Exercise Sheet X

Klaus Naumann, Günther Schindler & Christoph Klein

January 19, 2015

1 Matrix Multiply – GPU version using shared memory

We implemented a matrix multiplication kernel with the use of shared memory. We choosed a matrix size of $4 \cdot 10^6$ floating point and measured the kernel time (without copy processes between host and device) in dependency of the thread count per thread block. You can see our results in figure 1. We see a linear decrease in calculation time until we reach a certain thread count per block about 70.

To choose a reasonable thread count per block for further measurements we analyzed the occupancy of a streaming multiprocessor, which is the number of resident threads on the SM divided by the maximum possible number of resident threads. For the GeForce GTX 480 1536 resident threads per SM are possible. To determine the occupancy you have to consider the limiting factors:

- Maximal possible number of threads per SM (1536)
- Maximal available registers per SM (32768)
- Registers per thread (11)
- Available shared memory per SM (49 kB)
- Shared memory used by a thread block, which is in our case equal to the squared number of threads within a block times three (three matrices) times the size of a float (4 Byte)
- The granularity of the threads, as they are composed in blocks on the SM

With this considerations you can create the plot in figure 2. Generally a high number of threads per block should lead to good performance as more threads can reuse the data from the shared memory. Furthermore it should lead to good performance if we use as most shared memory as we can.

With this considerations and the shape of the graphs in figure 1 and 2 we choose the reasonable thread count of 729 per block.

With this thread count we variied the matrix size and measured the calculation time. You can see the result in figure 3. For large matrix sizes the time needed for copy processes is negligible.

In figure 4 you can see the speed up we achieved from the CPU blocked matrix multiplication to the GPU shared memory kernel with a blocksize of 729 threads. This sequential CPU matrix multiplication is optimized for cache usage and therefore executes the matrix multiplication

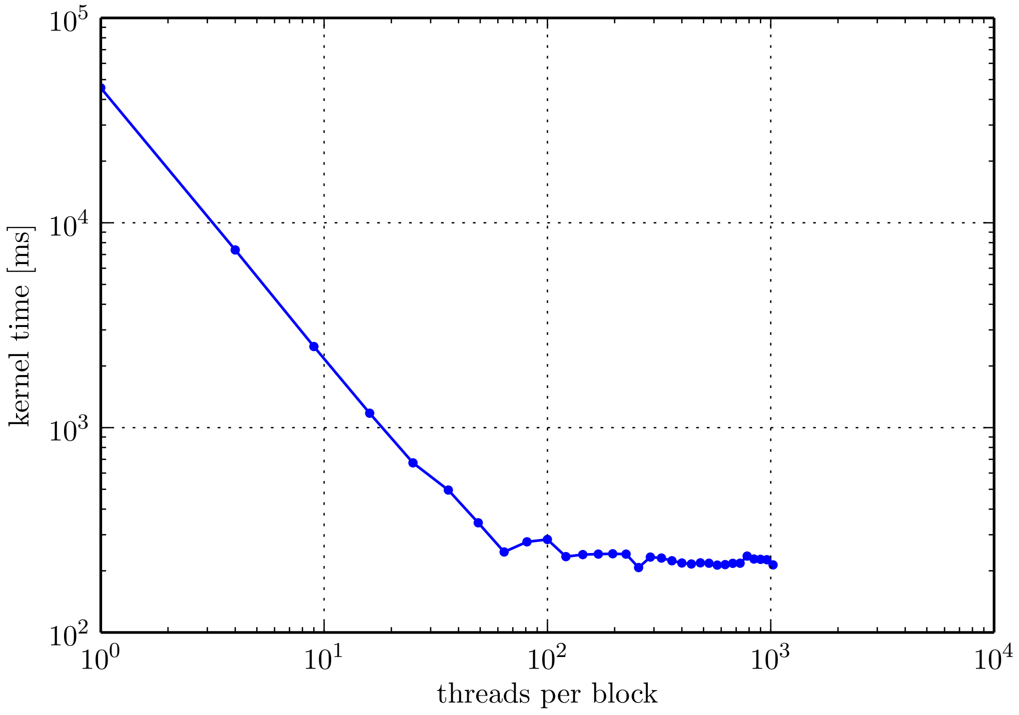

Figure 1: Pure kernel time for a matrix size of $4 \cdot 10^6$ floating point numbers using shared memory.

blockwise. We think that the peaks are at matrix widths, which are whole number multiples of the block width. The major trend of the graph is like linear (be aware of the log scale x-axis) function and seems to finish in a saturation.

Furthermore we plotted the speedup in dependency of the thread count per block in figure 5.

You can see our performance data in table 1. As the peak performance of the GTX 480 is about 1345 GFLOP/s we do not reach it at all. We expect a higher performance if we increase the problem size, as at a problem size of 8100 we start 12 thread blocks, although 26 are parallel executable (13 SMs and 2 blocks per SM are possible in this case).

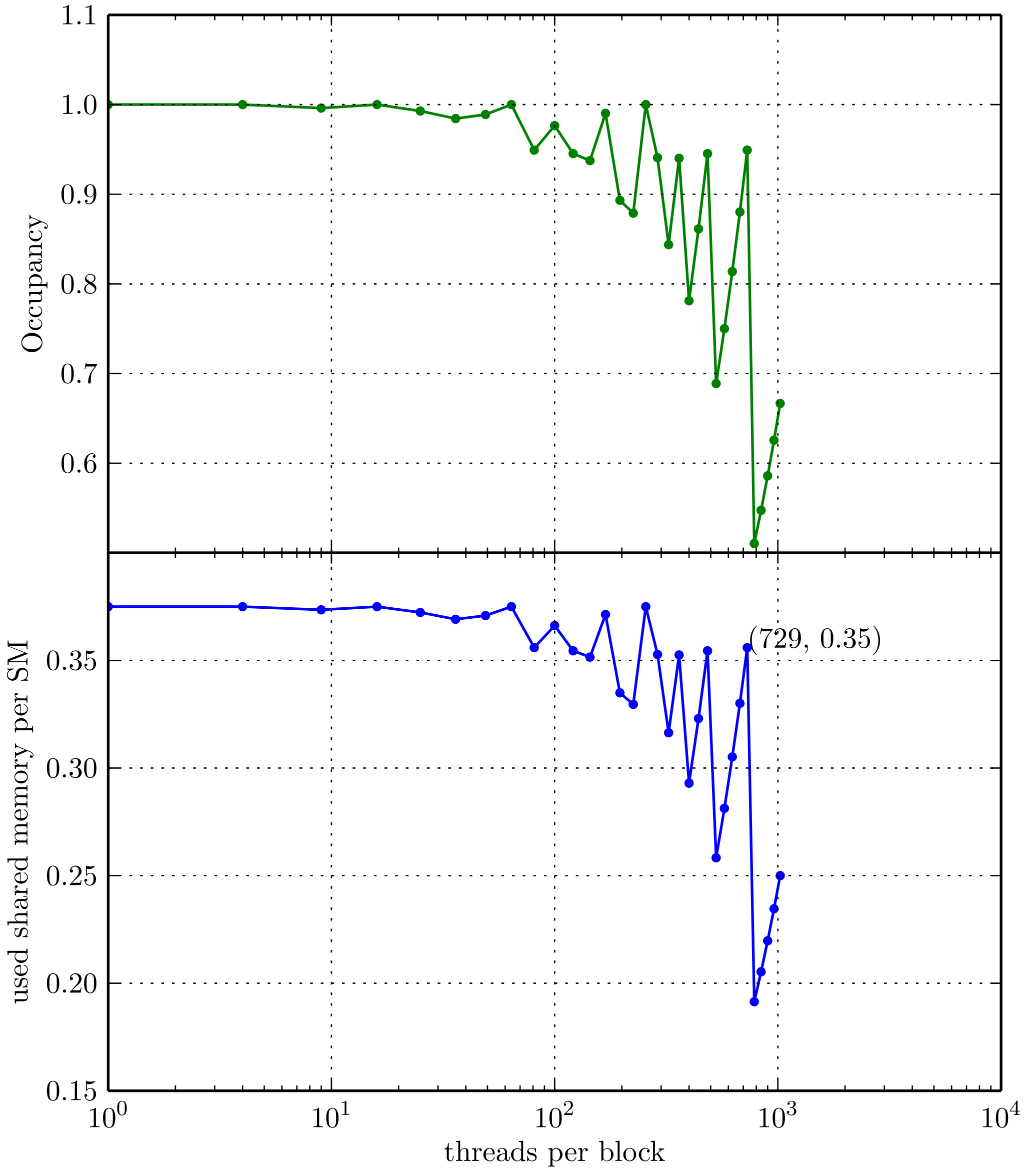

Figure 2: Occupancy and shared memory usage for the shared memory matrix multiplication kernel on the GeForce GTX 480.

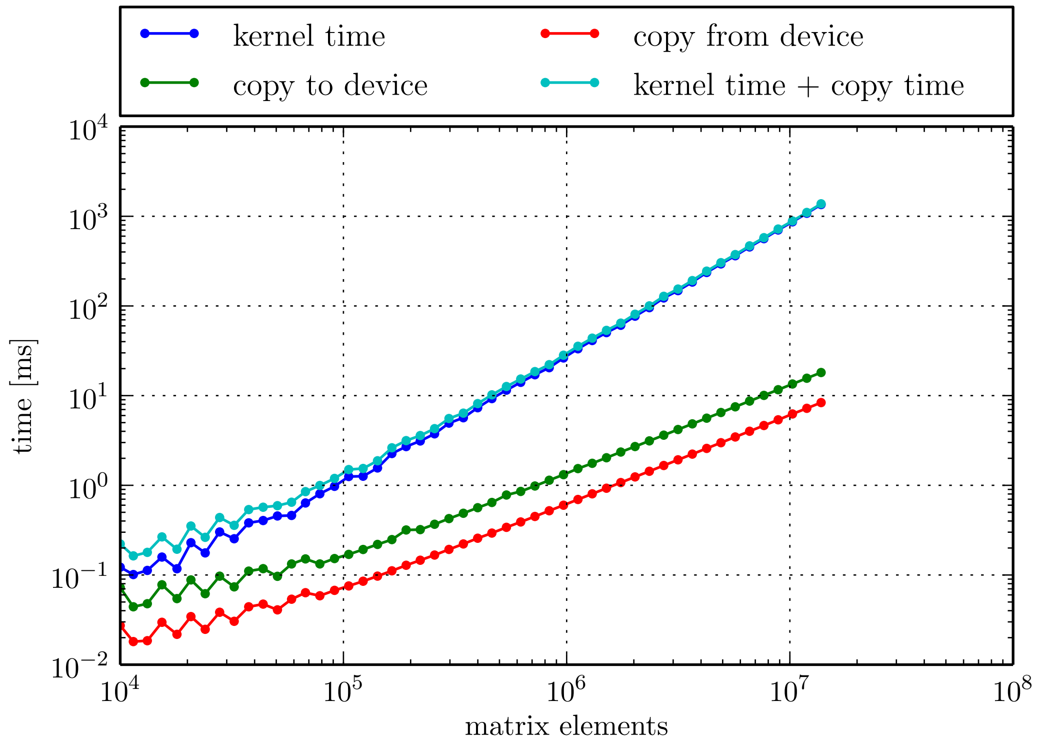

Figure 3: Matrix multiplication time for the shared memory kernel (blocksize = 729 threads) with and without the time needed for copy processes between host and device.

Problem size	Time [ms]	Speedup	GFLOP/s
121	0.0621	0.0653	4.28e-05
256	0.0589	0.1494	1.39e-04
484	0.0690	0.3997	3.09e-04
1024	0.0692	0.8887	9.47e-04
2025	0.0706	2.2903	2.58e-03
4096	0.0814	7.4046	6.44e-03
8100	0.1184	12.9734	1.23e-02

Table 1: Performance data with 729 threads per block. The problem size means the total amount of floating point numbers in the resulting matrix.

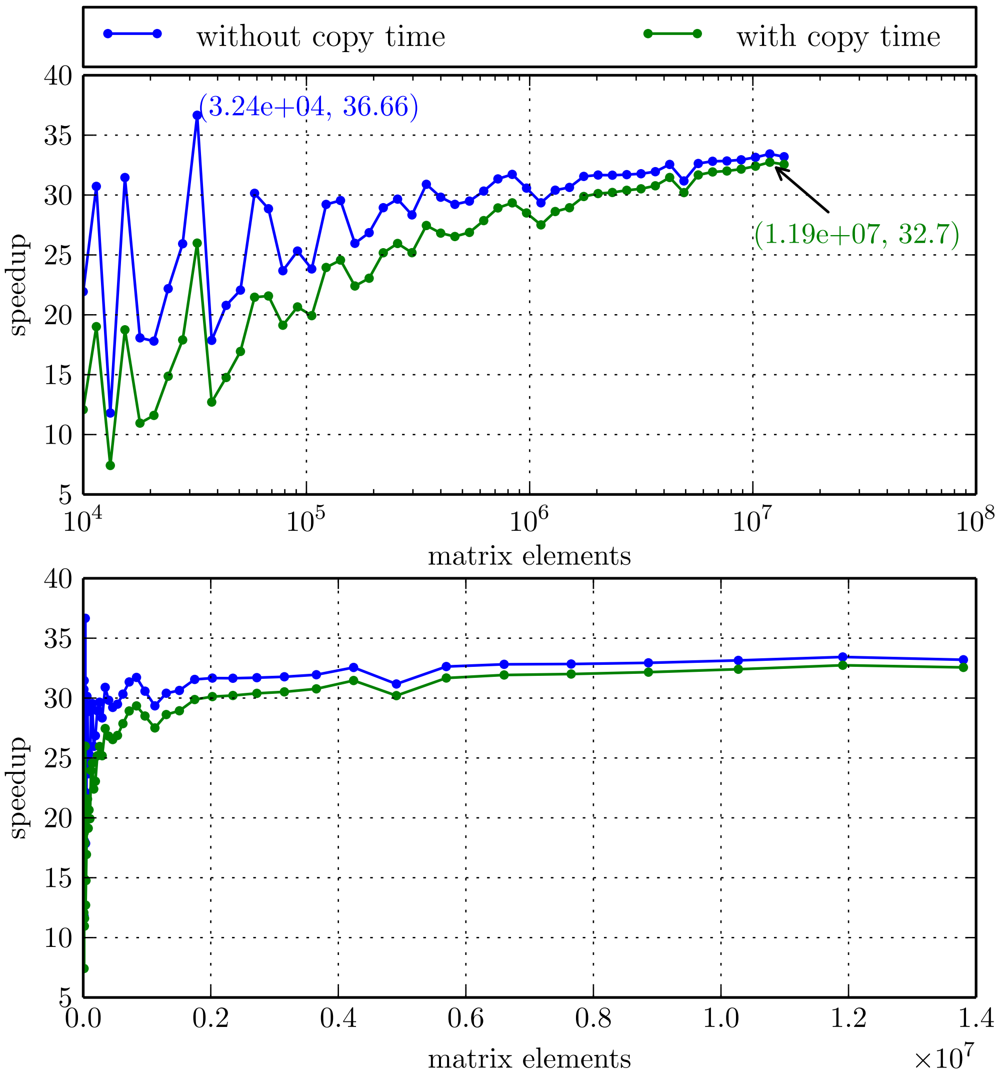

Figure 4: Speedup from CPU blocked matrix multiplication to GPU shared memory kernel with a blocksize of 729 threads.

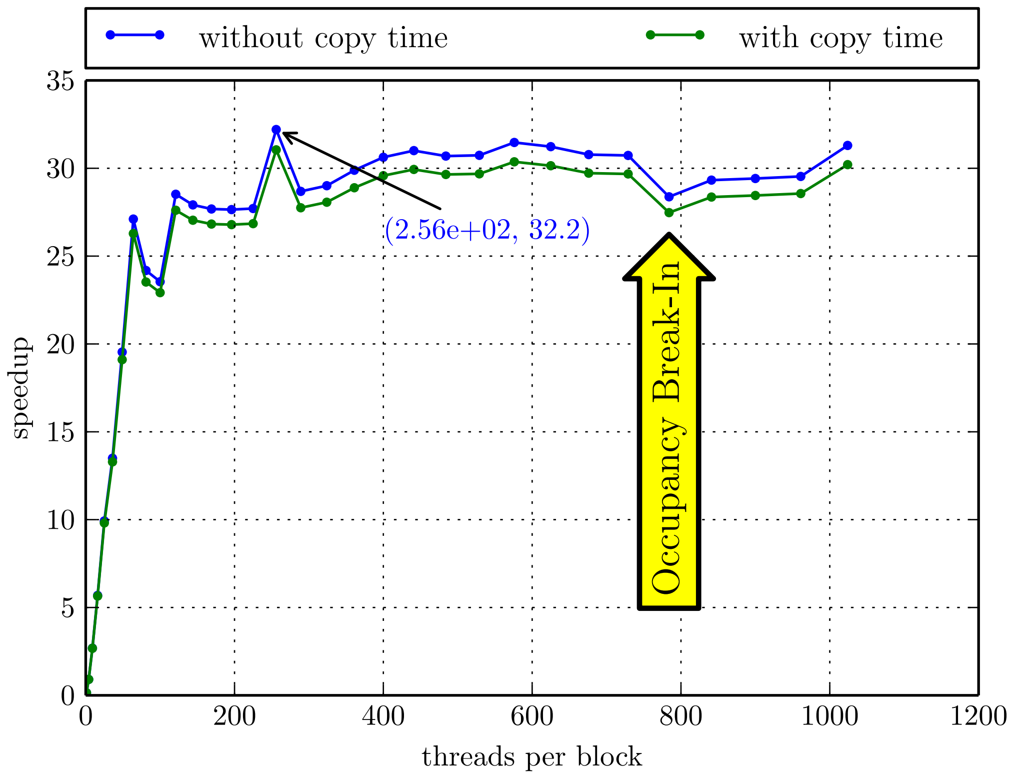

Figure 5: Speedup from CPU blocked matrix multiplication to GPU shared memory kernel with a matrix of $4 \cdot 10^6$ floating point numbers.