NHR FAU

FAU Friedrich-Alexander-Universität
Erlangen-Nürnberg

Performance Modeling

Analytic performance modeling and its use in scientific computing

Georg Hager, NHR@FAU

R.W. Hockney and I.J. Curington: $f_{1/2}$: A parameter to characterize memory and communication bottlenecks. Parallel Computing 10, 277-286 (1989). DOI: 10.1016/0167-8191(89)90100-2

W. Schönauer: Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers. Self-edition (2000)

S. Williams: Auto-tuning Performance on Multicore Computers. UCB Technical Report No. UCB/EECS-2008-164. PhD thesis (2008)

Motivation

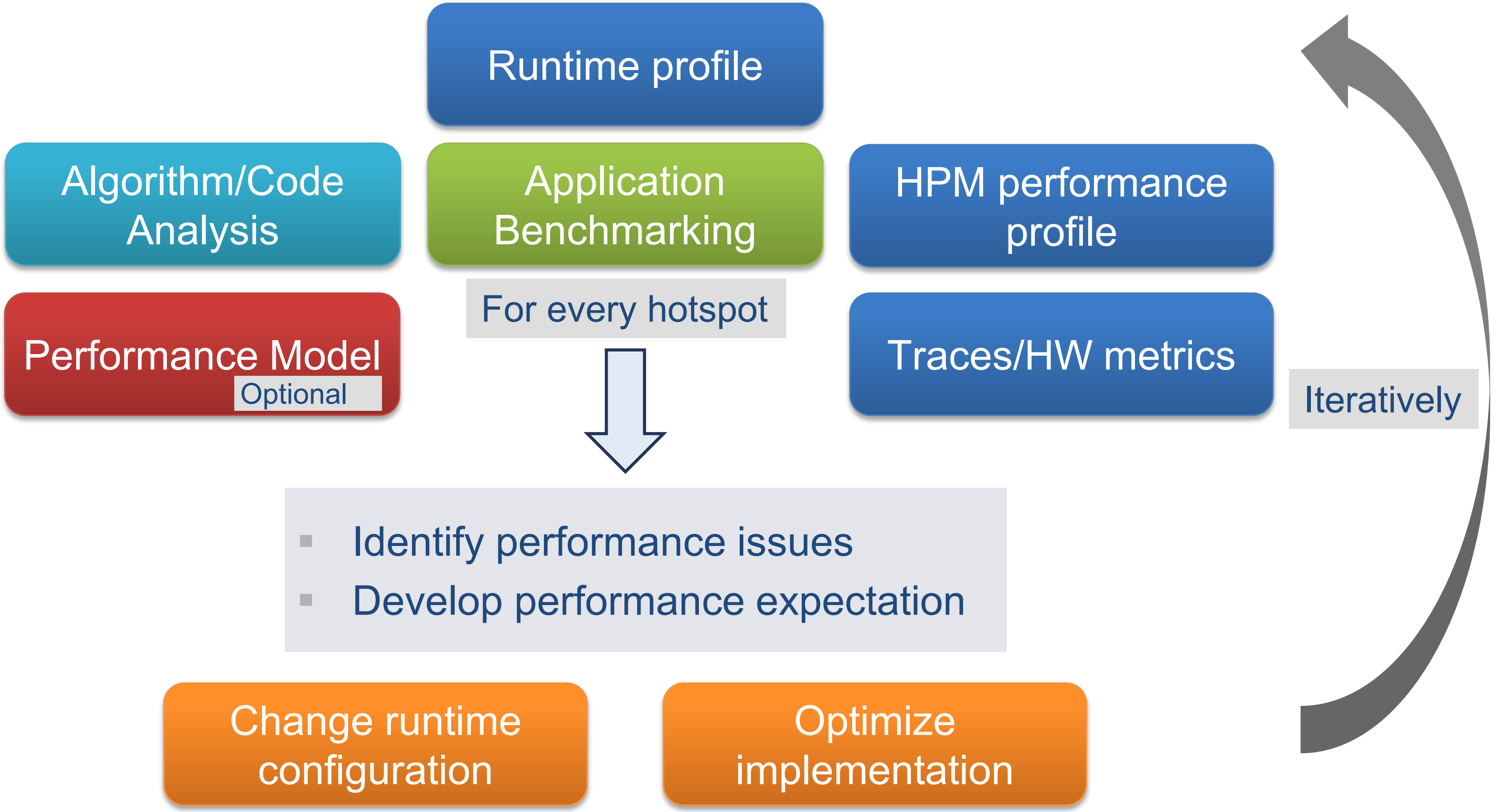
- Analytic performance modeling:

> An analytic white-box performance model is a simplified mathematical description of the hardware and its interaction with software. It is able to predict the runtime/performance of code from "first principles."

- Basic questions addressed by analytic performance models
  - What is the bottleneck?
  - What is the next bottleneck after optimization?
  - Impact of hardware features → co-design, architectural exploration
- What if the model fails?
  - We learn something
  - We may still be able to use the model in a less predictive way

Performance Engineering (PE)

Performance Engineering is a process



Runtime profile

Algorithm/Code Analysis

Application Benchmarking

HPM performance profile

Performance Model

Optional

For every hotspot

Traces/HW metrics

Iteratively

- Identify performance issues
- Develop performance expectation

Change runtime configuration

Optimize implementation

Getting a little more general

What data/knowledge can a model be based on?

- Only documented hardware properties + hypotheses
  - Purely analytic model

white box

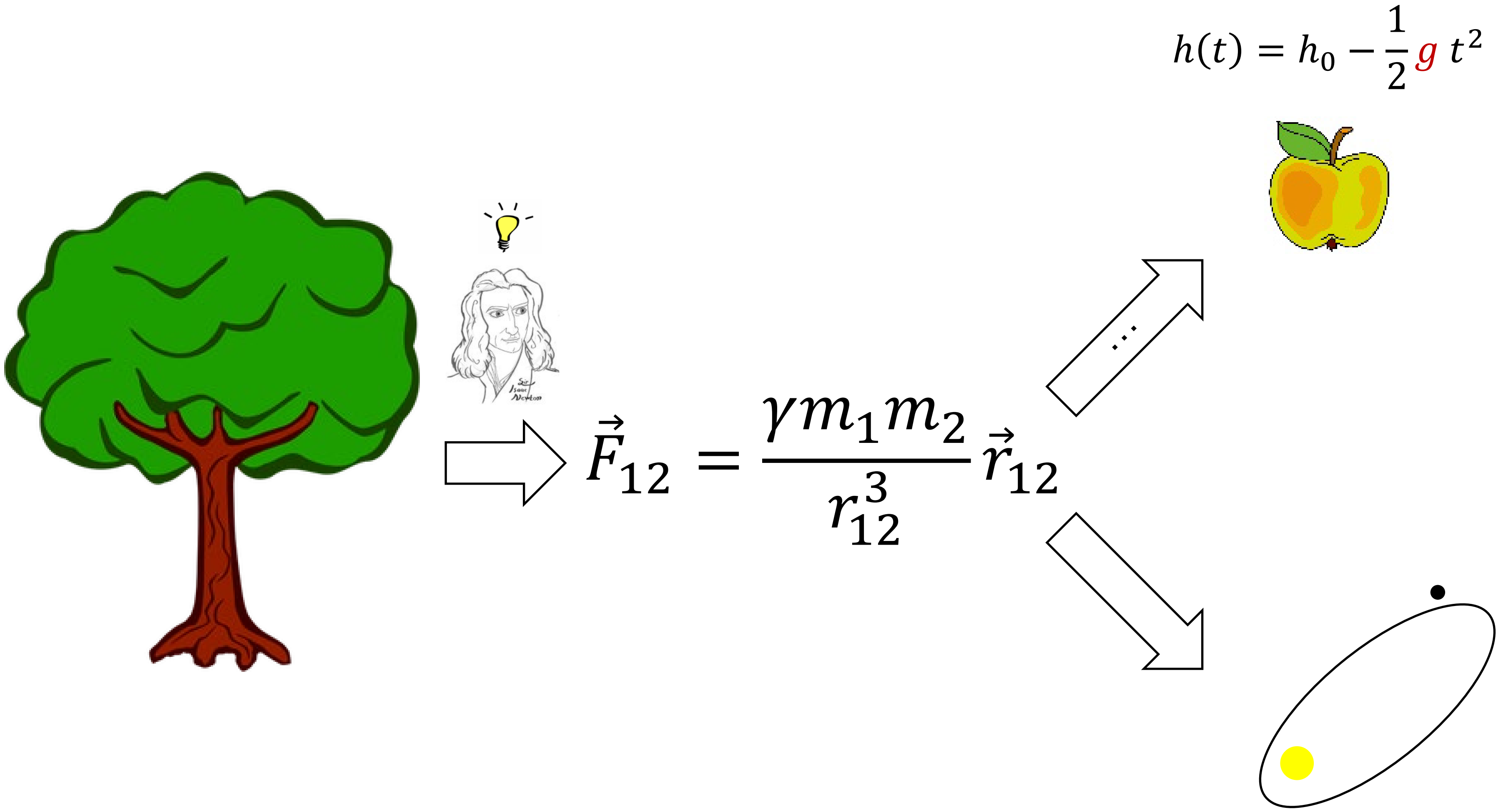
- Hardware properties + measurements + hypotheses
  - (Partly) phenomenological model

gray box

- Measured performance/speedup data + hypotheses
  - Curve-fitting analytic model

black box

An example from physics



$$h(t) = h_0 - \frac{1}{2}{\color{red}g}\,t^2$$

$$\vec{F}_{12} = \frac{\gamma m_1 m_2}{r_{12}^3}\,\vec{r}_{12}$$

Examples for white-/gray-box models in computing

program speedup

serial fraction

$$S(N) = \frac{1}{s + \frac{1-s}{N} + c(N)}$$

Amdahl's Law with communication

latency

msg. length

bandwidth

$$T_{PtP} = T_l + \frac{L}{B}$$

Hockney model for message transmission time

time for computation

time for data transfer

$$T_\mathrm{exec} = \max(T_\mathrm{calc}, T_\mathrm{data})$$

Roofline model for loop code execution time

non-overlapping execution

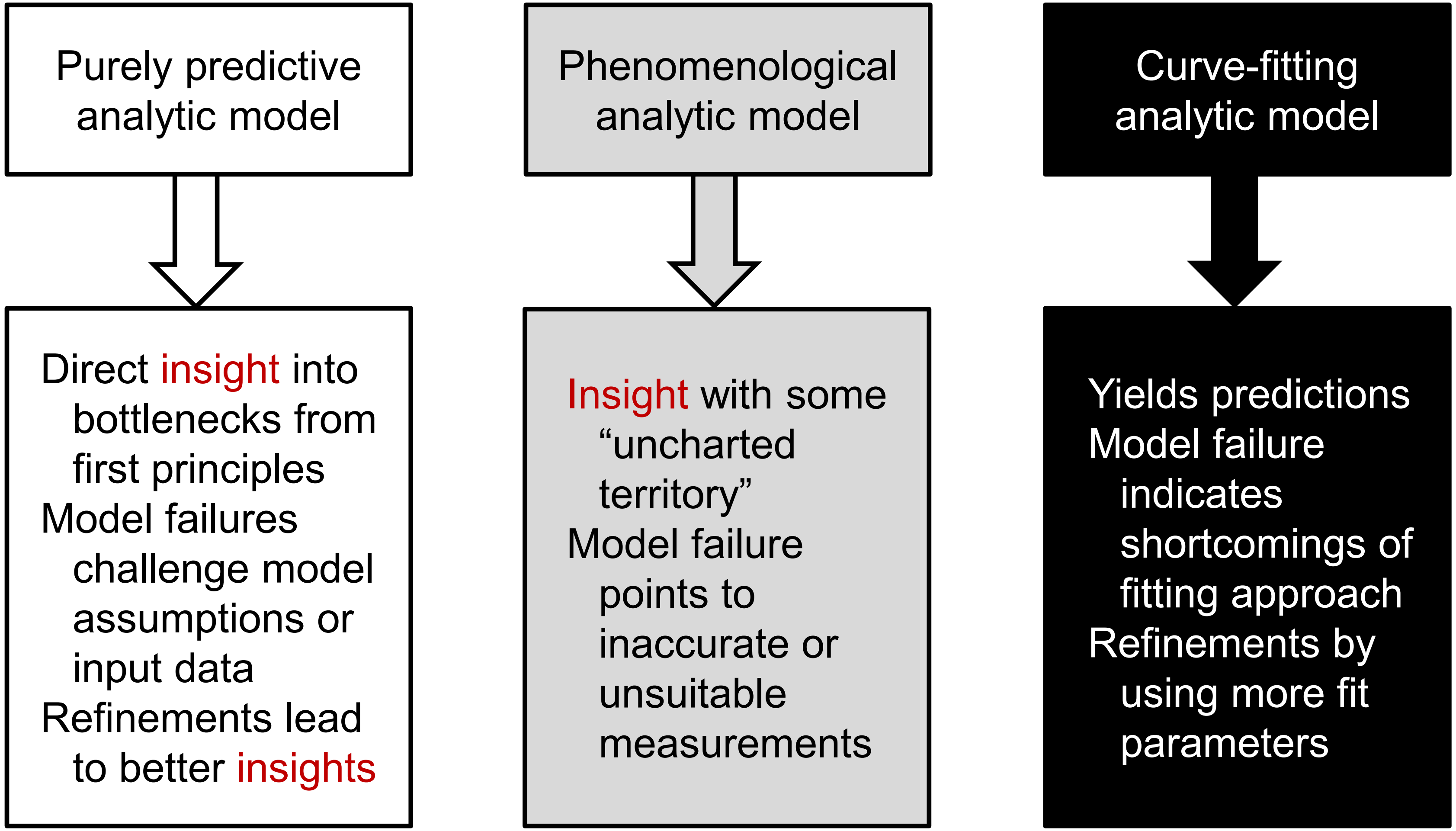
time for data transfer

$$T_\mathrm{exec} = f(T_{nOL}, T_\mathrm{data}, T_{OL})$$

overlapping execution

ECM model for loop code execution time

Models and insights



Purely predictive analytic model

- Direct insight into bottlenecks from first principles
- Model failures challenge model assumptions or input data
- Refinements lead to better insights

Phenomenological analytic model

- Insight with some “uncharted territory”
- Model failure points to inaccurate or unsuitable measurements

Curve-fitting analytic model

- Yields predictions
- Model failure indicates shortcomings of fitting approach
- Refinements by using more fit parameters

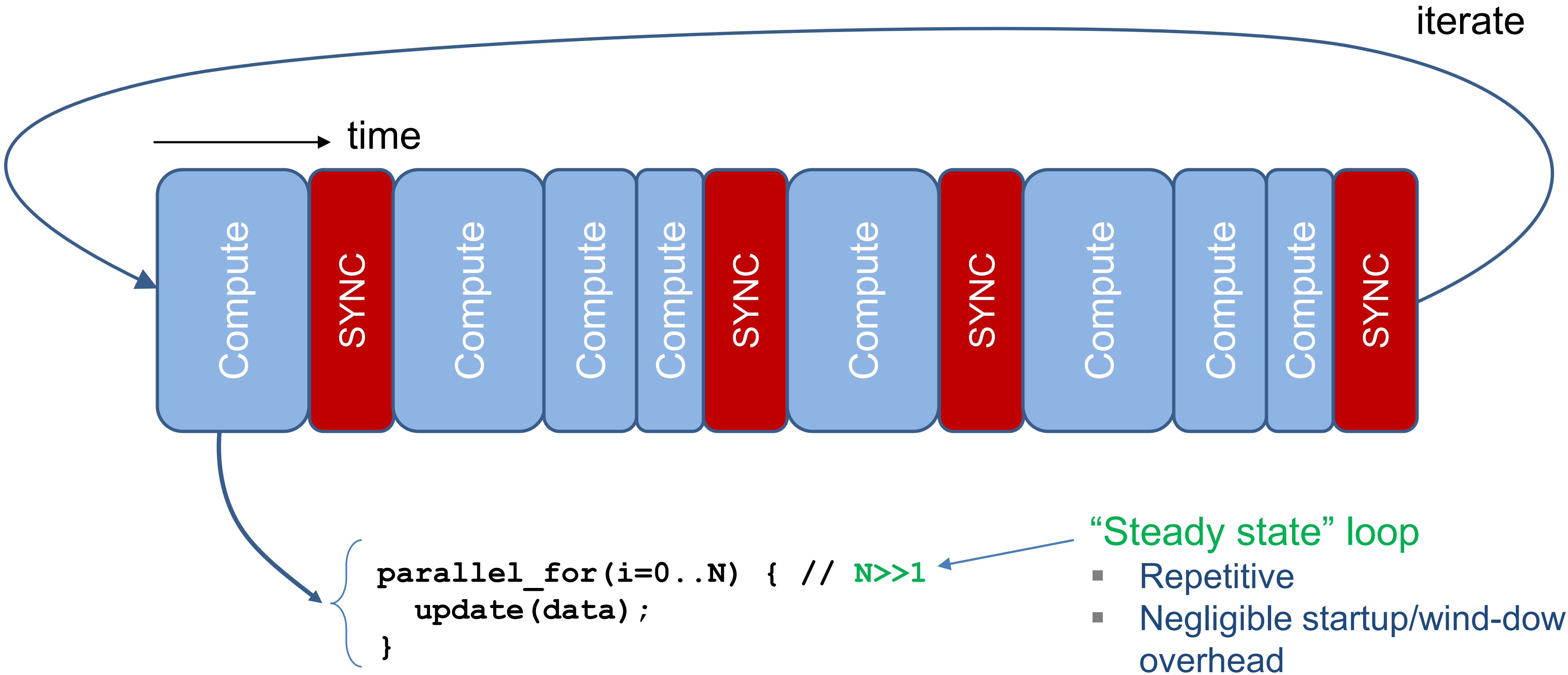
Resource-based performance models

The questions we ask

How much of `$RESOURCE` does `$STUFF` need on `$HARDWARE`, and why?

→ Analytic, resource-based, first-principles models

Structure of a solver code



```
parallel_for(i=0..N) { // N>>1
  update(data);
}
```

"Steady state" loop

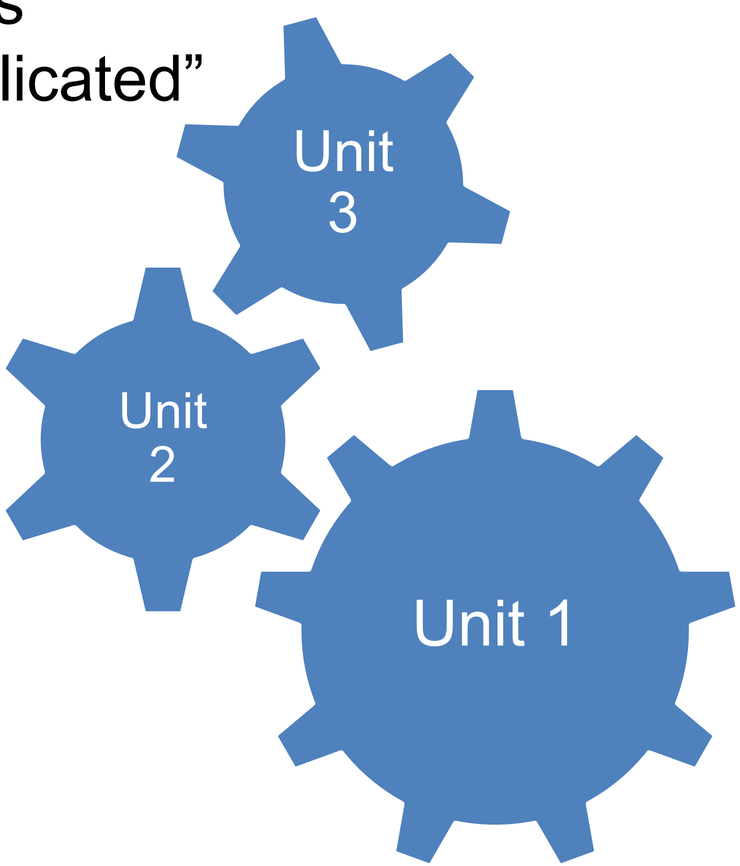
- Repetitive
- Negligible startup/wind-down overhead

Runtime model: $T = f(\texttt{\$STUFF}, \texttt{\$HARDWARE})$

Mechanistic vs. resource-based modeling

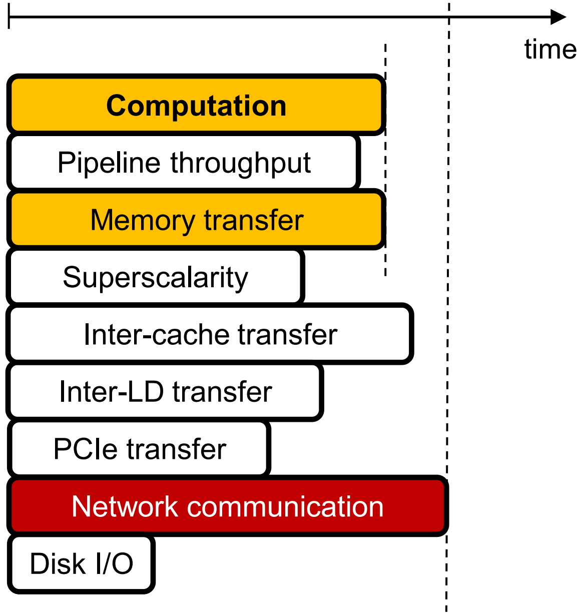
Mechanistic

- Cycle-by-cycle
- Latency
- Simulators
- “It’s complicated”



Resource based

- Resource utilization
- Data flow
- (Non-)overlapping components
- Simplify to make manageable
- If it doesn’t work, refine and iterate


time

Computation

Pipeline throughput

Memory transfer

Superscalarity

Inter-cache transfer

Inter-LD transfer

PCIe transfer

Network communication

Disk I/O

A general view on resource bottlenecks

What is the maximum performance when limited by a bottleneck?

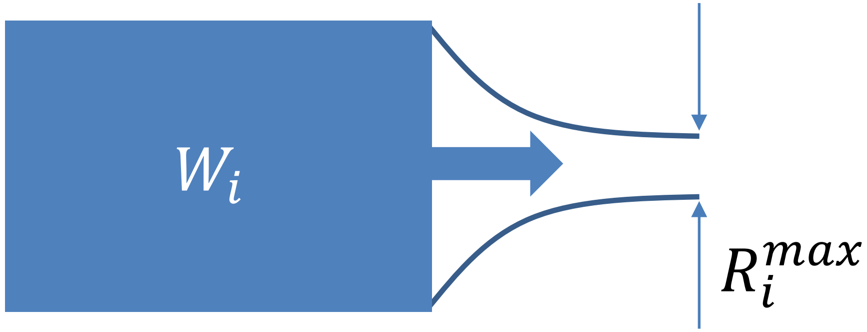
- Resource bottleneck i delivers resources at maximum rate R_i^{max}

Machine property

- W_i = needed amount of resources

Application property

- Minimum runtime: $T_i = \frac{W_i}{R_i^{max}} + \lambda_i$



Bottlenecks

Bottlenecked resources?

- Arithmetic (computational) throughput
- Overall instruction throughput
- Register-L1 data access
- Cache data transfer
- Memory data transfer
- Inter-ccNUMA domain data transfer
- Network data transfer
- …

Multiple bottlenecks

- Multiple bottlenecks → multiple min. runtimes:

$$T_{\mathrm{expect}} = f(T_1, \dots T_n)$$

🤔

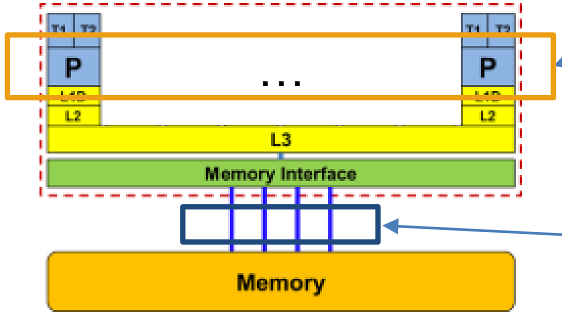
- Overall performance:

$$\boxed{P_{\mathrm{expect}} = \frac{W}{T_{\mathrm{expect}}}}$$

A bottleneck model of computing

Example: two bottlenecks

```
#pragma omp parallel for
for(i=0; i<10^7; ++i)
  a[i] = a[i] + s * c[i];
```



n-core CPU
(1 CMG A64FX 2 GHz)

$$R_{flops}^{max} = 768\,\frac{\text{Gflops}}{\text{s}}$$

$$R_{BW}^{max} = 210\,\frac{\text{Gbyte}}{\text{s}}$$

$$W_{flops} = 2 \times 10^7 \text{ flops}$$

$$W_{BW} = 3 \times 8 \times 10^7 \text{ bytes}$$

$$T_{flops} = \frac{2 \times 10^7 \text{ flops}}{768\,\frac{\text{Gflops}}{\text{s}}} = 26.0\ \mu s$$

$$T_{BW} = \frac{2.4 \times 10^8 \text{ bytes}}{210\,\frac{\text{Gbyte}}{\text{s}}} = 1.14 \text{ ms}$$

Bottleneck models

How do we reconcile the multiple bottlenecks?
I.e., what is the functional form of $f(T_1, \dots T_n)$?

→ pessimistic model (no overlap): $f(T_1, \dots T_n) = \sum_i T_i$
→ optimistic model (full overlap): $f(T_1, \dots T_n) = \max(T_1, \dots T_n)$

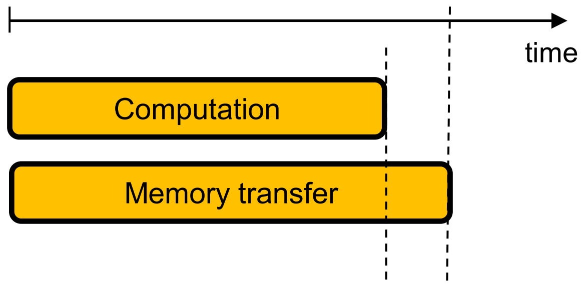
Roofline model
(Hockney et al., 1989
Williams et al., 2008)

Roofline for our example: $T_{\mathrm{min}} = \max\left(T_{flops}, T_{BW}\right) = 1.14\ \mathrm{ms}$

Maximum performance ("light speed"): $P_{\mathrm{expect}} = \frac{2\times 10^7}{1.14\times 10^{-3}} \frac{\mathrm{flops}}{\mathrm{s}} = 17.5\ \mathrm{Gflop/s}$

The Roofline Model

Deriving Roofline from the generic bottleneck model



- Two bottlenecks only
  - Peak computational rate R_{flops}^{max}
  - Peak memory bandwidth R_{BW}^{max}
- Full overlap assumption!
- Runtime prediction:

$$T_{pred}(T_{flops}, T_{BW}) = \max(T_{flops}, T_{BW}) = \max\left(\frac{W_{flops}}{R_{flops}^{max}}, \frac{W_{BW}}{R_{BW}^{max}}\right)$$

- Performance prediction in flop/s:

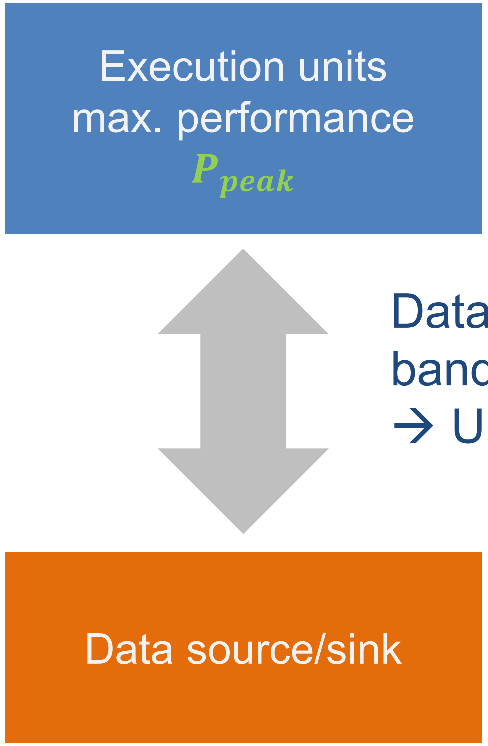
$$P_{pred} = \frac{W_{flops}}{T_{pred}} = \frac{W_{flops}}{\max\left(\frac{W_{flops}}{R_{flops}}, \frac{W_{BW}}{R_{BW}}\right)} = \min\left(R_{flops}, R_{BW} \times \frac{W_{flops}}{W_{BW}}\right)$$

The two-bottleneck model

$$P_{pred} = \min\left(\overbrace{\underbrace{R_{flops}}_{\text{Peak flop/s}}, \underbrace{R_{BW}}_{\text{Peak byte/s}}}^{\text{Machine model}} \times \overbrace{\underbrace{\frac{W_{flops}}{W_{BW}}}_{\text{Work-to-traffic ratio}}}^{\text{Application model}}\right)$$

The conventional form of the two-bottleneck model

Simplistic view of the hardware:



Execution units
max. performance
P_{peak}

Data source/sink

Data path,
bandwidth b_S
→ Unit: byte/s

Simplistic view of the software:

```
! may be multiple levels
do i = 1,<sufficient>
  <complicated stuff doing
    N flops causing
    V bytes of data transfer>
enddo
```

Computational intensity $I = \frac{N}{V}$
→ Unit: flop/byte

Naïve Roofline Model

How fast can tasks be processed? ***P*** [flop/s]

The bottleneck is either

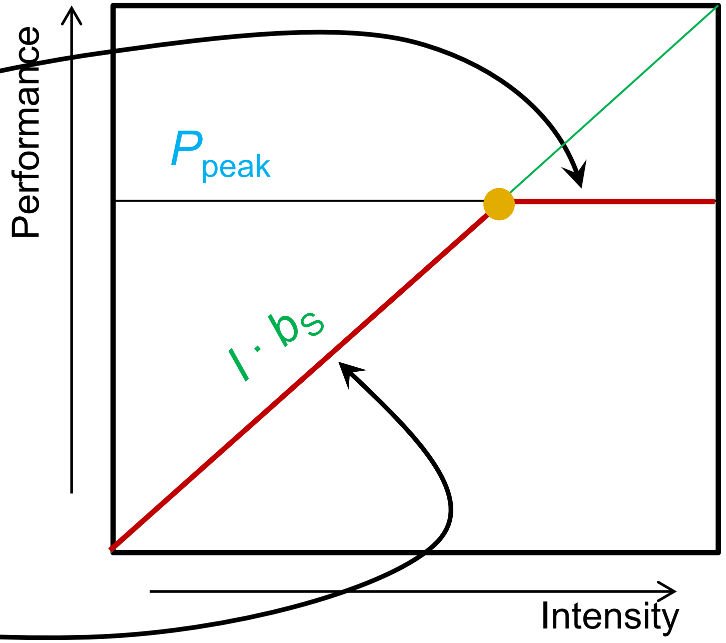
- The execution of work: P_{peak} [flop/s]
- The data path: $I \cdot b_S$ [flop/byte x byte/s]

$$P = \min(P_{\text{peak}}, I \cdot b_S)$$

This is the "Naïve Roofline Model"

- High intensity: P limited by execution
- Low intensity: P limited by data transfer
- "Knee" at $P_{peak} = I \cdot b_S$:
  Best use of resources

- Roofline is an "optimistic" model
  (think "light speed")



The Roofline Model in computing – Basics

Apply the naive Roofline model in practice

- Machine parameter #1: Peak performance: $P_{peak}\ \left[\frac{F}{s}\right]$
- Machine parameter #2: Memory bandwidth: $b_S\ \left[\frac{B}{s}\right]$
- Code characteristic: Computational intensity: $I\ \left[\frac{F}{B}\right]$

Machine model

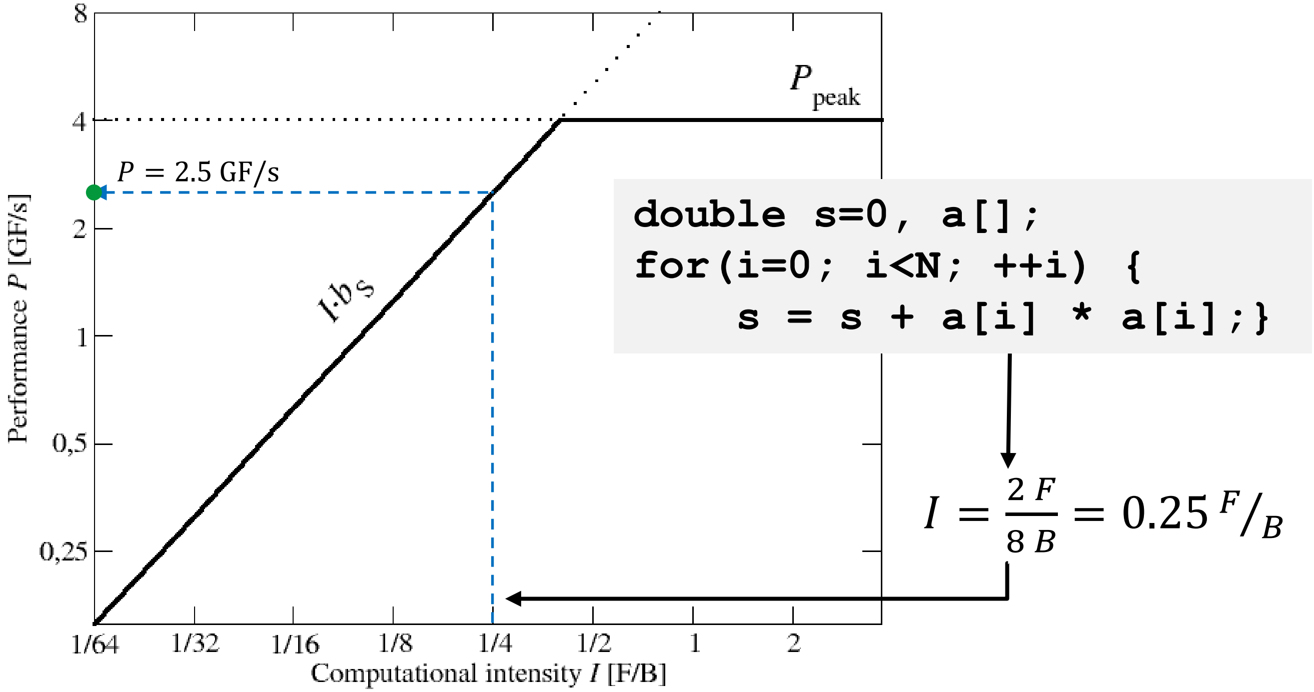
Application model

Machine properties:

$$\boldsymbol{P_{peak}} = 4\,\frac{\mathrm{GF}}{\mathrm{s}}$$

$$\boldsymbol{b_S} = 10\,\frac{\mathrm{GB}}{\mathrm{s}}$$

Application property: *I*

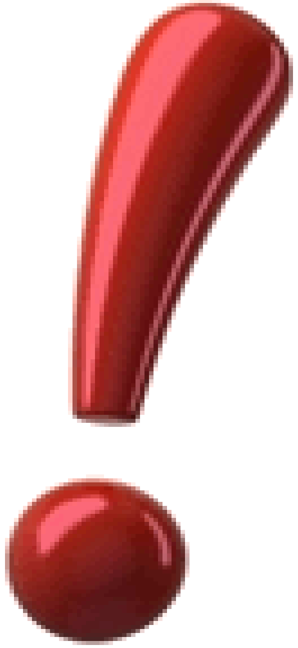


```
double s=0, a[];
for(i=0; i<N; ++i) {
    s = s + a[i] * a[i];}
```

$$I = \frac{2\ F}{8\ B} = 0.25\ {}^{F}/_{B}$$

Prerequisites for the Roofline Model

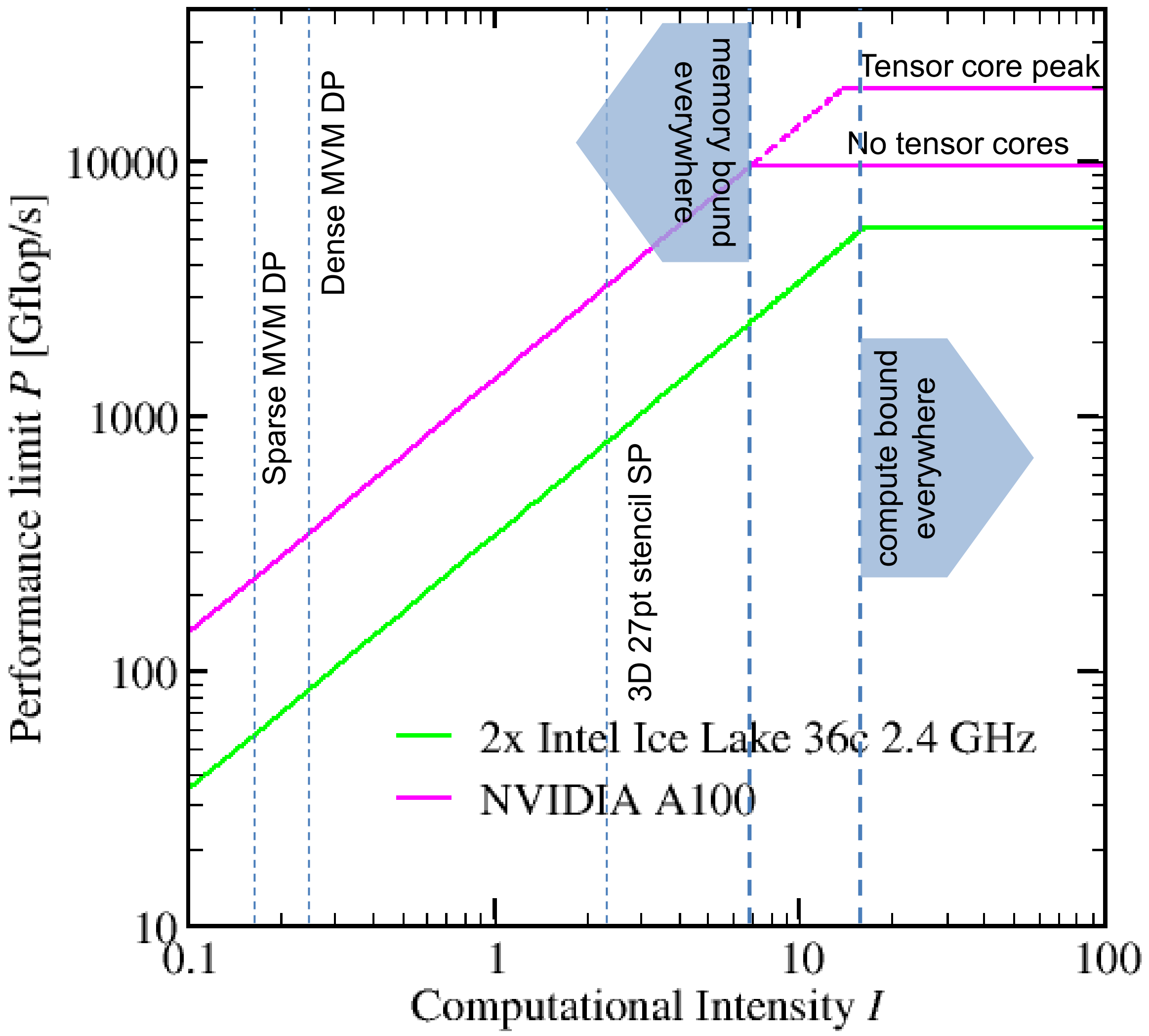
- Data transfer and core execution overlap perfectly!
  - Either the limit is core execution or it is data transfer
- Slowest limiting factor “wins”; all others are assumed to have no impact
  - If two bottlenecks are “close,” no interaction is assumed
- Data access latency is ignored, i.e. perfect streaming mode
  - Achievable bandwidth is the limit
- Chip must be able to saturate the bandwidth bottleneck(s)
  - Always model the full chip



Roofline for architecture and code comparison

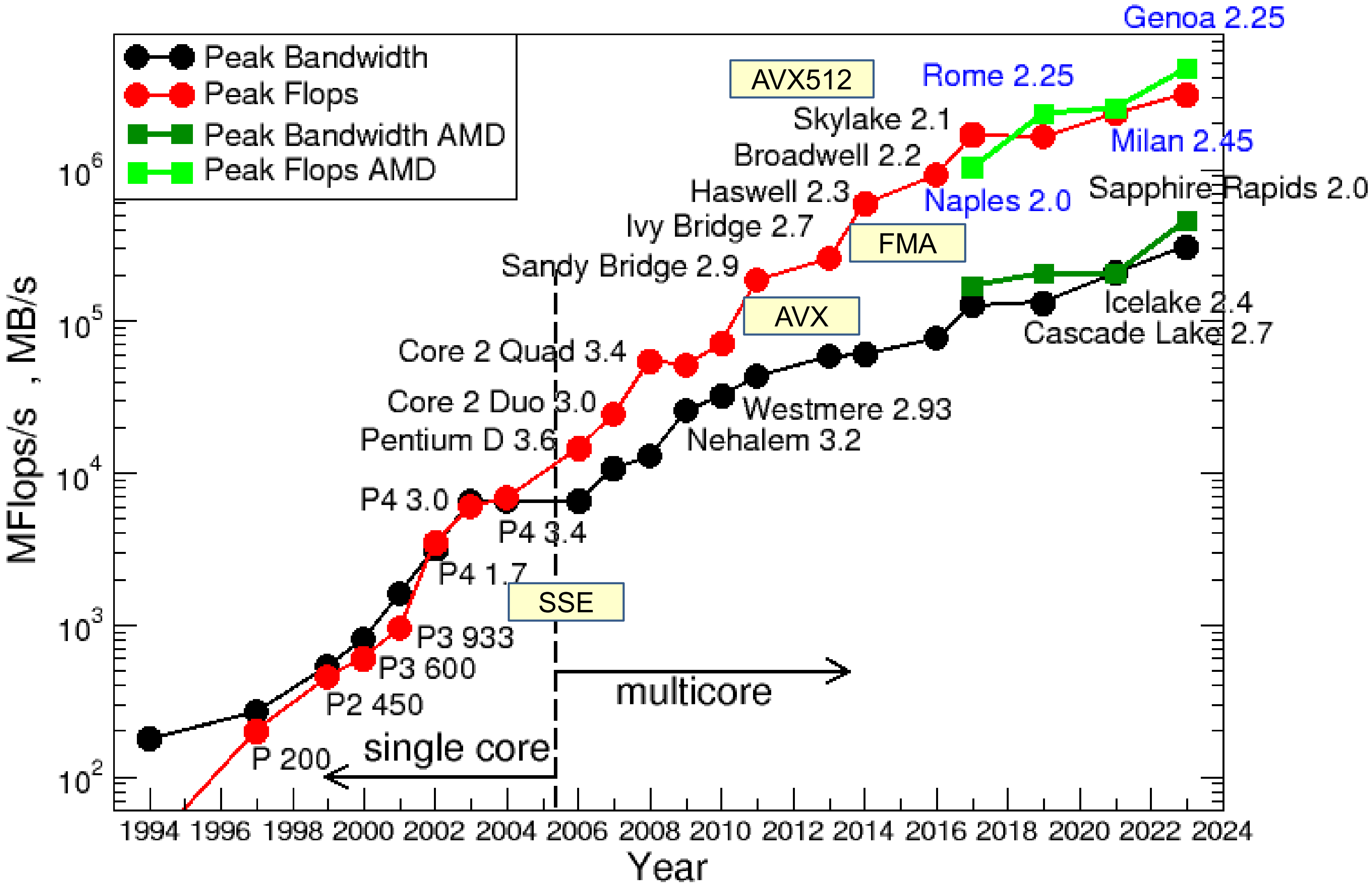
With Roofline, we can

- Compare capabilities of different machines
- Compare performance expectations for different loops

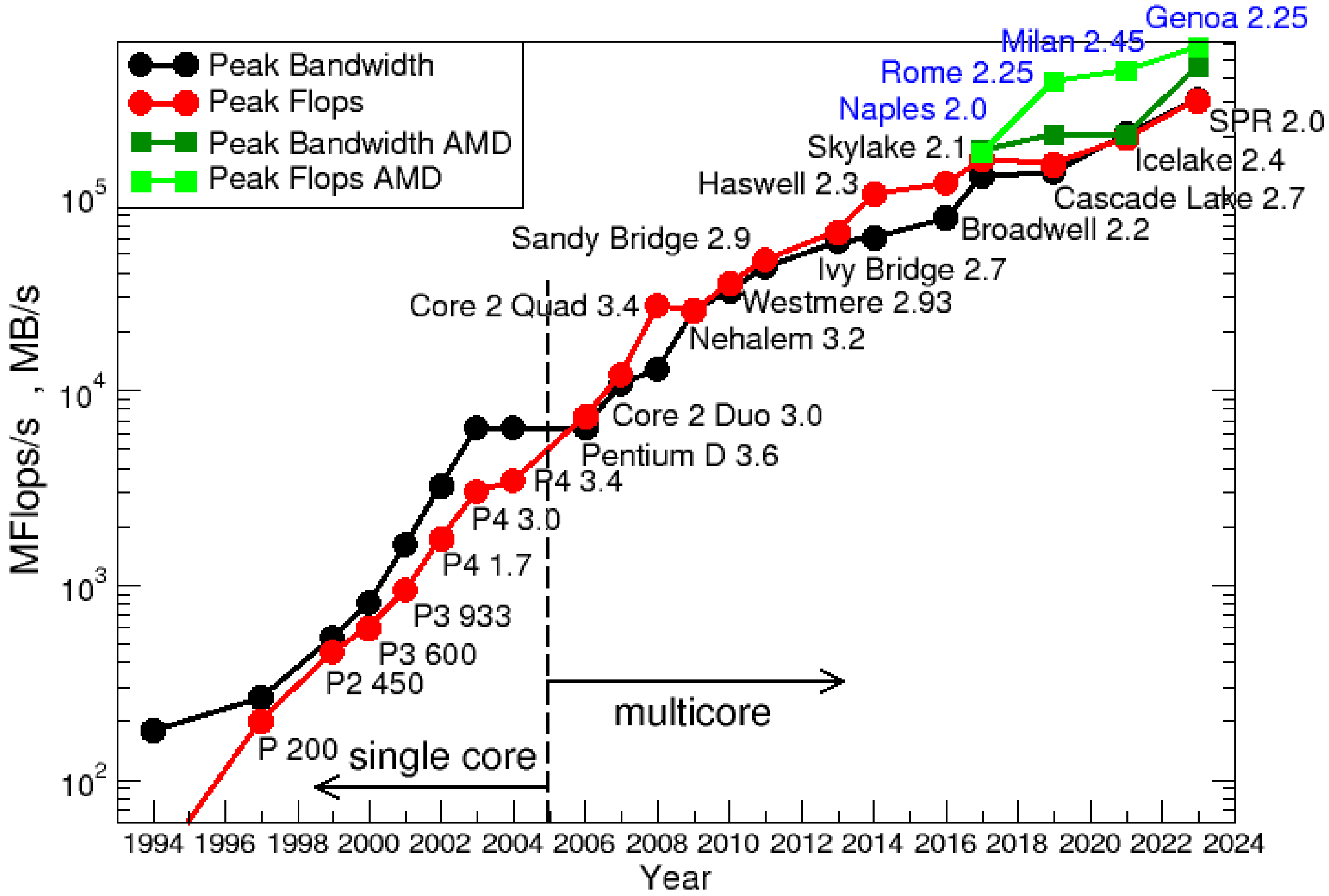


- Roofline always provides upper bound – but is it realistic?
  - Simple case: Loop kernel has loop-carried dependcncies → cannot achieve peak
  - Other bandwidth bottlenecks may apply

The “DRAM Gap” (Intel and AMD CPUs)



A different view of the DRAM gap: no SIMD, no FMA

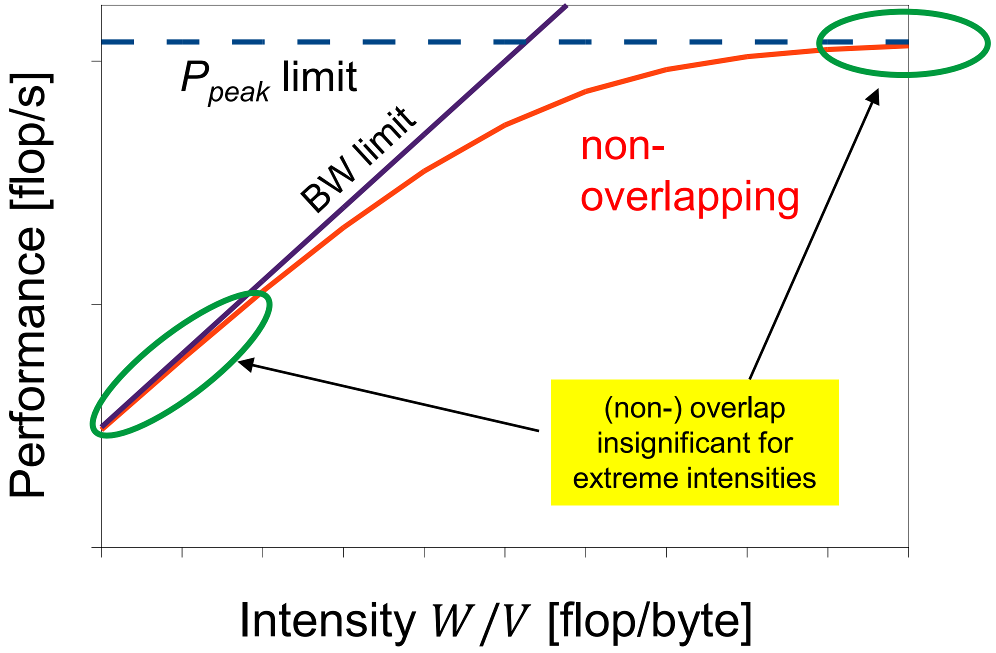


Interlude: a question

What would happen if we switched from the (optimistic) full overlap to the (pessimistic) non-overlap assumption in the two-bottleneck model?

$$T_{pred}\left(T_{flops},T_{BW}\right) = \max(T_{fl}$$

$$T_{pred}\left(T_{flops},T_{BW}\right) = T_{flops} +$$



(non-) overlap insignificant for extreme intensities

A refined Roofline Model

1. $P_{\max}$ = Applicable peak performance of a loop, assuming that data comes from the level 1 cache (this is not necessarily P_{peak})
   → e.g., $P_{\max}$ = 176 GFlop/s

2. b_S = Applicable (saturated) peak bandwidth of the slowest data path utilized
   → e.g., b_S = 56 GByte/s

3. I = Computational intensity (“work” per byte transferred) over the slowest data path utilized (code balance $B_C = I^{-1}$)
   → e.g., I = 0.167 Flop/Byte → B_C = 6 Byte/Flop

“Flop” is not the only useful unit of work!

Performance limit:

$$P = \min(P_{\max}, I \cdot b_S) = \min\left(P_{\max}, \frac{b_S}{B_C}\right)$$

[byte/s]

[byte/Flop]

A “simple” example: The sum reduction

Machine and code characteristics

- Clock speed: 2.2 GHz
- AVX-capable (256-bit SIMD)
- 1 ADD, 1 MUL instruction per cy
- 8 cores
- Single-precision peak:

$$P_{peak} = 8 \times 2 \times 8 \times 2.2\ \mathrm{Gflop/s}$$
$$= \mathbf{281.6\ Gflop/s}$$

- ADD pipeline latency: 3 cy
- Memory bandwidth $b_S = \mathbf{40\ Gbyte/s}$

```
float sum, a[N];
//...
#pragma omp parallel for \
        reduction(+:sum)
for (int i=0; i<N; i++){
    sum += a[i];
}
```

Arithmetic intensity:

$$I = \frac{1\ \mathrm{flop}}{4\ \mathrm{byte}} = 0.25\ \mathrm{F/B}$$

Code balance:

$$B_c = 4\ \mathrm{B/F}$$

Applicable peak performance

```
for (int i=0; i<N; i++){
    sum += a[i];
}
```

How fast can this loop possibly run with data in the L1 cache? (P_{max})

- Loop-carried dependency on summation variable
- Execution stalls at every ADD until previous ADD is complete

→No pipelining?

→No SIMD?

Applicable peak for the sum reduction (I)

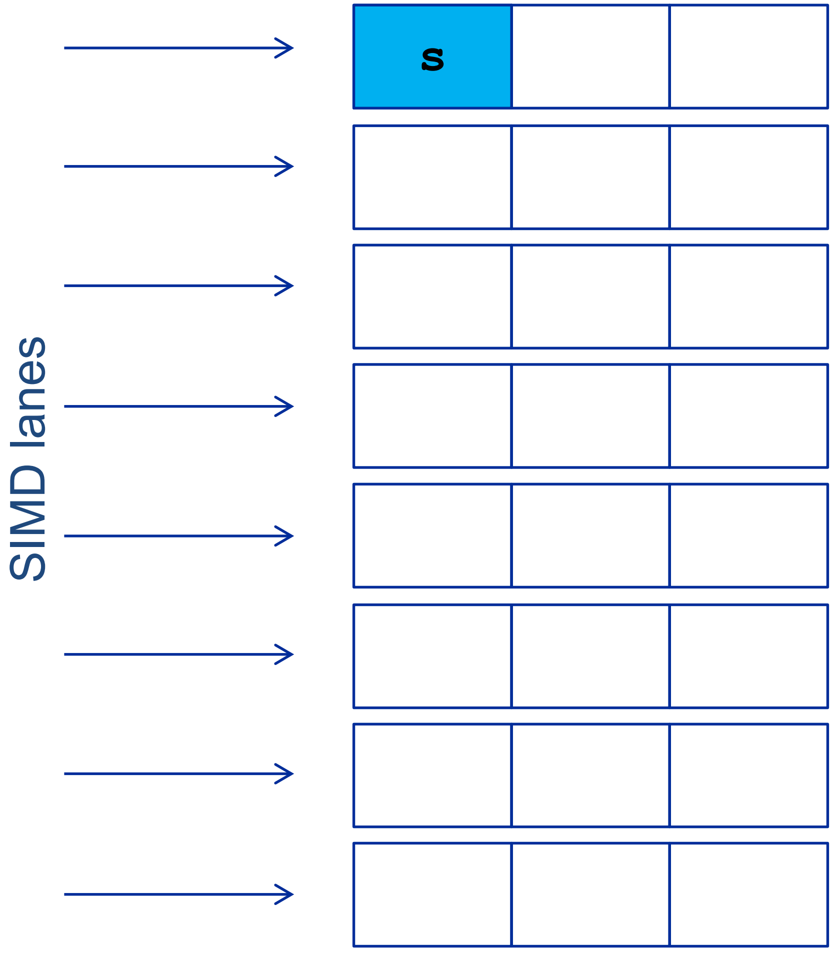
Plain scalar code, no SIMD

```
for (int i=0; i<N; i++){
    sum += a[i];
}
```

SIMD lane

```
LOAD r1.0 ← 0
i ← 1
loop:
  LOAD r2.0 ← a(i)
  ADD r1.0 ← r1.0 + r2.0
  ++i →? loop
result ← r1.0
```

ADD pipes utilization:



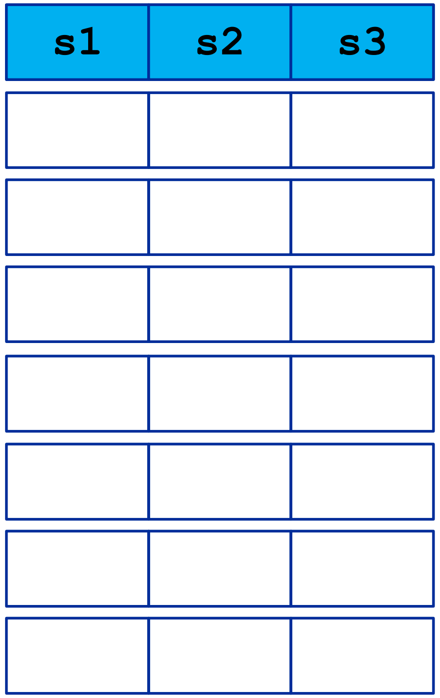
→ 1/24 of ADD peak

Applicable peak for the sum reduction (II)

Scalar code, 3-way “modulo variable expansion”

```
LOAD r1.0 ← 0
LOAD r2.0 ← 0
LOAD r3.0 ← 0
i ← 1

loop:
  LOAD r4.0 ← a(i)
  LOAD r5.0 ← a(i+1)
  LOAD r6.0 ← a(i+2)

  ADD r1.0 ← r1.0 + r4.0  # scalar ADD
  ADD r2.0 ← r2.0 + r5.0  # scalar ADD
  ADD r3.0 ← r3.0 + r6.0  # scalar ADD

  i+=3 →? loop
result ← r1.0+r2.0+r3.0
```

```
for (int i=0; i<N; i+=3){
    s1 += a[i+0];
    s2 += a[i+1];
    s3 += a[i+2];
}
sum = sum + s1+s2+s3;
```



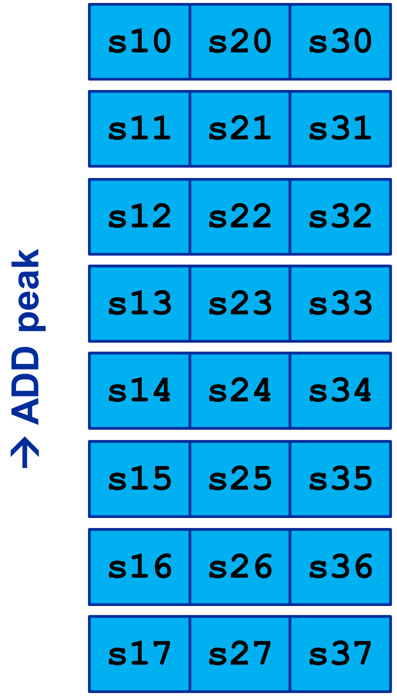
→ 1/8 of ADD peak

Applicable peak for the sum reduction (III)

SIMD vectorization (8-way MVE) x pipelining (3-way MVE)

```
for (int i=0; i<N; i+=24){
  s10 += a[i+0]; s20 += a[i+8];  s30 += a[i+16];
  s11 += a[i+1]; s21 += a[i+9];  s31 += a[i+17];
  s12 += a[i+2]; s22 += a[i+10]; s32 += a[i+18];
  s13 += a[i+3]; s23 += a[i+11]; s33 += a[i+19];
  s14 += a[i+4]; s24 += a[i+12]; s34 += a[i+20];
  s15 += a[i+5]; s25 += a[i+13]; s35 += a[i+21];
  s16 += a[i+6]; s26 += a[i+14]; s36 += a[i+22];
  s17 += a[i+7]; s27 += a[i+15]; s37 += a[i+23];
}
sum = sum + s10+s11+…+s37;
```

```
LOAD [r1.0,…,r1.7] ← [0,…,0]
LOAD [r2.0,…,r2.7] ← [0,…,0]
LOAD [r3.0,…,r3.7] ← [0,…,0]
i ← 1

loop:
  LOAD [r4.0,…,r4.7] ← [a(i),…,a(i+7)]      # SIMD LOAD
  LOAD [r5.0,…,r5.7] ← [a(i+8),…,a(i+15)]   # SIMD
  LOAD [r6.0,…,r6.7] ← [a(i+16),…,a(i+23)] # SIMD

  ADD r1 ← r1 + r4   # SIMD ADD
  ADD r2 ← r2 + r5   # SIMD ADD
  ADD r3 ← r3 + r6   # SIMD ADD

  i+=24 →? loop
result ← r1.0+r1.1+...+r3.6+r3.7
```



→ ADD peak

s10 s20 s30
s11 s21 s31
s12 s22 s32
s13 s23 s33
s14 s24 s34
s15 s25 s35
s16 s26 s36
s17 s27 s37

Sum reduction

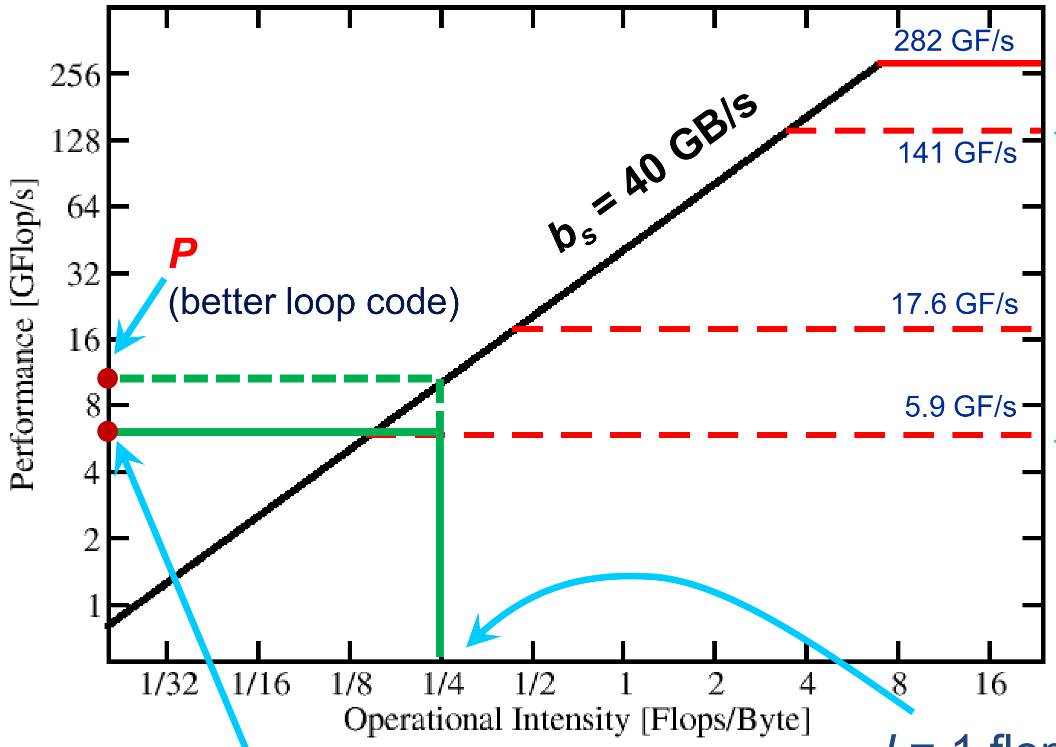
Questions

- When can this performance actually be achieved?
  - No data transfer bottlenecks
  - No other in-core bottlenecks
    - Need to execute (3 LOADs + 3 ADDs + 1 increment + 1 compare + 1 branch) in 3 cycles
- What does the compiler do?
  - If allowed and capable, the compiler will do this automatically
- Is the compiler allowed to do this at all?
  - Not according to language standards
  - High optimization levels can violate language standards
- What about the “accuracy” of the result?
  - Good question ;-)

Now let’s add the bandwidth ceiling

```
float sum, a[N];
//...
#pragma omp parallel for \
         reduction(+:sum)
for (int i=0; i<N; i++){
     sum += a[i];
}
```

$$P = \min(P_{\max}, I \cdot b_S)$$



P (better loop code)

P (worst loop code)

I = 1 flop / 4 byte (SP!)

Machine peak (ADD+MULT) Out of reach for this code

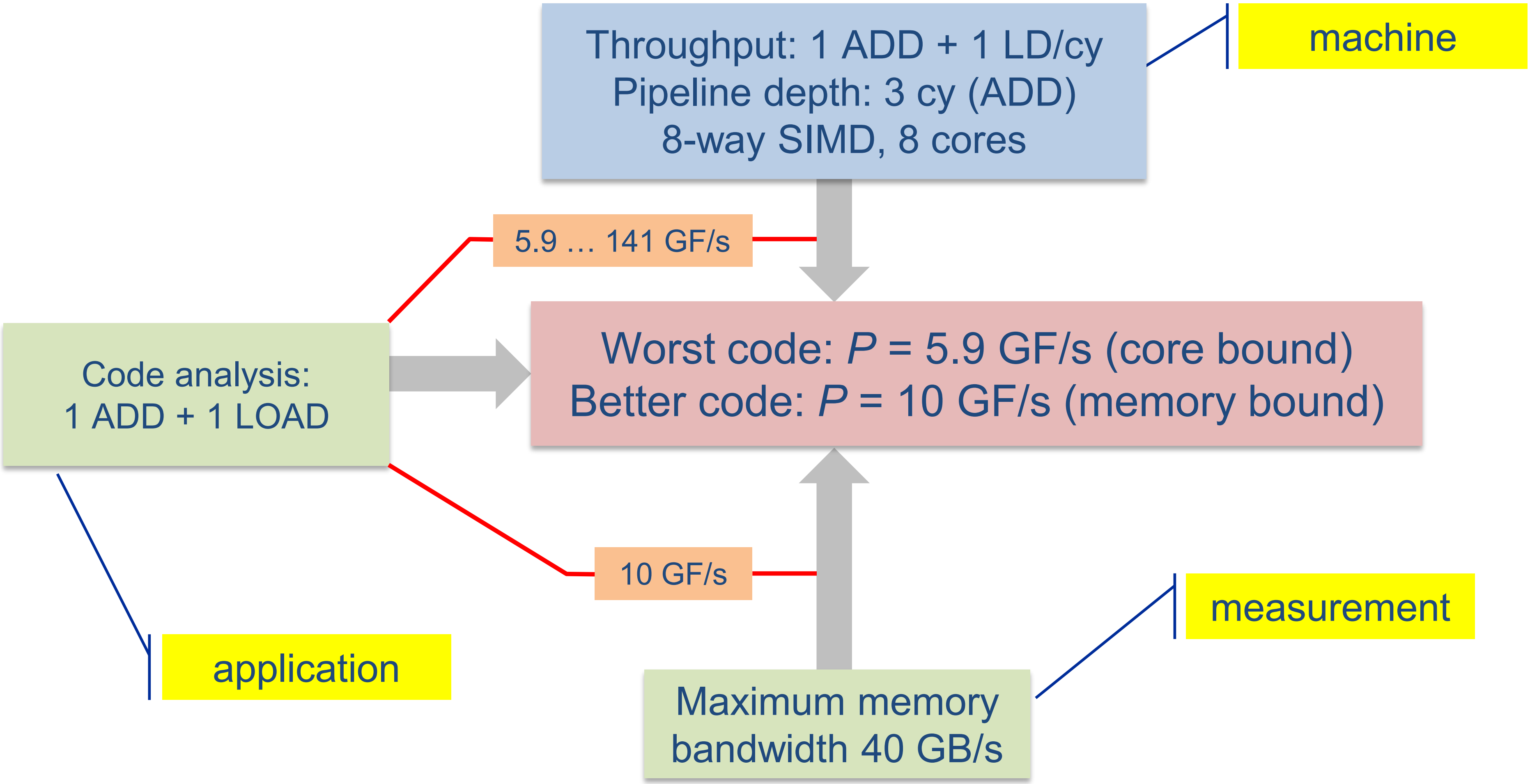
ADD peak (best possible code)

no SIMD

3-cycle latency per ADD if not unrolled

Input to the roofline model

… on the example of `do i=1,N; s=s+a(i); enddo`
in single precision



Throughput: 1 ADD + 1 LD/cy
Pipeline depth: 3 cy (ADD)
8-way SIMD, 8 cores

machine

5.9 … 141 GF/s

Code analysis:
1 ADD + 1 LOAD

Worst code: *P* = 5.9 GF/s (core bound)
Better code: *P* = 10 GF/s (memory bound)

10 GF/s

application

measurement

Maximum memory bandwidth 40 GB/s

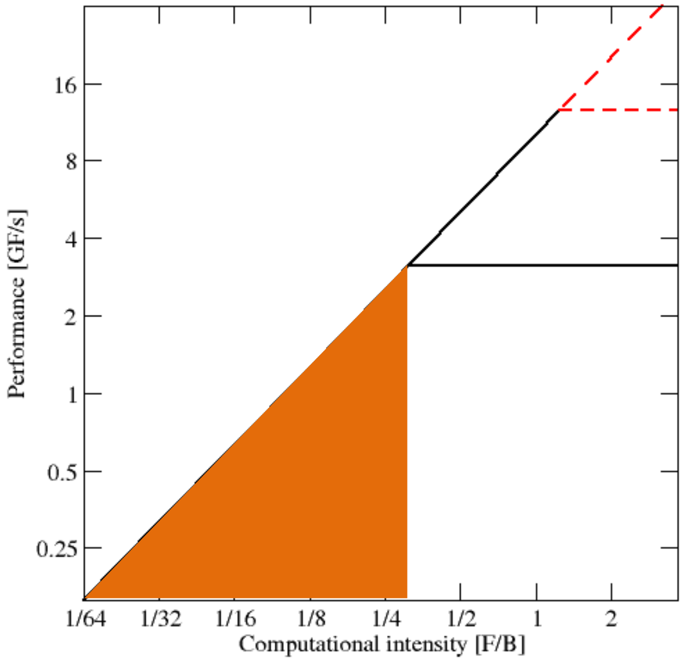
Factors to consider in the Roofline Model

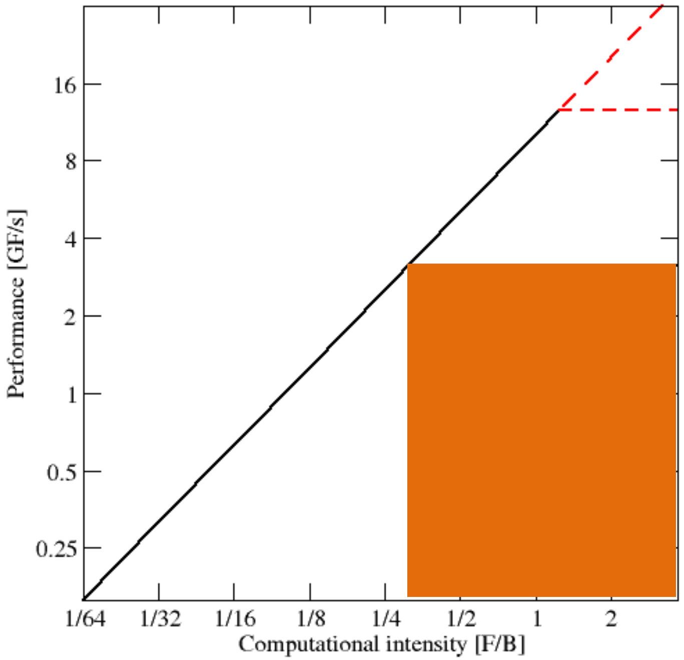
Bandwidth-bound

1. Accurate traffic calculation (write-allocate, strided access, cache reuse,…)
2. Practical ≠ theoretical BW limits
3. Saturation effects → consider full socket only

Core-bound

1. Multiple bottlenecks: LD/ST, arithmetic, pipelines, SIMD, execution ports
2. Limit is linear in # of cores

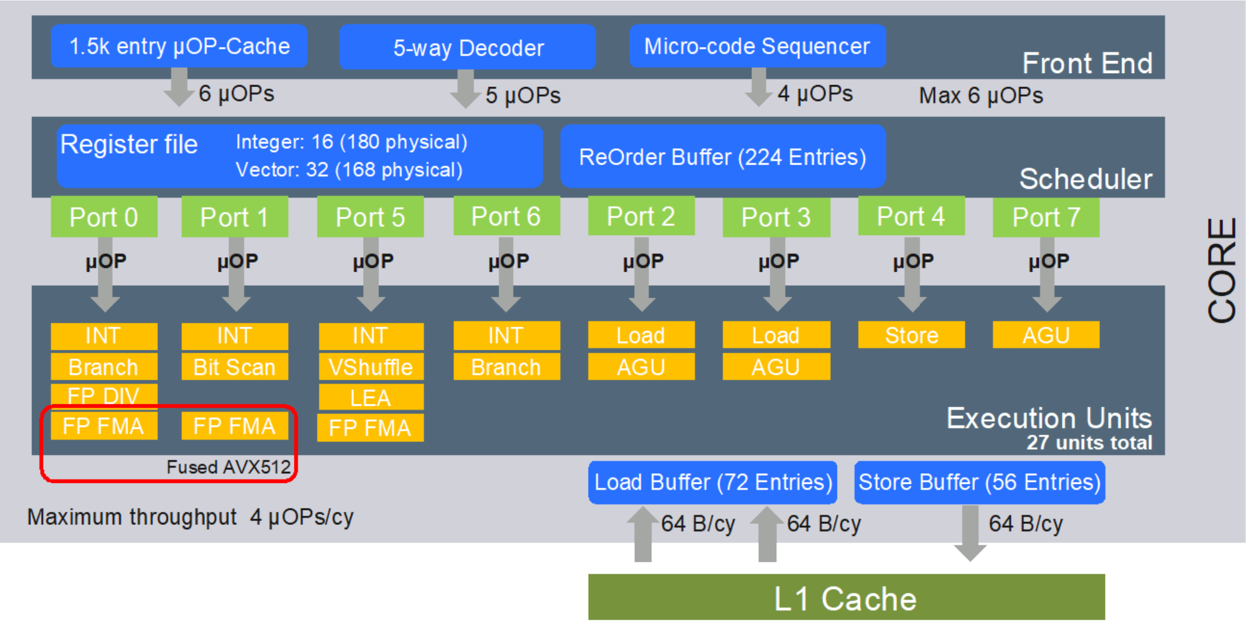




Complexities of in-core execution ($P_{\max}$)

Multiple bottlenecks:

- Decode/retirement throughput
- Port contention (direct or indirect)
- Arithmetic pipeline stalls (dependencies)
- Overall pipeline stalls (branching)
- L1 Dcache bandwidth (LD/ST throughput)
- Scalar vs. SIMD execution
- L1 Icache (LD/ST) bandwidth
- Alignment issues
- …



Skylake

Tool for $P_{\max}$ analysis: OSACA
http://tiny.cc/OSACA
DOI: 10.1109/PMBS49563.2019.00006
DOI: 10.1109/PMBS.2018.8641578

Hardware features of (some) Intel Xeon processors

Microarchitecture	Ivy Bridge EP	Broadwell EP	Cascade Lake SP	Ice Lake SP
Introduced	09/2013	03/2016	04/2019	06/2021
Cores	≤ 12	≤ 22	≤ 28	≤ 40
LD/ST throughput per cy:				
AVX(2), AVX512	1 LD + ½ ST	2 LD + 1 ST	2 LD + 1 ST	2 LD + 1 ST
SSE/scalar	2 LD || 1 LD & 1 ST			
ADD throughput	1 / cy	1 / cy	2 / cy	2 / cy
MUL throughput	1 / cy	2 / cy	2 / cy	2 / cy
FMA throughput	N/A	2 / cy	2 / cy	2 / cy
L1-L2 data bus	32 B/cy	64 B/cy	64 B/cy	64 B/cy
L2-L3 data bus	32 B/cy	32 B/cy	16+16 B/cy	16+16 B/cy
L1/L2 per core	32 KiB / 256 KiB	32 KiB / 256 KiB	32 KiB / 1 MiB	48 KiB / 1.25 MiB
LLC	2.5 MiB/core inclusive	2.5 MiB/core inclusive	1.375 MiB/core exclusive/victim	1.5 MiB/core exclusive/victim
Memory	4ch DDR3	4ch DDR3	6ch DDR4	8ch DDR4
Memory BW (meas.)	~ 48 GB/s	~ 62 GB/s	~ 115 GB/s	~ 160 GB/s

Source: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html

Code balance: more examples

```
double a[], b[];
for(i=0; i<N; ++i)
    a[i] = a[i] + b[i];
```

B_C = 24B / 1F = 24 B/F

I = 0.042 F/B

```
double a[], b[];
for(i=0; i<N; ++i)
    a[i] = a[i]+ s * b[i];
```

B_C = 24B / 2F = 12 B/F

I = 0.083 F/B

Scalar – can be kept in register

```
float s=0, a[];
for(i=0; i<N; ++i)
    s = s + a[i] * a[i];
```

B_C = 4B / 2F = 2 B/F

I = 0.5 F/B

Scalar – can be kept in register

```
float s=0, a[], b[];
for(i=0; i<N; ++i)
    s = s + a[i] * b[i];
```

B_C = 8B / 2F = 4 B/F

I = 0.25 F/B

Scalar – can be kept in register

Streaming, perfect spatial locality, no temporal locality → simple

```
float s=0, a[], b[];
for(i=0; i<N; ++i)
  for(j=0; j<N; ++j)
    b[i][j] = a[i][j]
            + a[i-1][j]
            + a[i+1][j];
```

B_C = 16B / 2F or

8B / 2F or even

20 B / 2F

???

Possible cache reuse → tricky!

And what about this?

```
float s=0, a[N], b[N];
int idx[N];
for(i=0; i<N; ++i)
    s = s + a[i]
          * b[idx[i]];
```

We'll get to it!

Is there anything to ease the construction of the model?

Code balance B_C

- Close inspection and hard thinking
- Simplifying assumptions
  - "What is the minimum possible amount of traffic?"
  - "What is the worst case?"
- Tools
  - Kerncraft
    https://github.com/RRZE-HPC/kerncraft

In-core $P_{\max}$

- Inspection of assembly code and manual modeling
- Simplifying assumptions
  - "What is the required minimum number of arithmetic/load/store instructions?"
  - $P_{\max} = P_{peak}$
- Tools
  - OSACA
    https://github.com/RRZE-HPC/OSACA

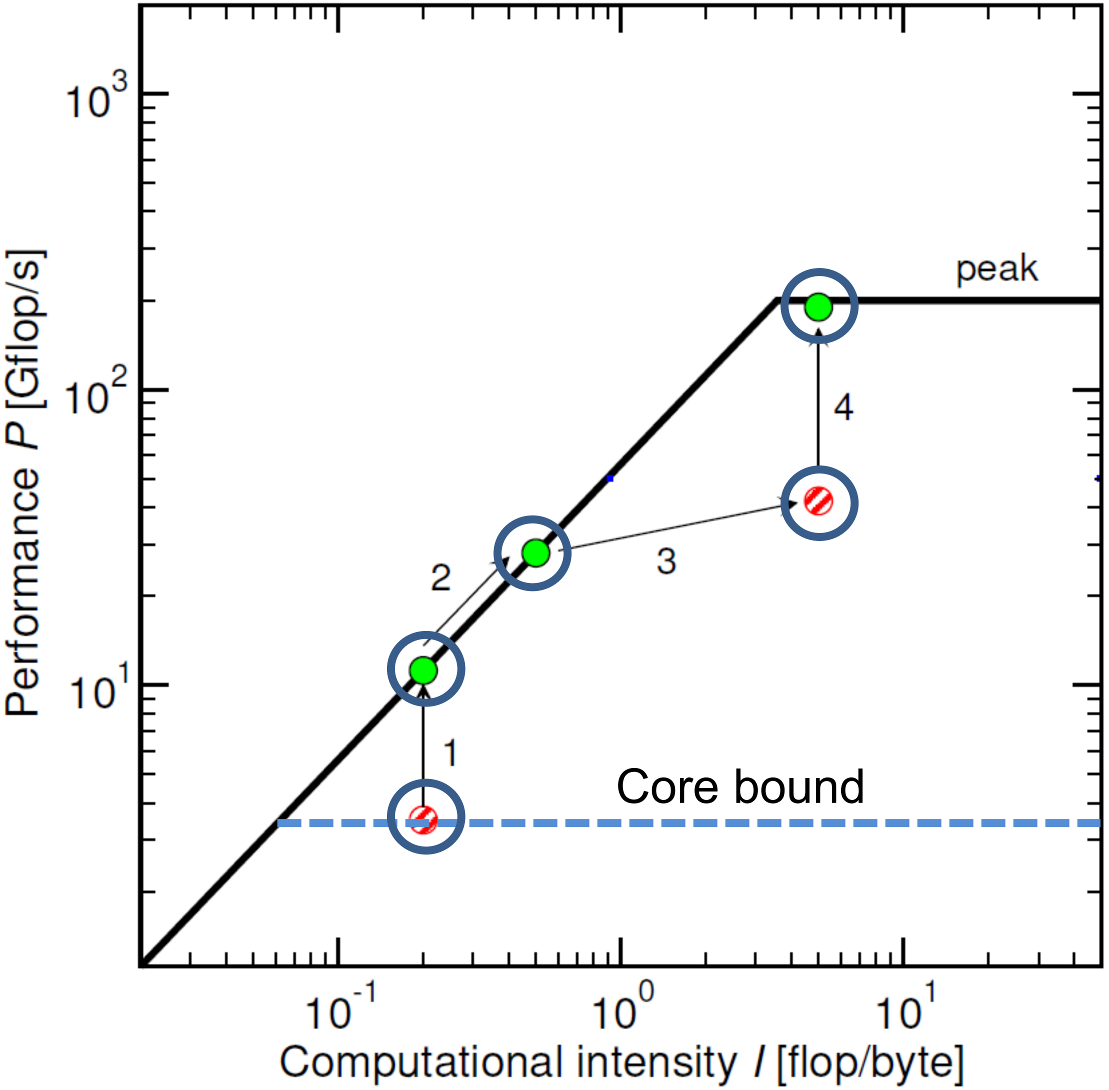
Tracking code optimizations in the Roofline Model

1. Hit the BW bottleneck by good serial code
   (e.g., Python → Fortran)

2. Increase intensity to make better use of BW bottleneck
   (e.g., spatial loop blocking)

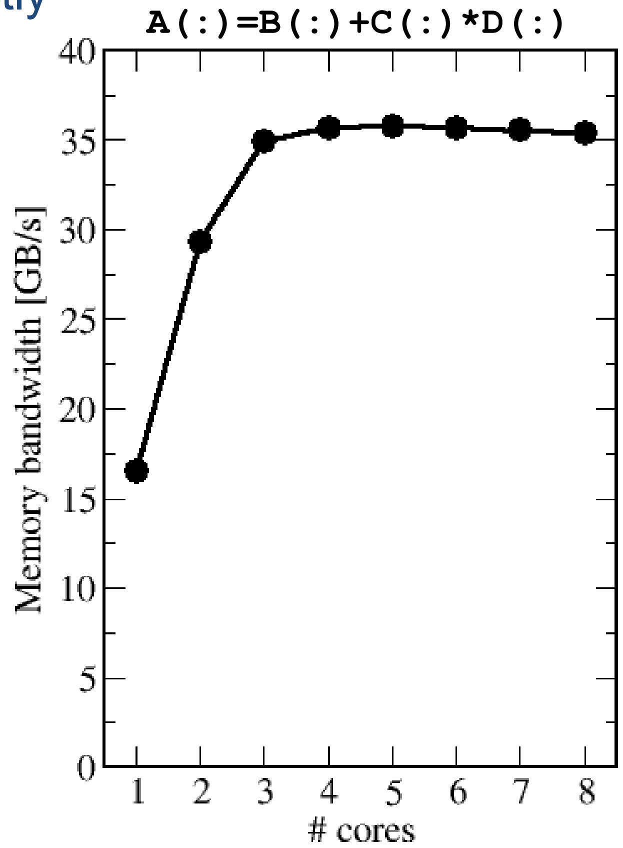
3. Increase intensity and go from memory bound to core bound
   (e.g., temporal blocking)

4. Hit the core bottleneck by good serial code
   (e.g., `-fno-alias`, SIMD intrinsics)



Shortcomings of the roofline model

- Saturation effects in multicore chips are not explained
  - Reason: “saturation assumption”
  - Cache line transfers and core execution do sometimes not overlap perfectly
  - It is not sufficient to measure single-core STREAM to make it work
  - Only increased “pressure” on the memory interface can saturate the bus
    → need more cores!



- In-cache performance is not accurately predicted

- The ECM performance model gives more insight:

G. Hager, J. Treibig, J. Habich, and G. Wellein: Exploring performance and power properties of modern multicore chips via simple machine models. Concurrency and Computation: Practice and Experience (2013).
DOI: 10.1002/cpe.3180 Preprint: arXiv:1208.2908

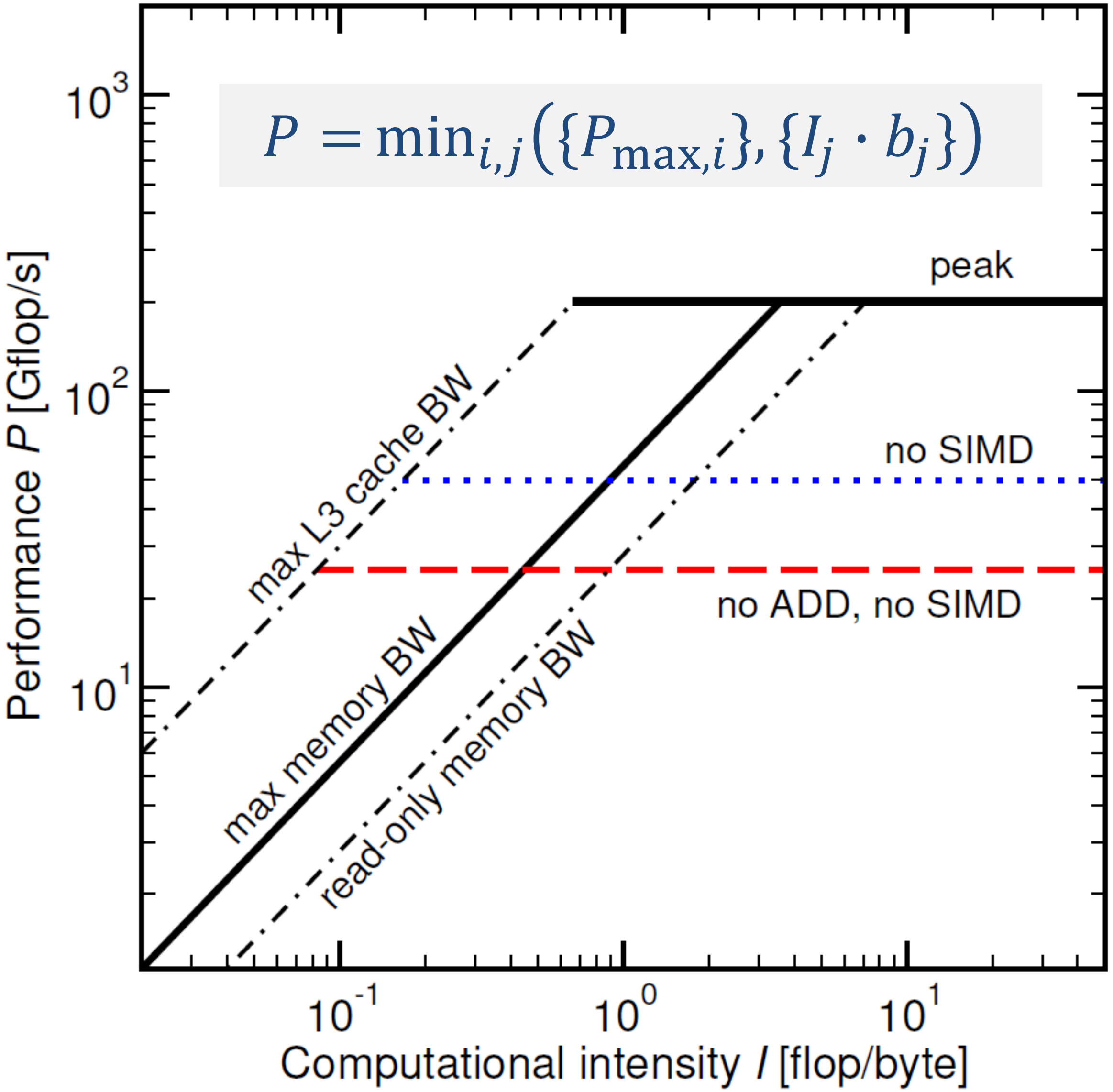
Multi-ceiling Roofline model: graphical representation

Multiple ceilings may apply

- Different bandwidths / data paths
  → different inclined ceilings
  → possibly different I (and B_c) for the same kernel

- Different $P_{\max}$
  → different flat ceilings

  In fact, $P_{\max}$ should always come from code analysis; generic ceilings are usually impossible to attain



$$P = \min_{i,j}\left(\{P_{\max,i}\}, \{I_j \cdot b_j\}\right)$$

What about multiple loops (i.e., solvers)?

Performance-based formulation is inadequate → go back to time

Solver: s components $j = 1 \dots s$, t_j = model time for component j

$$t_{solver} = \sum_{j=1}^{s} t_j = \sum_{j=1}^{s} f\left(T_{1,j}, \dots, T_{N,j}\right)$$

“Roofline”:

$$t_{solver} = \sum_{j=1}^{s} \max\left(T_{flops,j}, T_{BW,j}\right)$$

$$P_{solver} = \frac{\mathrm{Whatever{-}you{-}consider{-}a{-}good{-}metric{-}for{-}work}}{t_{solver}}$$

What if only the bandwidth bottleneck applies?

All loops memory bound → runtime entirely determined by data transfer

$$t_{solver} = \sum_{j=1}^{s} \max\left(\frac{W_{flops,j}}{R_{flops,j}}, \frac{W_{BW,j}}{R_{BW,j}}\right) = \sum_{j=1}^{s} \frac{W_{BW,j}}{R_{BW,j}} = \frac{1}{R_{BW}} \underbrace{\sum_{j=1}^{s} W_{BW,j}}_{\text{Overall data volume } V} = \frac{V}{b_S}$$

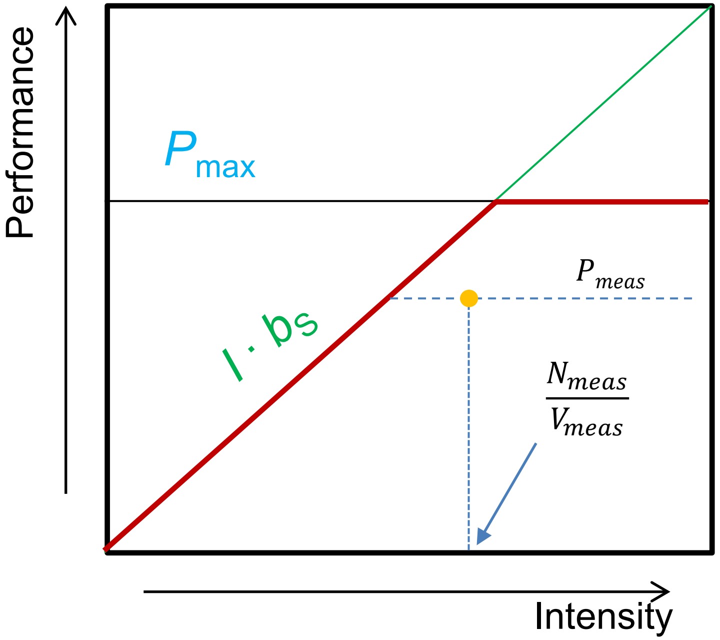
Overall “work”

$$P_{solver} = \frac{W}{t_{solver}} = \frac{W}{V} \times b_S = I \times b_S$$

Diagnostic / phenomenological Roofline modeling

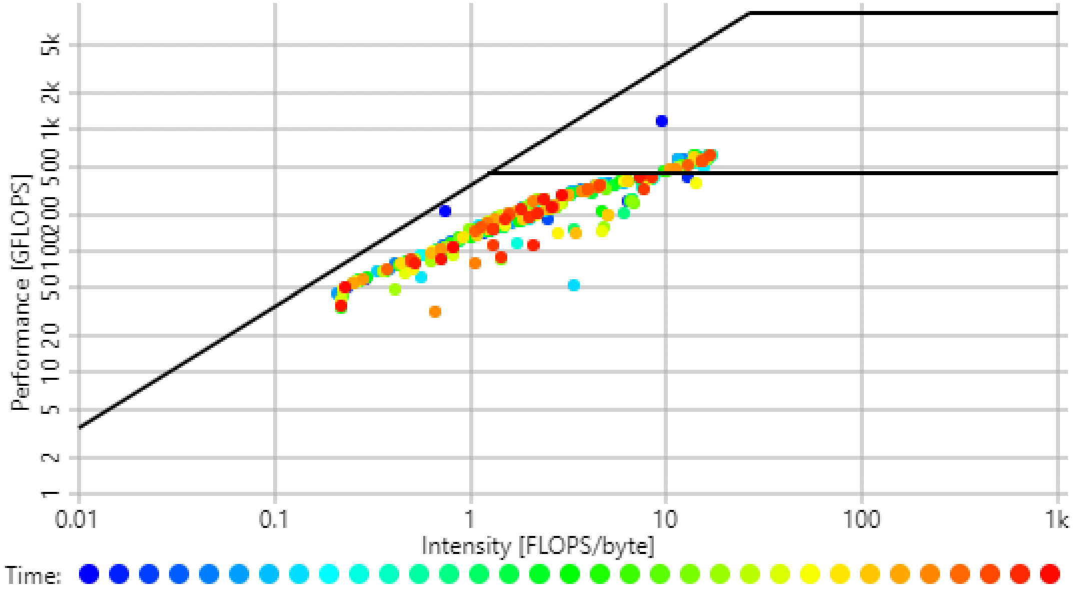
Diagnostic modeling

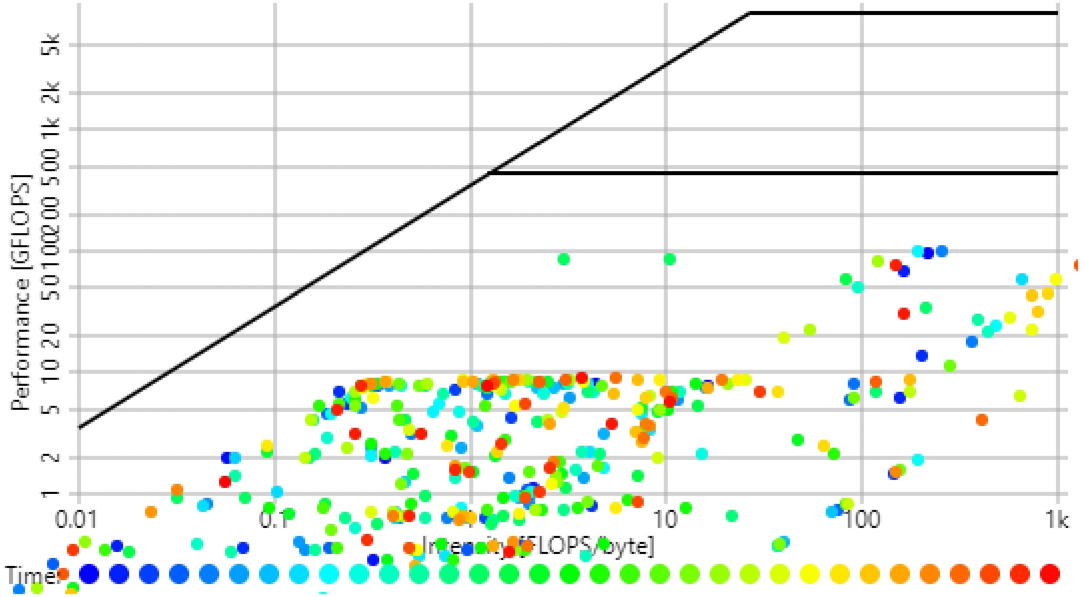
- What if we cannot predict the intensity/balance?
  - Code very complicated
  - Code not available
  - Parameters unknown
  - Doubts about correctness of analysis
- Measure data volume V_{meas} (and work N_{meas})
  - Hardware performance counters
  - Tools: likwid-perfctr, PAPI, Intel Vtune,…
- Insights + benefits
  - Compare analytic model and measurement → validate model
  - Can be applied (semi-)automatically
  - Useful in performance monitoring of user jobs on clusters



Roofline and performance monitoring of clusters

Two cluster jobs…

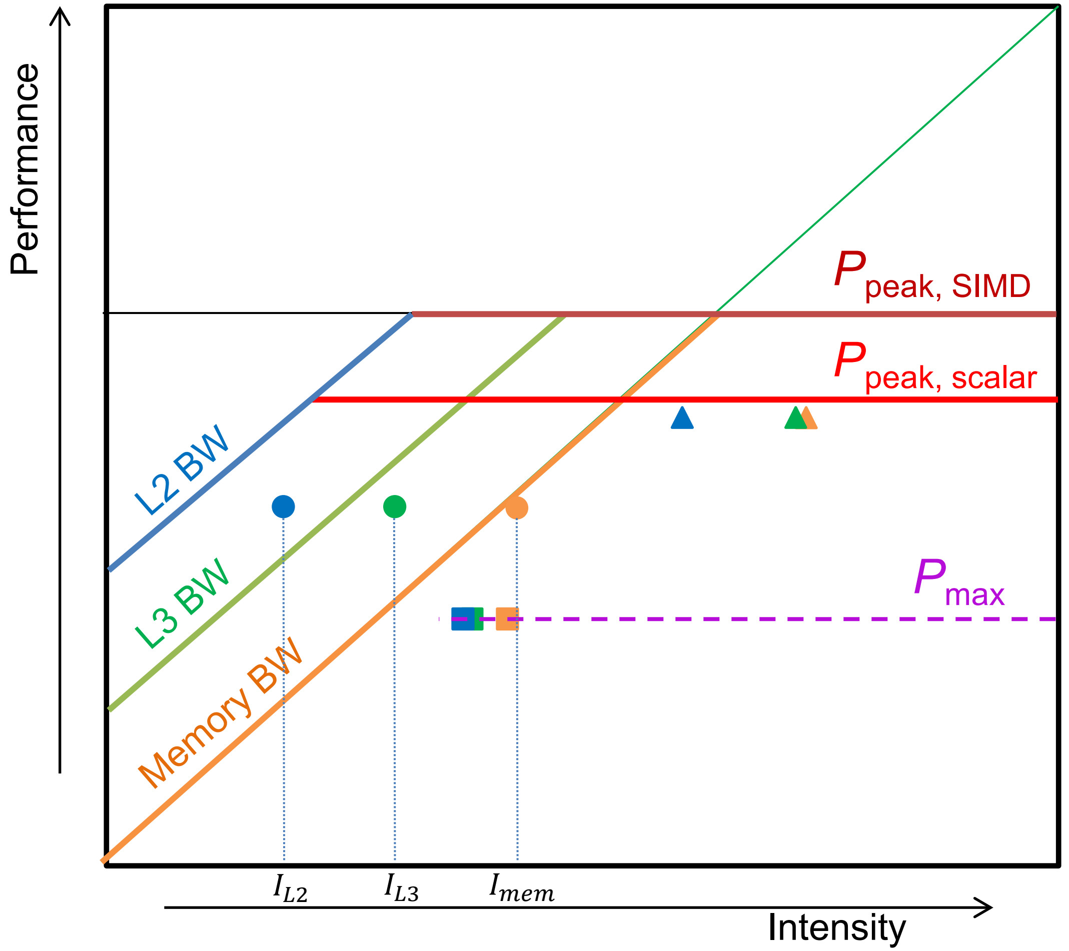




Which of them is “good” and which is “bad”?

Diagnostic modeling of a complex code (3 kernels)

Multiple bandwidth bottlenecks
→ need I for each one $(I_{mem}, I_{L3}, I_{L2}, \ldots)$



Kernel 1 ●

- Performance close to memory BW ceiling but far away from others
  → indicates **memory bound**

Kernel 2 ▲

- Performance not near any BW ceiling
- Performance close to scalar peak ceiling
  → indicates **scalar core-bound peak** code

Kernel 3 ■

- Performance not anywhere near any ceiling
  → There must be an (as yet) **unknown in-core performance limit** $P_{\max}$

Roofline conclusion

- Roofline = simple, first-principle, resource-based model for upper performance limit of data-streaming loops
  - Machine model ($P_{max}, b_S, \ldots$) + application model ($I_{mem}, I_{L3}, \ldots$)
  - Conditions apply, extensions exist

- Two modes of operation; per kernel:
  - Predictive: Calculate I_j, calculate upper limit, validate model, optimize, iterate
  - Diagnostic: Measure I_j and P, compare with ceilings

- Challenge of predictive modeling: Getting P_{max} and I_* right

Performance Analysis with Hardware Metrics

… on the example of likwid-perfctr

Tools for Node-level Performance Engineering

- **Node Information**
  */proc/cpuinfo, numactl, hwloc, **likwid-topology**, likwid-powermeter*

- **Affinity control** and data placement
  *OpenMP and MPI runtime environments, hwloc, numactl, **likwid-pin***

- **Runtime Profiling**
  Compilers, gprof, perf, HPC Toolkit, Intel Amplifier, …

- **Performance Analysis**
  *Intel VTune, **likwid-perfctr**, PAPI-based tools, HPC Toolkit, Linux perf*

- **Microbenchmarking**
  *STREAM, **likwid-bench**, lmbench, uarch-bench*

LIKWID performance tools

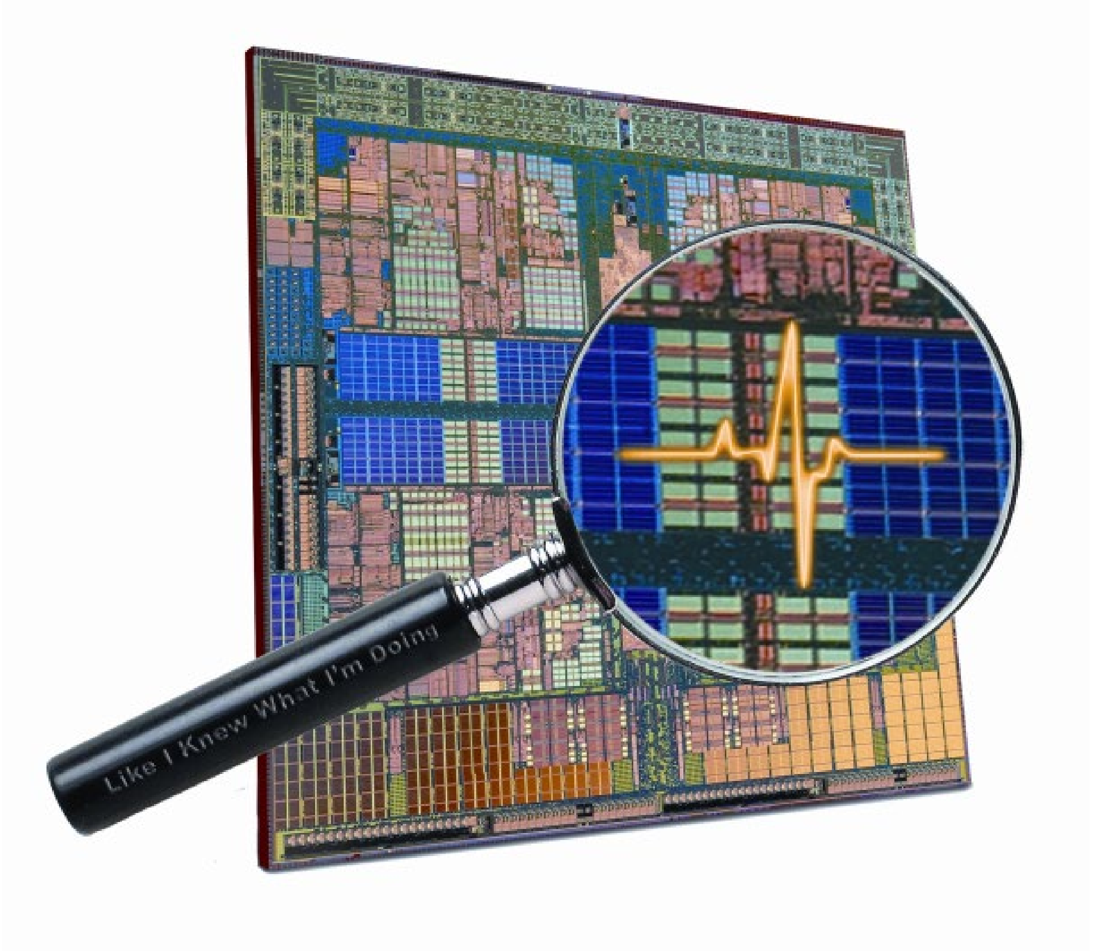
LIKWID tool suite:

Like
I
Knew
What
I’m
Doing

https://youtu.be/6uFl1HPq-88

Open source tool collection
(developed at NHR@FAU):

https://github.com/RRZE-HPC/likwid



J. Treibig, G. Hager, G. Wellein: *LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments.* PSTI2010, Sep 13-16, 2010, San Diego, CA. DOI: 10.1109/ICPPW.2010.38

LIKWID Tool Suite

- Command line tools for Linux:

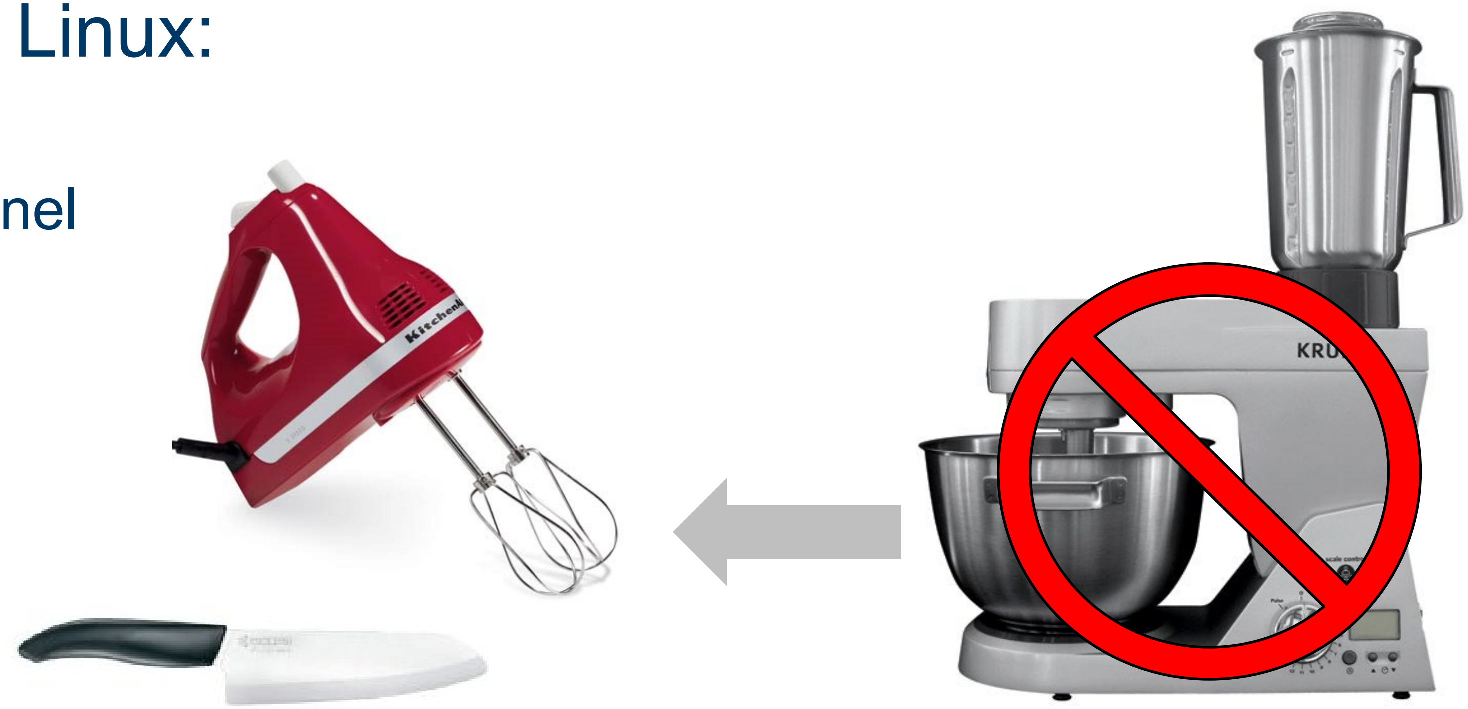
  easy to install

  works with standard Linux kernel

  simple and clear to use

  supports most X86 CPUs

  (also ARMv8, POWER9 and
  Nvidia GPUs)



- Current tools:

`likwid-topology` – Print thread and cache topology

`likwid-pin` – Pin threaded application without touching code

`likwid-mpirun` – Start MPI and MPI/OpenMP hybrid programs

`likwid-perfctr` – Measure performance counters

`likwid-bench` – Microbenchmarking tool and environment

… some more

Probing performance behavior

- How do we find out about the performance properties and resource usage of a running code?
- How do we validate analytic (white- or gray-box) performance models?
- How do we get the data for diagnostic RL modeling?

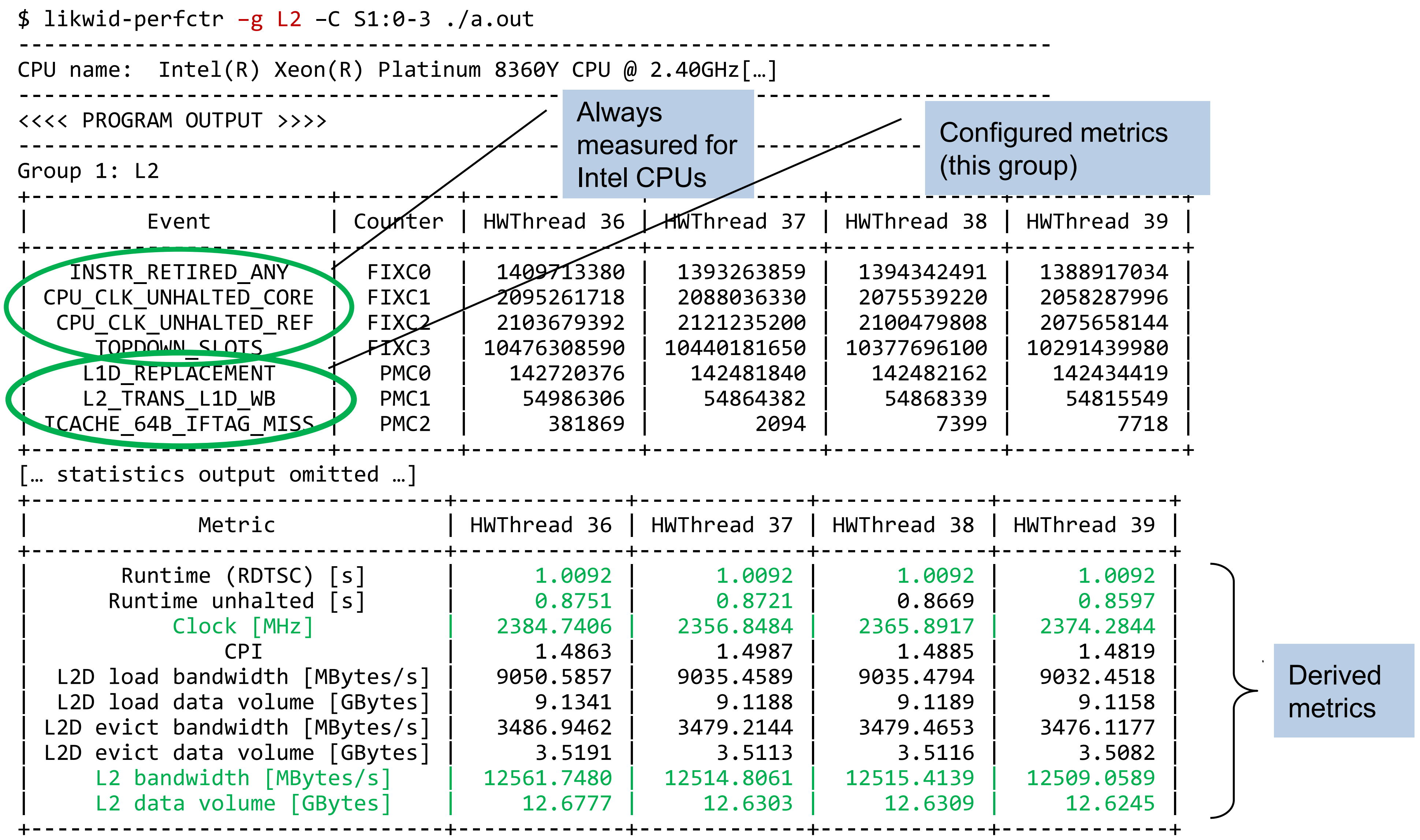
- Many tools exist, but a coarse overview is often sufficient: `likwid-perfctr`
- Simple end-to-end measurement of hardware performance metrics

  Operating modes:
  - Wrapper
  - Stethoscope
  - Timeline
  - Marker API

  Preconfigured and extensible metric groups, list with `likwid-perfctr -a`

```
BRANCH: Branch prediction miss rate/ratio
CACHE: Data cache miss rate/ratio
CLOCK: Clock frequency of cores
DATA: Load to store ratio
FLOPS_DP: Double Precision MFlops/s
FLOPS_SP: Single Precision MFlops/s
L2: L2 cache bandwidth in MBytes/s
L2CACHE: L2 cache miss rate/ratio
L3: L3 cache bandwidth in MBytes/s
L3CACHE: L3 cache miss rate/ratio
MEM: Main memory bandwidth in MBytes/s
TLB: TLB miss rate/ratio
ENERGY: Power and energy consumption
```

likwid-perfctr wrapper mode



```
$ likwid-perfctr -g L2 -C S1:0-3 ./a.out
--------------------------------------------------------------------------------
CPU name:  Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz[…]
--------------------------------------------------------------------------------
<<<< PROGRAM OUTPUT >>>>
--------------------------------------------------------------------------------
Group 1: L2
+------------------------+---------+--------------+--------------+--------------+--------------+
|          Event         | Counter | HWThread 36  | HWThread 37  | HWThread 38  | HWThread 39  |
+------------------------+---------+--------------+--------------+--------------+--------------+
|    INSTR_RETIRED_ANY   |  FIXC0  |   1409713380 |   1393263859 |   1394342491 |   1388917034 |
|  CPU_CLK_UNHALTED_CORE |  FIXC1  |   2095261718 |   2088036330 |   2075539220 |   2058287996 |
|  CPU_CLK_UNHALTED_REF  |  FIXC2  |   2103679392 |   2121235200 |   2100479808 |   2075658144 |
|     TOPDOWN_SLOTS      |  FIXC3  |  10476308590 |  10440181650 |  10377696100 |  10291439980 |
|    L1D_REPLACEMENT     |   PMC0  |    142720376 |    142481840 |    142482162 |    142434419 |
|     L2_TRANS_L1D_WB    |   PMC1  |     54986306 |     54864382 |     54868339 |     54815549 |
|  ICACHE_64B_IFTAG_MISS |   PMC2  |       381869 |         2094 |         7399 |         7718 |
+------------------------+---------+--------------+--------------+--------------+--------------+
[… statistics output omitted …]
+---------------------------------+--------------+--------------+--------------+--------------+
|              Metric             | HWThread 36  | HWThread 37  | HWThread 38  | HWThread 39  |
+---------------------------------+--------------+--------------+--------------+--------------+
|       Runtime (RDTSC) [s]       |       1.0092 |       1.0092 |       1.0092 |       1.0092 |
|      Runtime unhalted [s]       |       0.8751 |       0.8721 |       0.8669 |       0.8597 |
|           Clock [MHz]           |    2384.7406 |    2356.8484 |    2365.8917 |    2374.2844 |
|               CPI               |       1.4863 |       1.4987 |       1.4885 |       1.4819 |
|  L2D load bandwidth [MBytes/s]  |    9050.5857 |    9035.4589 |    9035.4794 |    9032.4518 |
|  L2D load data volume [GBytes]  |       9.1341 |       9.1188 |       9.1189 |       9.1158 |
| L2D evict bandwidth [MBytes/s]  |    3486.9462 |    3479.2144 |    3479.4653 |    3476.1177 |
| L2D evict data volume [GBytes]  |       3.5191 |       3.5113 |       3.5116 |       3.5082 |
|     L2 bandwidth [MBytes/s]     |   12561.7480 |   12514.8061 |   12515.4139 |   12509.0589 |
|     L2 data volume [GBytes]     |      12.6777 |      12.6303 |      12.6309 |      12.6245 |
+---------------------------------+--------------+--------------+--------------+--------------+
```

Always measured for Intel CPUs

Configured metrics (this group)

Derived metrics

likwid-perfctr stethoscope mode

- likwid-perfctr counts events on cores; it has no notion of what kind of code is running (if any)

  This allows you to "listen" to what is currently happening, without any overhead:

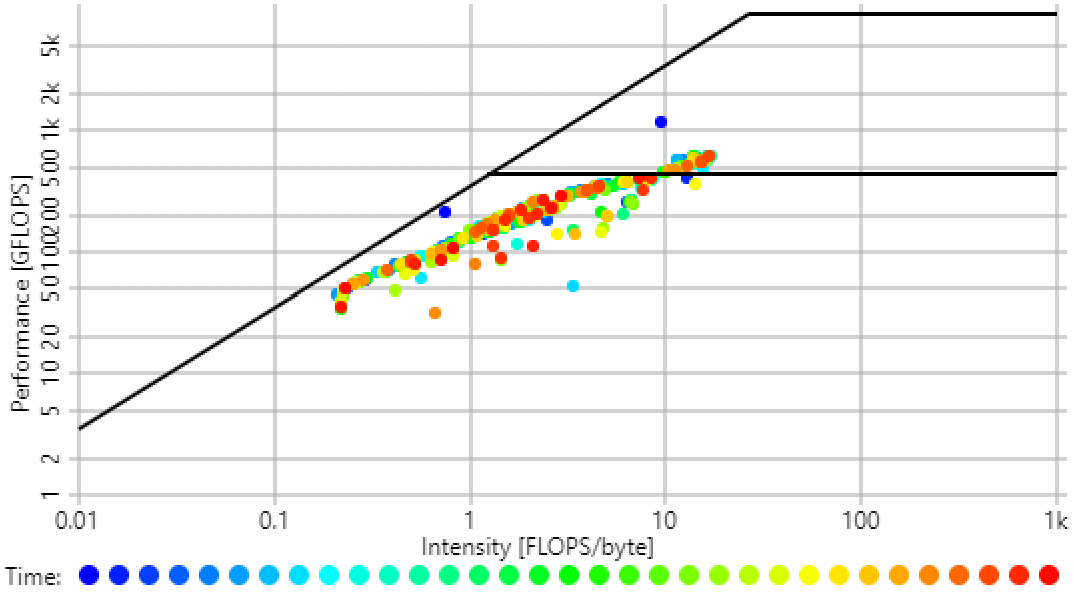
  ```
  $ likwid-perfctr -c N:0-11 -g FLOPS_DP  -S 10s
  ```

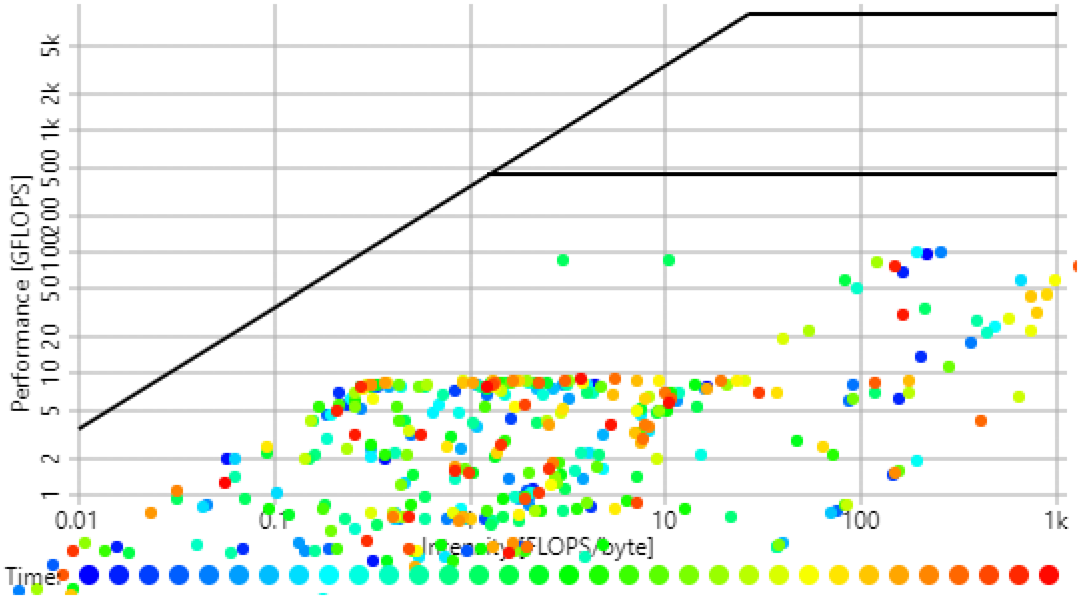
- Can be used as cluster/server monitoring tool

- Frequent use: monitor a long-running parallel application from outside

Cluster monitoring with likwid-perfctr

Two jobs on the NHR@FAU “Fritz” cluster





https://github.com/ClusterCockpit

`likwid-perfctr` with MarkerAPI

- The MarkerAPI can restrict measurements to code regions
- The API only reads counters.
  The configuration of the counters is still done by `likwid-perfctr`
- Multiple named regions allowed, accumulation over multiple calls
- Inclusive and overlapping regions allowed

- Caveat: Marker API can cause overhead; do not call too frequently!

```
#include <likwid-marker.h>

LIKWID_MARKER_INIT; // must be called from serial region
. . .
LIKWID_MARKER_START(“Compute”);
. . .
LIKWID_MARKER_STOP(“Compute”);
. . .
LIKWID_MARKER_START(“Postprocess”);
. . .
LIKWID_MARKER_STOP(“Postprocess”);
. . .
LIKWID_MARKER_CLOSE; // must be called from serial region
```

likwid-perfctr with MarkerAPI: OpenMP code (C)

```
#include <likwid-marker.h>

int main(...) {
  LIKWID_MARKER_INIT;
  #pragma omp parallel
  {
    LIKWID_MARKER_REGISTER(“MatrixAssembly”);
  }
  ...
  #pragma omp parallel
  {
    LIKWID_MARKER_START(“MatrixAssembly”);
    #pragma omp for
    for(int i=0; i<N; ++i) { /* Loop */ }
    LIKWID_MARKER_STOP(“MatrixAssembly”);
  }
  ...
  LIKWID_MARKER_CLOSE;
}
```

Optional: Prepare data structures (reduced overhead on 1st marker call)

Call markers in parallel region if data should be taken on all threads

https://github.com/RRZE-HPC/likwid/wiki/TutorialMarkerC

likwid-perfctr with MarkerAPI: OpenMP code (Fortran)

```
program p
  use likwid
  call likwid_markerInit
  !$omp parallel
    call likwid_markerRegisterRegion(“MatrixAssembly”)
  !$omp end parallel
  ...
  !$omp parallel
    call likwid_markerStartRegion(“MatrixAssembly”)
    !$omp do
    do i=1,N
     ! Loop
    enddo
    !$omp end do
    call likwid_markerStopRegion(“MatrixAssembly”)
  !$omp end parallel
  ...
  call likwid_markerClose
end program p
```

Optional: Prepare data structures (reduced overhead on 1st marker call)

Call markers in parallel region if data should be taken on all threads

https://github.com/RRZE-HPC/likwid/wiki/TutorialMarkerF90

Compiling, linking, and running with marker API

Compile:

```
cc -I /path/to/likwid.h -DLIKWID_PERFMON -c program.c
```

Activate LIKWID macros (C only)

Link:

```
cc -L /path/to/liblikwid program.o –o program -llikwid
```

Run:

```
likwid-perfctr -C <CPULIST> -g <GROUP> -m ./program
```

Activate markers

MPI:

```
likwid-mpirun –np 4 –pin <PINEXPR> –g <GROUP> -m ./program
```

→ One separate block of output for every marked region

So... what should I look at first?

Focus on resource utilization and instruction decomposition!

Metrics to look at:

- Operation throughput (Flops/s)
- Overall instruction throughput (IPC,CPI)
- Instruction breakdown:
  - FP instructions
  - loads and stores
  - branch instructions
  - other instructions
- Instruction breakdown w.r.t. SIMD width (scalar, SSE, AVX, AVX512 for x86)

- Data volumes and bandwidths to main memory (GB and GB/s)
- Data volumes and bandwidth to different cache levels (GB and GB/s)

Useful diagnostic metrics are:

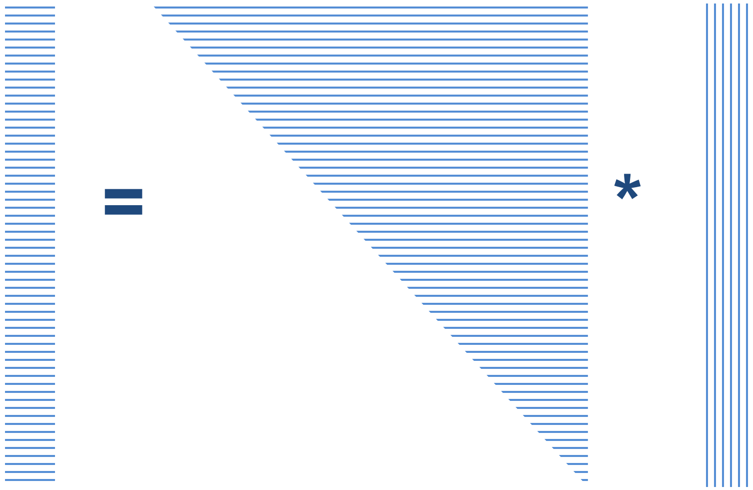
- Clock frequency (GHz)
- Power (W)

All the above metrics can be acquired using these performance groups:

```
MEM_DP, MEM_SP, BRANCH, DATA, L2,  L3
```

Example: triangular matrix-vector multiplication

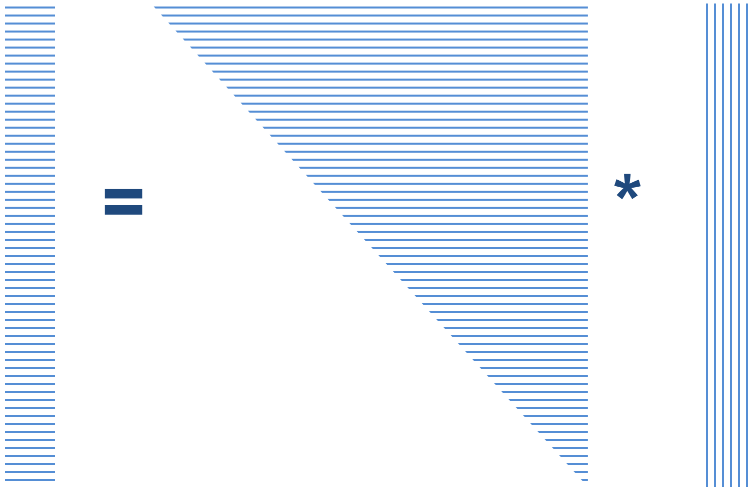
```
#define N 10000 // matrix in memory
#define ROUNDS 10
// Initialization
fillMatrix(mat, N*N, M_PI);
fillMatrix(bvec, N, M_PI);

// Calculation loop
#pragma omp parallel
{
    for (int k = 0; k < ROUNDS; k++) {
        #pragma omp for private(current,j)
        for (int i = 0; i < N; i++) {
            current = 0;
            for (int j = i; j < N; j++)
                current += mat[(i*N)+j] * bvec[j];
            cvec[i] = current;
        }
        while (cvec[N>>1] < 0) {dummy();break;}
    }
}
```



Prevent smart compilers from eliminating benchmark if **cvec** not used afterwards

Example: triangular matrix-vector multiplication

```
#include <likwid-marker.h>
[…] // defines, fillMatrix, init data
LIKWID_MARKER_INIT;
#pragma omp parallel
{
    for (int k = 0; k < ROUNDS; k++) {
        LIKWID_MARKER_START(“Compute”);
        #pragma omp for private(current,j)
        for (int i = 0; i < N; i++) {
            current = 0;
            for (int j = i; j < N; j++)
                current += mat[(i*N)+j] * bvec[j];
            cvec[i] = current;
        }
        LIKWID_MARKER_STOP(“Compute”);
        while (cvec[N>>1] < 0) {dummy();break;}
    }
}
LIKWID_MARKER_CLOSE;
```



Example: triangular matrix-vector multiplication

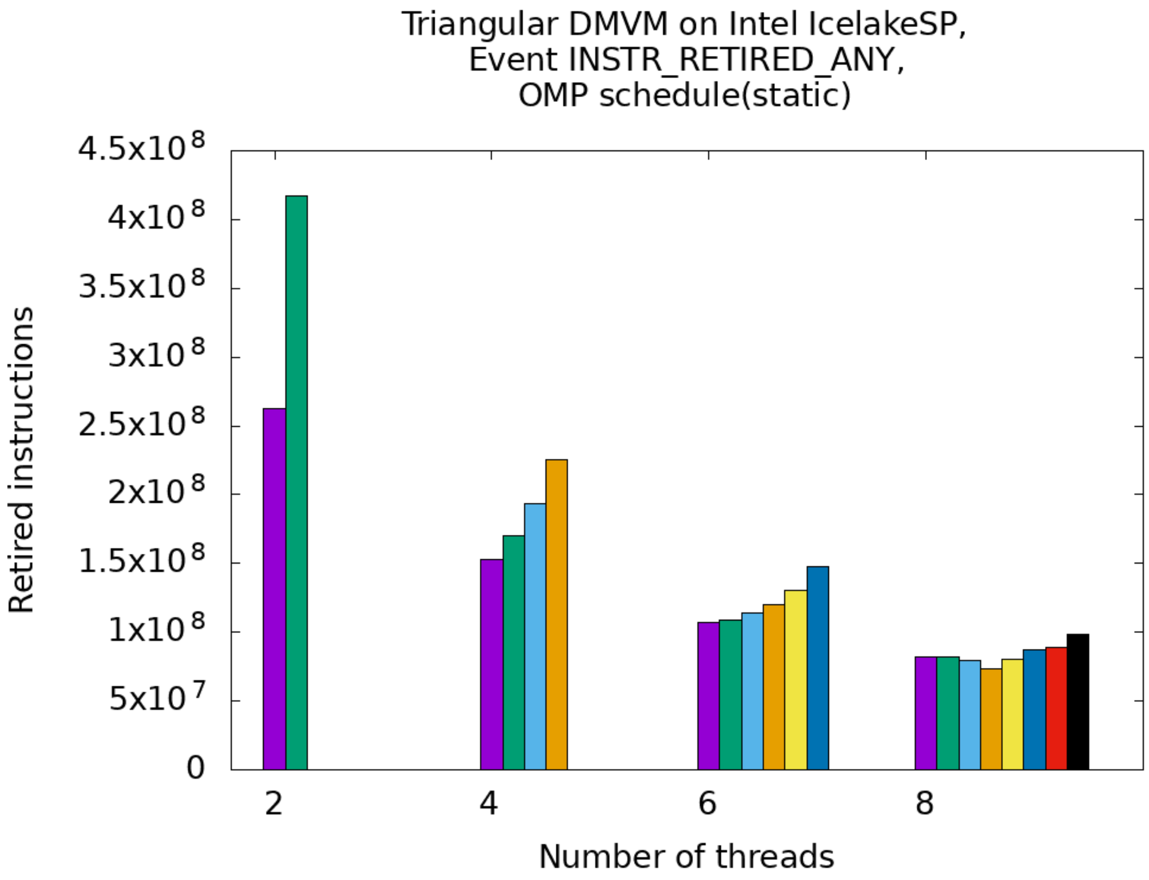
```
$ likwid-perfctr -C 0,1,2 -g L2 -m ./a.out
--------------------------------------------------------------------------------
CPU type:  Intel Icelake SP processor
CPU clock: 2.39 GHz
--------------------------------------------------------------------------------
<<<< PROGRAM OUTPUT >>>>
Region Compute, Group 1: L2
+-------------------+------------+------------+------------+
|    Region Info    | HWThread 0 | HWThread 1 | HWThread 2 |
+-------------------+------------+------------+------------+
| RDTSC Runtime [s] |   0.198263 |   0.198364 |   0.198246 |
|     call count    |         10 |         10 |         10 |
+-------------------+------------+------------+------------+

+-----------------------+---------+------------+------------+------------+
|         Event         | Counter | HWThread 0 | HWThread 1 | HWThread 2 |
+-----------------------+---------+------------+------------+------------+
|   INSTR_RETIRED_ANY   |  FIXC0  |  194399400 |  269695800 |  341470000 |
| CPU_CLK_UNHALTED_CORE |  FIXC1  |  458193600 |  464605300 |  433236300 |
|  CPU_CLK_UNHALTED_REF |  FIXC2  |  473442400 |  469863600 |  465054300 |
|     TOPDOWN_SLOTS     |  FIXC3  | 2290968000 | 2323026000 | 2166181000 |
|     L1D_REPLACEMENT   |   PMC0  |   69660770 |   41754150 |    7610321 |
|     L2_TRANS_L1D_WB   |   PMC1  |      43768 |     263047 |     442018 |
| ICACHE_64B_IFTAG_MISS |   PMC2  |       9698 |      11399 |      11571 |
+-----------------------+---------+------------+------------+------------+
```

???

Example: triangular matrix-vector multiplication

Retired instructions are misleading!

Waiting in implicit OpenMP barrier executes many instructions



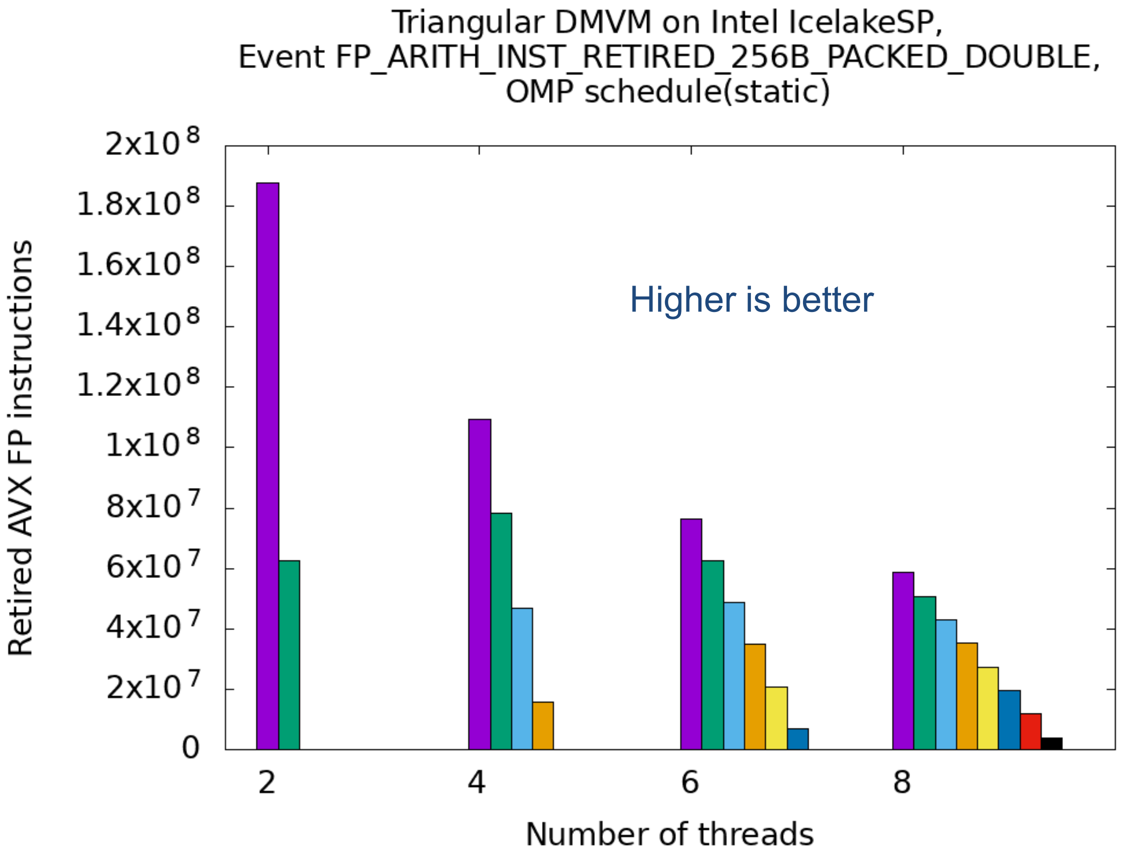
We need to measure actual work (or use a tool that can separate user from runtime lib instructions)

Example: triangular matrix-vector multiplication

Floating-point instructions reliable ↔ useful work metric

Caveats:

- FP instruction counters from SandyBridge to Haswell are only qualitatively correct
- Masked SIMD lanes cannot be counted directly on x86

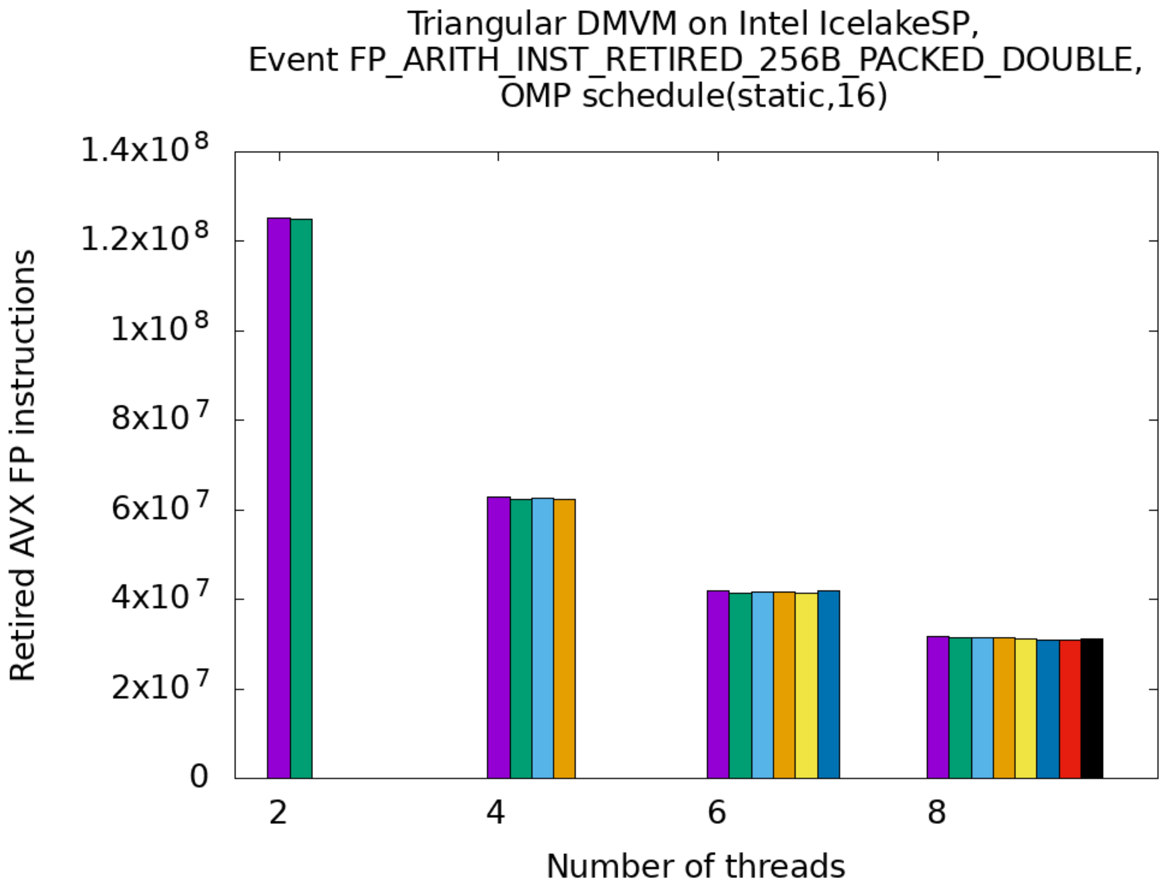


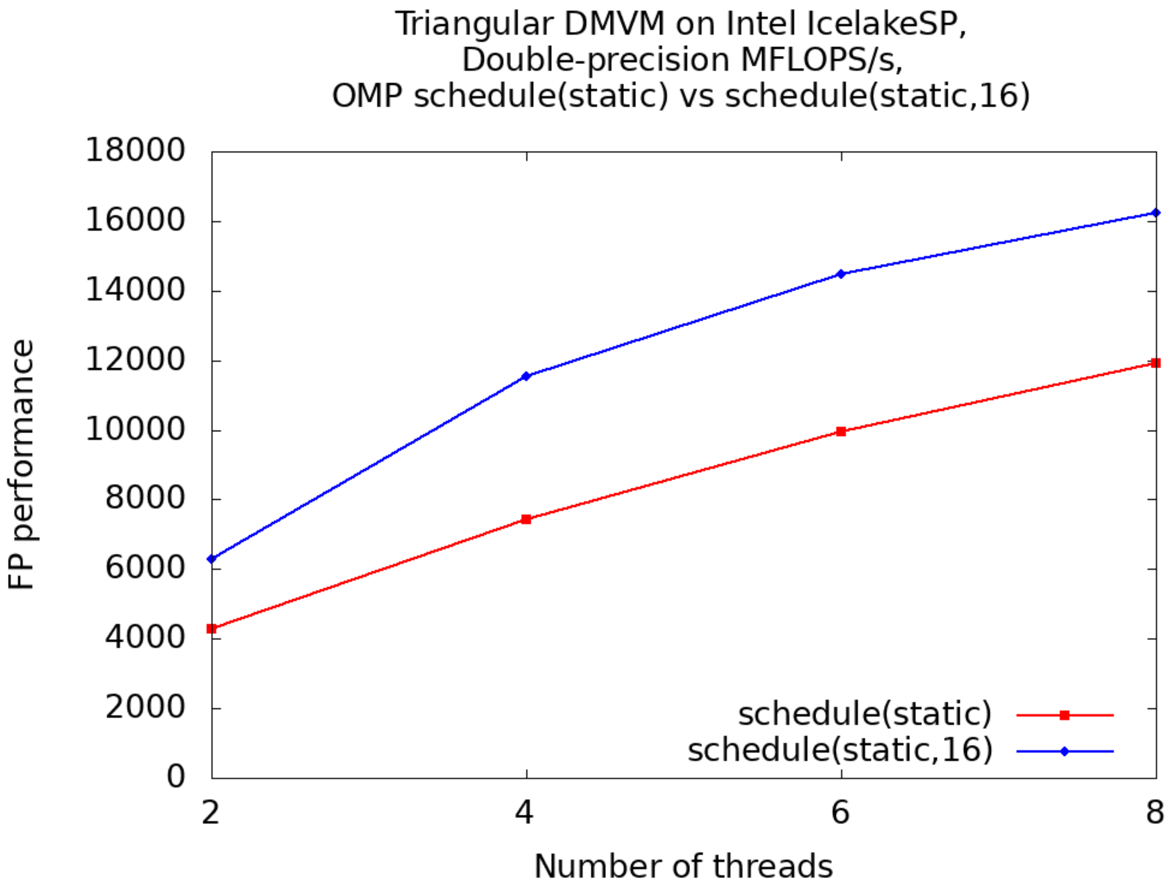
Example: triangular matrix-vector multiplication

Changing OMP schedule to static with chunk size 16 → smaller work packages per thread

No imbalance anymore!

Is it also faster?





Summary of hardware performance monitoring

- Useful only if you know what you are looking for
  - Hardware event counting bears the potential of acquiring massive amounts of data for nothing!

- Resource-based metrics are most useful
  - Cache lines transferred, work executed, loads/stores, cycles
  - Instructions, CPI, cache misses may be misleading

- Caveat: Processor work != user work
  - Waiting time in libraries (OpenMP, MPI) may cause lots of instructions
  - → distorted application characteristic
  - Advanced tools can discern user instructions from runtime library instructions

- Another very useful application of PM: validating performance models!
  - Roofline is data centric → measure data volume through memory hierarchy