GPU Computing

Exercise Sheet 05

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

January 6, 2025

1 Reading

John Nickolls and William J. Dally. 2010. The GPU Computing Era:

Primary Contribution

This paper makes several significant contributions. It clearly documents the historical evolution of GPUs from the mid 90s up until 2009, tracing their development from simple graphics controllers to the sophisticated parallel processors they are today. This includes the transition from fixed-function pipelines to programmable shaders and the increasing adoption of floating-point arithmetic for enhanced graphics processing. The authors introduce the CUDA progamming model with an enphasis on the hardware implementation.

Key Insight

The first key insight is that GPUs, originally designed for graphics rendering, have evolved into massively parallel processors that are ideally suited for a broad range of scientific and high-performance computing applications. The second key insight is how the CUDA architecture works both in software and hardware.

Opinion

This paper provides a comprehensive overview of the GPU computing landscape at the time. The authors do an excellent job of explaining the technical aspects of GPU architecture and the CUDA programming model in a clear and concise manner. The inclusion of specific examples and performance speedups helps to illustrate the potential of GPU computing for various applications, as was projected at the time.

Hindsight accuracy

This 2010 paper accurately predicted the rise of GPUs in high-performance computing, driven by the increasing demand for faster and higher-definition graphics. It highlighted the evolution of GPUs from simple graphics controllers to sophisticated parallel processors, capable of handling a broad range of scientific and high-performance computing applications. However, it understandably missed the mark on the transformative impact of cryptocurrency mining and the AI market, both of which have significantly influenced the GPU market. The authors also emphasized the importance of the CUDA parallel computing model in unlocking the general-purpose computing capabilities of GPUs.

2 Matrix Multiply – GPU naive version

We compared our matrices with CPU calculations for up to a size of 512^2. We used pinned memory wherever possible.

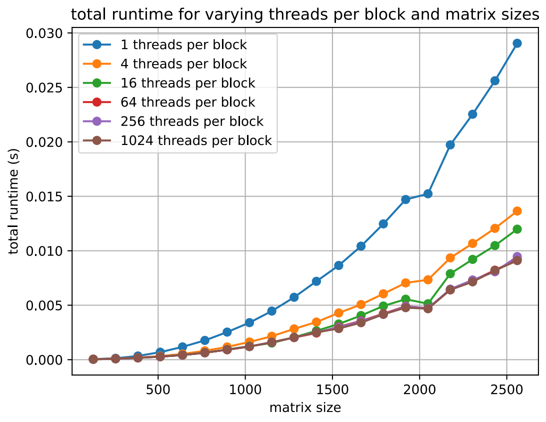

Figure 1: total runtime by varying threads per block and matrix sizes

We did not consider the transfer time relevant here, since it's constant between different thread counts and would only have made the graph more convoluted.

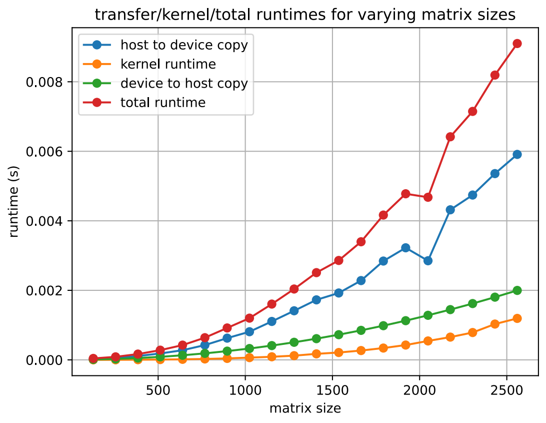

Figure 2: runtimes at a constant 1024 threads at different matrix sizes

For the highest speedup: Matrix size being 2560, and with comparison between a naive cpu implementation (38000 ms) and the "naive" gpu version (9 ms with data movements, 1.1ms without) would be about 4000x (with copies) or about 35000x (without).

3 Matrix Multiply – GPU version using shared memory

We compared our matrices with CPU calculations for up to a size of 512^2. We used pinned memory wherever possible.

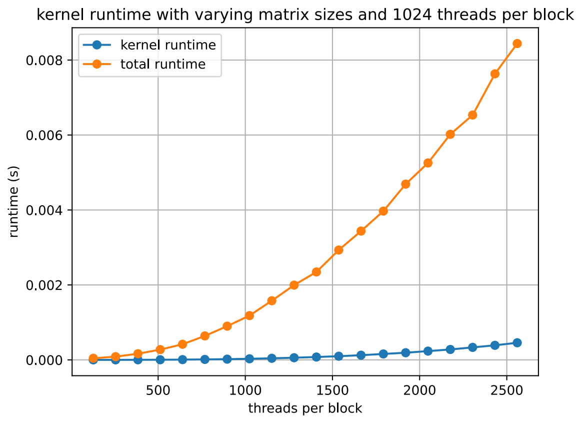

Figure 3: total runtime by varying matrix sizes

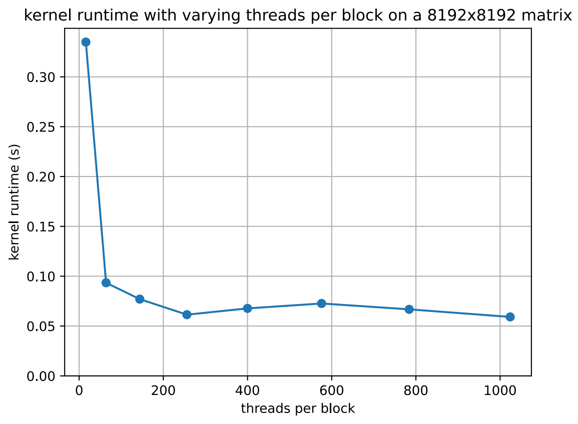

Figure 4: total runtime by varying thread number

Thus our shared memory version is about 2.6 faster in kernel runtime (for size 2560) than the naive version. The total runtime is dominated by copy operations however and only sped up by about 8% (at the same problem size).

For the highest speedup: Matrix size being 2560, and with comparison between a naive cpu implementation (38000 ms) and the "shared memory" gpu version (about 8.4 ms with data transfers, 0.4 ms without) which makes for speedups of 4500x and 95000x, respectively.

4 Bonus: Matrix Multiply - Caching turned on

We only timed kernel runtimes as transfer times would remain similar between the two.

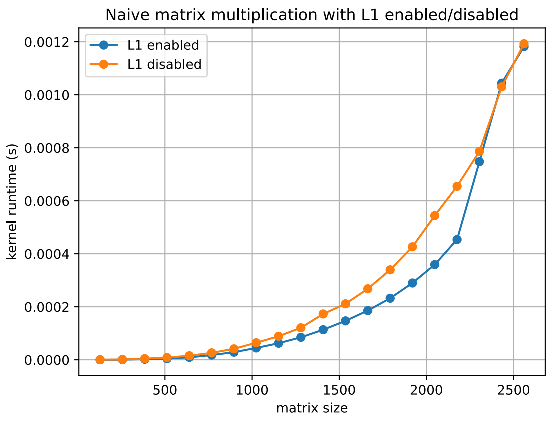

Figure 5: gpu naive kernel runtime with L1 enabled vs disabled

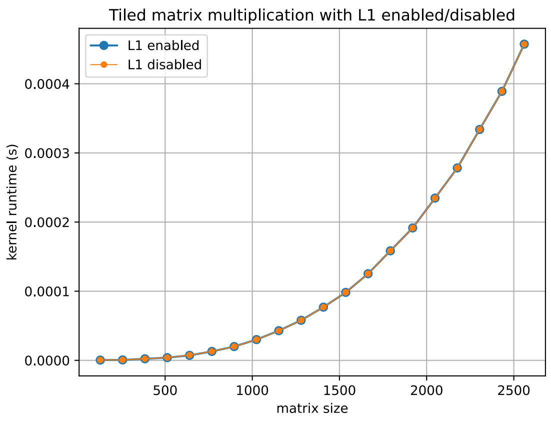

Figure 6: gpu tiled kernel runtime with L1 enabled vs disabled

This shows how the optimized version doesn't require L1 cache to function at all, as it loads its data into shared memory anyway and does so perfectly well.

Advantages of L1 cache are thus limited to suboptimal implementations and lower problem sizes, where some data may fit into L1 cache and manage to make a difference in runtime. We see this effect diminishing with larger problem sizes.

5 Willingness to present

- Reading: **false**
- Naive matmul: **true**
- Tiled matmul: **true**
- Caching turned on: **true**