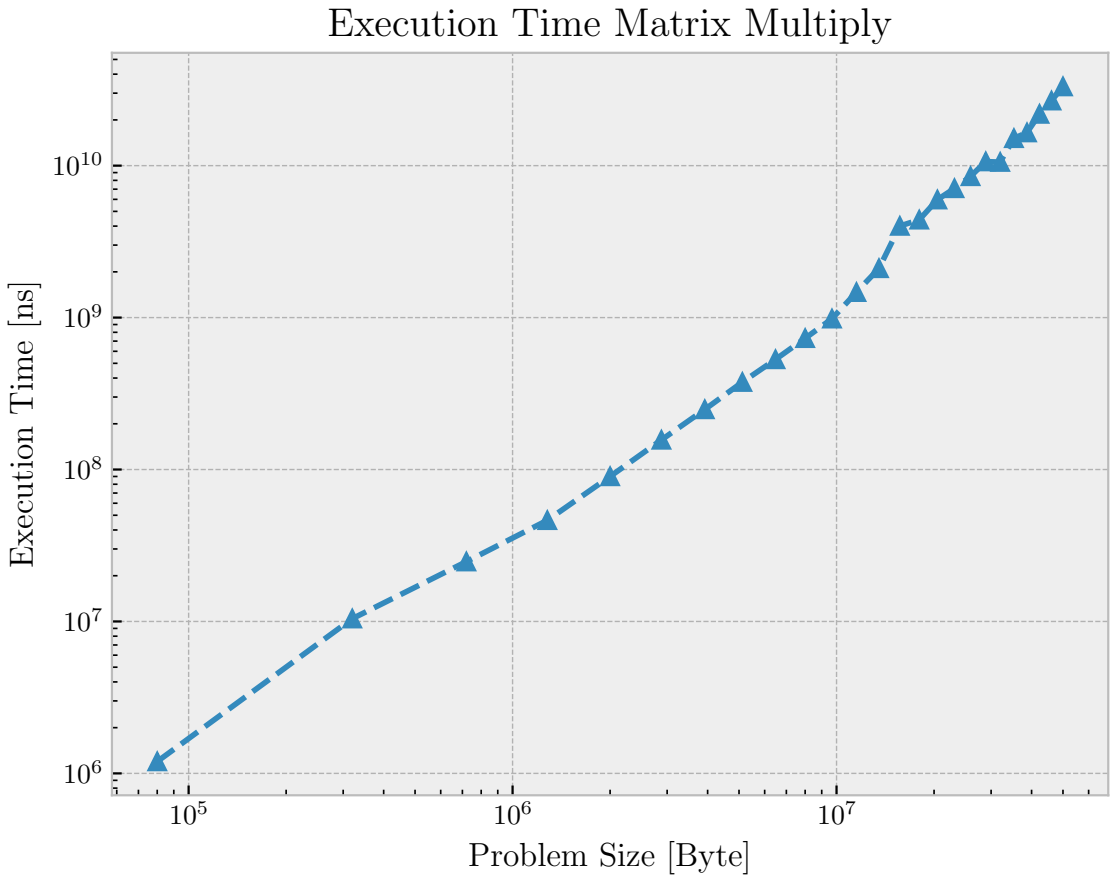

Figure 9: Execution time for matrix multiply depending on problem size

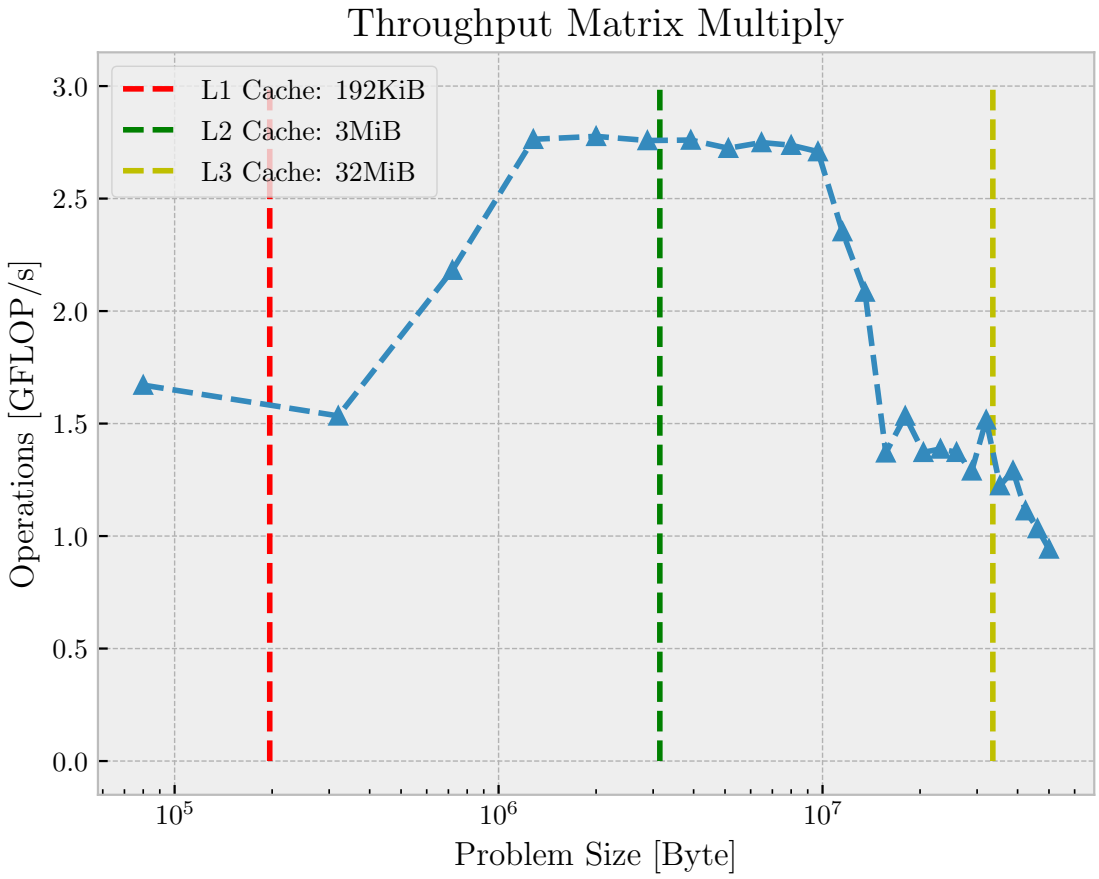

Figure 10: Throughput vs. problem size for matrix multiply. When growing beyond the L3 cache a dip in performance can be noticed. Another dip occures even earlier in between L2 and L3 this is probably due to the poor cache utilization of this implementation in general.

4.5 Willingness to present

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