

Programming Techniques for Supercomputers: Performance Modelling

Motivation

Roofline Model

Prof. Dr. G. Wellein^(a,b) , Dr. G. Hager^(a)

^(a)HPC Services – Regionales Rechenzentrum Erlangen

^(b)Department für Informatik

Friedrich-Alexander-Universität Erlangen-Nürnberg, Sommersemester 2026



A **performance model** brings together
what you need (**application** requirements) and
what you get (**hardware** capabilities)

A series of measurements from benchmarks
is NOT a performance model*

*Bill Gropp, PASC2015

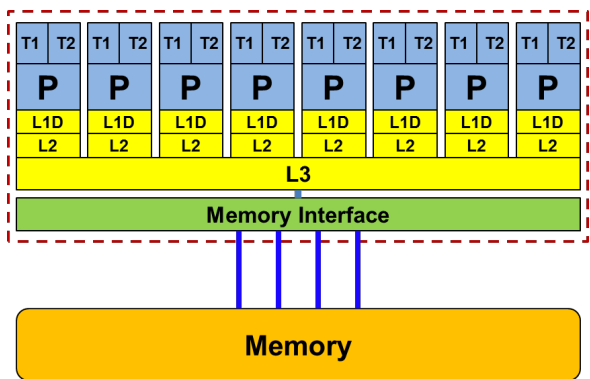
Scope of the lecture – a typical example

```

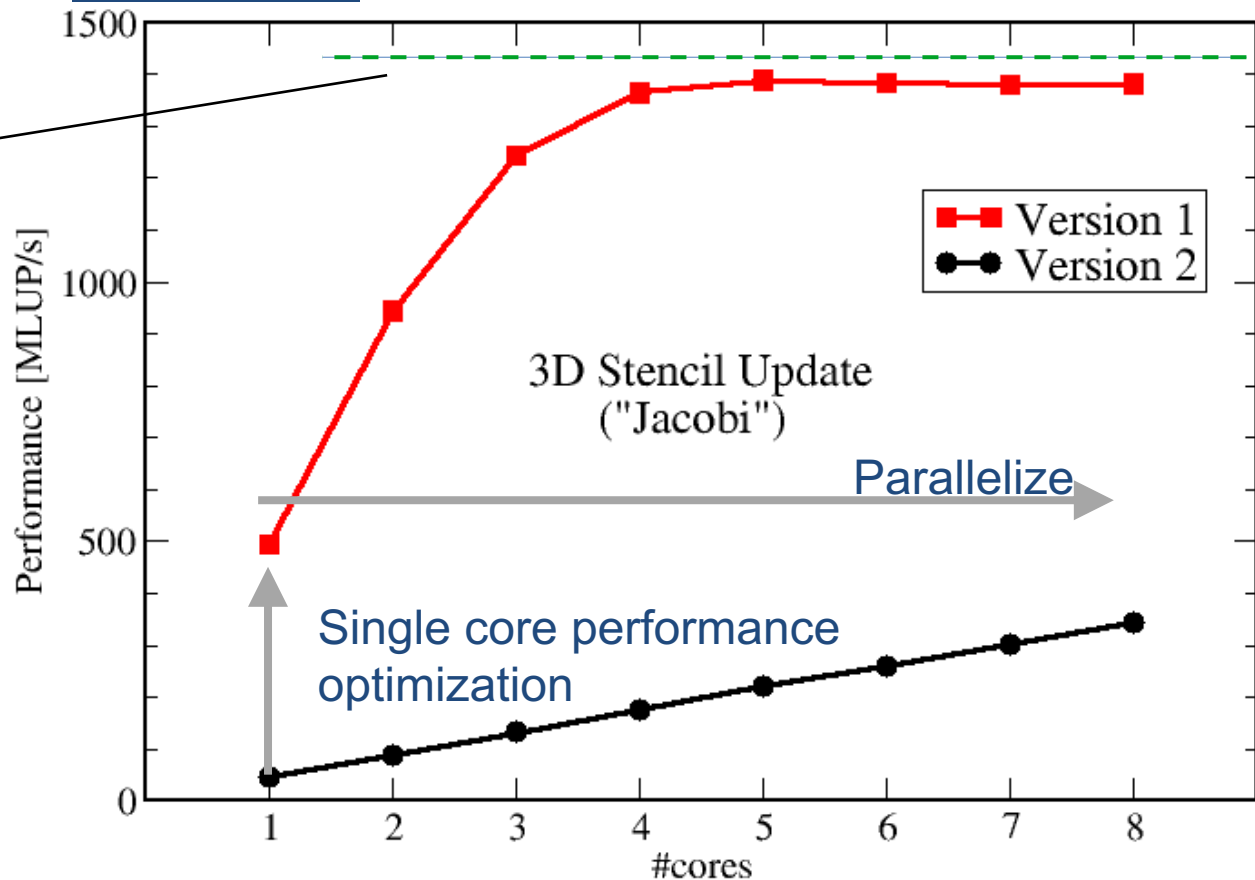
!$OMP PARALLEL DO
do k = 1 , 400
  do j = 1 , 400; do i = 1 , 400
    y(i,j,k) = b*( x(i-1,j,k)+ x(i+1,j,k)+ x(i,j-1,k)+
                  x(i,j+1,k)+ x(i,j,k-1)+ x(i,j,k+1))
  enddo; enddo
enddo
!$OMP END PARALLEL DO
    
```

Parallelize

Upper limit from simple performance model (roofline):
35 GB/s & 24 Byte/update



Intel® Xeon® Prozessor E5-2670



How model-building works: Physics

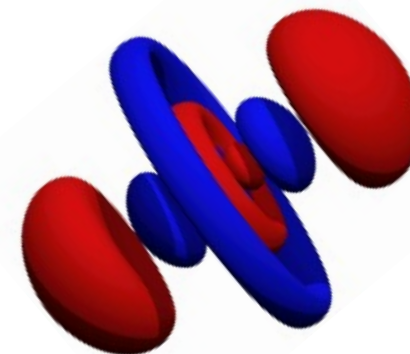
Newtonian mechanics



$$\vec{F} = m\vec{a}$$

Fails @ small scales!

Nonrelativistic
quantum
mechanics



Fails @ even smaller scales!

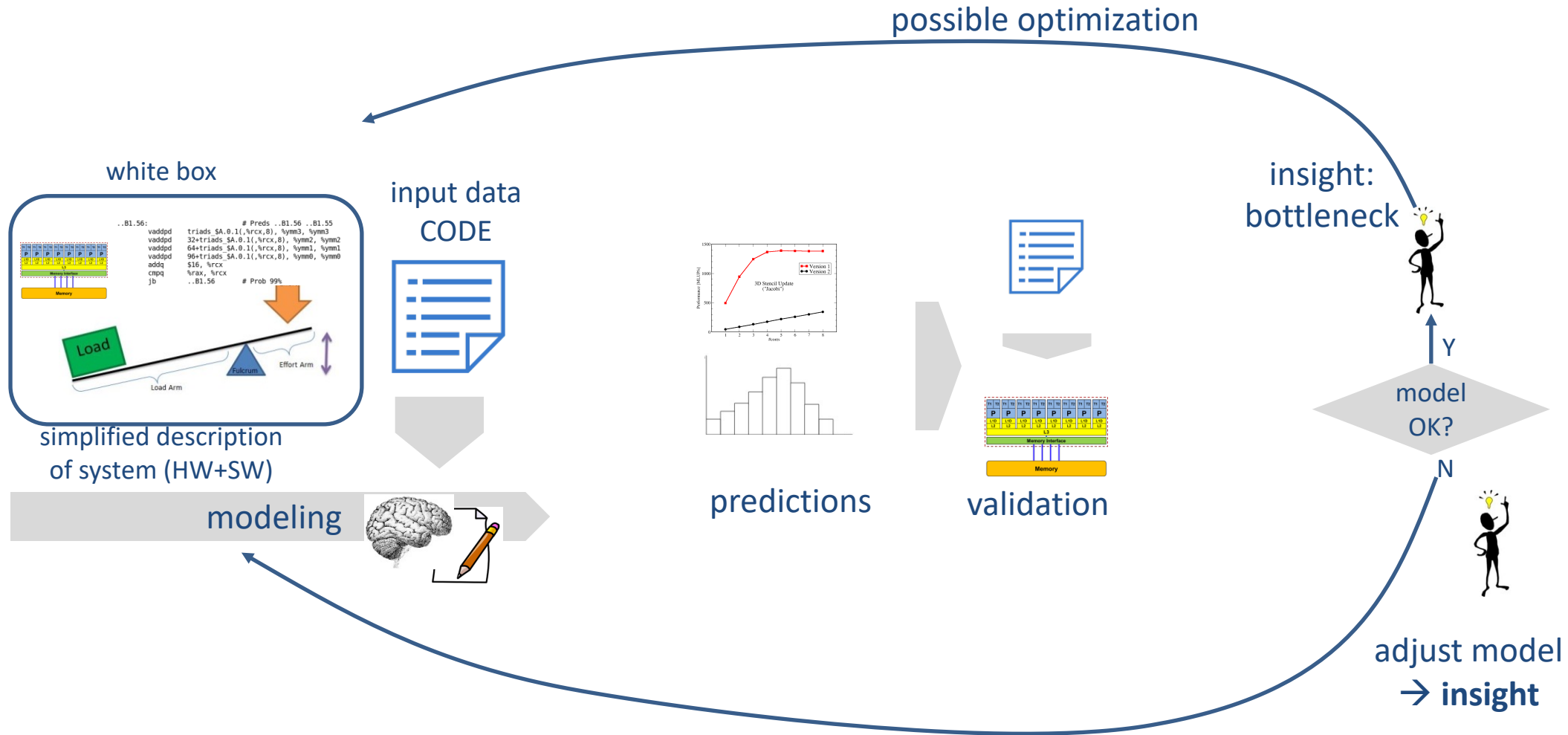
If a model fails,
we learn something!

Relativistic
quantum
field theory



$$U(1)_Y \otimes SU(2)_L \otimes SU(3)_C$$

Code optimization/parallelization – no black boxes!



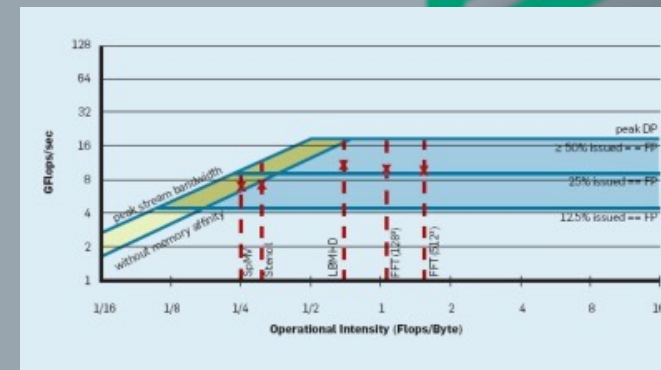
„Performance Engineering“

Questions to ask in high performance computing

- Do I understand the performance behavior of my code?
 - Does the performance **match a model** I have made?
 - What is the optimal performance for my code on a given machine?
 - **High Performance Computing == Computing at the bottleneck**
 - Can I change my code so that the “optimal performance” gets higher?
 - Circumventing/ameliorating the impact of the bottleneck
 - My **model does not work** – what’s wrong?
 - This is the good case, because **you learn something**
 - Performance monitoring / microbenchmarking may help clear up the situation
 - **Use your brain!** Tools may help, but you do the thinking.
-

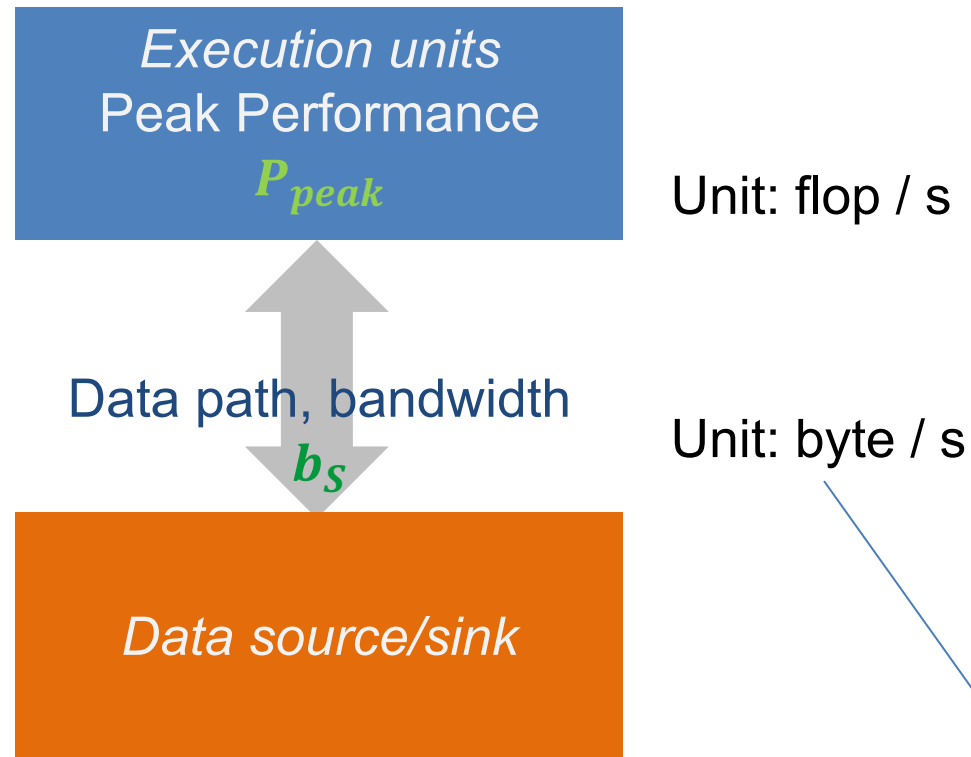
“Simple” performance modeling: The Roofline Model

Loop-based performance modeling:
Execution vs. data transfer



A simple performance model for loops

Simplistic view of hardware



Hardware Performance Bottlenecks:

- Peak Performance:
- Data path: „flop/s required by incoming data“

Simplistic view of software

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

Computational intensity

$$I = \frac{N}{V} \rightarrow \text{Unit: flop/byte}$$

P_{peak} byte/s * flop/byte [=flop/s] (“Memory bound”)
 (“Compute bound”)

Naïve Roofline Model

What performance can the software achieve on a given hardware? P [flop/s]

The performance bottleneck is either

- The execution of work (flops):
- The data path:
(requested flops by incoming data)

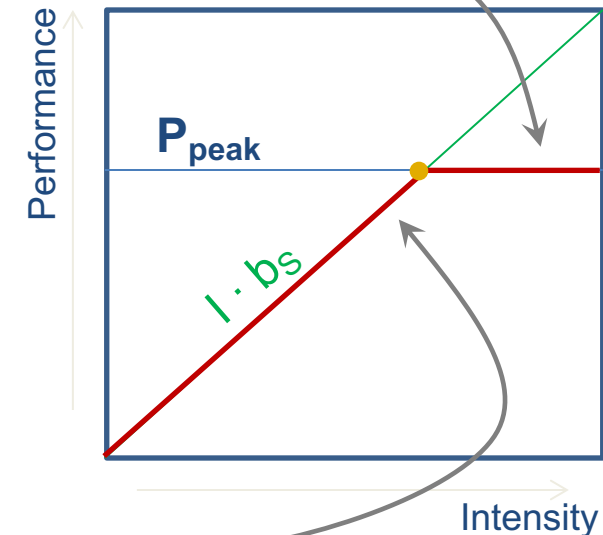
$$P_{\text{peak}} \quad [\text{flop/s}]$$

$$I \cdot b_S \quad [\text{flop/byte} \times \text{byte/s}]$$

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

This is the “Naïve Roofline Model”

- High intensity I : P limited by execution
- Low intensity I : P limited by data transfer
- “Knee” at $P_{\text{max}} = I \cdot b_S$: Best use of resources
- Roofline is an “optimistic” model (“light speed”)



Roofline Model (RLM) – Basics
Consider **two bottlenecks** only



The Roofline Model – Basics

- Hardware → Peak performance: $P_{peak} \left[\frac{F}{s} \right]$
- Hardware → Peak memory bandwidth: $b_S \left[\frac{B}{s} \right]$
- Application/SW → Computational Intensity: $I \left[\frac{F}{B} \right]$

Roofline Performance Model (RLM) - basics:

Machine model:

$$P_{peak} = 3 \frac{GF}{s}$$
$$b_S = 10 \frac{GB}{s}$$

Application model:

$$I = B_C^{-1} = 0.05 \frac{F}{B}$$

$$P = \min\left(P_{peak}, I * b_S \right) = \min\left(3 \frac{GF}{s}, 0.05 * 10 \frac{GF}{s} \right) = 0.5 \frac{GF}{s}$$

The Roofline Model: A graphical view

- Plot max. attainable performance P as a function of I (application) for a given hardware $\{P_{peak}, b_S\}$

Code feature

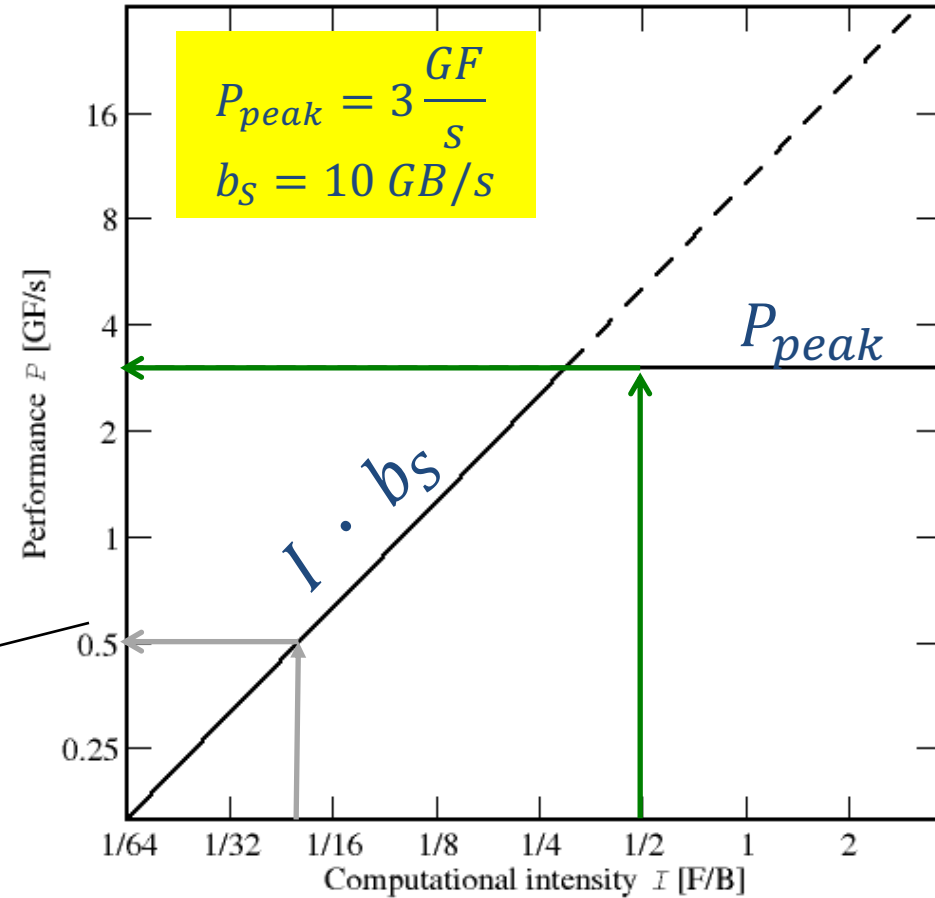
$$P = \min(P_{peak}, I \cdot b_S)$$

Hardware limitations

- Examples

- Vector triads (double prec.):
 $I = 0.05 F/B$
- Vector norm (single prec.)
 $s=s+a[i]*a[i] : I = 0.5 F/B$

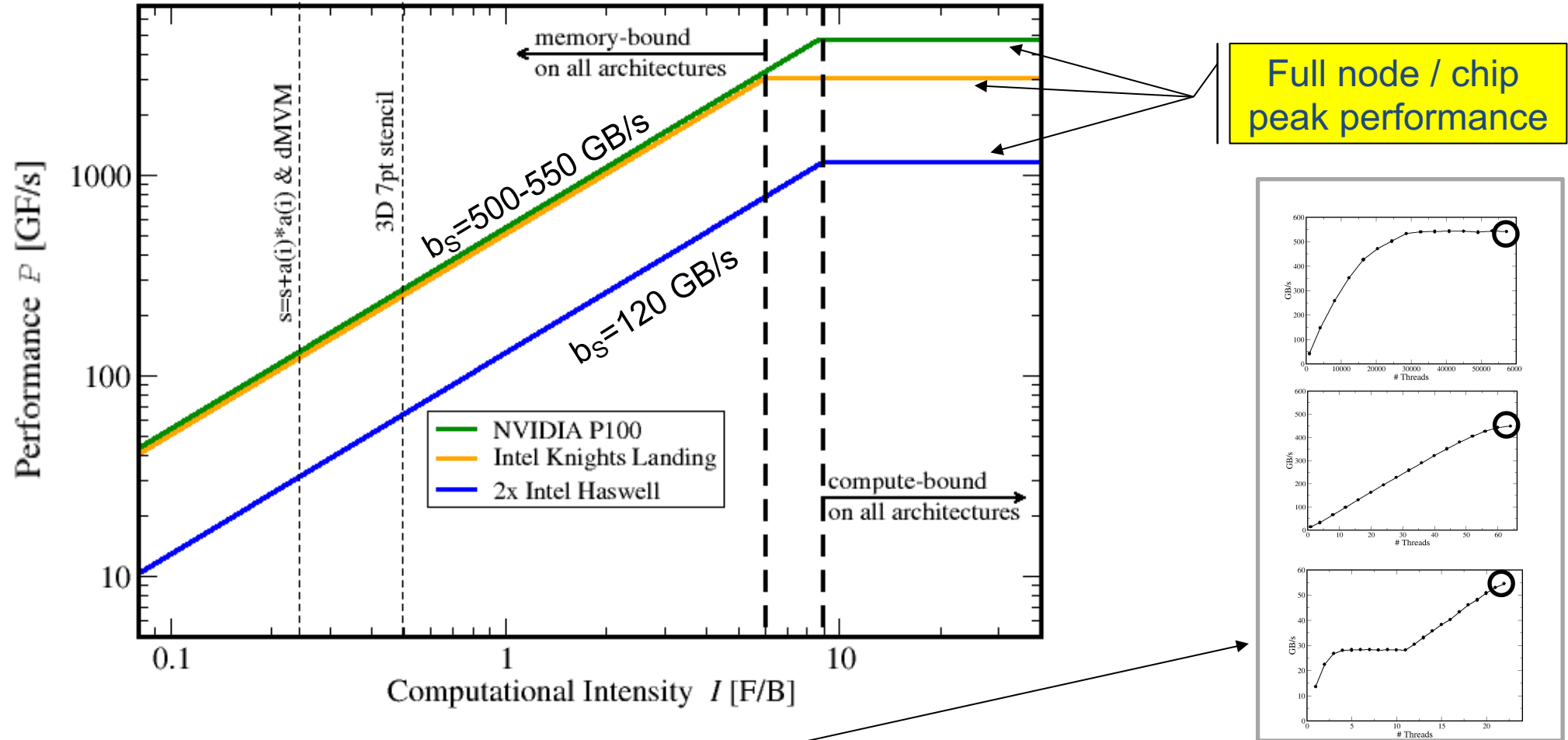
log-scale



log-scale

The Roofline Model – Basics

Compare capabilities of different machines



RLM assumption: Bandwidth saturation → Consider full chip / nodes!

The Roofline Model – Basics: Summary

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

Determine machine model for full chip/node/device:

- Peak performance $P_{peak} = P_{chip} = n_{core} \cdot n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f$
- Peak memory bandwidth: See fact sheet, e.g. $b_S = \#Channels \times f_{MEM} \times 8 \frac{B}{cycle}$

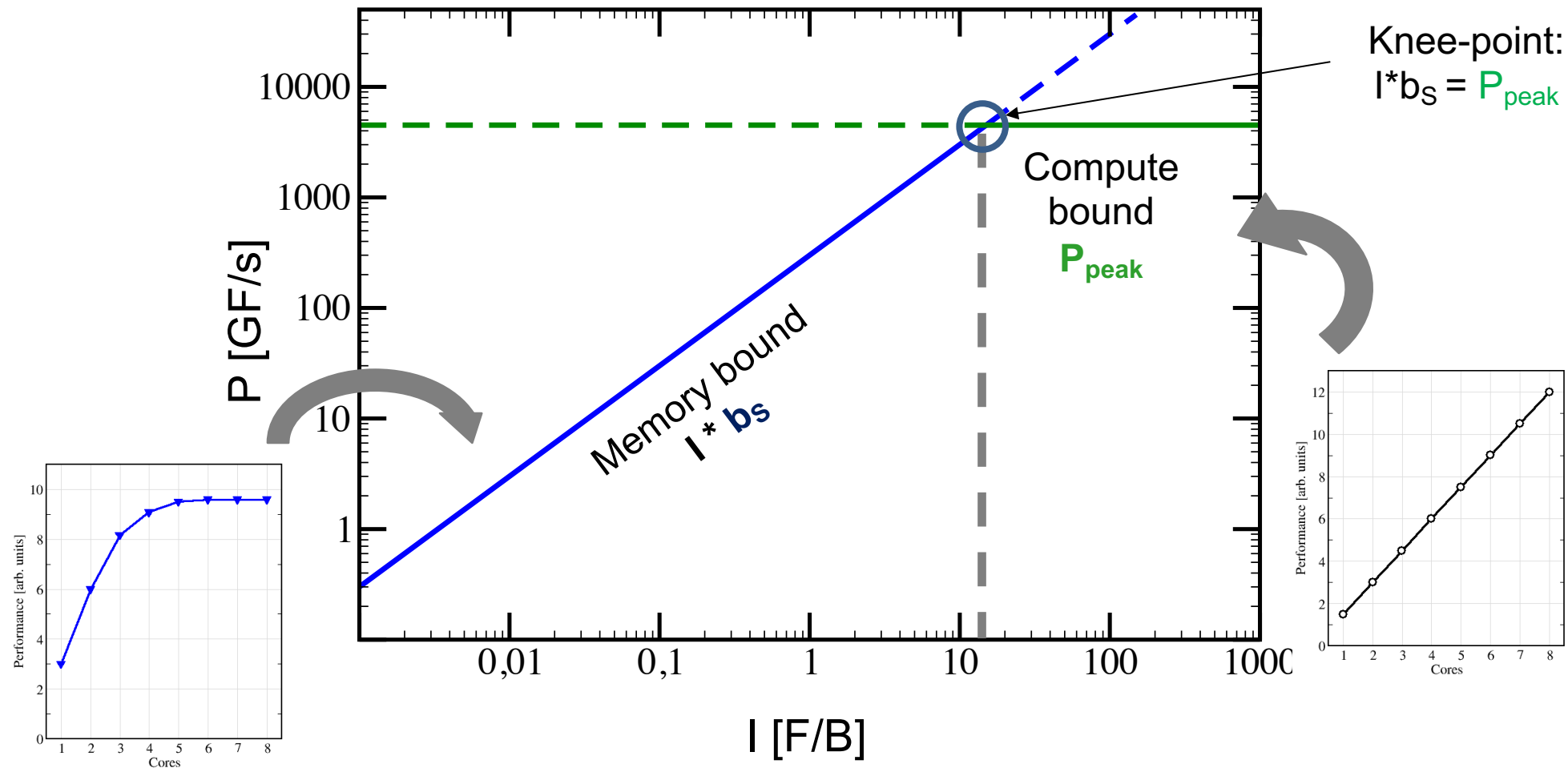
So far the model is very restricted:

- Machine and application models are completely independent
- RLM always provides upper bound – but is it realistic?
- Only two bottlenecks are considered
 - Peak Performance
 - Main memory transfers
- What if, e.g. there is no MULT and/or no SIMD vectorization?
→ P_{peak} is not a realistic limit! Implementation may have lower „horizontal roof“

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

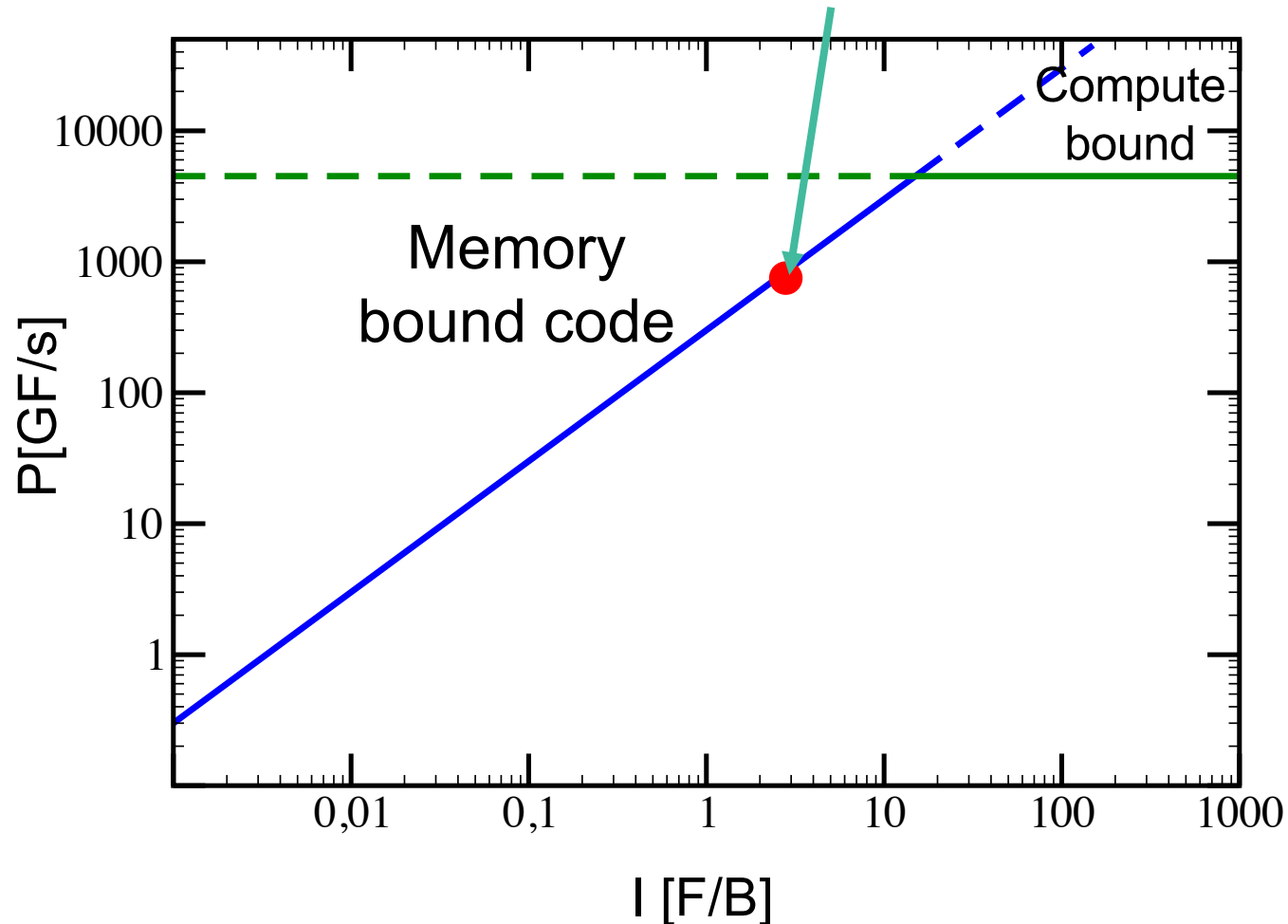
Roofline Model

Machine model with $P_{\text{peak}}=4.5 \text{ TF/s}$ and $b_s=300 \text{ GB/s}$



Roofline Model: Application information

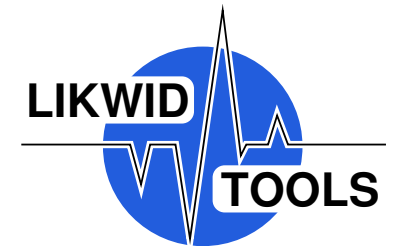
Measure application performance P and
calculate / measure application intensity I



Calculate I by
code inspection

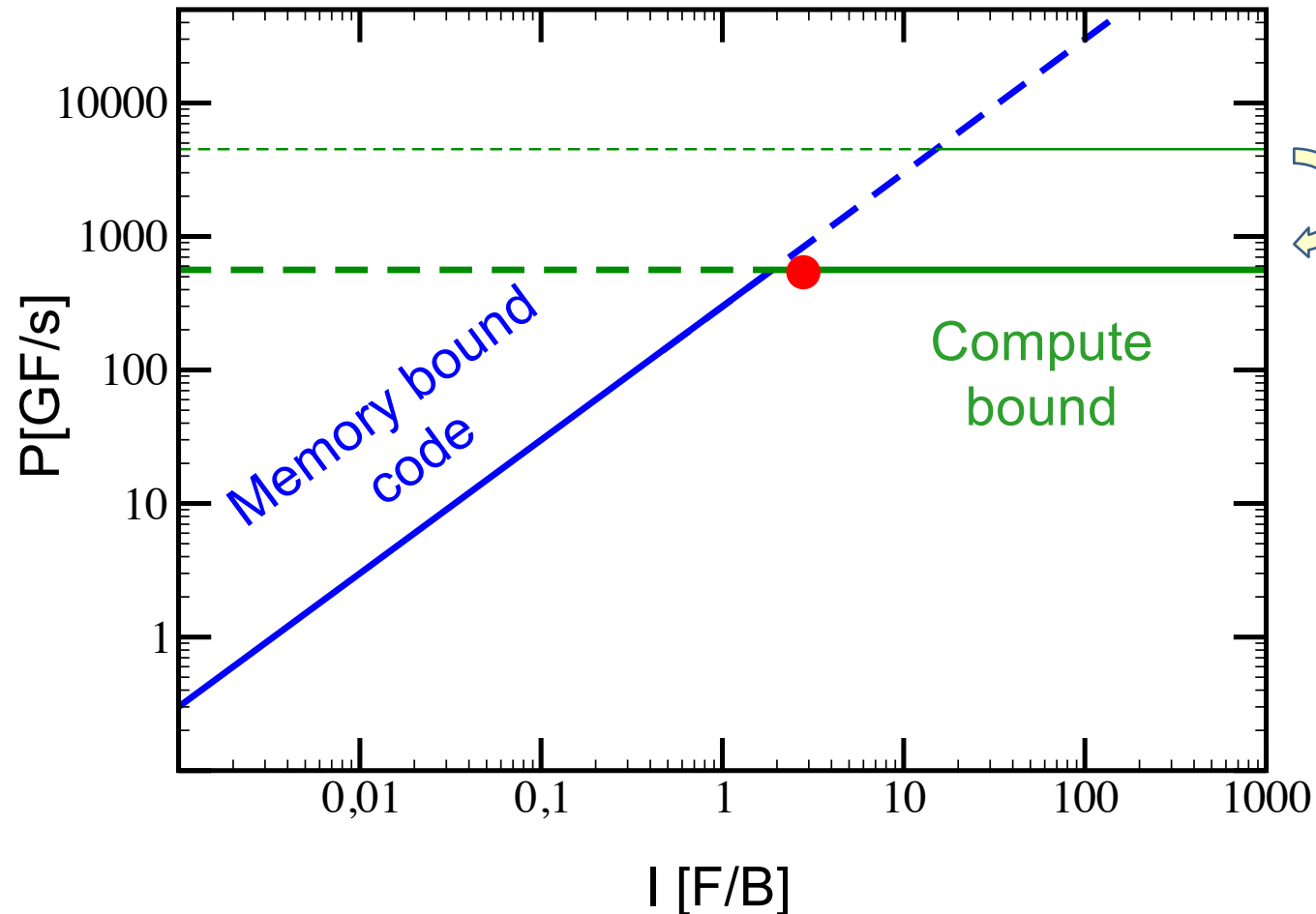
- Loop kernel
- Data structures

Measure I with



Lower horizontal roofs (P_{peak})

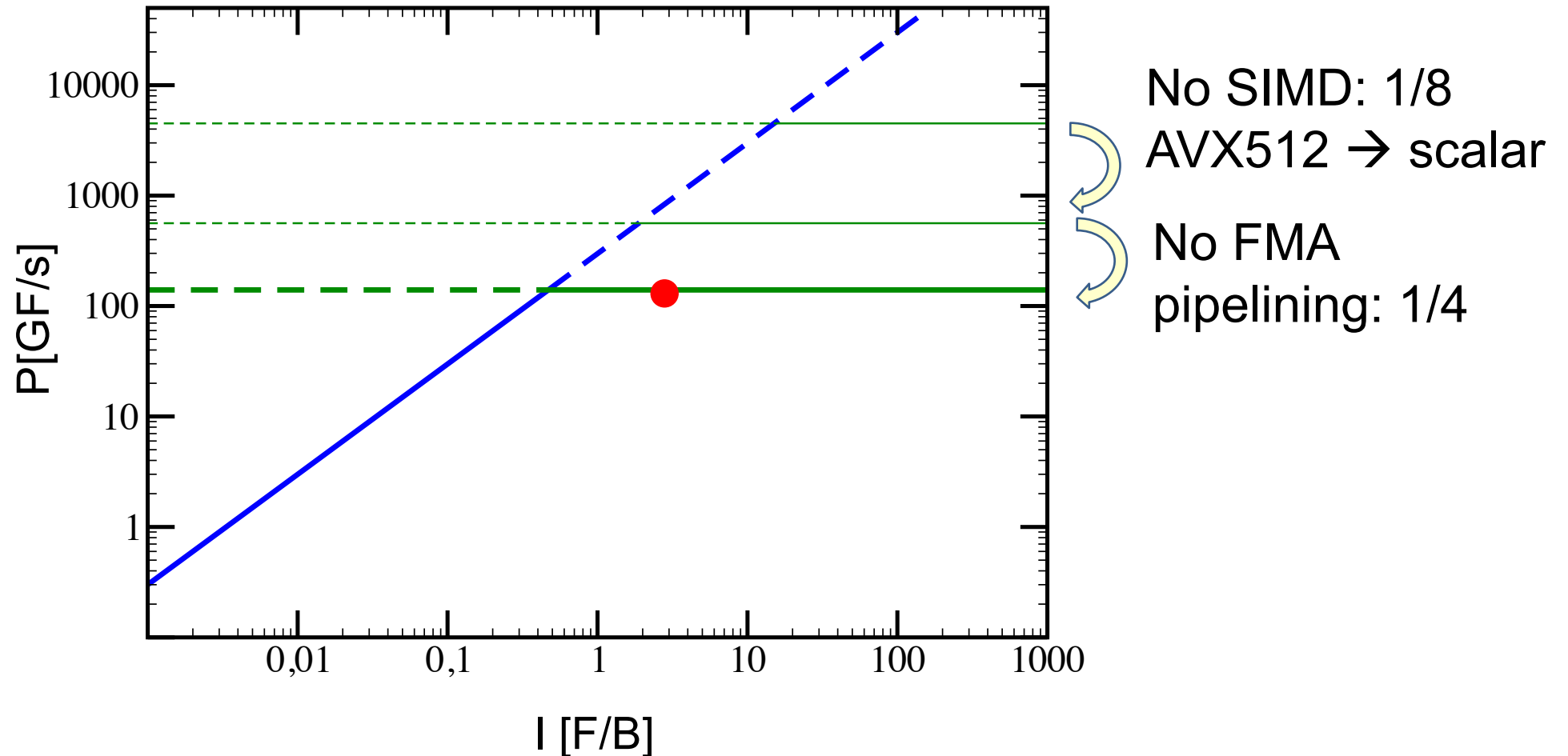
More realistic bounds for „bad“ implementations



No SIMD: 1/8
AVX512 \rightarrow scalar

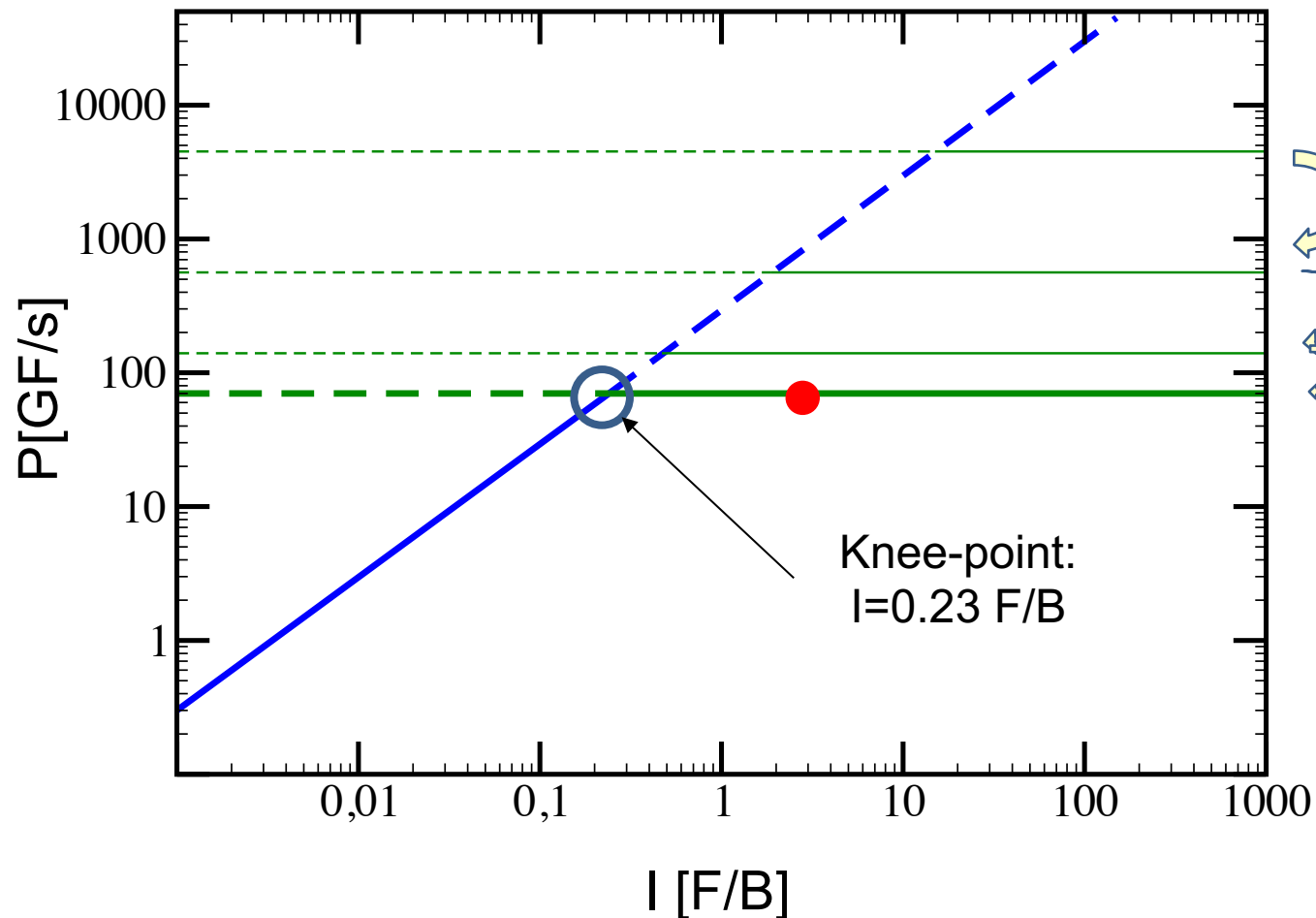
Lower horizontal roofs (P_{peak})

More realistic bounds for „bad“ implementations



Lower horizontal roofs (P_{peak})

No SIMD, no pipelining, 1 FMA only \rightarrow 64 x decrease in P_{Peak}



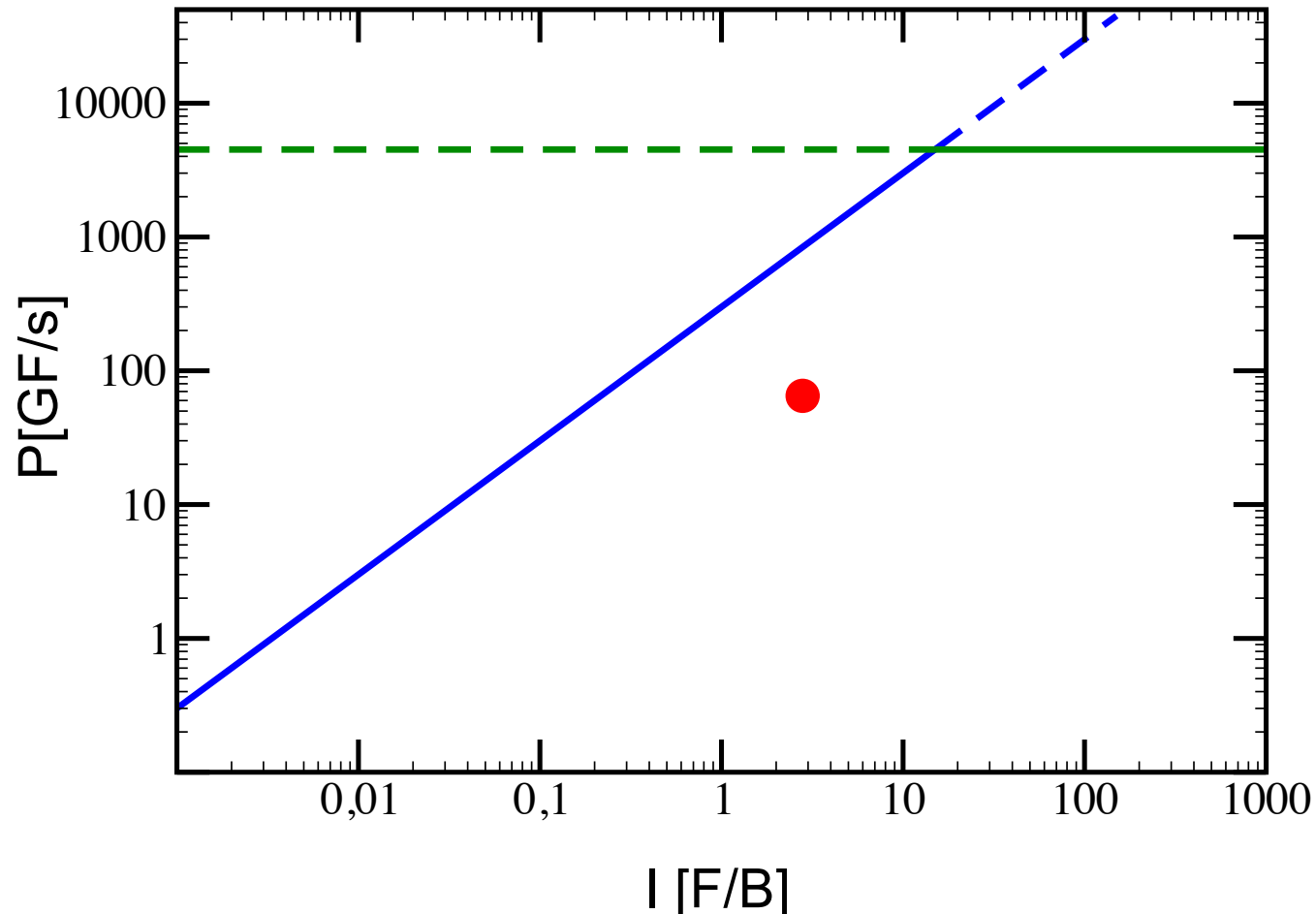
No SIMD: 1/8
AVX512 \rightarrow scalar

No FMA
pipelining: 1/4

Only 1 FMA: 1/2

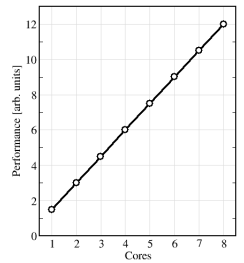
Lower horizontal roofs (P_{peak})

Reality: Lower horizontal roofs (P_{peak}) are typically not known

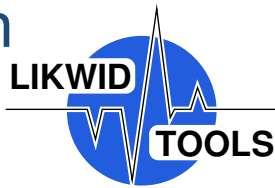


Indications:

- Linear scaling



- **Low** Memory Bandwidth Utilization

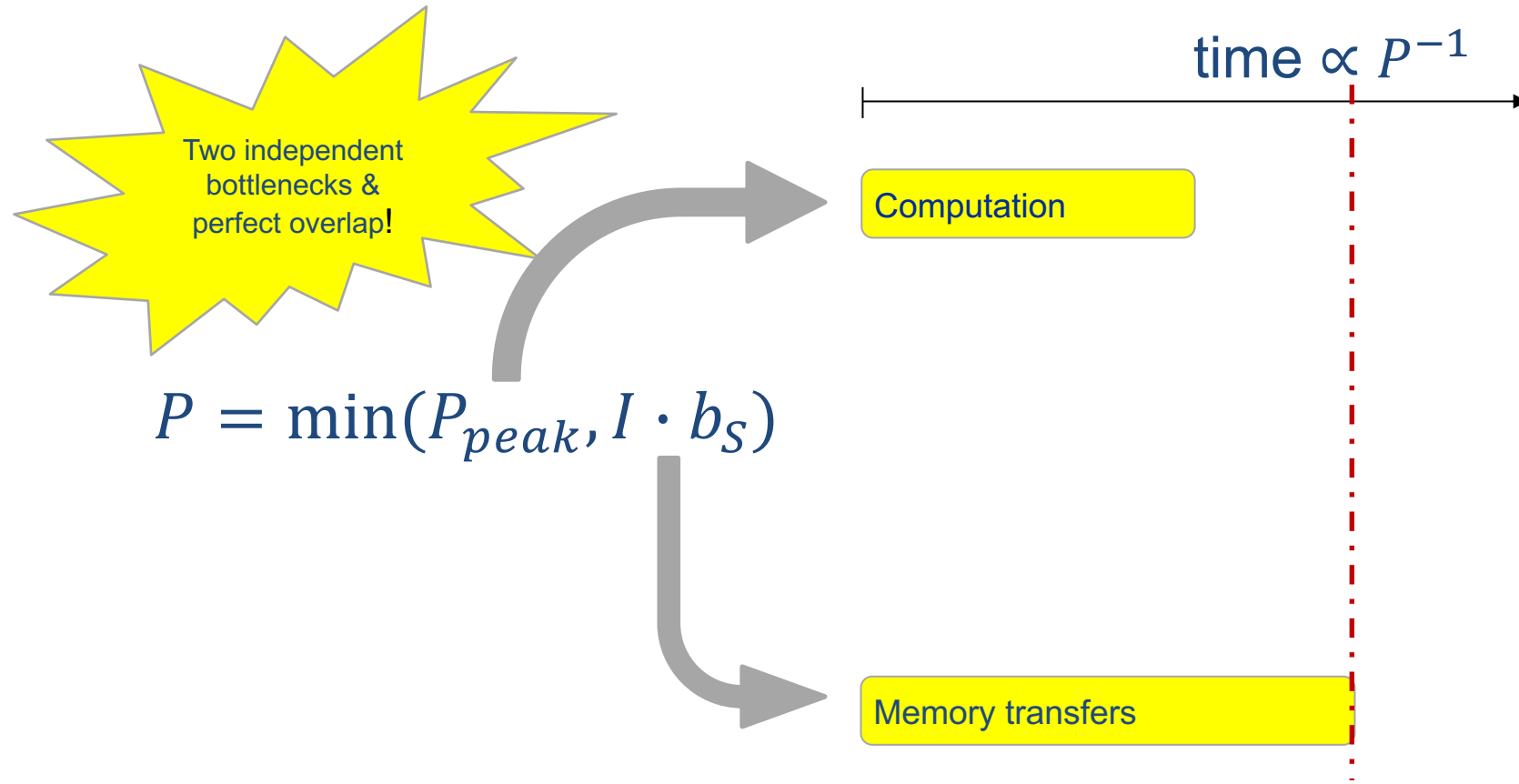


How to get realistic lower horizontal roofs?

The Roofline model: Extending more bottlenecks

Choose time based view:

Hardware bottlenecks impose upper (lower) performance (runtime) limits



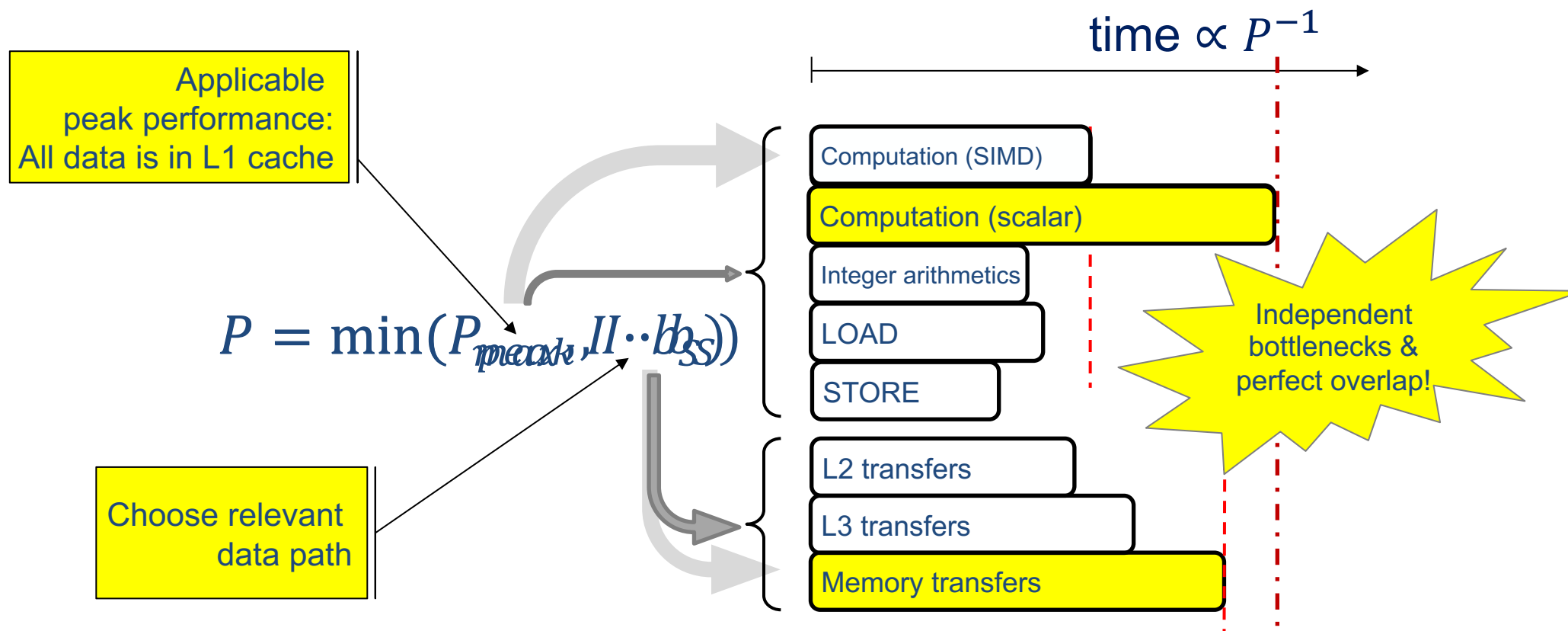
*Williams, Waterman, Patterson (2009), DOI: [10.1145/1498765.1498785](https://doi.org/10.1145/1498765.1498785)

Roofline Model (RLM) – Refined
Consider **multiple independent** bottlenecks



The Roofline model: Extending more bottlenecks

Extend towards multiple (independent) bottlenecks



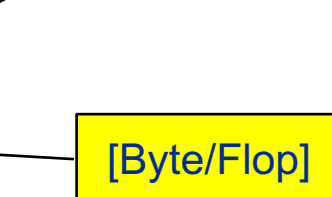
→ Model very successful if bottleneck can be saturated → full CPU chip

*Williams, Waterman, Patterson (2009), DOI: [10.1145/1498765.1498785](https://doi.org/10.1145/1498765.1498785)

The Roofline Model – refined

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

Expected performance:

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


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](https://doi.org/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)

The Roofline Model – getting it right

Applicable peak performance: $P_{max} = n_{core} * P_{max}^{core}$

- P_{max}^{core} : single core maximum performance from L1: determine according to slides 22-41@03b_05_13-2025_PTfS.pdf

Computational intensity: I

- Determine data transfer volume over slowest data path – for main memory:
 $I = 1/B_C^{mem}$ (for B_C see 05_05_20-2025_PTfS.pdf)

Applicable (saturated) peak bandwidth: b_S

- Determine with appropriate benchmark, e.g. for main memory choose the STREAM benchmark test that best matches your access pattern
 - See later for STREAM
 - Or write own microbenchmark if relevant access pattern not available, e.g. read-only
-

Realistic baseline for memory bandwidth: STREAM

- Assumption: STREAM (or similar, like vector triad) kernel benchmarks achieve an upper bandwidth limit from main memory
 - i.e., no code can draw more bandwidth
 - Theoretical BW limits are usually not achievable
 - Use STREAM as BW limit rather than the theoretical numbers!
- STREAM: <http://www.cs.virginia.edu/stream/>
 - Set of 4 standard benchmarks

COPY: $\mathbf{A}(:,) = \mathbf{C}(:,)$

SCALE: $\mathbf{A}(:,) = \mathbf{s} * \mathbf{C}(:,)$

ADD: $\mathbf{A}(:,) = \mathbf{B}(:,) + \mathbf{C}(:,)$

TRIAD: $\mathbf{A}(:,) = \mathbf{B}(:,) + \mathbf{s} * \mathbf{C}(:,)$

- In practice, COPY & SCALE (ADD & TRIAD) draw the same bandwidth
- Advantage of STREAM: Many results published, well-defined benchmark
- Disadvantage of STREAM: Reported and actual BW numbers may differ

STREAM: write-allocate and efficiency

Data transfer (including write-allocate)

Type	Kernel	Bytes/iteration assumed (with WA)	Flops/it.
COPY	$A(:) = B(:)$	16 (24)	0
SCALE	$A(:) = s*B(:)$	16 (24)	1
ADD	$A(:) = B(:)+C(:)$	24 (32)	1
TRIAD	$A(:) = B(:)+s*C(:)$	24 (32)	2

STREAM benchmark does not know about write-allocate

Type	with write-allocate			w/o write-allocate	
	reported	actual	b_S/b_{max}	reported	b_S/b_{max}
COPY	34079	$\xrightarrow{x3/2}$ 51119	0.75	47281	0.69
SCALE	33758	$\xrightarrow{x3/2}$ 50637	0.74	48025	0.70
ADD	38174	$\xrightarrow{x4/3}$ 50899	0.75	51068	0.75
TRIAD	38866	$\xrightarrow{x4/3}$ 51820	0.76	51107	0.75

2.3 GHz 14-core
Haswell (non-CoD)

State of the art compilers recognize the benchmark and avoid the write-allocate automatically

70-75% efficiency

Roofline Model (RLM) – Refined
Arithmetic Intensity / Code Balance: Gymnastics



Arithmetic Intensity / Code Balance: Basic Examples

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

$$B_C = 24B / 1F = 24 \text{ B/F}$$
$$I = 0.042 \text{ F/B}$$

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

$$B_C = 24B / 2F = 12 \text{ B/F}$$
$$I = 0.083 \text{ F/B}$$

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

Scalar – can be kept in register

$$B_C = 4B/2F = 2 \text{ B/F}$$
$$I = 0.5 \text{ F/B}$$

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

Scalar – can be kept in register

$$B_C = 8B / 2F = 4 \text{ B/F}$$
$$I = 0.25 \text{ F/B}$$

Scalar – can be kept in register

Approaches to determine Computational Intensity

1. Analysis of loop body → determine all load / stores that go to memory

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

- 3 LD (**b, c, d**) + 1 ST (**a**) + 1 WA (**a**) per iteration
 - Each LD / ST / WA is 8 Byte (**double**)
 - 2 FLOP
- $I = \frac{2 \text{ FLOP}}{5 * 8 \text{ Byte}} = \frac{1 \text{ FLOP}}{20 \text{ Byte}} \quad (B_C = \frac{20 \text{ Byte}}{1 \text{ FLOP}})$
- Cache vs. Memory Access??!! → DMVM; stencils, SpMV

2. Analysis of data structure → Assume each element is touched only once

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

- 4 arrays (of size: $N * 8 \text{ Byte}$) + WA on **a[]** → 2x
→ $5 * N * 8 \text{ Byte} = 40 * N \text{ Byte}$
- Total FLOP count: $2 * N \text{ FLOP}$
- $I = \frac{2 * N \text{ FLOP}}{40 * N \text{ Byte}} = \frac{1 \text{ FLOP}}{20 \text{ Byte}} \quad (B_C = \frac{20 \text{ Byte}}{1 \text{ FLOP}})$
- Lower bound for memory traffic → Upper bound for I

Approaches to determine Computational Intensity

```
double precision A(R,C), x(C), y(R)
...
do c = 1, C
  tmp=x(c)
  do r = 1, R
    y(r) = y(r) + A(r,c) * tmp
  enddo
enddo
```

Loop body analysis:

- LD A(r, c) to memory → 8 Byte
- x(c) ←→ register → 0 Byte
- LD/ST y(r) ←→ Cache → 0 Byte
→ 2 FLOP

$$\rightarrow I = \frac{2 \text{ FLOP}}{8 \text{ Byte}}$$

Data structure analysis:

- A(R,C) → 8 * R * C Byte
- X(C) → 8 * C Byte
- Y(R): LD/ST → 2*8 * R Byte
→ 2* R * C FLOP

$$\rightarrow I = \frac{2 * R * C \text{ FLOP}}{(R * C + C + 2 * R) * 8 \text{ Byte}} = \frac{2 \text{ FLOP}}{\left(1 + \frac{1}{R} + \frac{2}{C}\right) * 8 \text{ Byte}} \approx \frac{2 \text{ FLOP}}{8 \text{ Byte}}$$

$R, C \gg 1$

Roofline Model (RLM) – Refined Vector triads



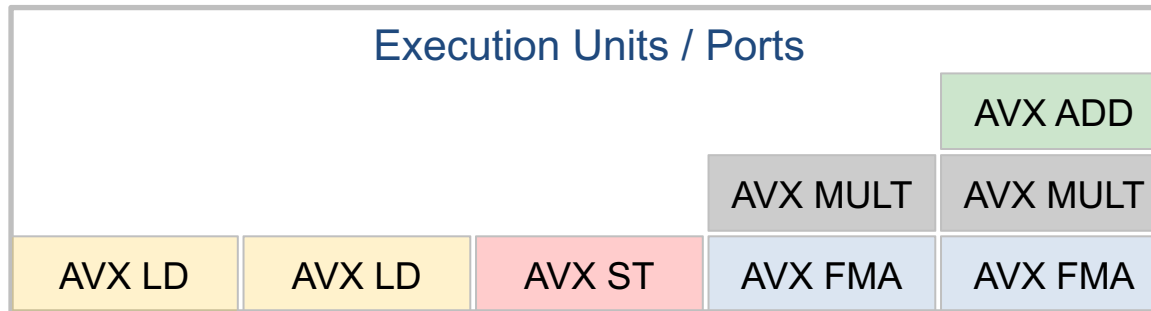
The Roofline Model – refined: Vector triads: P_{max}

- Machine: 7 cores of Haswell@2.3GHz ($n_{core} = 7; f = 2.3 \frac{Gcy}{s}$)

```
do i = 1,N
  A(i)=B(i)+C(i)*D(i)
enddo
```

For 1 AVX iteration (i:i+3)
3 AVX LDs + 1 AVX ST + 1 AVX FMA

AVX performance on 1 core Haswell / Broadwell



2 AVX iterations

- Bottleneck: LD
- 2 AVX iteration: $T_{max}^{inst} = 3cy$
- 2 AVX iteration \rightarrow 8 loop iterations \rightarrow 16 F

$$P_{max}^{core} = \frac{16F}{3cy} = 5.33 \frac{F}{cy}$$

The Roofline Model – refined: Vector triads: $I \cdot b_S$ & P

- Machine: 7 cores of Haswell @2.3 (CoD)

- STREAM triads BW: $b_S = 29 \frac{GB}{s}$

- Computational Intensity (incl. WA; double precision): $I = \frac{2F}{5 \cdot 8B} = 0.05 \frac{F}{B}$

```
do i = 1,N
  A(i) = B(i) + C(i) * D(i)
enddo
```

Putting it together

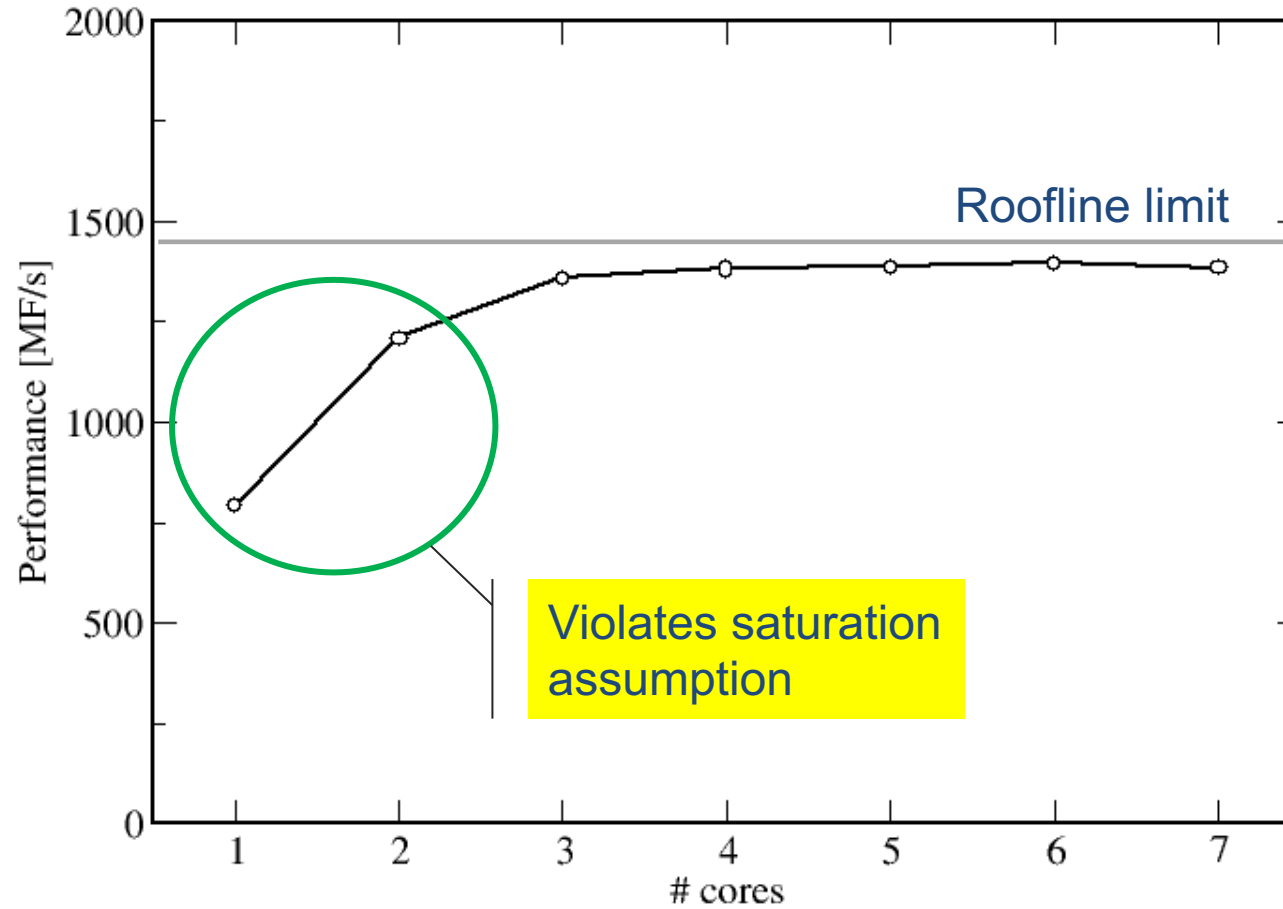
See
previous slide

$$P_{max} = 7 * 2.3 \frac{Gcy}{s} * 5.33 \frac{F}{cy} = 85.8 \frac{GF}{s}$$

$$P = \min \left(85.8 \frac{GF}{s}, 0.05 \frac{F}{B} * 29 \frac{GB}{s} \right) = \min \left(85.8 \frac{GF}{s}, 1.45 \frac{GF}{s} \right) = 1.45 \frac{GF}{s}$$

The Roofline Model – refined: Validate RLM

7 cores of Haswell @2.3 GHz (CoD)



Roofline Model (RLM) – Refined Dense Matrix Vector Multiplication



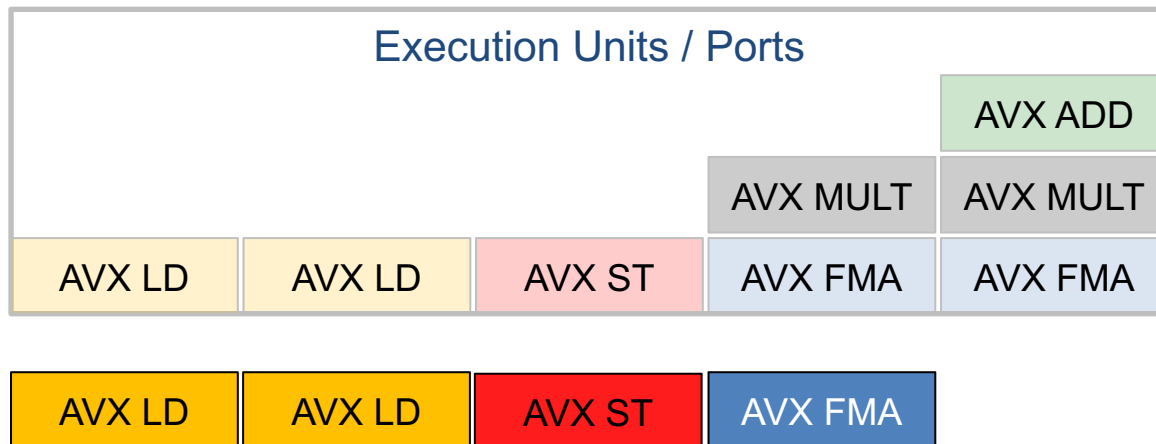
The Roofline Model – refined: Dense MVM : P_{max}

- Machine: 7 cores of Haswell @2.3GHz ($n_{core} = 7; f = 2.3 \frac{Gcy}{s}$)

```
do c = 1 , C
  tmp=x(c)
  do r = 1 , R
    y(r)=y(r) + A(r,c) * tmp
  enddo
enddo
```

For 1 AVX iteration (r:r+3)
2 AVX LDs + 1 AVX ST + 1 AVX FMA

AVX performance on 1 core Haswell / Broadwell



1 AVX iteration

- Bottleneck: LD
- 1 AVX iteration: $T_{max}^{inst} = 1cy$
- 1 AVX iteration \rightarrow 4 loop iterations \rightarrow 8 F

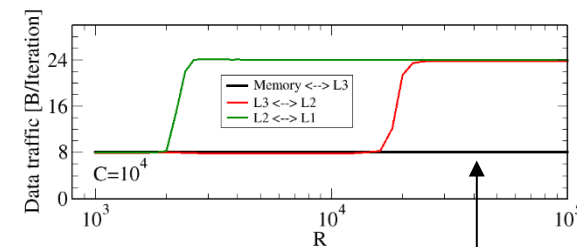
$$P_{max}^{core} = 8F / 1cy = 8 F / cy$$

The Roofline Model – refined: Dense MVM: $I \cdot b_S$ & P

- Machine: 7 cores of Haswell@2.3 GHz

- Read-OnlyBW: $b_S = 32 \frac{GB}{s}$

- Computational Intensity (double precision): $I = 1/B_C^{mem} = \frac{1}{\frac{8B}{2F}} = 0.25 \frac{F}{B}$



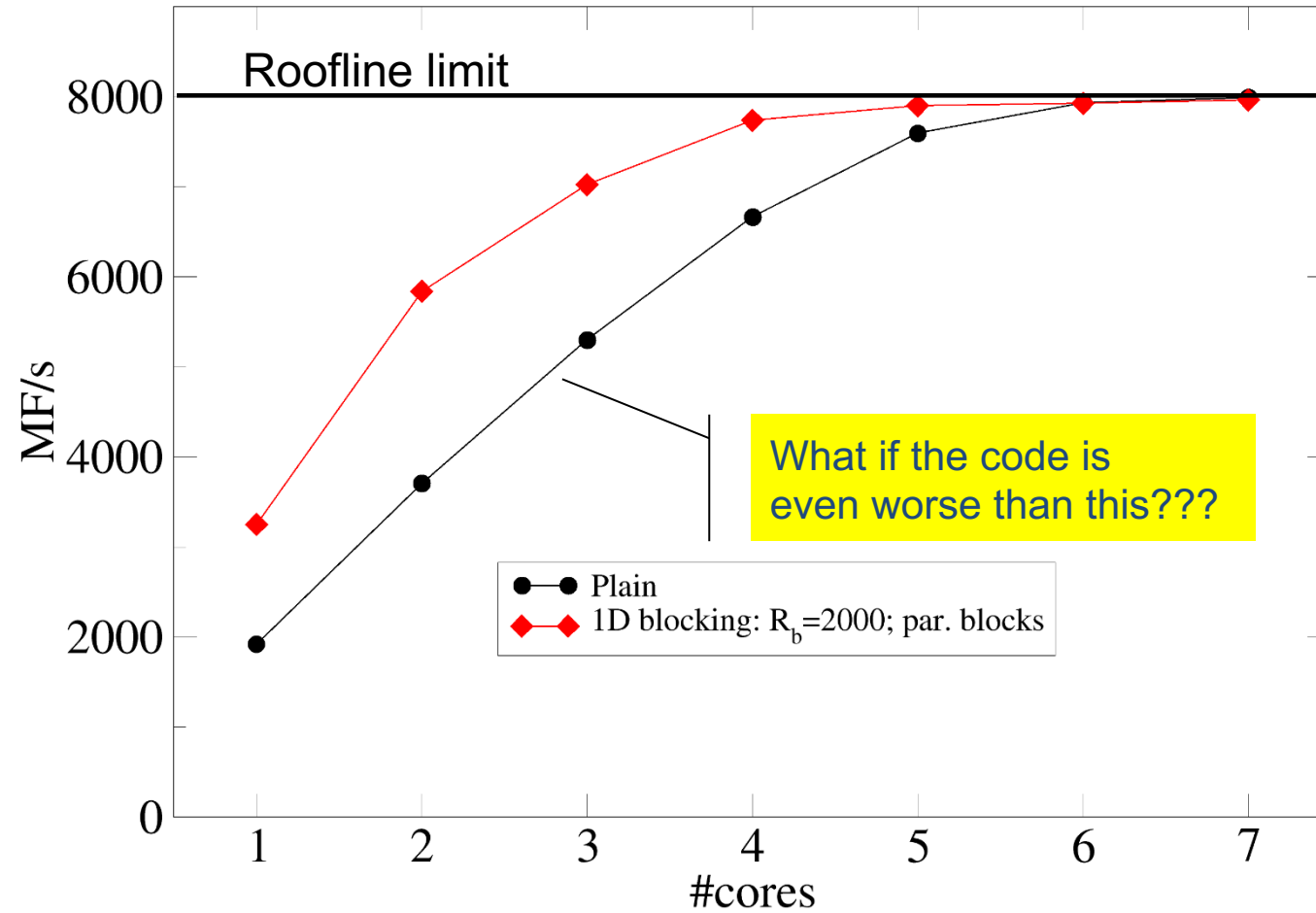
Putting it together

$$P_{max} = 7 * 2.3 \frac{Gcy}{s} * 8 \frac{F}{cy} = 128.8 \frac{GF}{s}$$

See
previous slide

$$P = \min \left(128.8 \frac{GF}{s}, 0.25 \frac{F}{B} * 32 \frac{GB}{s} \right) = \min \left(128.8 \frac{GF}{s}, 8 \frac{GF}{s} \right) = 8 \frac{GF}{s}$$

The Roofline Model – refined: Dense MVM: Validate



Roofline Model (RLM) – Refined Bad Code Implementation & Lower roofs

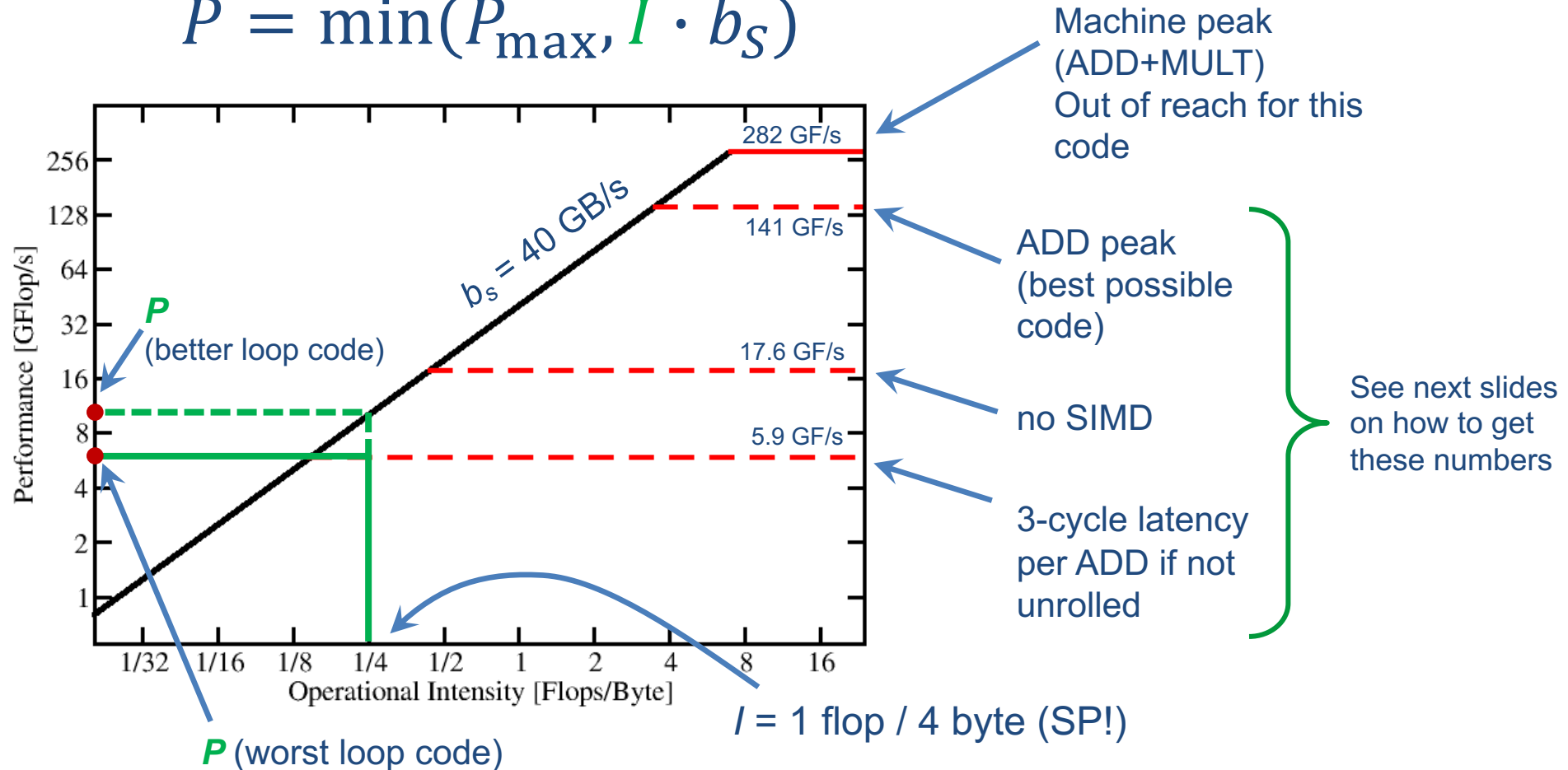


A not so simple Roofline example

Example: `do i=1,N; s=s+a(i); enddo`

in single precision on a 2.2 GHz Sandy Bridge (3-stage FP add pipeline) socket @ "large" N

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

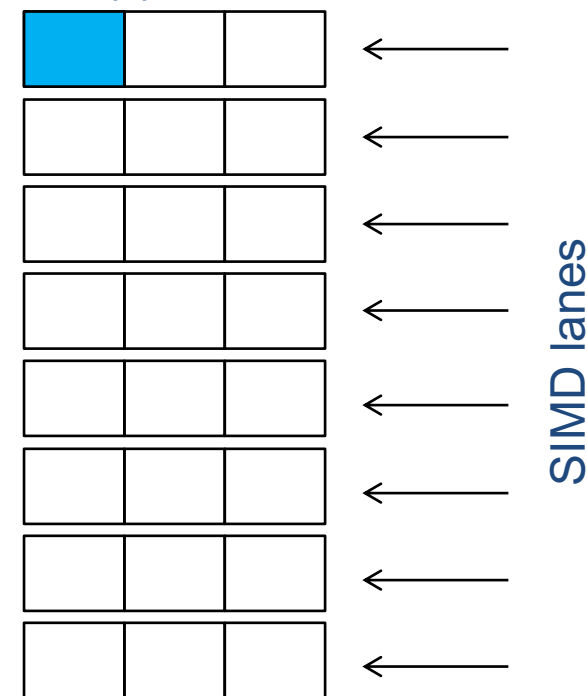


Applicable peak for the summation loop

Plain scalar code, no SIMD

```
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 summation loop

Scalar code, 3-way unrolling

```
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
```

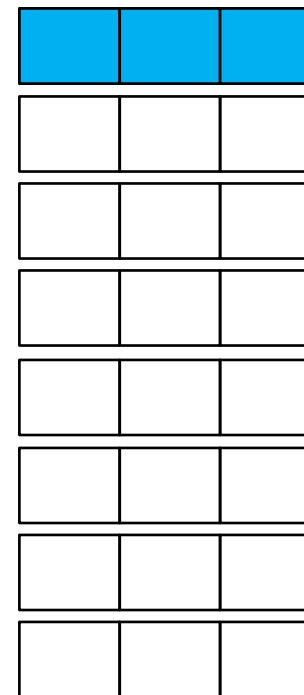
```
ADD r2.0 ← r2.0 + r5.0
```

```
ADD r3.0 ← r3.0 + r6.0
```

```
i+=3 →? loop
```

```
result ← r1.0+r2.0+r3.0
```

ADD pipes utilization:



→ 1/8 of ADD peak

Applicable peak for the summation loop

SIMD-vectorized, 3-way unrolled

```
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)]
```

```
LOAD [r5.0,...,r5.7] ← [a(i+8),...,a(i+15)]
```

```
LOAD [r6.0,...,r6.7] ← [a(i+16),...,a(i+23)]
```

```
ADD r1 ← r1 + r4
```

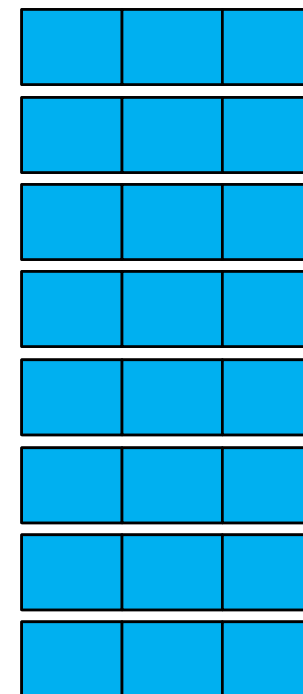
```
ADD r2 ← r2 + r5
```

```
ADD r3 ← r3 + r6
```

```
i+=24 →? loop
```

```
result ← r1.0+r1.1+...+r3.6+r3.7
```

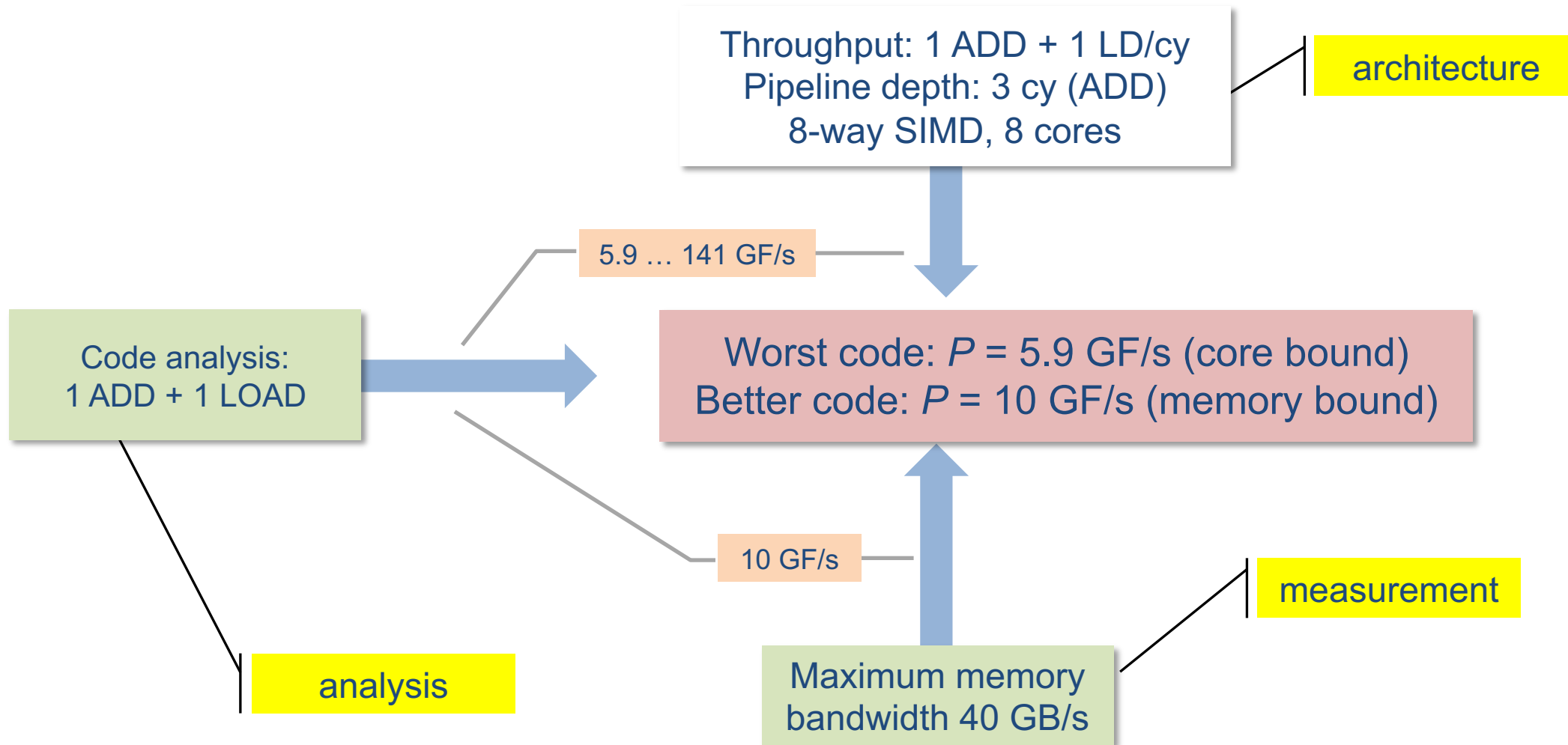
ADD pipes utilization:



→ ADD peak

Input to the roofline model

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



Roofline Model (RLM) – Refined Summary



Assumptions made by the Roofline Model

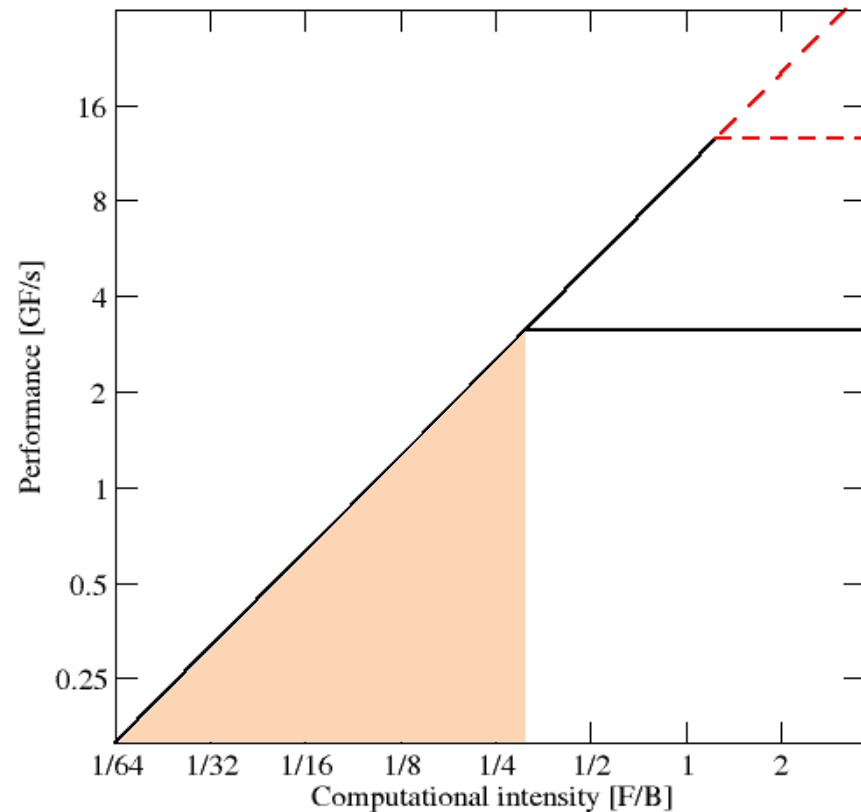
- **Data transfer and core execution overlap perfectly!**
 - Either the limit is core execution or it is data transfer
- **Slowest limiting factor “wins”**; all others are assumed to have no impact
 - If two bottlenecks are “close”, no interaction is assumed
- Data access latency is ignored, i.e. **perfect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



Factors to consider in the roofline model

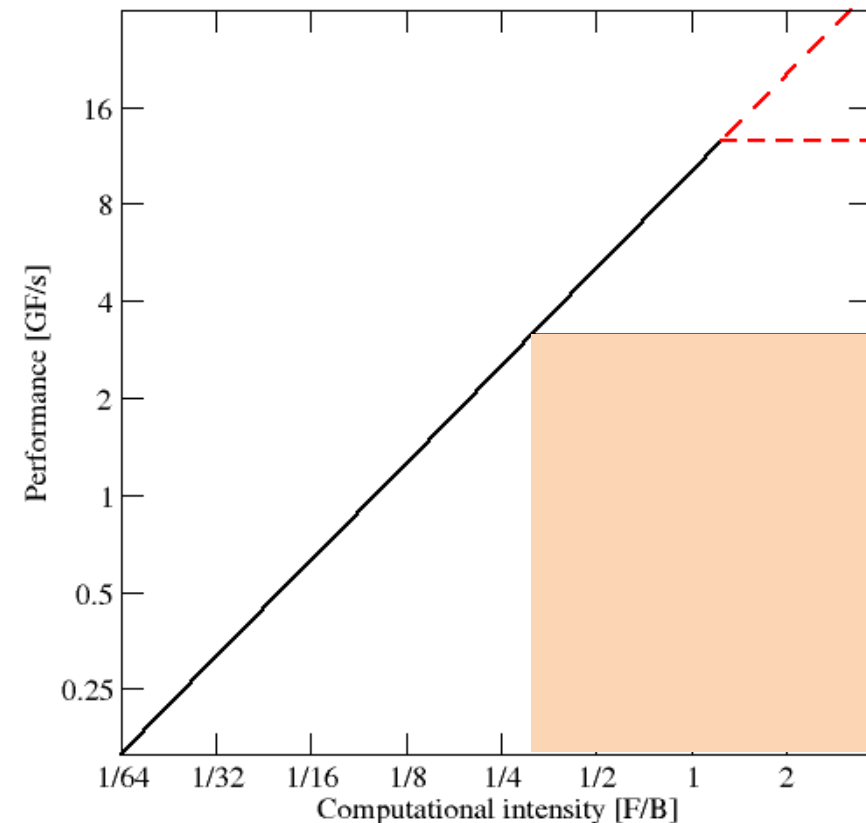
Bandwidth-bound (simple case)

- Accurate traffic calculation (write-allocate, strided access, ...) → Intensity calculation
- Attainable \neq theoretical BW
- Erratic access patterns may violate model assumptions



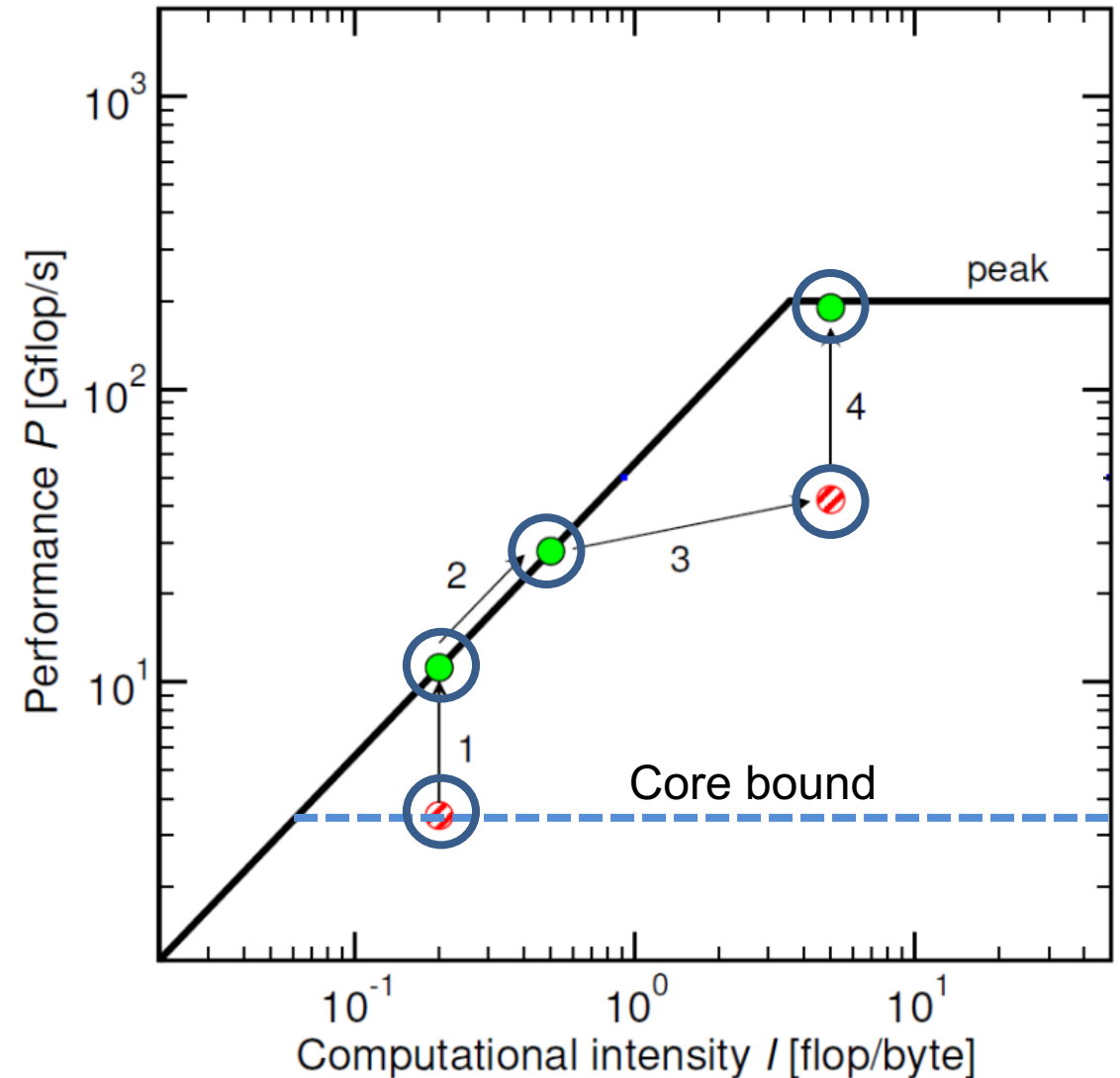
Core-bound (may be complex)

- Multiple bottlenecks: LD/ST, arithmetic, pipelines, SIMD, execution ports
- Limit is linear in # of cores (or clock speed)



Tracking code optimizations in the Roofline Model

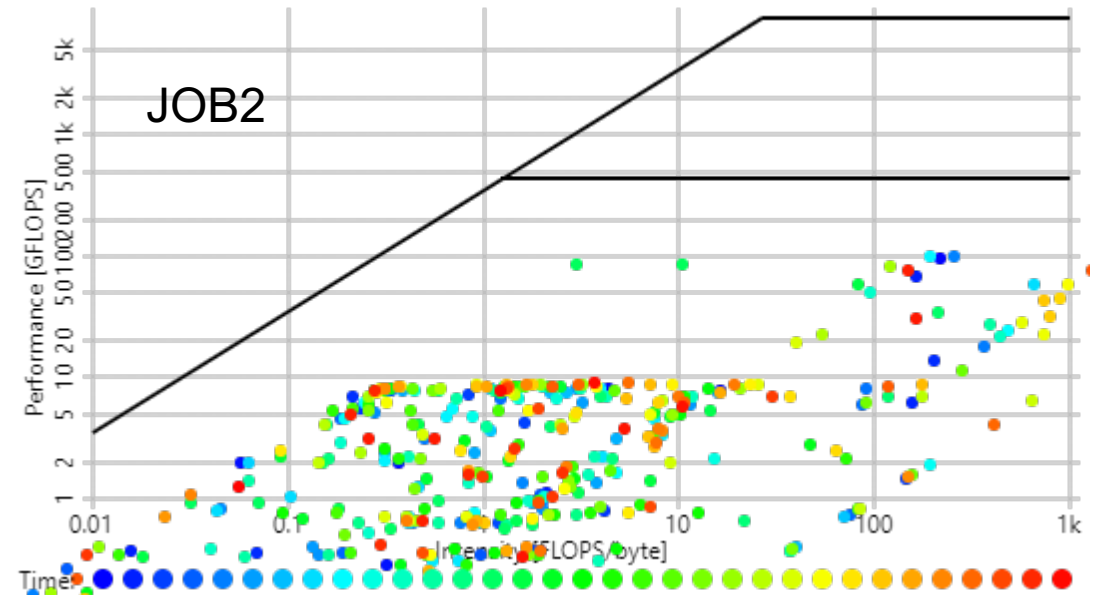
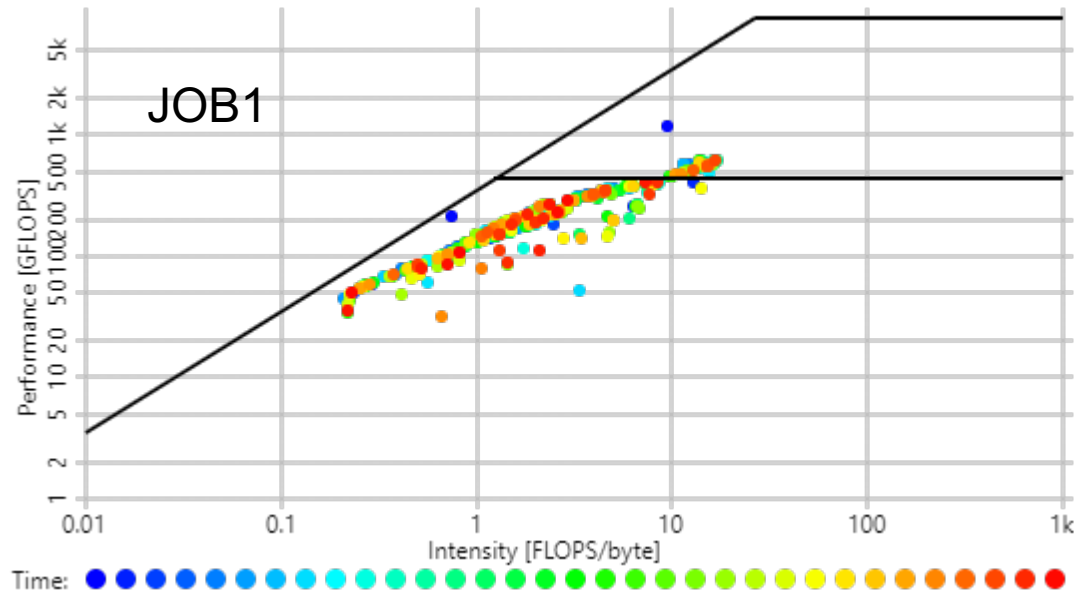
1. Hit the BW bottleneck by good serial code
(e.g., Ninja C++ → Fortran)
2. Increase intensity to make better use of BW bottleneck
(e.g., spatial loop blocking)
3. Increase intensity and go from memory bound to core bound
(e.g., temporal blocking)
4. Hit the core bottleneck by good serial code
(e.g., `-fno-alias`, SIMD intrinsics)



Monitoring jobs running on Fritz in the Roofline diagram

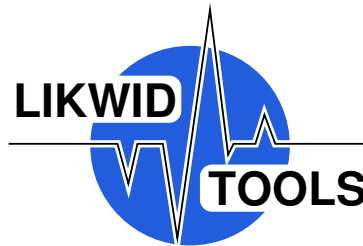
Two cluster jobs...

$$\text{Rooflines: } P = \min (P_{\text{peak}}, I * b_s)$$



ClusterCockpit 

<https://github.com/ClusterCockpit>



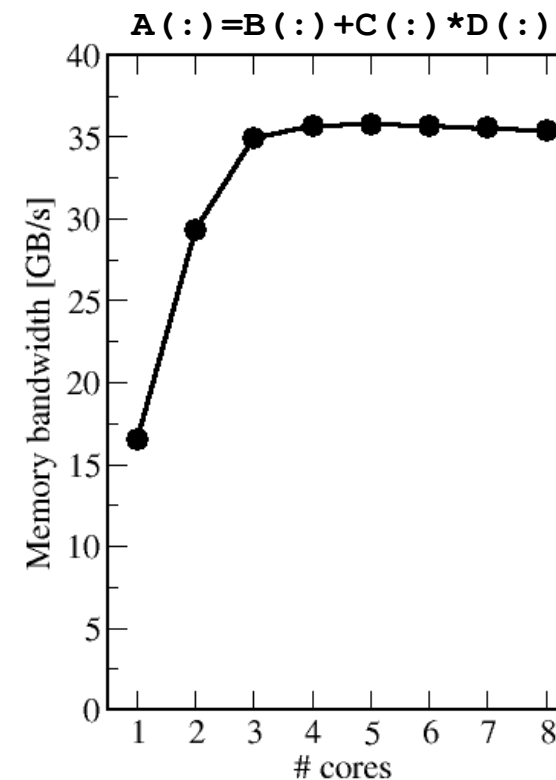
- LIKWID determines P and I regularly on each node
- ClusterCockpit collects data and presents it

Which is the "good" and the "bad" job?



Shortcomings of the roofline model

- **Saturation effects** in multicore chips are not explained
 - Reason: **Intra-Cache** and **memory transfers** do (frequently) **not overlap** on a **single core**
→ Overlapp only between cores
 - Increase “pressure” on memory interface until it saturates → bottleneck: b_s
 - It is not sufficient to measure single-core STREAM to make it work
- **In-cache performance** is not correctly predicted
- The **ECM performance model** gives more insight:



Roofline Model (RLM) – Refined
Code Balance and Machine Balance



Machine balance for hardware characterization

- For quick comparisons the concept of **machine balance** is useful

$$B_m = \frac{b_s}{P_{\text{peak}}}$$

- Machine Balance** = How much input data can be delivered for each FP operation? (“Memory Gap characterization”)
 - Assuming balanced MULT/ADD
- Rough estimate: $B_m \ll B_c \rightarrow$ **strongly memory-bound code**
- Typical values (main memory):

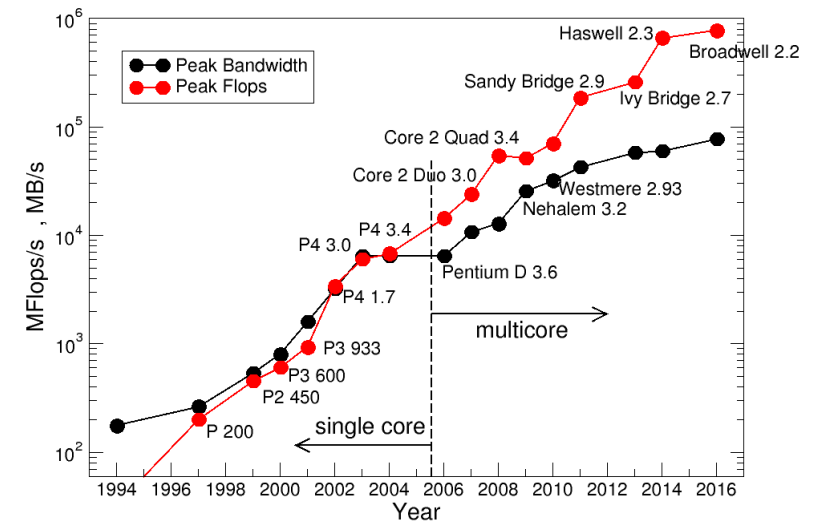
Intel Haswell 14-core 2.3 GHz

$$B_m = 60 \text{ GB/s} / (14 \times 2.3 \times 16) \text{ GF/s} \approx 0.12 \text{ B/F}$$

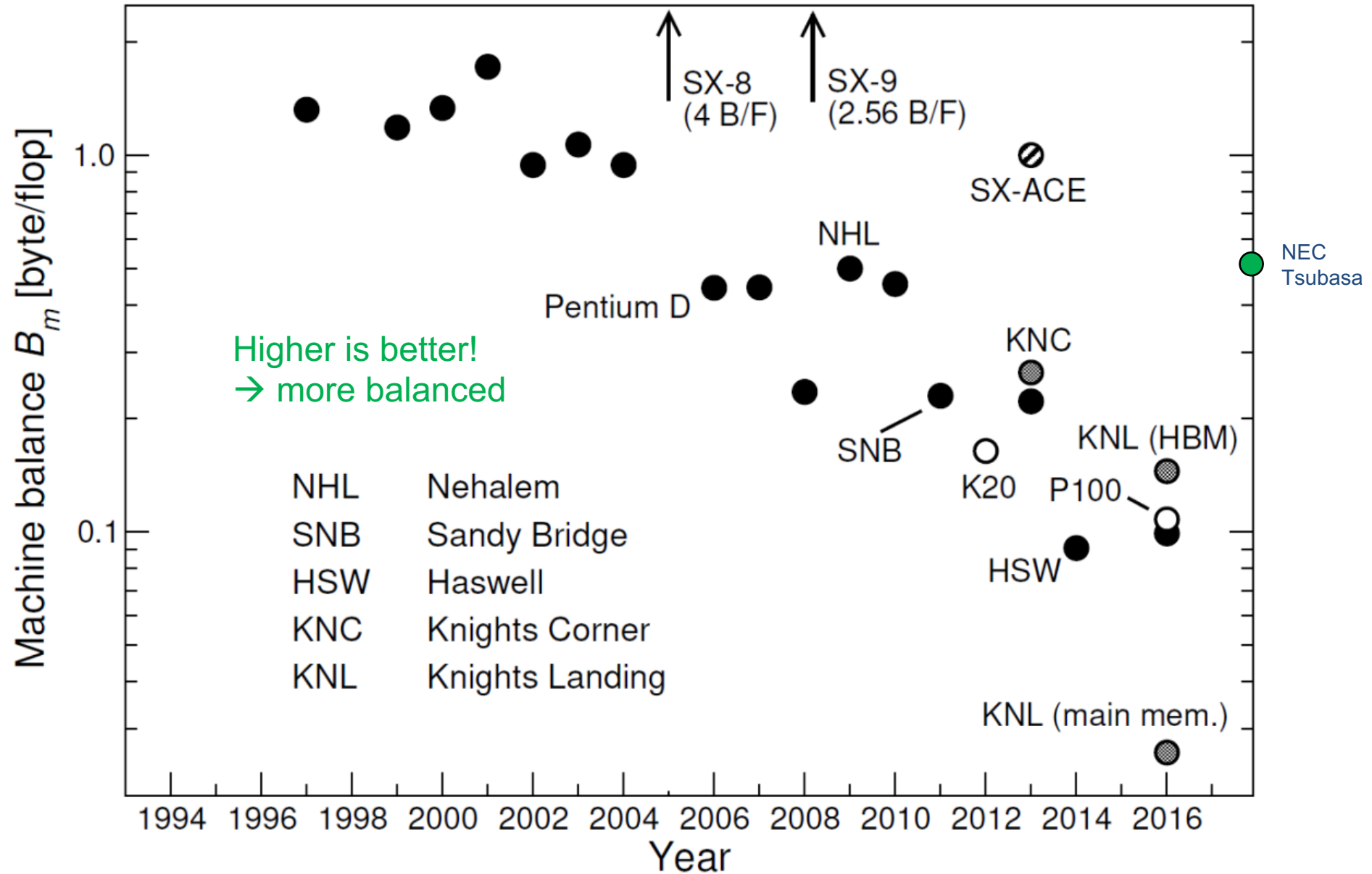
Intel Sandy Bridge 8-core 2.7 GHz $\approx 0.23 \text{ B/F}$

Nvidia P100 $\approx 0.10 \text{ B/F}$

Intel Xeon Phi Knights Landing (HBM) $\approx 0.16 \text{ B/F}$



Machine balance over time



RLM Case Study

Tall & Skinny Matrix-Transpose Times
Tall & Skinny Matrix (TSM)TTSM)
Multiplication



TSMTTSM Multiplication

- Block of vectors \rightarrow Tall & Skinny Matrix (e.g. $10^7 \times 10^1$ dense matrix)
- Row-major storage format
- Block vector subspace orthogonalization procedure requires, e.g. computation of scalar product between vectors of two blocks

- TSMTTSM Multiplication $K \gg N, M$

Assume: $\alpha = 1; \beta = 0$

$$C = \alpha A^T * B + \beta C$$

TSMTTSM Multiplication

- General rule for dense matrix-matrix multiply: Use vendor-optimized GEMM, e.g. from Intel MKL¹:

$$C_{mn} = \sum_{k=1}^K A_{mk} B_{kn}, \quad m = 1..M, n = 1..N$$

System	P _{peak} [GF/s]	b _s [GB/s]	Size	Perf.	Efficiency
Intel Xeon E5 2660 v2 10c@2.2 GHz	176 GF/s	52 GB/s	SQ	160 GF/s	91%
			TS	16.6 GF/s	6%
Intel Xeon E5 2697 v3 14c@2.6GHz	582 GF/s	65 GB/s	SQ	550 GF/s	95%
			TS	22.8 GF/s	4%

double

complex double

TS@MKL:
Good or bad?

Matrix sizes:

Square (SQ): M=N= K=15,000

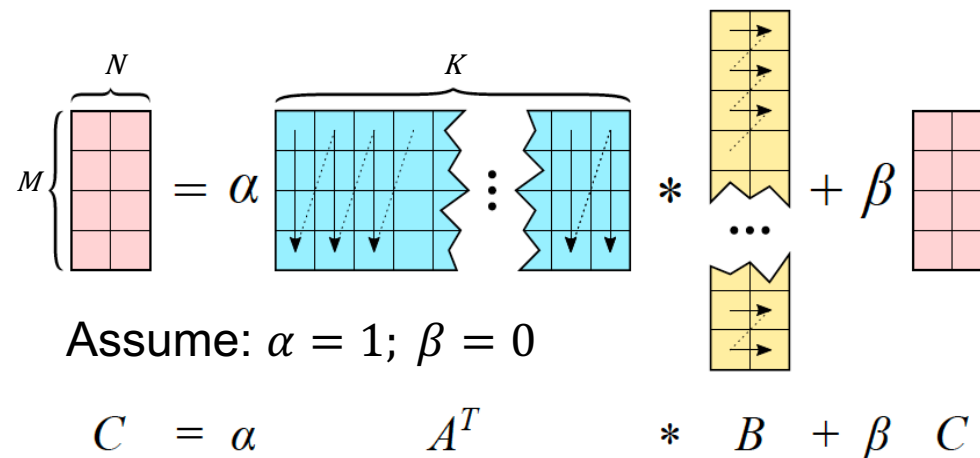
Tall&Skinny (TS): M=N=16 ; K=10,000,000

¹Intel Math Kernel Library (MKL) 11.3

TSMTTSM Roofline model

Computational intensity

$$I = \frac{\text{\#flops}}{\text{\#bytes (slowest data path)}}$$



Optimistic model (minimum data transfer) assuming $M = N \ll K$ and double precision:

$$I_d = \frac{2KMN}{8(2MN + KM + KN)} \frac{F}{B} \approx \frac{2MN}{8(M + N)} \frac{F}{B} = \frac{MF}{8B}$$

complex double:

$$I_z = \frac{8KMN}{16(2MN + KM + KN)} \frac{F}{B} \approx \frac{8MN}{16(M + N)} \frac{F}{B} = \frac{MF}{4B}$$

N = M

TSMTTSM Roofline performance prediction

Now choose $M = N = 16 \rightarrow I_d \approx \frac{16 \text{ F}}{8 \text{ B}}$ and $I_z \approx \frac{16 \text{ F}}{4 \text{ B}}$, i.e. $B_d \approx 0.5 \frac{\text{B}}{\text{F}}$, $B_z \approx 0.25 \frac{\text{B}}{\text{F}}$

Intel Xeon E5 2660 v2 ($b_s = 52 \frac{\text{GB}}{\text{s}}$) $\rightarrow P = I_d \times b_s = 104 \frac{\text{GF}}{\text{s}}$ (double)

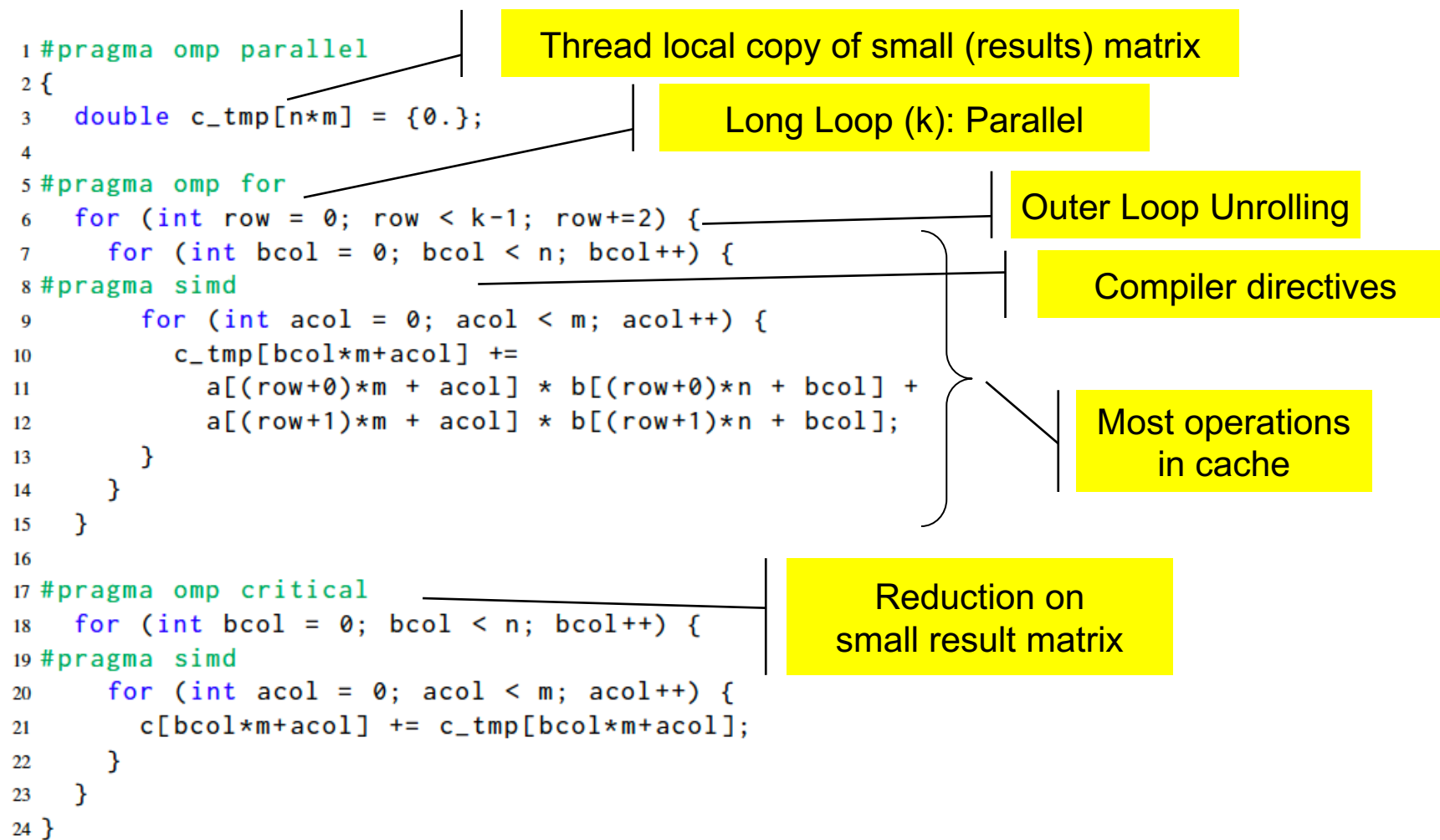
Measured (MKL): $16.6 \frac{\text{GF}}{\text{s}}$

Intel Xeon E5 2697 v3 ($b_s = 65 \frac{\text{GB}}{\text{s}}$) $\rightarrow P = I_z \times b_s = 240 \frac{\text{GF}}{\text{s}}$ (double complex)

Measured (MKL): $22.8 \frac{\text{GF}}{\text{s}}$

\rightarrow Potential speedup: 6–10x vs. MKL

Can we implement a better TSMTTSM kernel than Intel?



Not shown: Inner Loop boundaries (n,m) known at compile time (kernel generation)
k assumed to be even

TSMTTSM MKL vs. “hand crafted” (OPT)

TS: M=N=16 ; K=10,000,000

System	P _{peak} / b _S	Version	Performance	RLM Efficiency
Intel Xeon E5 2660 v2 10c@2.2 GHz	176 GF/s 52 GB/s	TS OPT	98 GF/s	94 %
		TS MKL	16.6 GF/s	16 %
Intel Xeon E5 2697 v3 14c@2.6GHz	582 GF/s 65 GB/s	TS OPT	159 GF/s	66 %
		TS MKL	22.8 GF/s	9.5 %