ETH zürich

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

Student: Konrad Handrick Discussed with: Felix Illes, Theo Smertnig

Solution for Project 1

Due date: 09.03.2020 (midnight)

HPC Lab for CSE 2020 — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to Moodle (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

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

1. Explaining Memory Hierarchies *(30 Points)*

Memory hierarchy on compute nodes on Euler cluster (and my local laptop, for completion):

	XeonGold 6150	XeonE5 2680v3	XeonE3 1585Lv5	i7-8565U
Main memory	36 GB	24GB	4GB	8GB
L3	25MB	25MB	30MB	8MB
L2	1024KB	1024KB	256KB	256KB
L1i	32KB	32KB	32KB	32KB
L1d	32KB	32KB	32KB	32KB

Table 1: Euler IV - 2.70GHz, Euler II - 2.50GHz, Euler III - 3GHz and 1.8GHz respectively

The distinction between L1d and L1i is on purpose because that way we are forced to think about the difference between instruction and data cache.

For the first case (csize=128 and stride=1), we see what is supposed to happen following the simplified mental image: small data together with small stride makes for fast access to L1 cache. Here all of the elements should in principle fit on the same cacheline.
In the other case (csize=2^{20} and stride=$csize/2$) we see the compiler doing its prefetch and branch

prediction magic such that while accessing one element, an instruction is issued to load the element farther away in memory to touch it in the cache in a fast way. Here probably efficient instructions to the TLB matter to access data that's not in cache anymore such that it can be prefetched in a fast manner.
From the plots we can assert the rule of thumb that small data and small strides, large data and large strides make for good features regarding temporal locality. The question is then how you divide the large amount of data such that you don't land in the R/W-"mountain" to be observed in the middle of the plots. Interestingly, the peak of the Euler node is smaller by a factor of 3 compared to my local machine.

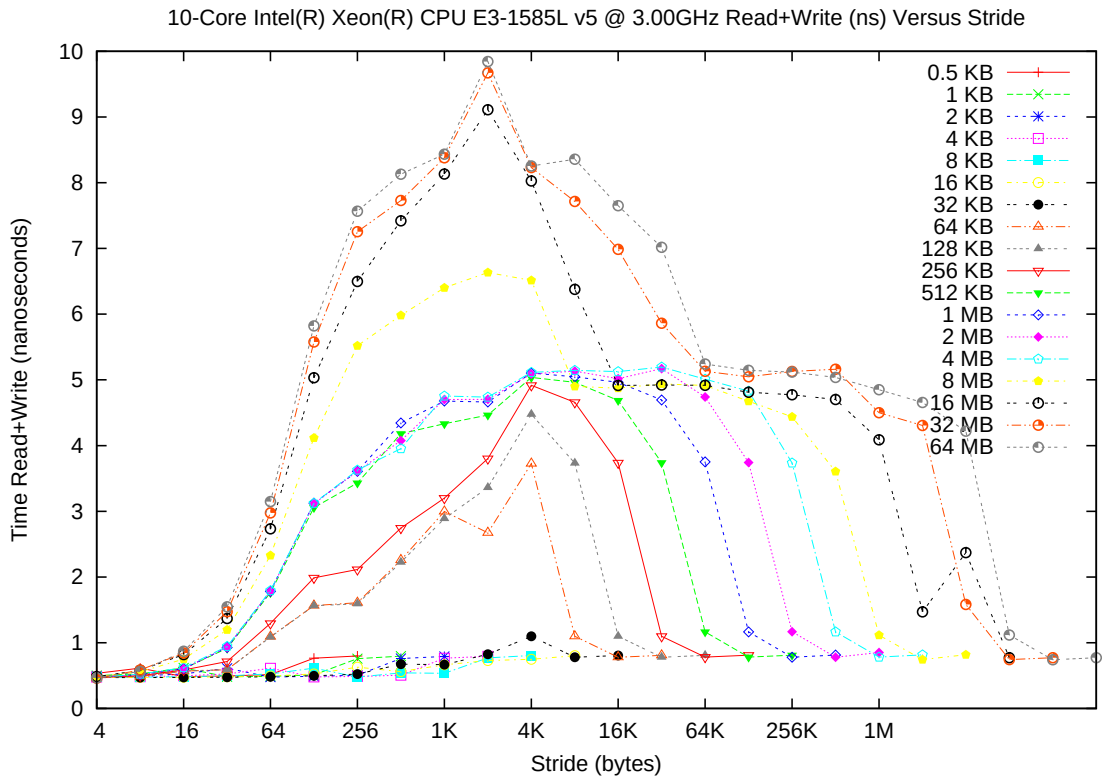

Figure 1: membench on Euler

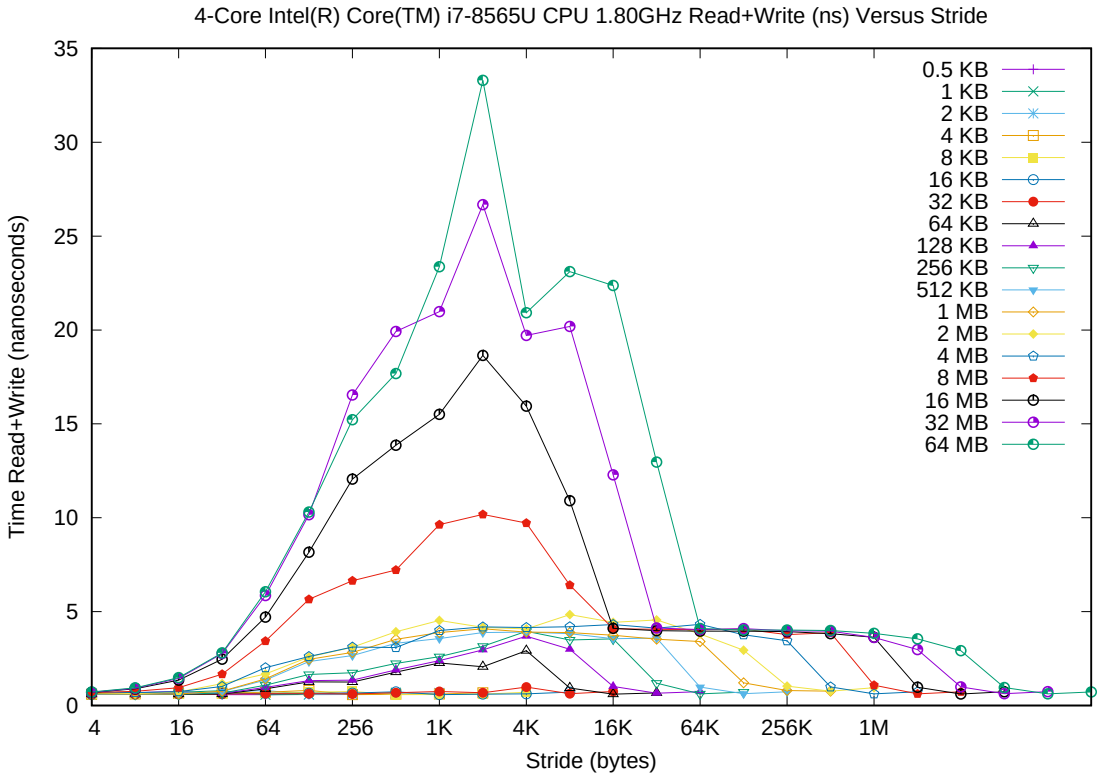

Figure 2: membench on local machine

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

I tried the following optimizations (chronologically):

- blocking (no memory, A not aligned/transposed)
- loop unrolling (manually and via -funroll-loops + pragmas)
- manual data prefetching (disabling via pragma noprefetch and then using intrinsics ie. $\_mm\_prefetch$)
- different flags, for example -fast -ansi-alias, -inline-forceinline
- pragma ivdep and pragma vector aligned
- SIMD instructions and loading the values for A manually, not having yet aligned the block
- finally understanding that the data layout has to be changed to load A into fast cache
- SIMD instructions now with FMA instead of own implementation and unrolling those manually
- pragma omp for
- pragma omp for collapse(2) nowait
- lastly, pragma omp task to maybe better share data that's locally useful via bus but this took too much time to debug

An optimization that is still to be tried and implemented would be small, inlined matrix multiplication using intrinsics.
In more detail, I tuned the blocking parameter to be 48 in my implementation which does not make too much sense to me because that number does not directly translate to any of the specs of the machines. The **blocking part** made for immediate performance gains that consistently improved the naive version. I at first did not understand how the "load A into fast memory" should be done so I deferred that to be done at a later point.
Loop unrolling made for small but also measurable progress towards making the procedure faster - the manual one that is. Trying to invoke different pragmas and compiler flags did not lead to the same kind of progress.
Manual prefetching always deteriorated the performance. The lesson to be learned is probably that compiler engineers know best how to make a compiler understand or that some access patterns need to be understood more deeply in connection to how the instructions are issued such that manual prefetching becomes productive.
Interestingly, only the -inline-forceinline flag together with the corresponding keyword in the code made for speedup using **function inlining**. Since function inlining always stays a recommendation to the compiler one cannot be too sure about it, even if it is stated explicitly. Enabling strict aliasing seemed like a good idea but it did not enhance performance overall.
The pragmas to **enable vectorization** (vector aligned) to the compiler did make the program faster but it seemed suboptimal to what can be achieved using well-tuned vectorization measures.
The first try at embedding the blocked matrix multiplication into the vectorized framework **using intrinsics** increased performance while also dipping at "bad" problem sizes.
At this point I finally reread the Project description and realized that for the kernel to be fast, A has to be loaded into "fast memory" - to make contiguous accesses to A I **introduced a transposition** to make it align well with the data layout. Accesses to B are, by the operation's definition, already col-major so nothing had to be done there. This improved matters a lot.
After the data was finally well aligned in fast cache I built in **Intel Intrinsics** (FMA for the AVX2-enabled lanes) with unrolled loops (row/col-wise, not block-wise) which yielded satisfying results. Naturally, this was not at all close to coming near the BLAS/LAPACK benchmark but it tripled the performance of the baseline implementation.
Finally I built at first **simple OpenMP pragmas** without a lot of thought into the single-core optimized blocked version which worked well.
To get a final result I introduced the **collapse clause** together with thread-private variables. Additionally, nowait was worked in to avoid useless synchronization steps.
Trying to work through OpenMP tasks proved difficult as there is a lot more nuance to how data can be shared and dependencies need to be declared to fully use the bus. This optimization hasn't been further pursued.
I did not achieve loading B and C into fast memory as proposed in the Project1 description of the blocked algorithm. Surely there can be achieved more.

I calculated the following for MAX_SPEED (hopefully I modified benchmark.c in the right way for all cases):

single core local: 1.8GHz x 8 single float lanes x 2 FLOPS/FMA
multi core local: likewise x 4 cores
single core Euler: 3 GHz x 8 single float lanes x 2 FLOPS/FMA = 48 GF/s
multi core Euler: 4 cores x 3GHz x 8 single float lanes x 2 FLOPS/FMA = 192 GF/s

Additionally, I used Intel's icc compiler for everything (module load intel/2018.1 usually). I put up a version that's cleaned up without the optimizations that didn't prove useful.

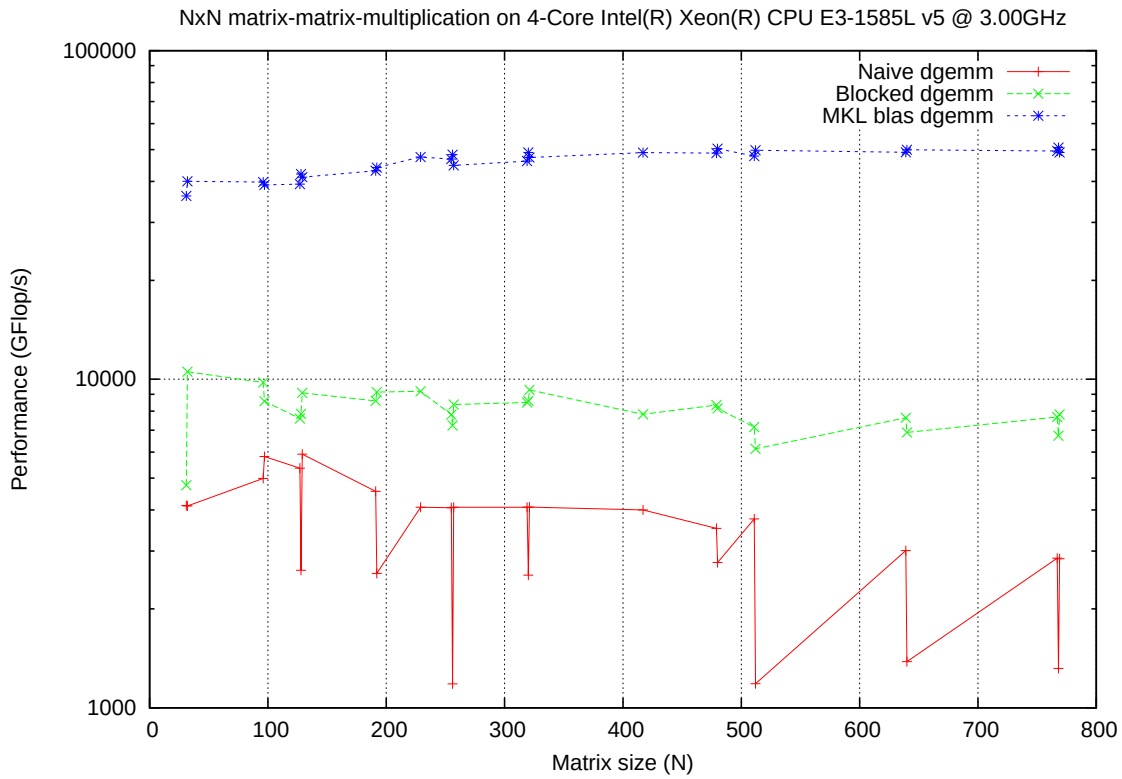

Figure 3: benchmark on Euler, single core

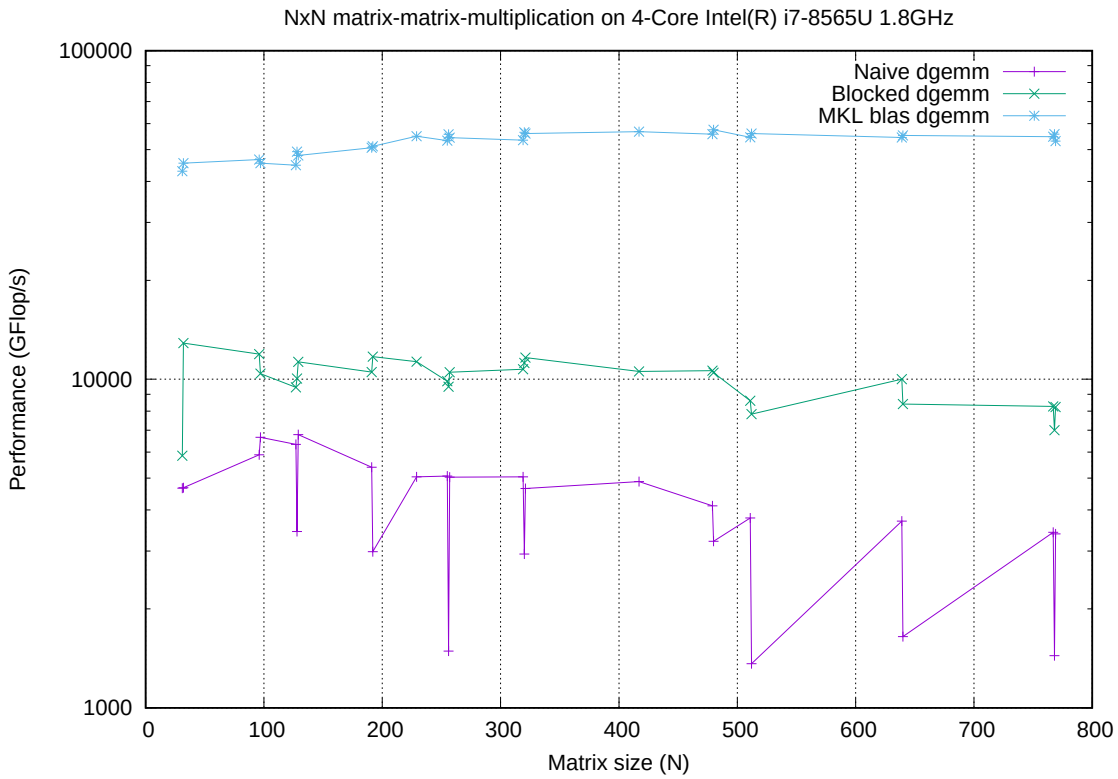

Figure 4: benchmark on local machine, single core

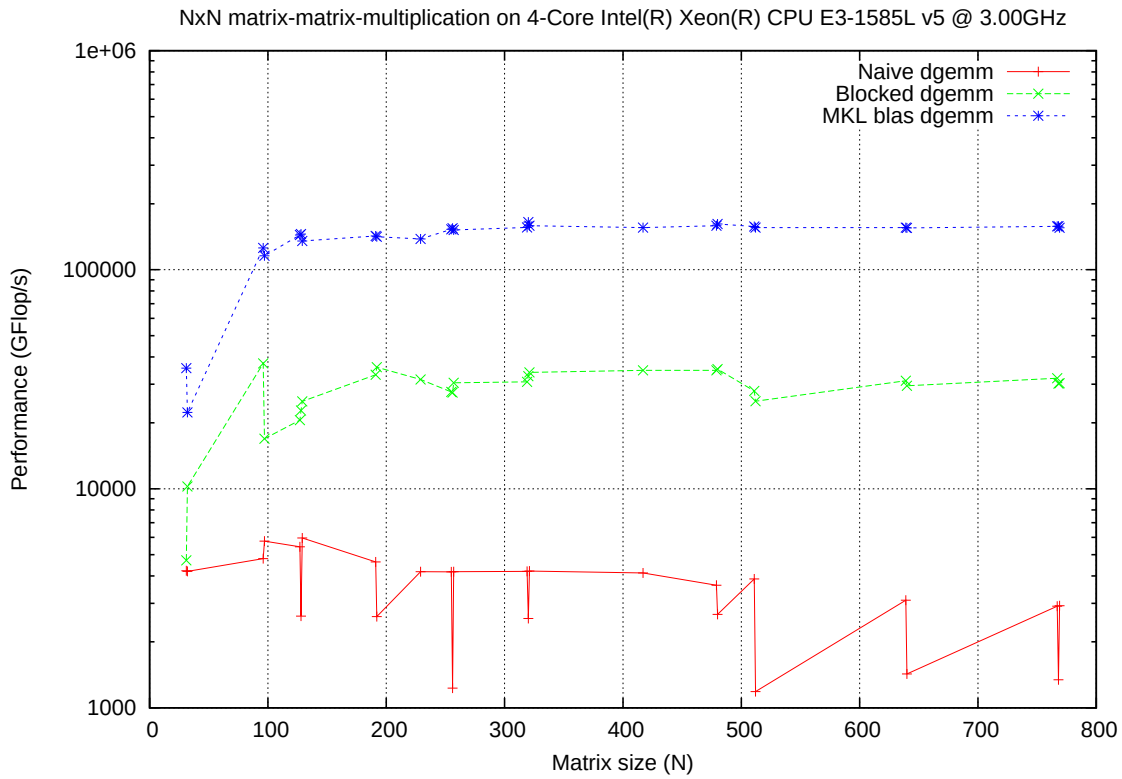

Figure 5: benchmark on Euler, multithreaded (4)

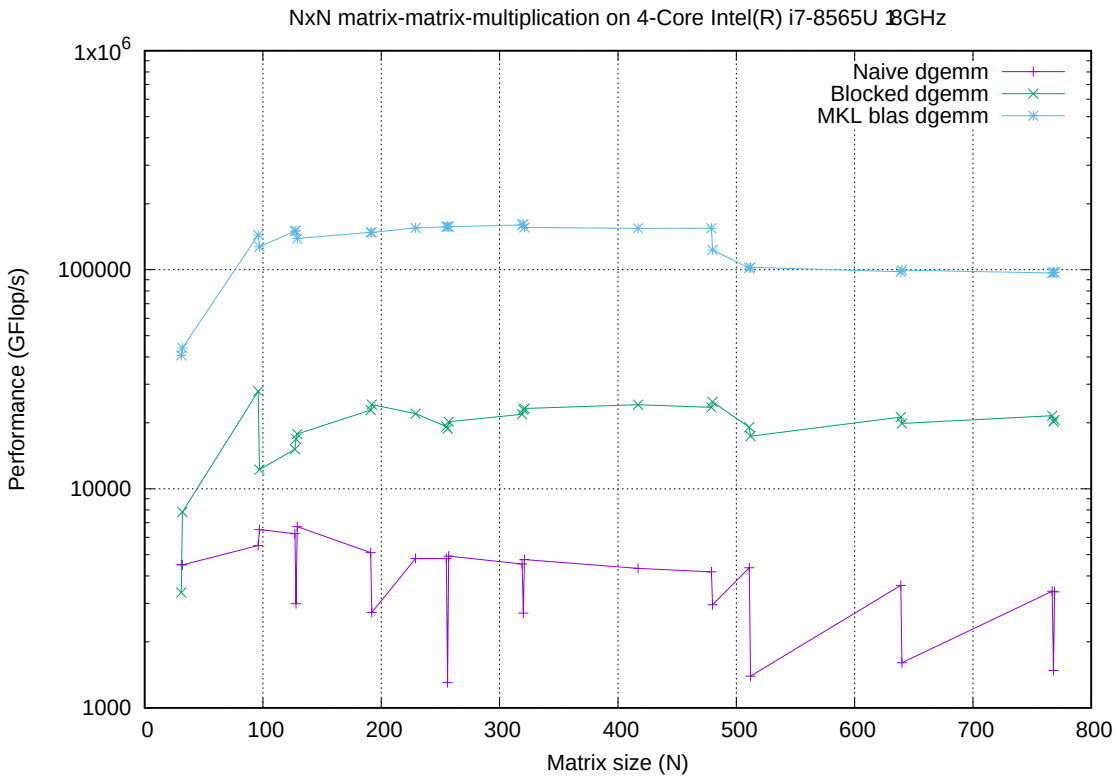

Figure 6: benchmark on local machine, multithreaded (4)