HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 10

Fabian Wermelinger
Harvard University
CS205

Thursday, February 23rd 2023

LAST TIME

- Uniform and non-uniform memory access (UMA and NUMA)
- Thread affinity and the NUMA first-touch policy
- Cache coherency protocols in shared memory (the missing piece in the [shared] memory consistency model)
- False sharing

TODAY

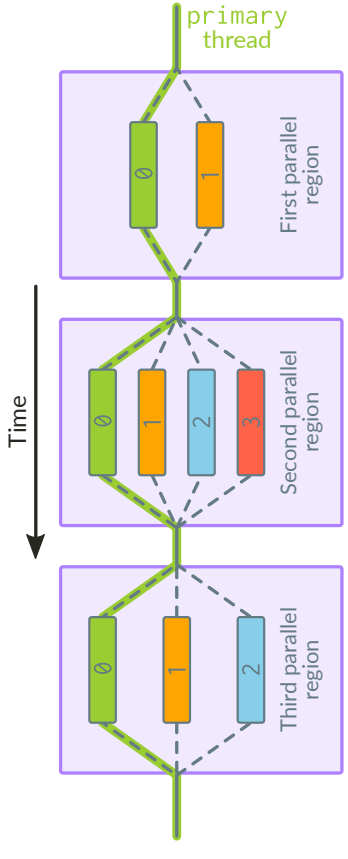
Main topic: ***Node-level performance analysis, Roofline model, hardware limitations***

Details:

- A few more OpenMP details regarding overhead and wait policies for threads
- What limits your performance on a single node?
- Computing nominal peak performance and peak bandwidth for CPUs (and GPUs)
- The operational intensity of compute kernels (algorithms in software)
- The roofline model
- Example performance analysis
- Topics of the reading assignment will show up throughout this lecture

OpenMP PERFORMANCE

What happens after OpenMP has spawned the first parallel region encountered in your code?



- Spawning and joining threads is expensive (it is cheaper than for processes but *still not for free*).
- Spawn and join operations involve the OS. Anything that involves the OS is slow (e.g. dynamic memory allocation).
- The OpenMP runtime spawns threads only *once* when it encounters the first parallel region. Threads are *re-used* at the next parallel region.
- This means that when threads are *joined*, only the primary thread continues and the others become *idle*.

OpenMP PERFORMANCE

How should *idle* threads spend their time?

- You can provide a *hint* to the OpenMP runtime how threads should wait

```
1 OMP_WAIT_POLICY='active'  # mostly active, i.e., consume CPU cycles
2 OMP_WAIT_POLICY='passive' # mostly passive, i.e., yield CPU resources or sleep
```

- *The OpenMP runtime is not required to follow your hint*. If you do not define the `OMP_WAIT_POLICY` variable then the behavior is *implementation defined*.
- You cannot change this variable once your program runs (→ *therefore environment variable option only*).

Wait policies:

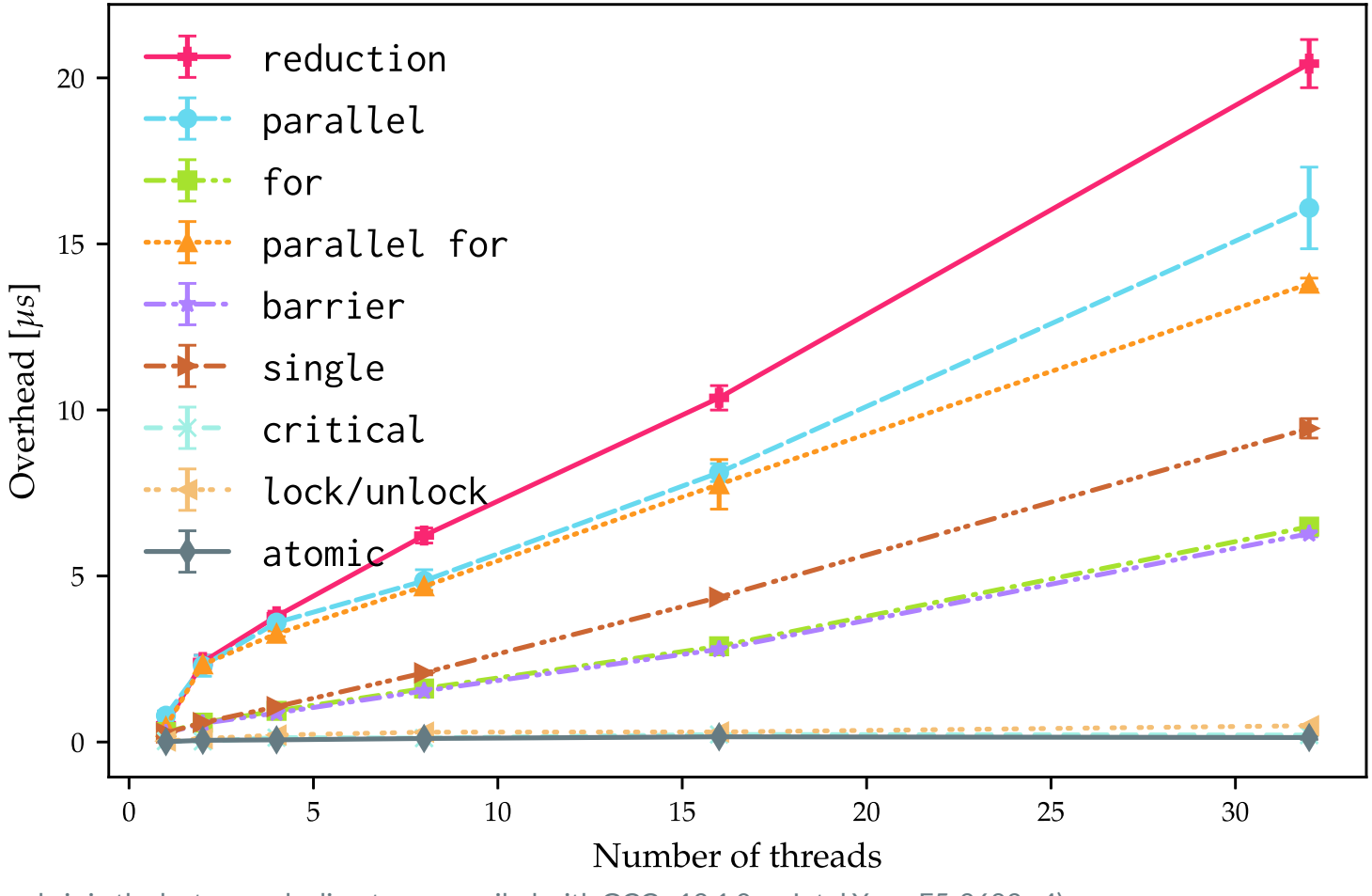
`active`
: Waiting threads should mostly be active. This means they consume CPU cycles while waiting. An example of this policy is the *spin-lock* (Peterson lock) where threads spin in a `while` loop.

`passive`
: Waiting threads should mostly be passive. This means they are suspended (yield or sleep) and do not consume CPU cycles. *It takes longer to wake up a passive thread*.

OpenMP CONSTRUCT OVERHEAD

What are the costs of using OpenMP constructs?

Compute node on the academic cluster:



(Microbenchmark code is in the lecture code directory, compiled with GCC v12.1.0 on Intel Xeon E5-2683 v4)

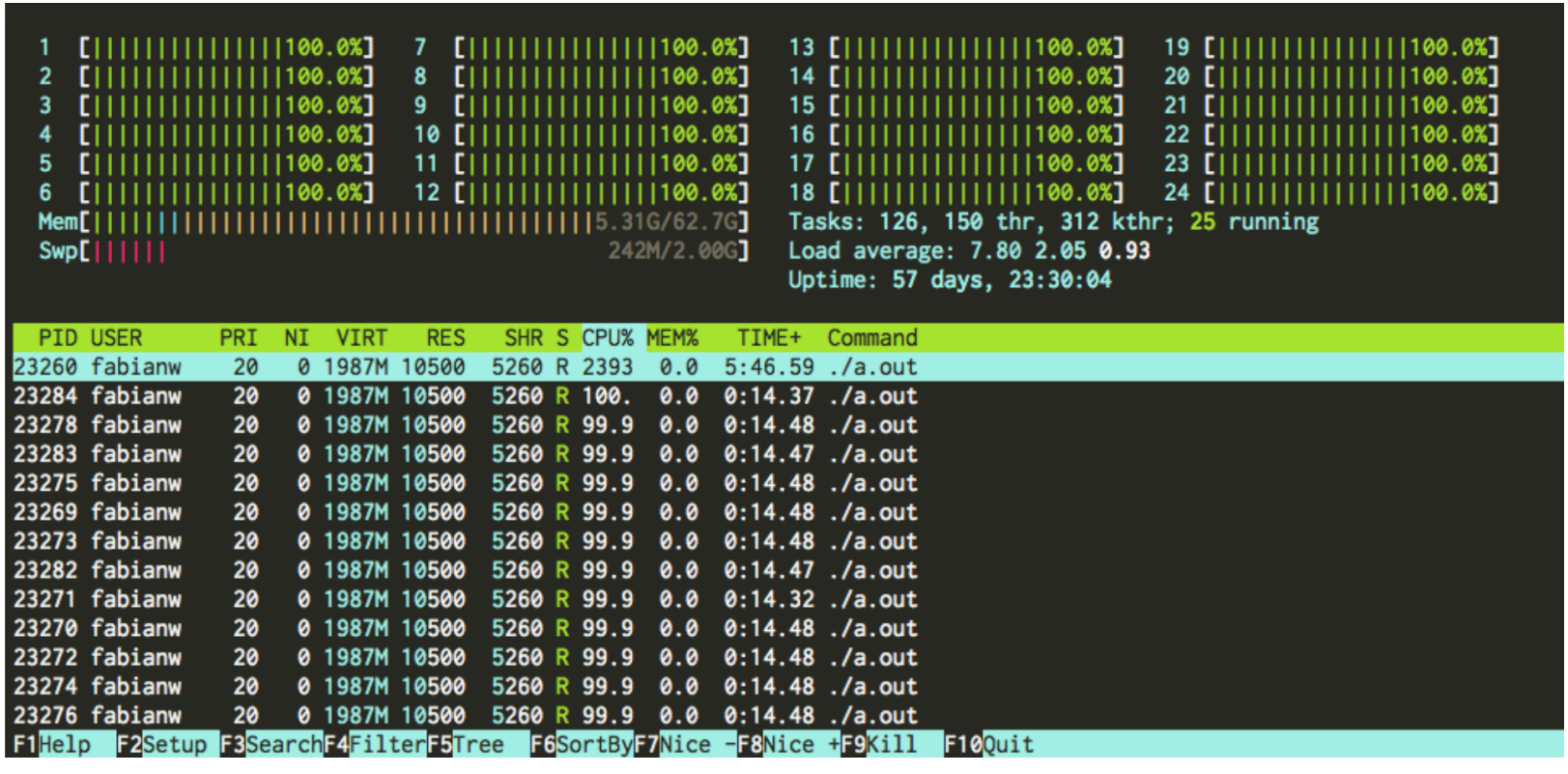
HOW DO YOU KNOW YOU DID A GOOD JOB?

- You are now at the point where you can write parallel code using shared memory.
- The next thing you should wonder about is how well did you parallelize the code and how good are your memory access patterns given the data structures you chose (***recall:*** *efficient use of caches is of utmost importance!*).
- There are two metrics you want to look at:
  1. ***Parallel scalability:*** How well does your parallelization *scale* when you add more processing units. *Communication overhead* will be the main concern here. If the communication *latency* is not hidden well, your code will not scale well. We wait until Lecture 15 when we have covered MPI.
  2. ***Node-level performance:*** How well does your parallel implementation exploit the different *forms of parallelism on the hardware* (e.g. one CPU only or full node with multiple sockets). Recall the forms of parallelism *on the hardware:*
     - Instruction-level parallelism (**ILP**; *example: processor pipelining*)
     - Data-level parallelism (**DLP**; *example: SIMD* → synonymous with *vectorization*)
     - Thread-level parallelism (**TLP**; *example: multicore*; we are not interested in communication overhead here but whether all your cores are busy or not)

HOW DO YOU CHECK NODE-LEVEL PERFORMANCE?

When you buy ***expensive hardware*** (CPU, GPU, etc.), *how can you know the level of hardware utilization?* (Return of investment)

Maybe you run a system monitor like `htop`:

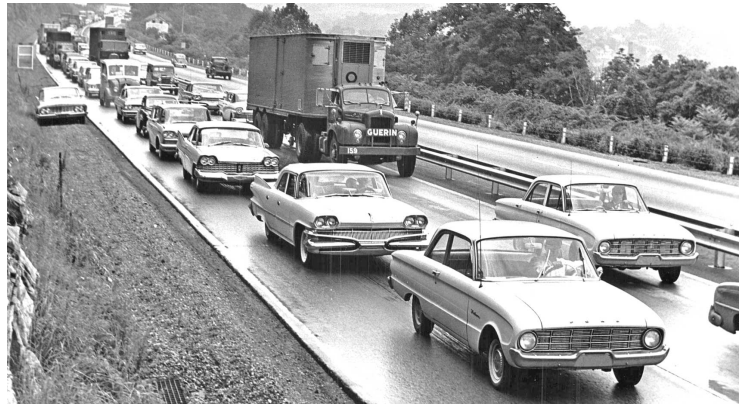


...but you should be critical when you see this! All cores at 100% what does this mean?

WHAT ARE THE HARDWARE LIMITS?

Recall: warehouse/factory situation

Warehouse (main memory)



Traffic (interconnect/bus)

Factory (processor)

- There are *two* main components on any computing platform which impose the limit on performance. ***What are they?*** (Reading assignment for today) You cannot do better than these hardware limits due to physical constraints.
- You already learned techniques to write software that improves *utilization* for these two hardware components.
- ***However:*** the performance of (most) your software is limited by *memory bandwidth*. *Understanding caches optimizes for peak memory bandwidth!*

OPERATIONAL INTENSITY

Operational intensity I expresses a relationship between

$$I = \frac{\text{Total number of operations (compute performance)}}{\text{Total bytes of DRAM traffic (memory performance)}}$$

in one single number. It depends on both, the architecture you run on *and* the compute kernels your are optimizing (i.e., your software). Operational intensity I tells you *the number of operations performed for every byte of DRAM traffic.*

- $I \gg 1$: the compute kernel performs many operations on any byte fetched from DRAM (with the help of caches). The kernel performance will be limited by the nominal peak operational performance (e.g., floating-point operations Gflop/s) imposed by the hardware (*compute bound*).
- $I \lessapprox 1$: the compute kernel is dominated by DRAM memory accesses with only few operations performed on each byte fetched. The kernel performance will be limited by the nominal peak memory bandwidth on the hardware (*memory bound*).

NOMINAL PEAK ARITHMETIC PERFORMANCE

- The nominal peak arithmetic performance is specific for a given architecture and depends on factors that must be looked up in the hardware documentation and software optimization manuals.
- Alternatively, microbenchmarks could be used to measure a nominal peak performance metric. Such a microbenchmark again depends on what the hardware supports.
- Relevant parameter include:
  - Processor clock rate
  - Number of cores on the CPU
  - The maximum number of flop that can be performed in every cycle
  - The SIMD vector width (depends on the data precision of your compute kernel)
- These parameter combine all the hardware parallelism we talked about:
  - **DLP:** SIMD vector width
  - **ILP:** clock rate, pipelining, maximum number of flop per cycle
  - **TLP:** number of cores

NOMINAL PEAK ARITHMETIC PERFORMANCE

Example: Intel Xeon E5-2683v4 (Broadwell architecture, early 2015)

We can look-up most parameter at the given link above:

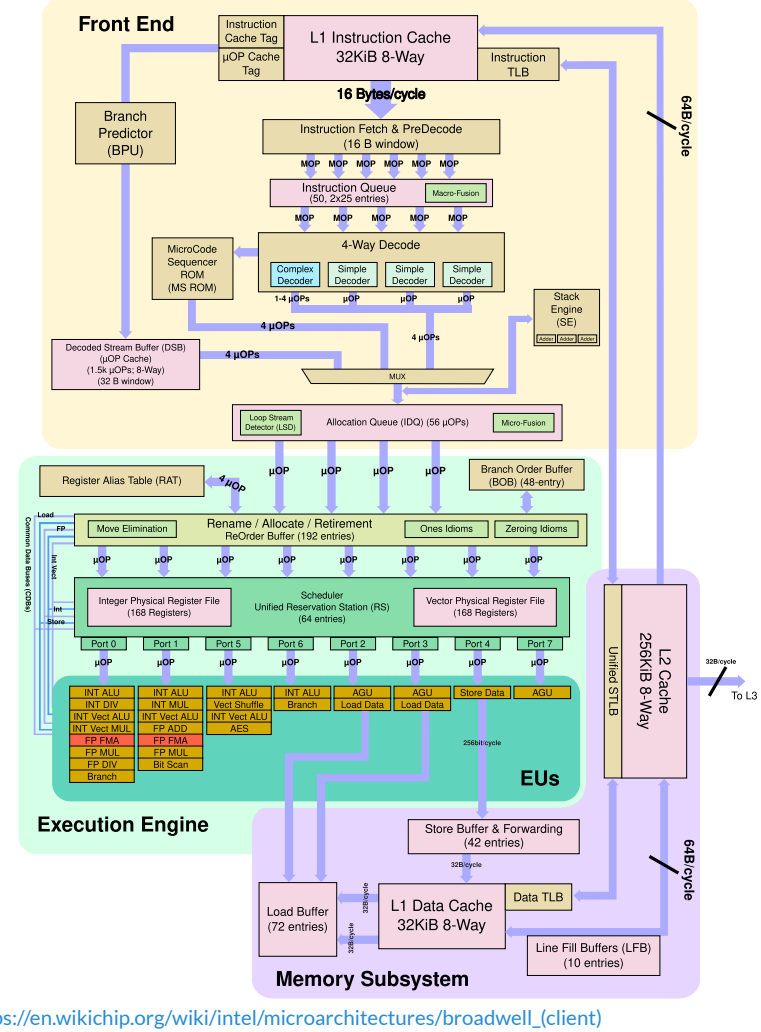
Base clock rate	$f = 2.10 \times 10^9 \,\mathrm{cycle/s}$
Number of cores	$n_c = 16$
Precision (single or double)	$p\,\mathrm{bit}$
SIMD vector width (AVX2)	$w_s = 256\,\mathrm{bit}$
SIMD lanes	$l_s = \frac{w_s}{p}$
Total flop per cycle (supports FMA3 instructions)	$\phi = 4\,\mathrm{Flop/cycle}$

Nominal peak arithmetic performance:

$$\pi = f \times n_c \times l_s \times \phi$$

Single precision	$\pi = 1075.2\,\mathrm{Gflop/s}$
Double precision	$\pi = 537.6\,\mathrm{Gflop/s}$

From the hardware manual:



https://en.wikichip.org/wiki/intel/microarchitectures/broadwell_(client)

NOMINAL PEAK MEMORY PERFORMANCE

Example: Intel Xeon E5-2683v4 (Broadwell architecture, early 2015)

We can look-up the peak memory bandwidth at the link above:

Peak memory bandwidth	$\beta = 76.8\,\mathrm{GB/s}$
Supported memory types	DDR4 1600/1866/2133/2400
Maximum memory channels	$c = 4$

DDR

Memories based on the DDR (double data rate) technology transfer *two* units of data per clock cycle. As a result they achieve double the transfer rate which is usually reflected in their labeled *frequency*. *Example:* the DDR4 2400 MHz memory above operates at an actual frequency of $f_{\mathrm{DDR}} = 1200\,\mathrm{MHz}$.

Maximum memory channels

The maximum number of channels that can transfer data units in parallel.

If you need to compute the peak bandwidth:

You must use the corresponding modules installed on your system or assume the fastest supported memory modules if you do not know better.

Nominal peak memory performance:

$$\beta = 2 \times f_{DDR} \times c \times w$$

w is the number of *bits* moved through *one* channel per cycle, typically 64 bit (word size). ***For this example:***

$$\begin{aligned}\beta &= 2400 \times 4 \times 64 \times 1/8 \\ &= 76800.0\,\mathrm{MB/s} = 76.8\,\mathrm{GB/s}\end{aligned}$$

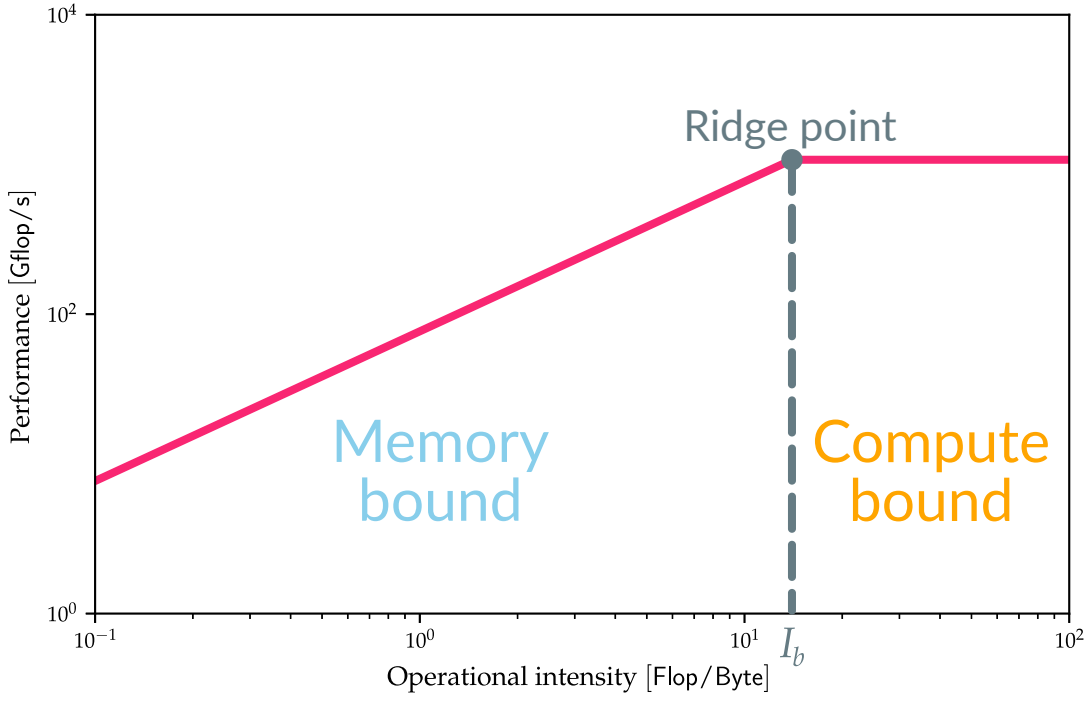
The factor $1/8$ converts bits to bytes.

THE ROOFLINE

The Intel Xeon E5-2683v4 CPU has the following *hardware limits* for single precision computations:

- $\pi = 1075.2\,\mathrm{Gflop/s}$
- $\beta = 76.8\,\mathrm{GB/s}$

Single precision roofline for Intel Xeon E5-2683v4:



- The Roofline model ties together *floating-point performance*, *operational intensity*, and *memory performance* in a 2D graph (Williams et al., 2009).
- The *ridge point* characterizes machine balance:

$$I_b = \frac{\pi}{\beta}$$

$$I < I_b \quad (\text{memory bound})$$

$$I > I_b \quad (\text{compute bound})$$

- The *attainable performance ceiling* is given by:

$$P = \min(\beta I, \pi)$$

THE ROOFLINE

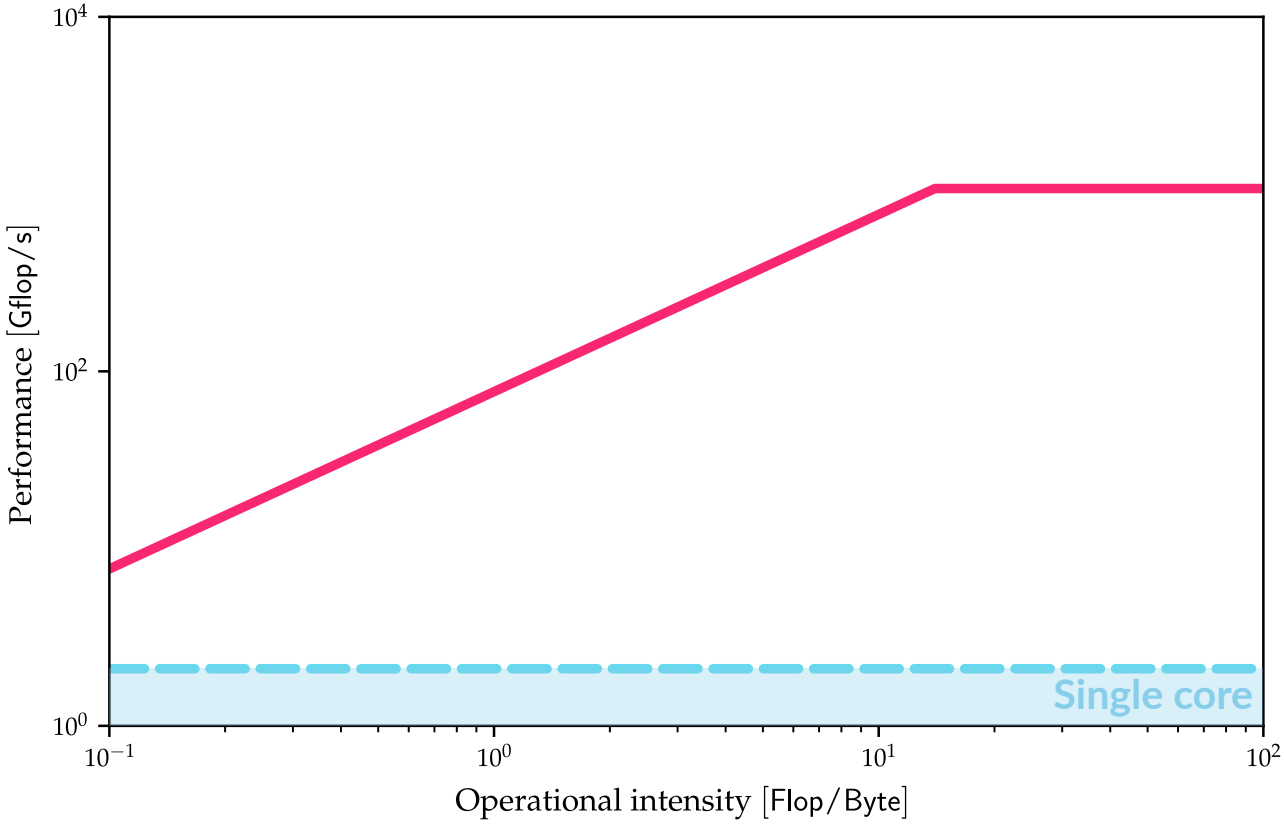
- The roofline model is a simple model based on a "bound and bottleneck" analysis (Williams et al., 2009).
- It provides valuable insight into the *primary factors* that affect performance by means of a 2D *log-log* plot.
- The main *system bottleneck* is highlighted and quantified.
- These bottlenecks are *computing performance* and *off-chip memory traffic* where *the latter is the **constraining** resource.*
- The relation of these two bottlenecks requires a new concept which is the *operational intensity* with units *flop per byte* (traffic between off-chip memory and processor).

The roofline model is crucial for:

1. Performance predictions
2. Helpful for software optimization

ROOFLINE CEILINGS FOR OPTIMIZATION GUIDANCE

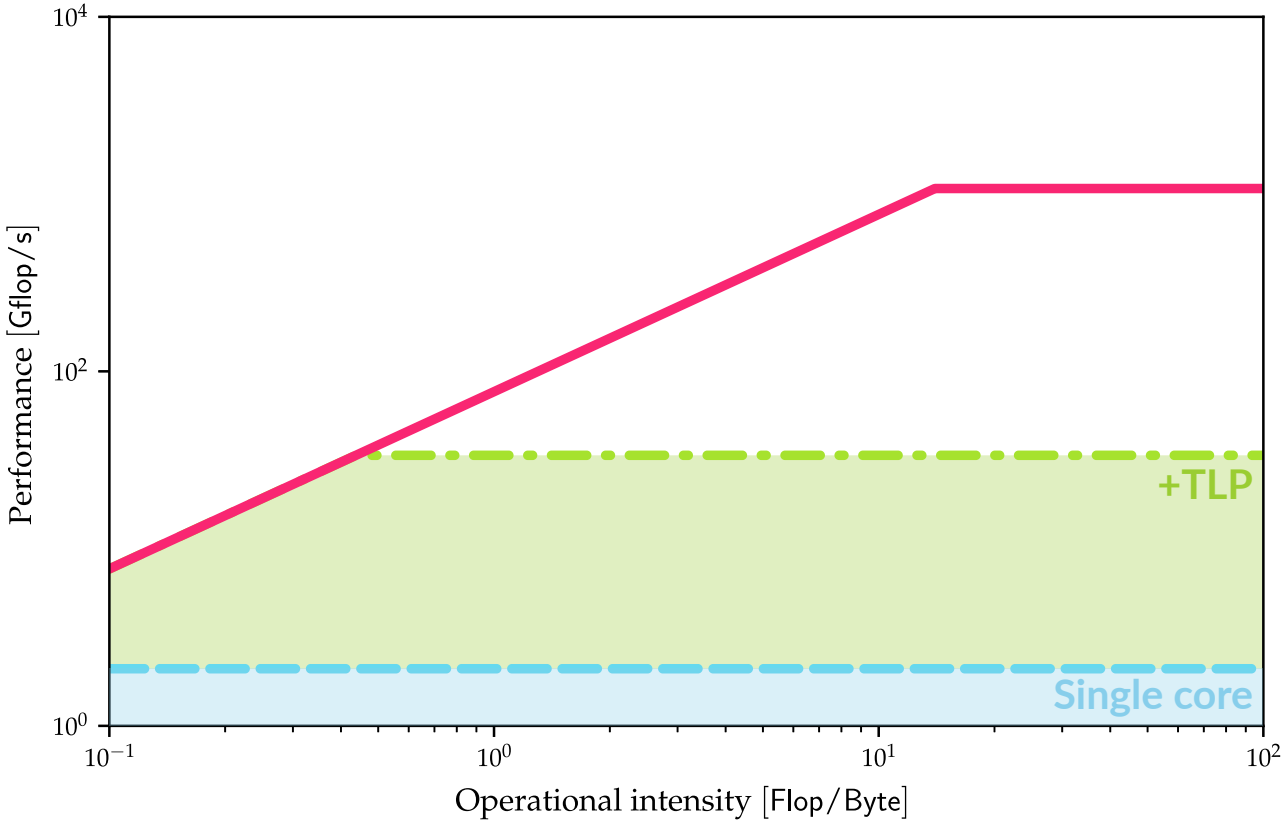
The ceiling for a sequential code:



- The example roofline on the right is for the Intel Xeon E5-2683v4 CPU.
- Sequential code cannot break through this ceiling.
- A single core *can not saturate* the memory bandwidth (we saw this in last lecture).
- Many resources wasted!

ROOFLINE CEILINGS FOR OPTIMIZATION GUIDANCE

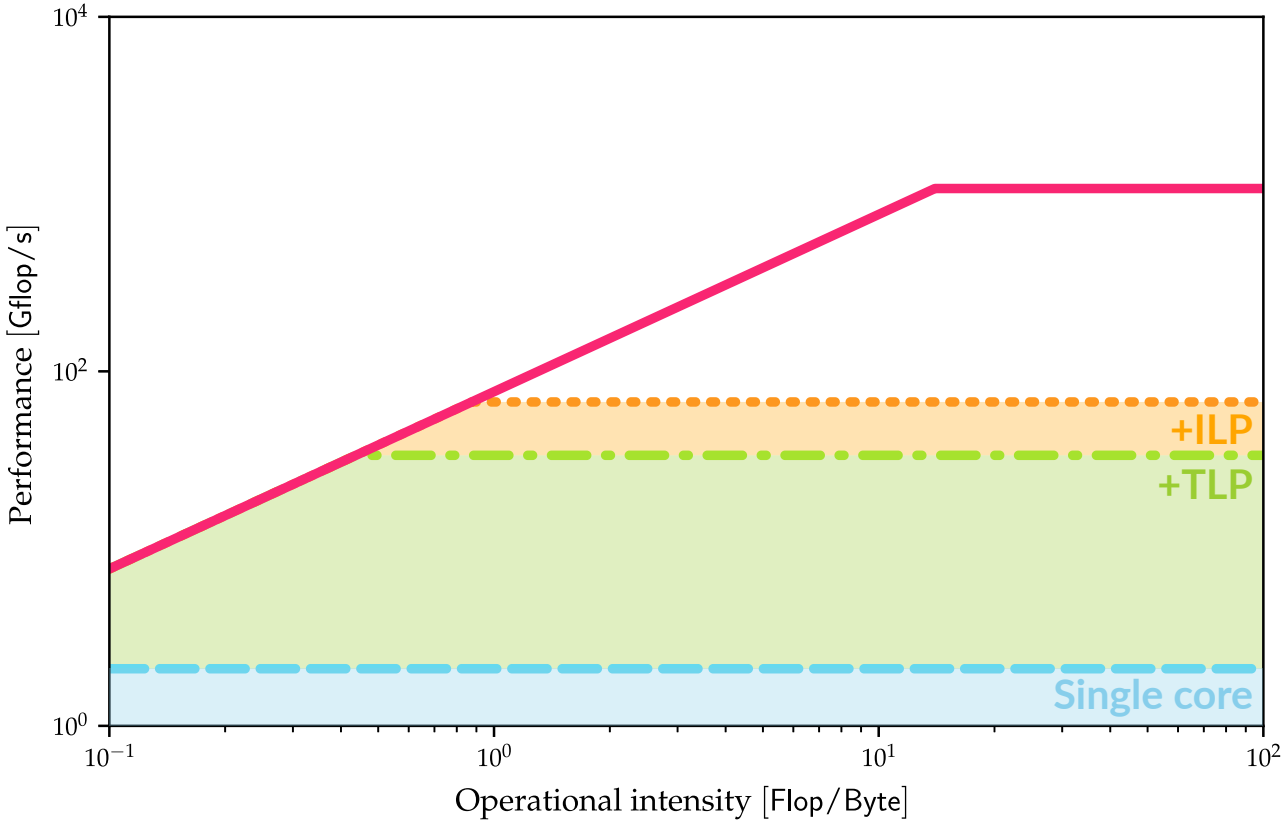
The ceiling for a multicore code:



- The example roofline on the right is for the Intel Xeon E5-2683v4 CPU.
- The easiest and first optimization to exploit the hardware is adding thread-level parallelism (TLP).
- OpenMP is one tool to implement this ceiling and can be achieved with small effort.

ROOFLINE CEILINGS FOR OPTIMIZATION GUIDANCE

The ceiling for added instruction-level parallelism (ILP):



- The example roofline on the right is for the Intel Xeon E5-2683v4 CPU.
- Many recent CPUs have multiple execution units for the same instruction, all of them can be executed in the *same cycle*.
- Such a CPU is called *superscalar*.
- One way to increase ILP is by unrolling loops. Many times the compiler will do this automatically when you specify (aggressive) optimizations, e.g., the `-O3` flag for GCC.

ROOFLINE CEILINGS FOR OPTIMIZATION GUIDANCE

The ceiling for added instruction-level parallelism (ILP):

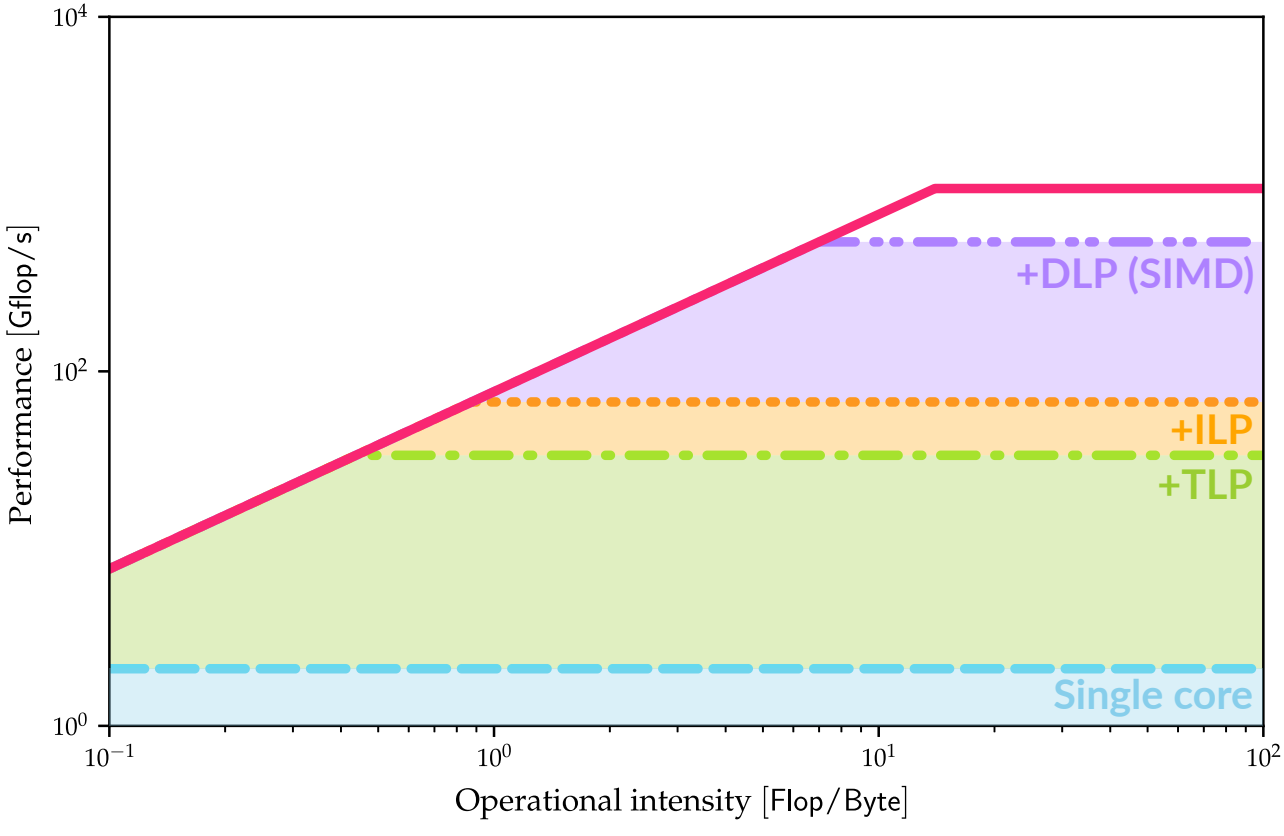
```c
int main(void)
{
    double A[1024];
    double B[1024];
    double C[1024];
    // manual 4-fold loop unroll
    for (int i = 0; i < 1024; i += 4) {
        A[i + 0] = A[i + 0] * B[i + 0] + C[i + 0];
        A[i + 1] = A[i + 1] * B[i + 1] + C[i + 1];
        A[i + 2] = A[i + 2] * B[i + 2] + C[i + 2];
        A[i + 3] = A[i + 3] * B[i + 3] + C[i + 3];
    }

    return 0;
}
```

- Example of a manual loop-unroll
- Most of the time you do not need to do it manually as the compiler will do that optimization for you.
- When you do, it will be for a *highly optimized specialization* of a specific compute kernel for which you have already applied the basic optimizations and need more fine tuning.

- The example roofline on the right is for the Intel Xeon E5-2683v4 CPU.
- Many recent CPUs have multiple execution units for the same instruction, all of them can be executed in the *same cycle*.
- Such a CPU is called *superscalar*.
- One way to increase ILP is by unrolling loops. Many times the compiler will do this automatically when you specify (aggressive) optimizations, e.g., the `-O3` flag for GCC.

ROOFLINE CEILINGS FOR OPTIMIZATION GUIDANCE

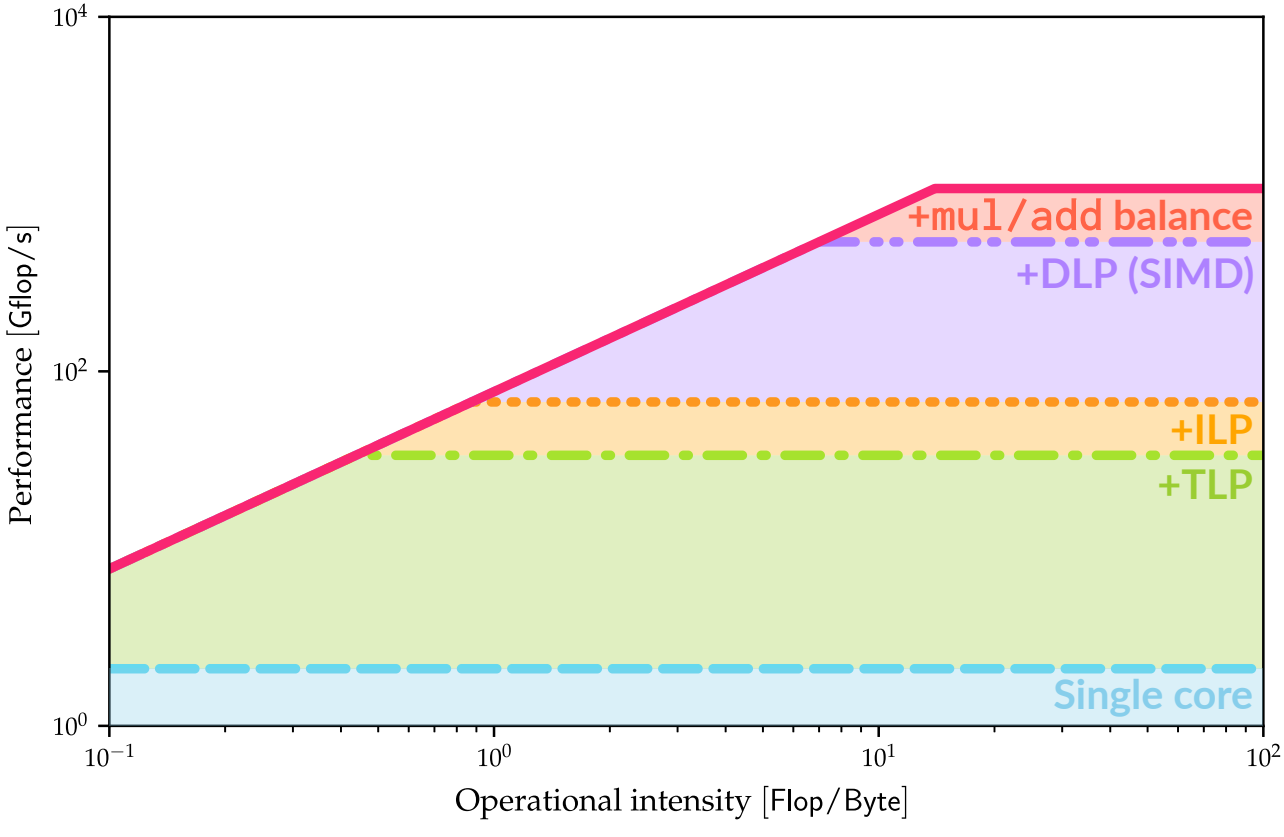
The ceiling for added data-level parallelism (DLP):



- The example roofline on the right is for the Intel Xeon E5-2683v4 CPU.
- Data-level parallelism involves making use of vector instructions that can perform an operation on multiple data elements (SIMD).
- The compiler can help with this optimization as well but the result depends on your algorithm as well. The compiler not always generates good vectorized code with its auto-vectorization capabilities.

ROOFLINE CEILINGS FOR OPTIMIZATION GUIDANCE

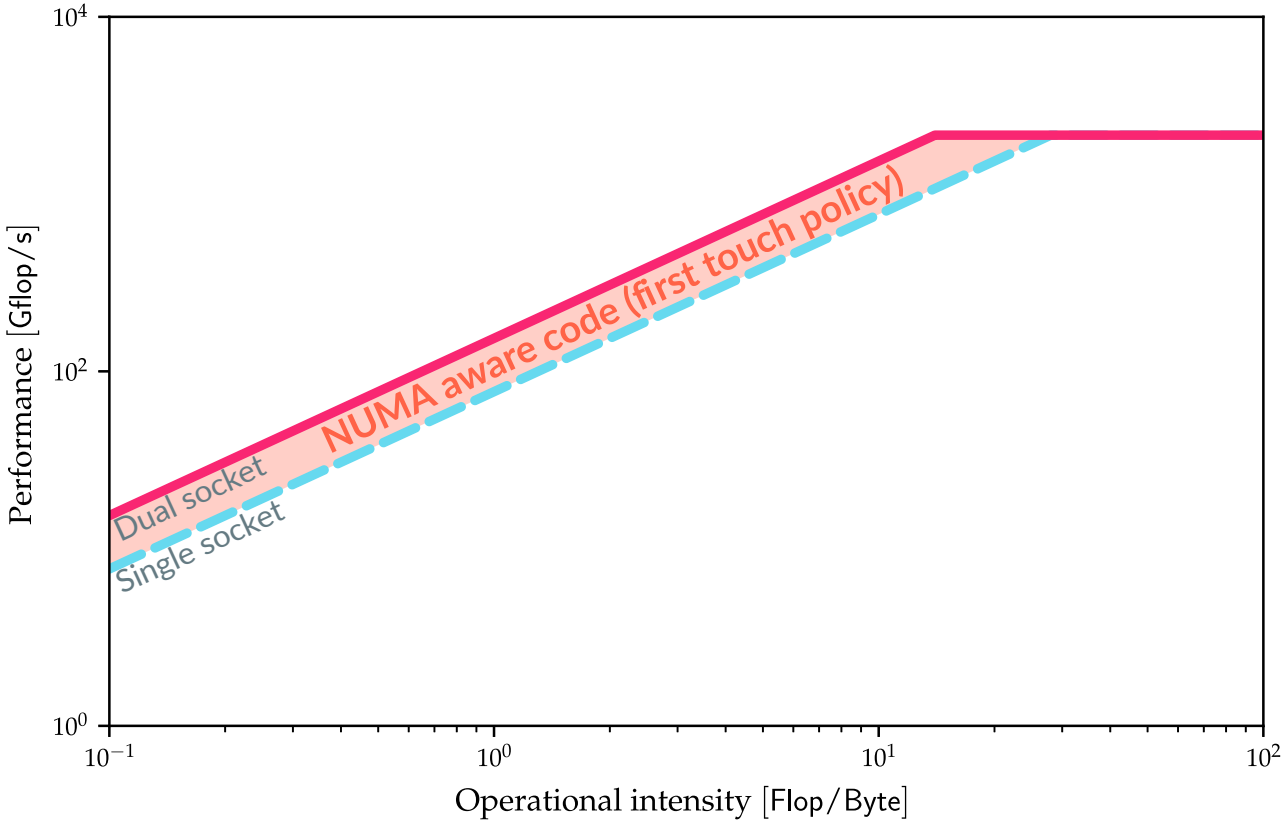
The ceiling for added data-level parallelism (DLP):



- The example roofline on the right is for the Intel Xeon E5-2683v4 CPU.
- Peak performance requires a significant mix of floating point instructions (mainly multiplication and addition).
- The best mix is achieved if multiplication and addition are *adjacent* operations, e.g., `a = a * b + c`
- Most recent CPUs support *fused multiply-add* instructions which allows for a multiplication and addition *in the same cycle*.

ROOFLINE CEILINGS FOR OPTIMIZATION GUIDANCE

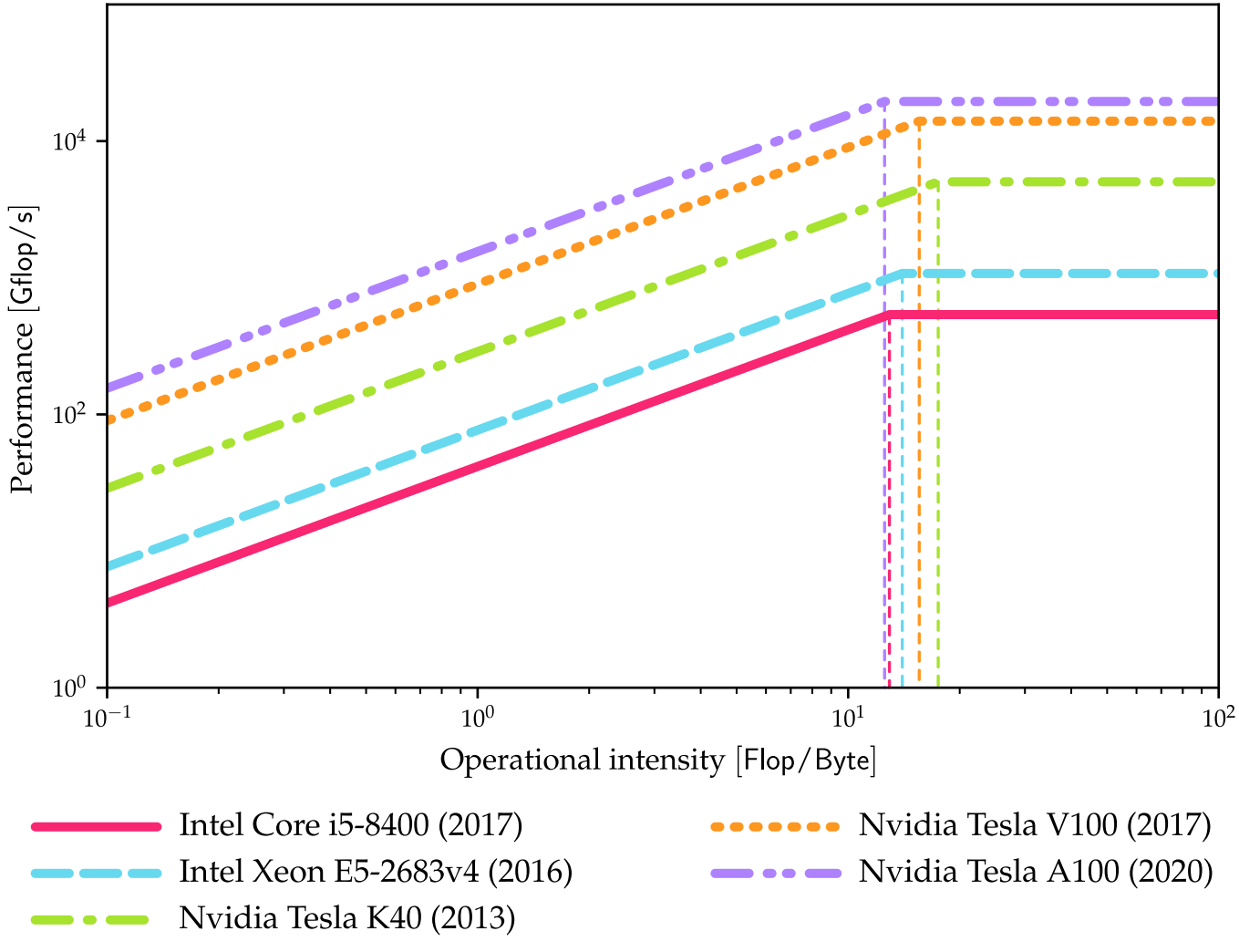
Performance also depends on memory affinity (NUMA first-touch policy):



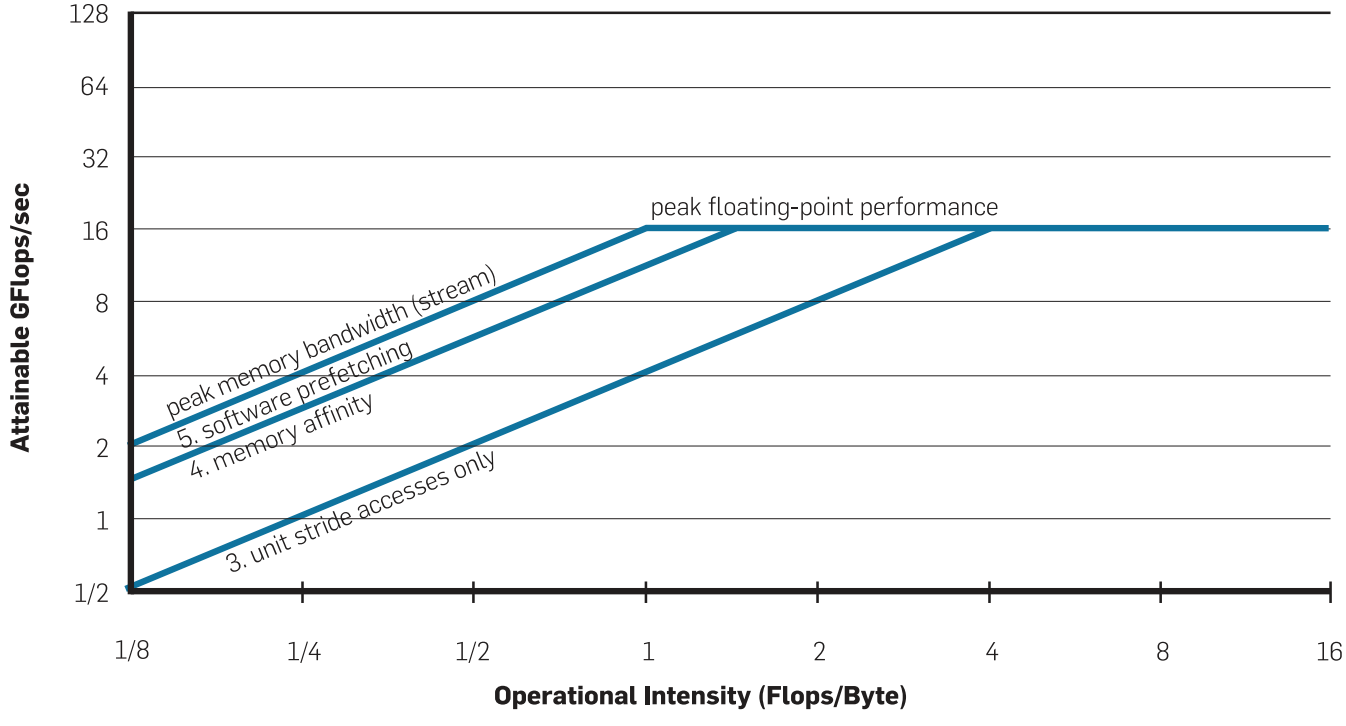
- The example roofline on the right is for the Intel Xeon E5-2683v4 CPU.
- Higher bandwidth moves bandwidth ceiling *in the vertical upwards direction*. ***Note:*** the *slope* remains one!
- More bandwidth helps to move the ridge point to the left.
- An architecture with the ridge point on the far left *is easier to program* than one with the ridge point on the far right, a characteristic for (older) GPUs.

ROOFLINES FOR DIFFERENT ARCHITECTURES

Compare the Intel Xeon E5-2683v4 CPU versus Nvidia Tesla GPUs:



THE BANDWIDTH CEILINGS



Williams et al., *Roofline: An insightful Visual Performance model for multicore Architectures* , ACM 2009.

- Unit-stride access is very important in all your applications → *exploit cache locality as best you can!*
- Multi-threaded execution on NUMA platforms: memory affinity is important! *NUMA first-touch policy*.
- Software prefetching ***is not*** a top-priority.

THE BANDWIDTH CEILINGS

A note on software prefetching:

- *Software prefetching* is a technique to load data into the cache before you reference it by issuing instructions that fetch cache lines in advance. *See the paper by D. Callahan et al. in the class repo.*
- Doing this wrong can hurt your performance as you may overwrite existing cache lines in the cache that you still need. Timing is important.
- Recent CPUs support *hardware prefetching* which works well almost always.
- ***The rule is*** that software prefetching is only considered (if at all) once you have fully exploited all other optimization strategies. *Suggested blog post to read*.
- GCC (and LLVM) compilers have builtin support for software prefetching. See the documentation here: https://gcc.gnu.org/projects/prefetch.html and `__builtin_prefetch` here: https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html.
- If you use this builtin in your project, *you must show the improvement by visualizing the two kernels in a roofline plot. Do not use it blindly and think it will be better!*

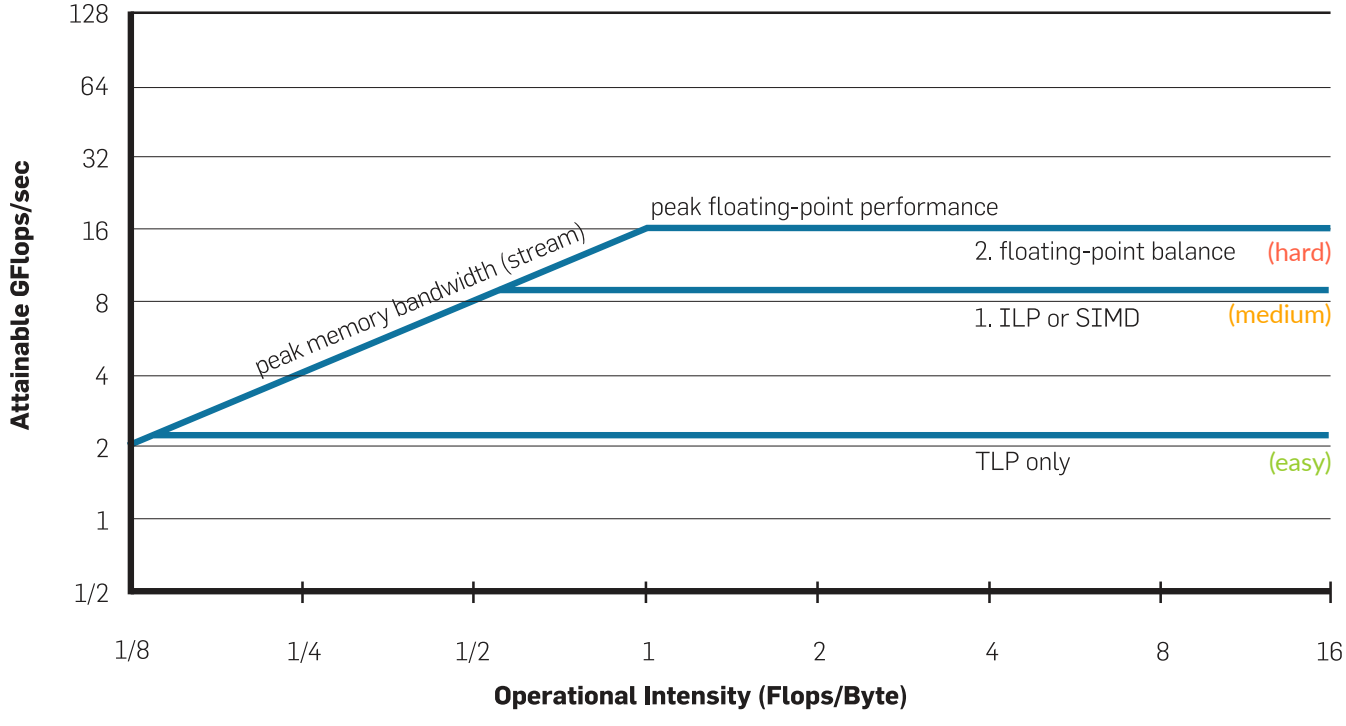
THE BANDWIDTH CEILINGS

A note on software prefetching: previous code for loop-unrolling

```c
int main(void)
{
    double A[1024];
    double B[1024];
    double C[1024];
    // manual 4-fold loop unroll
    for (int i = 0; i < 1024; i += 4) {
        __builtin_prefetch(&A[i + 4], 1, 1);
        __builtin_prefetch(&B[i + 4], 0, 1);
        __builtin_prefetch(&C[i + 4], 0, 1);
        A[i + 0] = A[i + 0] * B[i + 0] + C[i + 0];
        A[i + 1] = A[i + 1] * B[i + 1] + C[i + 1];
        A[i + 2] = A[i + 2] * B[i + 2] + C[i + 2];
        A[i + 3] = A[i + 3] * B[i + 3] + C[i + 3];
    }

    return 0;
}
```

- The generated code with and without the prefetch calls *performs the same in this case*. → the `__builtin_prefetch` calls in this example do not improve performance. *You could hurt performance by overuse of `__builtin_prefetch` calls*. Do not just use them blindly.
- The generated code *with* the prefetch calls is *larger*.

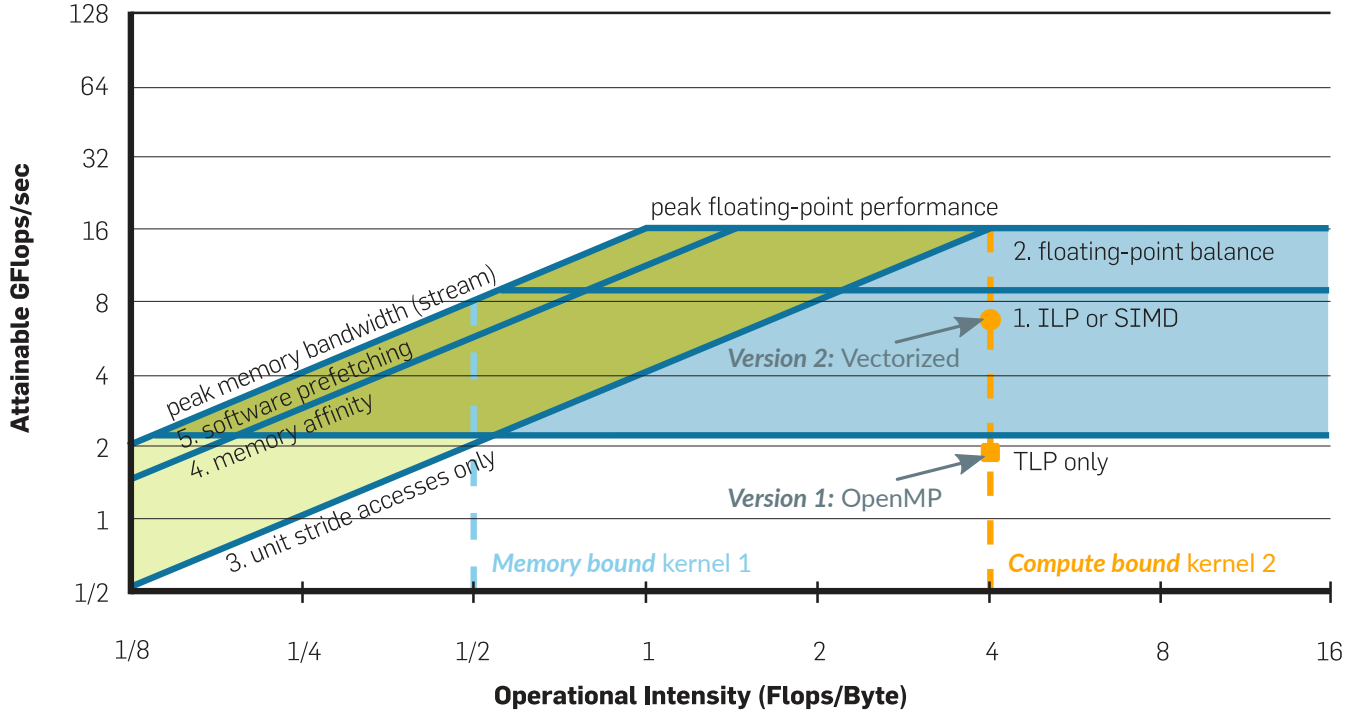
THE PERFORMANCE CEILINGS



Williams et al., *Roofline: An insightful Visual Performance model for multicore Architectures* , ACM 2009.

- The roofline *is an upper bound* to performance.
- Optimization steps should break through the ceiling of the current version of the compute kernel.
- What software optimizations should you perform, and in what order?

WHAT OPTIMIZATIONS AND IN WHAT ORDER?



Williams et al., *Roofline: An insightful Visual Performance model for multicore Architectures* , ACM 2009.

Ranking of ceilings:

- Ceilings are ranked *bottom to top*.
- Ceilings at the bottom are little effort and the compiler can often accomplish the optimization.
- Ceilings at the top are high effort ***or*** inherently lacking (algorithm dependent).

Optimizing a kernel:

- The optimization path for a given compute kernel is along the vertical line defined by the *operational intensity*.
- The optimization is incremental from bottom to top ceilings.

WHAT OPTIMIZATIONS AND IN WHAT ORDER?

Kernel optimizations from the paper:

Table 3: Kernel optimizations.[12, 28, 29]

Memory affinity. Reduce accesses to DRAM memory attached to the other socket.
Long unit-stride accesses. Change loop structures to generate long unit-stride accesses to engage the prefetchers; also reduces TLB misses.
Software prefetching. Software and hardware prefetching both used to get the most from memory systems.
Reduce conflict misses. Pad arrays to improve cache-hit rates.
Unroll and reorder loops. To expose sufficient parallelism and improve cache utilization, unroll and reorder loops to group statements with similar addresses; improves code quality, reduces register pressure, facilitates SIMD.
"SIMD-ize" code. The x86 compilers didn't generate good SSE code, so we made a code generator to produce SSE intrinsics.
Compress data structures (SpMV only). Since bandwidth limits performance, we used smaller data structures: 16b vs. 32b index and smaller representations of non-zero subblocks.[27]

Williams et al., *Roofline: An insightful Visual Performance model for multicore Architectures* , ACM 2009.

First optimization steps for memory bound kernels:

- Ignore ceilings and *maximize data reuse* by spatial and temporal locality → the upper ceiling are the *compulsory cache misses*.
- Minimize conflict misses by data reordering (and possibly padding).
- Minimize capacity misses by computation reordering.
- How to achieve reordering? *Examples:*
  - Loop tiling (or loop blocking)
  - Space filling curves
  - Block-structured memory layout

ROOFLINE EXERCISE

Addition of two vectors:

$$y = \alpha x + y \quad \alpha \in \mathbb{R} \text{ and } x, y \in \mathbb{R}^n$$

Compute kernel code:

```
1 void saxpy(const float a, const float *__restrict__ x, float *__restrict__ y, const int n)
2 {
3     for (int i = 0; i < n; ++i) {
4         y[i] = a * x[i] + y[i];
5     }
6 }
```

This kernel is called *SAXPY* which stands for *single-precision A times X plus Y*. It is a standard level 1 operation in BLAS (basic linear algebra subprograms).

Assume you start with a *cold cache*, what is the *asymptotic bound* of the operational intensity $I(n)$ for the saxpy kernel?

ROOFLINE EXERCISE

For each loop iteration i we perform the follwoing operations:

$$y_i = \alpha x_i + y_i \quad \forall i = 1, 2, \ldots, n$$

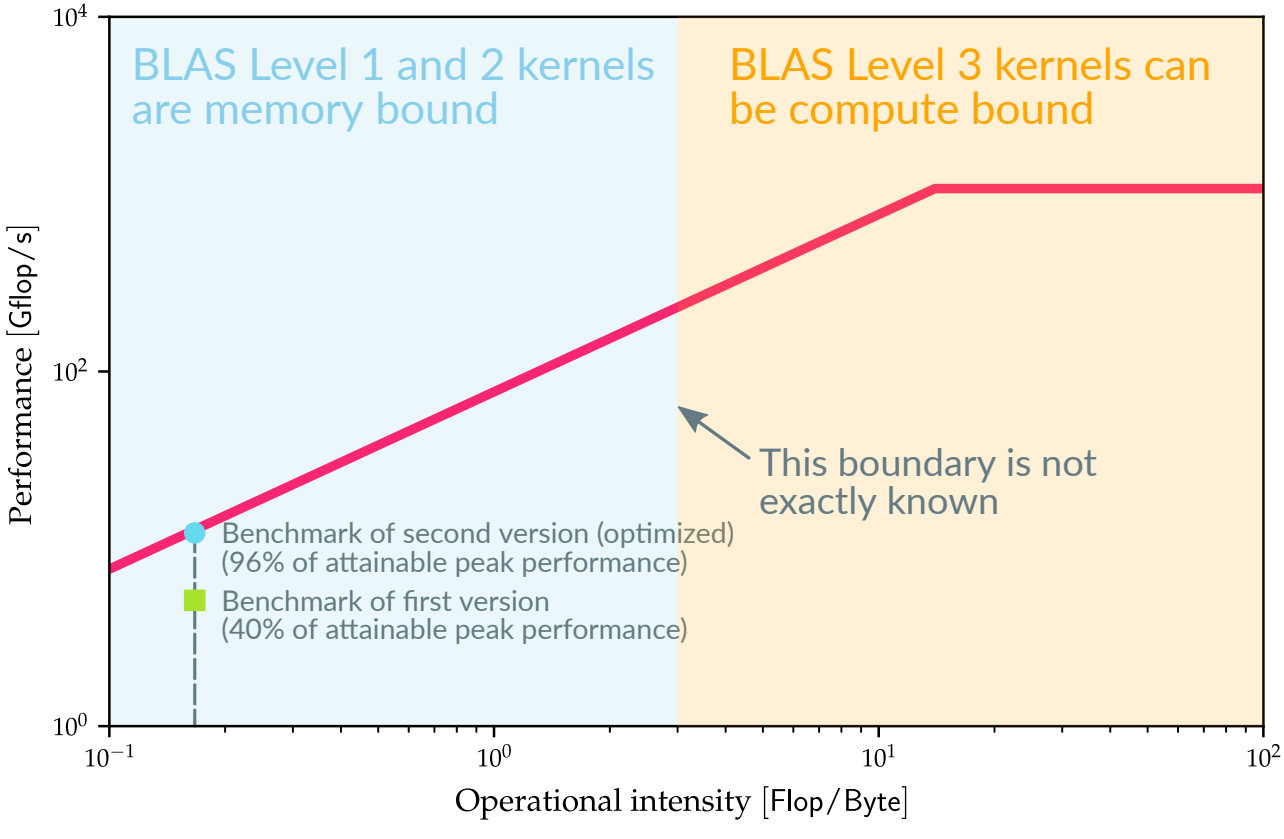
1. The number of flop we perform in each iteration is $f_i = 1_{\text{mul}} + 1_{\text{add}} = 2$ flop. The multiplication originates from the scalar α and the addition adds the scaled x_i and y_i together. The total number of flop for this kernel is $F = nf_i$.
2. The number of memory accesses is determined by the *compulsory misses* for this simple kernel. In each iteration we must *read* x_i and y_i and after the computation we must *write* the updated y_i. These are a total of $m_i = 2_{\text{read}} + 1_{\text{write}} = 3$ memory operations per loop iteration. The total number of memory accesses for this kernel is $M = nm_i$. *The total number of bytes transferred depends on the data precision of the computational kernel!* The total number of bytes is Mp, where p is the precision. For *single precision* $p = 4$ byte and $p = 8$ byte for *double precision*.
3. The operational intensity for the single precision kernel is now computed by

$$I(n) = \frac{F}{Mp} = \frac{nf_i}{nm_ip} = \frac{f_i}{m_ip} = \frac{1}{6}\,\frac{\text{Flop}}{\text{Byte}}.$$

The *asymptotic bound* of the operational intensity for the saxpy kernel is $\mathcal{O}(1)$ (it is constant $I(n) = I$) no matter what the size n of the problem is. ***What does this mean?***

ROOFLINE EXERCISE

Optimization iterations using roofline analysis for the saxpy (BLAS level 1) kernel (assuming an Intel Xeon E5-2683v4 architecture):



Memory bound kernels:

- Vector-vector and Matrix-vector operators (BLAS level 1 and 2). *This includes the majority of machine learning algorithms.*
- Explicit Runge-Kutta time integrators required to solve systems of ordinary differential equations.
- Working with unstructured meshes.
- Low-order finite difference / finite volume / finite element methods.
- How can you improve the operational intensity in your algorithms for better hardware utilization?

Library for access to performance counters: https://icl.utk.edu/papi

RECAP

- OpenMP wait policies for threads can be controlled with the `OMP_WAIT_POLICY` environment variable.
- Reported percentages of utilization from system monitors are not reliable metrics.
- The performance limiters on your hardware are peak floating-point performance and peak memory bandwidth.
- Operational intensity is a metric that tells you how many operations you perform on every byte fetched from or to DRAM.
- The roofline model is a simple yet elegant approach to determine your compute kernel performance and allows you to decide on the next optimization steps.
- Many compute kernels in practice are *memory bound*.

Further reading:

- Section 6.10, 6.11 and 6.12 in Patterson and Hennsey, *"Computer Organization and Design"*, Morgan Kaufmann 2018 (RISC-V edition)
- Section 1.6.1.3 in Eijkhout, *Introduction to High Performance Scientific Computing*, free PDF
- Chapter 2, 3 and 8 in Hager and Wellein, *"Introduction to High Performance Computing for Scientists and Engineers"*, CRC Press, 2011
- D. Callahan, K. Kennedy and A. Porterfield, *"Software Prefetching"*, ACM, 1991