Q&A for SC21 tutorial “Node-Level Performance Engineering”

Q: **Can you make a Xeon CPU with a large L3 cache act like it has a smaller L3 cache for the purpose of testing how code will run on a CPU with a smaller L3 cache?**

A: Cache pirating: https://ieeexplore.ieee.org/document/6047185 limits the available amount of cache by running a separate application that “steals” a well-defined amount of cache away from the app you want to study. Cache Allocation Technology on Intel CPUs (see Sect. 17.17 in https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf) is a hardware feature which allows you to limit the available amount of cache for a core. Other vendors (AMD, Arm-based) have similar technologies that go under different names.

Q: Is SMT detrimental in some situations? Perhaps for very parallel workloads, like large Matrix operations.

A: On Intel and AMD CPUs, most relevant resources are available to a single running thread even if SMT is enabled. This is not the case on other designs; e.g., on the Cavium/Marvell ThunderX2 the single-thread performance may suffer for some codes if SMT is on. Another effect is shared resources: If you run a software thread on every SMT hardware thread, all those software threads need resources (first and foremost cache). If the resources are scarce, this could mean that performance may suffer. We’ll see an example for such a code later in the part about stencil algorithms.

Q: Does the performance equation you provided, used in slide 20 presentation 01, account for memory transaction speed?

A: No it doesn’t. This is the pure “in-core” peak performance. It only considers computational resources.

Q: MPI rank mapping: What are the major considerations when thinking about mapping to L1, L2, L3 cache, ccNUMA?

A: There is no single answer. It depends on the hardware bottlenecks your code is up against and which resources it needs. (1) Affinity should always be enforced, no matter if you use OpenMP threads or MPI processes or something else. (2) If you know that each thread/process needs a lot of cache *and* your code is memory bound then you might want to consider not using all cores and “spread out” the threads/processes across the chip. If you know that neighboring processes communicate a lot, you may want to place them close together so they can use the shared cache(s) for exchanging messages. (3) For MPI codes, ccNUMA locality is often not much of a problem because each process allocates and initializes its own memory.

Q: When doing microbenchmarking, do you think it is best to disable turbo boosting so the CPU clock is a constant frequency? The ratio of CPU clocks to DRAM clocks would be changed if turbo boost is disabled unless the DRAM clock frequency is also reduced.

A: During microbenchmarking, we always fix the clock frequency (i.e., no Turbo). This just takes out one possible element of variation. And yes, the CPU-to-DRAM clock cycle ratio is changed if you change the CPU clock only; you have to take this into account if memory access is a relevant factor in the analysis. But even if you leave Turbo Mode on, you always have to measure what *actual* frequency the core/s was/were running at because this may change depending on the code.

Q: What is your opinion of using the GCC vector extensions (https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html) as a way to write code with good AVX-512 performance?

A: Whatever makes the compiler produce good code is OK. We usually try everything possible to write down plain C/Fortran and make the compiler understand what we want. Only if this fails do we recommend to revert to more low-level methods such as intrinsics or vector extensions, but these make the code less portable. As for how good the GVE code is compared with, e.g., intrinsics, we cannot tell - but I would favor intrinsics because more compilers support it (e.g., not all of the GVE is supported by the Intel compiler).

Q: The case study: stencils --- Is this from a published paper that I can read and cite?

A: The idea of layer conditions has been around for a while, although it was us who coined the term. The best writeup of the layer condition concept is probably in:
H. Stengel, J. Treibig, G. Hager, and G. Wellein: *Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory model*. Proc. ICS15, the 29th International Conference on Supercomputing, June 8-11, 2015, Newport Beach, CA.
DOI: 10.1145/2751205.2751240
Some useful links:

- Layer condition calculator: https://rrze-hpc.github.io/layer-condition
- Kerncraft modeling tool: https://github.com/RRZE-HPC/kerncraft
- INSPECT Intra-Node Stencil Performance Evaluation Collection: https://rrze-hpc.github.io/INSPECT/

Q: Are all SpMVM problems unique and require alpha to be derived from previous runs?

A: Basically, yes. However, in some cases alpha will be small - so small as to be negligible. This happens when the RHS fits completely in the cache. The optimistic (lower bound) nonzero alpha is

1/N_nzr. Anything beyond these special cases is a function of the matrix parameters (sparsity pattern, sizes) and the hardware (cache sizes, number of threads sharing a cache).

Q: Any thoughts on which compiler tends to perform better (on average) related to SIMD auto-vectorization? .e.g. comparing intel vs gcc.

A: There is no general answer. In cases where the compiler can really see through all abstractions and understands 100% what's going on, Intel tends to generate very good (sometimes beautiful) SIMD code. Looking through abstractions, however, was not its strong suit. But again, this is a moving target and such observations may change quickly. Also don't forget that SIMD isn't the only thing that gives you performance.

Q: In the SpMVM comparison between 3 chips, the fastest hardware is not constant or tied to Gbyte/s. How do you choose the proper hardware for a problem?

A: In the data on slide 15 of the SpMVM lecture, the A64FX has the highest memory bandwidth but it is not the fastest on all matrices. There are several reasons for that, but the crucial insight is that raw memory bandwidth, while important for SpMV, is not everything. Specifically, A64FX has some special properties: Even with optimal code it needs almost all 12 cores of a ccNUMA domain to saturate the memory bandwidth with SpMV. This means that anything that goes wrong (e.g., load imbalance or a compiler not producing optimal code) will immediately cause a performance hit. The GPU is much more tolerant in this respect because of its inherently dynamic execution and massive parallelism. On top, the A64FX has four ccNUMA domains. If you need dynamic scheduling to balance the load (because the matrix doesn't have a nicely even distribution of nonzeros), this is a problem because you will have nonlocal accesses and contention. This is not an issue on the GPU.