Introduction to High-Performance Computing

Session 07

Performance Optimization

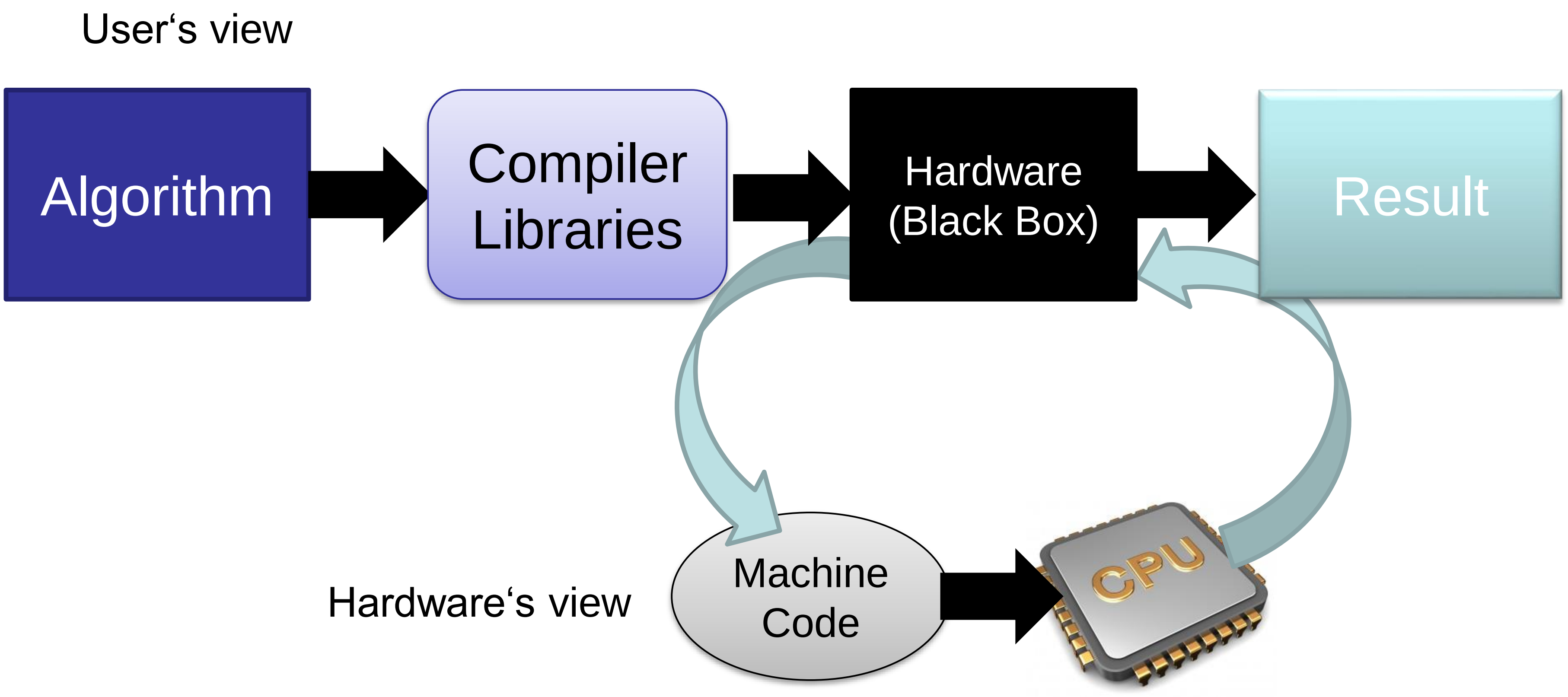
Performance Modelling

- the following slides are based on

http://moodle.rrze.uni-erlangen.de/course/view.php?id=311&username=guest&password=guest

  - 2-day course during MCS Summer School 2014 given by Georg Hager
  - 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/

Computer Software and Hardware



User's view

Algorithm

Compiler
Libraries

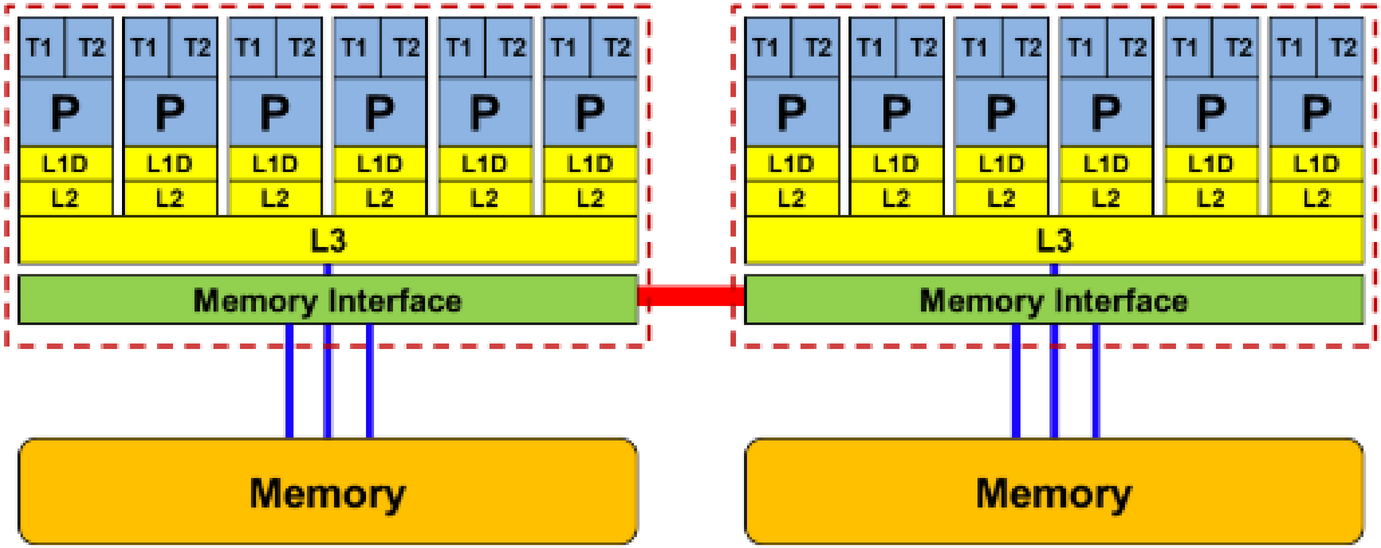
Hardware
(Black Box)

Result

Hardware's view

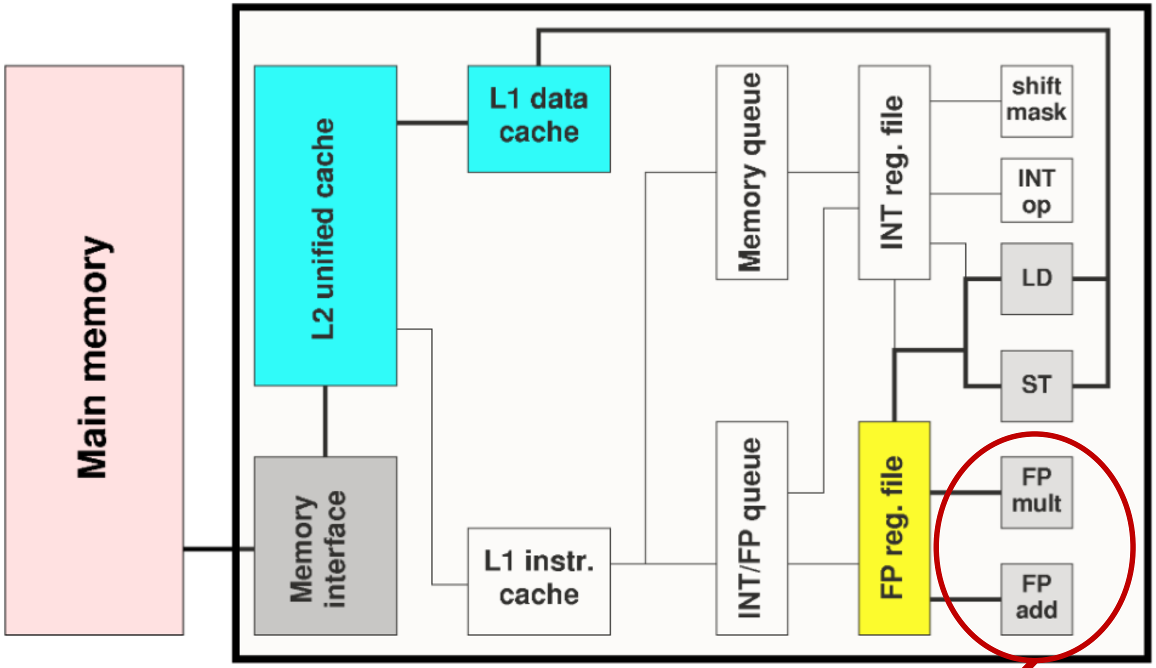
Machine
Code

Modern Computer Architecture



- today: dual-socket node
  - multiple cores per socket/CPU
  - ccNUMA architecture
  - socket interconnect

Detailed View Compute Core



Not shown: most of the control unit, e.g. instruction fetch/decode, branch prediction,...

execution units
(shown only for FP)

Execution Units Broadwell

- two EUs for FP instructions

EU 0	EU 1
FP FMA	FP FMA
FP MUL	FP MUL
FP DIV	FP ADD

- each EU can execute one FP instruction at a time

Example: Divide Throughput

- in the **`Pi.cpp`** code the function **`f(x)`** has one division

```
// define f so that integral of f from 0 to 1 is Pi
double f(const double x) {
  return (4.0/(1.0+x*x));
}
```

  - division is the dominant operation (other instructions can be hidden)
  - for n evaluations of f we get t $= n \cdot \frac{c}{\nu}$
  - Broadwell CPUs need $c = 5$ cycles/division (throughput) and assuming turbo mode (clock speed $\nu = 2.5\mathrm{GHz}$) we would expect $t = 0.2s$ for $n = 10^8$

Execution of Instructions

- programmer‘s view:

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

  - user work:
    N Flops (ADDs)

- hardware‘s view:

```
load r1 = A(i)
load r2 = B(i)
add r1 = r1 + r2
store A(i) = r1
inc i
branch top if i<N
```

programm performs computation, FLOP is the basic work done
processor executes instructions, instructions is the basic work done

Basic Compute Resources

- instruction execution
  - primary resource for computations, hardware is designed to increase instruction throughput as much as possible
  - difficult for general purpose computing, what is a typical workload?
- data movement
  - consequence of instruction execution
  - in the example two loads and one store (double 24 byte)

What is the bottleneck of an application?

Flop/s vs. Memory Bandwidth

- a floating-point operation (Flop) is the basic unit of work
  - theoretical peak performance Intel Xeon E5-2650 v4

$$P_{\text{peak}} = 422.5\ \text{GFlop/s}$$

  - equivalent to $16\ \text{Flop/(core} \cdot \text{cy)}$
- memory bandwidth
  - maximum for Intel Xeon E5-2650 v4 is $76.8\ \text{GB/s}$

(https://ark.intel.com/products/91767/Intel-Xeon-Processor-E5-2650-v4-30M-Cache-2_20-GHz)

  - equivalent to $35\ \text{Byte/cy}$

(more info: http://sites.utexas.edu/jdm4372/tag/memory-bandwidth/)

Example Bandwidth Limited Execution

- consider the vector-triad

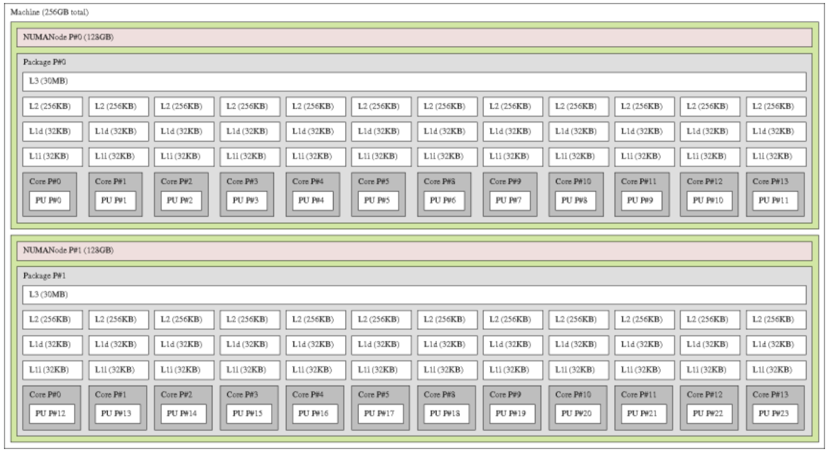
```
for (j=0; j<STREAM_ARRAY_SIZE; j++)
     a[j] = b[j]+scalar*c[j];
```

  - included in the STREAM benchmark (see https://www.cs.virginia.edu/stream/)
  - 2 Flop/iteration and 24 Byte/iteration
  - at $16\ \mathrm{Flop/cy}$ on a single core $192\ \mathrm{Byte/cy}$ are needed

➔ memory bandwidth is the limiting factor here

Hardware Locality

- compute nodes are increasingly complex
  - ccNUMA architectures
- the **`hwloc`** library provides some tools to (https://www.open-mpi.org/projects/hwloc/)
  - obtain information about the node topology (**`lstopo`**)
  - bind processes to specific cores/sockets/…
  - binding/pinning of threads may improve performance (**`hwloc-bind`** … **`<command>`**)
  - difficult to decide, e.g. is it better to use neighboring cores or different sockets?



STREAM Benchmark

https://www.cs.virginia.edu/stream/

- simple tool to measure memory bandwidth
  - timing of bandwidth-limited vector operations
- some results
  - single core bandwidth is about $20\ \mathrm{GB}/s$
  - maximum bandwidth measured is about $64\ \mathrm{GB}/s$ per socket and $128\ \mathrm{GB}/s$ per node (two sockets)
  - about half of the cores are need to get (close to) maximum bandwidth

Example

- 3d „Stencil“ update (Jacobi)

```
// serial
for (int i=1; i<Ni; i++)
  for (int j=1; j<Nj; j++)
    for (int k=1; k<Nk; k++)
      y[i][j][k] = w * (   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] );
```

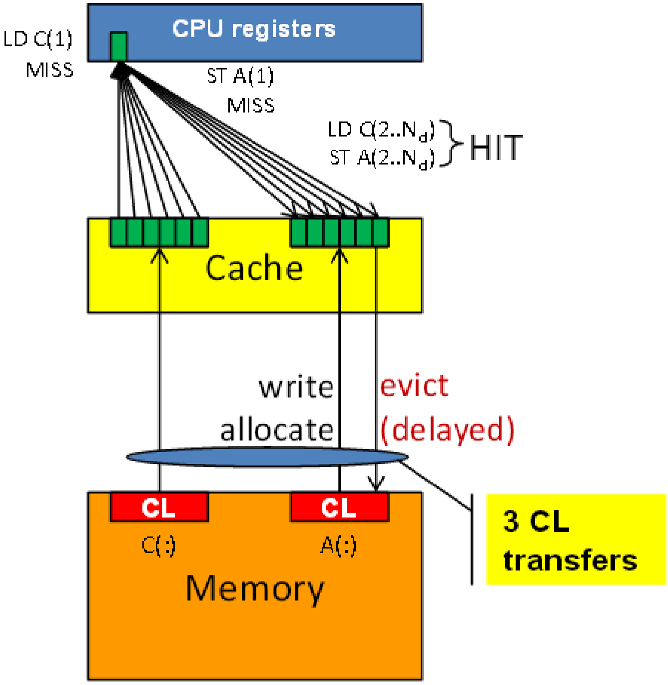
note that the order of the loops is important
(and depends on the ordering of multi-dimensional arrays in memory)

Memory Access Patterns

- **Caches help with getting instructions and data to the CPU “fast”**
- **How does data travel from memory to the CPU and back?**

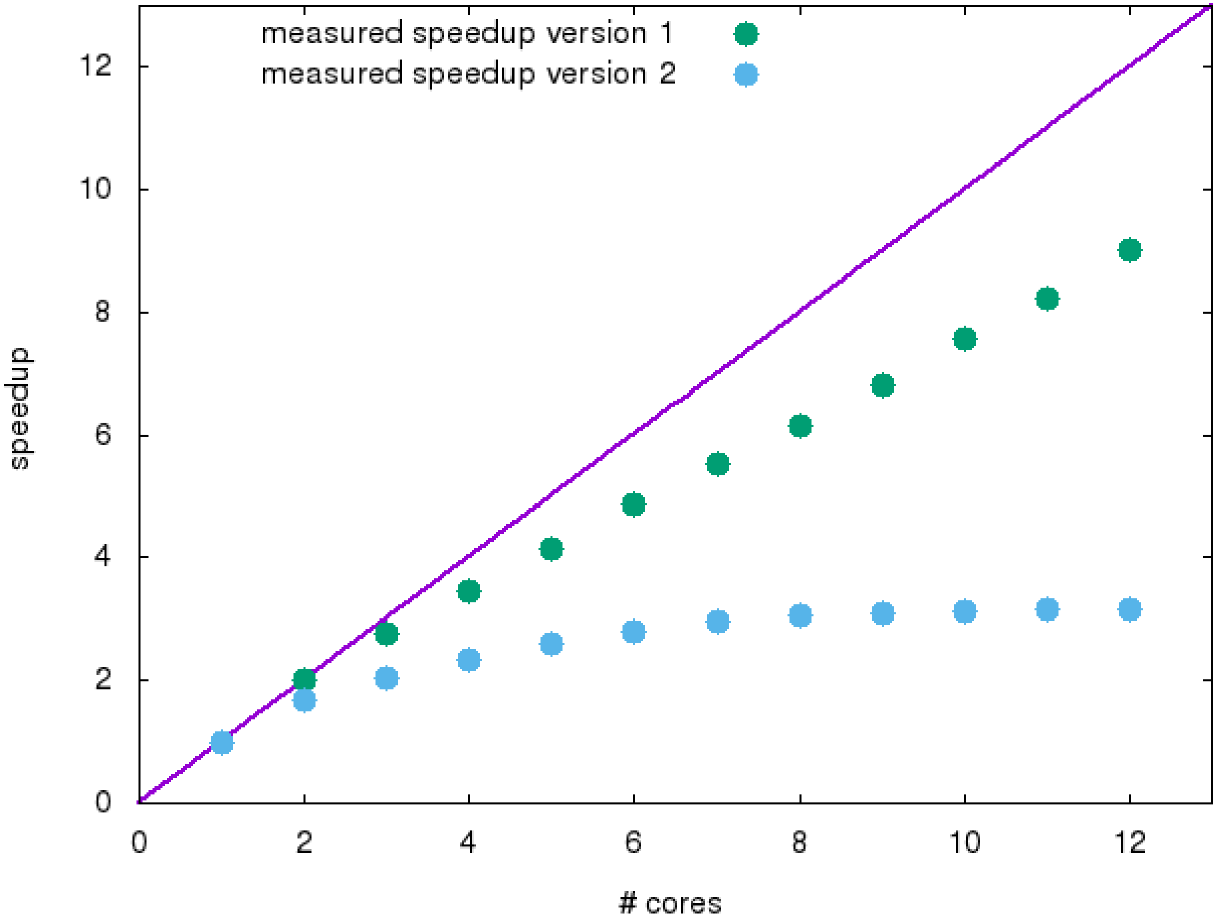
- Remember: Caches are organized in cache lines (e.g., 64 bytes)
- Only complete cache lines are transferred between memory hierarchy levels (except registers)
- MISS: Load or store instruction does not find the data in a cache level
  → CL transfer required

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



Parallel Speedup

version 1, very good scaling
version 2, almost no scaling



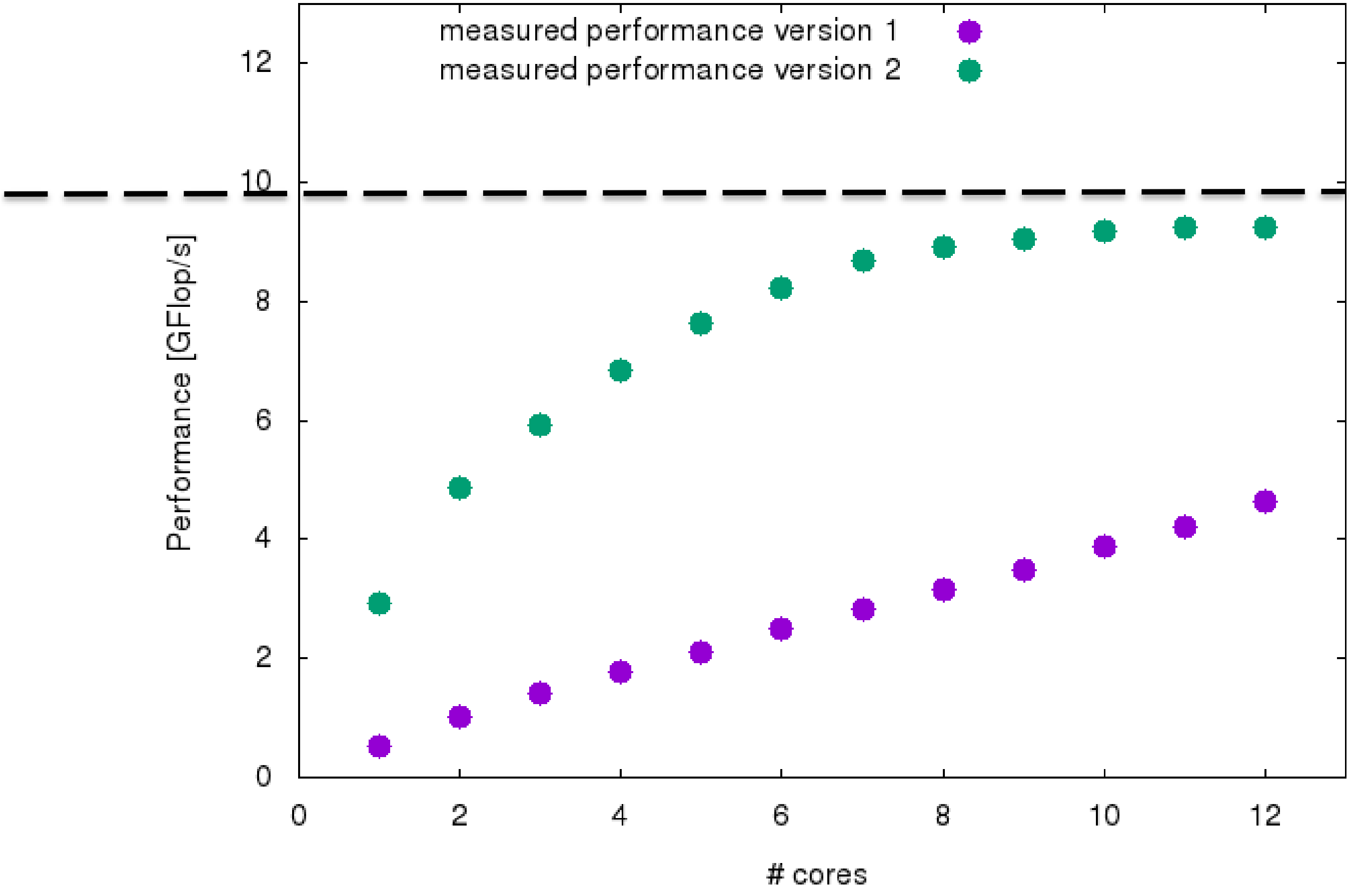
Parallel Performance

performance of version 2 is better by factor of few

bandwidth limit

48 Byte/iter

≙ 9.6 GFlop/s



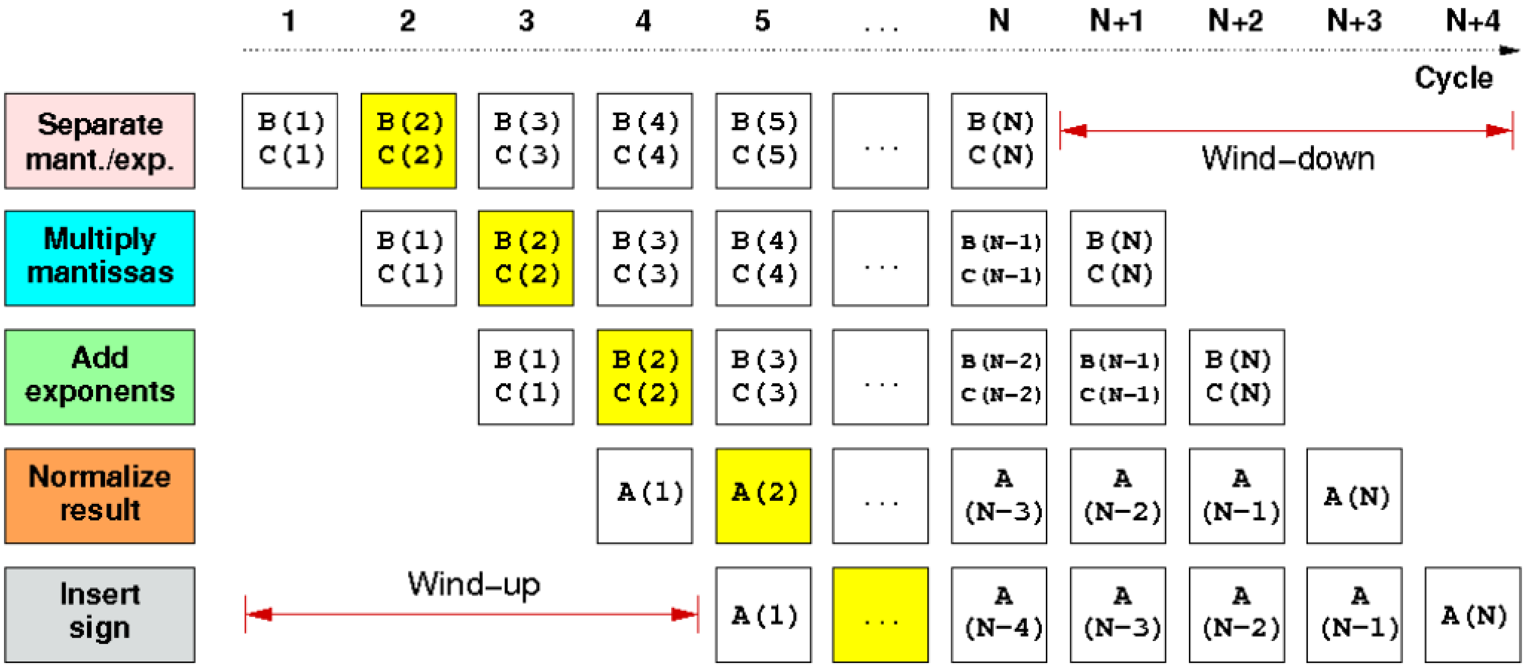
How is the Hardware optimized for performance?

- speedup memory access with cache (see before)
- pipelining of arithmethic units
- instruction pipeline
- instruction level parallelism
- simultaneous multi-threading (SMT)
- SIMD processing

Pipelining

- **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**

Pipelinig – 5 stage Multiplication

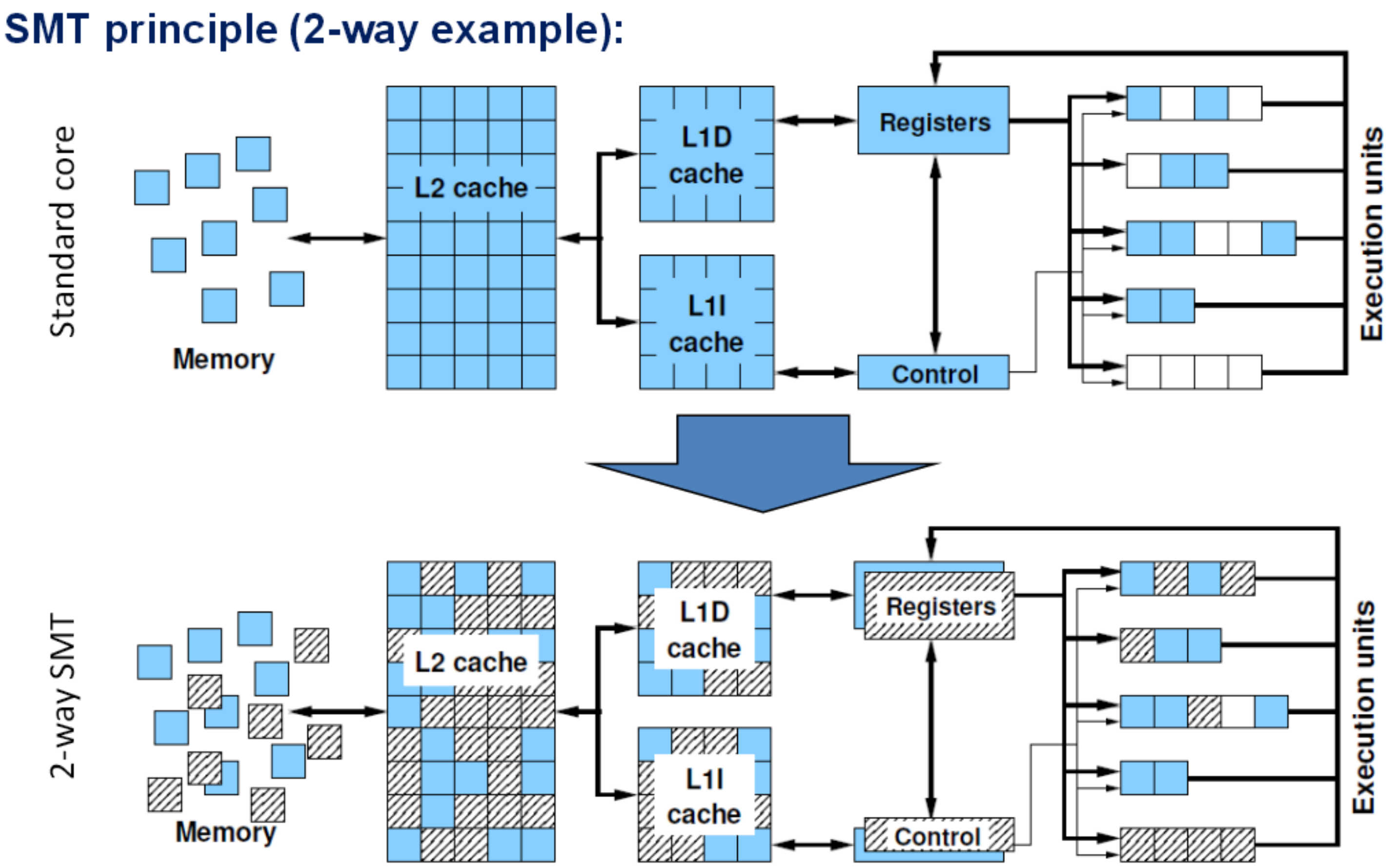


	1	2	3	4	5	...	N	N+1	N+2	N+3	N+4
Separate mant./exp.	B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	B(5) C(5)	...	B(N) C(N)	Wind–down			
Multiply mantissas		B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	...	B(N−1) C(N−1)	B(N) C(N)			
Add exponents			B(1) C(1)	B(2) C(2)	B(3) C(3)	...	B(N−2) C(N−2)	B(N−1) C(N−1)	B(N) C(N)		
Normalize result				A(1)	A(2)	...	A(N−3)	A(N−2)	A(N−1)	A(N)	
Insert sign	Wind–up				A(1)	...	A(N−4)	A(N−3)	A(N−2)	A(N−1)	A(N)

First result is available after 5 cycles (=latency of pipeline)!

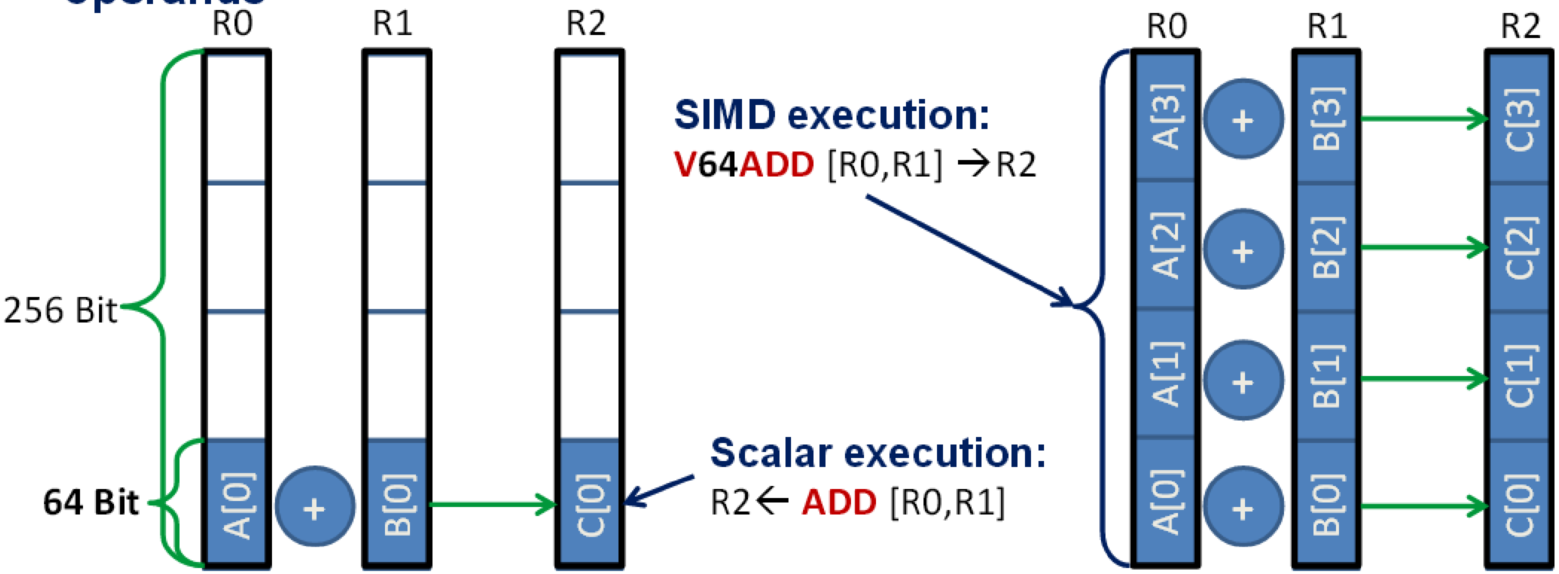
Wind-up/-down phases: Empty pipeline stages

simultaneous multi-threading (SMT)



SIMD processing

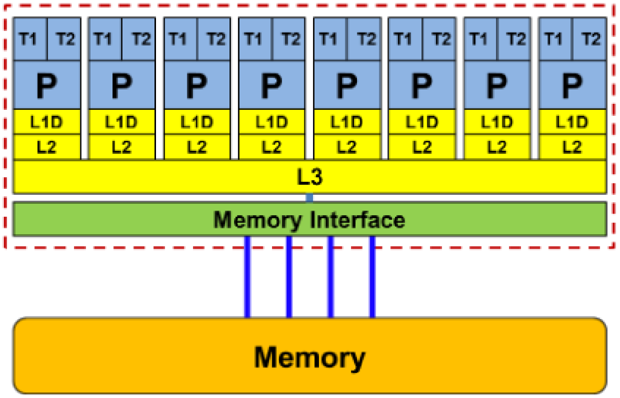
- **Single Instruction Multiple Data (SIMD) operations allow the concurrent execution of the same operation on "wide" registers**
- **x86 SIMD instruction sets:**
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX: register width = 256 Bit → 4 double precision floating point operands
- **Adding two registers holding double precision floating point operands**



SIMD execution:
V64ADD [R0,R1] → R2

Scalar execution:
R2 ← **ADD** [R0,R1]

Processor Peak Performance



Intel Xeon „Broadwell“
E5-2650 v4

Floating Point (FP) Performance:

$$P = n_{\text{core}} \cdot F \cdot S \cdot \nu$$

n_{core}	number of cores	12
F	FP instructions per cycle (2 FMA)	4
S	FP ops / instruction (256 Bit SIMD registers in AVX2)	4
ν	clock speed (affected by turbo/AVX modes)	2.2 GHz

TOP500 rank 1 (mid-90s)

$\boldsymbol{P = 422.4}$ **GFlop/s** (dp)

But: $\boldsymbol{P = 8.8}$ **GFlop/s** for serial, non-SIMD code

Performance Bottleneck

- many floating point computation on little data
  → bound by the processing speed of the CPU
  - possibly increase number of cores
  - make use of SIMD processing
  - note: recent CPU may have lower clock speed for AVX

- few floating point operation per data
  → bound by memory bandwidth
  - change algorithm/parallelization to make better use of cache
  - increase compute intensity

Examples

- OMP_Pi
  - how many CPU cycles are required for a DIV operation?
- STREAM
  - determine memory bandwidth
- Stencil
  - optimization vs. speedup
  - memory access pattern

measuring/getting optimal performance may require process binding