ERLANGEN REGIONAL COMPUTING CENTER

Architecture Specific Optimization Techniques

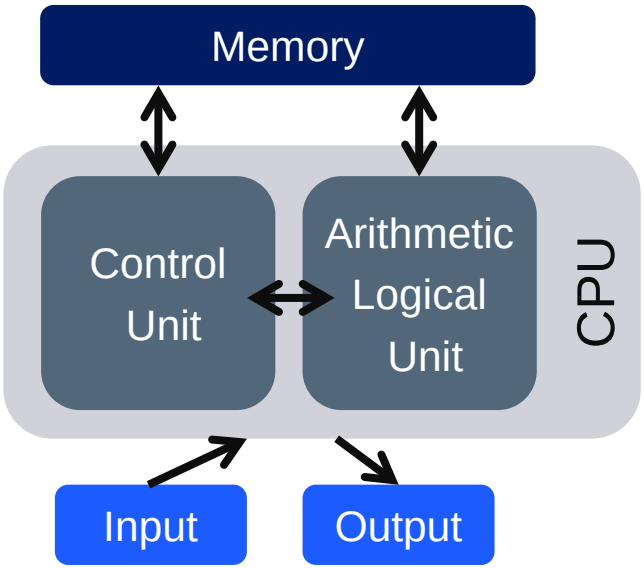
J. Eitzinger

PATC LRZ 2018, 26.3.2018

Schedule

Monday	Topic
9:00-10:30	Introduction to Computer architecture
10:30-10:45	Coffee Break
10:45-11:45	Node Topology and Performance Tools
11:45-12:30	Exercise 1: Stream Benchmark
12:30-13:30	Lunch Break
13:30-14:30	Basics of Performance Engineering
14:30-15:30	Exercise 2: In-cache triad
15:45-16:00	Coffee Break
16:00-17:00	Performance Modelling

Stored Program Computer: Base setting




```
for (int j=0; j<size; j++){
    sum = sum + V[j];
}
```

```
401d08:   f3 0f 58 04 82        addss  xmm0,[rdx + rax * 4]
401d0d:   48 83 c0 01           add    rax,1
401d11:   39 c7                 cmp    edi,eax
401d13:   77 f3                 ja     401d08
```

Architect's view:
Make the common case fast !

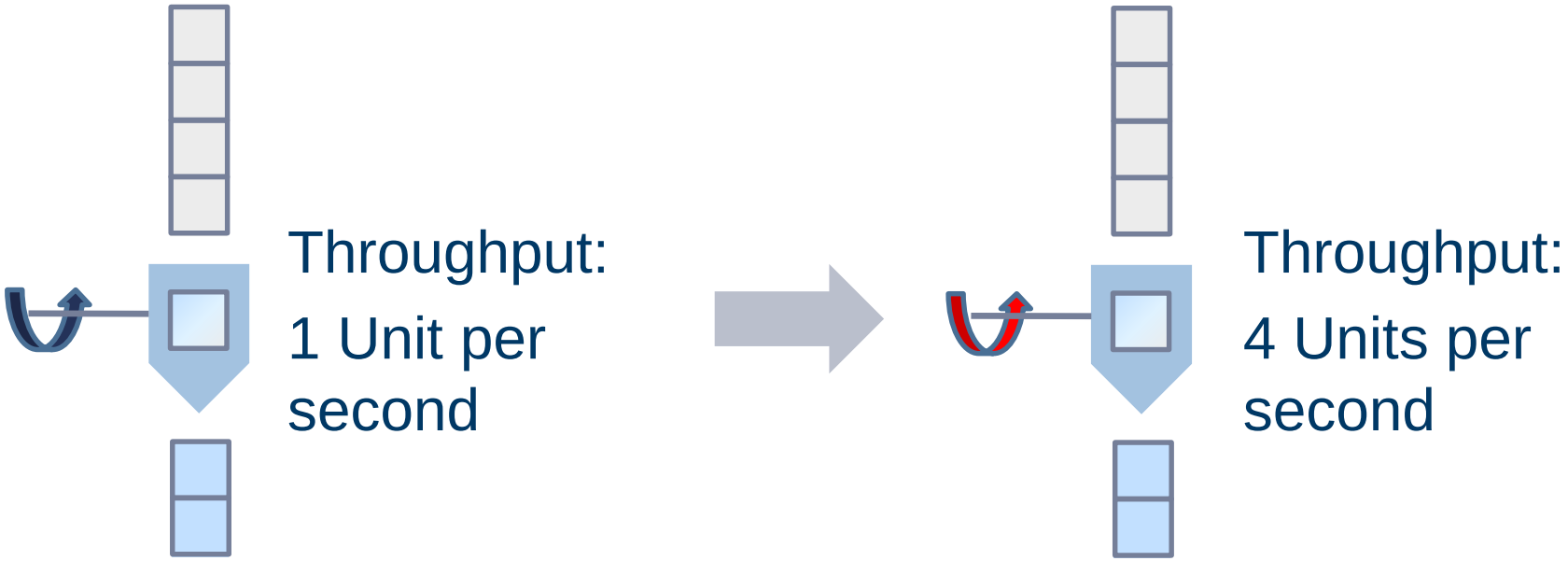
- Improvements for **relevant** software
- What are the **technical** opportunities?
- **Economical** concerns
- **Marketing** concerns

Execution and memory

Strategies

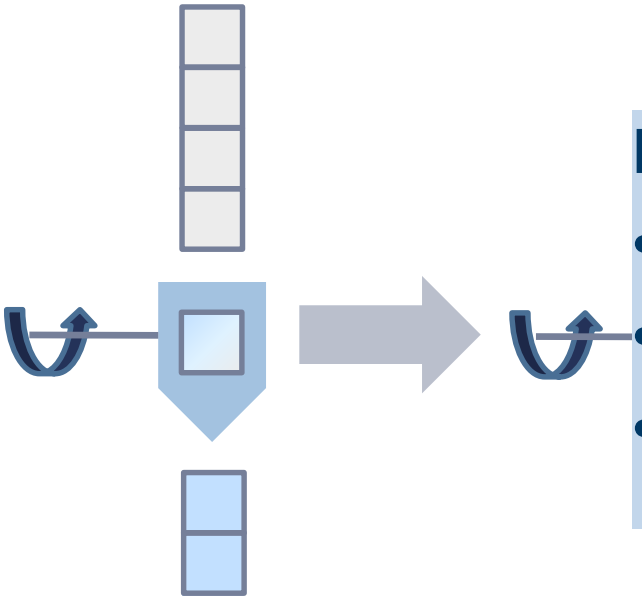
- Increase clock speed
- Parallelism
- Specialization

Performance increase by clock increase



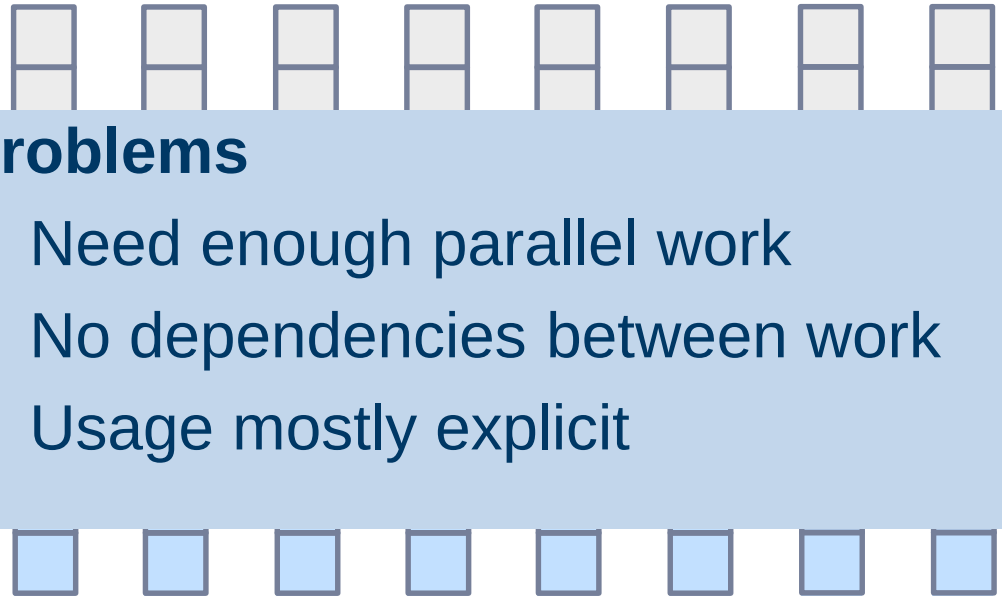
Limit: Physical limitations for cooling!

Performance increase by parallelization



Problems

- Need enough parallel work
- No dependencies between work
- Usage mostly explicit



Throughput:

1 Unit per second

Throughput:

8 Unit per second

Excursion in memory bandwidth

Some thoughts on efficiency …

Common lore: *Efficiency is the fraction of peak performance you reach!*

Example: STREAM triad (A(:)= B(:)+C(:)*d) with data not fitting into cache.

Intel Xeon X5482 (Harpertown 3.2 GHz): 553 Mflops/s (8 cores)

Efficiency 0.54% of peak

Intel Xeon E5-2680 (SandyBridge EP 2.7 GHz) 4357 Mflops/s (16 cores)

Efficiency 1.2% of peak

What can we do about it? **Nothing!**

Excursion in memory bandwidth

A better way to think about efficiency

Reality: This code is bound by main memory bandwidth.

HPT 6.6 GB/s (8.8 GB/s with WA)

SNB 52.3 GB/s (69.6 GB/s with WA)

Efficiency increase: None !

Architecture improvement: 8x

In both cases this is near 100% of achievable memory bandwidth.

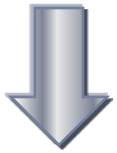
To think about efficiency you should focus on the utilization of the relevant resource!

Hardware-Software Co-Design? From algorithm to execution

Notions of work:

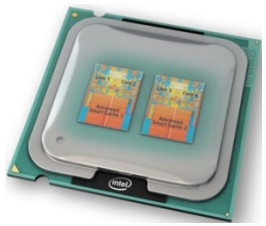
- Application Work
  - Flops
  - LUPS
  - VUPS
- Processor Work
  - Instructions
  - Data Volume

Algorithm



Programming language

Machine code



Example: Threaded vector triad in C

Consider the following code:

```
#pragma omp parallel private(j)
{
for (int j=0; j<niter; j++) {
#pragma omp for
   for (int i=0; i<size; i++) {
      a[i] = b[i] + c[i] * d[i];
   } /* global synchronization */
}
}
```

Setup:

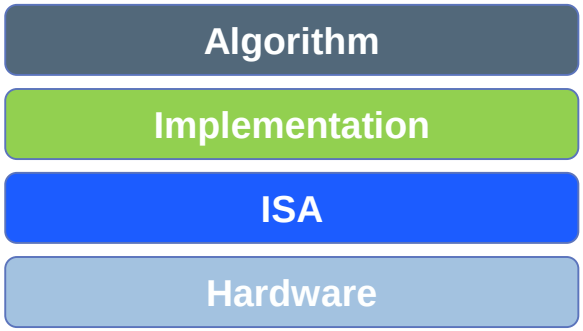
32 threads running on a dual socket 8-core SandyBridge-EP

gcc 4.7.0

Every single synchronization in this setup costs in the order of **60000 cycles** !

Why are we doing this?

Abstraction concept

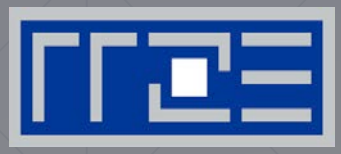


Pragmatic solution:

- Optimizing libraries: Proven working solution
- Application code consolidation: Just a few community codes in every application class

- Stage 1: It just works! (aka: The compiler will fix it)
- Stage 2: We need new programming models!
- Stage 3: You need to modernize your code. (aka: It is your fault)

HARDWARE OPTIMIZATIONS FOR SINGLE-CORE EXECUTION



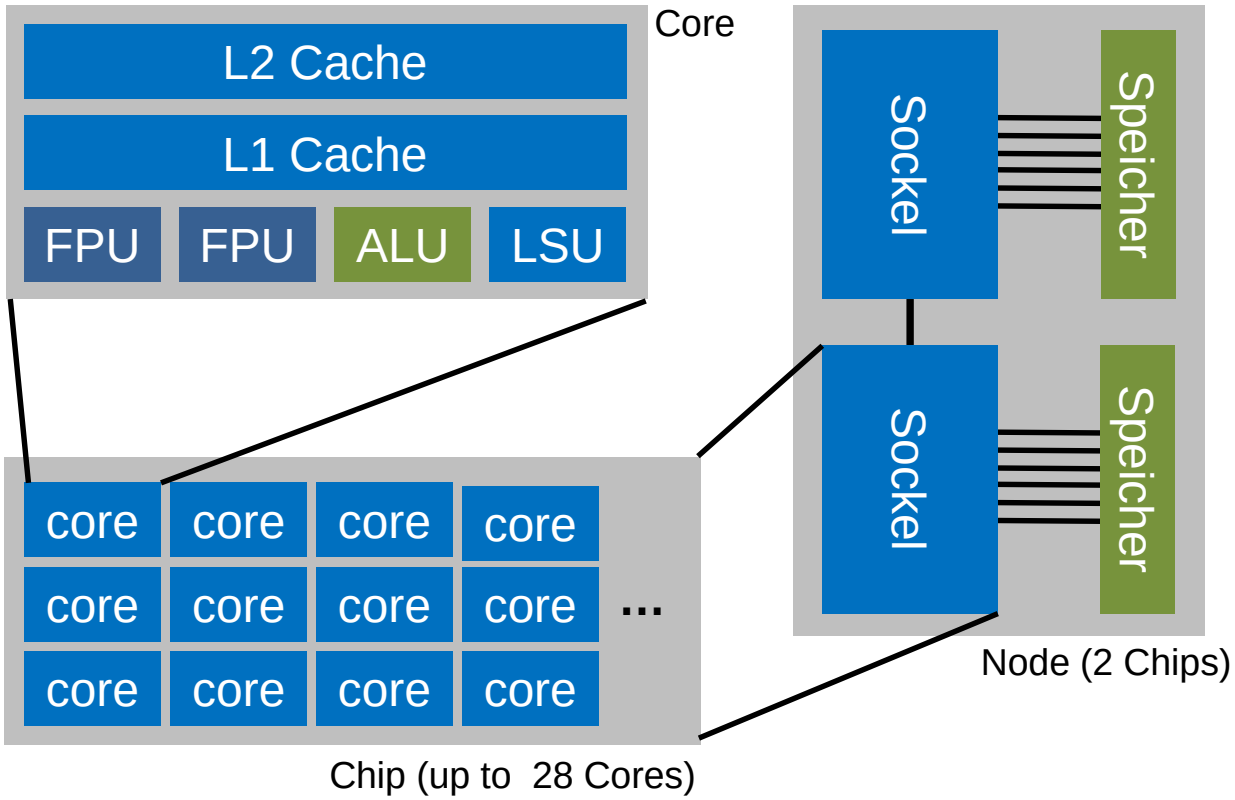
- ILP
- SIMD
- SMT
- Memory hierarchy

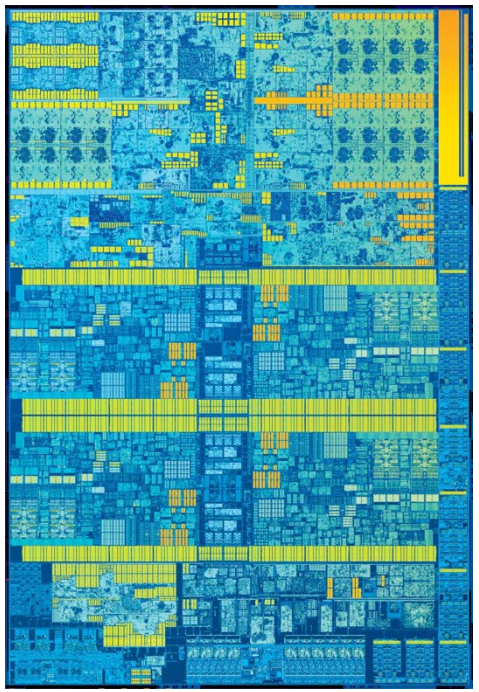
Common technologies

- Instruction Level Parallelism (**ILP**)
  - Instruction pipelining
  - Superscalar execution
  - Out-of-order execution
- Memory Hierarchy
- Branch Prediction Unit, Hardware Prefetching
- Single Instruction Multiple Data (**SIMD**)
- Simultaneous Multithreading (**SMT**)

Cycle
Pipeline latency
Stages
Wind-up
Bubbles
Wind-down
CPI
Scheduler
Hazard
Caches
Write allocate
Temporal locality
Cache-line
Speculative execution
Lanes
Register width
Packed
Scalar

Multicore architectures

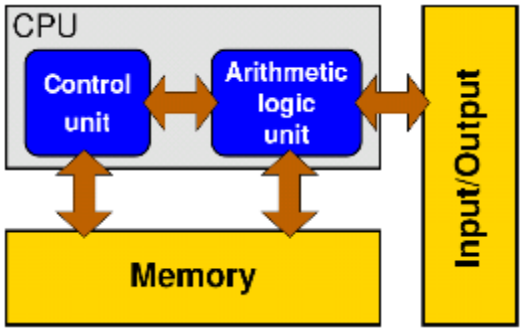




Ca. 8 Mrd. transistors in 500 mm^2

© Intel

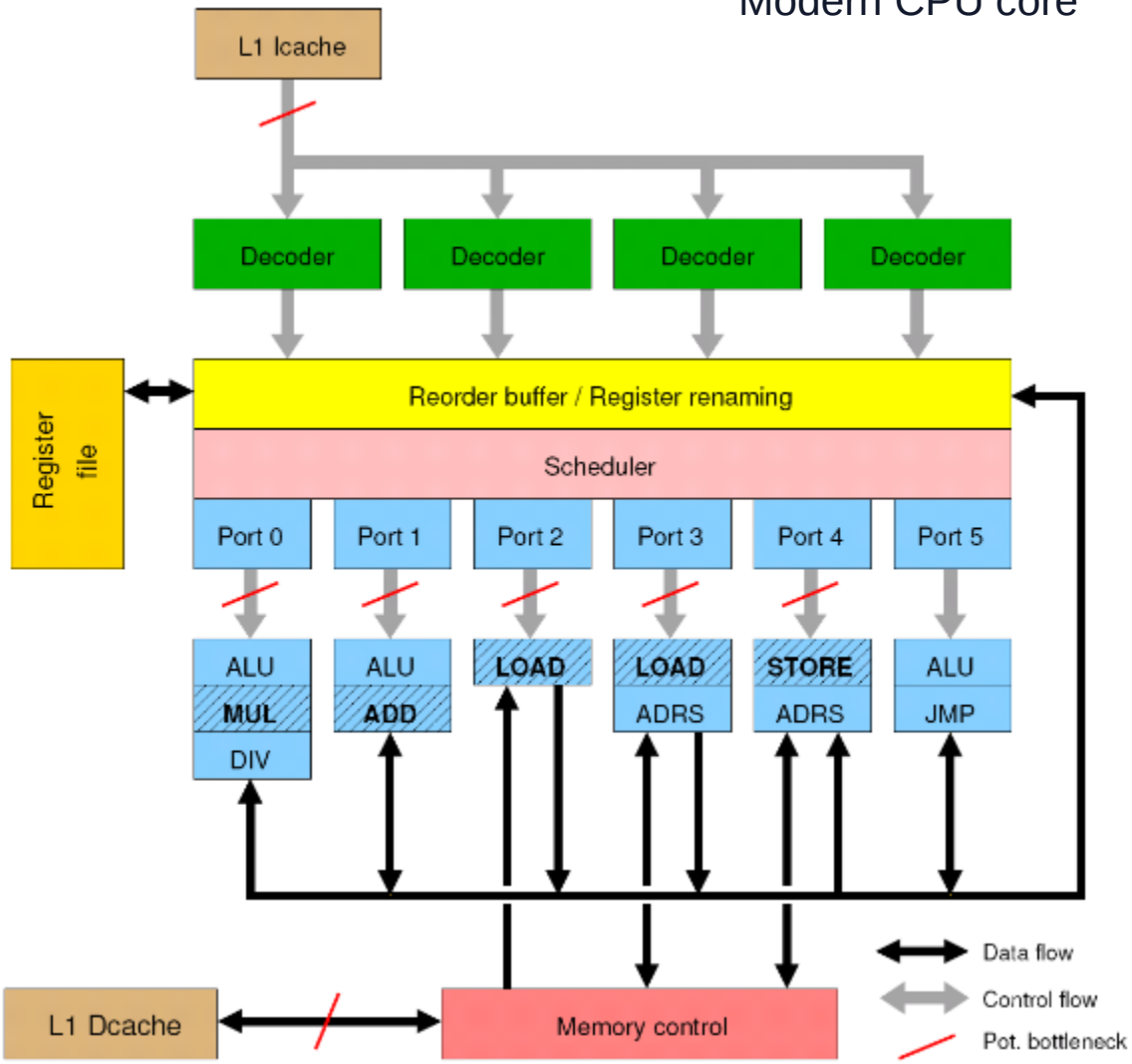
General-purpose cache based microprocessor core



Stored-program computer

- Implements “Stored Program Computer” concept (Turing 1936)
- Similar designs on all modern systems

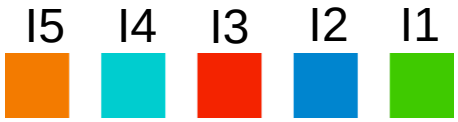
Modern CPU core



Instruction level parallelism

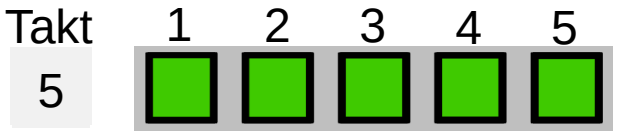
Pipelining

Instructions



Single instruction takes 5 cycles

Stages



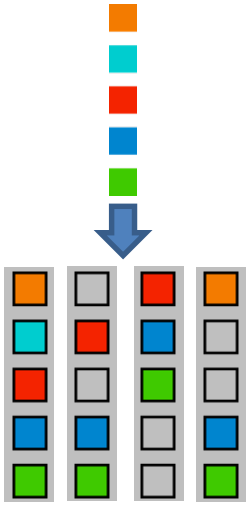
Throughput:

1 instruction per cycle

Speedup by factor 5

Superscalar execution

4-fach superskalar



Throughput:

4 instructions per cycle

Pipelining of arithmetic/functional units

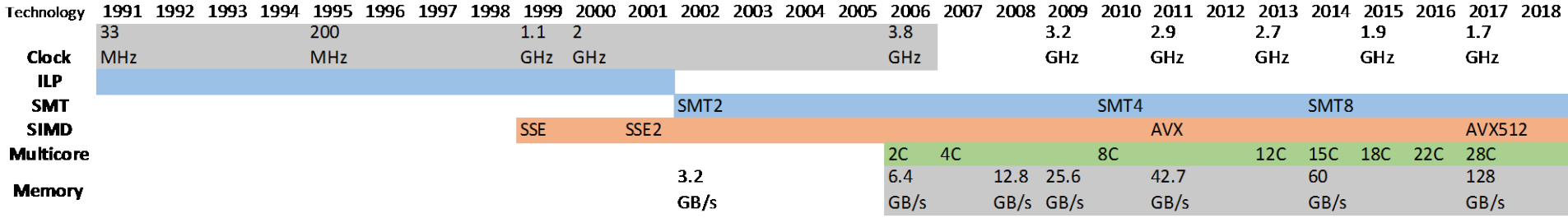
- Idea:
  - Split complex instruction into several simple / fast steps (stages)
  - Each step takes the same amount of time, e.g. a single cycle
  - Execute different steps on different instructions at the same time (in parallel)

- Allows for shorter cycle times (simpler logic circuits), e.g.:
  - floating point multiplication takes 5 cycles, but
  - processor can work on 5 different multiplications simultaneously
  - one result at each cycle after the pipeline is full

- Drawback:
  - Pipeline must be filled - startup times (#Instructions >> pipeline steps)
  - Efficient use of pipelines requires large number of independent instructions → instruction level parallelism
  - Requires complex instruction scheduling by compiler/hardware – software-pipelining / out-of-order

- Pipelining is widely used in modern computer architectures

Technologies Driving Performance



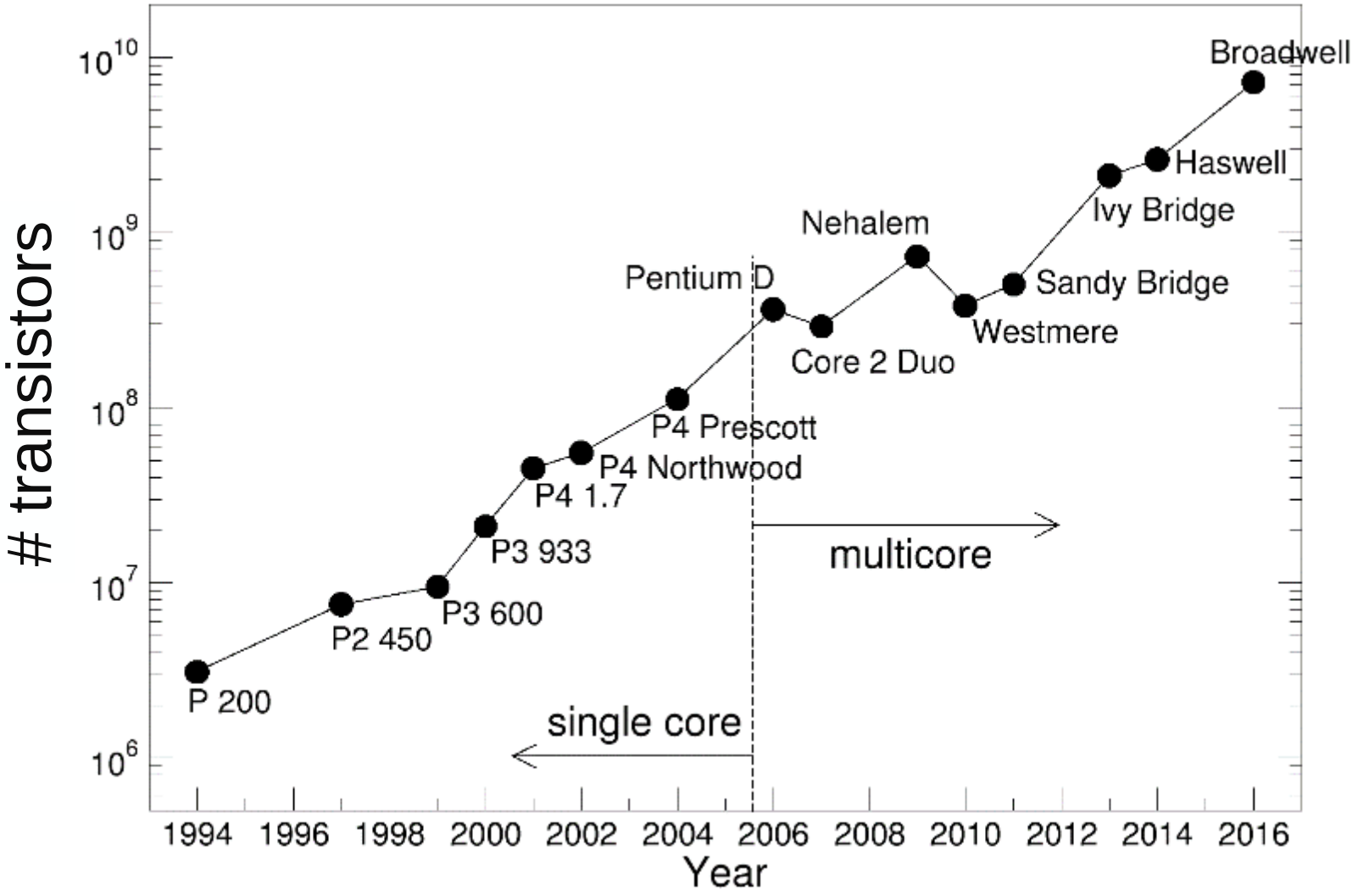
ILP **Obstacle**: Not more parallelism available

Clock **Obstacle**: Power/Heat dissipation

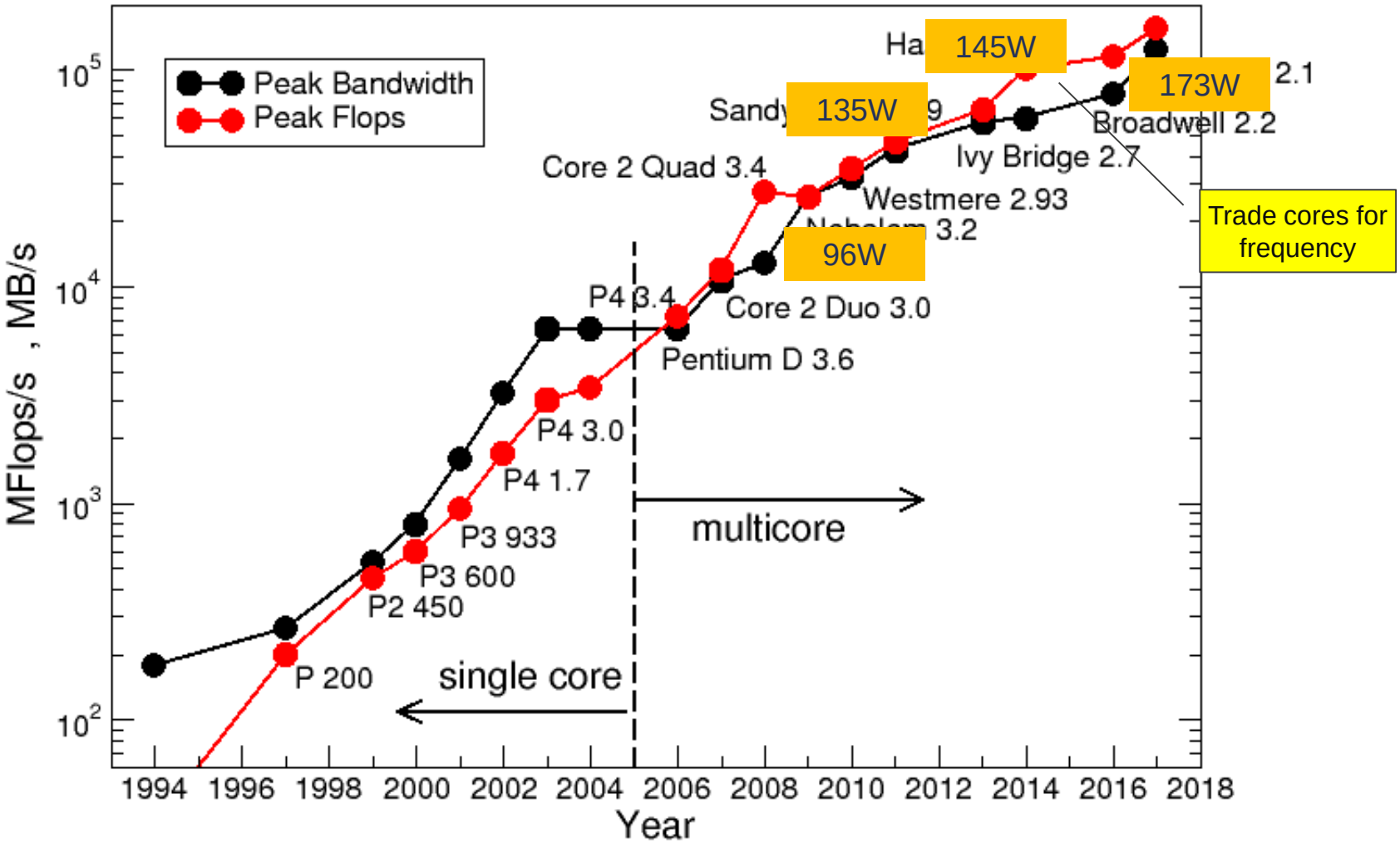
Multi- Manycore **Obstacle**: Getting data to/from cores

SIMD **Obstacle:** Power

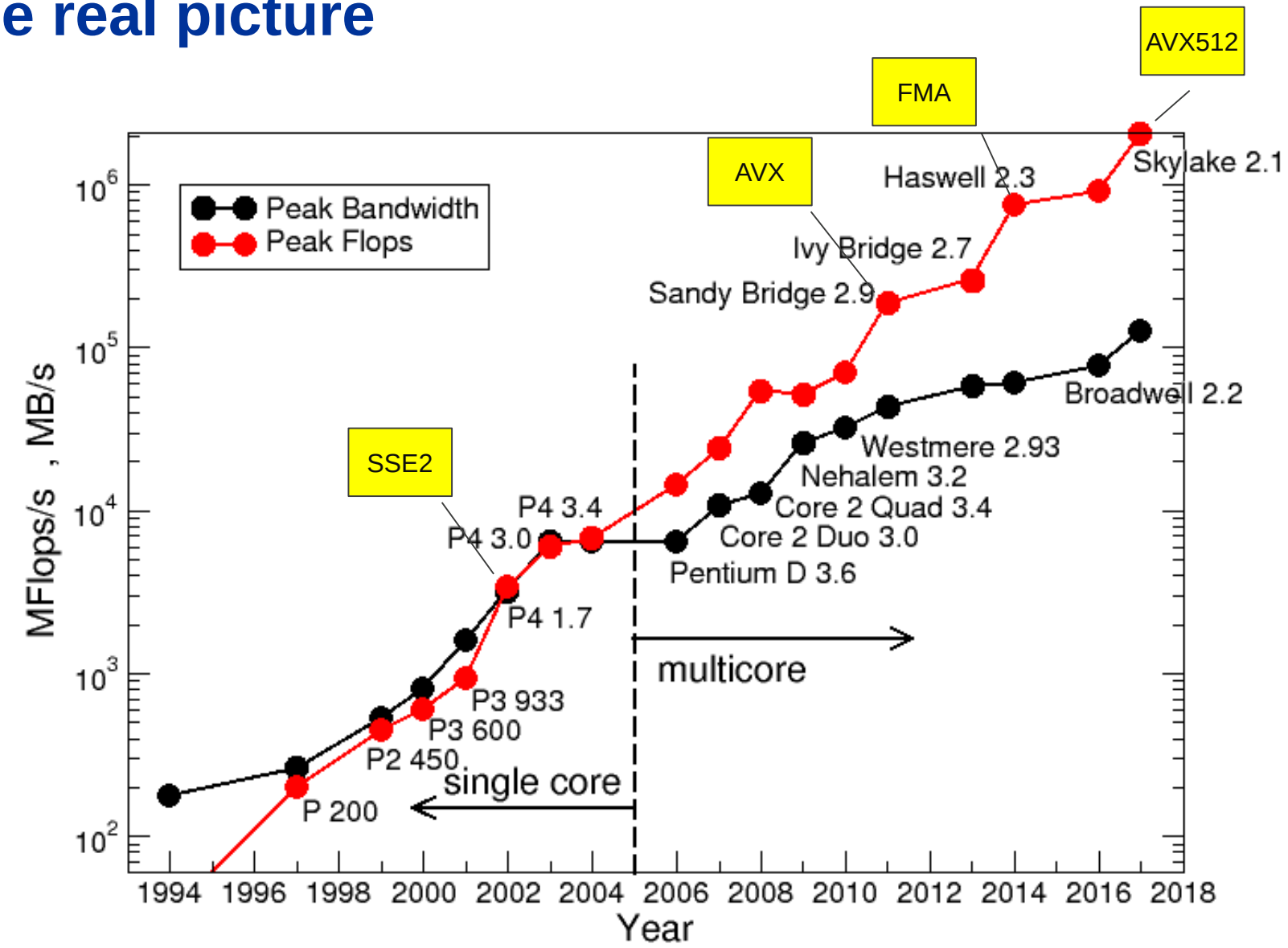
Moores Law: Single chip transistor count



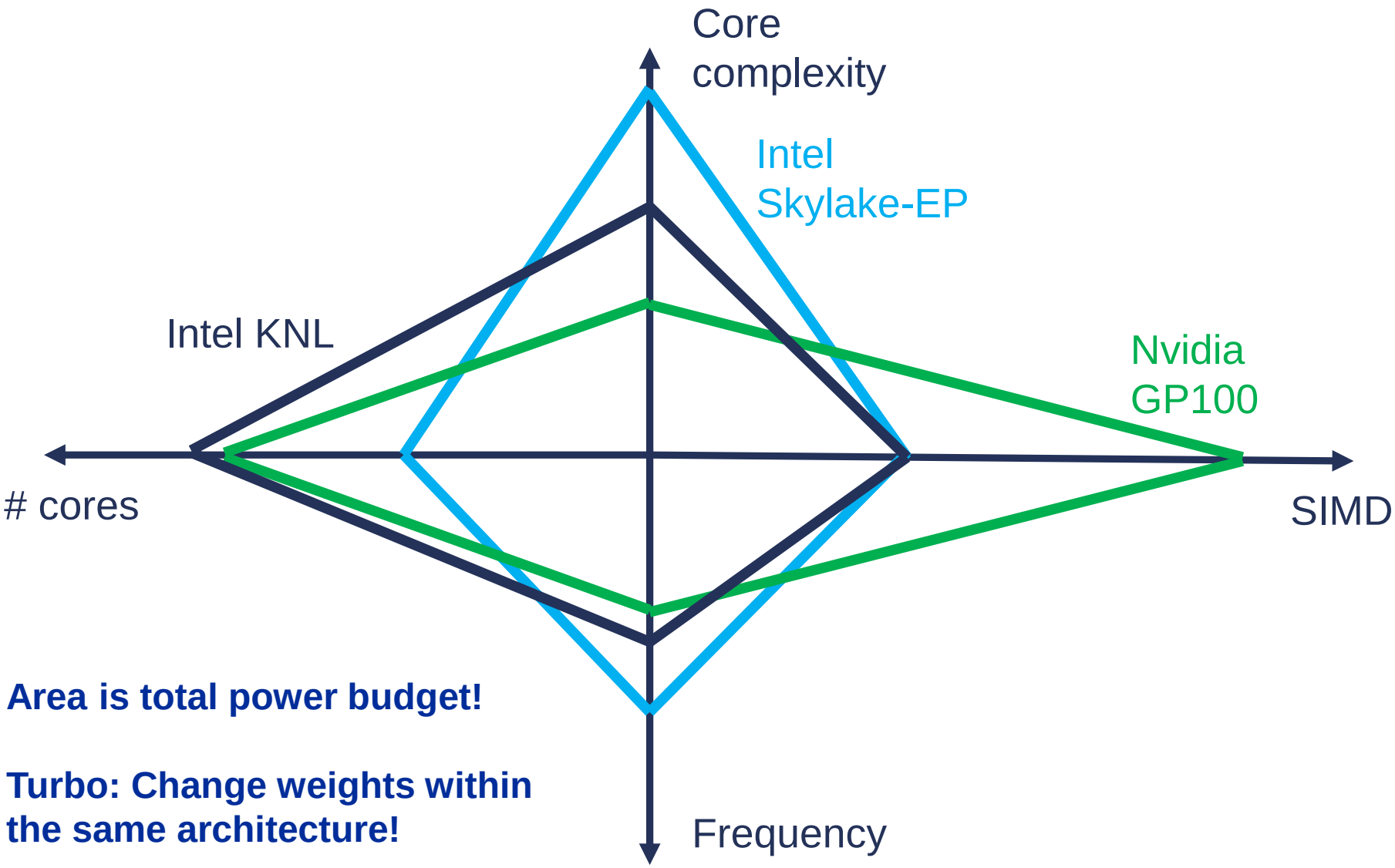
History of Intel chip performance



The real picture



Finding the right compromise



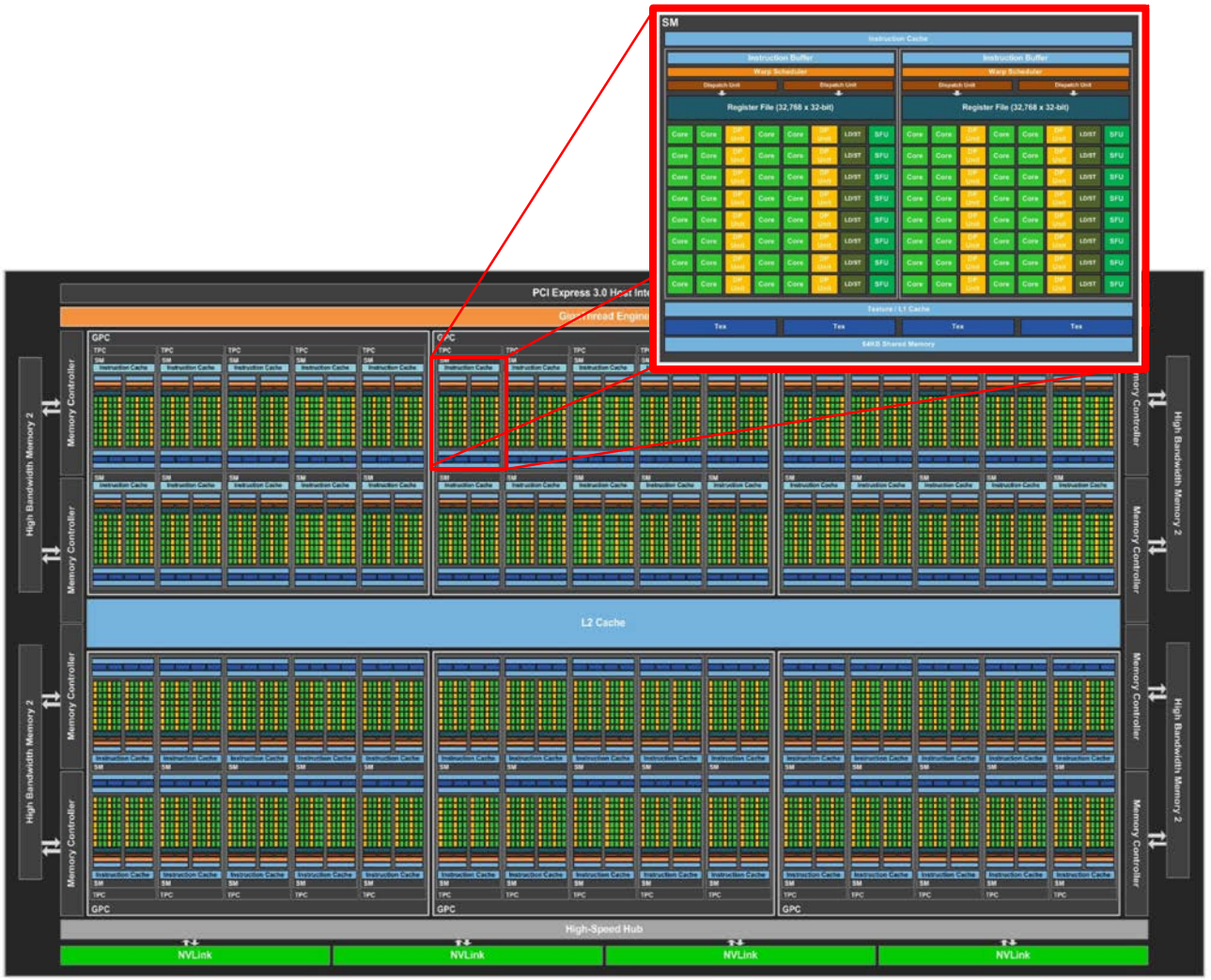
Area is total power budget!

Turbo: Change weights within the same architecture!

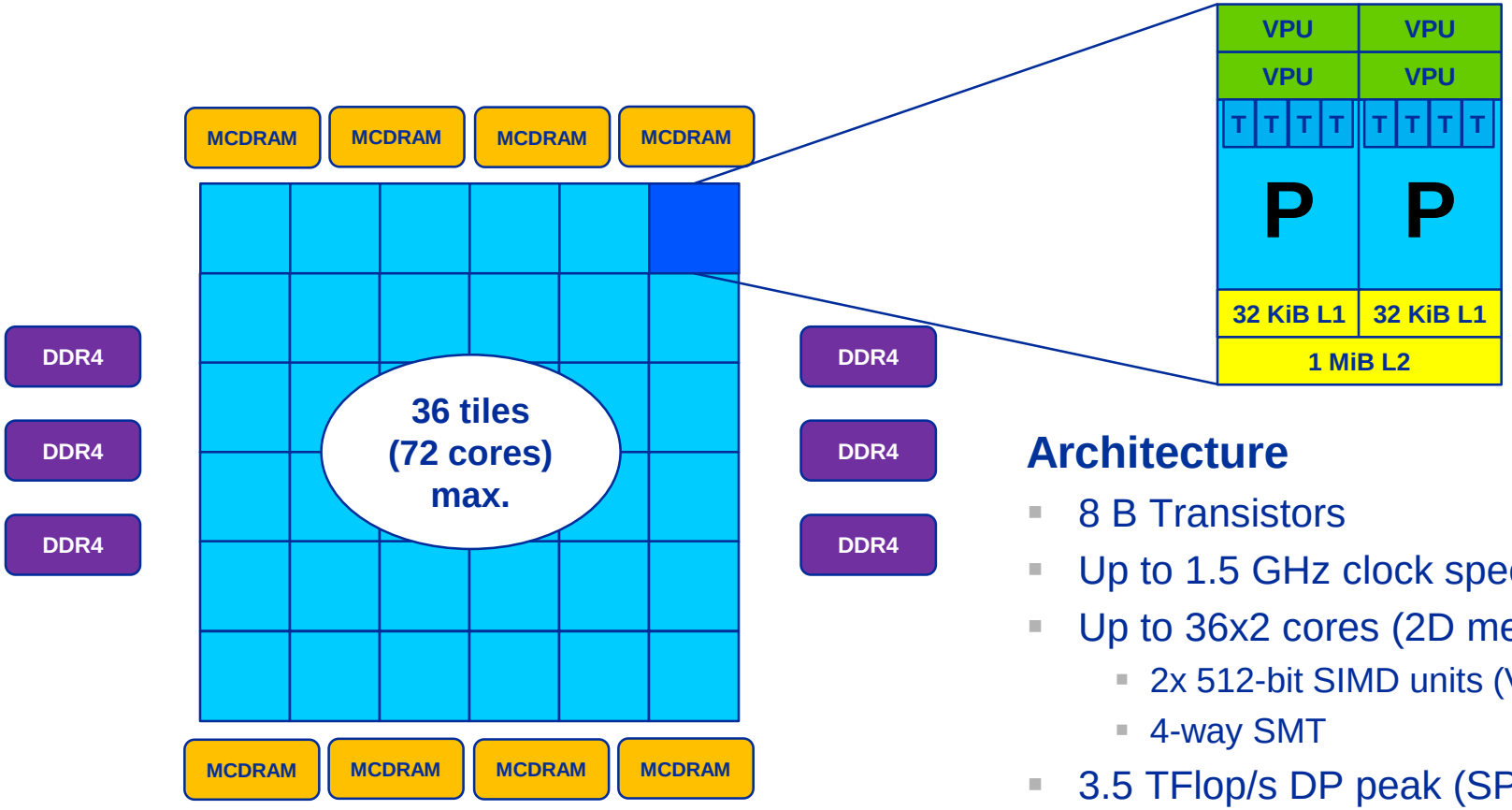
NVidia Pascal GP100 block diagram

Architecture

- 15.3 B Transistors
- ~ 1.4 GHz clock speed
- Up to 60 “SM” units
  - 64 SP “cores” each
  - 32 DP “cores” each
  - 2:1 SP:DP performance
- 5.7 TFlop/s DP peak
- 4 MB L2 Cache
- 4096-bit HBM2
- MemBW ~ 732 GB/s (theoretical)
- MemBW ~ 510 GB/s (measured)



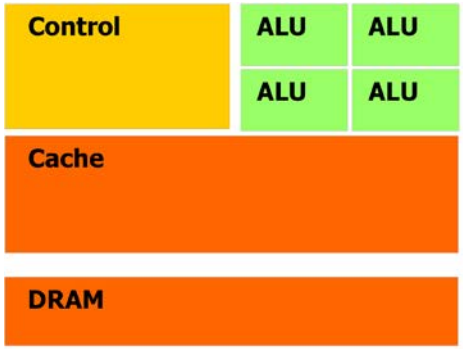
Intel Xeon Phi “Knights Landing” block diagram

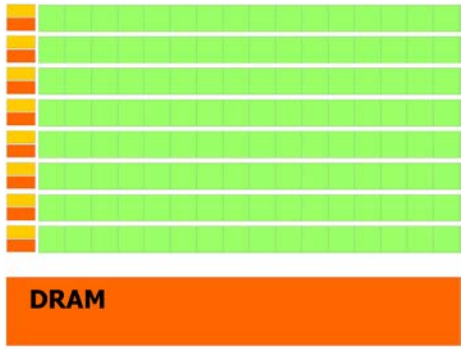


Architecture

- 8 B Transistors
- Up to 1.5 GHz clock speed
- Up to 36x2 cores (2D mesh)
  - 2x 512-bit SIMD units (VPU) each
  - 4-way SMT
- 3.5 TFlop/s DP peak (SP 2x)
- 36 MiB L2 Cache
- 16 GiB MCDRAM
  - MemBW ~ 470 GB/s (measured)
- Large DDR4 main memory
  - MemBW ~ 90 GB/s (measured)

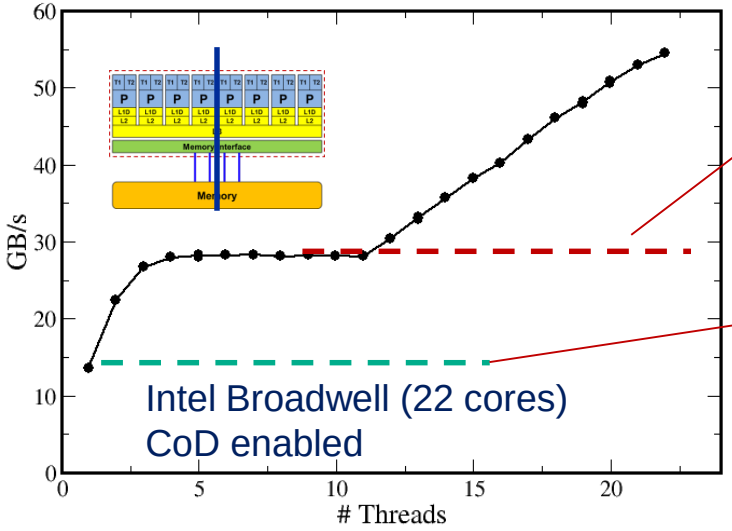
Trading single thread performance for parallelism:
GPGPUs vs. CPUs





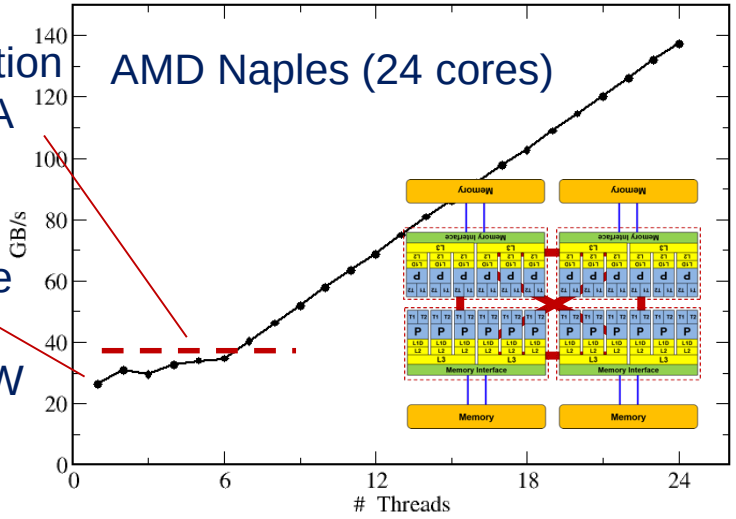
	Intel Xeon Platinum 8170 “Skylake”	Intel Xeon Phi 7250 “Knights Landing”	NVidia Tesla P100 “Pascal”
Cores@Clock	26 @ ≥2.1 GHz	68 @ 1.4 GHz	56 SMs @ ~1.3 GHz
SP Performance/core	147.2 GFlop/s	89.6 GFlop/s	~166 GFlop/s
Threads@STREAM	~8	~40	> 10000
SP peak	3.83 TFlop/s	6.1 TFlop/s	~9.3 TFlop/s
Stream BW (meas.)	115.8 GB/s	450 GB/s (MCDRAM)	510 GB/s
Transistors / TDP	8 Billion / 173 W	8 Billion / 215W	14 Billion/300W

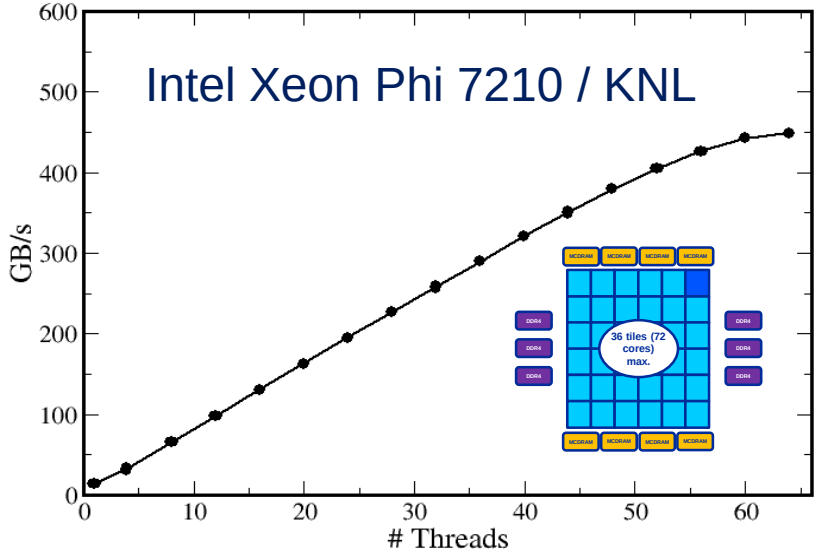
Attainable memory bandwidth: Comparing architectures

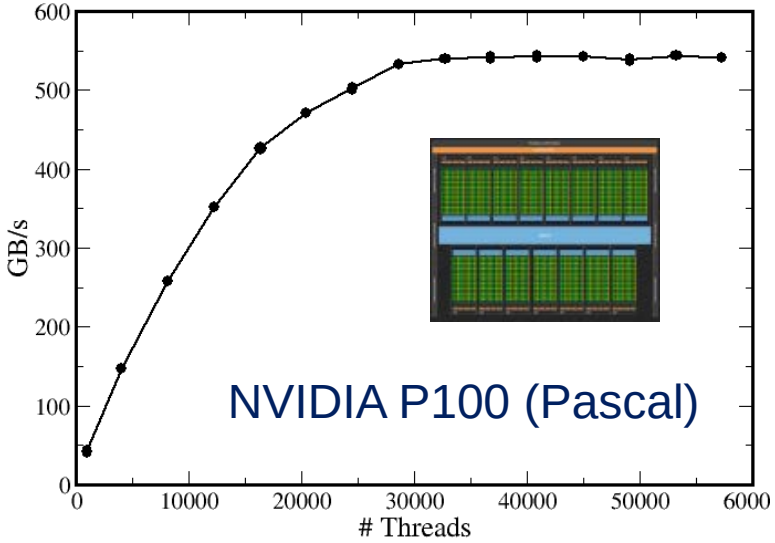


BW saturation in NUMA domain

Single core does not saturate BW



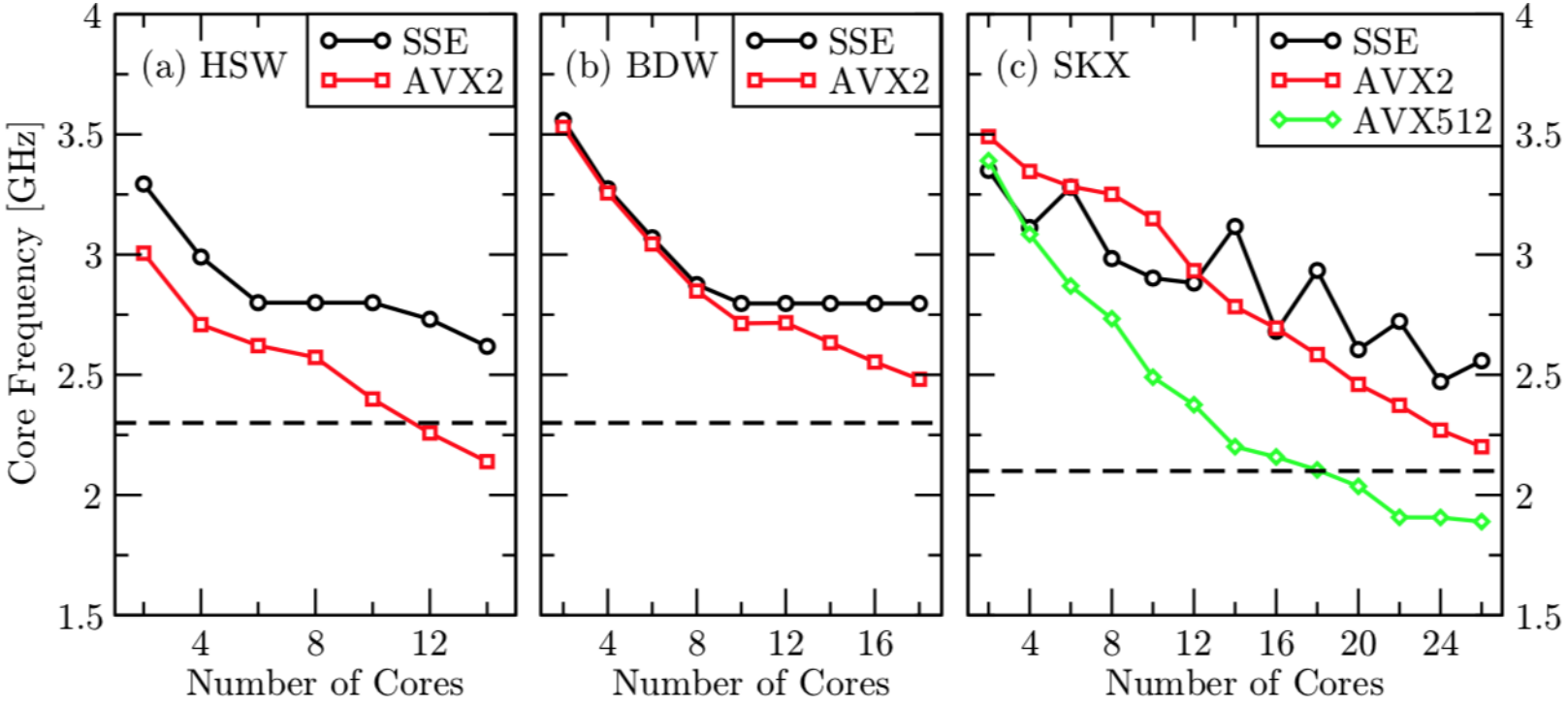




SIMD and Turbo mode

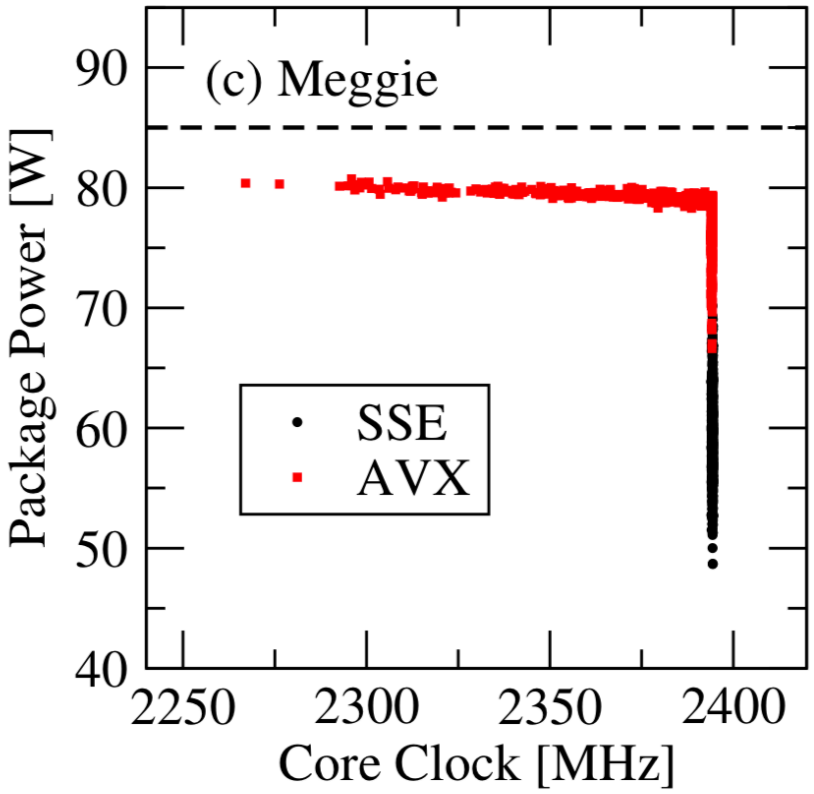
Haswell 2.3 GHz

Broadwell 2.3 GHz



Turn off Turbo is not an option because base AVX clock is low!

And there is no guarantee



1456 Xeon E5-2630v4
10 cores 2.2 GHz

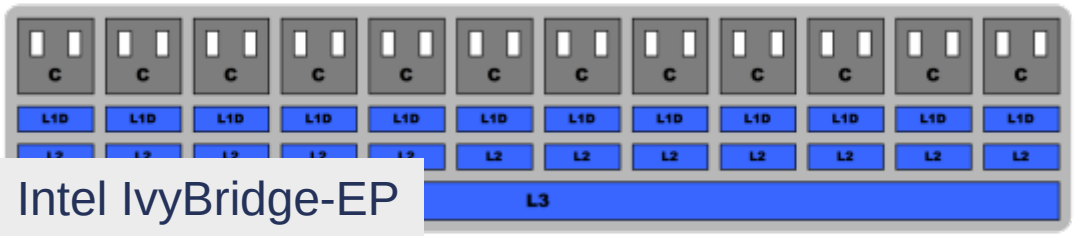
Maximum DP floating point (FP) performance

$$P_{core} = n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f$$

- n_{super}^{FP}: Super-scalarity
- n_{FMA}: FMA factor
- n_{SIMD}: SIMD factor
- f: Clock Speed

uArch	n_{super}^{FP}	n_{FMA}	n_{SIMD}	n_{cores}	Release	Model	P_{core} [GF/s]	P_{chip} [GF/s]	P_{serial} [GF/s]	TDP	GF/ Watt
Sandy Bridge	2	1	4	8	Q1/2012	E5-2680	11.7	173	7	130	1,33
Ivy Bridge	2	1	4	10	Q3/2013	E5-2690-v2	24	240	7,2	130	1,85
KNC	1	2	8	61	Q2/2014	7120A	10.6	1210	1,3	300	4,03
Haswell	2	2	4	14	Q3/2014	E5-2695-v3	21.6	425	6,6	120	3,54
Broadwell	2	2	4	22	Q1/2016	E5-2699-v4	17.6	704	7,2	145	4,85
Pascal	1	2	32	56	Q2/2016	GP100	36.8	4700	1,5	300	15,67
KNL	2	2	8	72	Q4/2016	7290F	35.2	2995	3,4	260	11,52
Skylake	2	2	8	26	Q3/2017	8170	23.4	1581	7,6	165	9,58

The driving forces behind performance 2012



Intel IvyBridge-EP

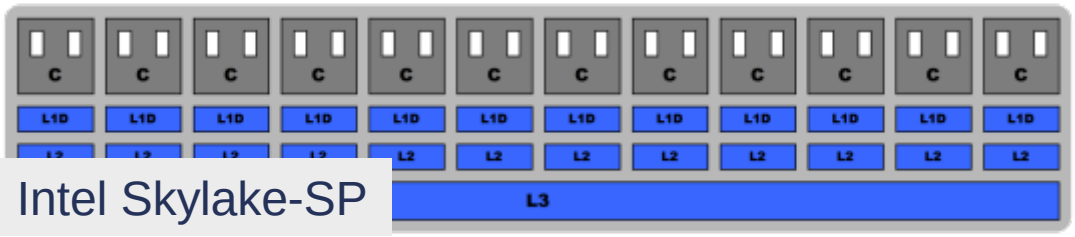
$$P = n_{core} * F * S * \nu$$

	Intel IvyBridge-EP
Number of cores n_{core}	**12**
FP instructions per cycle F	2
FP ops per instructions S	**4 (DP) / 8 (SP)**
Clock speed [GHz] n	2.7
Performance [GF/s] P	259 (DP) / 518 (SP)

TOP500 rank 1 (1996)

But: P=5.4 GF/s for serial, non-SIMD code

The driving forces behind performance 2018



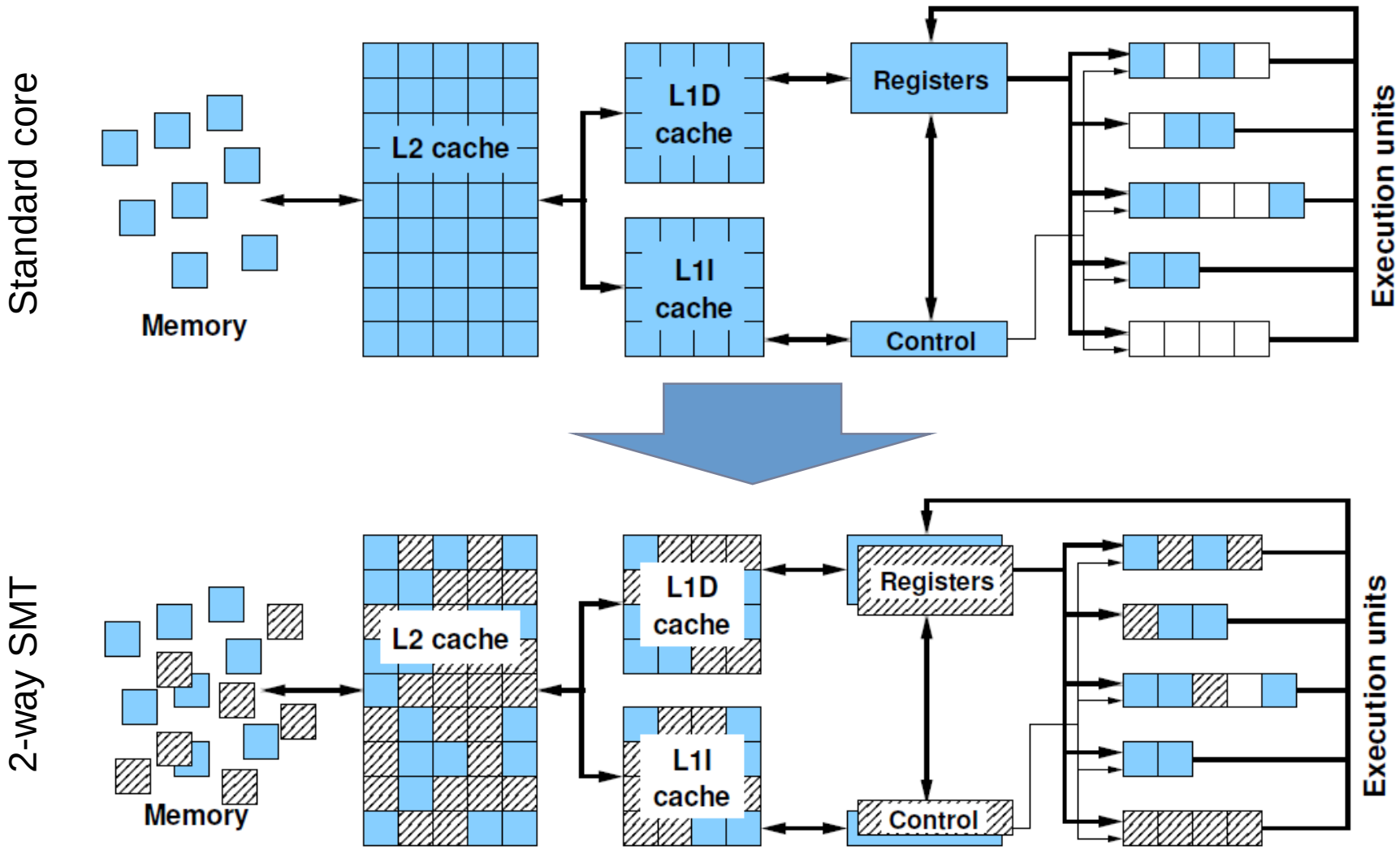
Intel Skylake-SP

$$P = n_{core} * F * M * S * \nu$$

	Intel IvyBridge-EP
Number of cores n_{core}	**28**
FP instructions per cycle F	2
FMA factor M	2
FP ops per instructions S	**8 (DP) / 16 (SP)**
Clock speed [GHz] n	2.3 (scalar 2.8)
Performance [GF/s] P	2060 (DP) / 4122 (SP)

But: P=5.6 GF/s for serial, non-SIMD code

Core details: Simultaneous multi-threading (SMT)



Data parallel execution units (SIMD)

```
for (int j=0; j<size; j++){
    A[j] = B[j] + C[j];
}
```

Register widths

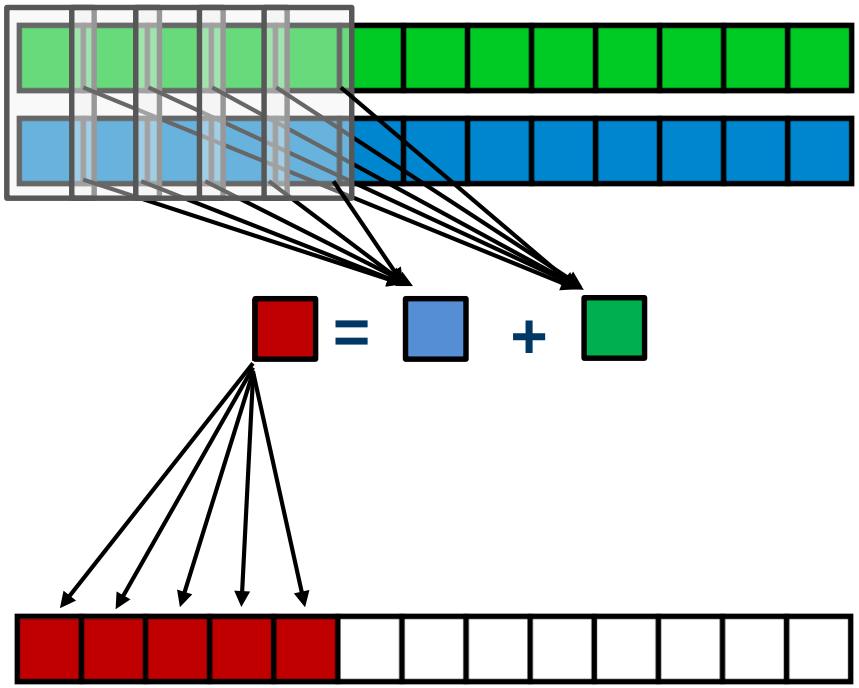
- 1 operand

- 2 operands (SSE)

- 4 operands (AVX)

- 8 operands (AVX512)

Scalar execution



Data parallel execution units (SIMD)

```
for (int j=0; j<size; j++){
    A[j] = B[j] + C[j];
}
```

Register widths

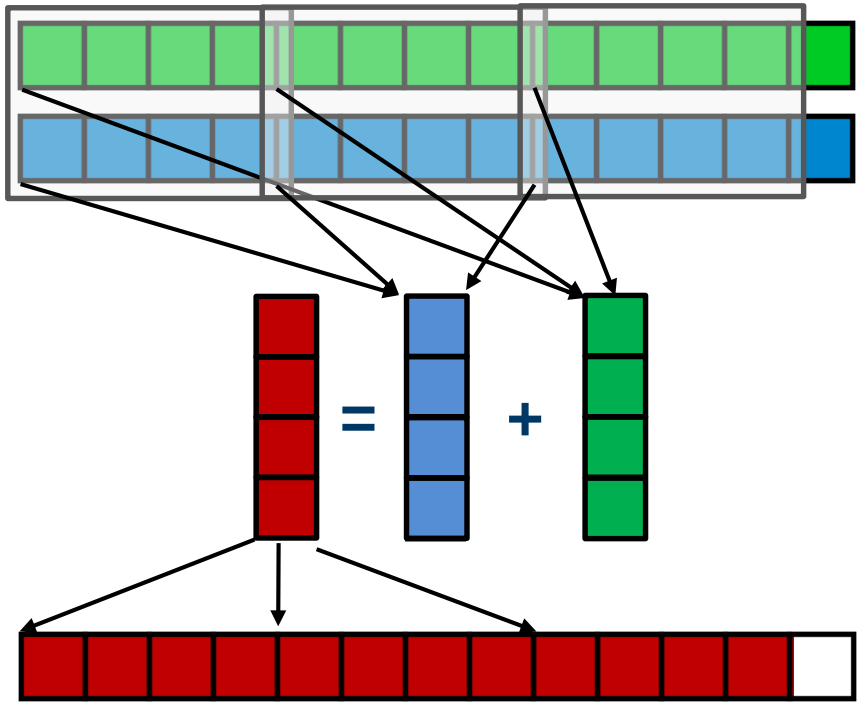
- 1 operand

- 2 operands (SSE)

- 4 operands (AVX)

- 8 operands (AVX512)

SIMD execution



SIMD processing – Basics

Steps (done by the compiler) for “SIMD processing”

```
for(int i=0; i<n;i++)
        C[i]=A[i]+B[i];
```

“Loop unrolling”

```
for(int i=0; i<n;i+=4){
        C[i]  =A[i]  +B[i];
        C[i+1]=A[i+1]+B[i+1];
        C[i+2]=A[i+2]+B[i+2];
        C[i+3]=A[i+3]+B[i+3];}
//remainder loop handling
```

Load 256 Bits starting from address of `A[i]` to register `R0`

Add the corresponding 64 Bit entries in `R0` and `R1` and store the 4 results to `R2`

Store `R2` (256 Bit) to address starting at `C[i]`

```
LABEL1:
        VLOAD R0 ← A[i]
        VLOAD R1 ← B[i]
        V64ADD[R0,R1] → R2
        VSTORE R2 → C[i]
        i←i+4
        i<(n-4)? JMP LABEL1
//remainder loop handling
```

SIMD processing – Basics

No SIMD vectorization for loops with data dependencies:

```
for(int i=0; i<n;i++)
       A[i]=A[i-1]*s;
```

"Pointer aliasing" may prevent SIMDfication

```
void f(double *A, double *B, double *C, int n) {
       for(int i=0; i<n; ++i)
          C[i] = A[i] + B[i];
}
```

C/C++ allows that **A → &C[-1]** and **B → &C[-2]**

→ C[i] = C[i-1] + C[i-2]: dependency → No SIMD

If "pointer aliasing" is not used, tell it to the compiler:

-fno-alias (Intel), **-Msafeptr** (PGI), **-fargument-noalias** (gcc)

restrict keyword (C only!):

```
void f(double restrict *A, double restrict *B, double restrict *C, int n) {…}
```

Why and how?

Why check the assembly code?

- Sometimes the only way to make sure the compiler "did the right thing"
  - Example: "LOOP WAS VECTORIZED" message is printed, but Loads & Stores may still be scalar!
- Get the assembler code (Intel compiler):

```
icc –S –O3  -xHost  triad.c  -o a.out
```

- Disassemble Executable:

```
objdump –d  ./a.out | less
```

The x86 ISA is documented in:

Intel Software Development Manual (SDM) 2A and 2B

AMD64 Architecture Programmer's Manual Vol. 1-5

Basics of the x86-64 ISA

- Instructions have 0 to 3 operands (4 with AVX-512)
- Operands can be registers, memory references or immediates
- Opcodes (binary representation of instructions) vary from 1 to 17 bytes
- There are two assembler syntax forms: Intel (left) and AT&T (right)
- Addressing Mode: BASE + INDEX * SCALE + DISPLACEMENT
- C: **`A[i]`** equivalent to **`*(A+i)`** (a pointer has a type: **`A+i*8`**)

```
movaps [rdi + rax*8+48], xmm3
add rax, 8
js 1b
```

```
movaps    %xmm4, 48(%rdi,%rax,8)
addq      $8, %rax
js        ..B1.4
```

```
401b9f: 0f 29 5c c7 30      movaps %xmm3,0x30(%rdi,%rax,8)
401ba4: 48 83 c0 08         add    $0x8,%rax
401ba8: 78 a6               js     401b50 <triad_asm+0x4b>
```

Basics of the x86-64 ISA with extensions

16 general Purpose Registers (**64bit**):

`rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8-r15`

alias with eight 32 bit register set:

`eax, ebx, ecx, edx, esi, edi, esp, ebp`

8 opmask registers (16bit or 64bit, AVX512 only):

`k0–k7`

Floating Point **SIMD** Registers:

`xmm0-xmm15 (xmm31)`	SSE (128bit) alias with 256-bit and 512-bit registers
`ymm0-ymm15 (xmm31)`	AVX (256bit) alias with 512-bit registers
`zmm0-zmm31`	AVX-512 (512bit)

SIMD instructions are distinguished by:

VEX/EVEX prefix:	**`v`**
Operation:	**`mul, add, mov`**
Modifier:	nontemporal (**`nt`**), unaligned (**u**), aligned (**a**), high (**h**)
Width:	scalar (**s**), packed (**p**)
Data type:	single (**s**), double (**d**)

ISA support on KNL

KNL supports all **legacy** ISA extensions:

`MMX, SSE, AVX, AVX2`

Furthermore **KNL** supports:

- AVX-512 Foundation (F), KNL and Skylake
- AVX-512 Conflict Detection Instructions (CD), KNL and Skylake
- AVX-512 Exponential and Reciprocal Instructions (ER), KNL
- AVX-512 Prefetch Instructions (PF), KNL

AVX-512 extensions only supported on **Skylake**:

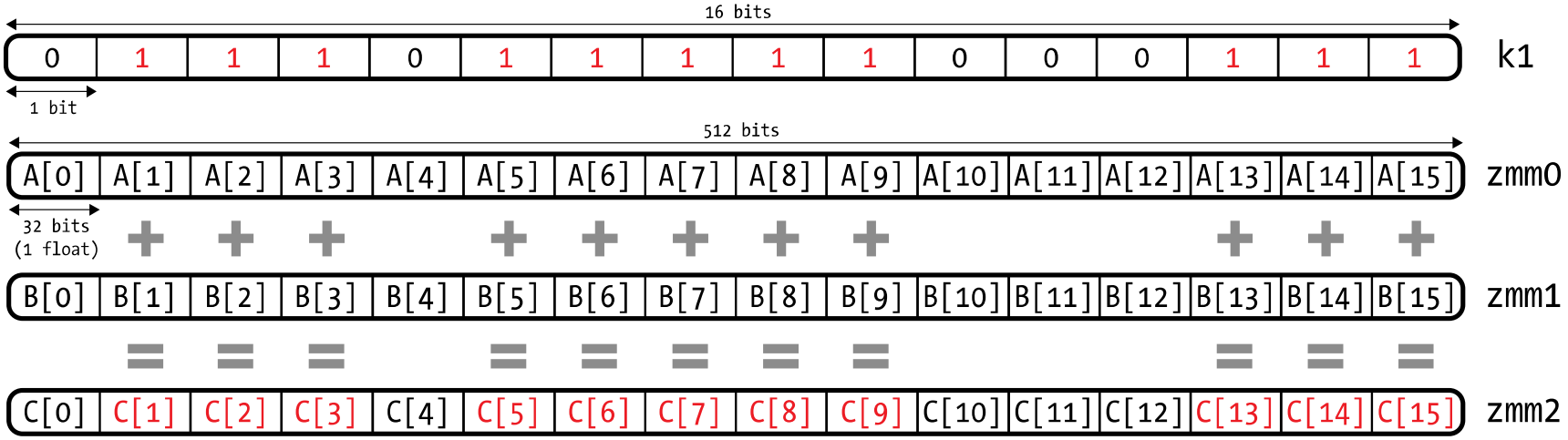
- AVX-512 Byte and Word Instructions (BW)
- AVX-512 Doubleword and Quadword Instructions (DQ)
- AVX-512 Vector Length Extensions (VL)

ISA Documentation:

Intel Architecture Instruction Set Extensions Programming Reference

Example for masked execution

Masking for predication is very helpful in cases such as e.g. remainder loop handling or conditional handling.



Architecture specific issues KNC vs. KNL

KNC architectural issues

- Fragile single core performance (in-order, pairing, SMT)
- No proper hardware prefetching
- Shared access on segmented LLC costly

KNL fixes most of these issues and is more accessible!

Advices for KNL

- 1 thread per core is usually best, sometime two threads per core
- Large pages can improve performance significantly (2M,1G)
- Consider the `-no-prec-div` option to enable AVX-512 ER instructions
- Aggressive software prefetching is usually not necessary
- MCDRAM is the preferred target memory (try cache mode first)
- Alignment restrictions and penalties are similar to Xeon. We experienced a benefit from alignment to page size with the MCDRAM.

Case Study: Simplest code for the summation of the elements of a vector (single precision)

```
float sum = 0.0;

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

To get object code use `objdump -d` on object file or executable or compile with `-S`

AT&T syntax:
`addss 0(%rdx,%rax,4),%xmm0`

Instruction code:

```
401d08:   f3 0f 58 04 82      addss  xmm0,[rdx + rax * 4]
401d0d:   48 83 c0 01         add    rax,1
401d11:   39 c7               cmp    edi,eax
401d13:   77 f3               ja     401d08
```

(final sum across xmm0 omitted)

Instruction address

Opcodes

Assembly code

Case Study: Vector Triad (DP) on IvyBridge-EP

```
for (int i = 0; i < length; i++) {
    A[i] = B[i] + D[i] * C[i];
}
```

To get object code use `objdump -d` on object file or executable or compile with `-S`

Assembly code (-O1):

CLANG

```
LBB0_3
movsd   xmm0, [rdx]
mulsd   xmm0, [rcx]
addsd   xmm0, [rsi]
movsd   [rax], xmm0
add     rsi, 8
add     rdx, 8
add     rcx, 8
add     rax, 8
dec     edi
jne     LBB0_3
```

ICC

```
..B1.6:
movsd   xmm0, [r12+rax*8]
mulsd   xmm0, [r13+rax*8]
addsd   xmm0, [r14+rax*8]
movsd   [r15+rax*8], xmm0
inc     rax
cmp     rax, rbx
jl      ..B1.6
```

GCC

```
.L4:
movsd xmm0,[rbx+rax]
mulsd xmm0,[r12+rax]
addsd xmm0,[r13+0+rax]
movsd [rbp+0+rax],xmm0
add rax, 8
cmp rax, r14
jne .L4
```

7 instructions per loop iteration

Case Study: Vector Triad (DP) –O3 (Intel compiler)

```
..B1.19:
        movsd     xmm0,  [r15+rsi*8]
        movsd     xmm3,  [16+r15+rsi*8]
        movsd     xmm5,  [32+r15+rsi*8]
        movsd     xmm7,  [48+r15+rsi*8]
        movhpd    xmm0,  [8+r15+rsi*8]
        movhpd    xmm3,  [24+r15+rsi*8]
        movhpd    xmm5,  [40+r15+rsi*8]
        movhpd    xmm7,  [56+r15+rsi*8]
        mulpd     xmm0,  [r14+rsi*8]
        mulpd     xmm3,  [16+r14+rsi*8]
        mulpd     xmm5,  [32+r14+rsi*8]
        mulpd     xmm7,  [48+r14+rsi*8]
        movsd     xmm2,  [r13+rsi*8]
        movsd     xmm4,  [16+r13+rsi*8]
        movsd     xmm6,  [32+r13+rsi*8]
        movsd     xmm8,  [48+r13+rsi*8]
        movhpd    xmm2,  [8+r13+rsi*8]
        movhpd    xmm4,  [24+r13+rsi*8]
        movhpd    xmm6,  [40+r13+rsi*8]
        movhpd    xmm8,  [56+r13+rsi*8]
        addpd     xmm2, xmm0
        addpd     xmm4, xmm3
        addpd     xmm6, xmm5
        addpd     xmm8, xmm7
        movaps    [rdx+rsi*8], xmm2
        movaps    [16+rdx+rsi*8], xmm4
        movaps    [32+rdx+rsi*8], xmm6
        movaps    [48+rdx+rsi*8], xmm8
        add       rsi, 8
        cmp       rsi, r9
        jb        ..B1.19
```

SSE

3.86 instructions per loop iteration

Case Study: Vector Triad (DP) –O3 –xHost

```
..B1.15:
vmovupd   xmm2, [r15+rsi*8]
vmovupd   xmm10, [32+r15+rsi*8]
vmovupd   xmm3, [rdx+rsi*8]
vmovupd   xmm11, [32+rdx+rsi*8]
vmovupd   xmm0, [r14+rsi*8]
vmovupd   xmm9, [32+r14+rsi*8]
vinsertf128 ymm4, ymm2,[16+r15+rsi*8], 1
vinsertf128 ymm12,ymm10,[48+r15+rsi*8],1
vinsertf128 ymm5, ymm3,[16+rdx+rsi*8], 1
vinsertf128 ymm13,ymm11,[48+rdx+rsi*8],1
vmulpd    ymm7, ymm4, ymm5
vmulpd    ymm15, ymm12, ymm13
vmovupd   xmm4, [64+rdx+rsi*8]
vmovupd   xmm12, [96+rdx+rsi*8]
vmovupd   xmm3, [64+r15+rsi*8]
vmovupd   xmm11, [96+r15+rsi*8]
vmovupd   xmm2, [64+r14+rsi*8]
vmovupd   xmm10, [96+r14+rsi*8]
vinsertf128 ymm14,ymm9,[48+r14+rsi*8], 1
vinsertf128 ymm6,ymm0,[16+r14+rsi*8], 1
vaddpd    ymm8, ymm6, ymm7          vaddpd
ymm0, ymm14, ymm15
```

AVX

```
vmovupd   [r13+rsi*8], ymm8
vmovupd   [32+r13+rsi*8], ymm0
vinsertf128 ymm5, ymm3, [80+r15+rsi*8], 1
vinsertf128 ymm13,ymm11,[112+r15+rsi*8], 1
vinsertf128 ymm6, ymm4,  [80+rdx+rsi*8], 1
vinsertf128 ymm14,ymm12,[112+rdx+rsi*8], 1
vmulpd    ymm8, ymm5, ymm6
vmulpd    ymm0, ymm13, ymm14
vinsertf128 ymm7, ymm2, [80+r14+rsi*8], 1
vinsertf128 ymm15,ymm10,[112+r14+rsi*8], 1
vaddpd    ymm9, ymm7, ymm8
vaddpd    ymm2, ymm15, ymm0
vmovupd   [64+r13+rsi*8], ymm9
vmovupd   [96+r13+rsi*8], ymm2
add       rsi, 16
cmp       rsi, r9
jb        ..B1.15
```

2.44 instructions per loop iteration

Benefit of SIMD limited by *serial* fraction!

Case Study: Vector Triad (DP) –O3 –xHost

`#pragma vector aligned`

SSE

```
..B1.7:
movaps    xmm0, [r13+rcx*8]
movaps    xmm2, [16+r13+rcx*8]
movaps    xmm3, [32+r13+rcx*8]
movaps    xmm4, [48+r13+rcx*8]
mulpd     xmm0, [rbp+rcx*8]
mulpd     xmm2, [16+rbp+rcx*8]
mulpd     xmm3, [32+rbp+rcx*8]
mulpd     xmm4, [48+rbp+rcx*8]
addpd     xmm0, [r12+rcx*8]
addpd     xmm2, [16+r12+rcx*8]
addpd     xmm3, [32+r12+rcx*8]
addpd     xmm4, [48+r12+rcx*8]
movaps    [r15+rcx*8], xmm0
movaps    [16+r15+rcx*8], xmm2
movaps    [32+r15+rcx*8], xmm3
movaps    [48+r15+rcx*8], xmm4
add       rcx, 8
cmp       rcx, rsi
jb        ..B1.7
```

2.38 instructions per loop iteration

AVX

```
..B1.7:
vmovupd   ymm0, [r15+rcx*8]
vmovupd   ymm4, [32+r15+rcx*8]
vmovupd   ymm7, [64+r15+rcx*8]
vmovupd   ymm10,[96+r15+rcx*8]
vmulpd    ymm2, ymm0, [rdx+rcx*8]
vmulpd    ymm5, ymm4, [32+rdx+rcx*8]
vmulpd    ymm8, ymm7, [64+rdx+rcx*8]
vmulpd    ymm11, ymm10, [96+rdx+rcx*8]
vaddpd    ymm3, ymm2, [r14+rcx*8]
vaddpd    ymm6, ymm5, [32+r14+rcx*8]
vaddpd    ymm9, ymm8, [64+r14+rcx*8]
vaddpd    ymm12, ymm11, [96+r14+rcx*8]
vmovupd   [r13+rcx*8], ymm3
vmovupd   [32+r13+rcx*8], ymm6
vmovupd   [64+r13+rcx*8], ymm9
vmovupd   [96+r13+rcx*8], ymm12
add       rcx, 16
cmp       rcx, rsi
jb        ..B1.7
```

1.19 instructions per loop iteration

Case Study: Vector Triad (DP) –O3 –xHost #pragma vector aligned on Haswell-EP

```
..B1.7:
vmovupd   ymm2, [r15+rcx*8]
vmovupd   ymm4, [32+r15+rcx*8]
vmovupd   ymm6, [64+r15+rcx*8]
vmovupd   ymm8, [96+r15+rcx*8]
vmovupd   ymm0, [rdx+rcx*8]
vmovupd   ymm3, [32+rdx+rcx*8]
vmovupd   ymm5, [64+rdx+rcx*8]
vmovupd   ymm7, [96+rdx+rcx*8]
vfmadd213pd ymm2, ymm0, [r14+rcx*8]
vfmadd213pd ymm4, ymm3, [32+r14+rcx*8]
vfmadd213pd ymm6, ymm5, [64+r14+rcx*8]
vfmadd213pd ymm8, ymm7, [96+r14+rcx*8]
vmovupd   [r13+rcx*8], ymm2
vmovupd   [32+r13+rcx*8], ymm4
vmovupd   [64+r13+rcx*8], ymm6
vmovupd   [96+r13+rcx*8], ymm8
add       rcx, 16
cmp       rcx, rsi
jb        ..B1.7
```

AVX + FMA3

1.19 instructions per loop iteration

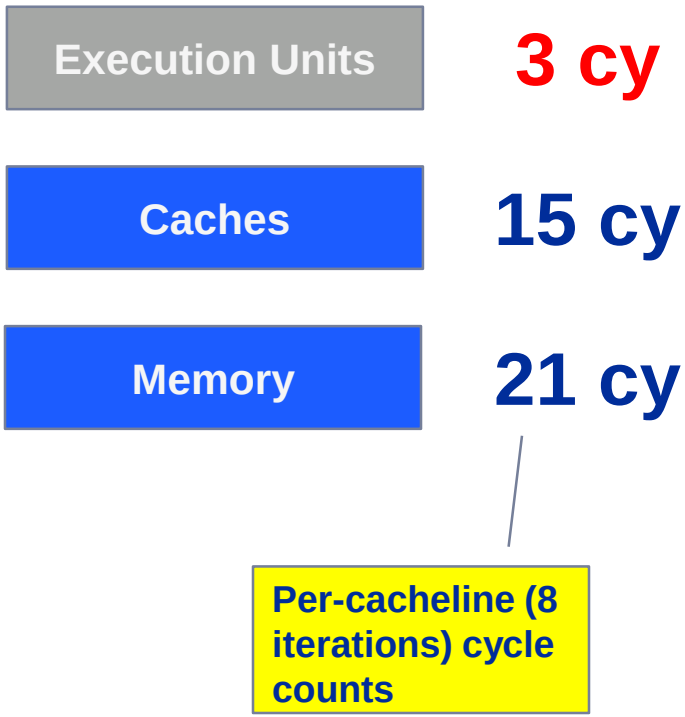
On X86 ISA instruction are converted to so-called µops (elementary ops like load, add, mult). For performance the number of µops is important.

23 uops vs. 27 µops (AVX)

SIMD processing – The whole picture

SIMD influences instruction execution in the core – other runtime contributions stay the same!

AVX example:



Comparing total execution time:

	Execution	Cache	Memory
Scalar	12		
SSE	6	15	21
AVX	3		

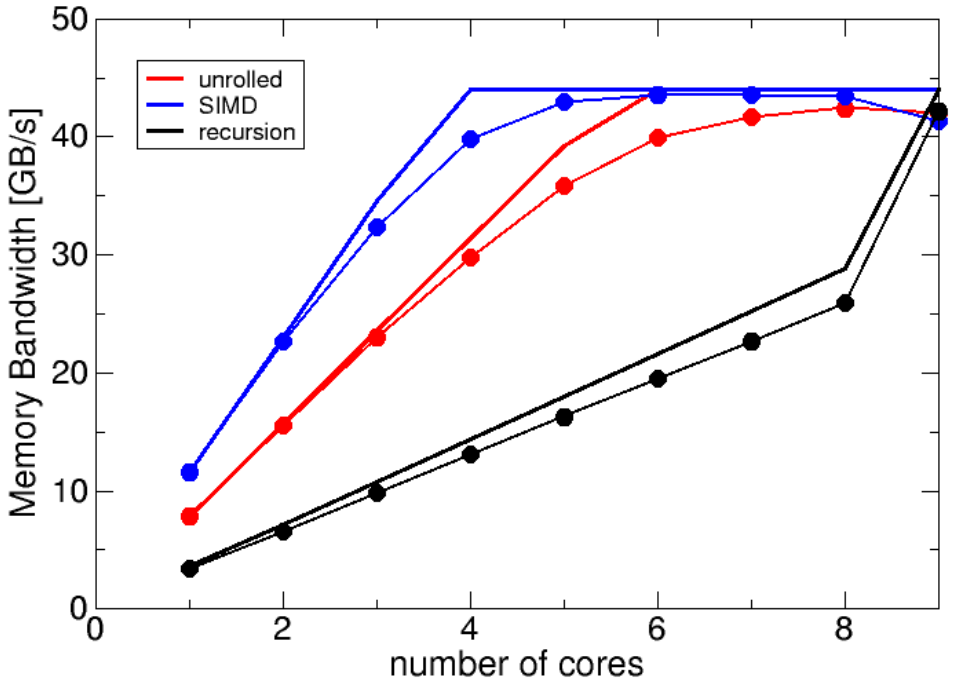
Total runtime with data loaded from memory:

Scalar	48
SSE	42
AVX	39

SIMD only effective if runtime is dominated by instructions execution!

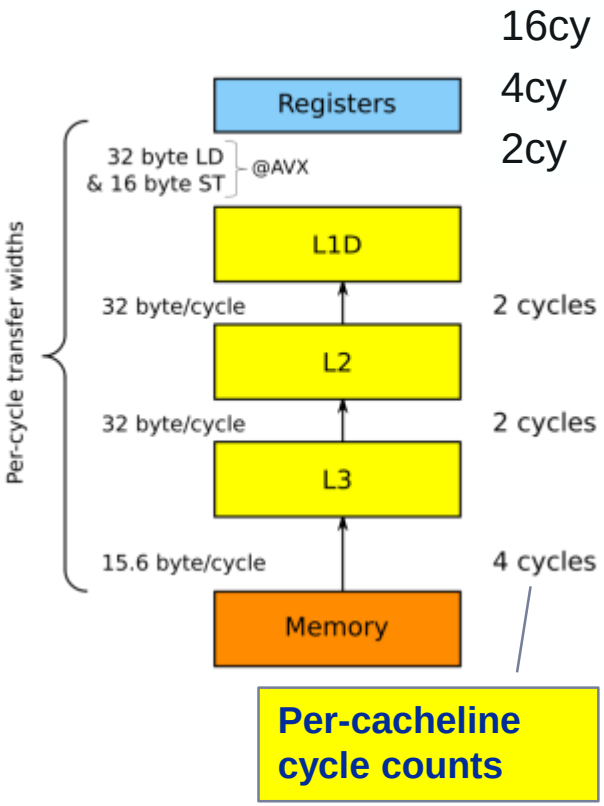
Limits of SIMD processing

- Only part of application may be vectorized, arithmetic vs. load/store (Amdahls law), data transfers
- Memory saturation often makes SIMD obsolete



ry

le solution:
e cache
idth



Per-cacheline cycle counts

Rules for vectorizable loops

1. Countable
2. Single entry and single exit
3. Straight line code
4. No function calls (exception intrinsic math functions)

Better performance with:

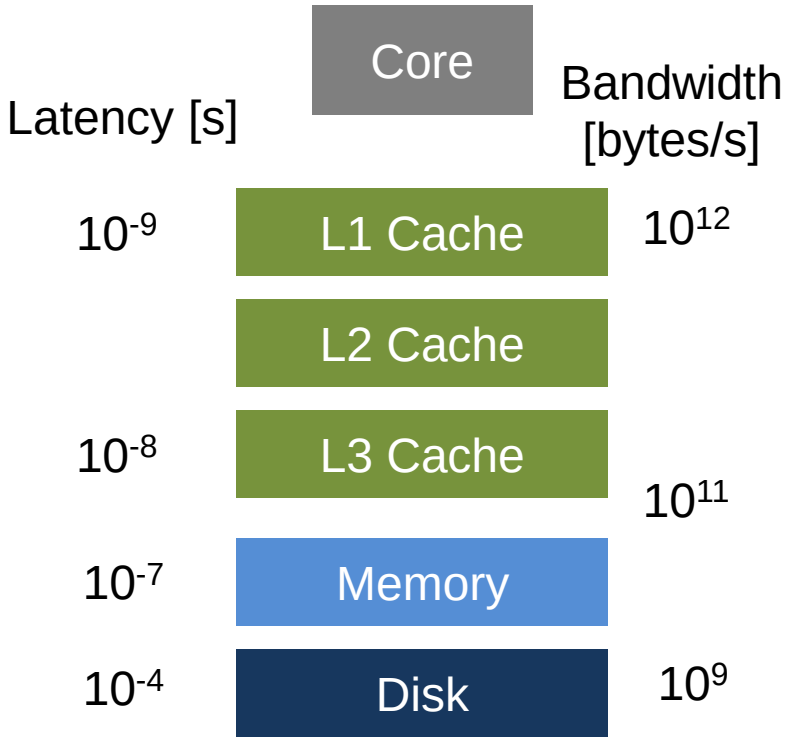
1. Simple inner loops with unit stride
2. Minimize indirect addressing
3. Align data structures (SSE 16bytes, AVX 32bytes)
4. In C use the restrict keyword for pointers to rule out aliasing

Obstacles for vectorization:

- Non-contiguous memory access
- Data dependencies

Memory hierarchy

You can **either** build a *small* und *fast* memory **or** a *large* and *slow* memory.



Purpose of many optimizations is therefore to load data mostly from fast memory layers.

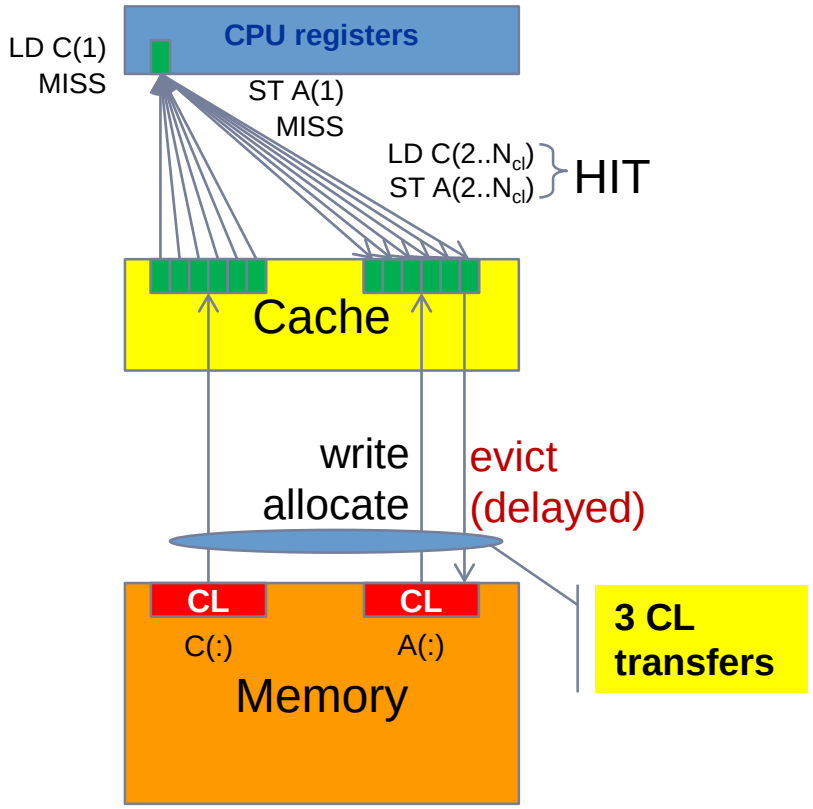
Registers and caches: Data transfers in a memory hierarchy

How does data travel from memory to the CPU and back?

Remember: Caches are organized in **cache lines** (e.g., 64 bytes)
Only **complete cache lines** are transferred between memory hierarchy levels (except registers)

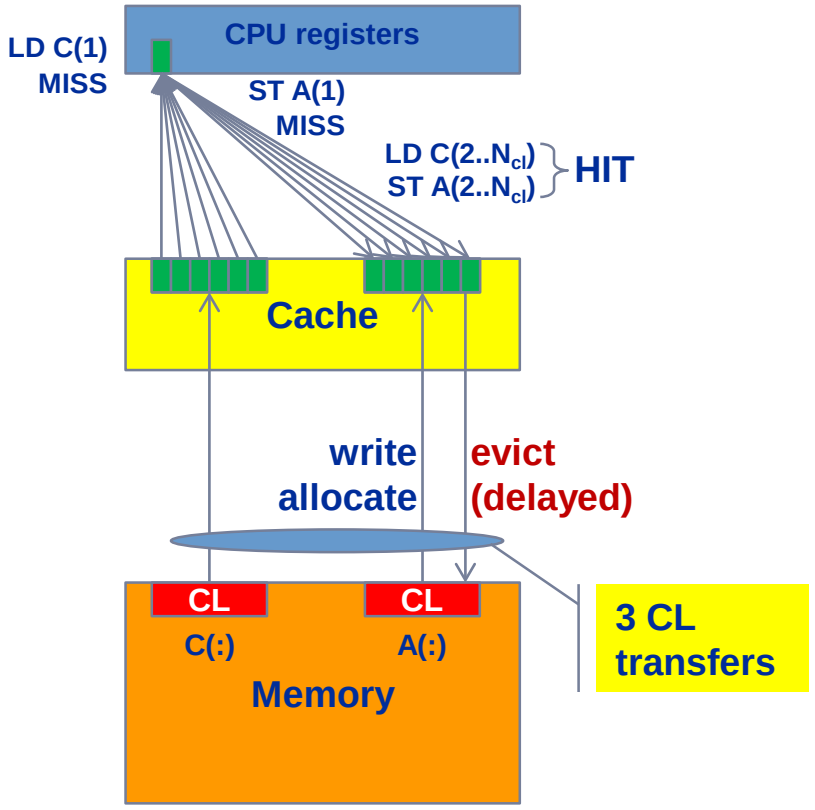
MISS: Load or store instruction does not find data in a cache level
→ CL transfer required

Example: Array copy **`A(:)=C(:)`**

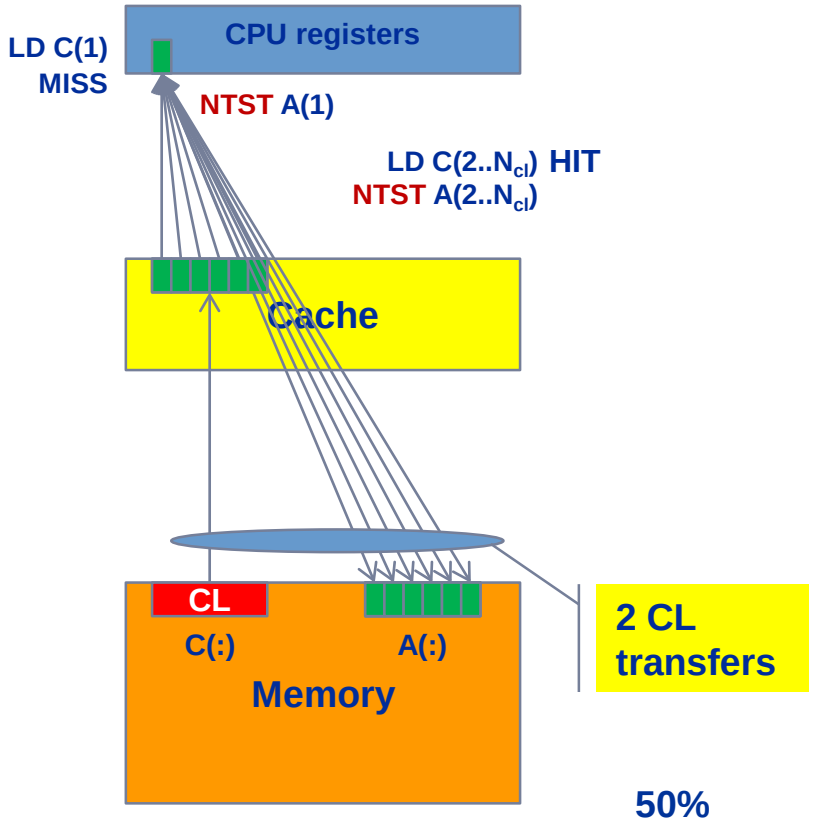


Recap: Data transfers in a memory hierarchy

- How does data travel from memory to the CPU and back?
- Example: Array copy `A(:)= C(:)`



Standard stores



Nontemporal (NT) stores

50% performance boost for COPY

Fusion: SIMD and the memory hierarchy

SIMD optimizations often also involves data structure changes:

- Enable **block wise** load and store.
- Reduce runtime contribution from data transfers by **blocking.** Load or store data at least from L2 cache. Promote temporal and spatial data access locality
- Promote good use of **hardware prefetcher**. Long streaming data access patterns.

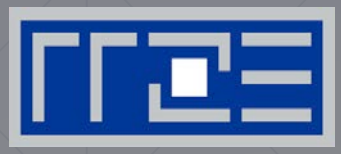
- Above requirements may collide with object oriented programming paradigm: **array of structures** vs **structure of arrays**

Conclusions about core architectures

- All efforts are targeted on increasing **instruction throughput**
- Every hardware optimization puts an **assumption** against the executed software
- One can distinguish transparent and **explicit** solutions

- Common technologies:
  - Instruction level parallelism (**ILP**)
  - Data parallel execution (**SIMD**), does not affect instruction throughput
  - Exploit temporal data access locality (**Caches**)
  - Hide data access latencies (**Prefetching**)
  - Avoid hazards

PRELUDE:
SCALABILITY 4 THE WIN!



Scalability Myth: Code scalability is the key issue

Lore 1

In a world of highly parallel computer architectures only highly scalable codes will survive

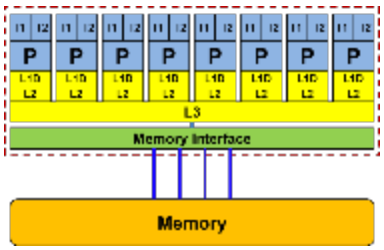
Lore 2

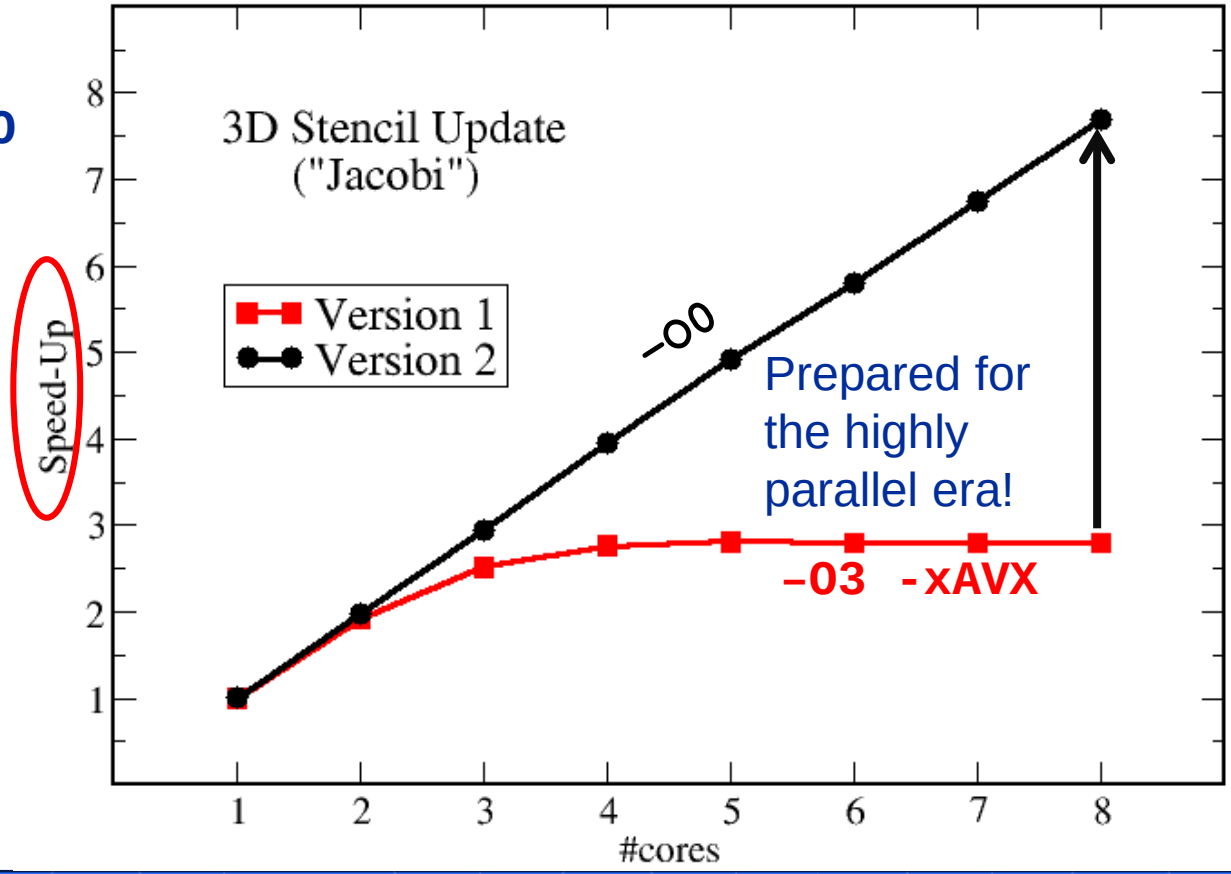
Single core performance no longer matters since we have so many of them and use scalable codes

Scalability Myth: Code scalability is the key issue

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

Changing only the compile options makes this code scalable on an 8-core chip

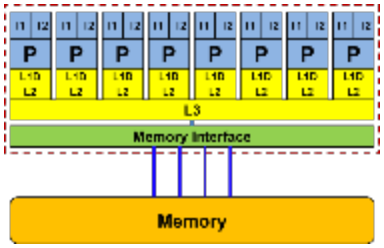


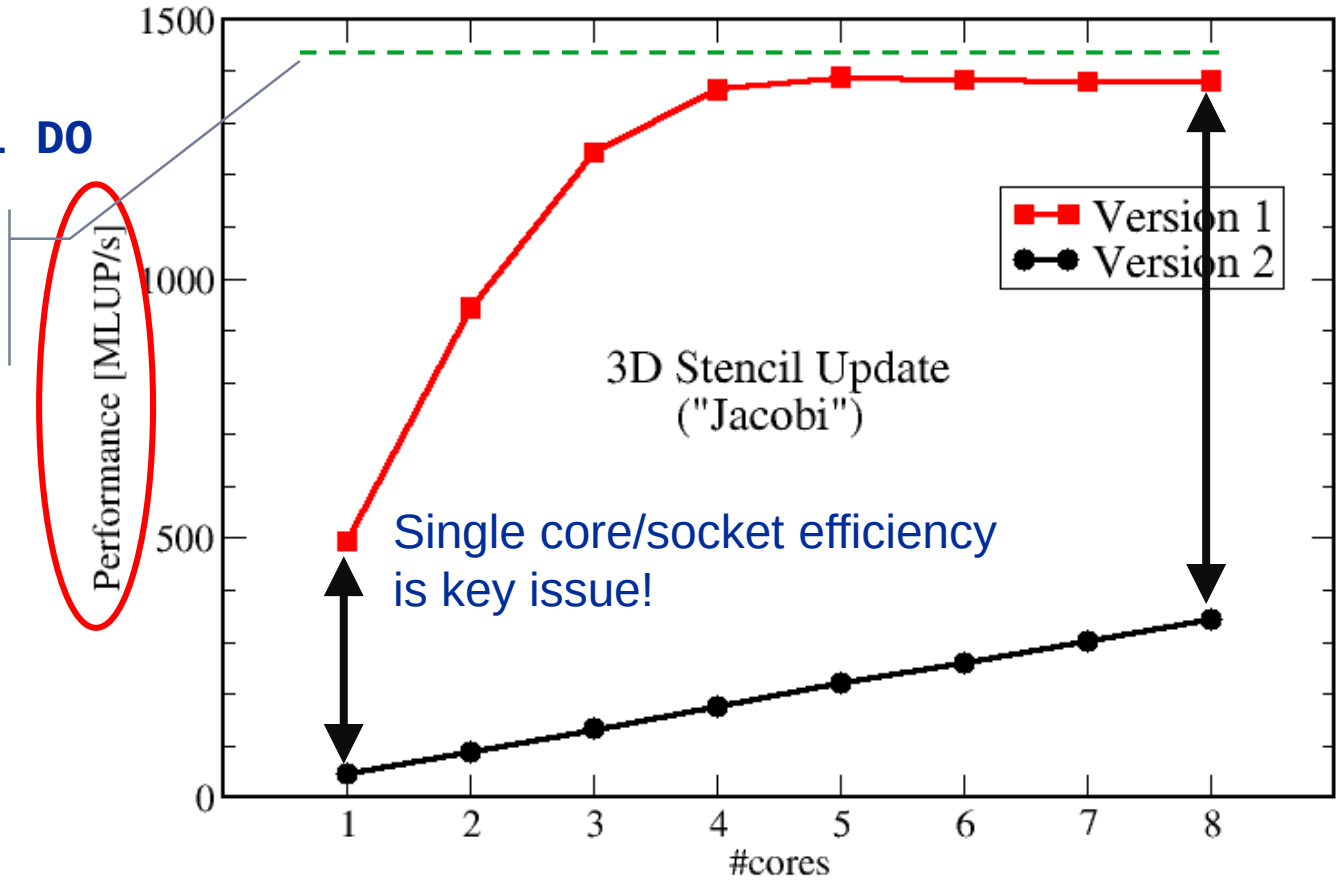


Scalability Myth: Code scalability is the key issue

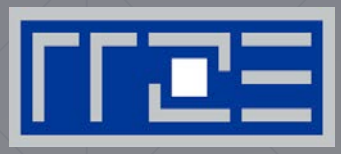
```
!$OMP PARALLEL DO
do k = 1 , Nk
  do j = 1 , Nj; do i = 1 , Ni
     y(i,j,k)= b*(  x(i-1,j,k)+ x(i+1,j,k)+ x(i,j-1,k)+
                    x(i,j+1,k)+ x(i,j,k-1)+ x(i,j,k+1))
    enddo; enddo
enddo
!$OMP END PARALLEL DO
```

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





TOPOLOGY OF MULTI-CORE / MULTI-SOCKET SYSTEMS



- Chip Topology
- Node Topology
- Memory Organisation

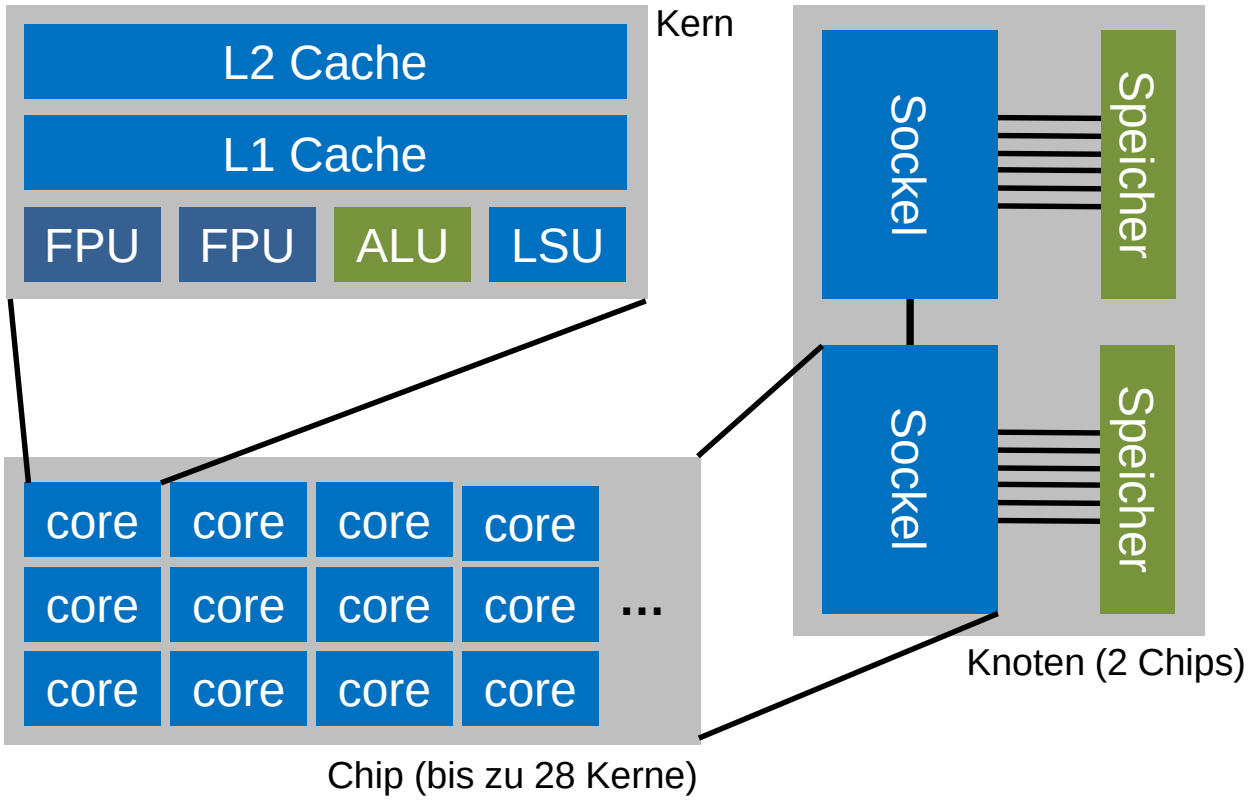
Building blocks for multi-core compute nodes

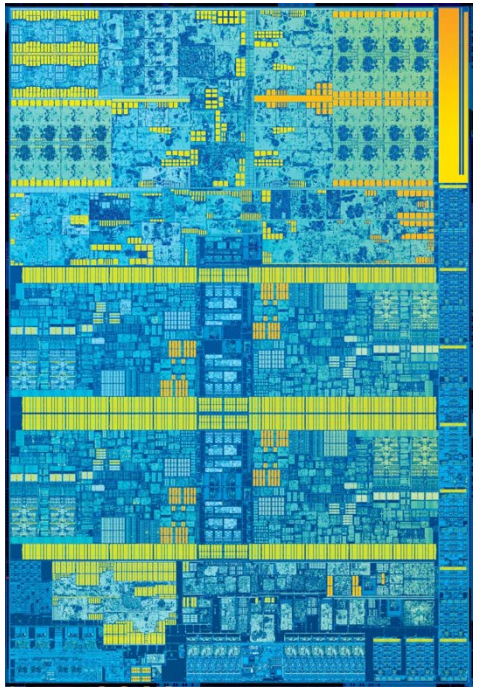
- **Core**: Unit reading and executing instruction stream
- **Chip**: One integrated circuit die
- **Socket**/Package: May consist of multiple chips

- Memory Hierarchy:
  - Private caches
  - Shared caches
  - **ccNUMA**: Replicated memory interfaces

Multicore architecture

Mehrkern-Architekturen

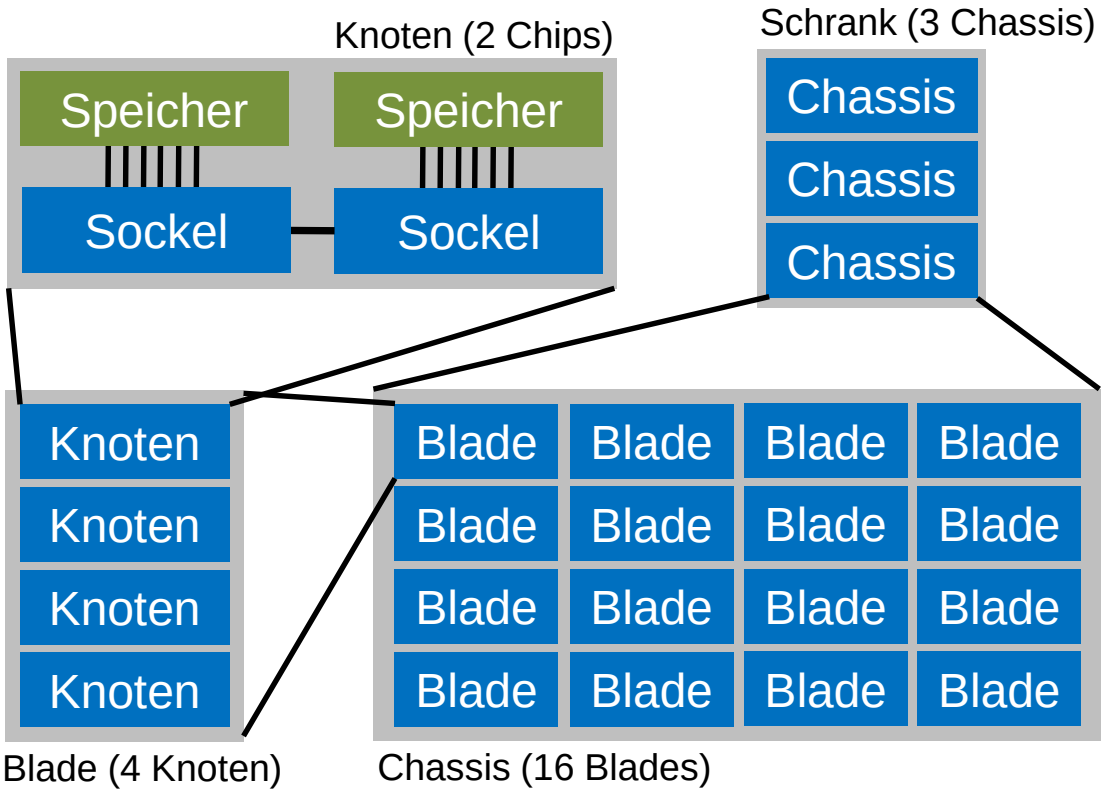




Ca. 8 Mrd. Transistoren auf 500 mm^2

© Intel

Topology of Super computers

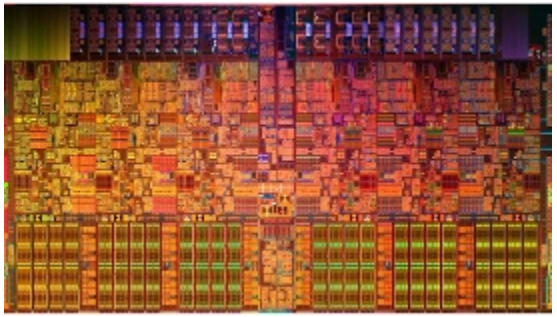


SuperMUC © LRZ

Ein System besteht aus **vielen** Schränken!

Chip Topologies

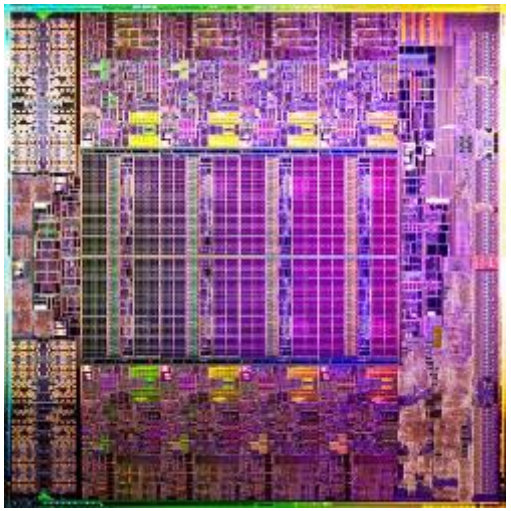
- Separation into core and uncore
- Memory hierarchy holding together the chip design
- L1 (L2) private caches
- L3 cache shared (LLC)



Westmere-EP, 6C, 32nm 248mm^2

- Serialized LLC → not scalable

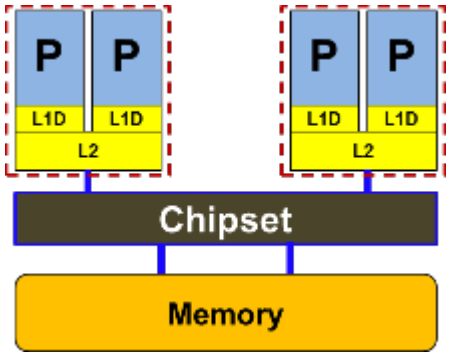
- Segmented ring bus, distributed LLC → scalable design



SandyBridge-EP, 8C, 32nm 435mm^2

From UMA to ccNUMA Memory architectures

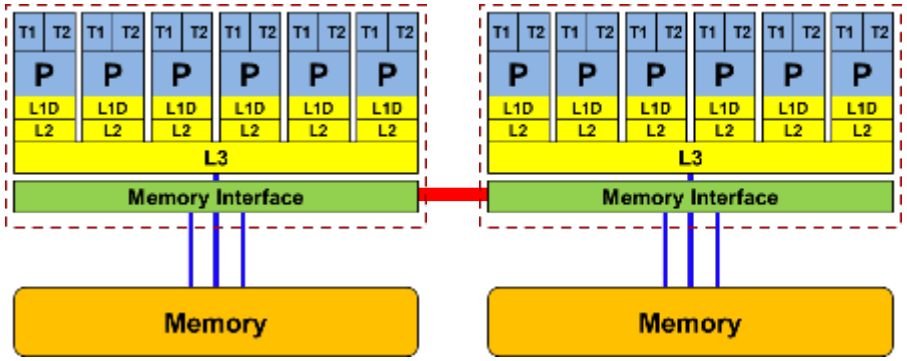
Yesterday (2006): Dual-socket Intel "Core2" node:



Uniform Memory Architecture (UMA)

Flat memory ; symmetric MPs

Today: Dual-socket Intel (Westmere,…) node:



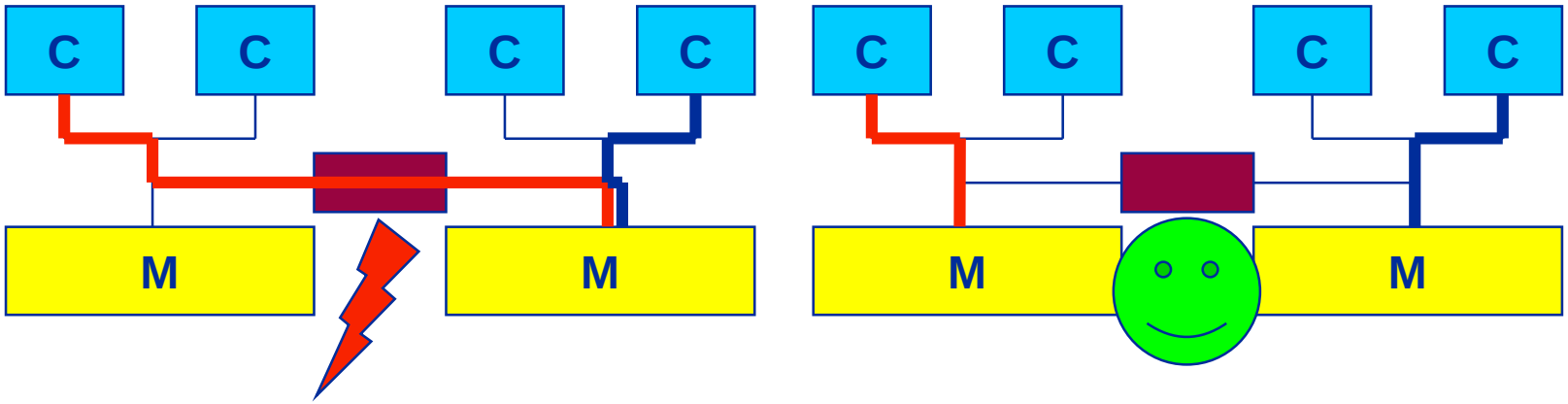
Cache-coherent Non-Uniform Memory Architecture (**ccNUMA**)

HT / QPI provide scalable bandwidth at the price of ccNUMA architectures, but:
Where does my data finally end up?

ccNUMA performance problems

"The other affinity" to care about

- **ccNUMA:**
  - Whole memory is transparently accessible by all processors
  - but physically distributed
  - with varying bandwidth and latency
  - and potential contention (shared memory paths)
- **How do we make sure that memory access is always as "local" and "distributed" as possible?**



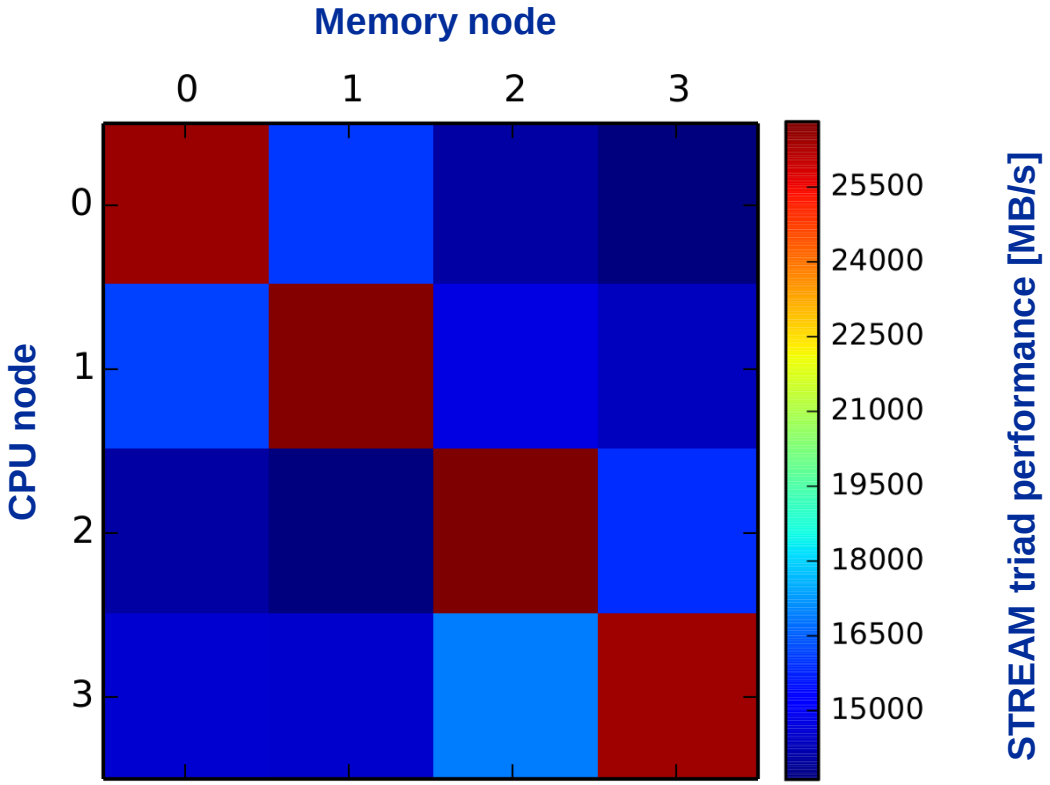
  - Page placement is implemented in units of OS pages (often 4kB, possibly more)

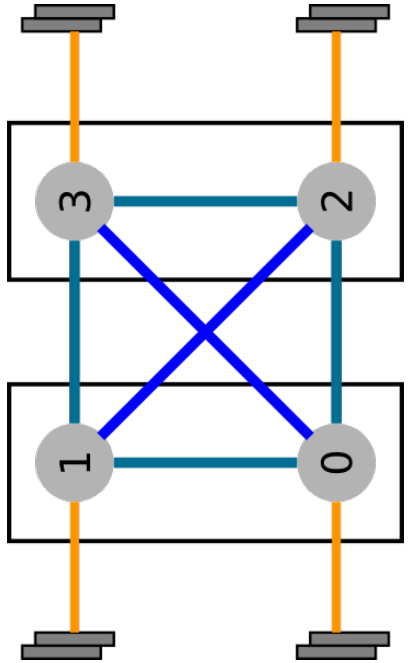
Intel Broadwell EP node

*2 chips, 2 sockets, 11 cores per ccNUMA domain (**CoD** mode)*

ccNUMA map: Bandwidth penalties for remote access

- Run 11 threads per ccNUMA domain (half chip)
- Place memory in different domain → 4x4 combinations
- STREAM copy benchmark using standard stores





ccNUMA default memory locality

"Golden Rule" of ccNUMA:

A memory page gets mapped into the local memory of the processor that first touches it!

- Except if there is not enough local memory available

Caveat: "touch" means "**write**", not "**allocate**"

Example:

```
double *huge = (double*)malloc(N*sizeof(double));

for(i=0; i<N; i++)
   huge[i] = 0.0;
```

Memory not mapped here yet

Mapping takes place here

It is sufficient to touch a single item to map the entire page

Initialization by parallel first touch

Initialize data in parallel to ensure placement into locality domains:

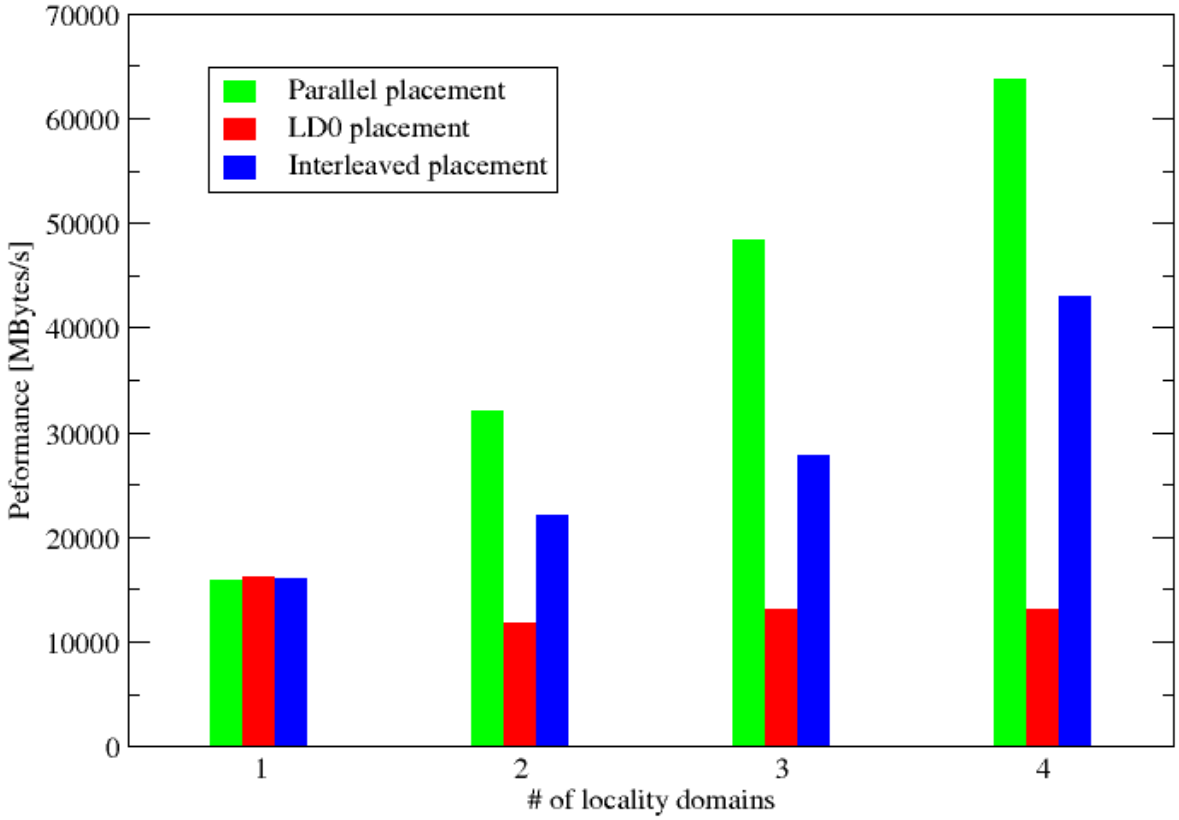
```
double *huge = (double*)malloc(N*sizeof(double));
// parallel init of data
#pragma omp parallel for schedule(static)
for(i=0; i<N; i++)
   huge[i] = 0.0;
// ...

// actual work done on data
#pragma omp parallel for reduction(+:sum) schedule(static)
for(i=0; i<N; i++)
   sum += huge[i];
```

The curse and blessing of interleaved placement:
OpenMP STREAM on a Cray XE6 Interlagos node

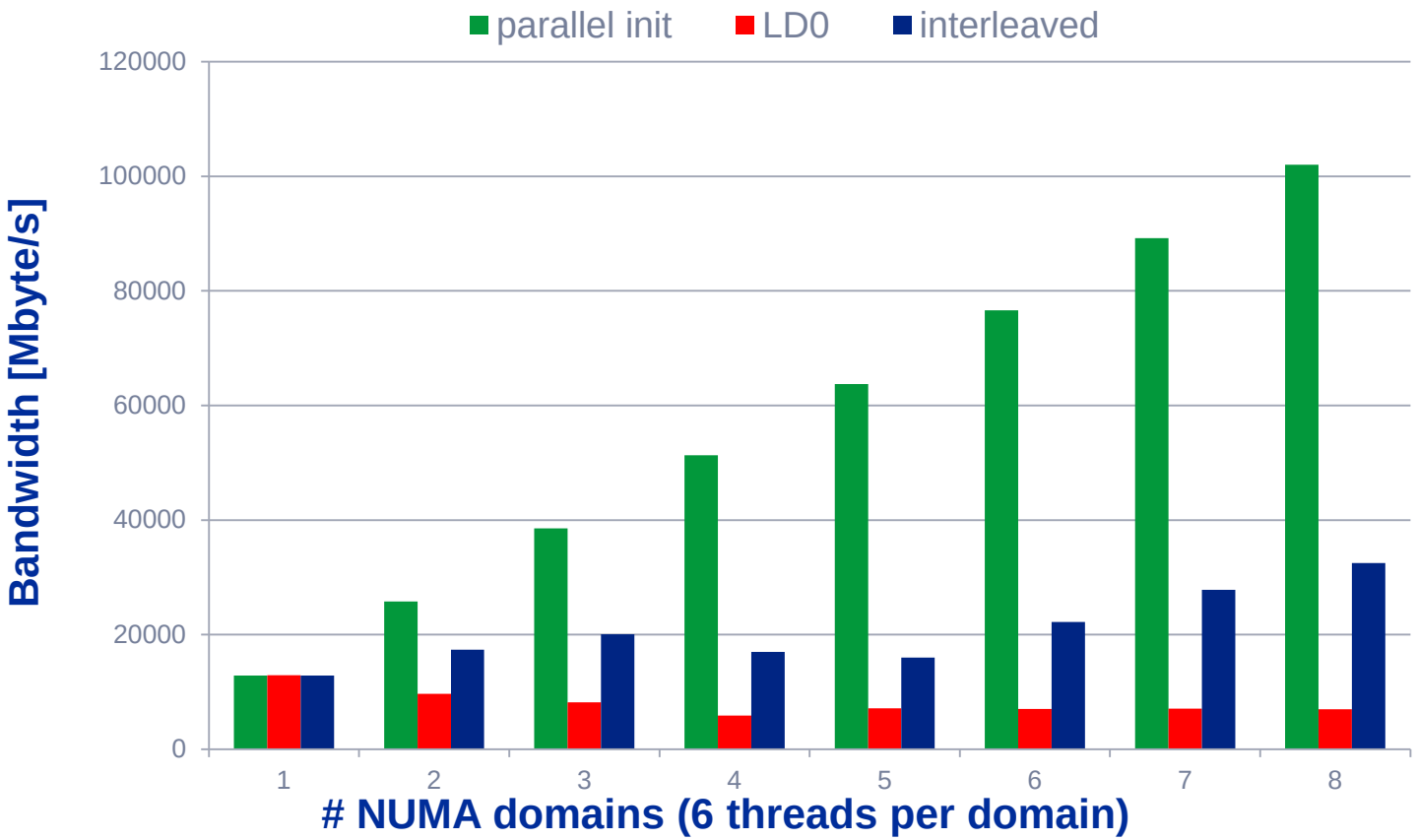
Parallel init: Correct parallel initialization

LD0: Force data into LD0 via **`numactl –m 0`**

Interleaved: **`numactl --interleave <LD range>`**



The curse and blessing of interleaved placement:
same on 4-socket (48 core) Magny Cours node



Conclusions about Node Topologies

Modern computer architecture has a **rich "topology"**

Node-level **hardware parallelism** takes many forms

- Sockets/devices – CPU: 1-8, GPGPU: 1-6
- Cores – moderate (CPU: 4-16) to massive (GPGPU: 1000's)
- SIMD – moderate (CPU: 2-8) to massive (GPGPU: 10's-100's)

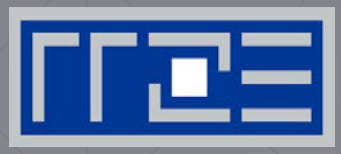
Exploiting performance: **parallelism + bottleneck awareness**

- **"High Performance Computing" == computing at a bottleneck**

Performance of programs is sensitive to architecture

- Topology/affinity influences overheads of popular programming models
- Standards do not contain (many) topology-aware features
  - Things are starting to improve slowly (MPI 3.0, OpenMP 4.0)
- Apart from overheads, performance features are largely independent of the programming model

MULTICORE PERFORMANCE AND TOOLS
PROBING NODE TOPOLOGY



- Standard tools
- likwid-topology

Tools for Node-level Performance Engineering

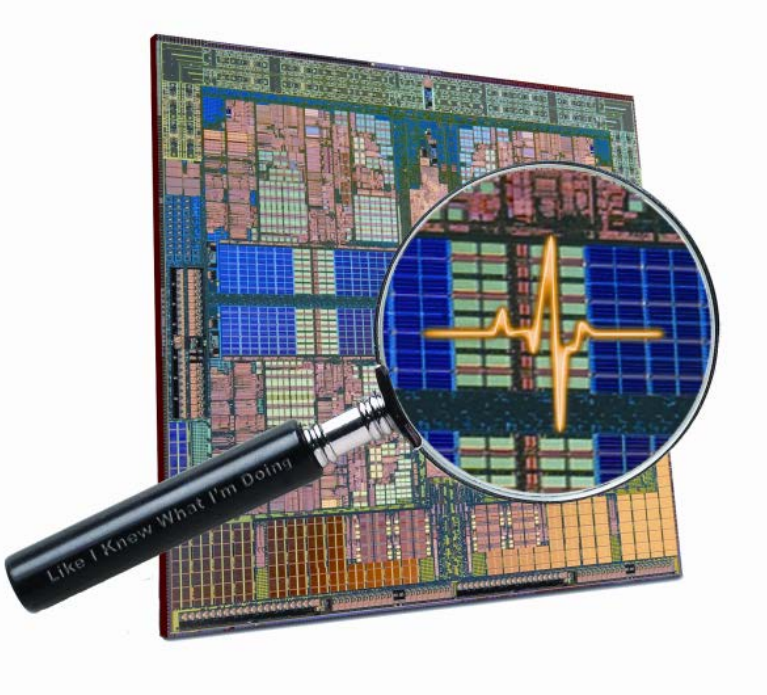
- Gather Node Information
  hwloc, likwid-topology, likwid-powermeter
- Affinity control and data placement
  OpenMP and MPI runtime environments, hwloc, numactl, likwid-pin
- Runtime Profiling
  Compilers, gprof, HPC Toolkit, …
- Performance Profilers
  Intel Vtune™, likwid-perfctr, PAPI based tools, Linux perf, …
- Microbenchmarking
  STREAM, likwid-bench, lmbench

How do we figure out the node topology?

LIKWID tool suite:

Like
I
Knew
What
I'm
Doing

Open source tool collection (developed at RRZE):
http://code.google.com/p/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**
- **http://arxiv.org/abs/1004.4431**

Likwid Tool Suite

- Command line tools for Linux:
  - easy to install
  - works with standard linux kernel
  - simple and clear to use
  - supports Intel and AMD CPUs

- Current tools:
  - **likwid-topology**: Print thread and cache topology
  - **likwid-pin**: Pin threaded application without touching code
  - **likwid-perfctr:** Measure performance counters
  - **likwid-powermeter**: Query turbo mode steps. Measure ETS.
  - **likwid-bench**: Low-level bandwidth benchmark generator tool

Output of `likwid-topology -g`

on one node of Intel Haswell-EP

```
--------------------------------------------------------------------------------
CPU name:       Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz
CPU type:       Intel Xeon Haswell EN/EP/EX processor
CPU stepping:   2
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                        2
Cores per socket:               14
Threads per core:               2
--------------------------------------------------------------------------------
HWThread        Thread          Core            Socket          Available
0               0               0               0               *
              0 1               0               *

...
43               1              1               1               *
44               1              2               1               *
--------------------------------------------------------------------------------
Socket 0:       ( 0 28 1 29 2 30 3 31 4 32 5 33 6 34 7 35 8 36 9 37 10 38 11 39 12 40 13 41 )
Socket 1:       ( 14 42 15 43 16 44 17 45 18 46 19 47 20 48 21 49 22 50 23 51 24 52 25 53 26 54 27 55 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level:                          1
Size:                           32 kB
Cache groups:   ( 0 28 ) ( 1 29 ) ( 2 30 ) ( 3 31 ) ( 4 32 ) ( 5 33 ) ( 6 34 ) ( 7 35 ) ( 8 36 ) ( 9 37 ) ( 10 38 ) ( 11 39 ) ( 12 40 ) ( 13 41
) ( 14 42 ) ( 15 43 ) ( 16 44 ) ( 17 45 ) ( 18 46 ) ( 19 47 ) ( 20 48 ) ( 21 49 ) ( 22 50 ) ( 23 51 ) ( 24 52 ) ( 25 53 ) ( 26 54 ) ( 27 55 )
--------------------------------------------------------------------------------
Level:                          2
Size:                           256 kB
Cache groups:   ( 0 28 ) ( 1 29 ) ( 2 30 ) ( 3 31 ) ( 4 32 ) ( 5 33 ) ( 6 34 ) ( 7 35 ) ( 8 36 ) ( 9 37 ) ( 10 38 ) ( 11 39 ) ( 12 40 ) ( 13 41
) ( 14 42 ) ( 15 43 ) ( 16 44 ) ( 17 45 ) ( 18 46 ) ( 19 47 ) ( 20 48 ) ( 21 49 ) ( 22 50 ) ( 23 51 ) ( 24 52 ) ( 25 53 ) ( 26 54 ) ( 27 55 )
--------------------------------------------------------------------------------
Level:                          3
Size:                           17 MB
Cache groups:   ( 0 28 1 29 2 30 3 31 4 32 5 33 6 34 ) ( 7 35 8 36 9 37 10 38 11 39 12 40 13 41 ) ( 14 42 15 43 16 44 17 45 18 46 19 47 20 48 )
( 21 49 22 50 23 51 24 52 25 53 26 54 27 55 )
--------------------------------------------------------------------------------
```

All physical processor IDs

Output of likwid-topology continued

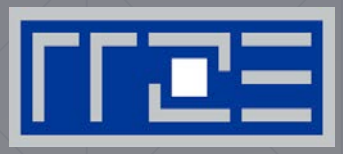
```
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:                   4
--------------------------------------------------------------------------------
Domain:                         0
Processors:             ( 0 28 1 29 2 30 3 31 4 32 5 33 6 34 )
Distances:                      10 21 31 31
Free memory:                    13292.9 MB
Total memory:                   15941.7 MB
--------------------------------------------------------------------------------
Domain:                         1
Processors:             ( 7 35 8 36 9 37 10 38 11 39 12 40 13 41 )
Distances:                      21 10 31 31
Free memory:                    13514 MB
Total memory:                   16126.4 MB
--------------------------------------------------------------------------------
Domain:                         2
Processors:             ( 14 42 15 43 16 44 17 45 18 46 19 47 20 48 )
Distances:                      31 31 10 21
Free memory:                    15025.6 MB
Total memory:                   16126.4 MB
--------------------------------------------------------------------------------
Domain:                         3
Processors:             ( 21 49 22 50 23 51 24 52 25 53 26 54 27 55 )
Distances:                      31 31 21 10
Free memory:                    15488.9 MB
Total memory:                   16126 MB
--------------------------------------------------------------------------------
```

Output of likwid-topology continued

Cluster on die mode and SMT enabled!

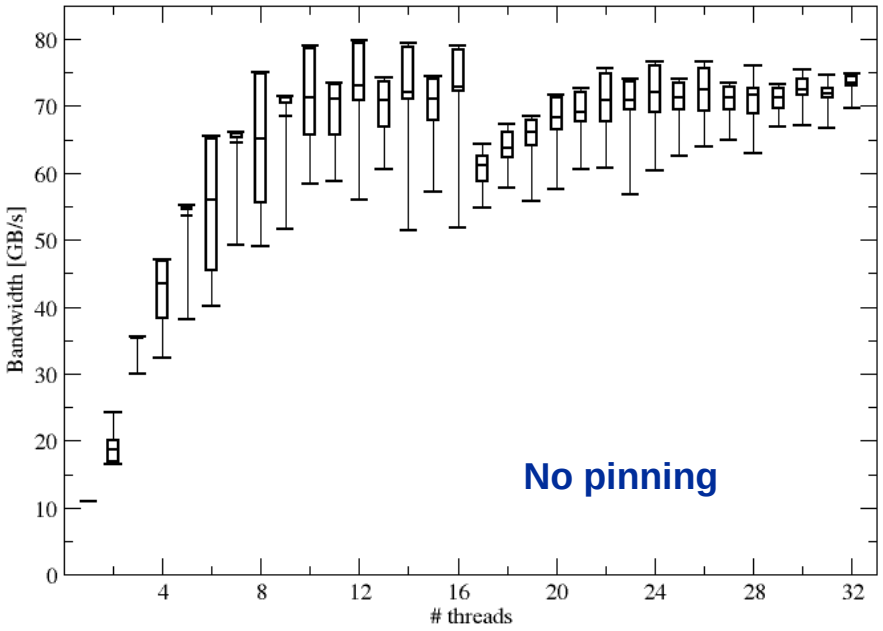
```
********************************************************************************
Graphical Topology
********************************************************************************
Socket 0:
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  0 28  | |  1 29  | |  2 30  | |  3 31  | |  4 32  | |  5 33  | |  6 34  | |  7 35  | |  8 36  | |  9 37  | | 10 38  | | 11 39  | | 12 40  | | 13 41  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +-------------------------------------------------------------------------------+ +-------------------------------------------------------------------------------+ |
| |                                      17MB                                     | |                                      17MB                                     | |
| +-------------------------------------------------------------------------------+ +-------------------------------------------------------------------------------+ |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------+
Socket 1:
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| | 14 42  | | 15 43  | | 16 44  | | 17 45  | | 18 46  | | 19 47  | | 20 48  | | 21 49  | | 22 50  | | 23 51  | | 24 52  | | 25 53  | | 26 54  | | 27 55  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +-------------------------------------------------------------------------------+ +-------------------------------------------------------------------------------+ |
| |                                      17MB                                     | |                                      17MB                                     | |
| +-------------------------------------------------------------------------------+ +-------------------------------------------------------------------------------+ |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

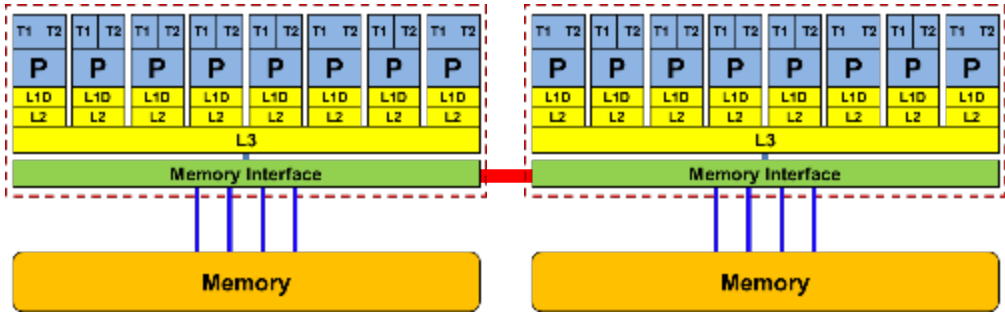
ENFORCING THREAD/PROCESS-CORE AFFINITY UNDER THE LINUX OS



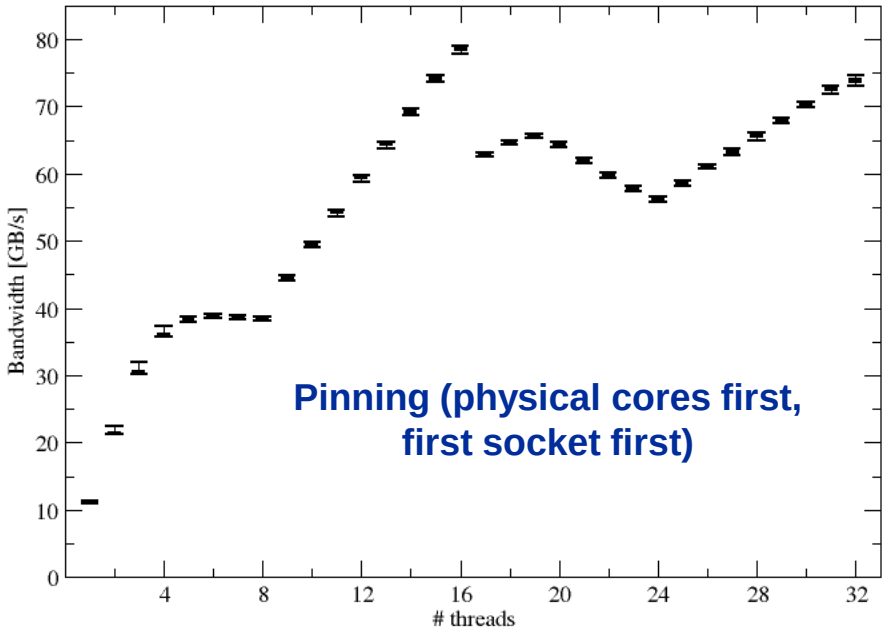
- Standard tools and OS affinity facilities under program control
- likwid-pin

Example: STREAM benchmark on 16-core Sandy Bridge:
Anarchy vs. thread pinning





- **There are several reasons for caring about affinity:**
- **Eliminating performance variation**
- **Making use of architectural features**
- **Avoiding resource contention**
- **Benchmark how code reacts to variations**



More thread/Process-core affinity (“pinning”) options

- Highly OS-dependent system calls
  - But available on all systems

    Linux: `sched_setaffinity()`
    Windows: `SetThreadAffinityMask()`

- Hwloc project (http://www.open-mpi.de/projects/hwloc/)
- Support for “semi-automatic” pinning in some compilers/environments
  - All modern compilers with OpenMP support
  - Generic Linux: **`taskset`**, **`numactl`**, **`likwid-pin`** (see below)
  - OpenMP 4.0 (see any recent OpenMP/hybrid programming tutorial)
- Affinity awareness in MPI libraries
  - SGI MPT
  - OpenMPI
  - Intel MPI
  - …

Likwid-pin

Overview

- Pins processes and threads to specific cores without touching code
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Based on combination of wrapper tool together with overloaded pthread library → binary must be dynamically linked!
- Can also be used as a superior replacement for taskset
- Supports logical core numbering within a node

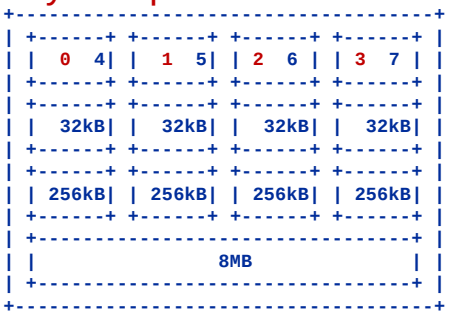
- Usage examples:
  - **`likwid-pin -c 0-3,4,6  ./myApp parameters`**
  - **`likwid-pin -c S0:0-7  ./myApp parameters`**
  - **`likwid-pin –c N:0-15 ./myApp parameters`**

LIKWID terminology

Thread group syntax

- The OS numbers all processors (hardware threads) on a node
- The numbering is enforced at boot time by the BIOS
- LIKWID introduces thread groups consisting of processors sharing a topological entity (e.g. socket or shared cache)
- A thread group is defined by a single character + index

Physical processors first!



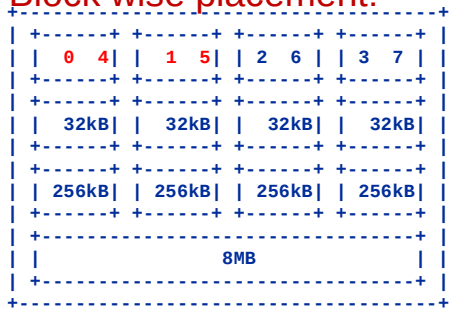
- Example for likwid-pin:

```
likwid-pin –c S1:0-3,6,7 ./a.out
```

- Thread group expression may be chained with @:

```
likwid-pin –c S0:0-3@S1:0-3 ./a.out
```

Block wise placement!



- Alternative expression based syntax:

```
likwid-pin –c E:S0:4:2:2 ./a.out
```

```
E:<thread domain>:<num threads>:<chunk size>:<stride>
```

- Xeon Phi: `likwid-pin –c E:N:60:2:4 ./a.out`

Likwid

Currently available thread domains

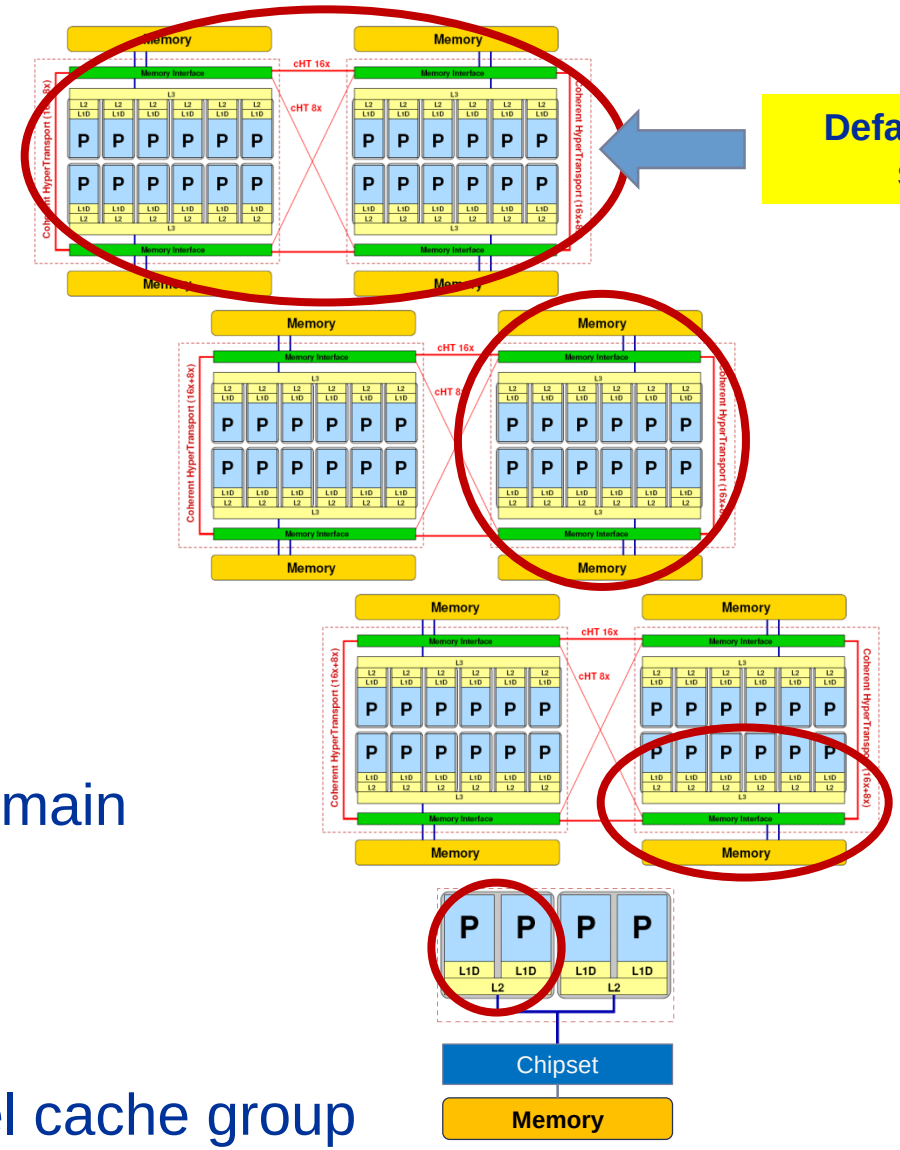
Possible unit prefixes

N node

S socket

M NUMA domain

C outer level cache group



Default if –c is not specified!

Likwid-pin

Example: Intel OpenMP

Running the STREAM benchmark with likwid-pin:

```
$ likwid-pin -c S0:0-3 ./stream
[likwid-pin] Main PID -> core 0 - OK
-------------------------------------------------
 Double precision appears to have 16 digits of accuracy
 Assuming 8 bytes per DOUBLE PRECISION word
-------------------------------------------------
 Array size =   20000000
 Offset     =         32
 The total memory requirement is  457 MB
 You are running each test  10 times
 --
 The *best* time for each test is used
 *EXCLUDING* the first and last iterations
[pthread wrapper] [pthread wrapper] PIN_MASK: 0->1  1->2  2->3
[pthread wrapper] SKIP MASK: 0x1
        threadid 140370139711232 -> SKIP
        threadid 140370117211968 -> core 1 - OK
        threadid 140370113013632 -> core 2 - OK
        threadid 140369974597568 -> core 3 - OK

  [... rest of STREAM output omitted ...]
```

Main PID always pinned

Skip shepherd thread

Pin all spawned threads in turn

1. Runtime profile / Call graph (gprof): Where are the hot spots?
2. Instrument hot spots (prepare for detailed measurement)
3. Find performance signatures

Possible signatures:

- Bandwidth saturation
- Instruction throughput limitation (real or language-induced)
- Latency impact (irregular data access, high branch ratio)
- Load imbalance
- ccNUMA issues (data access across ccNUMA domains)
- Pathologic cases (false cacheline sharing, expensive operations)

likwid-perfctr can help here

Goal: Come up with educated guess about a performance-limiting motif (Performance Pattern)

Probing performance behavior

- How do we find out about the performance properties and requirements of a parallel code?
  - Profiling via advanced tools is often overkill
- A coarse overview is often sufficient
  - likwid-perfctr (similar to “perfex” on IRIX, “hpmcount” on AIX, “lipfpm” on Linux/Altix)
  - Simple end-to-end measurement of hardware performance metrics
  - “Marker” API for starting/stopping counters
  - Multiple measurement region support
  - Preconfigured and extensible metric groups, list with **`likwid-perfctr -a`**

```
BRANCH: Branch prediction miss rate/ratio
CACHE: Data cache miss rate/ratio
CLOCK: Clock of cores
DATA: Load to store ratio
FLOPS_DP: Double Precision MFlops/s
FLOPS_SP: Single Precision MFlops/s
FLOPS_X87: X87 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
```

likwid-perfctr

Example usage with preconfigured metric group

```
$ likwid-perfctr -g L2 -C S1:0-3 ./a.out
--------------------------------------------------------------------------------
CPU name:       Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz [...]
--------------------------------------------------------------------------------
<<<< PROGRAM OUTPUT >>>>
--------------------------------------------------------------------------------
Group 1: L2
+----------------------+---------+------------+------------+------------+------------+
|        Event         | Counter |   Core 14  |   Core 15  |   Core 16  |   Core 17  |
+----------------------+---------+------------+------------+------------+------------+
|   INSTR_RETIRED_ANY  |  FIXC0  | 1298031144 | 1965945005 | 1854182290 | 1862521357 |
| CPU_CLK_UNHALTED_CORE|  FIXC1  | 2353698512 | 2894134935 | 2894645261 | 2895023739 |
|  CPU_CLK_UNHALTED_REF|  FIXC2  | 2057044629 | 2534405765 | 2535218217 | 2535560434 |
|     L1D_REPLACEMENT  |   PMC0  |  212900444 |  200544877 |  200389272 |  200387671 |
|     L2_TRANS_L1D_WB  |   PMC1  |  112464863 |  99931184  |  99982371  |  99976697  |
|      ICACHE_MISSES   |   PMC2  |    21265   |    26233   |    12646   |    12363   |
+----------------------+---------+------------+------------+------------+------------+
[... statistics output omitted ...]
+--------------------------------+------------+------------+------------+------------+
|             Metric             |   Core 14  |   Core 15  |   Core 16  |   Core 17  |
+--------------------------------+------------+------------+------------+------------+
|       Runtime (RDTSC) [s]      |   1.1314   |   1.1314   |   1.1314   |   1.1314   |
|      Runtime unhalted [s]      |   1.0234   |   1.2583   |   1.2586   |   1.2587   |
|           Clock [MHz]          |  2631.6699 |  2626.4367 |  2626.0579 |  2626.0468 |
|              CPI               |   1.8133   |   1.4721   |   1.5611   |   1.5544   |
|  L2D load bandwidth [MBytes/s] | 12042.7388 | 11343.8446 | 11335.0428 | 11334.9523 |
|  L2D load data volume [GBytes] |   13.6256  |   12.8349  |   12.8249  |   12.8248  |
| L2D evict bandwidth [MBytes/s] |  6361.5883 |  5652.6192 |  5655.5146 |  5655.1937 |
| L2D evict data volume [GBytes] |   7.1978   |   6.3956   |   6.3989   |   6.3985   |
|     L2 bandwidth [MBytes/s]    | 18405.5299 | 16997.9477 | 16991.2728 | 16990.8453 |
|     L2 data volume [GBytes]    |   20.8247  |   19.2321  |   19.2246  |   19.2241  |
+--------------------------------+------------+------------+------------+------------+
```

Always measured

Configured metrics (this group)

Derived metrics

likwid-perfctr

Best practices for runtime counter analysis

Things to look at (in roughly this order)

- Excess work
- Load balance (flops, instructions, BW)
- In-socket memory BW saturation
- Flop/s, loads and stores per flop metrics
- SIMD vectorization
- CPI metric
- # of instructions,
  branches, mispredicted branches

Caveats

- Load imbalance may not show in CPI or # of instructions
  - Spin loops in OpenMP barriers/MPI blocking calls
  - Looking at “top” or the Windows Task Manager does not tell you anything useful
- In-socket performance saturation may have various reasons
- Cache miss metrics are sometimes misleading

likwid-perfctr

Marker API (C/C++ and Fortran)

- A marker API is available to restrict measurements to code regions
- The API only turns counters on/off. The configuration of the counters is still done by **`likwid-perfctr`**
- Multiple named region support, accumulation over multiple calls
- Inclusive and overlapping regions allowed

```
#include <likwid.h>
. . .
LIKWID_MARKER_INIT;                    // must be called from serial region
#pragma omp parallel
{
  LIKWID_MARKER_THREADINIT;            // only reqd. if measuring multiple threads
}
. . .
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
```

- Activate macros with `-DLIKWID_PERFMON`
- Run `likwid-perfctr` with `–m` switch to enable marking
- See https://github.com/RRZE-HPC/likwid/wiki/TutorialMarkerF90 for Fortran example

likwid-perfctr

Compiling, linking, and running with the marker API

Compile:

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

Link:

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

Run:

```
likwid-perfctr -C <MASK> -g <GROUP> -m ./a.out
```

→ One separate block of output for every marked region

→ Caveat: marker API can cause overhead; do not call too frequently!

ERLANGEN REGIONAL COMPUTING CENTER

Basics of Performance Engineering

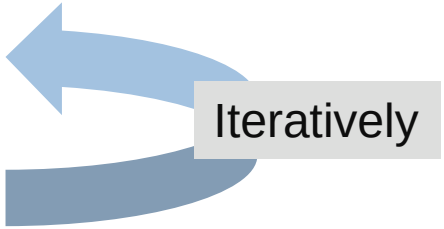
J. Eitzinger

PATC LRZ 2018, 26.3.2018

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Basics of Optimization

1. Define relevant test cases
2. Establish a sensible performance metric
3. Acquire a runtime profile (sequential)
4. Identify hot kernels (Hopefully there are any!)
5. Carry out optimization process for each kernel



Motivation:

- Understand observed performance
- Learn about code characteristics and machine capabilities
- Deliberately decide on optimizations

Best Practices Benchmarking

Preparation

- Reliable timing (Minimum time which can be measured?)
- Document code generation (Flags, Compiler Version)
- Get exclusive System
- System state (Clock, Turbo mode, Memory, Caches)
- Consider to automate runs with a skript (Shell, python, perl)

Doing

- Affinity control
- Check: Is the result reasonable?
- Is result deterministic and reproducible.
- Statistics: Mean, Best ??
- Basic variants: Thread count, affinity, working set size (Baseline!)

Best Practices Benchmarking cont.

Postprocessing

- Documentation
- Try to understand and explain the result
- Plan variations to gain more information
- Many things can be better understood if you plot them (gnuplot, xmgrace)

Focus on resource utilization

1. **Instruction execution**

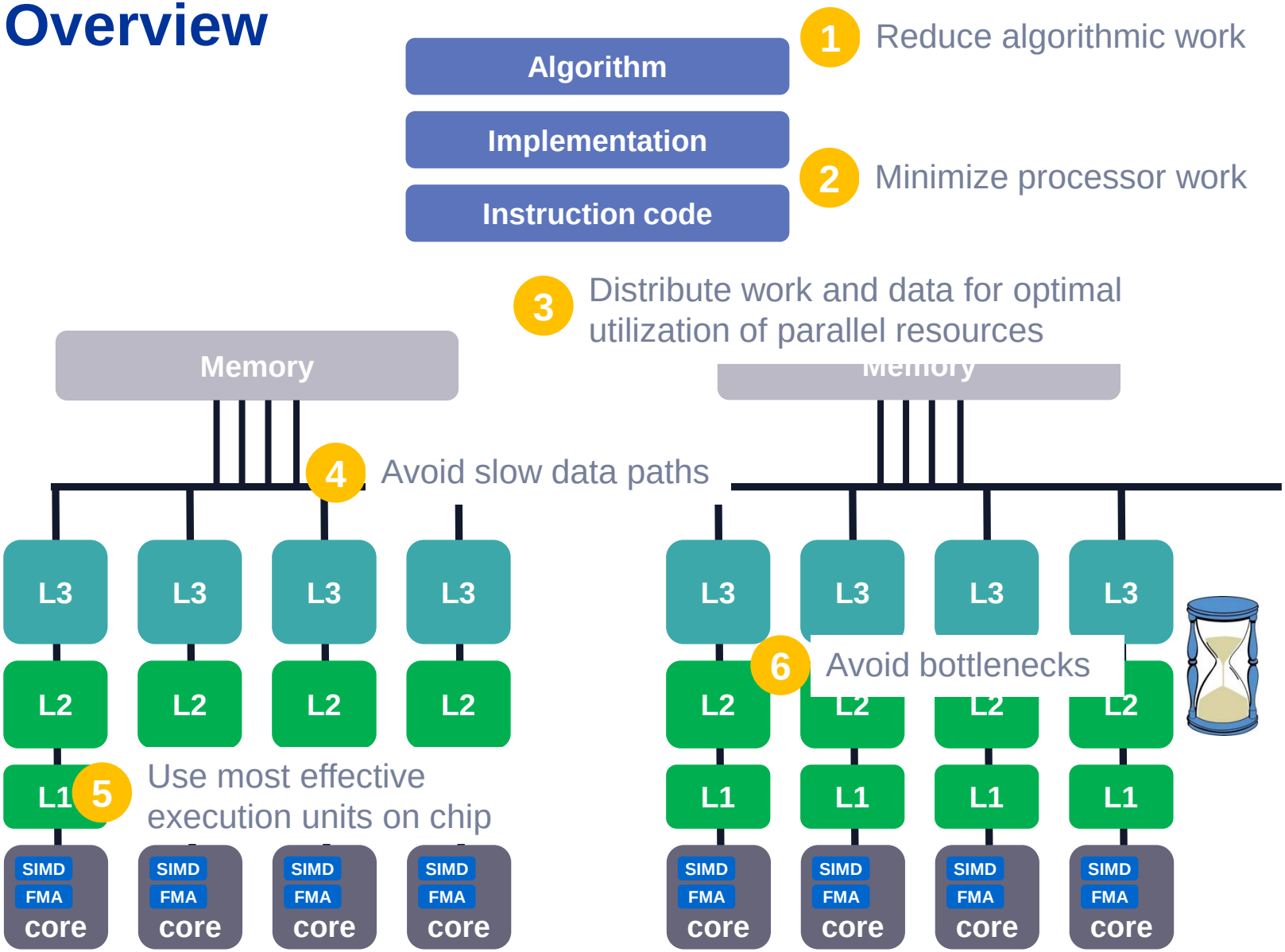
   Primary resource of the processor.

2. **Data transfer bandwidth**

   Data transfers as a consequence of instruction execution.

What is the **limiting resource**?
Do you fully **utilize** available **resources**?

Overview


Algorithm
Implementation
Instruction code
1
Reduce algorithmic work
2
Minimize processor work
3
Distribute work and data for optimal utilization of parallel resources
Memory
Memory
4
Avoid slow data paths
L3
L2
L1
5
Use most effective execution units on chip
SIMD
FMA
core
6
Avoid bottlenecks

Thinking in Bottlenecks

- A bottleneck is a performance limiting setting
- Microarchitectures expose numerous bottlenecks

Observation 1:

Most applications face a single bottleneck at a time!

Observation 2:

There is a limited number of relevant bottlenecks!

Process vs. Tool

Reduce complexity!

We propose a human driven process to enable a systematic way to success!

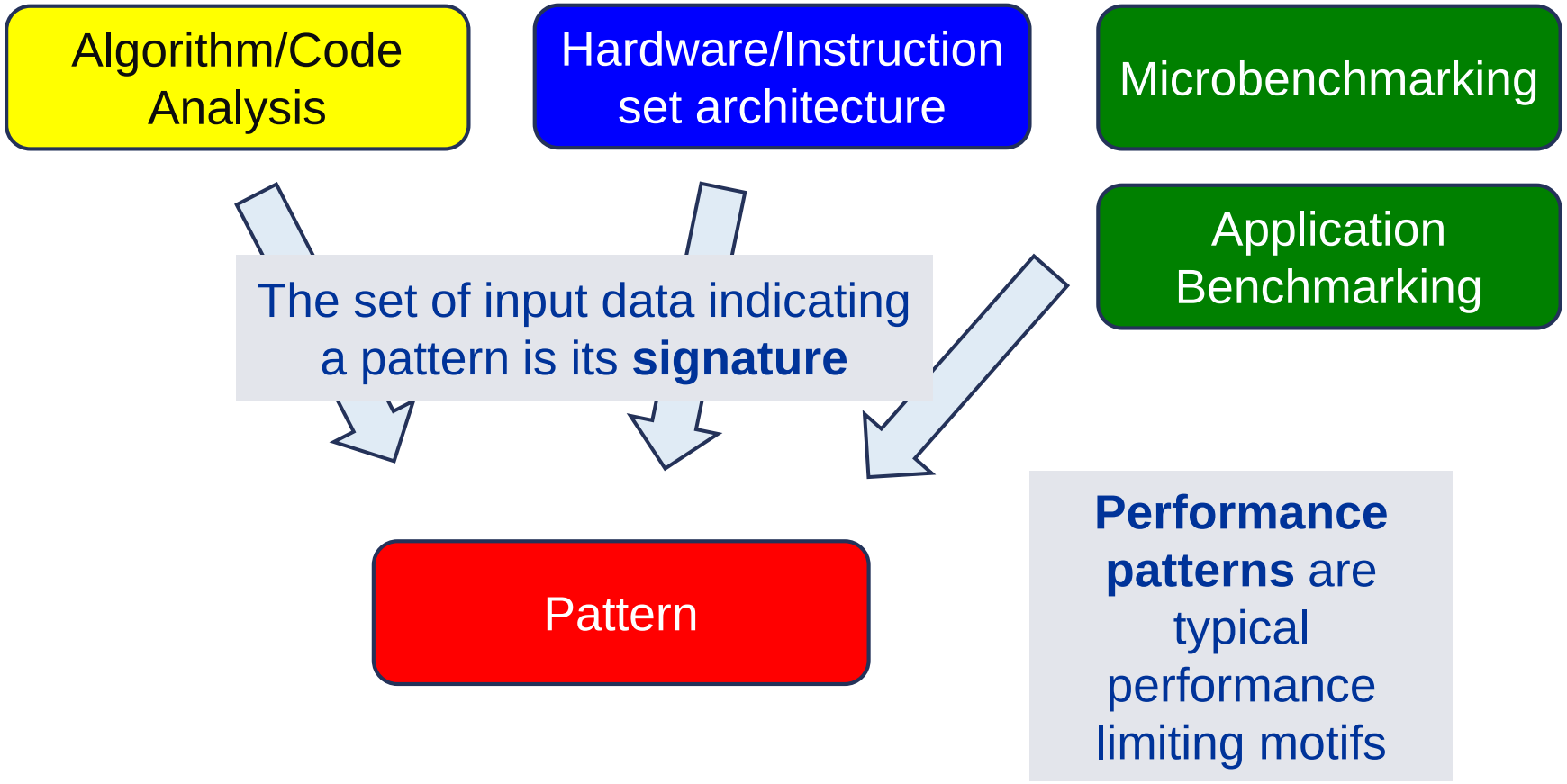
- Executed by humans.
- Uses tools by means of data acquisition only.

Uses one of the most powerful tools available:

Your brain !

You are a investigator making sense of what's going on.

Performance Engineering Process: Analysis



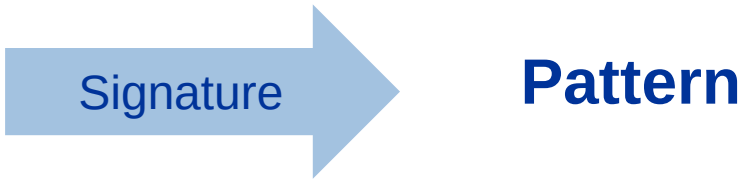
Step 1 **Analysis:** Understanding observed performance

Performance analysis phase

Understand observed performance: **Where am I?**

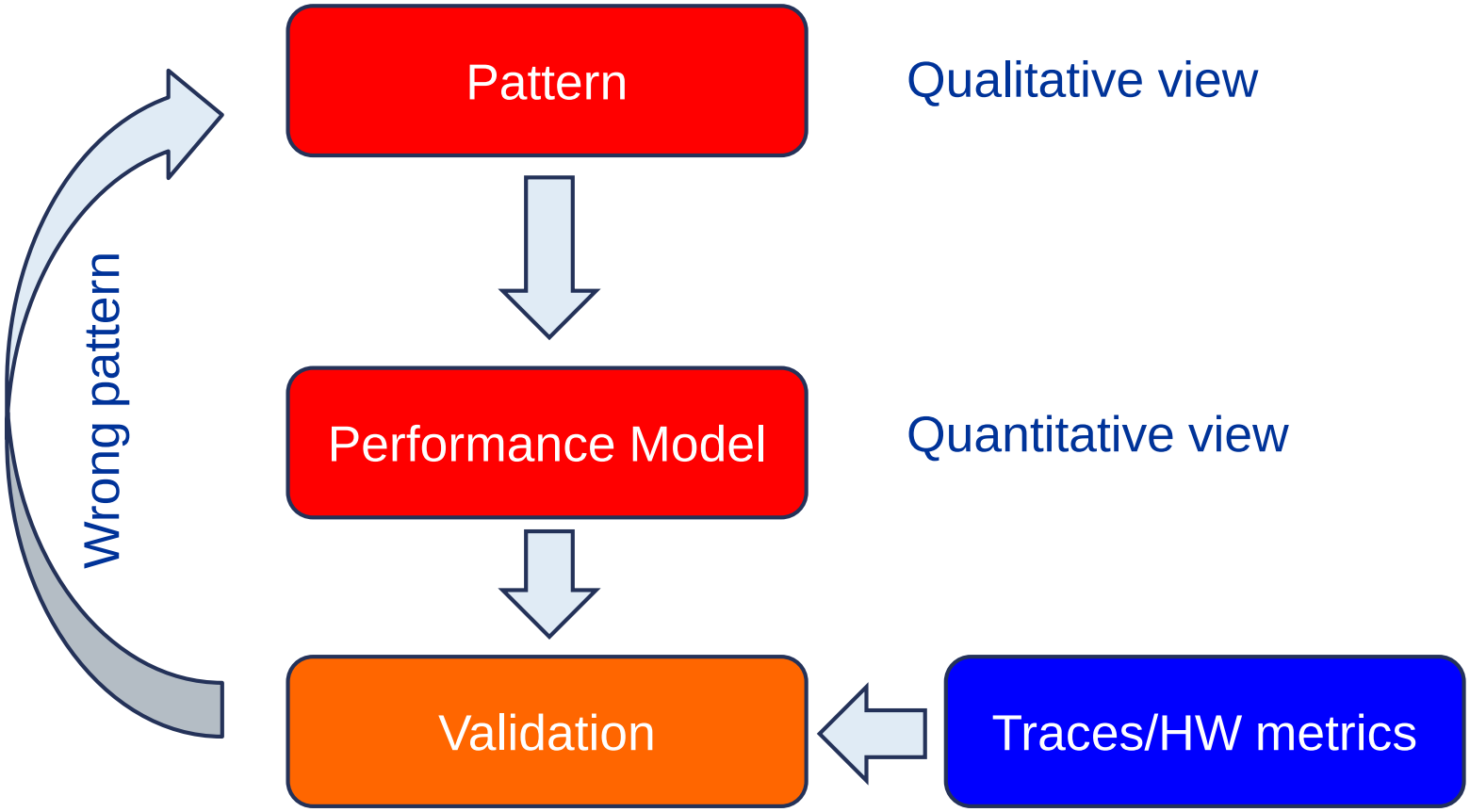
Input:

- Static code analysis
- HPM data
- Scaling data set size
- Scaling number of used cores
- Microbenchmarking



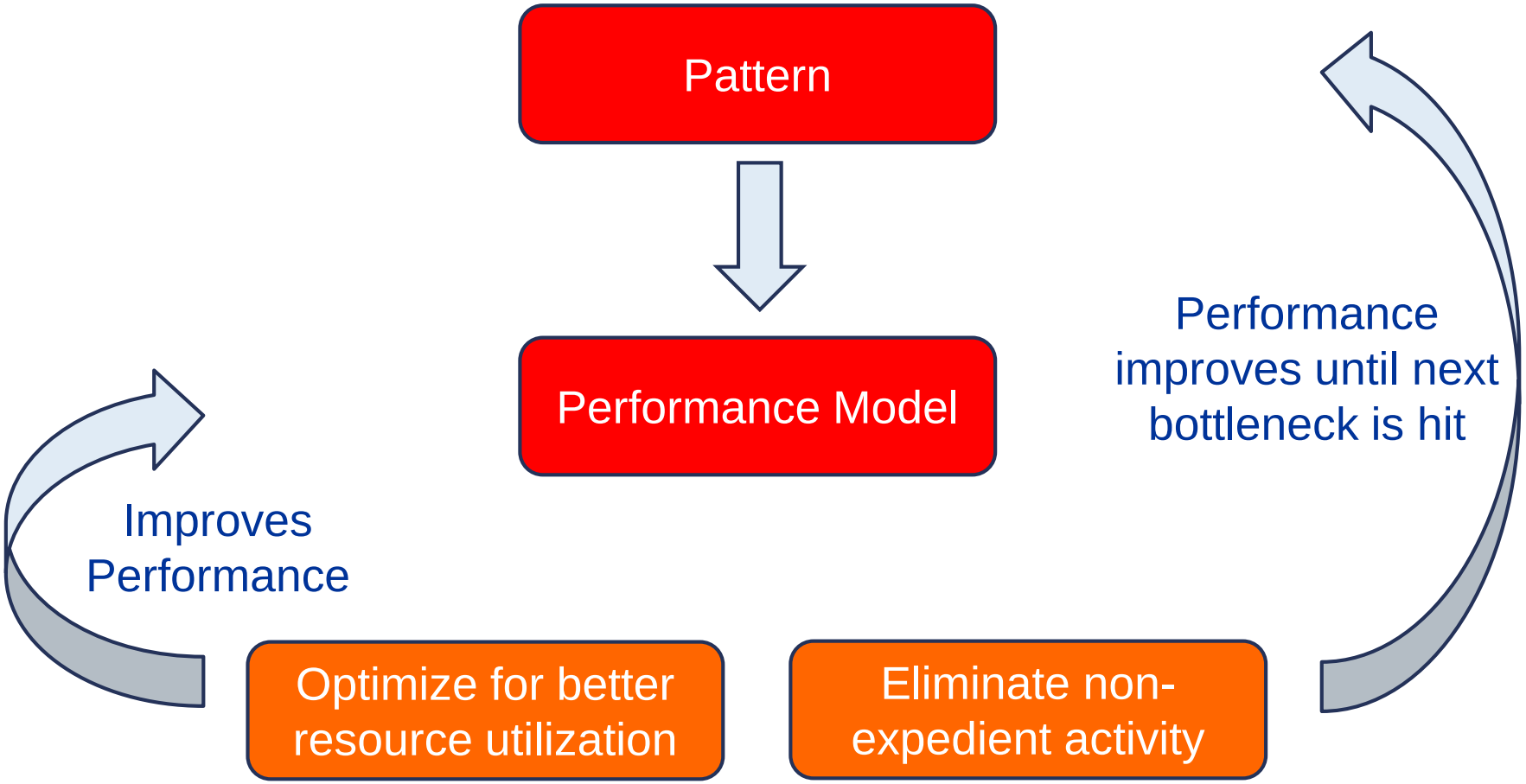
Performance patterns are typical performance limiting motives.
The set of input data indicating a pattern is its **signature**.

Performance Engineering Process: Modelling



Step 2 **Formulate Model:** Validate pattern and get quantitative insight.

Performance Engineering Process: Optimization



Step 3 **Optimization:** Improve utilization of offered resources.

Performance pattern classification

1. Maximum resource utilization
   (computing at a bottleneck)

2. Optimal use of parallel resources

3. Hazards
   (something “goes wrong”)

4. Use of most effective instructions

5. Work related
   (too much work or too inefficiently done)

Patterns (I): Bottlenecks & parallelism

Pattern	Performance behavior	Metric signature, LIKWID performance group(s)
Bandwidth saturation	Saturating speedup across cores sharing a data path	Bandwidth meets BW of suitable streaming benchmark (MEM, L3)
ALU saturation	Throughput at design limit(s)	Good (low) CPI, integral ratio of cycles to specific instruction count(s) (FLOPS_*, DATA, CPI)
Bad ccNUMA page placement	Bad or no scaling across NUMA domains, performance improves with interleaved page placement	Unbalanced bandwidth on memory interfaces / High remote traffic (MEM)
Load imbalance / serial fraction	Saturating/sub-linear speedup	Different amount of “work” on the cores (FLOPS_*); note that instruction count is not reliable!

Patterns (II): Hazards

Pattern	Performance behavior	Metric signature, LIKWID performance group(s)
False sharing of cache lines	Large discrepancy from performance model in parallel case, bad scalability	Frequent (remote) CL evicts (CACHE)
Pipelining issues	In-core throughput far from design limit, performance insensitive to data set size	(Large) integral ratio of cycles to specific instruction count(s), bad (high) CPI (FLOPS_*, DATA, CPI)
Control flow issues	See above	High branch rate and branch miss ratio (BRANCH)
Micro-architectural anomalies	Large discrepancy from simple performance model based on LD/ST and arithmetic throughput	Relevant events are very hardware-specific, e.g., memory aliasing stalls, conflict misses, unaligned LD/ST, requeue events
Latency-bound data access	Simple bandwidth performance model much too optimistic	Low BW utilization / Low cache hit ratio, frequent CL evicts or replacements (CACHE, DATA, MEM)

Patterns (III): Work-related

Pattern		Performance behavior	Metric signature, LIKWID performance group(s)
Synchronization overhead		Speedup going down as more cores are added / No speedup with small problem sizes / Cores busy but low FP performance	Large non-FP instruction count (growing with number of cores used) / Low CPI (FLOPS_*, CPI)
Instruction overhead		Low application performance, good scaling across cores, performance insensitive to problem size	Low CPI near theoretical limit / Large non-FP instruction count (constant vs. number of cores) (FLOPS_*, DATA, CPI)
Excess data volume		Simple bandwidth performance model much too optimistic	Low BW utilization / Low cache hit ratio, frequent CL evicts or replacements (CACHE, DATA, MEM)
Code composition	**Expensive instructions**	Similar to instruction overhead	Many cycles per instruction (CPI) if the problem is large-latency arithmetic
	Ineffective instructions		Scalar instructions dominating in data-parallel loops (FLOPS_*, CPI)

Where to start

Look at the code and understand what it is doing!

Scaling runs:

- Scale #cores inside ccNUMA domain
- Scale across ccNUMA domains
- Scale working set size (if possible)

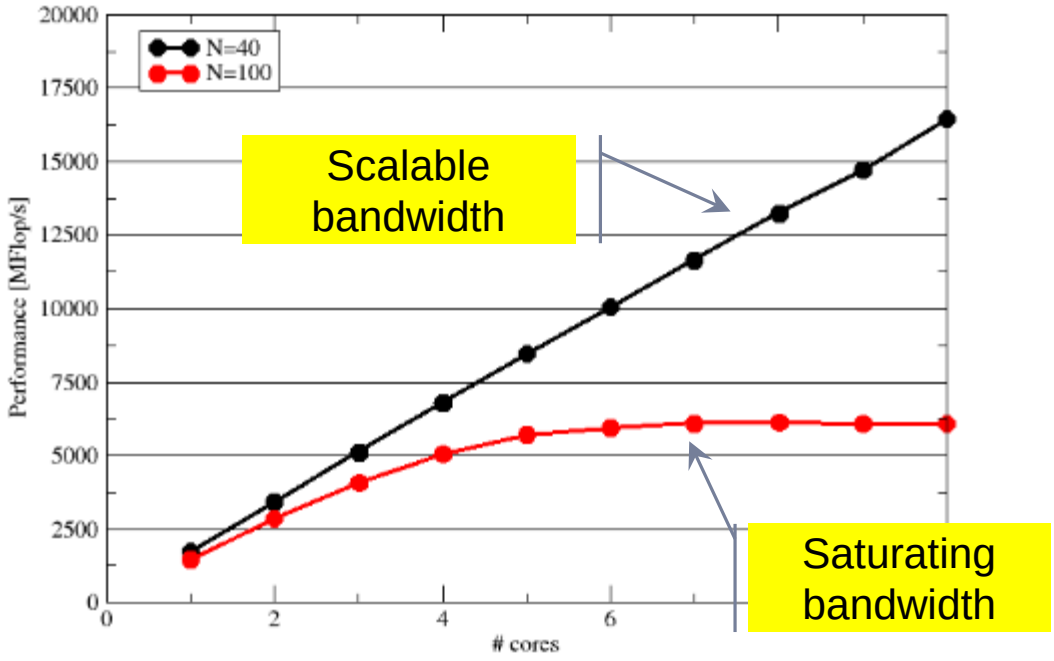
HPM measurements:

- Memory Bandwidth
- Instruction decomposition: Arithmetic, data, branch, other
- SIMD vectorized fraction
- Data volumes inside memory hierarchy
- CPI

Pattern: Bandwidth Saturation

Always check this first!

1. Perform scaling run inside ccNUMA domain
2. Measure memory bandwidth with HPM
3. Compare to micro benchmark with similar data access pattern



Measured bandwidth spmv:
45964 MB/s
Synthetic load benchmark:
47022 MB/s

Pattern: Instruction Overhead

1. Perform a HPM instruction decomposition analysis
2. Measure resource utilization
3. Static code analysis

Instruction decomposition	Inlining failed	Inefficient data structures
Arithmetic FP	**12%**	**21%**
Load/Store	30%	50%
Branch	24%	10%
Other	34%	19%

C++ codes which suffer from overhead (inlining problems, complex abstractions) need a lot more overall instructions related to the arithmetic instructions

- Often (but not always) “good” (i.e., low) CPI
- Low-ish bandwidth
- Low # of floating-point instructions vs. other instructions

Pattern: Inefficient Instructions

1. HPM measurement: Relation packed vs. scalar instructions
2. Static assembly code analysis: Search for scalar loads

Small fraction of packed instructions

No AVX

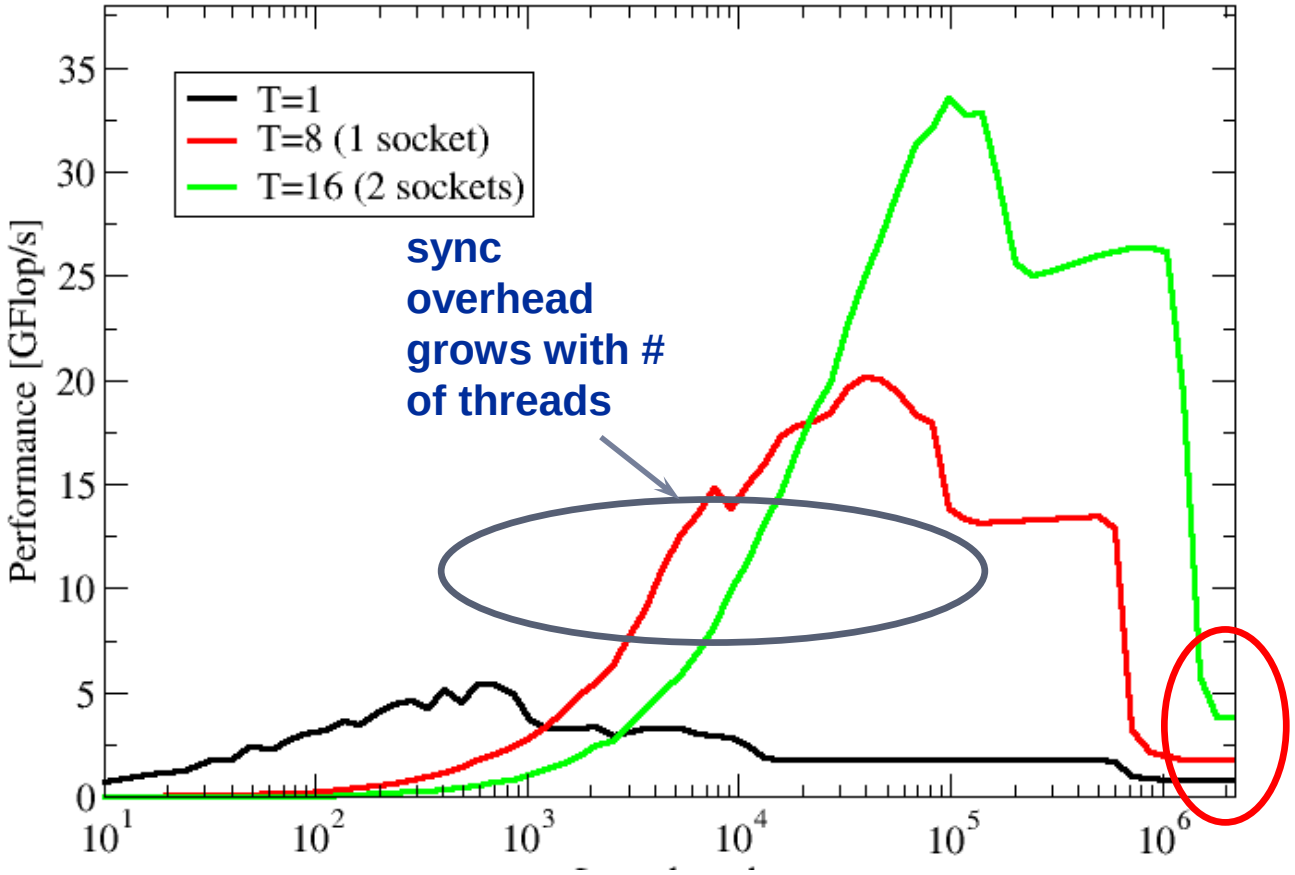
	core 0	core 1	core 2	core 3	
INSTR_RETIRED_ANY	2.19445e+11	1.7674e+11	1.76255e+11	1.75728e+11	1.75578e+11
CPU_CLK_UNHALTED_CORE	1.4396e+11	1.28759e+11	1.28846e+11	1.28898e+11	1.28905e+11
CPU_CLK_UNHALTED_REF	1.20204e+11	1.0895e+11	1.09024e+11	1.09067e+11	1.09074e+11
FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE	1.1169e+09	1.09639e+09	1.09739e+09	1.10112e+09	1.10033e+09
FP_COMP_OPS_EXE_SSE_FP_SCALAR_DOUBLE	3.62746e+10	3.45789e+10	3.45446e+10	3.44553e+10	3.44820e+10
SIMD_FP_256_PACKED_DOUBLE	0	0	0	0	0

- There is usually no counter for packed vs scalar (SIMD) loads and stores.
- Also the compiler usually does not distinguish!

Only solution: Inspect code at assembly level.

Pattern: Synchronization overhead

1. Performance is decreasing with growing core counts
2. Performance is sensitive to topology
3. Static code analysis: Estimate work vs. barrier cost.

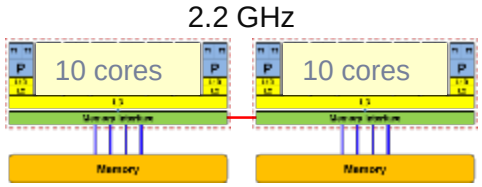


Thread synchronization overhead on IvyBridge-EP

Barrier overhead in CPU cycles

2 Threads	Intel 16.0	GCC 5.3.0
Shared L3	**599**	**425**
SMT threads	**612**	**423**
Other socket	**1486**	**1067**

Strong topology dependence!



Full domain	Intel 16.0	GCC 5.3.0
Socket (10 cores)	1934	1301
Node (20 cores)	**4999**	**7783**
Node +SMT	5981	9897

Overhead grows with thread count

- **Strong dependence on compiler, CPU and system environment!**
- **`OMP_WAIT_POLICY=ACTIVE` can make a big difference**

Thread synchronization overhead on Xeon Phi 7210 (64-core)

Barrier overhead in CPU cycles (Intel C compiler 16.03)

2 threads on distinct cores: 730

	SMT1	SMT2	SMT3	SMT4
One core	n/a	**963**	1580	2240
Full chip	5720	8100	9900	**11400**

Still the pain may be much larger, as more work can be done in one cycle on Phi compared to a full Ivy Bridge node

3.2x cores (20 vs 64) on Phi

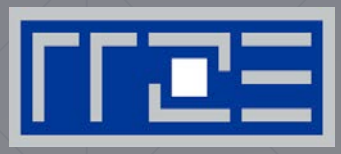
4x more operations per cycle per core on Phi

→ 4 · 3.2 = 12.8x more work done on Xeon Phi per cycle

1.9x more barrier penalty (cycles) on Phi (11400 vs. 6000)

→ One barrier causes 1.9 · 12.8 ≈ 24x more pain ☺.

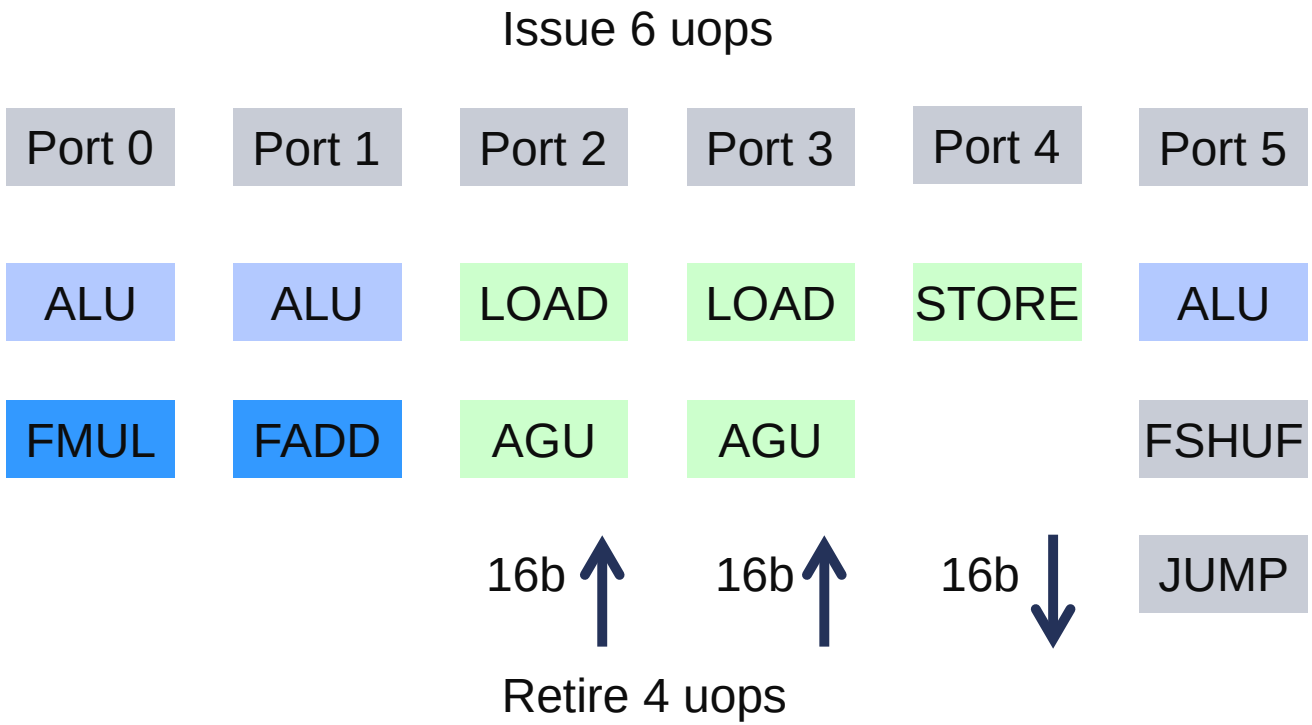
“SIMPLE” PERFORMANCE MODELING: THE ROOFLINE MODEL



Loop-based performance modeling:
Execution vs. data transfer

Preliminary: Estimating Instruction throughput

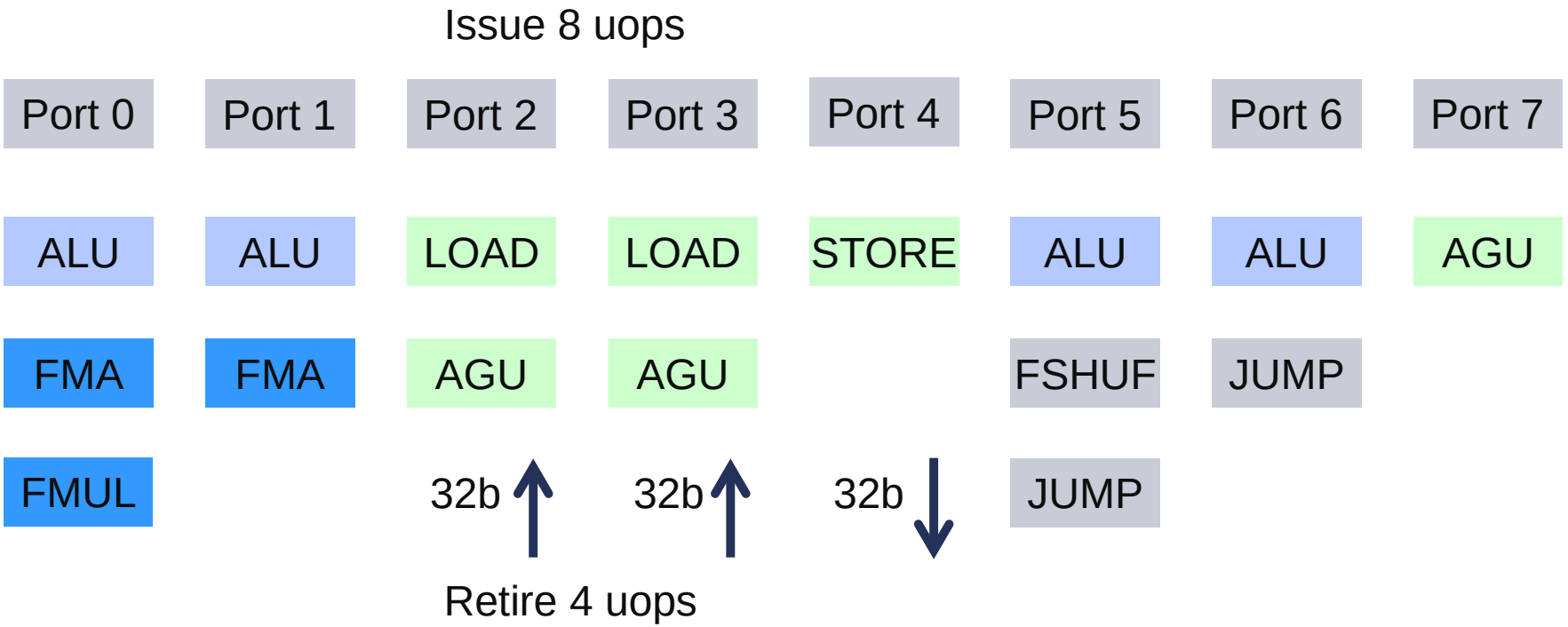
How to perform a instruction throughput analysis on the example of Intel's port based scheduler model.



SandyBridge

Preliminary: Estimating Instruction throughput

Every new generation provides incremental improvements.
The OOO microarchitecture is a blend between P6 (Pentium Pro) and P4 (Netburst) architectures.



Haswell

Exercise: Estimate performance of triad on SandyBridge @3GHz

```
double  *A, *B, *C, *D;
for (int i=0; i<N; i++) {
   A[i] = B[i] + C[i] * D[i]
}
```

How many cycles to process one 64byte cacheline?

64byte equivalent to 8 scalar iterations or **2** AVX vector iterations.

Cycle 1: load and ½ store and mult and add

Cycle 2: load and ½ store

Cycle 3: load **Answer: 6 cycles**

Exercise: Estimate performance of triad on SandyBridge @3GHz

```
double  *A, *B, *C, *D;
for (int i=0; i<N; i++) {
   A[i] = B[i] + C[i] * D[i]
}
```

Whats the performance in GFlops/s and bandwidth in MBytes/s ?

One AVX iteration (3 cycles) performs 4x2=8 flops.

(3 GHZ / 3 cycles) * 4 updates * 2 flops/update = **8 GFlops/s**

4 GUPS/s * 4 words/update * 8byte/word = **128 GBytes/s**

The Roofline Model[1,2]

1. P_{max} = **Applicable peak performance** of a loop, assuming that data comes from L1 cache (this is not necessarily P_{peak})

2. I = **Computational intensity ("work" per byte transferred)** over the slowest data path utilized ("the bottleneck")
   - Code balance $B_C = I^{-1}$

3. b_S = **Applicable peak bandwidth** of the slowest data path utilized

Expected performance:

[F/B] [B/s]

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

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)

“Simple” Roofline: The vector triad

Vector triad `A(:)=B(:)+C(:)*D(:)` on a 2.7 GHz 8-core SNB chip

Consider full chip (8 cores):

Memory bandwidth: b_S = **40 GB/s**

Code balance (incl. write allocate):

B_c = (4+1) Words / 2 Flops = 20 B/F → ***I* = 0.05 F/B**

→ $I \cdot b_S$ **= 2.0 GF/s** (1.1% of peak performance)

P_{peak} / core = 21.7 Gflop/s ((4+4) Flops/cy x 2.7 GHz)

P_{max} / core = 7.2 Gflop/s (1 AVX LD/cy)

→ P_{max} **= 8 * 7.2 Gflop/s = 57.6 Gflop/s** (33% peak)

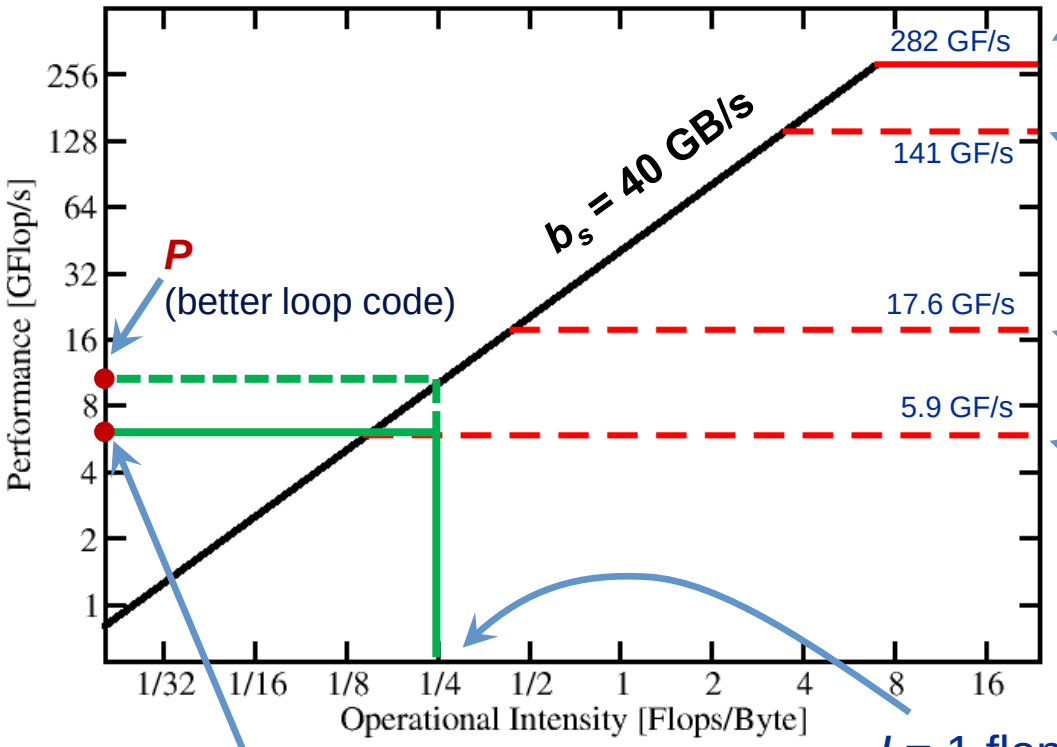
$$P = \min(P_{\max}, I \cdot b_S) = \min(57.6, 2.0)\,\mathrm{GFlop/s} = 2.0\,\mathrm{GFlop/s}$$

A not so simple Roofline example

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

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ "large" N

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



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

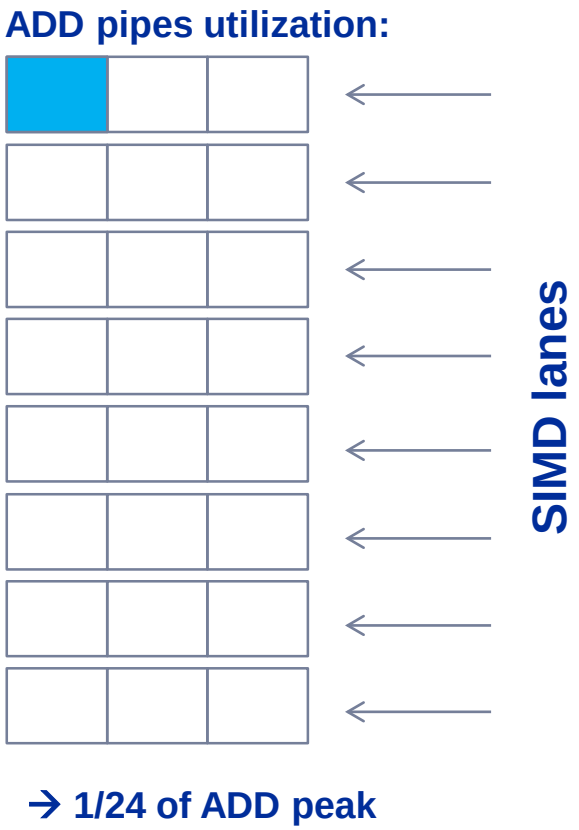
How do we get these numbers???

I = 1 flop / 4 byte (SP!)

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
```



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:

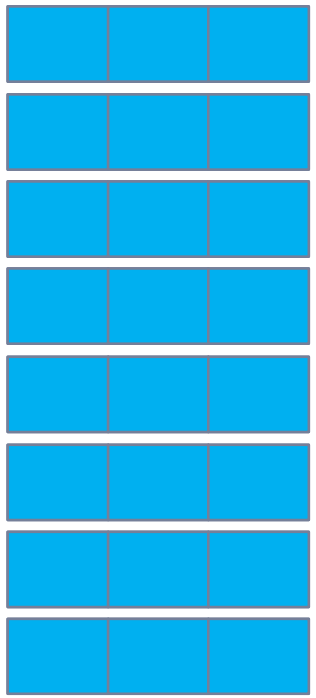
→ 1/8 of ADD peak

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
```

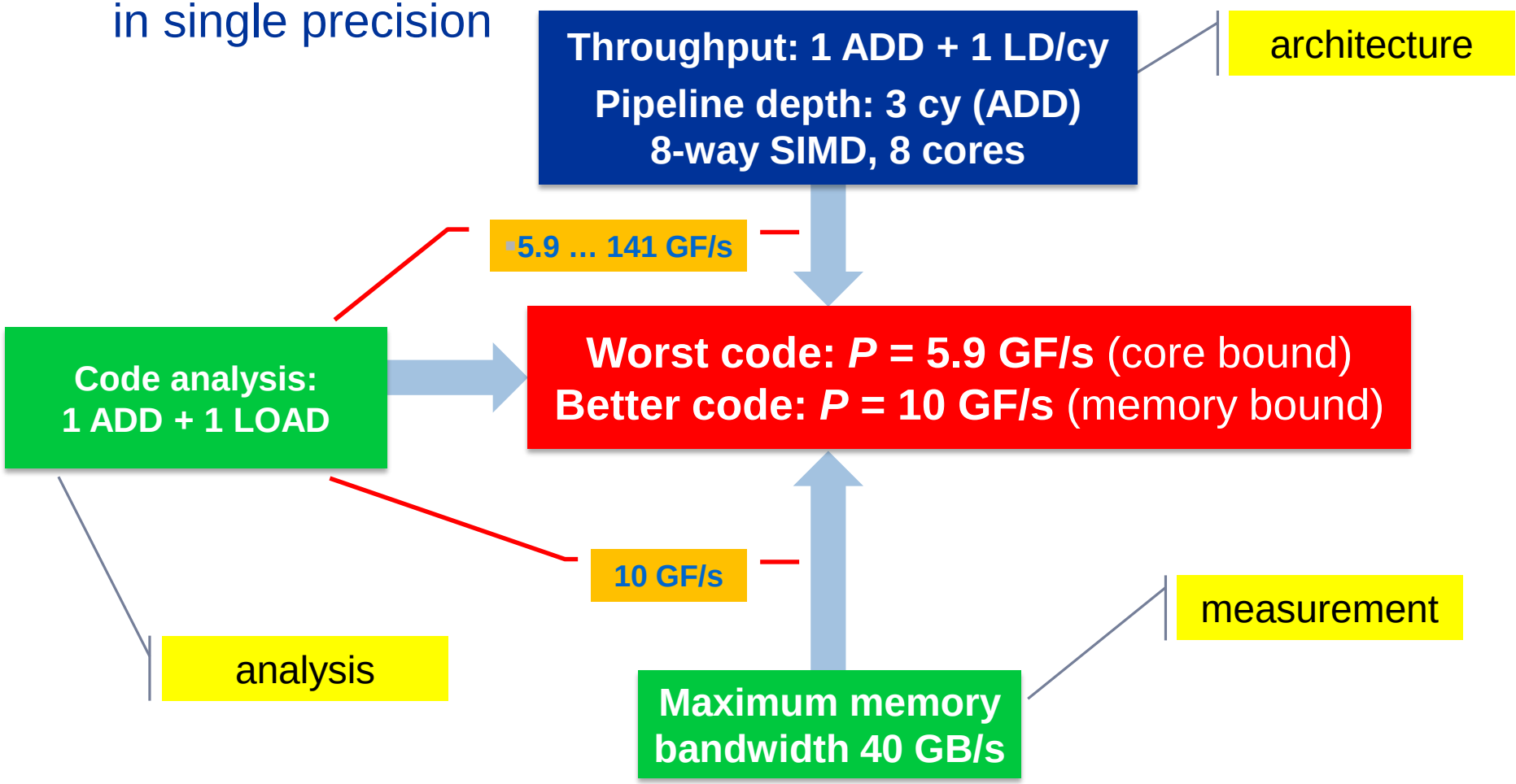
ADD pipes utilization:



→ ADD peak

Input to the roofline model

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



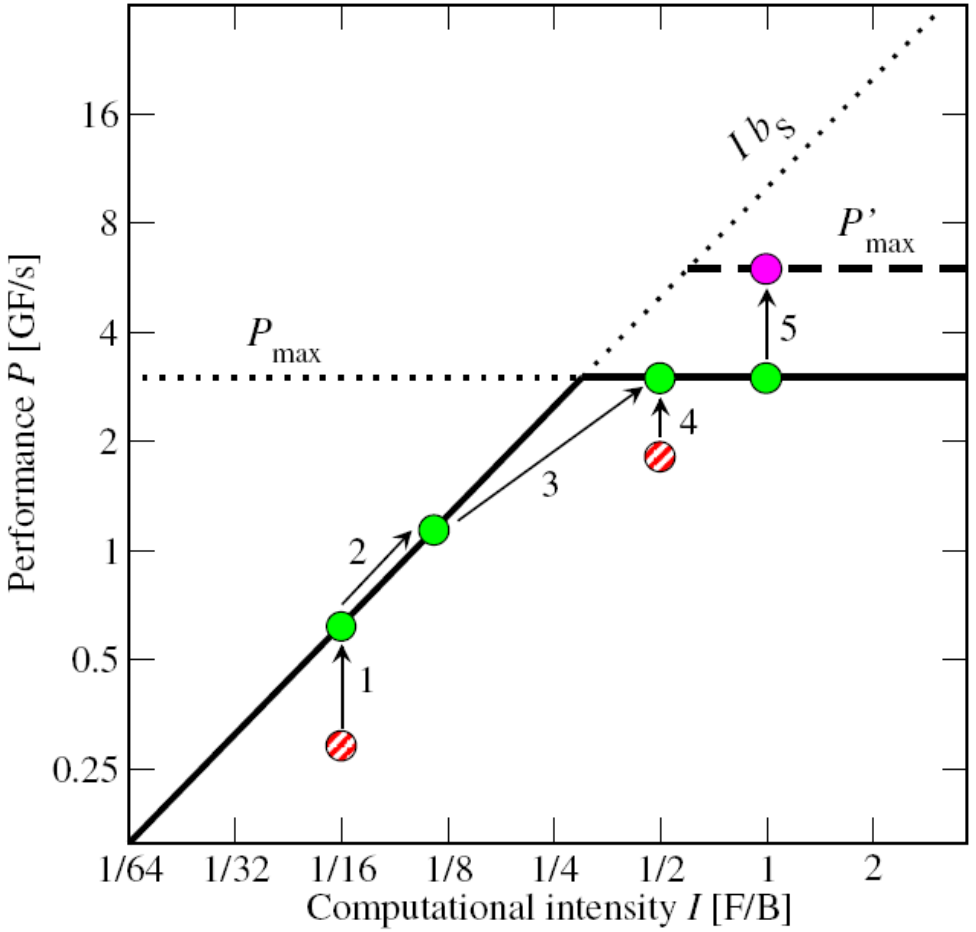
Assumptions for the Roofline Model

The roofline formalism is based on some (crucial) **assumptions**:

- There is a clear concept of **“work” vs. “traffic”**
  - “work” = flops, updates, iterations…
  - “traffic” = required data to do “work”
- **Attainable bandwidth of code = input parameter!** Determine effective bandwidth via simple streaming benchmarks to model more complex kernels and applications
- **Data transfer and core execution overlap** perfectly!
- **Slowest data path is modeled only**; all others are assumed to be infinitely fast
- The **bandwidth** of the slowest data path can be **utilized to 100% (“saturation”)**
- Latency effects are ignored, i.e. **perfect streaming mode**

Typical code optimizations in the Roofline Model

1. Hit the BW bottleneck by good serial code
2. Increase intensity to make better use of BW bottleneck
3. Increase intensity and go from memory-bound to core-bound
4. Hit the core bottleneck by good serial code
5. Shift P_{max} by accessing additional hardware features or using a different algorithm/implementation

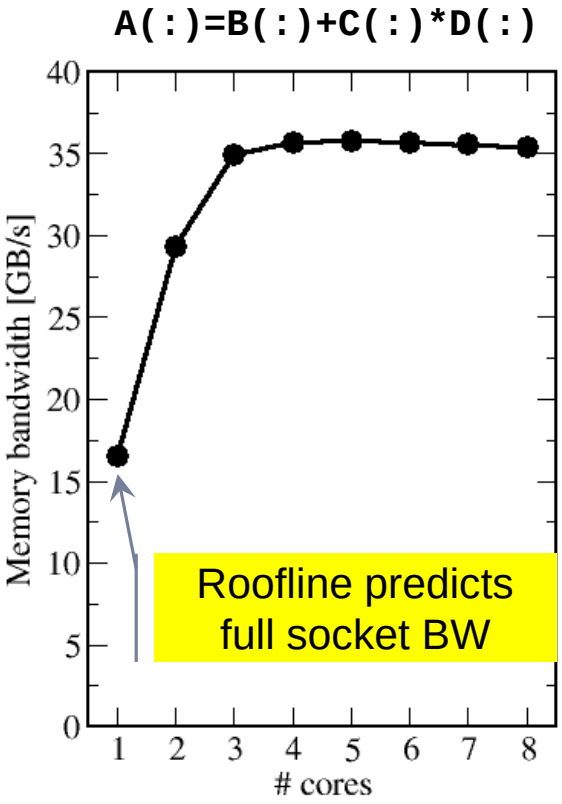


Shortcomings of the roofline model

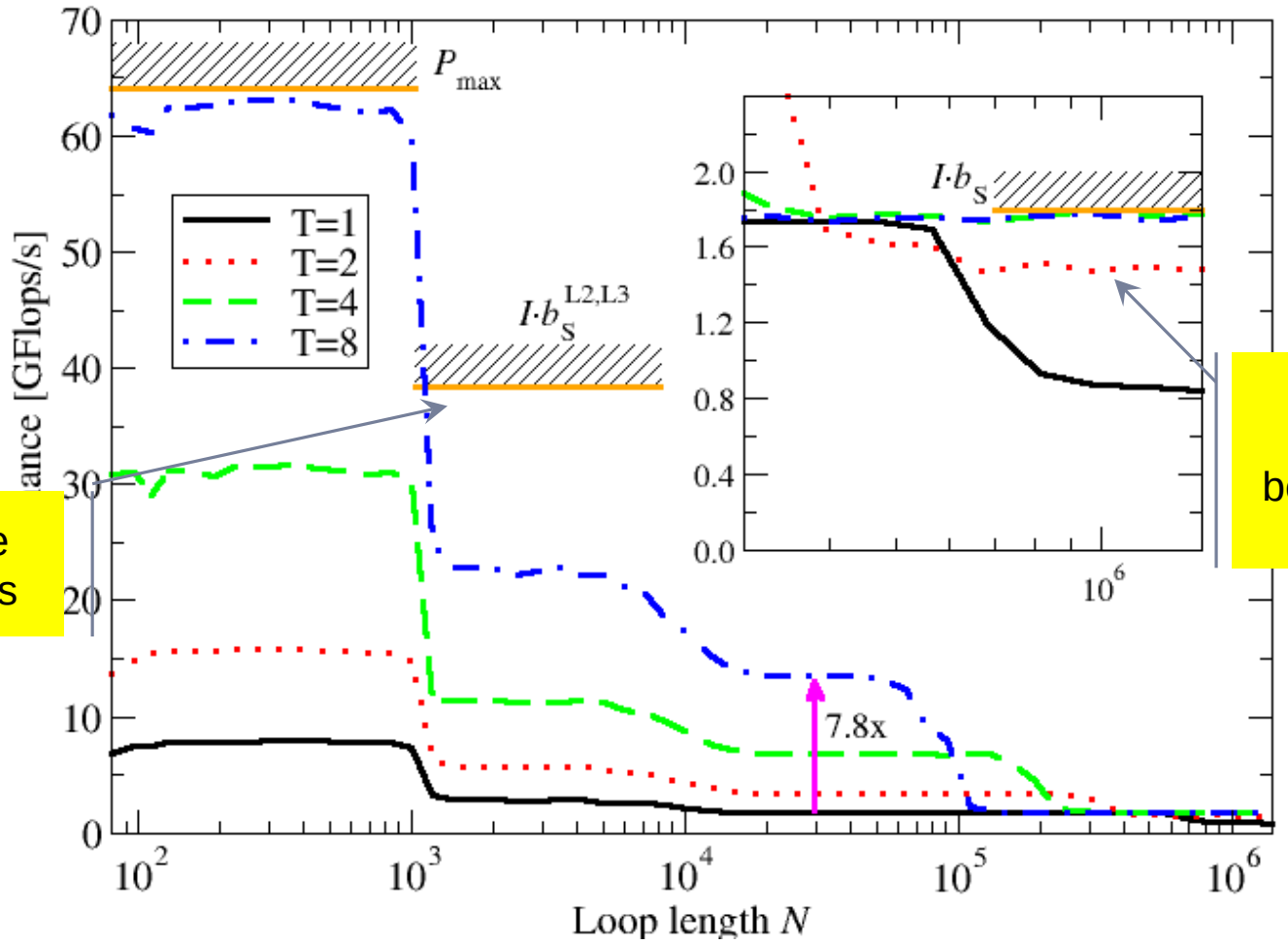
Saturation effects in multicore chips are not explained

- Reason: **"saturation assumption"**
- Cache line transfers and core execution do sometimes not overlap perfectly
- Only **increased "pressure" on the memory interface** can saturate the bus
  → need more cores!

ECM model gives more insight



Where the roofline model fails



In cache situations

In memory performance below saturation point

ECM Model

ECM = “Execution-Cache-Memory”

Assumptions:

Single-core execution time is composed of

1. In-core execution
2. Data transfers in the memory hierarchy

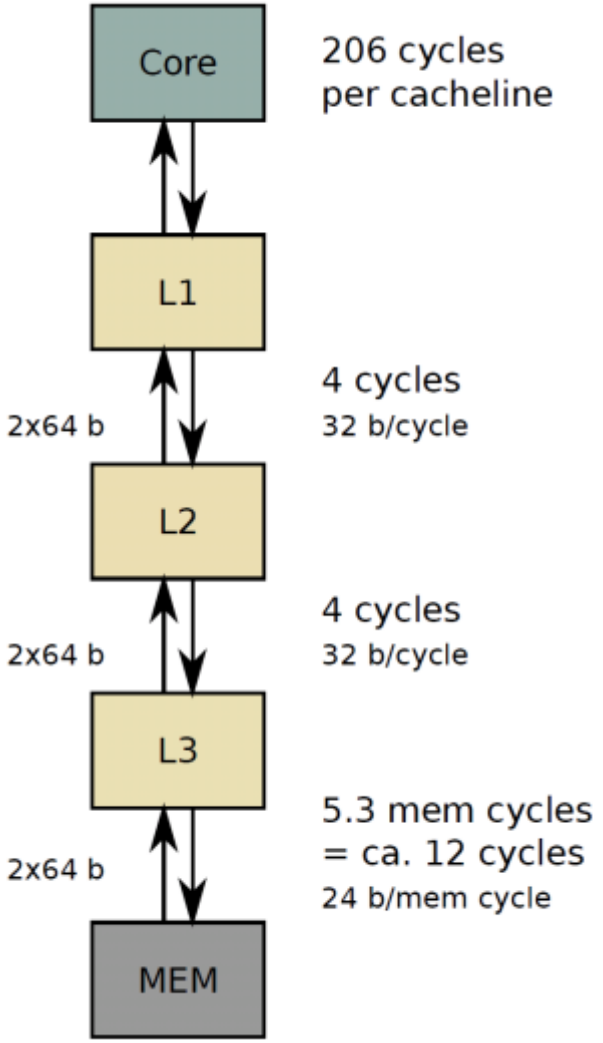
Data transfers may or may not overlap with each other or with in-core execution

Scaling is linear until the relevant bottleneck is reached

Input:

Same as for Roofline

+ **data transfer times** in hierarchy



Introduction to ECM model

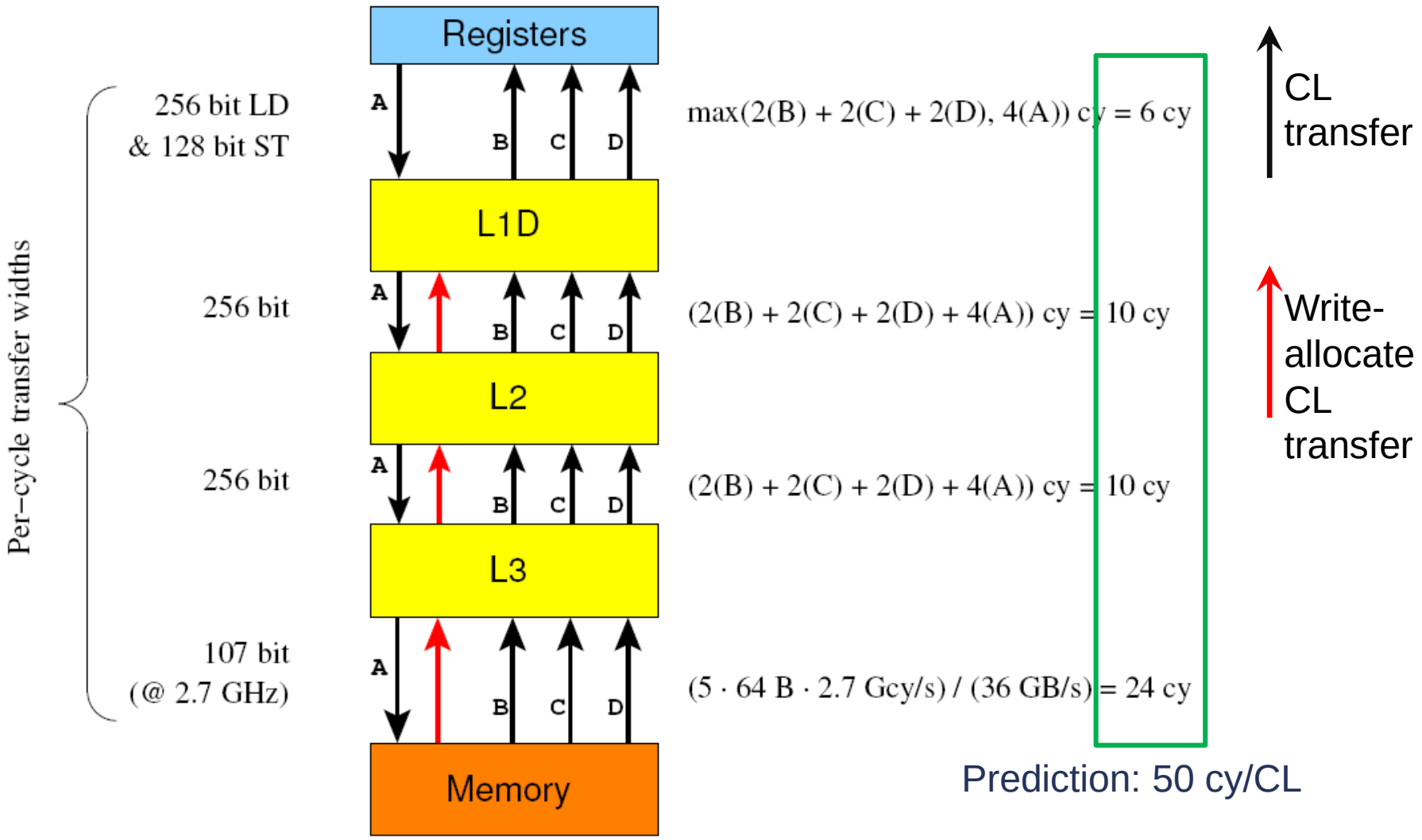
ECM = “Execution-Cache-Memory”

- Analytical performance model
- Focus on resource utilization
  - Instruction Execution
  - Data Movement
- Lightspeed assumption:
  - Optimal instruction throughput
  - Always bandwidth bound

The RULES™ for x86 CPUs

1. Single-core execution time is composed of
   1. In-core execution
   2. Data transfers in the memory hierarchy
2. All timings are in units of one CL
3. LOADS in the L1 cache do not overlap with any other data transfer
4. Scaling across cores is linear until a shared bottleneck is hit

ECM for `A(:)=B(:)+C(:)*D(:)` on 2.7 GHz SNB core



Multicore scaling in the ECM model

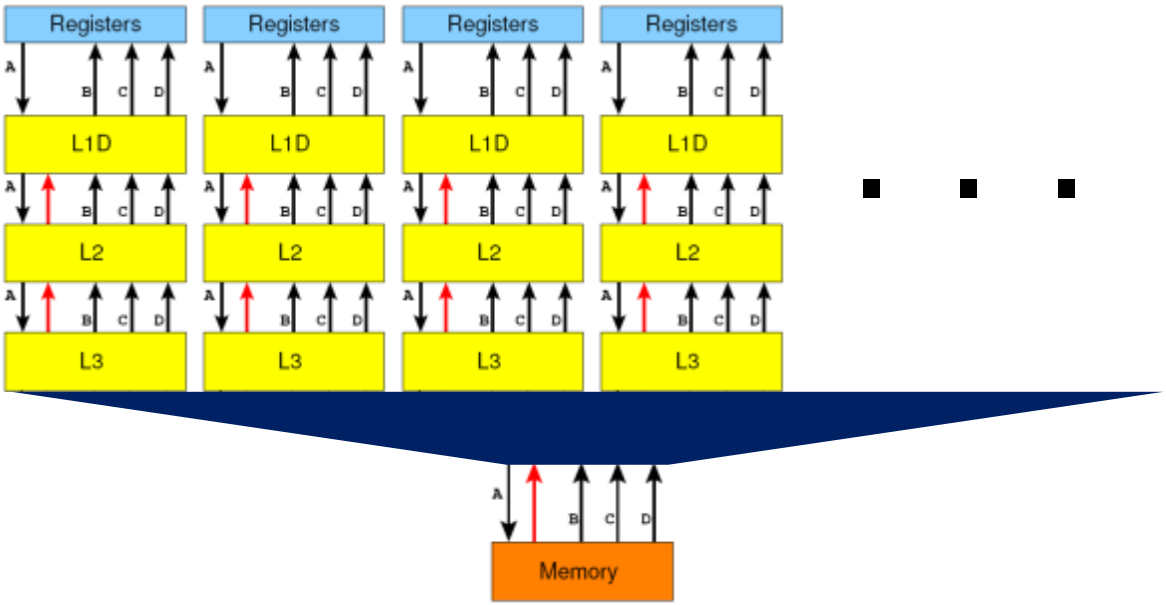
Identify relevant **bandwidth bottlenecks**

- L3 cache
- Memory interface

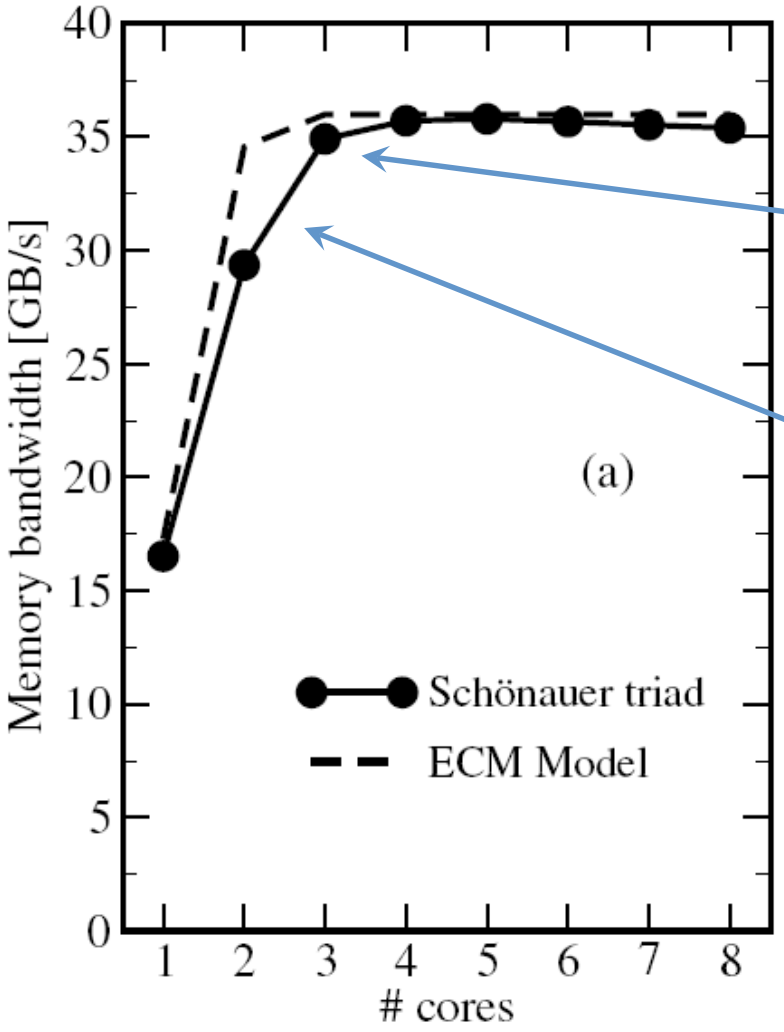
Scale single-thread performance until **first bottleneck** is hit:

$$n \text{ cores: } P(n) = \min(nP_0, I \cdot b_S)$$

Example: Scalable L3 on Sandy Bridge



ECM prediction vs. measurements for `A(:)=B(:)+C(:)*D(:)`, no overlap



Model: Scales until saturation sets in

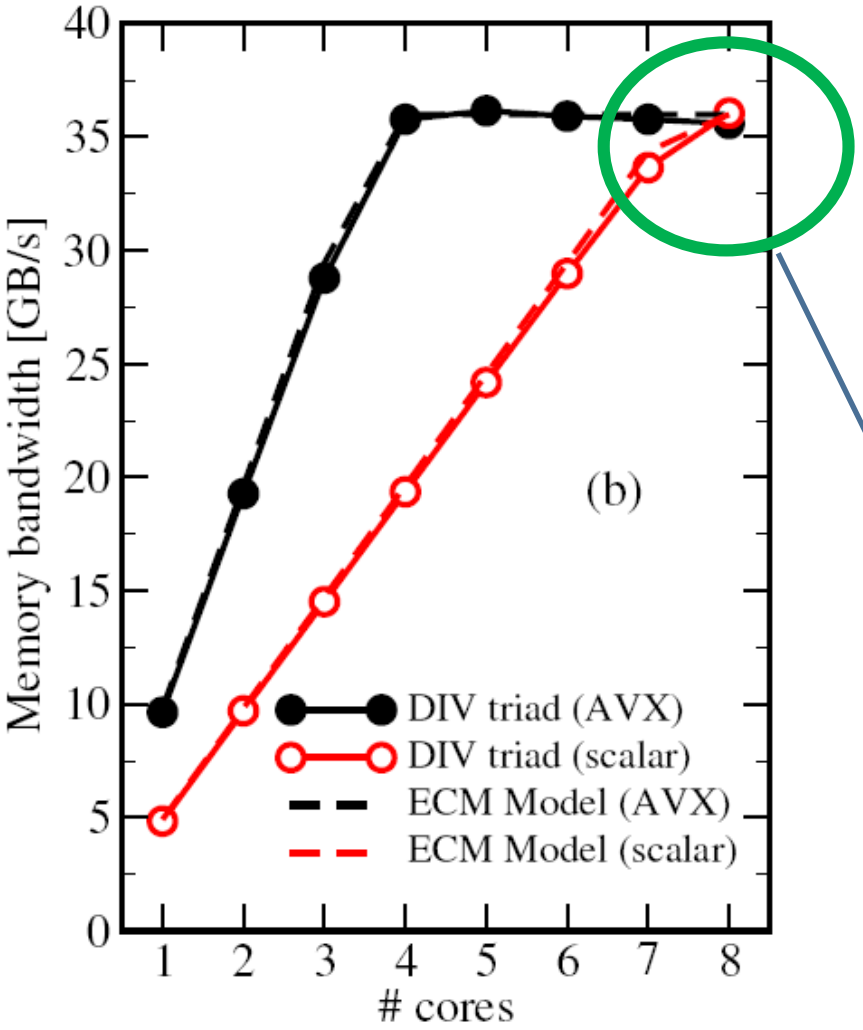
Saturation point (# cores) well predicted

Measurement: scaling not perfect

Caveat: This is specific for this architecture and this benchmark!

Check: Use “overlappable” kernel code

ECM prediction vs. measurements for `A(:)=B(:)+C(:)/D(:)` with full overlap



In-core execution is dominated by divide operation (44 cycles with AVX, 22 scalar)

→ **Almost perfect agreement** with ECM model

Parallelism "heals" bad single-core performance … just barely!

Summary: The ECM Model

- The ECM model is a powerful analysis tool to get insight into:
  - Runtime contributions
  - Bottleneck identification
  - Runtime overlap

It can predict single core performance for any memory hierarchy level and provide an estimate of multicore scalability.

References

- J. Treibig and G. Hager: *Introducing a Performance Model for Bandwidth-Limited Loop Kernels*. Proceedings of the Workshop “Memory issues on Multi- and Manycore Platforms” at PPAM 2009, the 8th International Conference on Parallel Processing and Applied Mathematics, Wroclaw, Poland, September 13-16, 2009. Lecture Notes in Computer Science Volume 6067, 2010, pp 615-624. DOI: 10.1007/978-3-642-14390-8_64 (2010).
- 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, DOI: 10.1002/cpe.3180 (2013).
- M. Wittmann, G. Hager, T. Zeiser, J. Treibig, and G. Wellein: *Chip-level and multi-node analysis of energy-optimized lattice-Boltzmann CFD simulations.* Concurrency Computat.: Pract. Exper. (2015), DOI: 10.1002/cpe.3489

- H. Stengel, J. Treibig, G. Hager, and G. Wellein: *Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory model*. Proc. ICS’15, the 29th International Conference on Supercomputing, Newport Beach, CA, June 8-11, 2015. DOI: 10.1145/2751205.2751240

Further references

- M. Wittmann, G. Hager, J. Treibig and G. Wellein: *Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters*. Parallel Processing Letters **20** (4), 359-376 (2010). DOI: 10.1142/S0129626410000296
- J. Treibig, G. Hager, H. G. Hofmann, J. Hornegger, and G. Wellein: *Pushing the limits for medical image reconstruction on recent standard multicore processors*. International Journal of High Performance Computing Applications **27**(2), 162-177 (2013). DOI: 10.1177/1094342012442424
- S. Kronawitter, H. Stengel, G. Hager, and C. Lengauer: *Domain-Specific Optimization of Two Jacobi Smoother Kernels and their Evaluation in the ECM Performance Model*. Parallel Processing Letters **24**, 1441004 (2014). DOI: 10.1142/S0129626414410047
- J. Hofmann, D. Fey, M. Riedmann, J. Eitzinger, G. Hager, and G. Wellein: *Performance analysis of the Kahan-enhanced scalar product on current multi- and manycore processors*. Concurrency & Computation: Practice & Experience (2016). Available online, DOI: 10.1002/cpe.3921.