ERLANGEN REGIONAL
COMPUTING CENTER

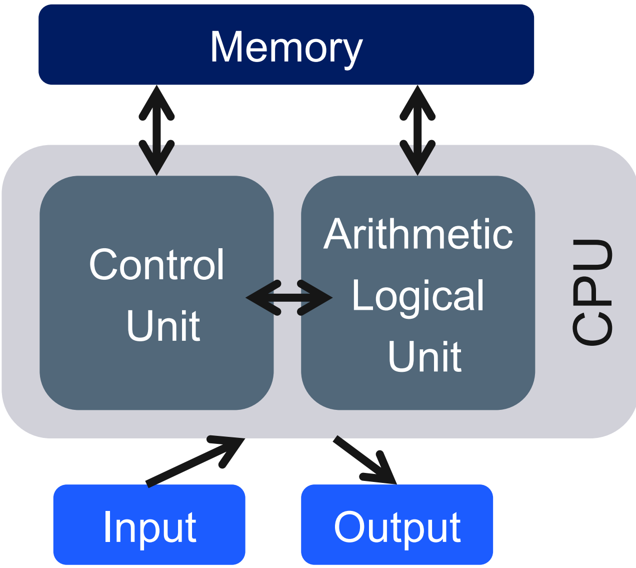
ProPE: Node Level Performance Engineering and Performance Patterns

J. Eitzinger

PPCES 2018, 15.3.2018

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

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 !

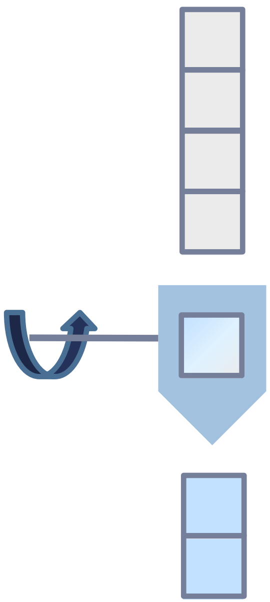
- Improvements for **relevant** software
- What are the **technical** opportunities?
- **Economical** concerns
- **Marketing** concerns

Strategies

Execution and memory

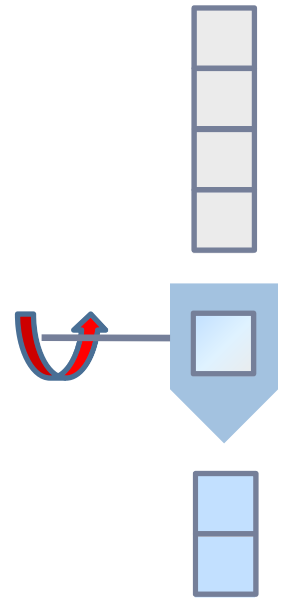
- Increase clock speed
- Parallelism
- Specialization

Performance increase by clock increase



Throughput:

1 Unit per second

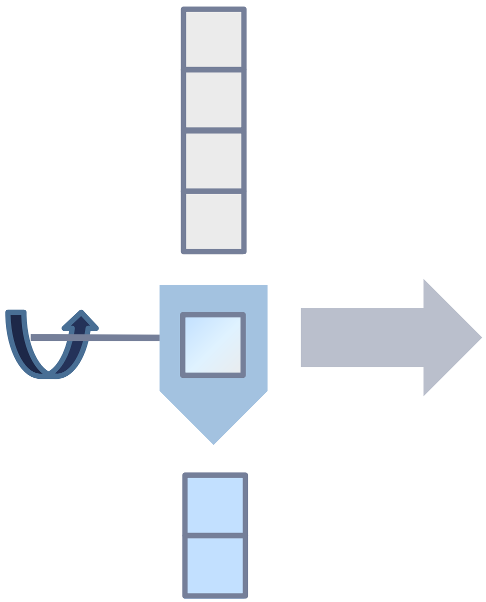


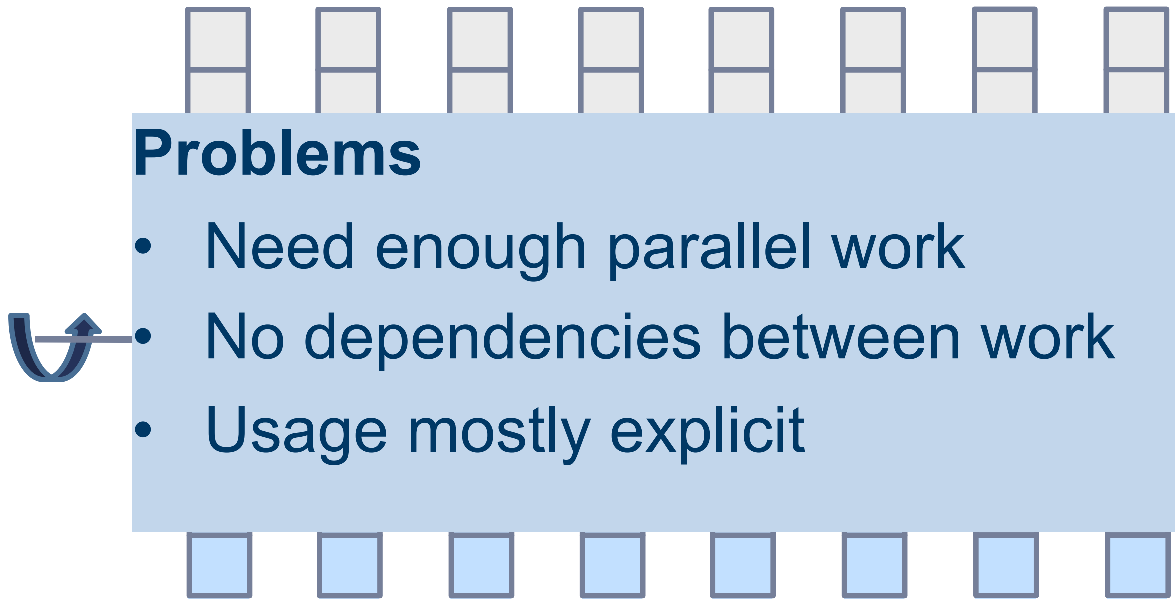
Throughput:

4 Units per second

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

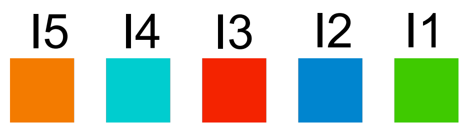
Throughput:

8 Unit per second

Instruction level parallelism

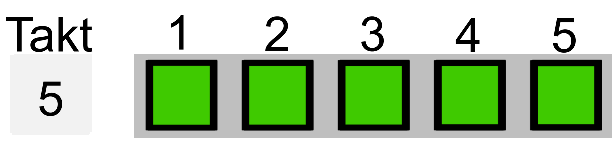
Pipelining

Instructions



Single instruction takes 5 cycles

Stages



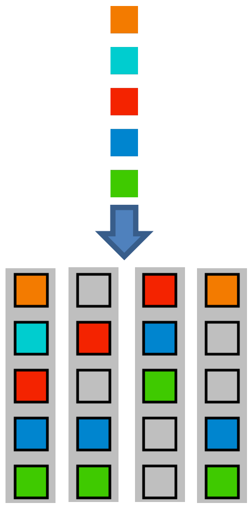
Throughput:

1 instruction per cycle

Speedup by factor 5

Superscalar execution

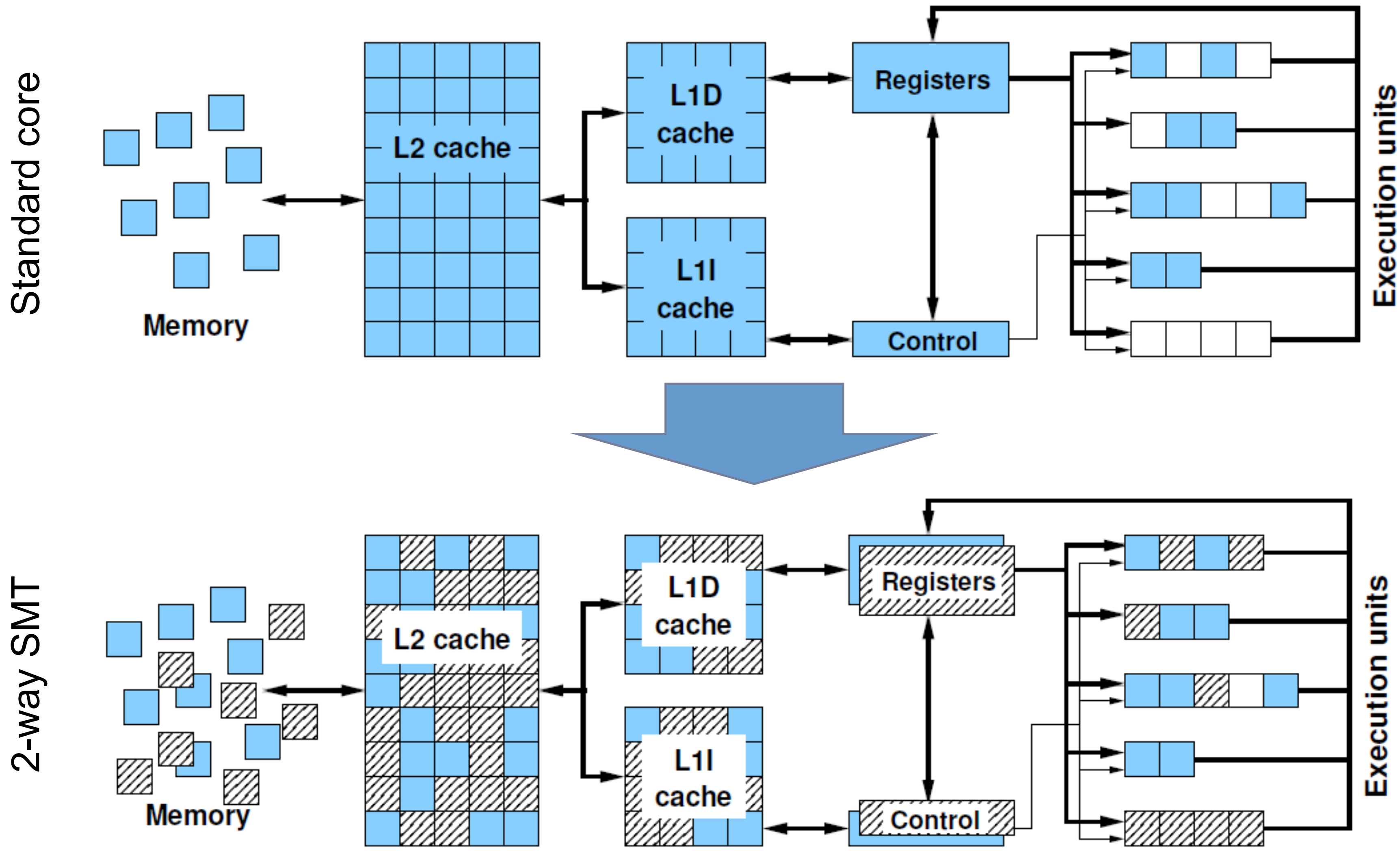
4-fach superskalar



Throughput:

4 instructions per cycle

Core details: Simultaneous multi-threading (SMT)



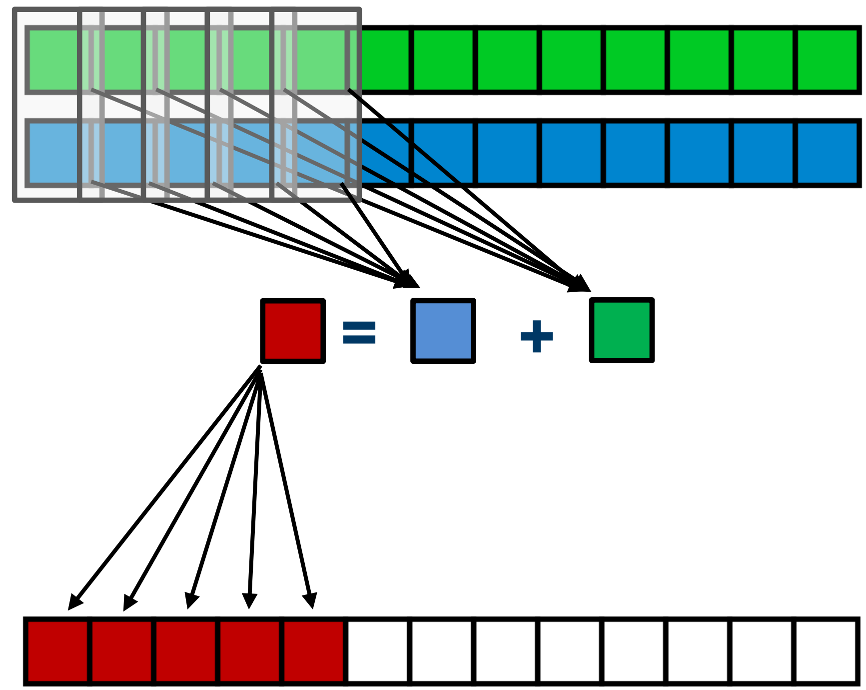
Data parallel execution units (SIMD)

```
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

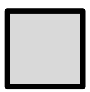


Data parallel execution units (SIMD)

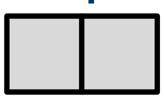
```
for (int j=0; j<size; j++){
     A[j] = B[j] + C[j];
}
```

Register widths

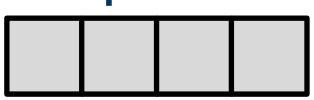
- 1 operand



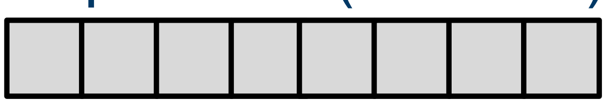
- 2 operands (SSE)



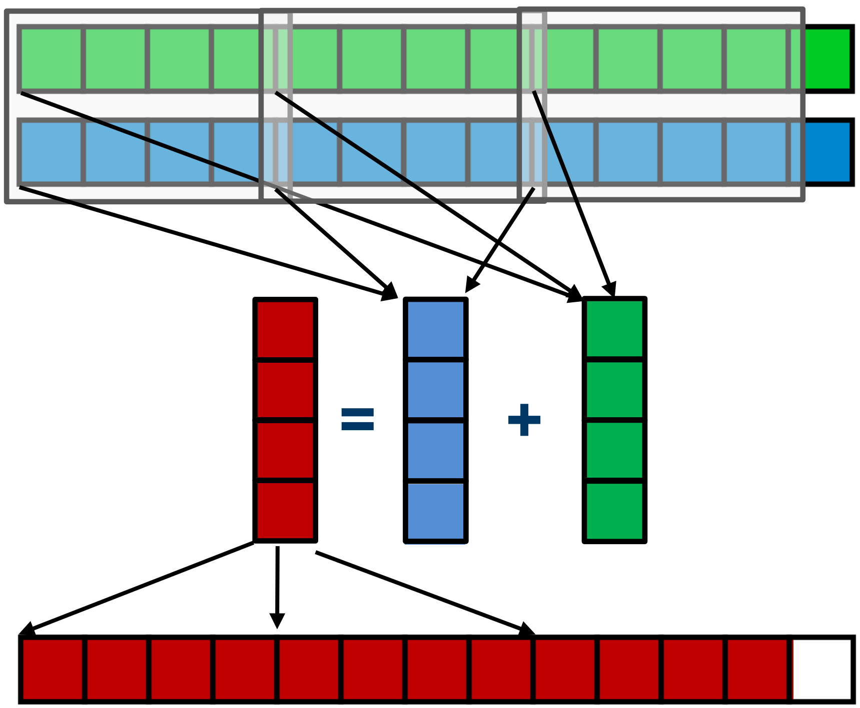
- 4 operands (AVX)



- 8 operands (AVX512)



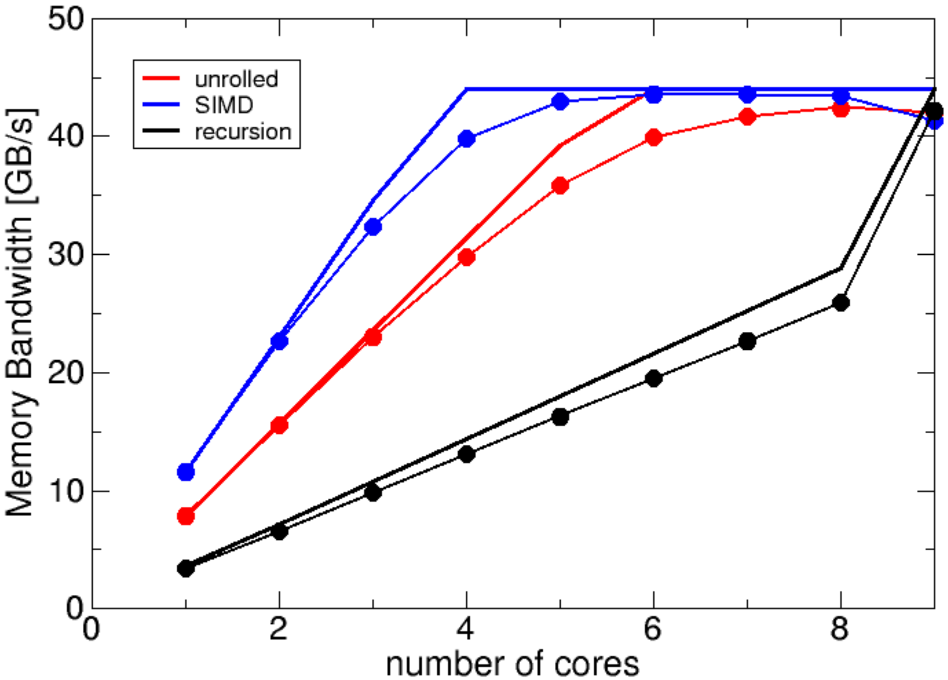
SIMD execution



Limits of SIMD processing

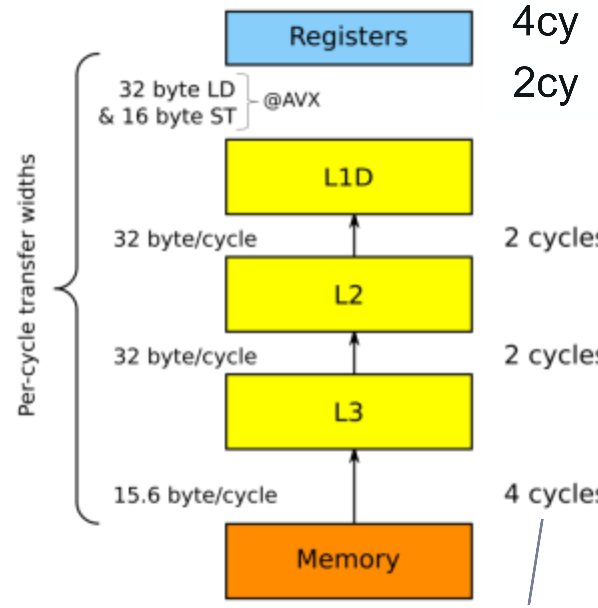
- Only part of application may be vectorized, arithmetic vs. load/store (Amdahls law), data transfers
- Memory saturation often makes SIMD obsolete

16cy
4cy
2cy



ry

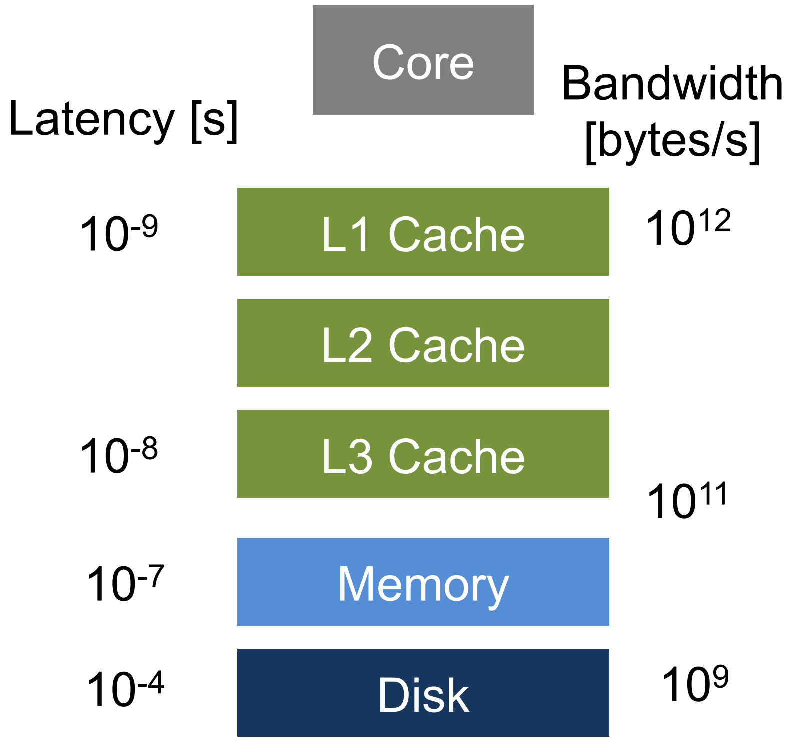
le solution:
e cache
idth



Per-cacheline cycle counts

Memory hierarchy

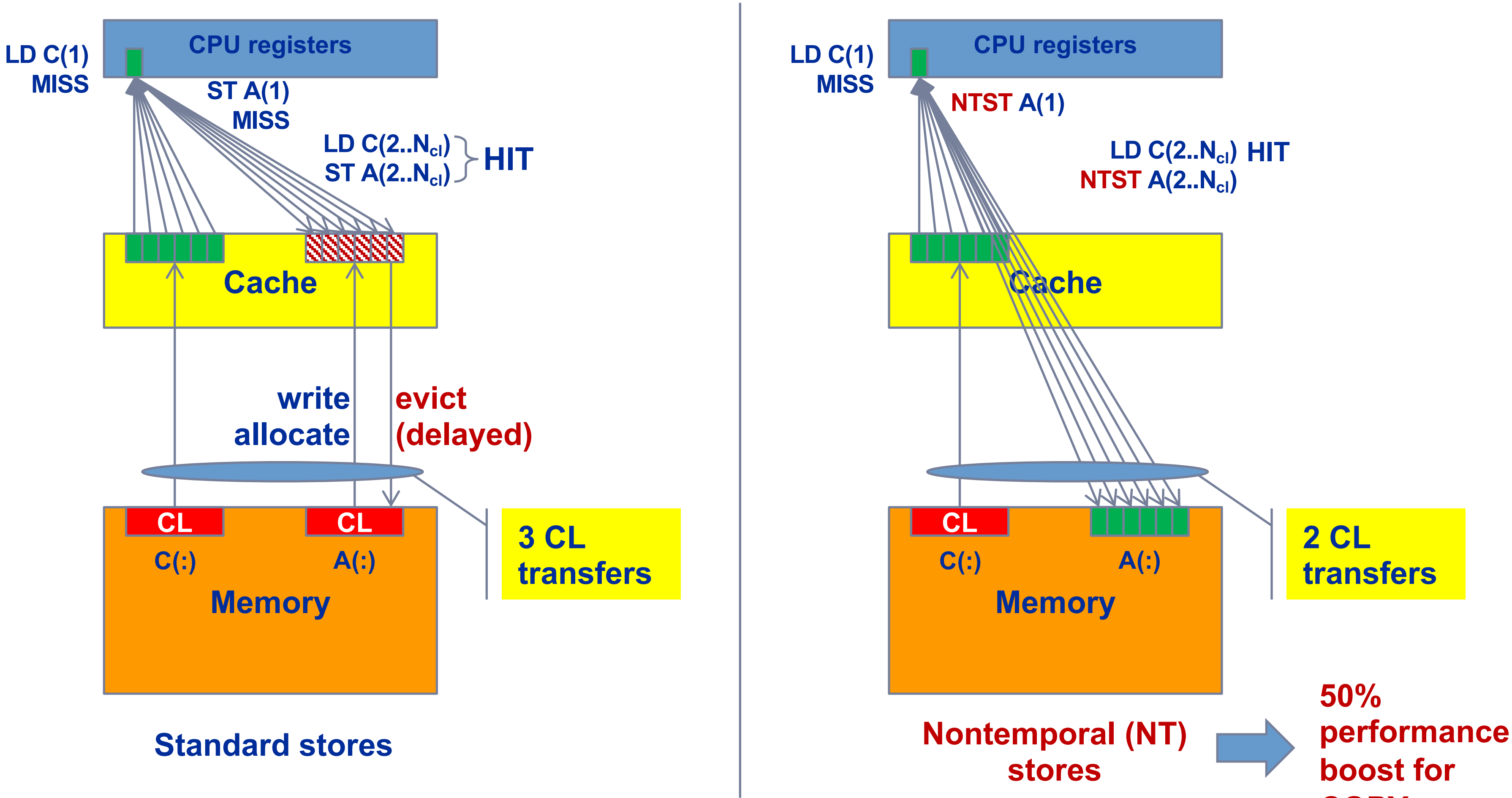
You can **either** build a *small* und *fast* memory **or** a *large* and *slow* memory.



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

Data transfers in a memory hierarchy

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



50% performance boost for COPY

Technologies Driving Performance

Technology	1991	1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018
Clock	33 MHz				200 MHz				1.1 GHz	2 GHz						3.8 GHz			3.2 GHz		2.9 GHz		2.7 GHz		1.9 GHz		1.7 GHz	
ILP																												
SMT												SMT2								SMT4				SMT8				
SIMD									SSE		SSE2										AVX						AVX512	
Multicore																2C	4C			8C			12C	15C	18C	22C	28C	
Memory												3.2 GB/s				6.4 GB/s		12.8 GB/s	25.6 GB/s		42.7 GB/s			60 GB/s			128 GB/s	

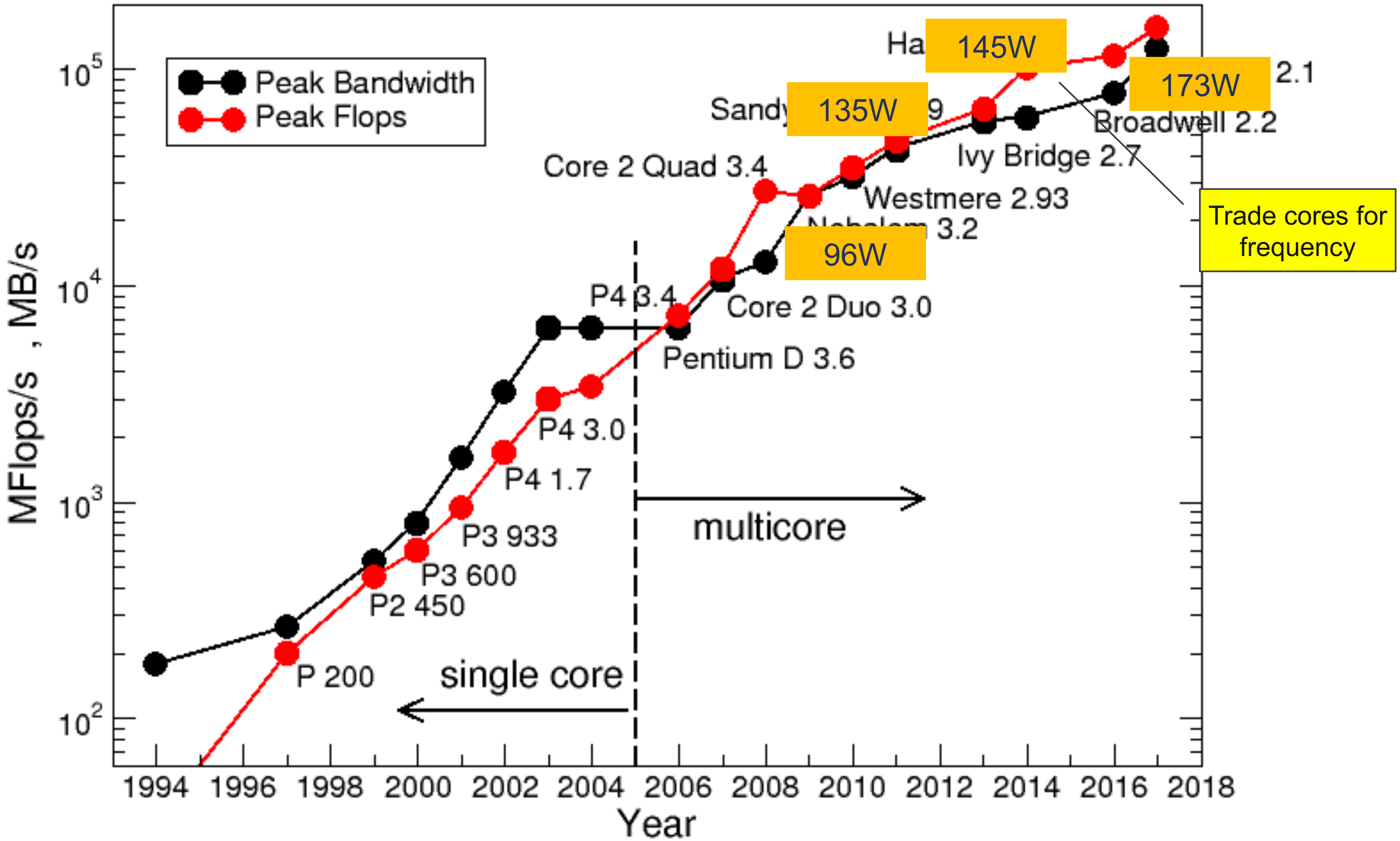
ILP **Obstacle**: Not more parallelism available

Clock **Obstacle**: Power/Heat dissipation

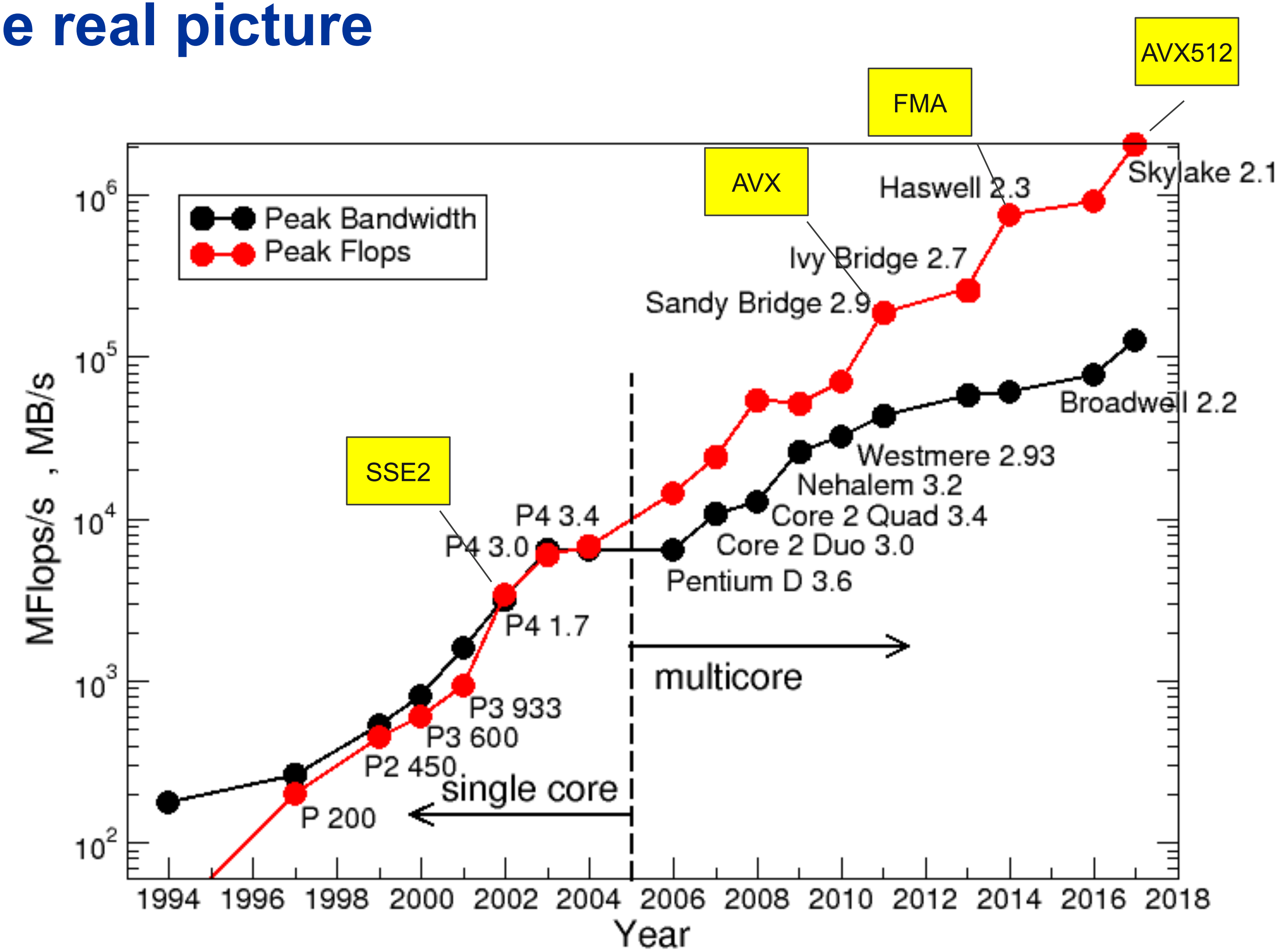
Multi- Manycore **Obstacle**: Getting data to/from cores

SIMD **Obstacle:** Power

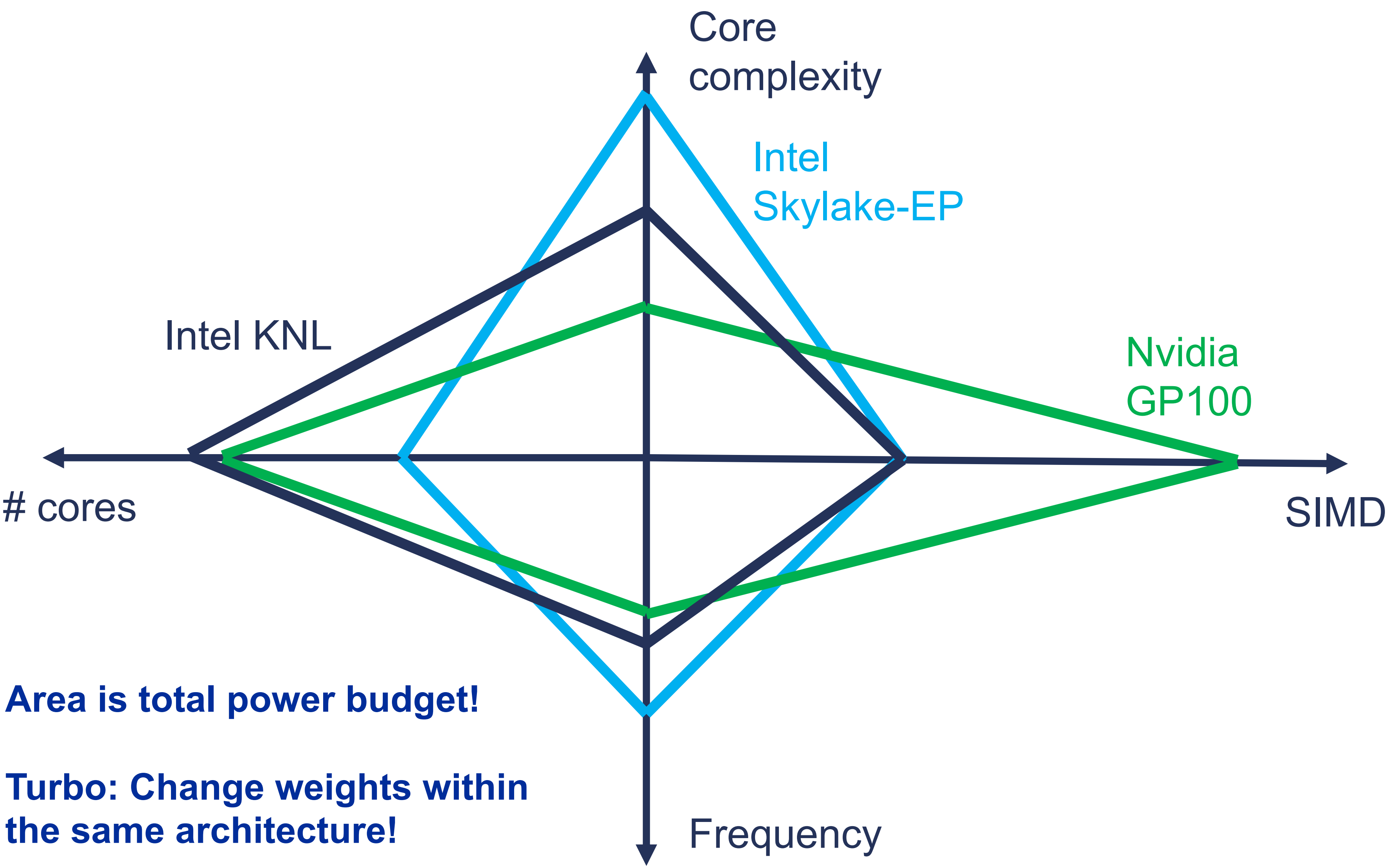
History of Intel chip performance



The real picture



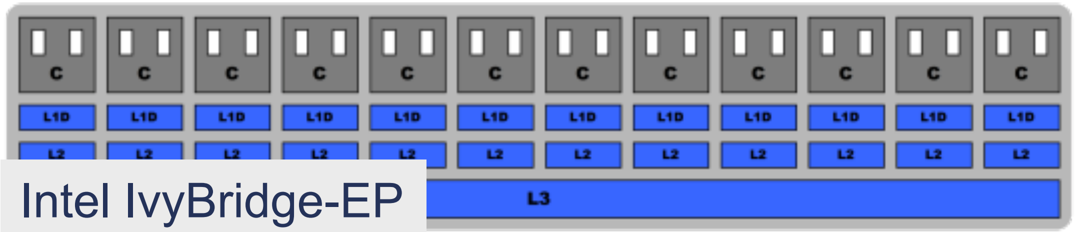
Finding the right compromise



Area is total power budget!

Turbo: Change weights within the same architecture!

The driving forces behind performance 2012



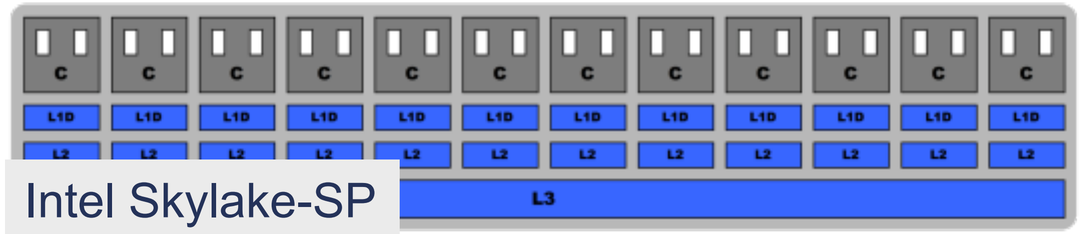
$$P = n_{core} * F * S * \nu$$

	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



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

	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

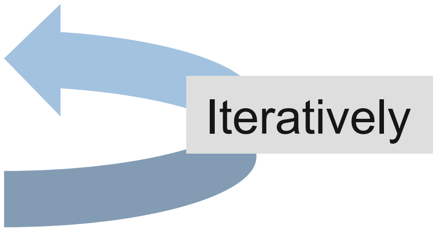
PATTERN BASED PERFORMANCE ENGINEERING

Best Practices

Basic PE Process

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
- Develop a well founded performance expectation
- Deliberately decide on optimizations

Best practices for benchmarking

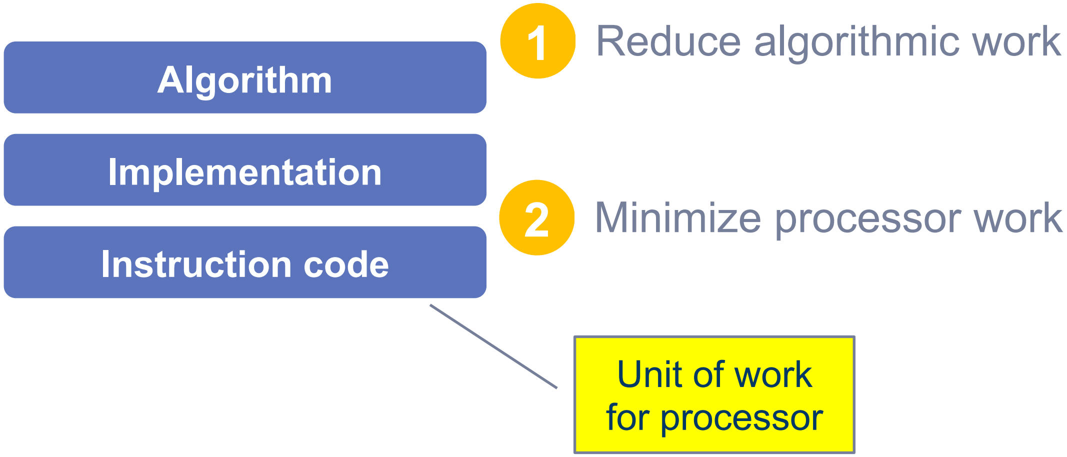
- **Preparation**
  - Reliable timing (minimum time which can be measured?)
  - Document code generation (flags, compiler version)
  - Get access to an exclusive system
  - System state (clock speed, turbo mode, memory, caches)
  - Consider to automate runs with a script (shell, python, perl)

- **Doing**
  - Affinity control
  - Check: Is the result reasonable?
  - Is result deterministic and reproducible?
  - Statistics: Mean, Best ?
  - Basic variants: Thread count, affinity, working set size

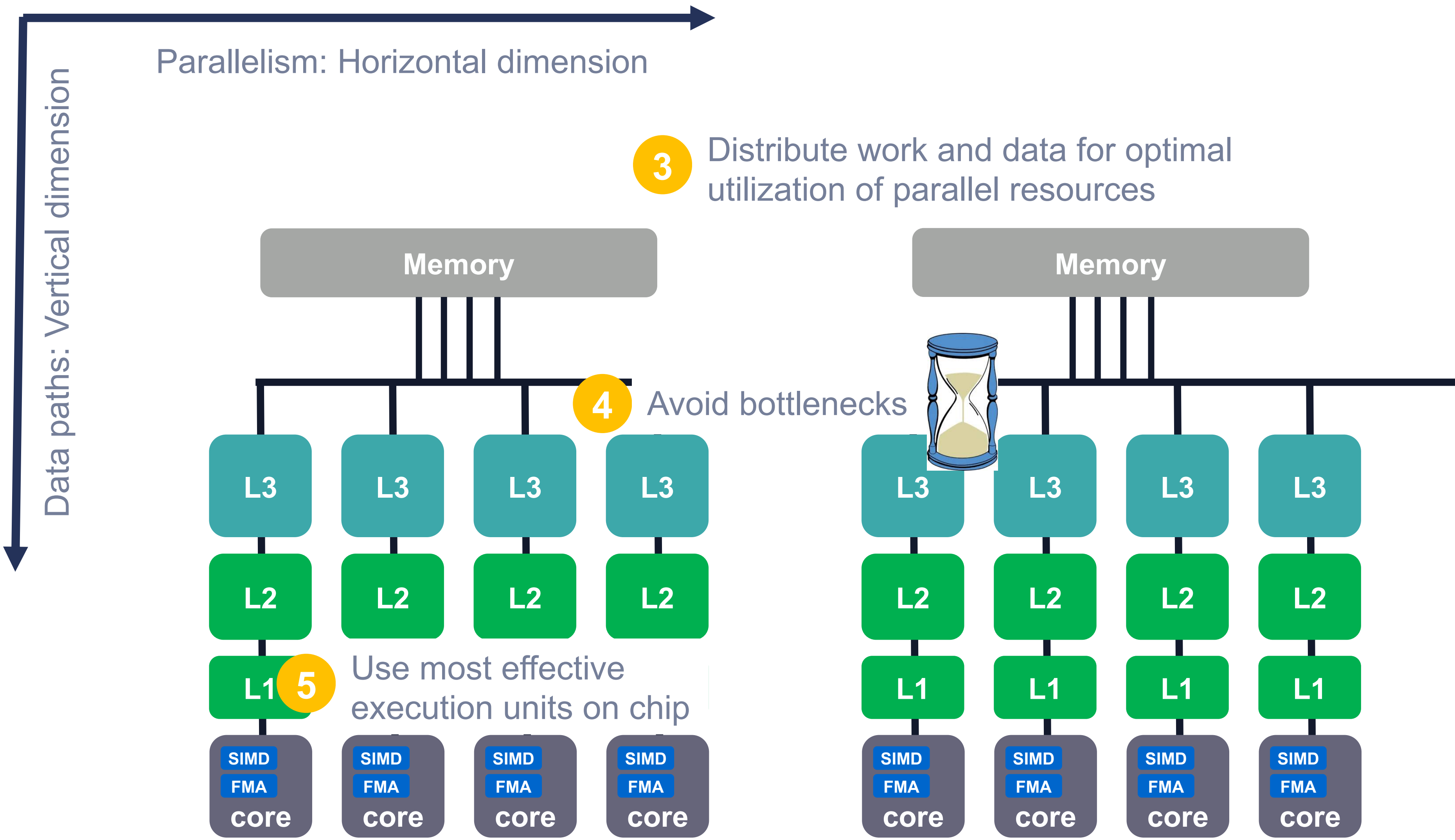
Performance Engineering Tasks: Software

Optimizing software for a specific hardware requires to align several orthogonal.

On the software side it is mostly about reducing algorithmic and processor work. Still decisions here may also restrict the options on the hardware side.



Performance Engineering Tasks: Hardware



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!

Performance Engineering Process: Analysis

Algorithm/Code Analysis

Hardware/Instruction set architecture

Microbenchmarking

Application Benchmarking

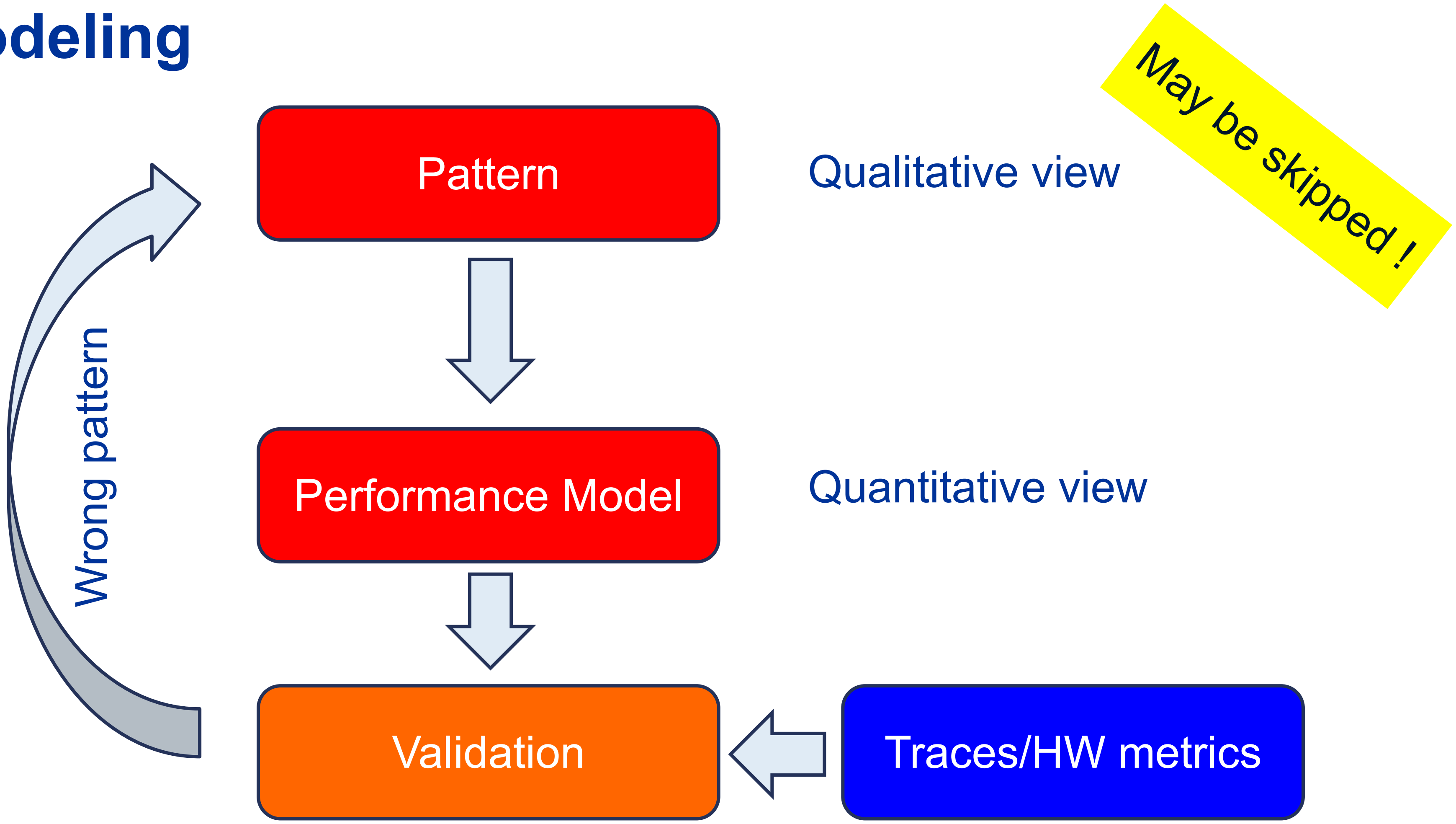
The set of input data indicating a pattern is its **signature**

Pattern

Performance patterns are typical performance limiting motifs

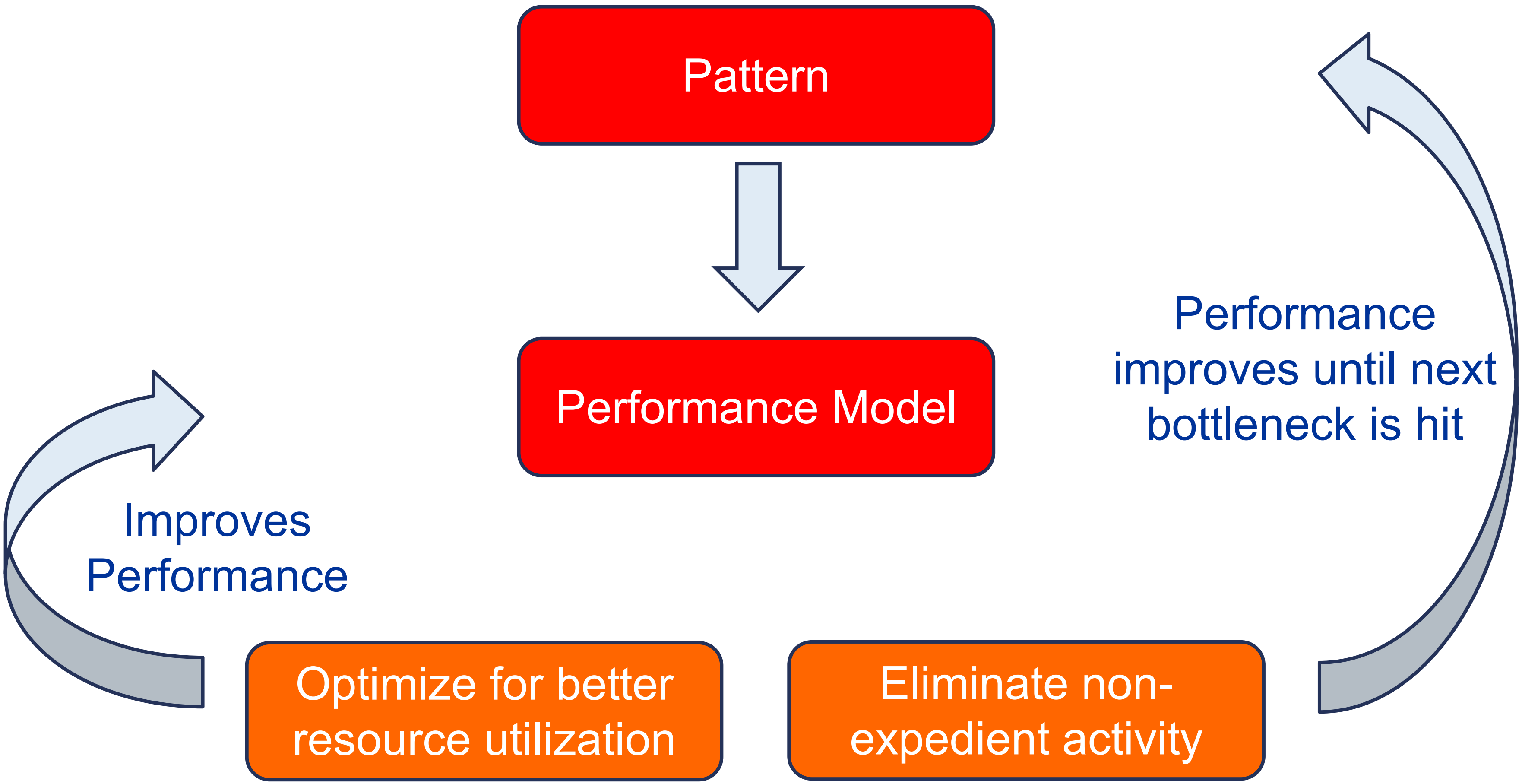
Step 1 **Analysis:** Understanding observed performance

Performance Engineering Process: Modeling



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

Performance Engineering Process: Optimization



Step 3 **Optimization:** Improve utilization of available 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)

Patterns conclusion

- Pattern signature = performance behavior + hardware metrics
  - Hardware metrics alone are almost useless without a pattern
- Patterns are applied hotspot (loop) by hotspot
- Patterns map to typical execution bottlenecks
- Patterns are extremely helpful in classifying performance issues
  - The first pattern is always a hypothesis
  - Validation by tanking data (more performance behavior, HW metrics)
  - Refinement or change of pattern
- Performance models are crucial for most patterns
  - Model follows from pattern

References

Book:

- G. Hager and G. Wellein: Introduction to High Performance Computing for Scientists and Engineers. CRC Computational Science Series, 2010. ISBN 978-1439811924 http://www.hpc.rrze.uni-erlangen.de/HPC4SE/

Papers:

- J. Hammer, G. Hager, J. Eitzinger, and G. Wellein: Automatic Loop Kernel Analysis and Performance Modeling With Kerncraft. Proc. PMBS15, the 6th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, in conjunction with ACM/IEEE Supercomputing 2015 (SC15), November 16, 2015, Austin, TX. DOI: 10.1145/2832087.2832092, Preprint: arXiv:1509.03778
- 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 and Computation: Practice and Experience (2015). DOI: 10.1002/cpe.3489 Preprint: arXiv:1304.7664
- H. Stengel, J. Treibig, G. Hager, and G. Wellein: Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory model. Proc. ICS15, DOI: 10.1145/2751205.2751240, Preprint: arXiv:1410.5010
- G. Hager, J. Treibig, J. Habich and G. Wellein: Exploring performance and power properties of modern multicore chips via simple machine models. Computation and Concurrency: Practice and Experience (2013). DOI: 10.1002/cpe.3180, Preprint: arXiv:1208.2908

References

Papers continued:

- J. Treibig, G. Hager and G. Wellein: Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering. Workshop on Productivity and Performance (PROPER 2012) at Euro-Par 2012, August 28, 2012, Rhodes Island, Greece. DOI: 10.1007/978-3-642-36949-0_50. Preprint: arXiv:1206.3738
- J. Treibig, G. Hager, H. 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, (published online before print). DOI: 10.1177/1094342012442424
- J. Treibig, G. Hager and G. Wellein: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. Proc. PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, September 13, 2010. DOI: 10.1109/ICPPW.2010.38. Preprint: arXiv:1004.4431
- J. Treibig, G. Wellein and G. Hager: Efficient multicore-aware parallelization strategies for iterative stencil computations. Journal of Computational Science 2 (2), 130-137 (2011). DOI 10.1016/j.jocs.2011.01.010
- J. Treibig, G. Hager and G. Wellein: Multicore architectures: Complexities of performance prediction for Bandwidth-Limited Loop Kernels on Multi-Core Architectures. DOI: 10.1007/978-3-642-13872-0_1, Preprint: arXiv:0910.4865.