GPU Computing

Exercise Sheet 06

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

December 16, 2024

1 Reading

Roofline: an insightful visual performance model for multicore architectures

The paper introduces a visual computational model called the *Roofline model* which helps to understand and improve performance in multicore computer architectures. At the same time, it highlights the relationship between computational power and memory bandwidth of a given computer running a certain kernel.

The *Roofline model* is designed as an alternative approach to existing methods that attempt to predict performance. Instead, based on the principle of "bound and bottleneck analysis", the model provides insights into which factors limit computing performance. The authors introduce a concept called "operational intensity", which measures the efficiency of floating-point operations relative to data movement from memory. Specifically, it visualizes operations per byte of DRAM traffic, where total bytes are defined as those going to main memory after passing the cache hierarchy.

The two lines, typically found in the *Roofline model* graph, are derived from the peak computational performance (horizontal roof) and the maximum floating-point performance that the computer's memory system can support for a given operational intensity. The ridge point of the two lines indicates the minimum operational intensity required to achieve maximum performance. The more this point is to the right, the more difficult it is for programmers to achieve peak performance. Most importantly, the two ceilings of the model suggest ways of optimizing a program to run more efficiently on a specific hardware architecture. Hereby, the model not only helps in identifying what optimizations should be done in which order but also the potential performance reward for each.

Using four kernels, taken from a collection of commonly used algorithmic procedures called the "Seven Dwarfs", the authors demonstrate the model's validity. Ultimately, with regard to the recent trends towards very diverse parallel computing architectures, the paper is providing guidance for design architects to identify which systems are well-suited to run important kernel programs.

2 Reduction – CPU sequential version

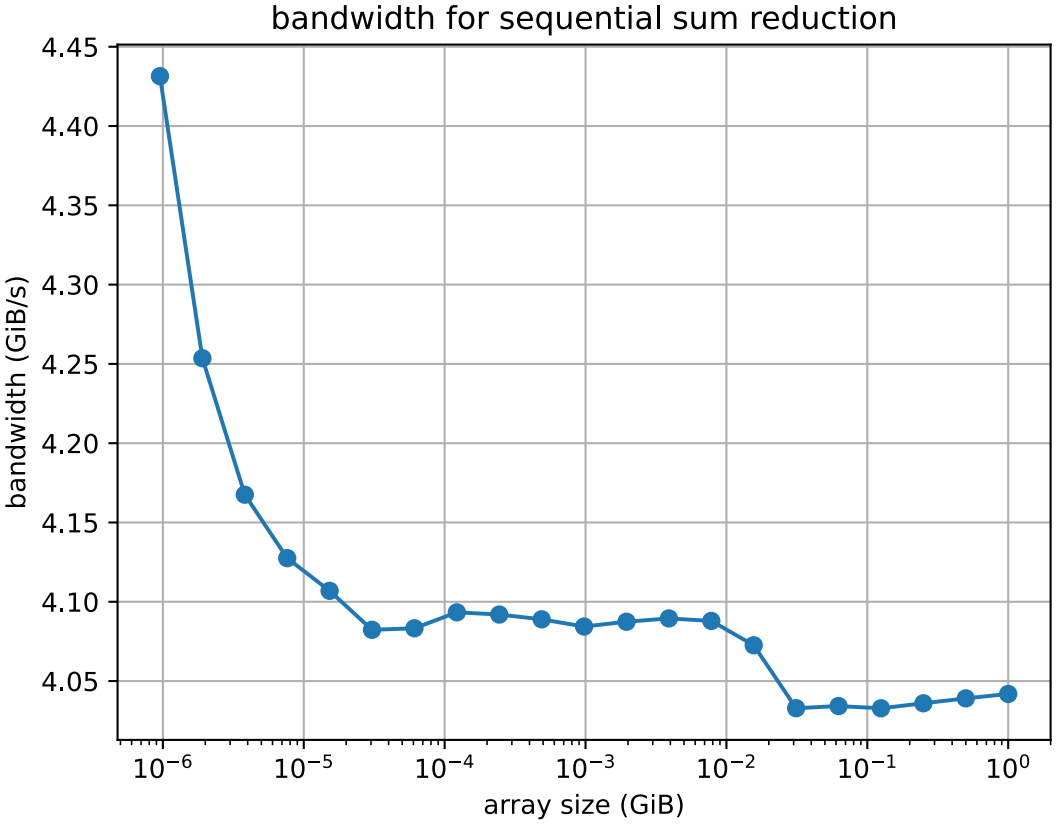

Figure 1: bandwidth for sequential CPU sum reduction

In this section, we implement a sequential CPU version of a global sum reduction, using `std::accumulate()`. We choose different problem sizes, ranging from 10^3 to 10^9 `float` elements. The selected sizes ensure that we are working with an amount of data that is well beyond all CPU cache levels.

For large problem sizes, the bandwidth settles at around 4 GiB/s. For smaller sizes, the bandwidth is a bit higher, presumably because the memory is being cached between timing iterations.

3 Reduction - GPU naive version

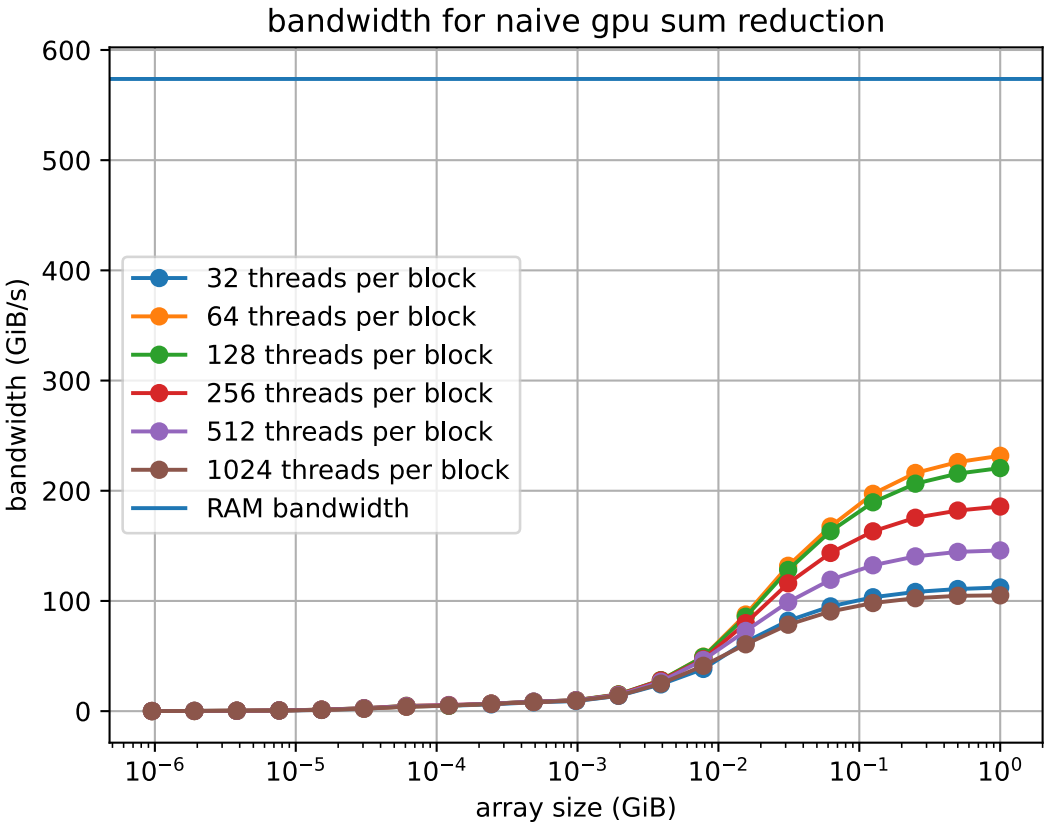

Figure 2: bandwidth for naive sum reduction kernel

In this section we implement a naive sum reduction kernel, as presented in the lecture. One thread is spawned for each input element, and elements within one thread block are summed together in an interleaved fashion. Afterwards, more kernels are launched as required, until only one partial sum remains (Note that the exercise said to only launch 2 kernels, but that would limit the maximum problem size to 1024×1024 elements?).

This naive implementation achieves a maximum throughput of 216GiB/s, roughly 37% of device memory bandwidth. It achieves the maximum throughput at a block size of 64, presumably because larger thread blocks lead to a lot of idle threads during the reduction. For small sizes however, the bandwidth is largely throttled by the kernel launch time.

4 Reduction - GPU optimized version

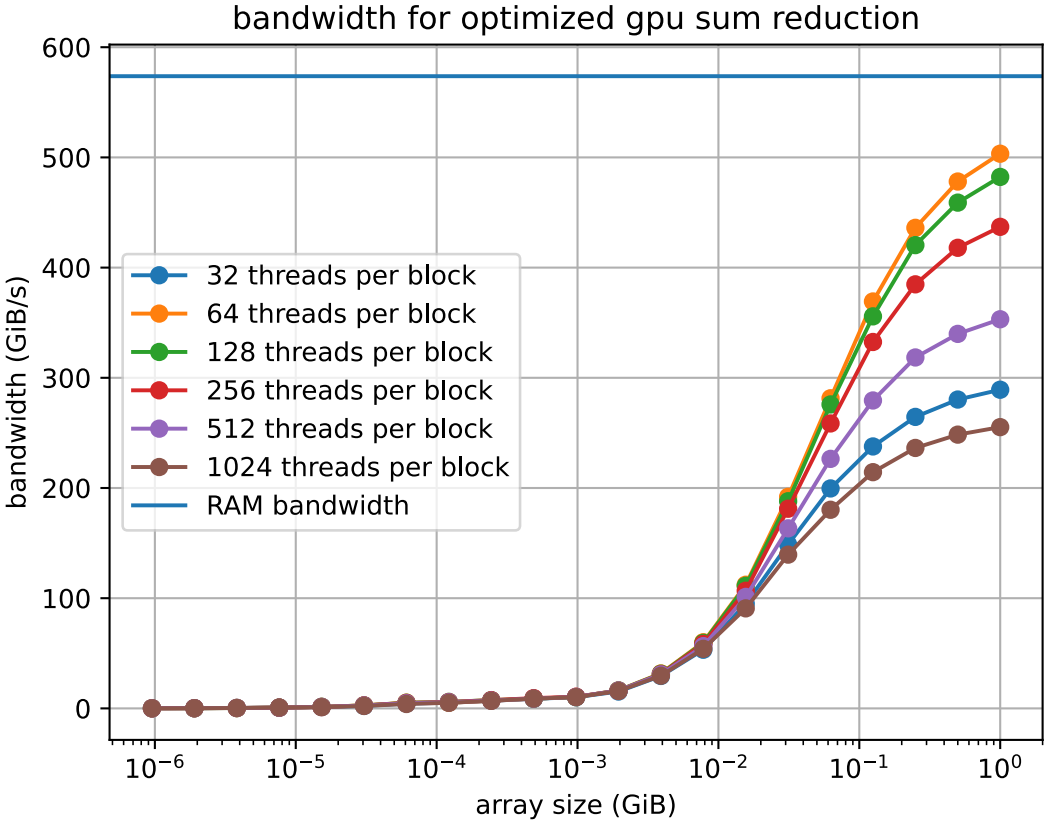

Figure 3: bandwidth for optimized sum reduction kernel

In this section we refine our sum reduction kernel using optimization techniques presented in the lecture. We rename our threads to use sequential addressing instead of interleaving addressing, which allows for load coalescing and reduces branch divergence. We also make every thread load two input elements, cutting the number of required threads in half. Finally, we unroll the last warp, which reduces branch divergence and the number of required thread block synchronizations.

This optimized implementation achieves a maximum throughput of 503GiB/s, roughly 88% of device memory bandwidth. Like the naive kernel, it achieves maximum throughput at a block size of 64, likely due to similar reasons as for the naive kernel.

It is roughly 125x faster than the sequential CPU version and roughly 2.3x faster than the naive version.

5 Reduction - GPU Volta optimized version

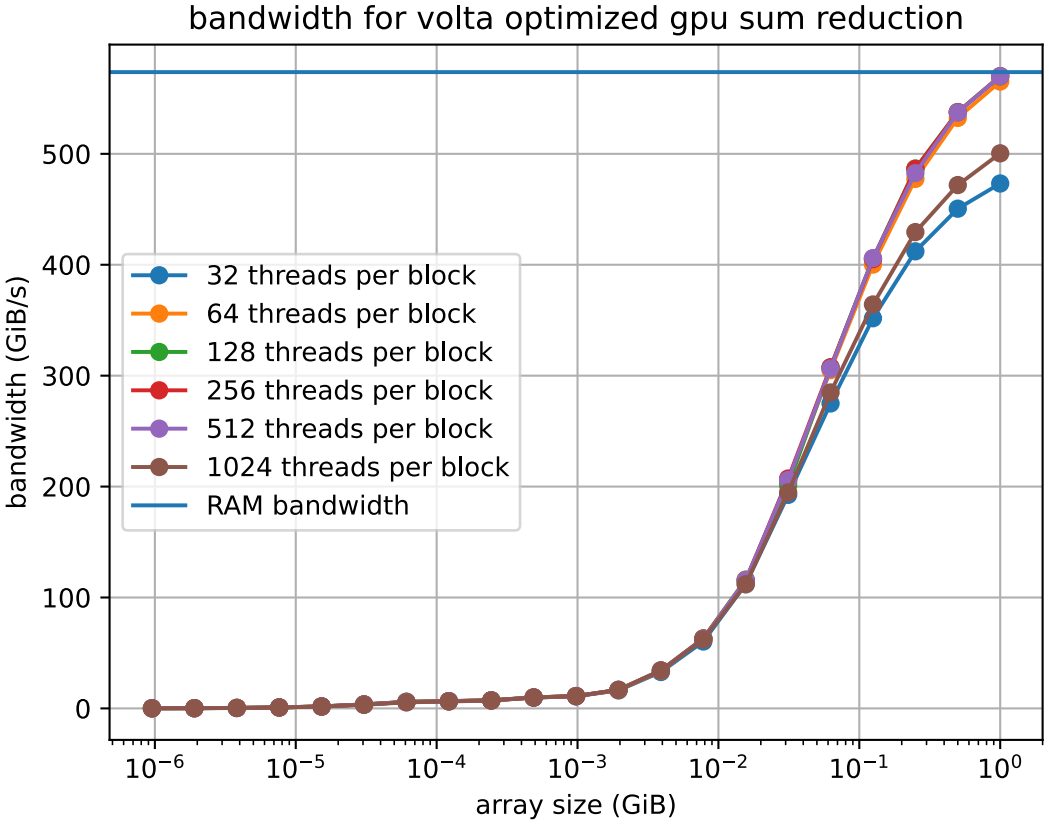

Figure 4: bandwidth for Volta optimized sum reduction kernel

In this section, we implement a reduction kernel that is optimized for Volta (technically Kepler) or newer GPUs. It takes advantage of Cooperative Groups and atomics to minimize shared memory accesses and branch divergence and to compute the entire sum reduction in only one kernel launch.

First, the kernel reads in 8 input elements per thread. The value 8 was chosen experimentally to lead to the best performance. Then, it uses the Cooperative Groups `cg::reduce()` to add together all values inside each warp. Internally, this uses the `__sync_shfl_xor()` intrinsic (introduced in Kepler), which directly accesses the registers of the other threads in the warp, circumventing the need to roundtrip to shared memory. The low lane of each warp then writes that warp's sum into a 32-float array in shared memory (this works because there are at most $\frac{1024}{32} = 32$ warps per thread block). Finally, the block is synchronized and the total sum of the block is calculated with another warp level reduction on warp 0 of the thread block. This block sum is then added atomically onto the output value, using the `atomicAdd()` intrinsic.

This implementation achieves a maximum throughput of 570GiB/s, saturating the device memory's read bandwidth. This time, maximum throughput is achieved with 512 threads, indicating that this kernel can more effectively utilize the threads in each thread block.

It is roughly 142x faster than the CPU version and 13% faster than the optimized kernel from exercise 3.

6 Willingness to present

- Reading: **true**
- Reduction - CPU sequential version: **true**
- Reduction - GPU naive version: **true**
- Reduction - GPU optimized version: **true**
- Reduction - GPU Volta optimized version: **true**