ETH zürich

High-Performance Computing Lab for CSE **2020**

Student: Jan Stratmann Discussed with: Dominik Helmreich, Aurelio Dolfini

Solution for Project 1

Due date: 02.03.2020, 11:30pm

In this project you will practice memory access optimization, performance-oriented programming, and OpenMP parallelizaton on Euler.

1. Explaining Memory Hierarchies *(30 Points)*

1.1 Parameters Using the recommended commands I was able to identify the memory parameters for the Intel Xeon E3-1585L v5 processor. This means the data is correct for the Euler III cluster. The node had 32 GB of DDR4 main memory, which, according to the Scientific Computing website, is clocked at 2133MHz. Each node features one of the above quad-core processors with 8 MB of L3 cache, which is shared among all four cores. Each core additionally features 256 KB of L2 cache as well as two 32KB L1 caches (one for data and one for instructions).

1.2 Memory Benchmark When running the (revised) memory benchmark, one would expect to see differences when jumping from 32 KB to 64 KB, 256 KB to 512 KB, and from 8MB to 16 MB of array length. This is due to the different cache levels not being able to hold arrays larger than 32 KB, 256 KB, and 8MB, respectively. These expectations were largely met, although I could observe a significant jump in load time from 4 MB to 8 MB as well. I would attribute this to the shared nature of the L3 cache.

1.3 Access Patterns For the leftmost case in *Fig. 1* corresponding to $csize = 128$ and $stride = 1$, data is read sequentially from memory. This allows for optimizations such as prefetching and therefore also for small access times regardless of the actual $csize$ value. In the case of $csize = 2^20$ and $stride = csize/2$, we are looking at a 4 MB long array with stride 2 MB. This means we are only accessing two memory locations which can easily be kept in L1 cache (remember that the cache is hot when the test starts). Therefore, access times are low in this example as well.

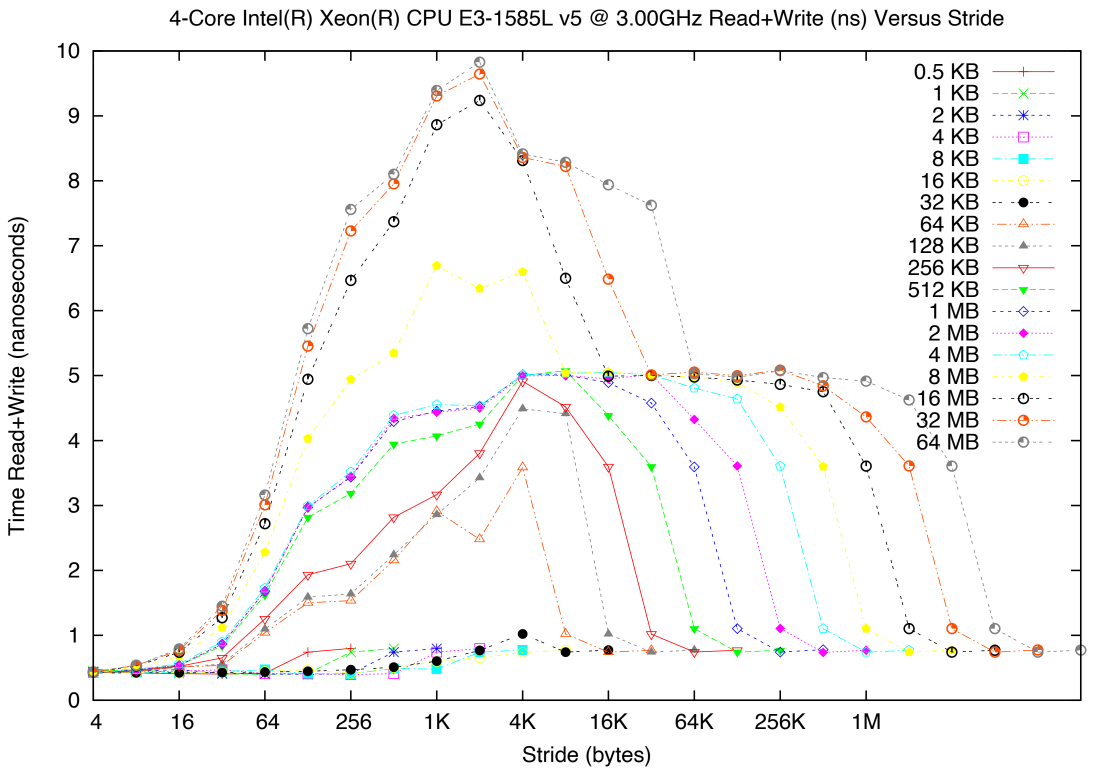

Figure 1: Memory Hierarchy on Euler III

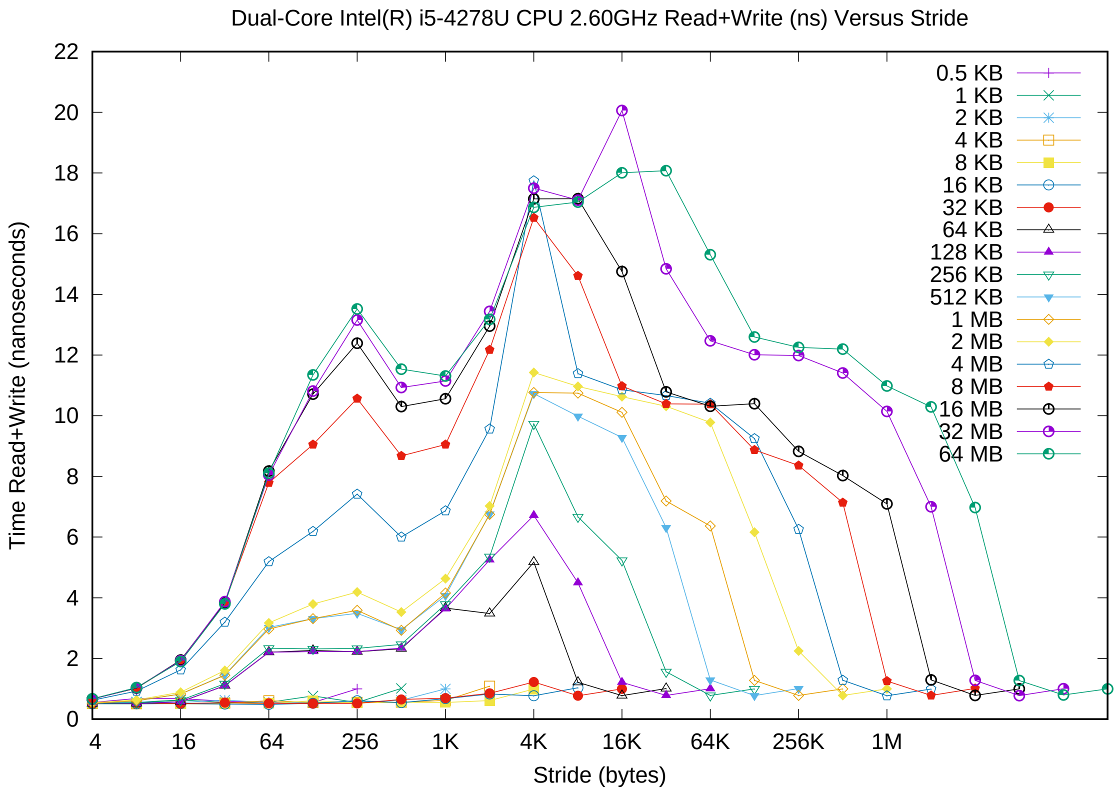

Figure 2: Memory Hierarchy on Local Machine

Generally speaking, traversing an array with small stride alwdsays leads to some degree of spatial locality. Since we execute the test multiple times, we also benefit from temporal locality. This comes in especially handy when only a few values are accessed because of relatively large stride values (e.g. $stride = csize/2$). In such cases, we can profit off the 2-associativity of the L1 cache, as well as the 16-associativity of the L2 cache. This manifests itself in access times equal to five nanosecond or below if all traversed data points fit into the L2 cache. If we only read and write to two values, we even get access times of one second or below.

1.4 Local Machine The Memory Benchmark was also run on my local machine, a mid-2014 MacBook Pro with an Intel Dual-Core i5-4278U processor clocked at 2.6 GHz. Besides a 32 KB L1 instruction cache it features a 32 KB L1 data cache and 256 KB L2 cache per core as well as a 3MB shared L3 cache. The machine also features 8 GB of DDR3 memory clocked at 1600MHz. The Benchmark as seen in *Fig. 2* exhibits similar characteristics, although absolute worst-case access times are slightly higher due to the smaller L3 cache. Although this cannot be inferred from the graphs, I would also expect the best-case access times to be slightly higher than on Euler, as my local machine has a lower clock rate.

2. Optimize Square Matrix-Matrix Multiplication *(70 Points)*

2.1 Single Thread Multiplication As advised in the project description, I started by trying to block operations. This is done by extending the code to be a six-loop construct rather than consisting of only three loops, where the inner three loops basically do the same thing as the outer three loops. The only difference is that the inner three loops operate on a block that is limited

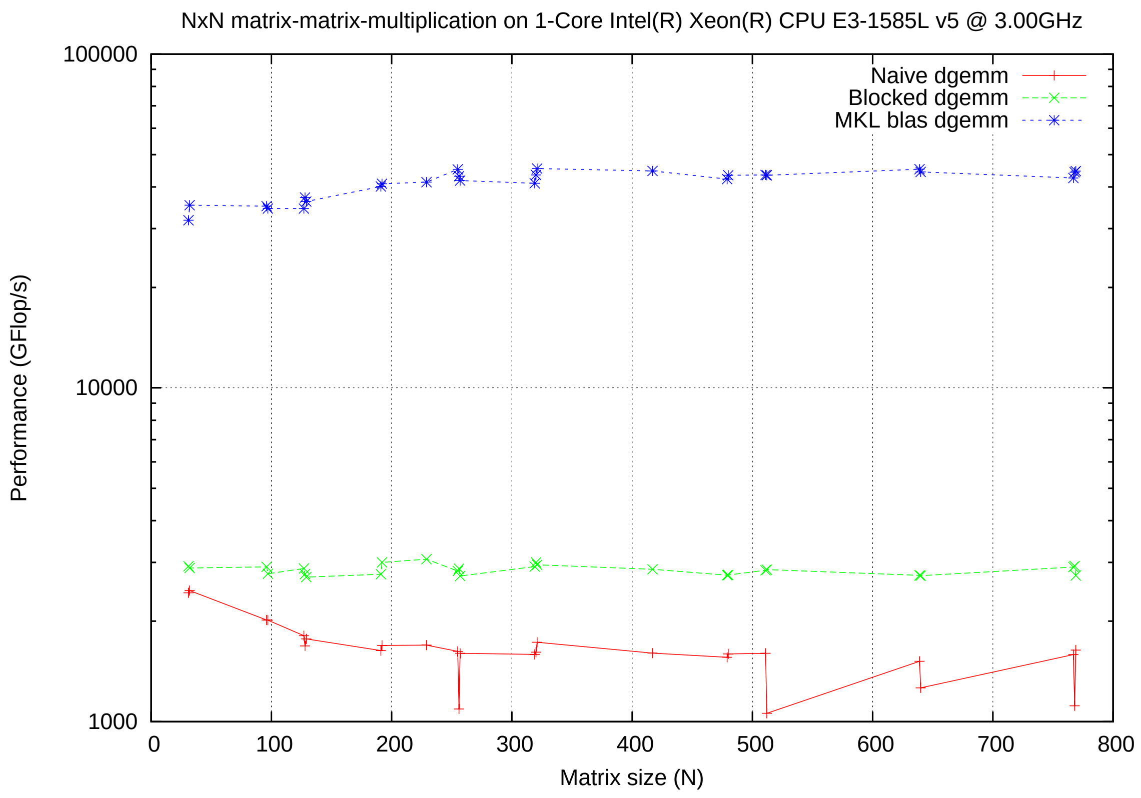

Figure 3: Single-Core Matrix Multiplication on Euler III

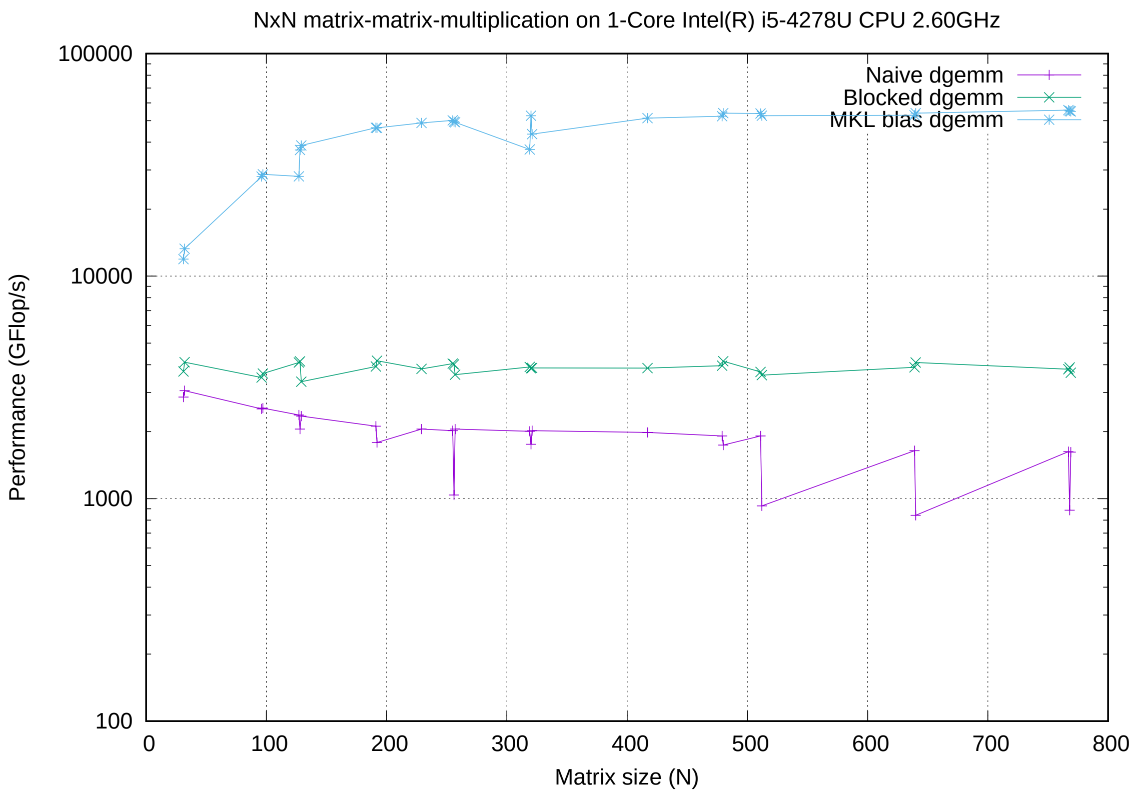

Figure 4: Single-Core Matrix Multiplication on Local Machine

by a certain block size. After some initial indexing errors (I implicitly assumed that matrix sizes where powers of two in the beginning), my code was finally correct. I did not, however, achieve any speedup compared to the naive implementation. This was due to an unfortunate traversing order in the given template, as it did not feature sequential memory access. As soon as I fixed the code to perform unit stride, the blocking optimization finally led to some improvement. Additionally, I then moved the index computations out of the innermost loop to further speed up the multiplication. This did not lead to the desired effect, which is why I reverted to the previous version. Although I didn't manage to reach the maximum amount of floating point operations per second, I did achieve almost constant MFlops/s (see *Fig. 3*. The slight variations can be attributed to system noise, since this task only uses one core of the 12 available per processor (remember that we are running on the same processor as in task one). Additionally, corner cases also influence the measurements a tiny bit. The same is true when testing on my local machine, as seen in *Fig. 4*. Unsurprisingly, the optimal block size was also the same for both machines (16 in both cases). This behaviour was to be expected, since both feature Intel Skylake processors and the same L1 and L2 cache sizes.

Despite my efforts, the key takeaway from *Figs. 3 and 4* is to use library functions where available. This is especially handful when working on clusters, since they sometimes offer hand-optimized libraries tailored to the specific architectures available (e.g. the Intel MKL on Euler or the Accelerate Framework on Macs).

2.2 Multi-Thread Matrix Multiplication on Euler When parallelizing the above code using OpenMP, this simply translated to using the standard pragma *omp parallel for*. It made sense to choose the number of OMP threads equal to the number of cores used. Also, I chose to parallelize the outermost loop of the function to minimize parallelization overhead. I also thought about using dynamic scheduling and briefly tested it. But since we are handling dense matrices in this task,

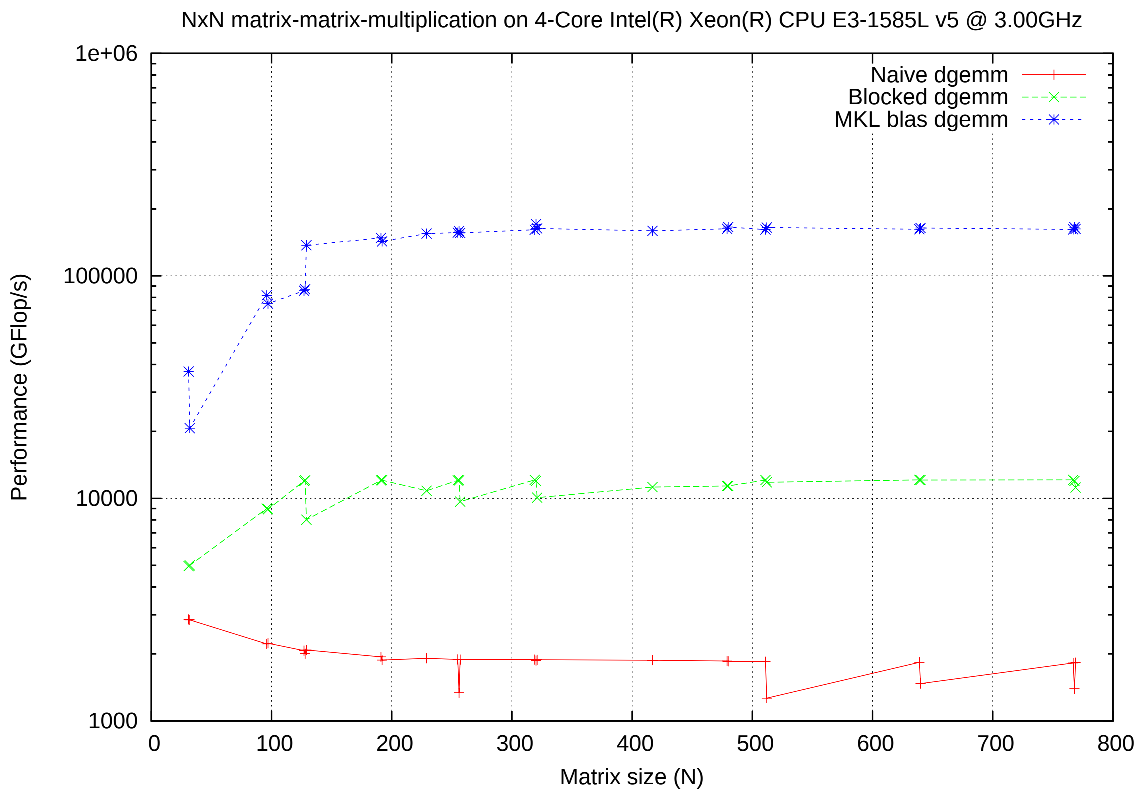

Figure 5: Multi-Core Matrix Multiplication on Euler III

using dynamic scheduling didn't make any sense. Unfortunately, OpenMP is not available to test on my local machine, which is why I cannot compare performance accross architectures in this case.