# GPU Computing

Exercise Sheet 04

Maximilian Richter, Jan Kränzke, Markus Everling,
Nicolas Schledorn, Olga Sergeyeva, Florian Feltz

November 25, 2024

## 1 Reading

### 1.1 Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. SIGARCH Comput. Archit. News 38, 3 (June 2010), 451-460.

#### 1.1.1 Primary contribution

The paper's primary contribution is the indication that (platform-specific) optimization outweighs the difference in raw compute power between a CPU and GPU. Namely claimed speedups are not nearly as impressive when CPU baselines are sufficiently optimized. They also contribute various ways of optimization and detail their applicability.

#### 1.1.2 Key insight of the contribution

The paper (as the title indicates) hints that the performance gap between CPU and GPU for some given applications is not as large as previously published. They prove this by optimizing CPU baselines to a similar degree as the GPU versions. It also provides background on how this optimization was done and how given properties of a workload can be understood in the context of optimization (of parallelism, comparing types of parallelism and their suitability for app and architecture) and CPU and GPU architectures (comparing architectural features and relating them to data flow).

#### 1.1.3 Our opinion / reaction to the content

Beyond reckoning with unrealistic speedups from using GPUs instead of CPUs for various workloads, insights into which hardware features/limitations (might) help or bound performance on either processor seem helpful to learn. For me, personally, I'm hoping that...

- correlation of measured performance to limiting factors
- Dissection of why certain types of data flow are more problematic than others (and giving ways to mitigate them)
- giving general notes on optimization

... might help in forming intuition (and knowledge) on points to keep in mind when formulating problems into better optimized programs in the future.

### 1.1.4 Rating

As such, I'm inclined to accept the paper. It provides scientific (re-)evaluation of previous papers and methods, analysis of a (back then) rather newer CPU and condensed knowledge about optimization of such programs. I find the use of a 2008 GPU and an over one year newer CPU a little suspicious, but found it difficult to evaluate the historical significance of this difference (as it relates to historical market prices and historical operative costs of each device that may go beyond the scope of this reviewing exercise).

## 1.2 NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro 28, 2 (March 2008)

### 1.2.1 Primary contribution

The main contribution appears to be insight into high- to low-level architecture and design considerations of NVIDIA Tesla processors. This includes in-depth information on hardware and software implementation of graphics pipelines and (CUDA) compute with this (more novel back then) unified, programmable GPU architecture.

### 1.2.2 Key insight of the contribution

The paper provides a general architectural diagram and goes into detail on components of the design from top to bottom, notably how streaming multiprocessors (SMs), an important basic building block of the architecture, are built and connected to the rest of the GPU. Details on other low-level functionality, such as which compound operations were chosen to be implemented in hardware, are also mentioned.

It also takes care to relate concepts of programming to where they are handled in hardware, notably explaining the handling of SIMT as warps.

### 1.2.3 Our opinion / reaction to the content

~~Oh wow this paper really does make festive use of abbreviations and acronyms!~~

Many of the concepts in this paper were already touched on in the lecture, so it would probably have been more interesting for me to read in advance.

### 1.2.4 Rating

Guess I'd go for accept on this one. A paper simply capturing what seems like the state of an architecture feels odd to me, but thinking back to the days of this paper, this wasn't just that, but rather a presentation of contemporary design research and development on NVIDIA's part as the entire industry was changing its way of making GPUs.

### 1.2.5 Additional comments on historical background / "correctness" (Optional)

Though not easy for me to determine just how exactly the described architecture matches current-gen GPUs, as I didn't read a paper on a recent GPU architecture, it seems safe to say the basic concepts translate to much of what we were taught.

Considering how current-gen GPUs still seem to use "unified" architecture, much of it probably held up to the test of time.

## 2 Shared Memory Analysis - Basics

Here some values for our first benchmark of single-block copies. We are calling the kernel 1024 times. This initial kernel only copied the data once, so we could only read/write 48KiB per SM.

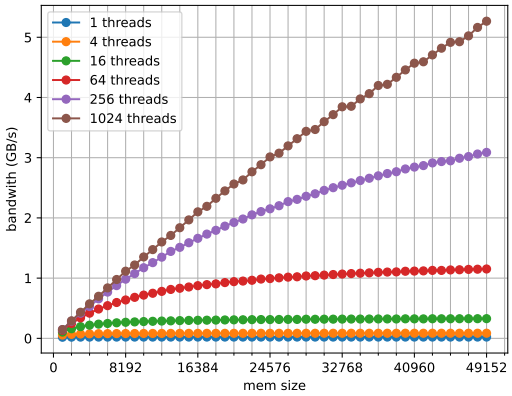


**Figure 1:** Throughput of different memory sizes from global to shared mem. (single block)

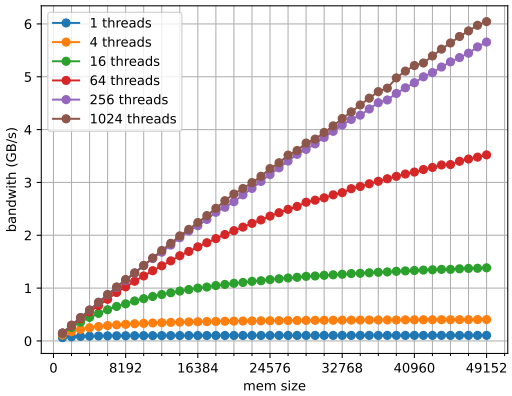


**Figure 2:** Throughput of different memory sizes from shared to global mem. (single block)

These looked a bit odd to us. We decided to rerun the benchmark, but this time looping inside the kernel to copy 16 times.

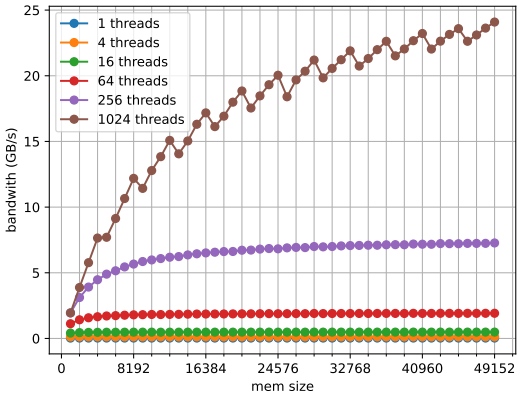


**Figure 3:** Throughput of different memory sizes from global to shared mem. (single block, 16 repetitions inside the kernel)

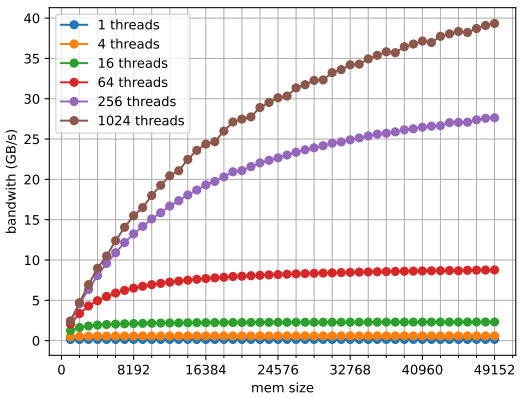


**Figure 4:** Throughput of different memory sizes from shared to global mem. (single block, 16 repetitions inside the kernel)

We got some other values this time.

In conclusion, we are not 100% certain on how to benchmark this correctly and isolate this case from interfering factors (caches, perhaps?). I'll optimistically assume we successfully made launch times insignificant with our repeating copies?

I tried to investigate on my own... same stuff but with 4 repetitions (instead of 16):

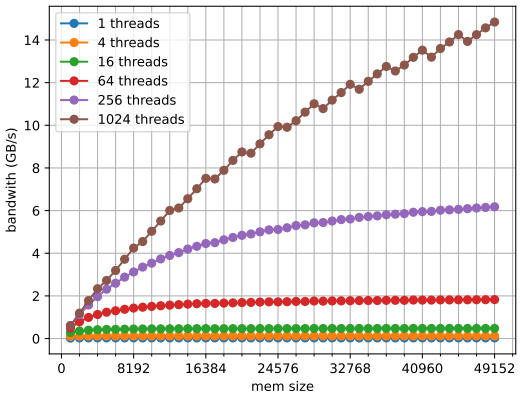


**Figure 5:** Throughput of different memory sizes from global to shared mem. (single block, 16 repetitions inside the kernel)

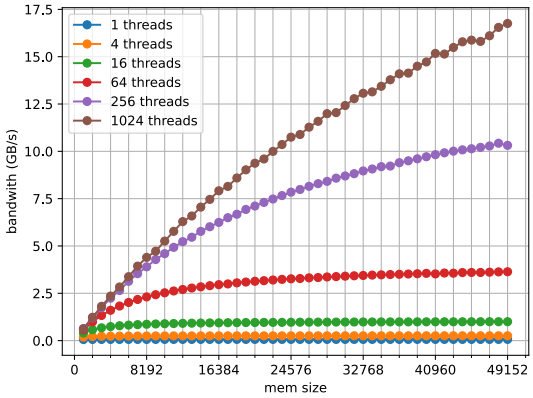


**Figure 6:** Throughput of different memory sizes from shared to global mem. (single block, 16 repetitions inside the kernel)

I give up. This could certainly use some more investigation with guidance from someone more knowledgeable. For us, this will be the result to work with.

Either way, 1024 is certainly the optimal thread count!

The task gets a bit confusing here.

> Now choose an arbitrary (but reasonable) data size. For both directions, vary the number of thread blocks to maximize performance and report it. Note: If you use multiple thread blocks, then each one has its own shared memory so the total throughput is the aggregated throughput of each thread block. Also consider that the amount of data moved depends on the block count.

So? We understand that the throughput will be aggregate.

If it is merely the aggregate throughput that we intend to maximize, things are simple enough; Therefore, the throughput is expected to plateau at a grid size similar to the number of SMs or somewhere slightly above that value.

But in that case why these extensive warnings if the task remains menial? I thought we were told to consider these things in our result calculations. It seems they are relevant only for understanding, not the correct solution of the task itself.

In accordance with our assumptions, we will 'reasonably' choose the largest possible data size per SM: 48 KiB. We will then require a sufficiently large block in global memory to avoid double writes or any such hazards. Our total consumption will therefore be $maxGridSize * 48KiB \approx 2.5MB$. Manageable!

Graphs for varying grid size...

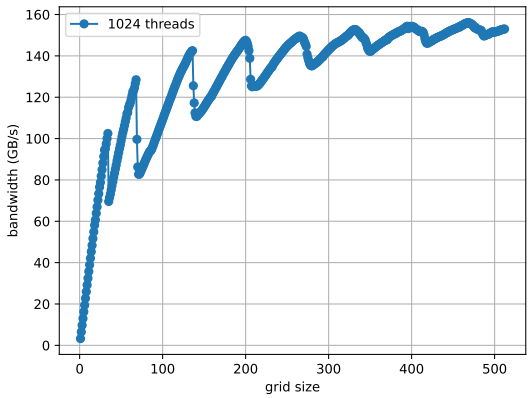


**Figure 7:** Throughput of different grid sizes when copying global to shared)

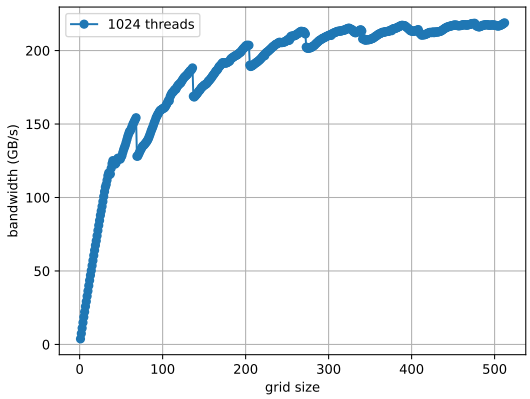


**Figure 8:** Throughput of different grid sizes when copying shared to global)

Graphs for register copies:

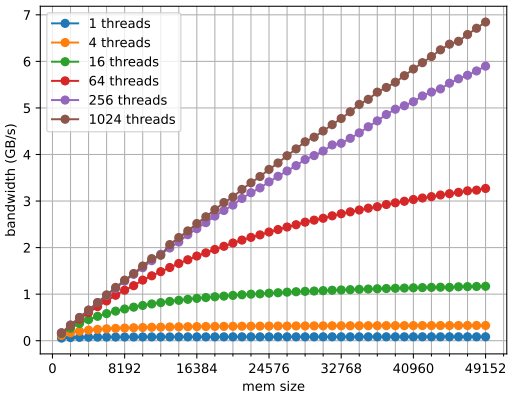


**Figure 9:** Throughput of shared memory sizes when reading to register)

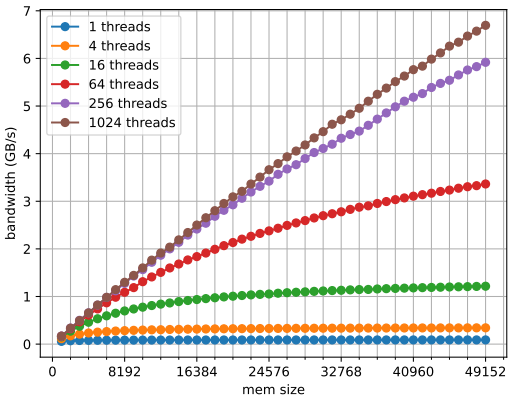


**Figure 10:** Throughput of shared memory sizes when writing from register)

# 3 Shared Memory Analysis - Conflicts

As Figure 11 shows, the performance drops at certain strides. When the stride causes threads to access the same bank (e.g., stride%banks== 0), bank conflicts arise, increasing latency. The observed performance degradation pattern helps deduce the number of shared memory banks, typically 32 for most CUDA architectures. This could also be confirmed by our measurements. Interesntingly, also multiples of 4 are also experiencing performance losses. To avoid conflicts, it is typical to use strides that are co-prime with the number of banks or use padding in shared memory arrays.

# 4 Matrix Multiply – CPU sequential version

Based on our research, we based our statements on the assumption of a Epyc 7302p CPU. We assume a total size of 128 MiB for the L3 cache.

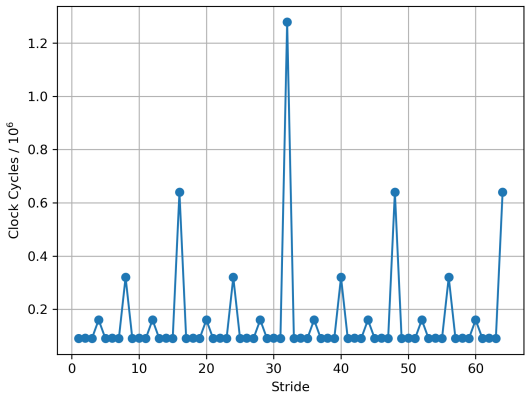


**Figure 11:** Effect of different strides when accessing shared memory and thread-local registers.

| C[i,j] | C[,0] | C[,1] | C[,2] | C[,3] | C[,4] |
|---|---|---|---|---|---|
| C[0,] | 0 | 30 | 60 | 90 | 120 |
| C[1,] | 0 | 40 | 80 | 120 | 160 |
| C[2,] | 0 | 50 | 100 | 150 | 200 |
| C[3,] | 0 | 60 | 120 | 180 | 240 |
| C[4,] | 0 | 70 | 140 | 210 | 280 |

## 4.1 Output Matrix C for 5x5 input

## 4.2 Measuring memory-bound performance

For n x n integer matrices we assume a memory size of

$$size = 3 * N * N * 4$$

since we have 3 matrices, with N * N elements each and 4 bytes per int. To find N, where the cache is no longer effective, we calculate

$$N = \sqrt{\frac{134217728}{3 * 4}} \approx 3344 \approx 27 * 128$$

### 4.3 Run time using varying problem sizes

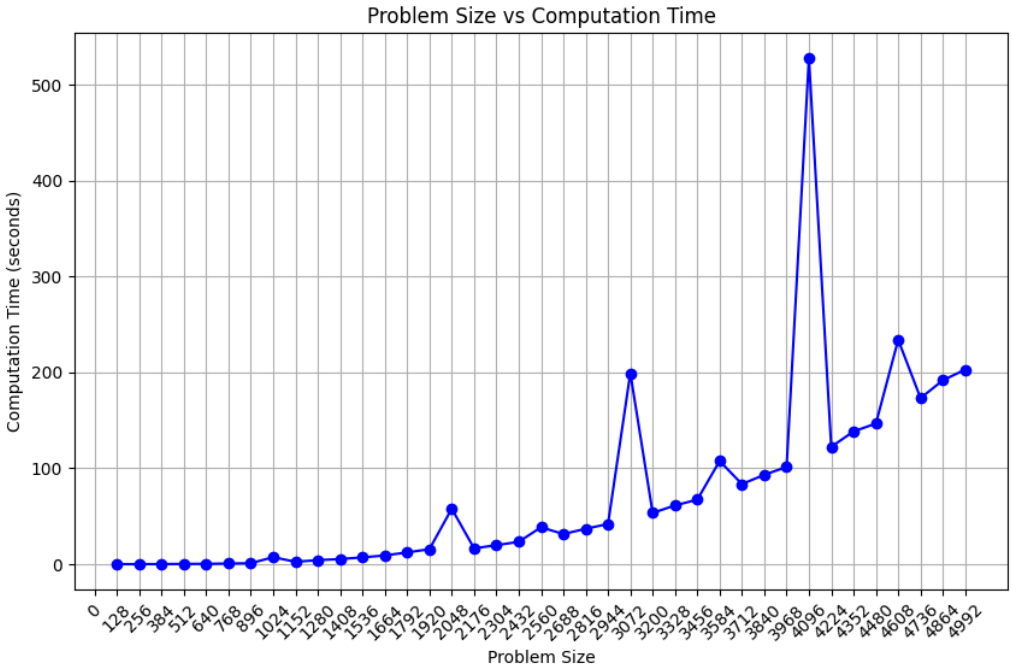


**Figure 12:** Computation time of NxN matrix multiplication using different problem sizes N (dimension)

From the plot it is visible that non-power-of-2 matrix sizes can perform better than exact powers of 2. Cache thrashing results in frequent cache misses and memory accesses, thus increased memory latency. 1024×1024 and 2048×2048 matrices might be sizes where the working set exceeds certain cache levels.

To calculate the sustained GFLOP/s for our matrix multiplication at memory-bound performance we use

$$GFLOP/s = \frac{2 * N^3}{t} = \frac{2 * 3328^3}{61.2455} \approx 1.2$$

Therewith, the performance of the CPU for this matrix multiplication example is far behind of what is typically achieved with a GPU. This is due to the unoptimized multiplication algorithm which results in ineffective memory access patterns as described above. Eventually, this lowers the total GFLOP/s achievable due to memory bandwidth limitations. Implementing cache-aware algorithms should help with this issue.

## 5 Willingness to present

- 1 Reading: **False**
- 2 Shared Memory Analysis - Basis: **True**
- 3 Shared Memory Analysis - Conflicts: **True**

- 4 Matrix multiply - CPU sequential version: **True**