703308 VO High-Performance Computing
Node-Level Performance

Philipp Gschwandtner

Motivation

- So far, we've discussed MPI, domain decomposition, load balancing, one-sided communication, etc.
  - primarily significant in inter-node performance discussion

- What about intra-node?
  - optimizing instruction set architecture use
  - SIMD/vectorization
  - ccNUMA, data placement, cache optimizations
  - thread/core mappings
  - …?

"node-level" performance

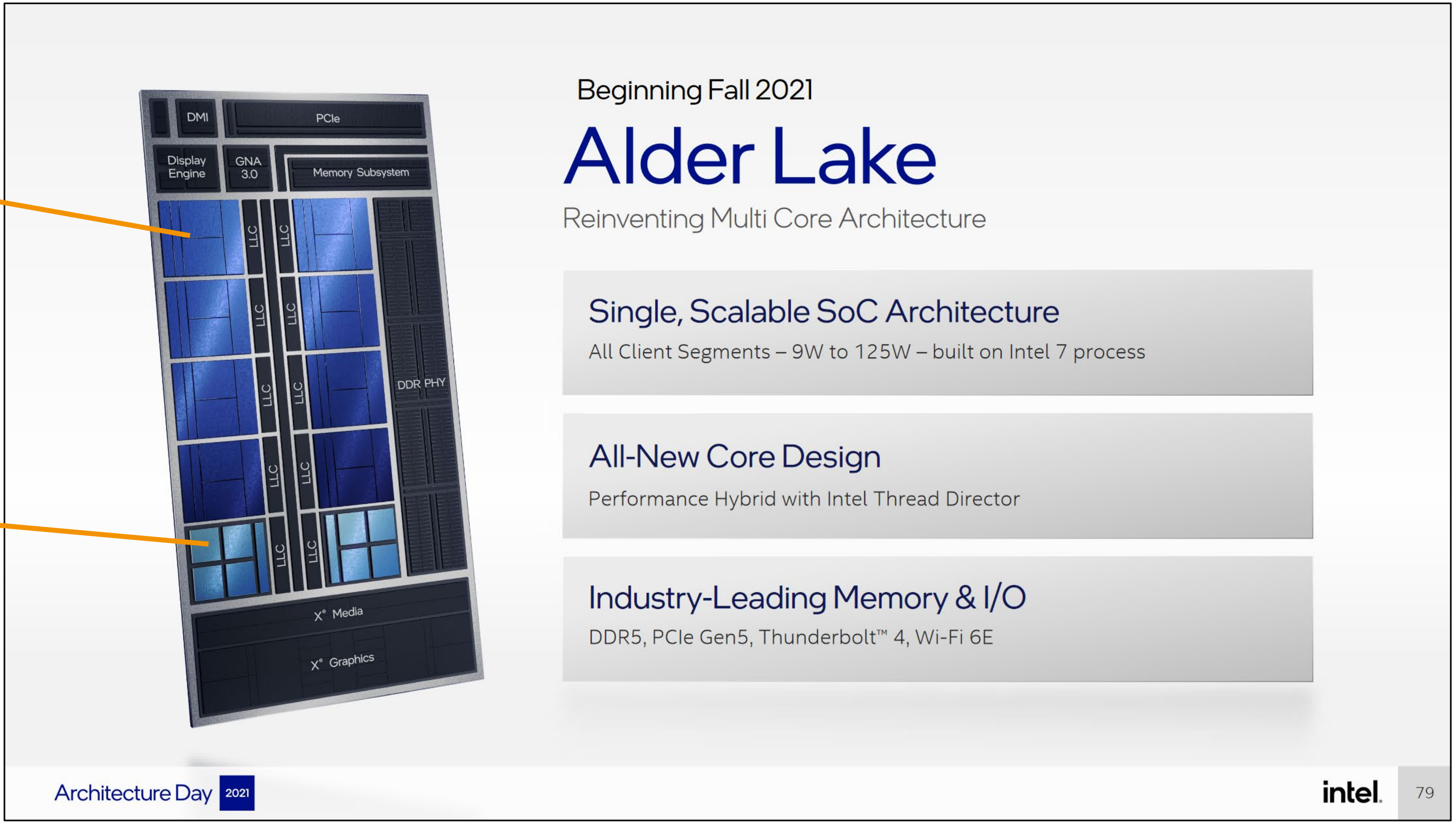
Overview

- On-chip heterogeneity
- Vectorization
- Common node-level pitfalls, esp. with OpenMP

On-chip heterogeneity

P core
(Golden Cove)
up to 5 GHz,
HT, private L2

E core
(Gracemont)
up to 4 GHz, no
HT, shared L2



Beginning Fall 2021

Alder Lake

Reinventing Multi Core Architecture

Single, Scalable SoC Architecture

All Client Segments – 9W to 125W – built on Intel 7 process

All-New Core Design

Performance Hybrid with Intel Thread Director

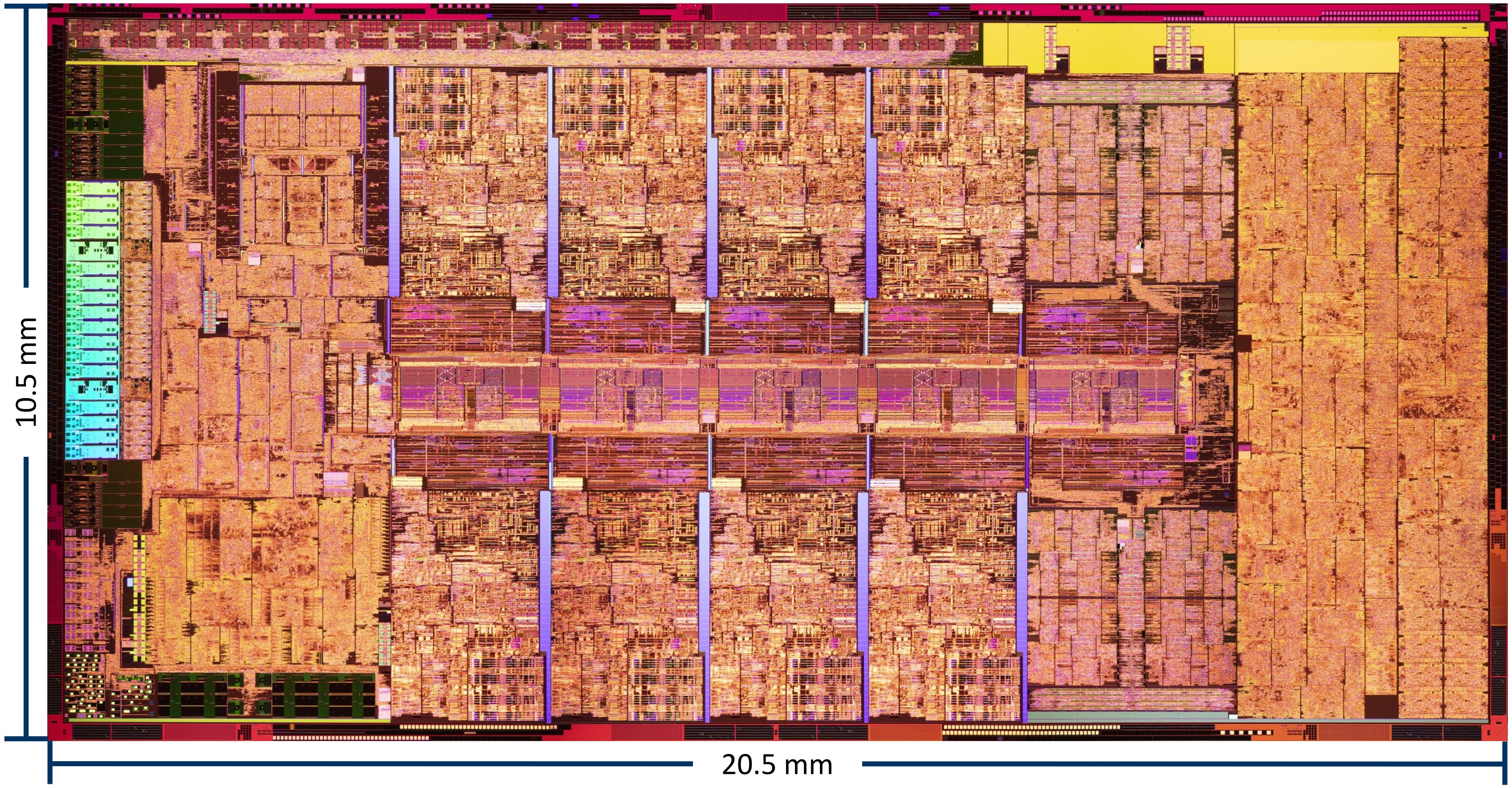
Industry-Leading Memory & I/O

DDR5, PCIe Gen5, Thunderbolt™ 4, Wi-Fi 6E

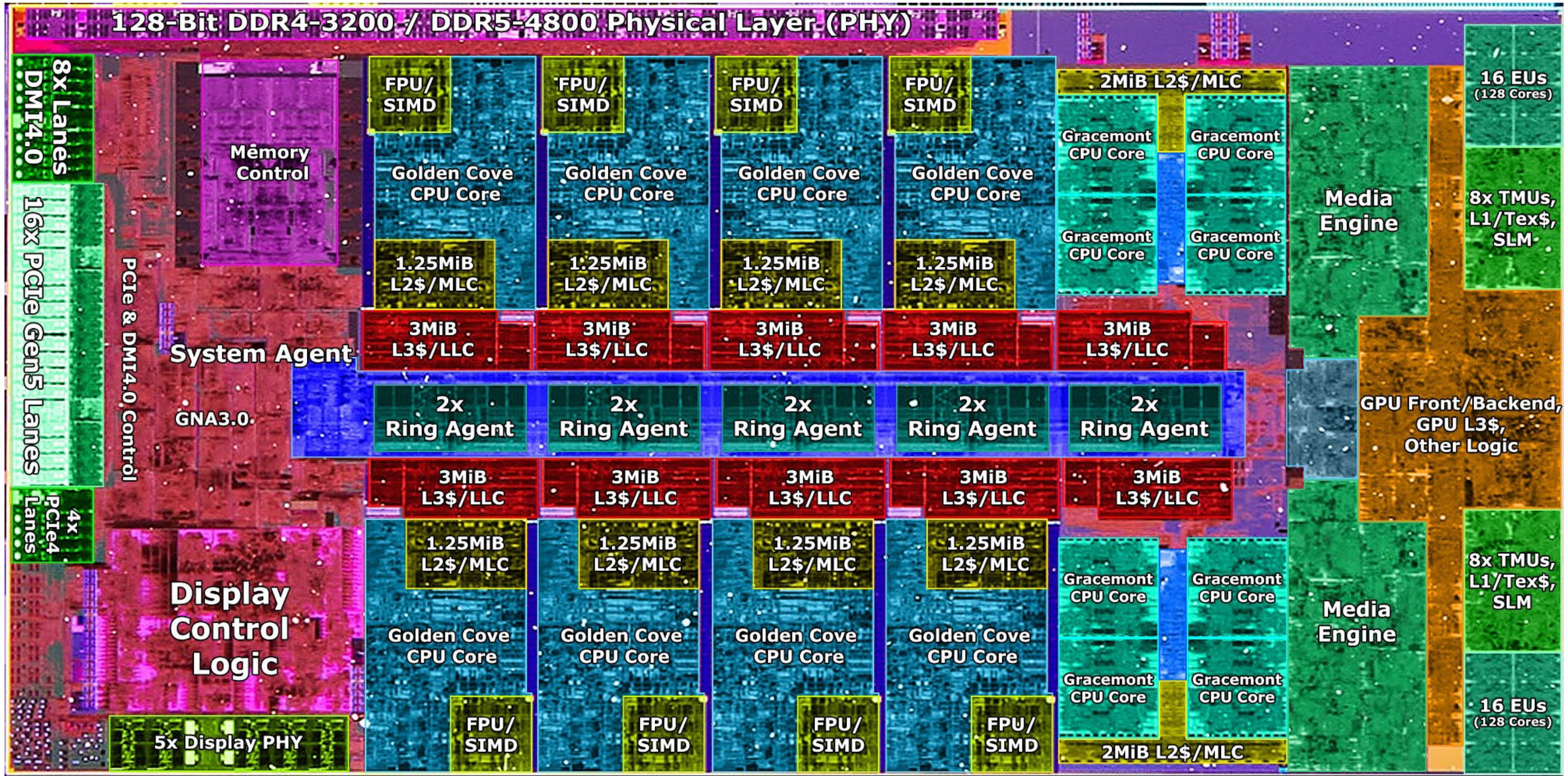
Architecture Day 2021

intel. 79

Alder Lake die shot



Alder Lake die shot annotated



10nm ESF/Intel 7 Alder Lake die shot (~209mm²) from Intel via Andreas Schilling on Twitter:
https://twitter.com/aschilling/status/1453391035577495553

Die shot interpretation by Locuza, October 2021

Alder Lake take-aways

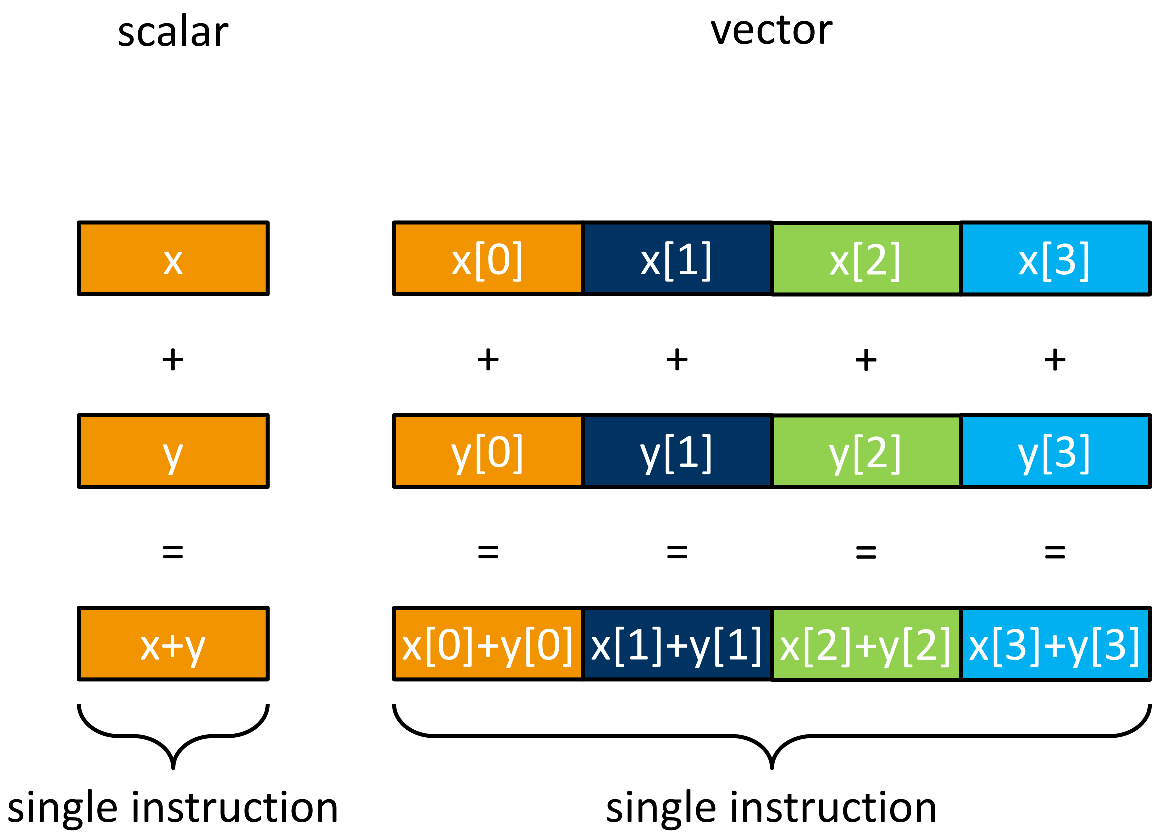
- Affinity keeps gaining significance
  - fast P-cores and efficient E-cores on the same chip
    (note that ARM has been doing this for years with big.LITTLE but for different reasons)
  - OS scheduler and programs need to be aware
  - e.g. previous images show an 8P+8E configuration

- L3 cache layout leads to on-chip NUMA
  - all L3 cache is accessible to every core but not with the same performance
  - has been the case at least since Haswell (~2014)

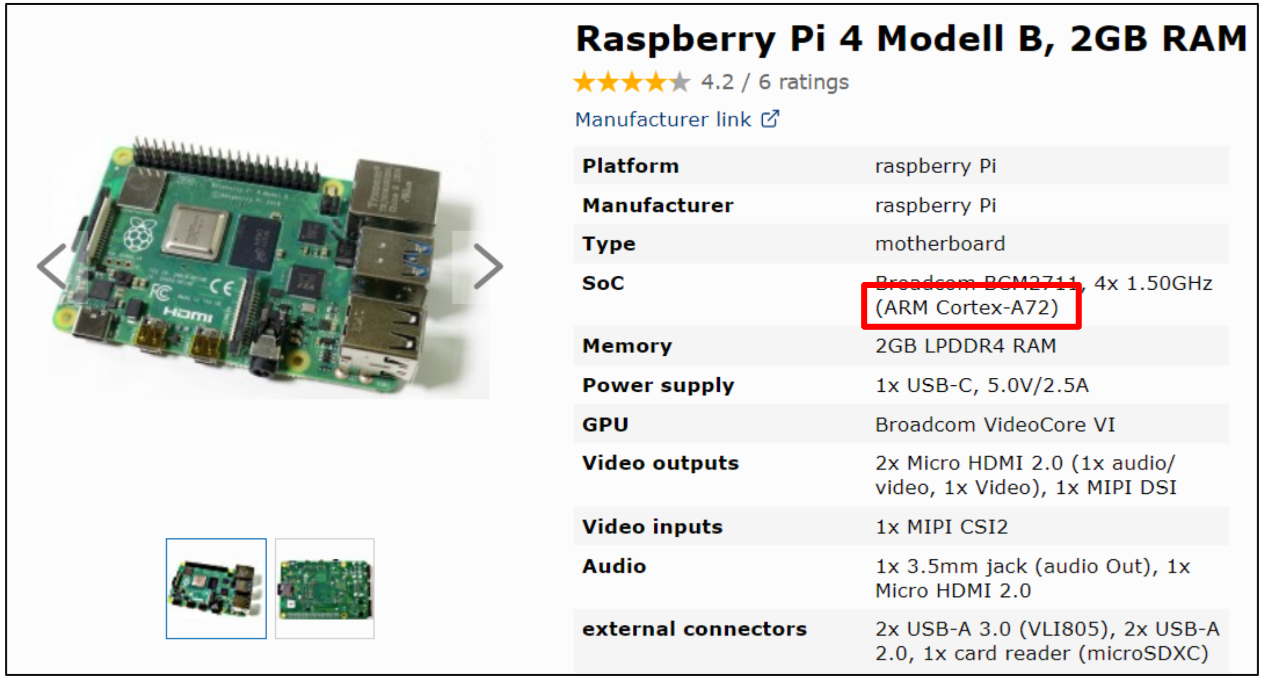
- Vectorization units take up a lot of transistor space

Vectorization

- modern CPUs have vector units
  - allow multiple data per instruction
  - performance gains of up to e.g. 4x (“SIMD width”)
  - no thread- or process parallelism involved!
- available operations and width depend on your software/hardware stack
  - hard to code manually (compiler intrinsics or assembly)



Vectorization is ubiquitous



Raspberry Pi 4 Modell B, 2GB RAM

★★★★★ 4.2 / 6 ratings

Manufacturer link

Platform	raspberry Pi
Manufacturer	raspberry Pi
Type	motherboard
SoC	Broadcom BCM2711, 4x 1.50GHz (ARM Cortex-A72)
Memory	2GB LPDDR4 RAM
Power supply	1x USB-C, 5.0V/2.5A
GPU	Broadcom VideoCore VI
Video outputs	2x Micro HDMI 2.0 (1x audio/ video, 1x Video), 1x MIPI DSI
Video inputs	1x MIPI CSI2
Audio	1x 3.5mm jack (audio Out), 1x Micro HDMI 2.0
external connectors	2x USB-A 3.0 (VLI805), 2x USB-A 2.0, 1x card reader (microSDXC)

ARM Neon
Samsung Gear
S2 3G (~2015)

ARM Neon
Apple M1 (~2020)

ARM SVE
Fujitsu A64FX (Fugaku's CPUs)

Intel AVX2
Alder Lake (~2021)

Vectorization hardware cheat sheet

- Intel/AMD
  - 64 1997: MMX – 64 bit, integer only, 2x 32 bit or 4x 16 bit or …
  - 1999: SSE – 128 bit, adds float, 4x 32 bit
  - 2000: SSE2 – 128 bit, adds double-precision (2x 64 bit) and 4x 32 bit integer
  - 128 2004: SSE3 – incremental update to SSE2, adds DSP and same-register
  - 2006 SSSE3 – supplement to SSE3, adds permutation and fixed-point
  - 2006: SSE4 – adds higher-level features such as dot product or population count
  - 2011: AVX – extension to 256 bit for floats, 3 operands
  - 256 2013: AVX2 – adds 256 bit integer support and fused multiply-add (FMA)
  - 512 2016: AVX-512 – extension to 512 bit, several sets of instructions depending on CPU

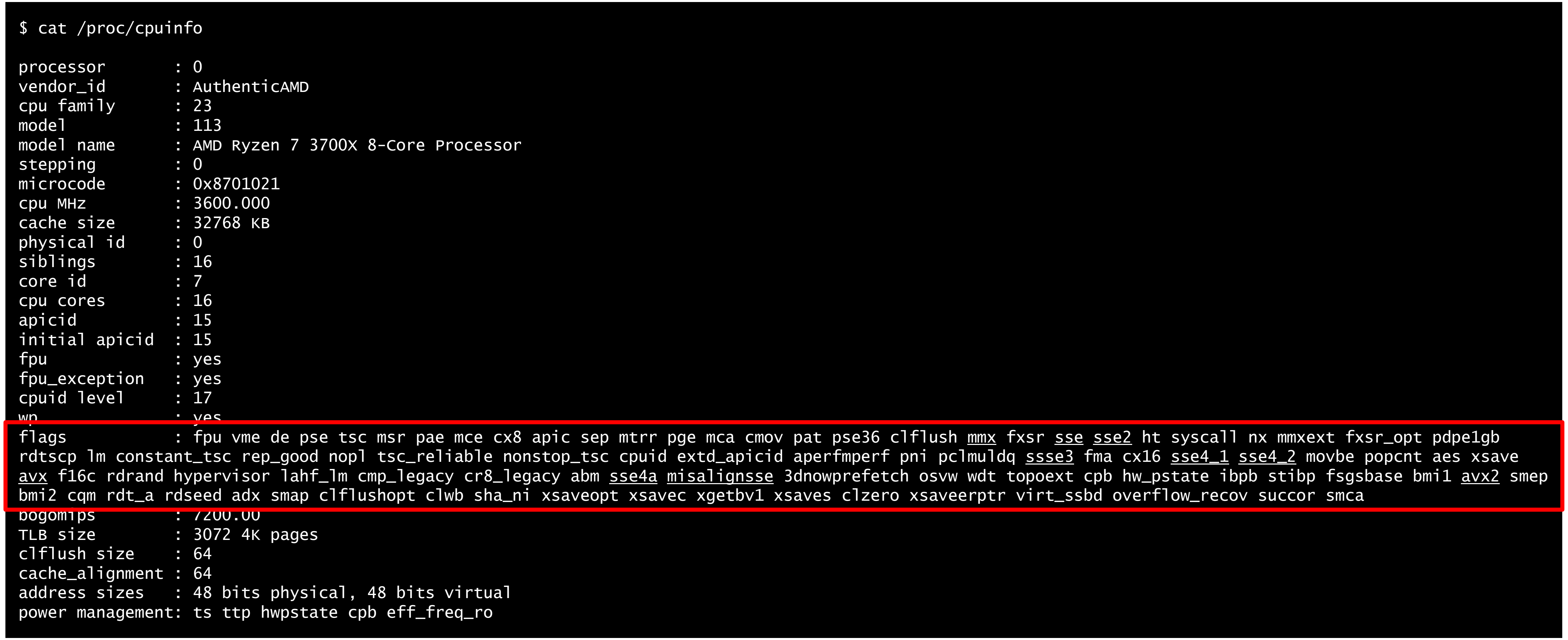
Vectorization hardware cheat sheet cont’d

- ARM
  - Neon: 128 bit width, integer and single-precision float (double-precision for 64 bit CPUs)
  - Helium: light-weight variant of Neon, less registers, aimed at low-power
  - SVE: new & incompatible with Neon, up to 2048 bit width, directly aimed at HPC and ML
  - SVE2: extends instruction set for multimedia, communication (LTE), etc. use cases

- Apple M1
  - implements Neon

- IBM
  - VMX/VMX128: integer and single-precision floats, up to 4x 32 bit, used in e.g. Xbox 360
  - VSX: extends to 2x 64 bit and double-precision

How to check support using `/proc/cpuinfo`



```
$ cat /proc/cpuinfo

processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 113
model name      : AMD Ryzen 7 3700X 8-Core Processor
stepping        : 0
microcode       : 0x8701021
cpu MHz         : 3600.000
cache size      : 32768 KB
physical id     : 0
siblings        : 16
core id         : 7
cpu cores       : 16
apicid          : 15
initial apicid  : 15
fpu             : yes
fpu_exception   : yes
cpuid level     : 17
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb
rdtscp lm constant_tsc rep_good nopl tsc_reliable nonstop_tsc cpuid extd_apicid aperfmperf pni pclmuldq ssse3 fma cx16 sse4_1 sse4_2 movbe popcnt aes xsave
avx f16c rdrand hypervisor lahf_lm cmp_legacy cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw wdt topoext cpb hw_pstate ibpb stibp fsgsbase bmi1 avx2 smep
bmi2 cqm rdt_a rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 xsaves clzero xsaveerptr virt_ssbd overflow_recov succor smca
bogomips        : 7200.00
TLB size        : 3072 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management: ts ttp hwpstate cpb eff_freq_ro
```

CPUID

- Allows you to check for CPU properties and features
  - e.g. CPU vendor, model, family, revision
  - instruction set features (e.g. AVX support)
  - other features (e.g. `constant_tsc`)

- virtually every CPU these days supports the CPUID instruction

```
mov eax, 0x0
cpuid

; output will be:
; ebx: 0x756E6547
; edx: 0x49656E69
; ecx: 0x6C65746E
; spelling „GenuineIntel“ in ASCII
```

Controlling vectorization in software

- automatic vectorization by the compiler
  - e.g. gcc: `-ftree-vectorize -fopt-info-vec-all -march=tigerlake`
  - e.g. MSVC: `/Qvec-report:2 /arch:avx2`

- manual vectorization
  - using OpenMP: `#pragma omp simd ...`
    - Visual Studio: requires 2019 or higher and `/openmp:experimental`
  - using compiler-specific intrinsics in C/C++: `_mm_add_ps(...)`
  - using ISA-specific assembly: `vaddps`

Controlling instruction sets in general

- `-march`
  - assume at minimum this architecture to be available
  - e.g. compiling `-march=tigerlake` and running on a Sandy Bridge will crash your program with SIGILL (illegal instruction), but only if modern instructions are actually emitted by compiler

- `-mtune`
  - optimize for a specific architecture but do not impose as minimum requirement
  - e.g. `-march=sandybridge -mtune=tigerlake`

- `-m<ISA-feature>` or `–mno-<ISA-feature>`
  - enables/disables the specified instruction set, allows more fine-grained use of ISA features compared to –march
  - e.g. `–mavx2`

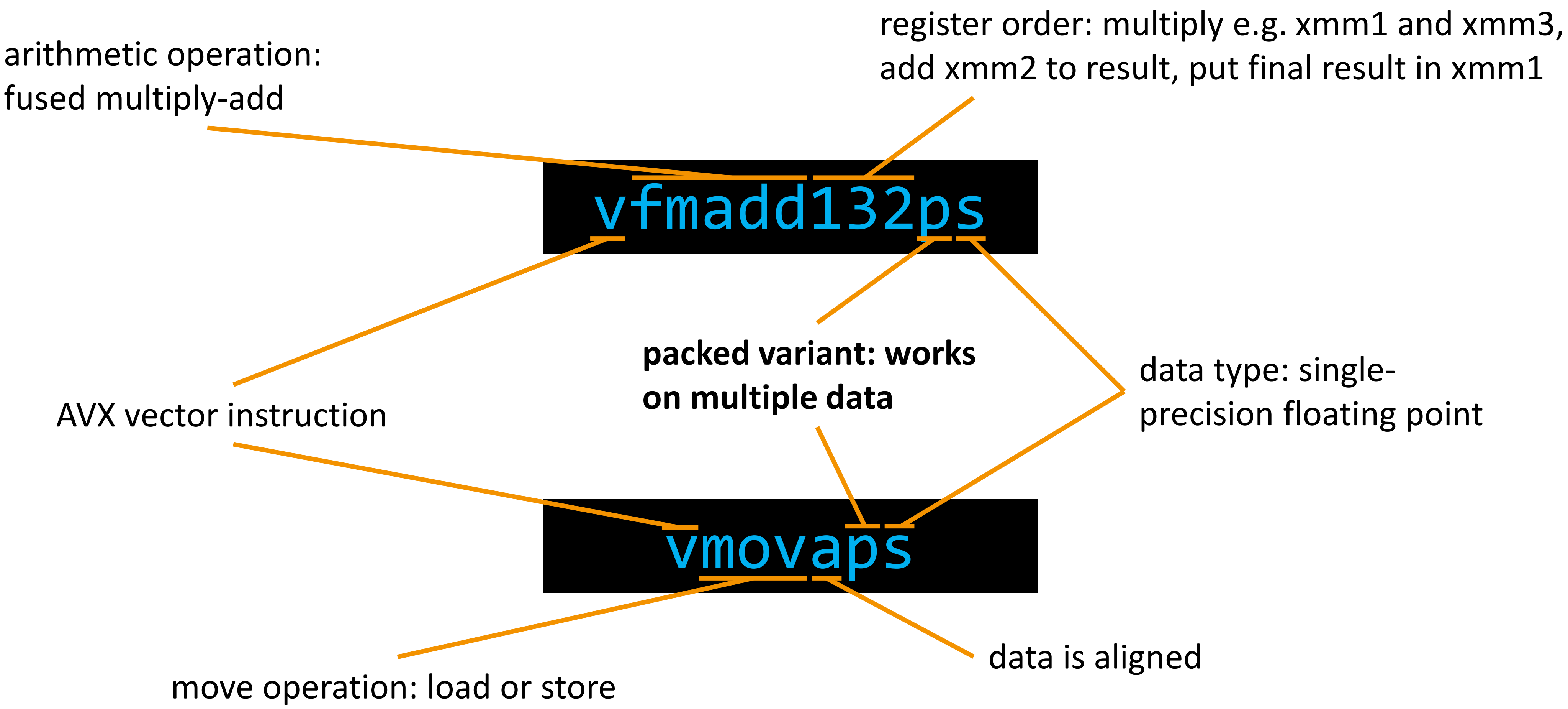
- Two special presets
  - `-march=generic`: target a generic ISA of “current” CPUs (can change between compiler versions!)
  - `-march=native`: target the architecture you are currently compiling on
  - also work for `–mtune`

Verifying vectorization

- indirect means (derived metrics)
  - wall time
  - performance/throughput
  - CPU core clock frequency (AVX)
  - …
- direct means
  - compiler output (=assembly code)
  - vectorization performance counters

- Be aware however, vectorized instructions exist in scalar and packed form
  - packed: working on multiple data
  - scalar: working on single data only
- Packed is what you’re aiming for!

Reading vectorized assembly



arithmetic operation: fused multiply-add

register order: multiply e.g. xmm1 and xmm3, add xmm2 to result, put final result in xmm1

`vfmadd132ps`

AVX vector instruction

packed variant: works on multiple data

data type: single-precision floating point

`vmovaps`

move operation: load or store

data is aligned

What could go wrong with automatic vectorization?

- compiler-based auto-vectorization (e.g. GCC's `-ftree-vectorize`)
  - requires analysis and heuristics
  - has lots of points of failure
    - loop-carried dependencies
    - pointer aliasing
    - memory alignment
    - data type mixing
    - too complex control flow / code in general
    - numerical stability issues

```
void foo(double* a, double* b) {
  for(int i=0; i<32; ++i) {
    a[i] = b[i] * 2;
  }
}
```

OpenMP `simd` directive

- portable vectorization without compiler- or hardware-specific intrinsics
  - no need to know about GCC/LLVM/Intel/ARM…
- not the be-all end-all solution to vectorization but can help a lot
- can be combined with `for` directive
  - distributes vectorized loop iteration chunks among threads

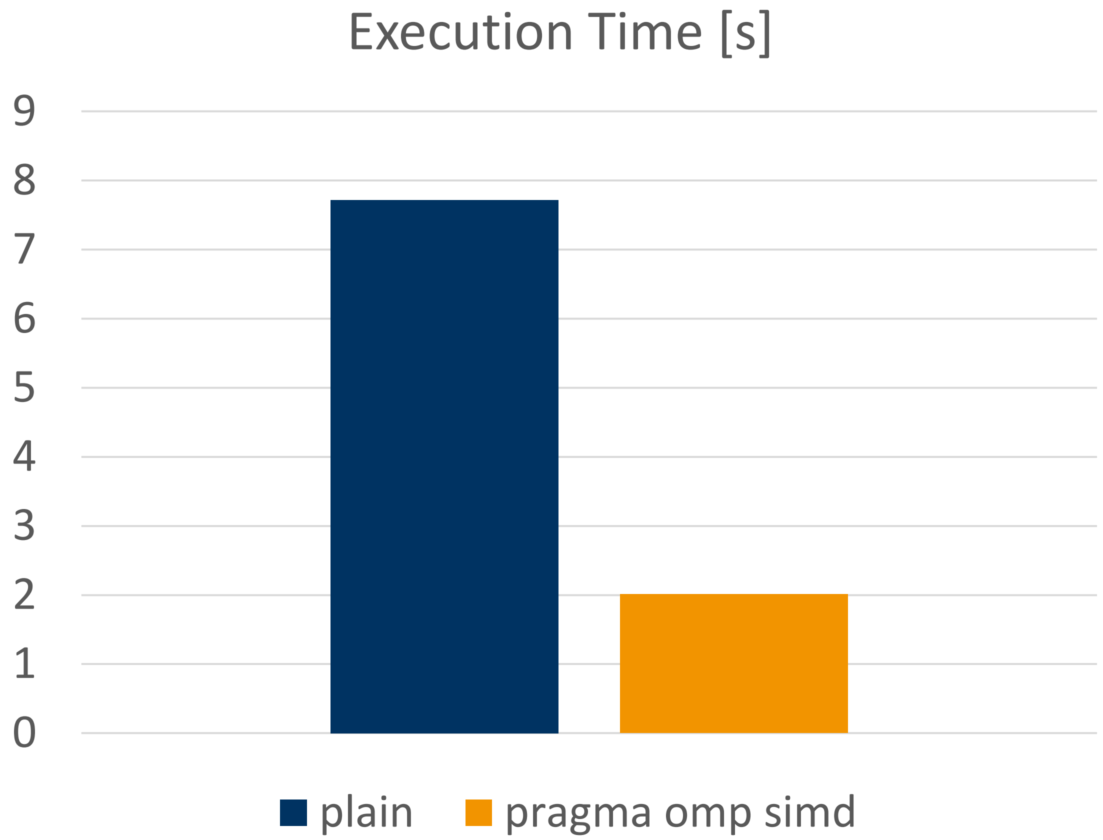
```
// note: aligned_alloc requires -std=c11
int* a = aligned_alloc(32,
                       sizeof(int)*SIZE);
int* b = aligned_alloc(32,
                       sizeof(int)*SIZE);

// initialize a, b, and f...

#pragma omp simd aligned(a,b:32)
for(int i = 0; i < SIZE; ++i) {
  b[i] += a[i] * f;
}
```

Vectorization performance comparison

- LCC2, gcc/8.2.0, single-threaded,
  - `-march=native`
    `-mtune=native`
    `-O2`

- 10^8 vector sums on integer arrays of length 64
  - code example of previous slide

- execution time reduced by 3.84x
  - 4 integers per operation + some overhead



Vectorization hazards

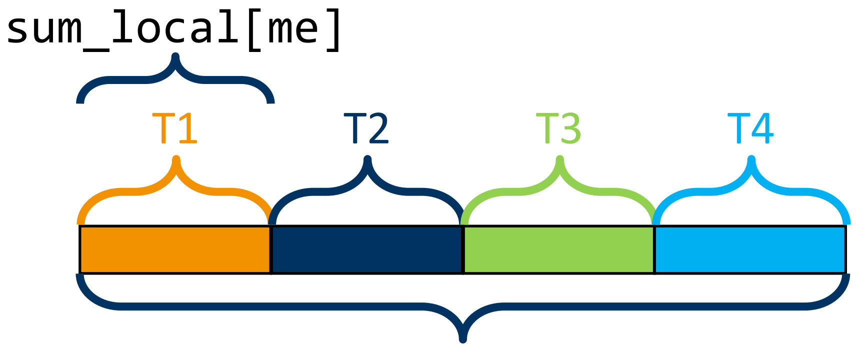
- ❌ Need data types and arithmetic operations suitable for vectorization
  - might require changing your implementation
- ❌ Automatic only: Program analysis must deem vectorization to be safe
  - can override manually using e.g. OpenMP or intrinsics
- 🐌 Need enough instructions in stream to mitigate any static overheads
  - okay if run repetitively in a loop
- 🐌 Data should be properly aligned (though diminishing impact with each CPU generation update)
  - okay if using aligned allocations and hinting alignment to compiler
- 🐌 Code should not be memory-bound
  - okay if using loop blocking/tiling to fit data chunks in L1 cache
- 🐌 Code should not be branch-bound
  - okay if using loop unrolling (often done automatically by compiler anyway)

- Any of the above can cause vectorization to fail (not vectorized (❌) or low performance (🐌))

Common Node-Level (esp. OpenMP) Pitfalls

False sharing

- common performance pitfall in shared-memory programming
  - cache coherence tries to keep all data up-to-date and valid for all threads
  - can unnecessary coherence traffic and cache misses for multi-threaded programs



sum_local[me]

T1 T2 T3 T4

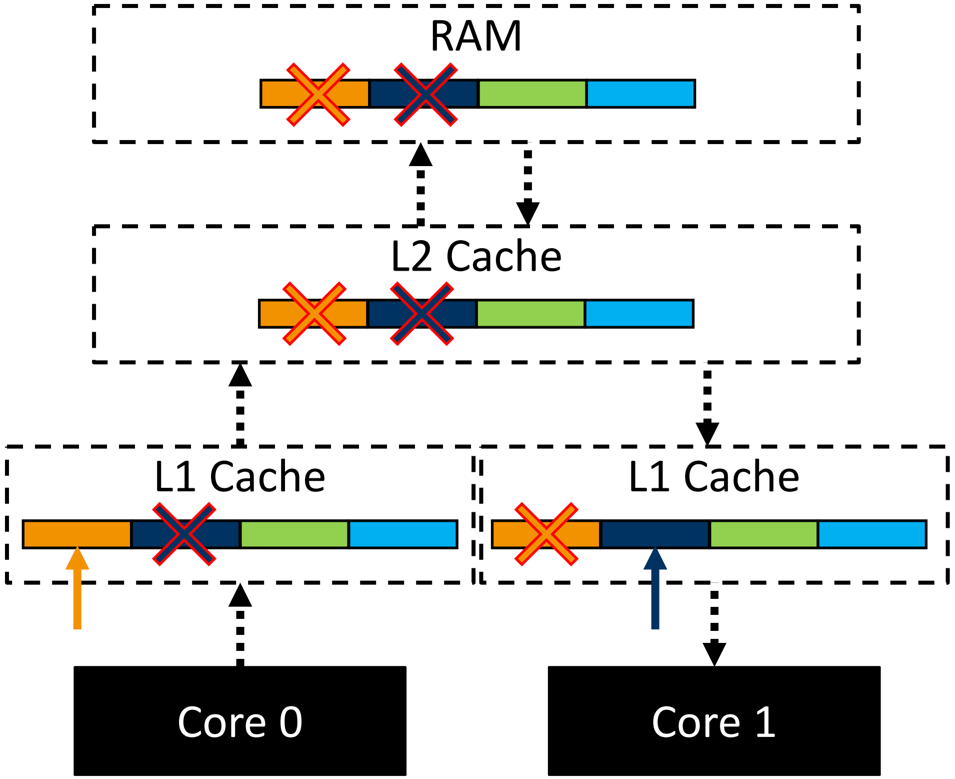
cache line (64 bytes)

```
double sum = 0.0;
double sum_local[MAX_NUM_THREADS];

#pragma omp parallel
{
  int me = omp_get_thread_num();
  sum_local[me] = 0.0;

  #pragma omp for
  for (int i = 0; i < N; i++)
    sum_local[me] += x[i] * y[i];

  #pragma omp atomic
  sum += sum_local[me];
}
```

False sharing cont'd

- thread 1
  - reads first 8 bytes
    - causes entire cache line to be fetched
  - writes first 8 bytes
    - entire cache line invalidated for thread 2

- thread 2
  - reads second 8 bytes
    - causes entire cache line to be fetched
  - writes second 8 bytes
    - entire cache line invalidated for thread 1



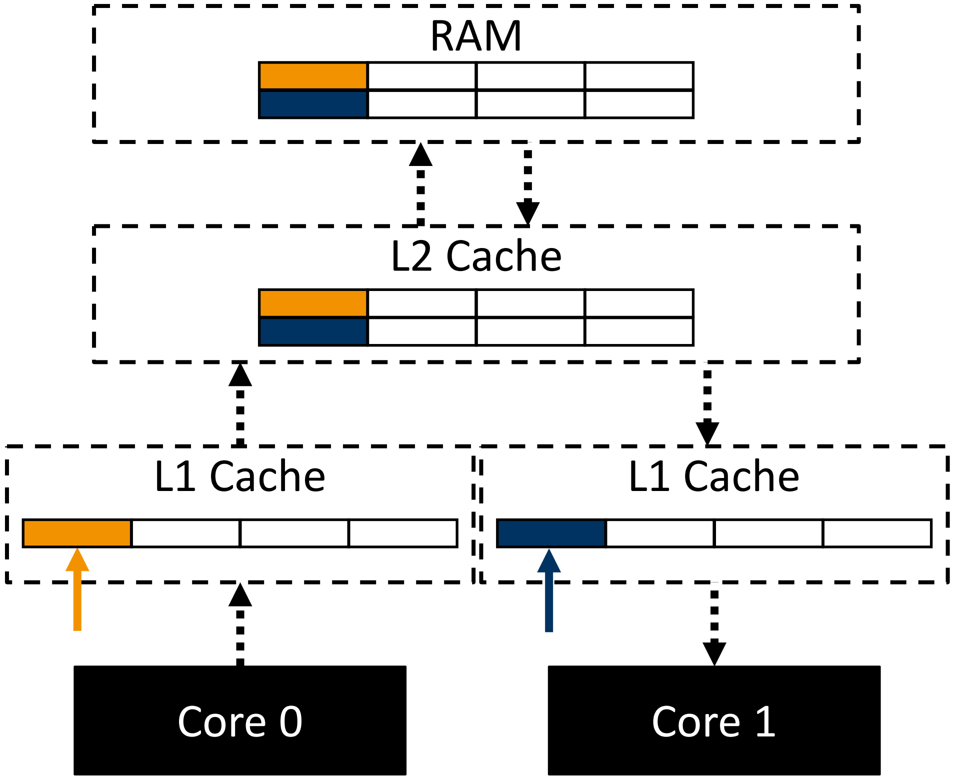
Possible solution to false sharing: padding

```
double sum = 0.0;
double sum_local[MAX_NUM_THREADS];

#pragma omp parallel
{
  int me = omp_get_thread_num();
  sum_local[me] = 0.0;

  #pragma omp for
  for (int i = 0; i < N; i++)
    sum_local[me] += x[i] * y[i];

  #pragma omp atomic
  sum += sum_local[me];
}
```

```
double sum = 0.0;
double sum_local[MAX_NUM_THREADS][8];

#pragma omp parallel
{
  int me = omp_get_thread_num();
  sum_local[me][0] = 0.0;

  #pragma omp for
  for (int i = 0; i < N; i++)
    sum_local[me][0] += x[i] * y[i];

  #pragma omp atomic
  sum += sum_local[me][0];
}
```

Possible solution to false sharing: padding cont’d

- thread 1
  - reads first 8 bytes of first cache line
    - causes first cache line to be fetched
  - writes first 8 bytes
- thread 2
  - reads first 8 bytes of second cache line
    - causes second cache line to be fetched
  - writes second 8 bytes
- drawback: memory footprint increase

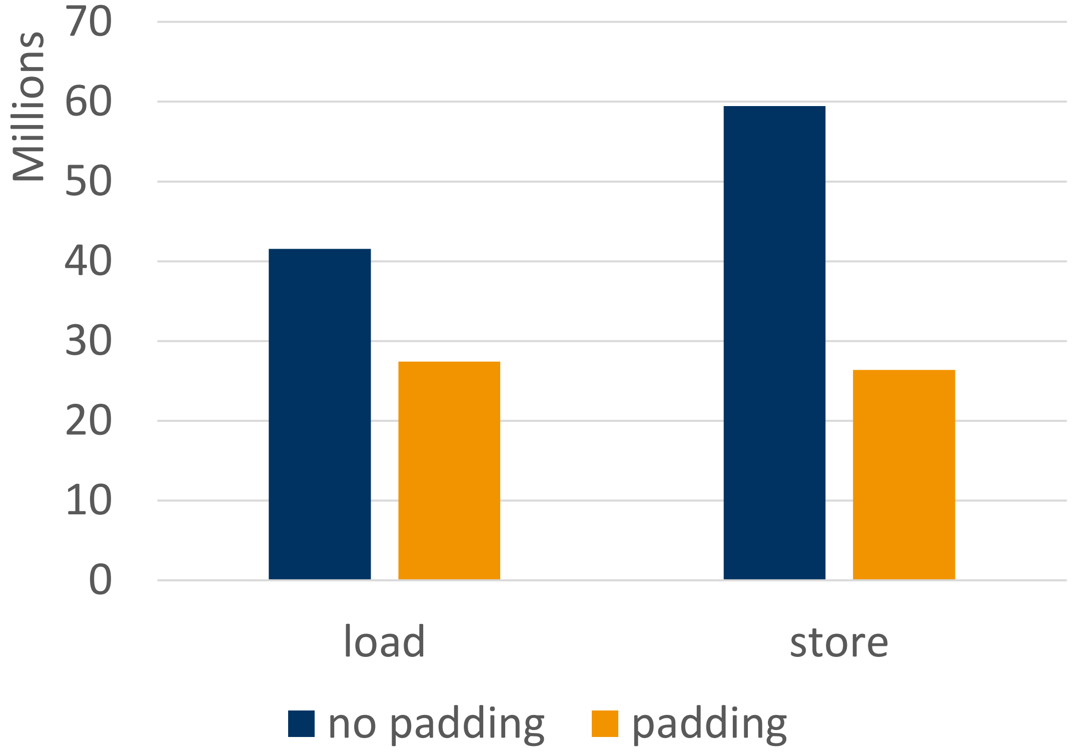


How to determine padding size

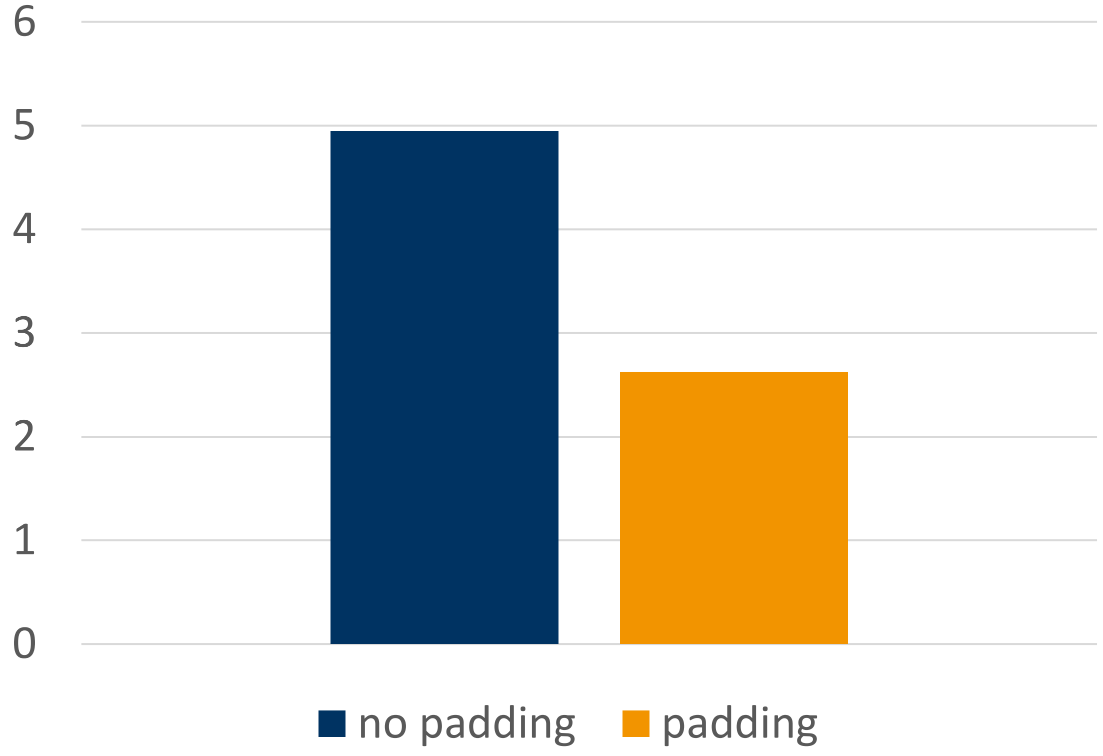
- Read documentation to check for your specific CPU
  - most x86 and ARM systems use 64 byte cache lines
  - IBM POWER: 128 bytes, IBM z: 256 bytes
  - do not hard-code this, bad practice

- Better yet, ask tools, e.g.
  - `std::hardware_destructive_interference_size`: C++17 constant for L1 cache line size (constructive variant also available)
  - `cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size`: ask OS kernel for CPU0 and cache level 1 (index: level & instruction/data cache identifier)
  - `papi_mem_info`: executable, part of PAPI library, gives cache hierarchy information

False sharing performance comparison (LCC2, 10^9 iterations)

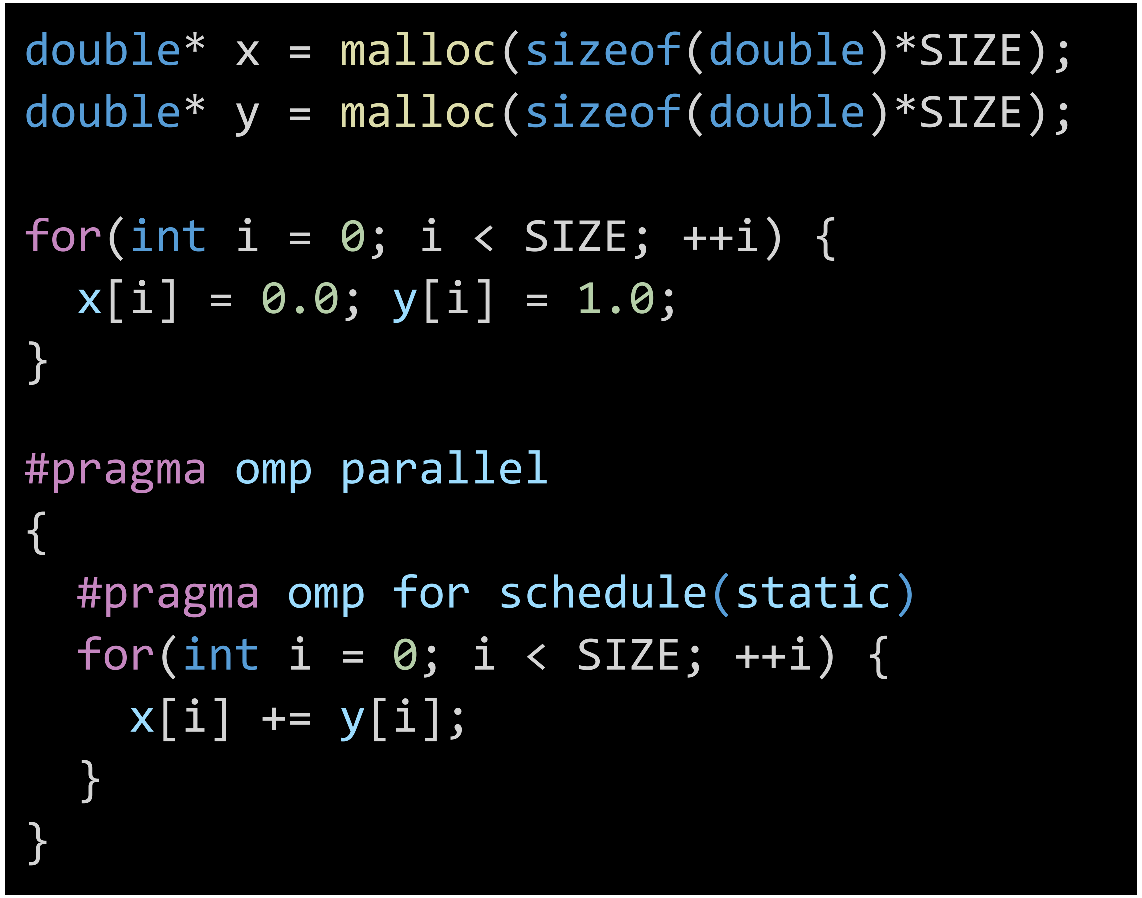


Last-level Cache Misses

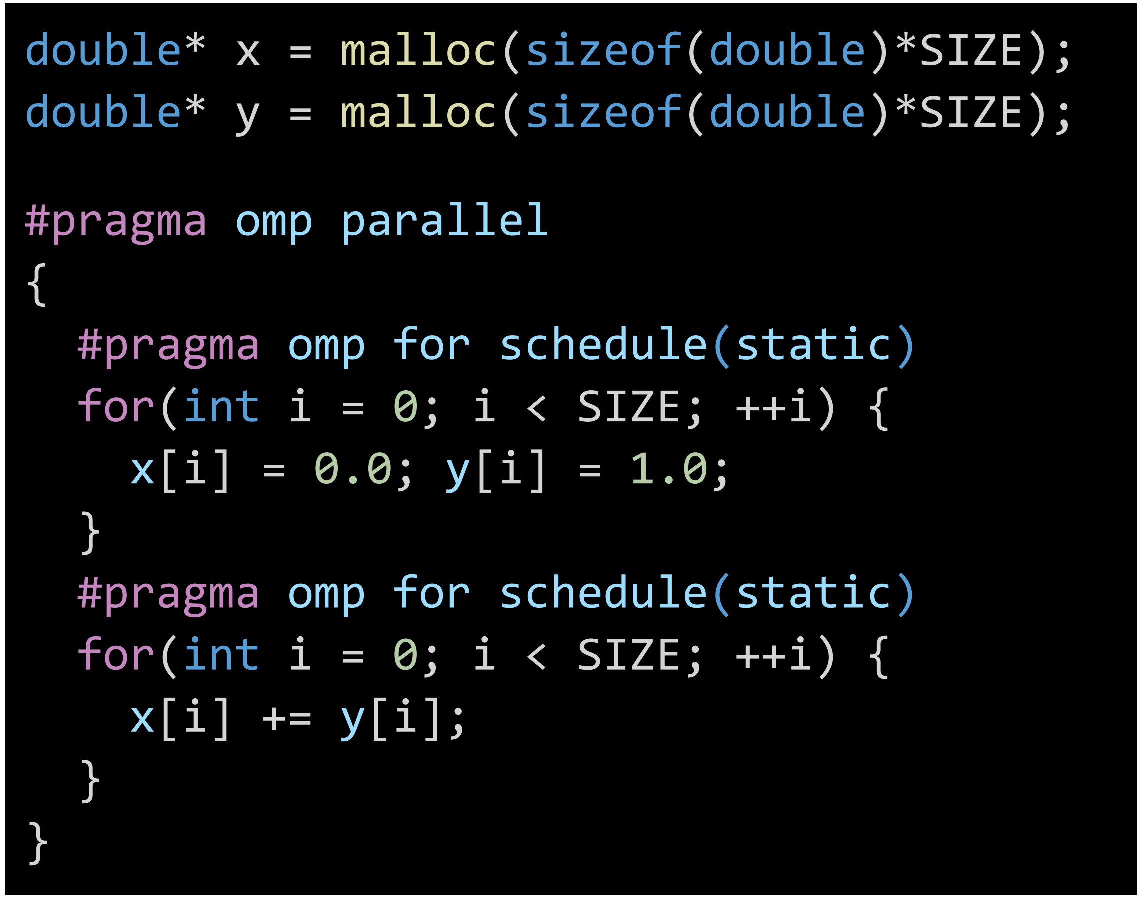


Wall Time [s]

First touch & NUMA – how to initialize your data?



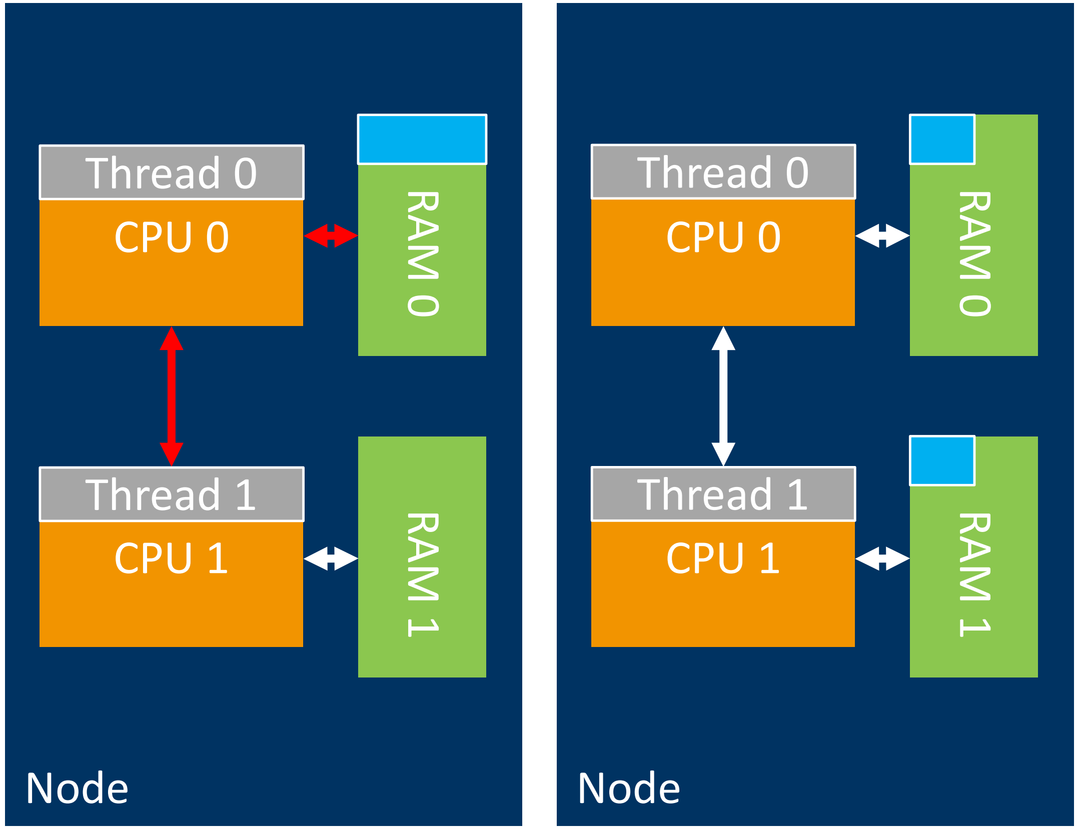
```c
double* x = malloc(sizeof(double)*SIZE);
double* y = malloc(sizeof(double)*SIZE);

for(int i = 0; i < SIZE; ++i) {
  x[i] = 0.0; y[i] = 1.0;
}

#pragma omp parallel
{
  #pragma omp for schedule(static)
  for(int i = 0; i < SIZE; ++i) {
    x[i] += y[i];
  }
}
```



```c
double* x = malloc(sizeof(double)*SIZE);
double* y = malloc(sizeof(double)*SIZE);

#pragma omp parallel
{
  #pragma omp for schedule(static)
  for(int i = 0; i < SIZE; ++i) {
    x[i] = 0.0; y[i] = 1.0;
  }
  #pragma omp for schedule(static)
  for(int i = 0; i < SIZE; ++i) {
    x[i] += y[i];
  }
}
```

Sequential vs. parallel initialization on NUMA

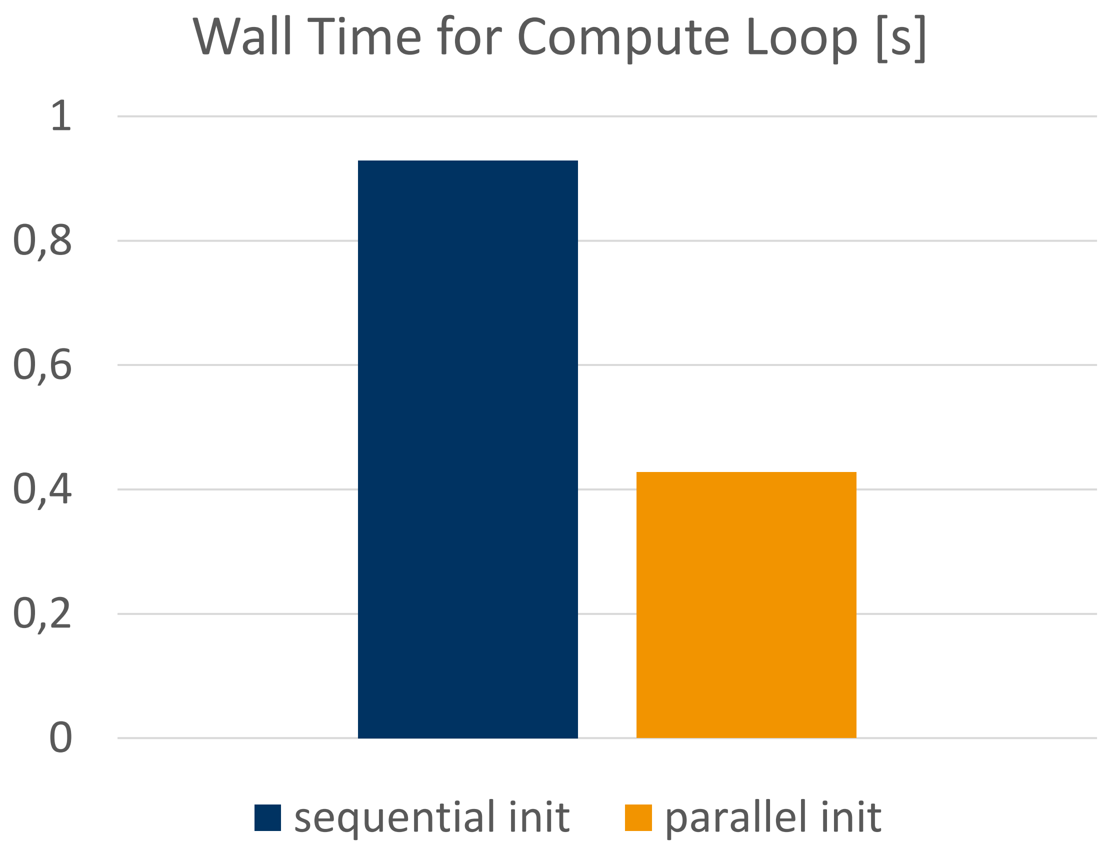
- data is not allocated upon allocation but upon first access (“*first touch*”)
  - happens when you initialize data in the RAM module of the initializing thread
- sequential initialization
  - all data resides with RAM modules of the core of the initializing thread
  - causes bottleneck on single memory bus, additional inter-CPU traffic and higher latency for core 1
- parallel initialization
  - data resides with RAM of the threads initializing the respective chunk of data
  - only downside: need to have same domain decomposition & parallelization in initialization and computation



Performance impact of first touch and NUMA

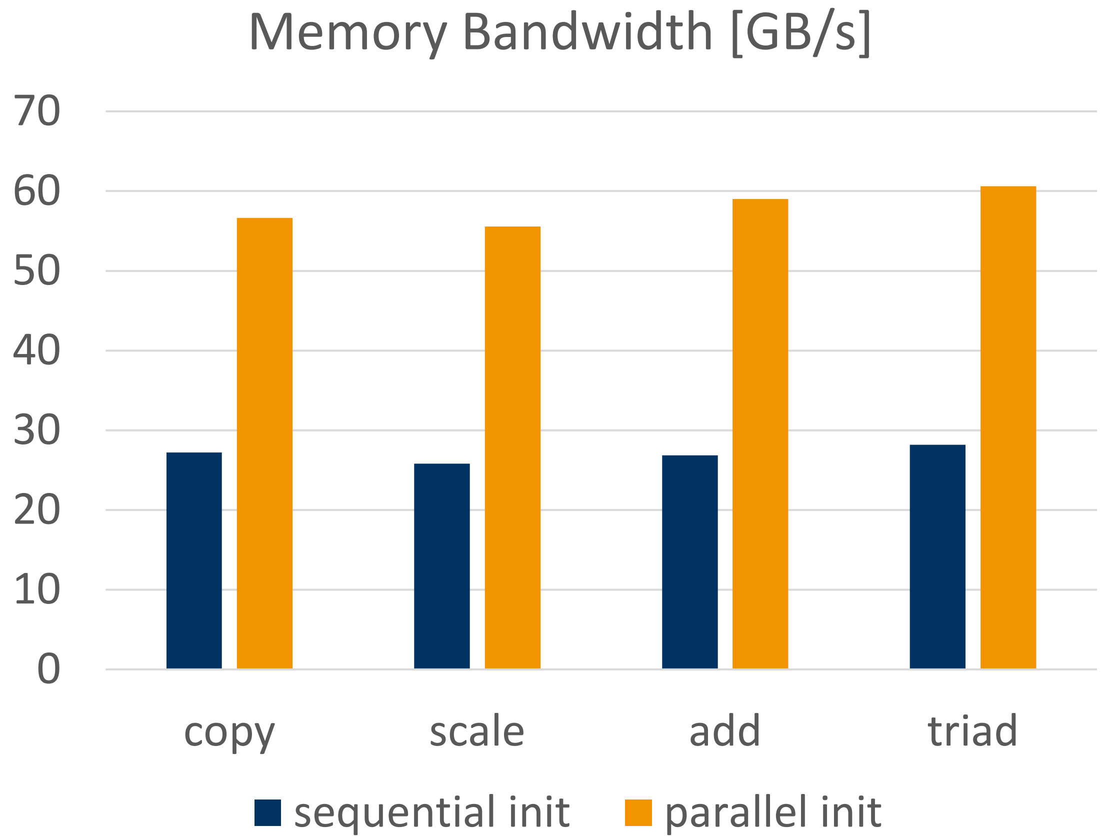
- hudson server (2x Intel Xeon E5-2699 v3 18-core), gcc 6.3.0, 10^8 double elements, 10 repetitions

- performance improvement of compute loop (not initialization!) of 2.17x



Performance impact of first touch and NUMA cont’d

- same platform, stream memory benchmark, 3 threads per CPU
  - https://www.cs.virginia.edu/stream/
- between 2.08x and 2.20x higher bandwidth
- impact can vary a lot and depends also on your hardware platform



Summary

- Alder Lake, hardware architecture characteristics, implications
- Vectorization
- common shared-memory programming pitfalls

Image Sources

- Intel Architecture Day Slide: https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf
- Alder Lake Die Shots: https://www.reddit.com/r/intel/comments/qhbbow/10nm_esf_intel_7_alder_lake_die_shot/
- Raspberry Pi 4: https://geizhals.eu/raspberry-pi-4-modell-b-a2081127.html
- Samsung Gear S2 3G: https://geizhals.eu/samsung-gear-s2-3g-black-a1318676.html
- Apple M1: https://www.computerbase.de/2020-11/apple-m1-analyse/
- Fujitsu A64FX: https://www.hpcwire.com/2020/02/03/fujitsu-arm64fx-supercomputer-to-be-deployed-at-nagoya-university/
- Snail: https://live.staticflickr.com/3620/3391918403_188330c938_b.jpg