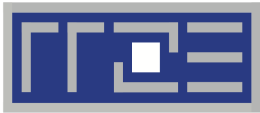


Single Instruction Multiple Data (SIMD) processing

High Performance Computing

SIMD terminology

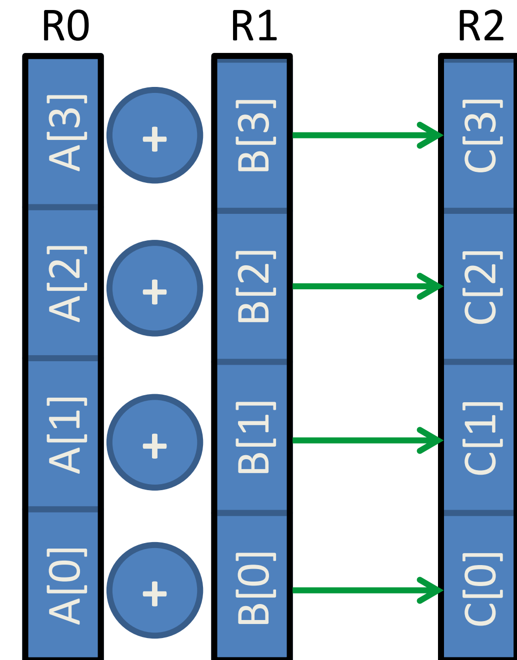
FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

A word on terminology

- SIMD == "one instruction → several operations"
- "SIMD width" == number of operands that fit into a register
- No statement about parallelism among those operations
- Original vector computers: long registers, pipelined execution, but no parallelism (within the instruction)

Today

- x86: most SIMD instructions fully parallel
  "Short Vector SIMD"
  Some exceptions on some archs (e.g., vdivpd)
- NEC Tsubasa: 32-way parallelism but SIMD width = 256 (DP)

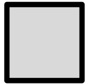


Scalar execution units


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

Register widths

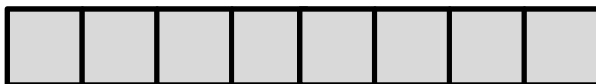
- 1 operand



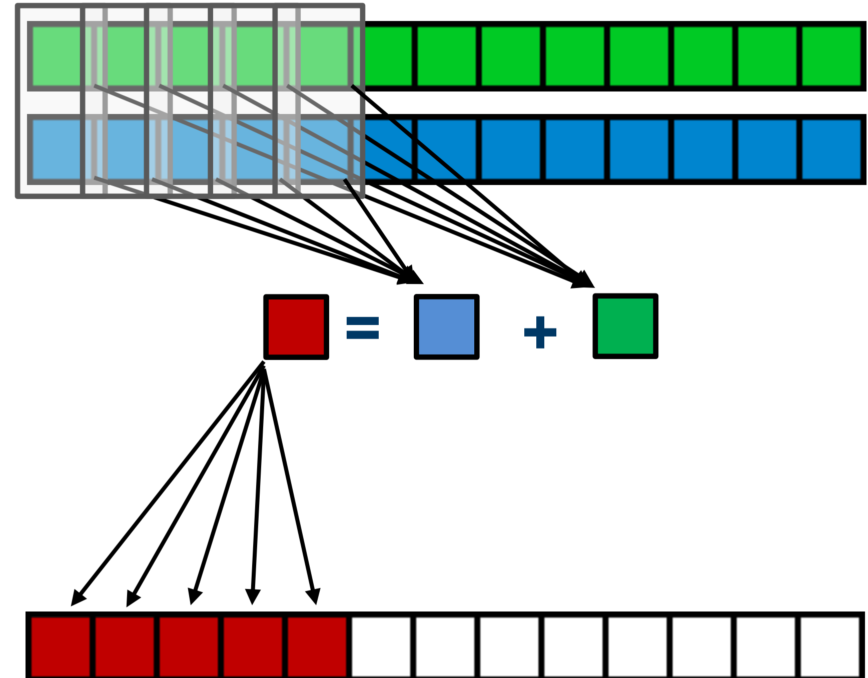
- 2 operands (SSE)

- 4 operands (AVX)

- 8 operands (AVX512)



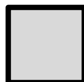
Scalar execution



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

Register widths

- 1 operand

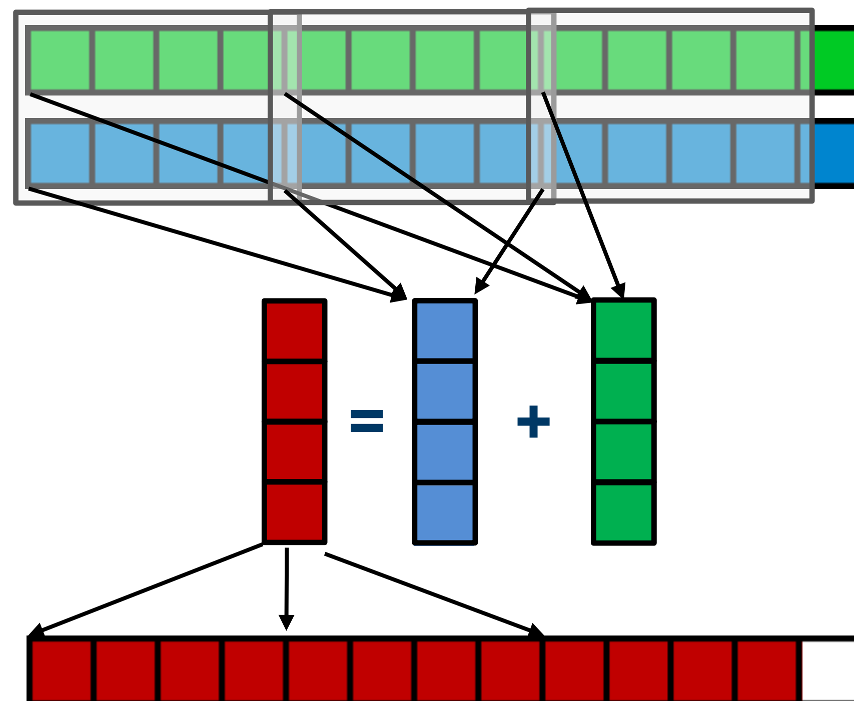


- 2 operands (SSE)

- 4 operands (AVX)

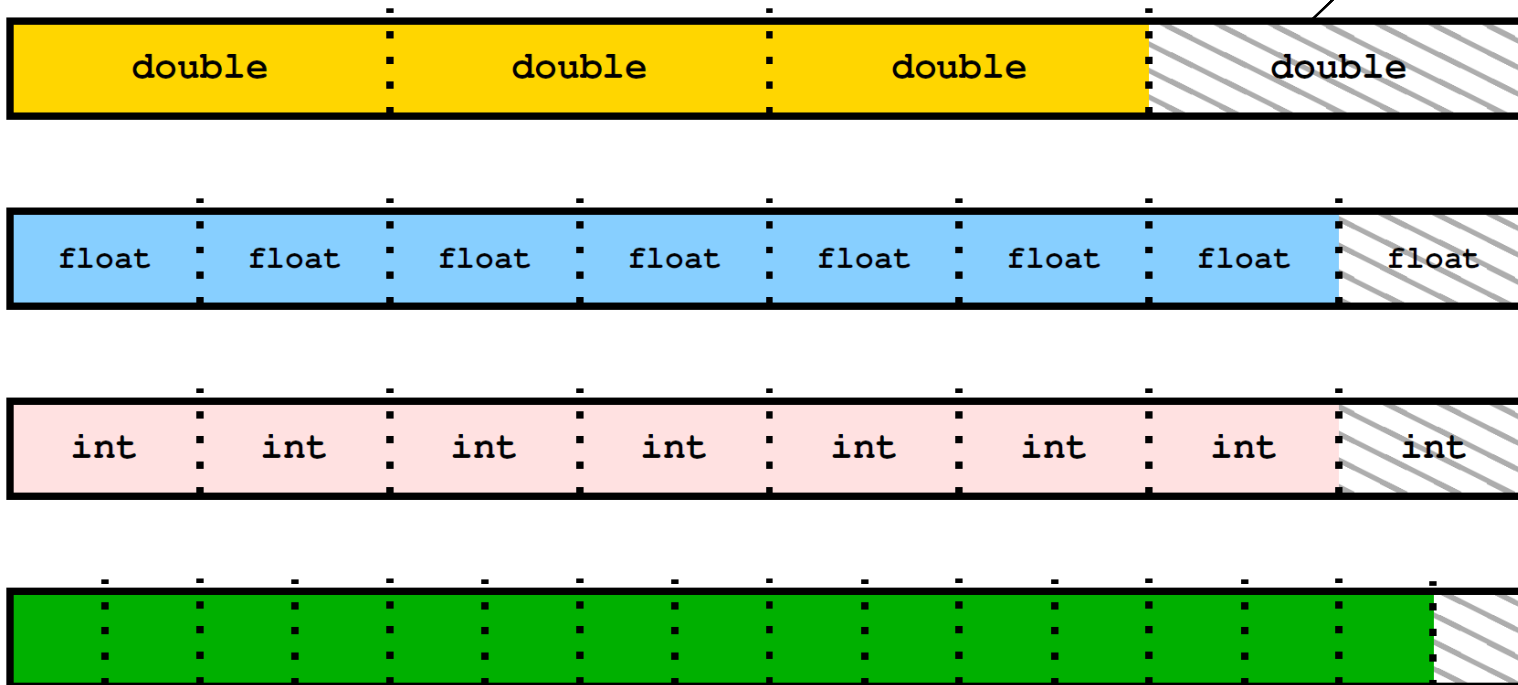
- 8 operands (AVX512)

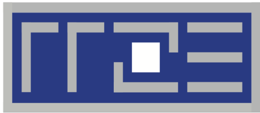
SIMD execution



Data types in 32-byte SIMD registers

- **Supported data types depend on actual SIMD instruction set**





SIMD

The Basics

High Performance Computing

SIMD processing – Basics

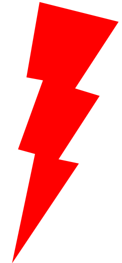
FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Steps (done by the compiler) for "SIMD processing"

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

"Loop unrolling"

This should not be done by hand!



```
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
```

```
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
```

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

Add the corresponding 64 Bit entries in `R0` and `R1` and store the 4 results to `R2` → V64ADD[R0,R1] → R2

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

SIMD processing: roadblocks

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

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 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) {…}
```

How to leverage SIMD: your options

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Options:

- The compiler does it for you
  (but: aliasing, alignment, language, abstractions)
- Compiler directives (pragmas)
- Alternative programming models for compute kernels (OpenCL, ispc)
- Intrinsics (restricted to C/C++)
- Implement directly in assembler

To use intrinsics the following headers are available:

- **`xmmintrin.h` (SSE)**
- **`pmmintrin.h` (SSE2)**
- **`immintrin.h` (AVX)**
- **`x86intrin.h` (all extensions)**

```
for (int j=0; j<size; j+=16){
    t0 = _mm_loadu_ps(data+j);
    t1 = _mm_loadu_ps(data+j+4);
    t2 = _mm_loadu_ps(data+j+8);
    t3 = _mm_loadu_ps(data+j+12);
    sum0 = _mm_add_ps(sum0, t0);
    sum1 = _mm_add_ps(sum1, t1);
    sum2 = _mm_add_ps(sum2, t2);
    sum3 = _mm_add_ps(sum3, t3);
}
```

Vectorization compiler options (Intel)

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

- The compiler will vectorize starting with `–O2`.
- To enable specific SIMD extensions use the –x option:
  - `-xSSE2` vectorize for SSE2 capable machines

  Available SIMD extensions:
  `SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, ...`

  - `-xAVX` on Sandy/Ivy Bridge processors
  - `-xCORE-AVX2` on Haswell/Broadwell
  - `-xCORE-AVX512` on Skylake (certain models)
  - `-xMIC-AVX512` on Xeon Phi Knights Landing

  Recommended option:
  - `-xHost` will optimize for the architecture you compile on
    (Caveat: do not use on standalone KNL, use MIC-AVX512)
  - To really enable 512-bit SIMD with current Intel compilers you need to set:
    `-qopt-zmm-usage=high`

User-mandated vectorization (OpenMP 4)

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

- Since OpenMP 4.0 SIMD features are a part of the OpenMP standard
- `#pragma omp simd` enforces vectorization
- Essentially a standardized “go ahead, no dependencies here!”
  - **Do not lie** to the compiler here!
- Prerequesites:
  - Countable loop
  - Innermost loop
  - Must conform to for-loop style of OpenMP worksharing constructs
- There are additional clauses:
  `reduction, vectorlength, private, collapse, ...`

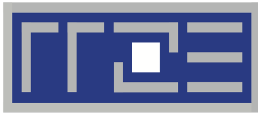
```
for (int j=0; j<n; j++) {
  #pragma omp simd reduction(+:b[j:1])
  for (int i=0; i<n; i++) {
     b[j] += a[j][i];
  }
}
```

X86 SIMD and Alignment

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

- Alignment issues
  - Alignment of arrays should optimally be on SIMD-width address boundaries to allow packed aligned loads (and NT stores on x86)
  - Otherwise the compiler will revert to unaligned loads/stores
  - Modern x86 CPUs have less (not zero) impact for misaligned LOAD/STORE, but Xeon Phi KNC relies heavily on it!
  - How is manual alignment accomplished?

- Stack variables: **`alignas`** keyword (C++11/C11)
- Dynamic allocation of aligned memory (**`align`** = alignment boundary)
  - C before C11 and C++ before C++17:
    `posix_memalign(void **ptr, size_t align, size_t size);`
  - C11 and C++17:
    `aligned_alloc(size_t align, size_t size);`



SIMD

Reading Assembly Language

(Don’t Panic)

High Performance Computing

Assembler: Why and how?

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Why check the assembly code?

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

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

1. **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

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

- 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 15 (?) 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`)

Intel syntax

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

AT&T syntax

```
movaps     %xmm3, 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         addq   $0x8,%rax
401ba8: 78 a6               js     401b50 <triad_asm+0x4b>
```

Basics of the x86-64 ISA with extensions

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

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)	

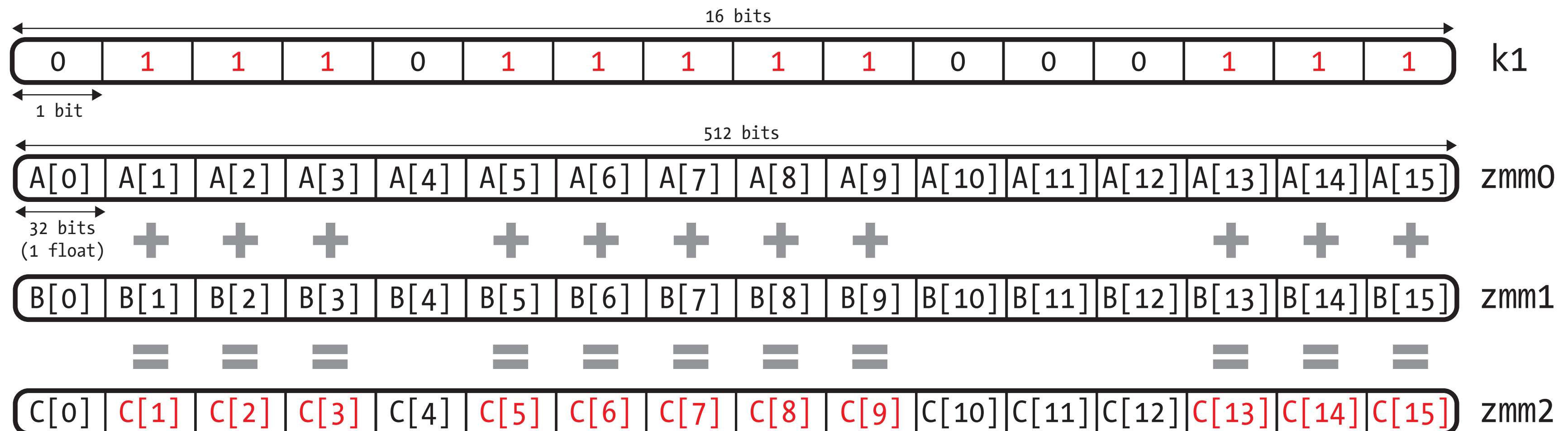
SIMD instructions are distinguished by:

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`**)

Example for masked execution

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

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



ISA support on Intel chips

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

The current **Skylake** architecture supports all **legacy** ISA extensions:

`MMX, SSE, AVX, AVX2`

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

- AVX-512 Foundation (F)
- AVX-512 Conflict Detection Instructions (CD)
- AVX-512 Byte and Word Instructions (BW)
- AVX-512 Doubleword and Quadword Instructions (DQ)
- AVX-512 Vector Length Extensions (VL)

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

ISA Documentation:

Intel Architecture Instruction Set Extensions Programming Reference

Case Study: Sum reduction (DP)

```
double 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`

Assembly code w/ `-O1` (Intel syntax, Intel compiler):

```
.label:
        addsd  xmm0,[rdi + rax * 8]
        inc    rax
        cmp    rax, rsi
        jl     .label
```

AT&T syntax:
`addsd 0(%rdi,%rax,8),%xmm0`

Sum reduction (DP) – AVX512

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Assembly code w/ `-O3 -xCORE-AVX512 -qopt-zmm-usage=high` :

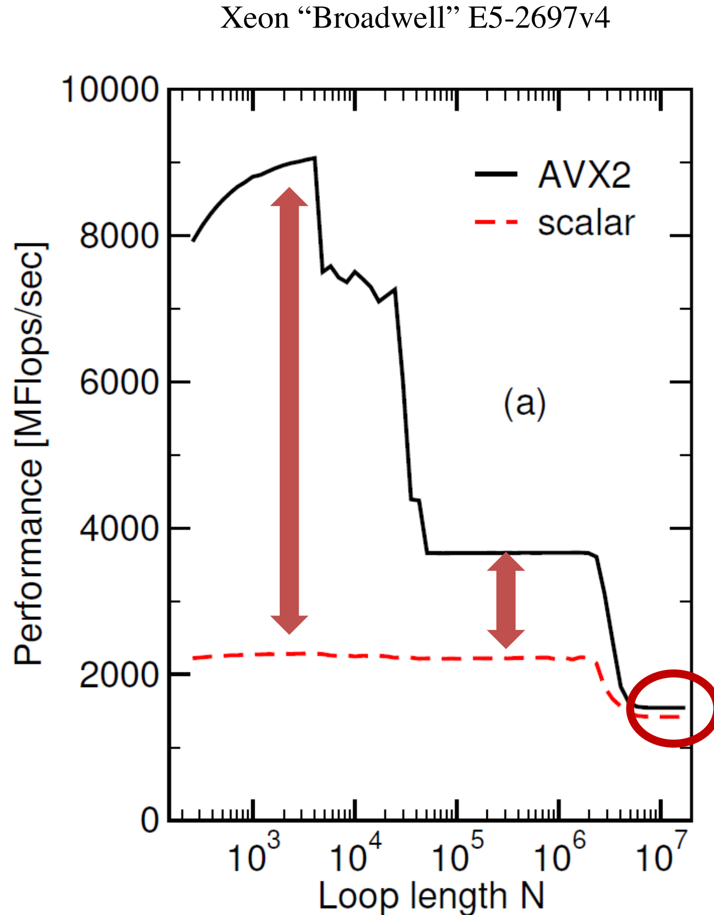
```
.label:
        vaddpd     zmm1, zmm1, [rdi+rcx*8]
        vaddpd     zmm4, zmm4, [64+rdi+rcx*8]
        vaddpd     zmm3, zmm3, [128+rdi+rcx*8]
        vaddpd     zmm2, zmm2, [192+rdi+rcx*8]
        add        rcx, 32
        cmp        rcx, rdx
        jb         .label
;
        vaddpd     zmm1, zmm1, zmm4
        vaddpd     zmm2, zmm3, zmm2
        vaddpd     zmm1, zmm1, zmm2
; [… SNIP …]
        vshuff32x4 zmm2, zmm1, zmm1, 238
        vaddpd     zmm1, zmm2, zmm1
        vpermpd    zmm3, zmm1, 78
        vaddpd     zmm4, zmm1, zmm3
        vpermpd    zmm5, zmm4, 177
        vaddpd     zmm6, zmm4, zmm5
        vaddsd     xmm0, xmm6, xmm0
```

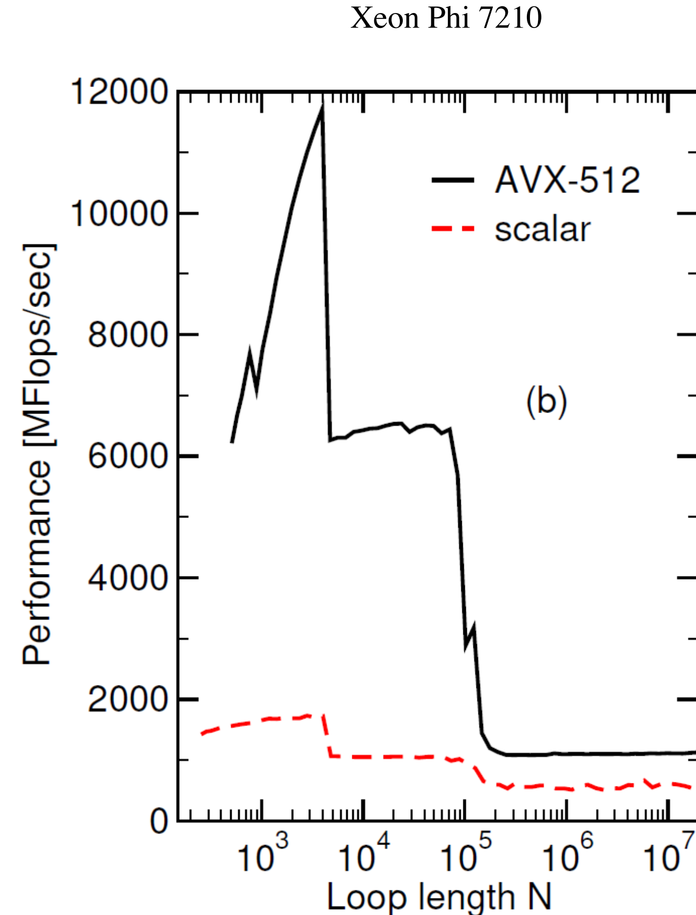
Remainder omitted (points to "; [… SNIP …]")

Bulk loop code (8x4-way unrolled) — annotates the loop from `.label:` to `jb .label`

Sum up 32 partial sums into **`xmm0.0`** — annotates the code after the loop

Sum reduction (DP) – seq. performance





SIMD is an in-core performance feature! If the bottleneck is data transfer, its benefit is limited.

SIMD with masking

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

```
double sum = 0.0;

for (int i=0; i<size; i++){
    if(data[i]>0.0)
      sum += data[i];
}
```

Bulk loop code (8x4-way unrolled)

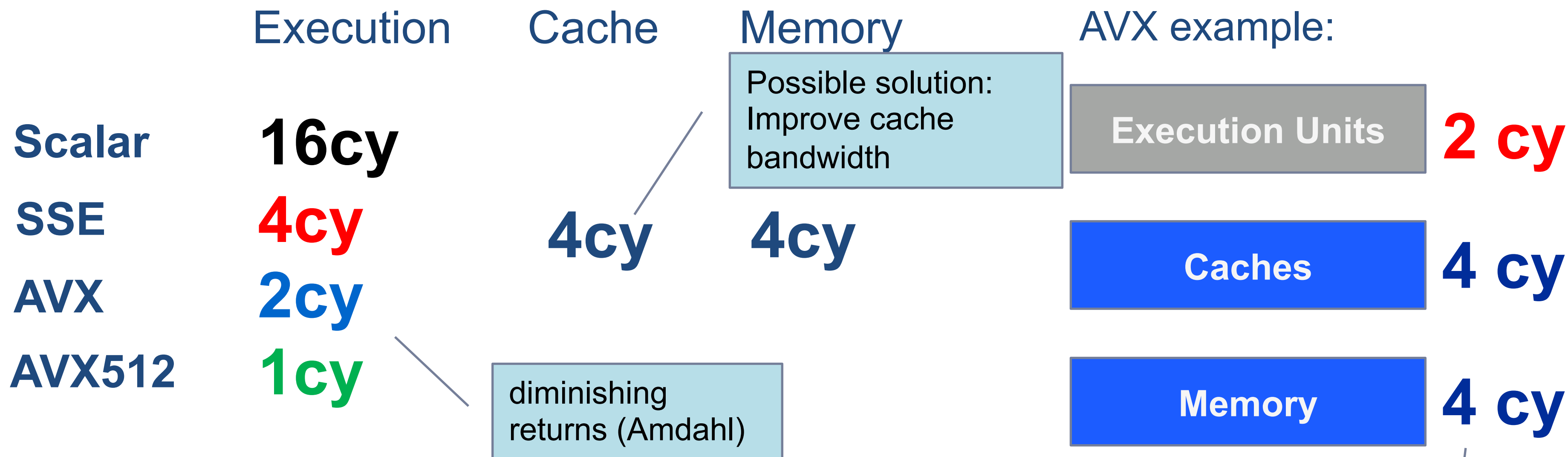
```
.label:
        vmovups    (%r12,%rsi,8), %zmm5
        vmovups    64(%r12,%rsi,8), %zmm6
        vmovups    128(%r12,%rsi,8), %zmm7
        vmovups    192(%r12,%rsi,8), %zmm8
        vcmpgtpd         , %zmm5, %k1
        vcmpgtpd         , %zmm6, %k2
        vcmpgtpd         , %zmm7, %k3
        vcmpgtpd         , %zmm8, %k4
        vaddpd     %zmm5, %zmm0, %zmm0{%k1}
        vaddpd     %zmm6, %zmm3, %zmm3{%k2}
        vaddpd     %zmm7, %zmm2, %zmm2{%k3}
        vaddpd     %zmm8, %zmm1, %zmm1{%k4}
        addq       $32, %rsi
        cmpq       %rdx, %rsi
        jb         .label
```

SIMD mask generation

masked SIMD ADDs (accumulates)

Limits of SIMD processing

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



Total runtime with data loaded from memory:

Scalar 24 SSE 12 AVX 10 AVX512 9

Per-cacheline (8 iterations) cycle counts

Rules and guidelines for vectorizable loops

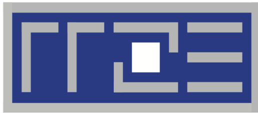
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

1. Inner loop
2. Countable (loop length can be determined at loop entry)
3. Single entry and single exit
4. Straight line code (no conditionals) – unless masks can be used
5. No (unresolvable) read-after-write data dependencies
6. No function calls (exception intrinsic math functions)

Better performance with:

1. Simple inner loops with unit stride (contiguous data access)
2. Minimize indirect addressing
3. Align data structures to SIMD width boundary (minor impact)

In C use the **`restrict`** keyword and/or **`const`** qualifiers and/or compiler options to rule out array/pointer aliasing



Efficient parallel programming on ccNUMA nodes

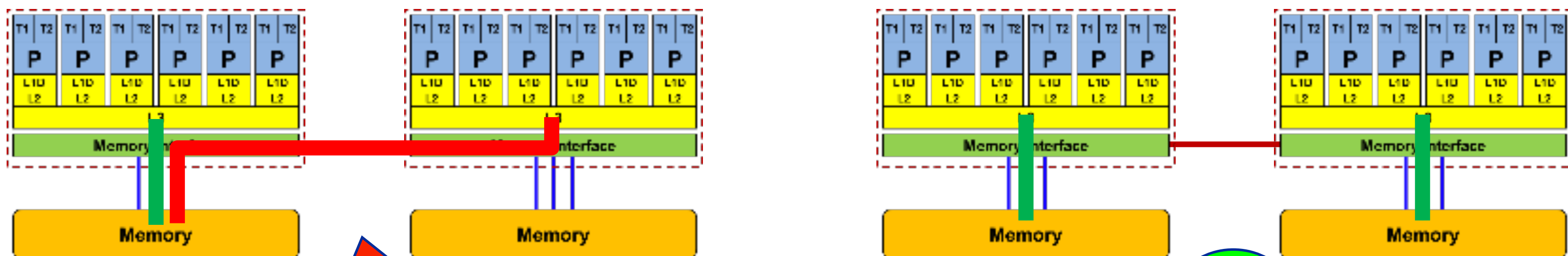
Performance characteristics of ccNUMA nodes

First touch placement policy

High Performance Computing

ccNUMA – The other affinity to care about

- ccNUMA:
  - Whole memory is transparently accessible by all processors
  - but physically distributed across multiple locality domains (LDs)
  - 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?

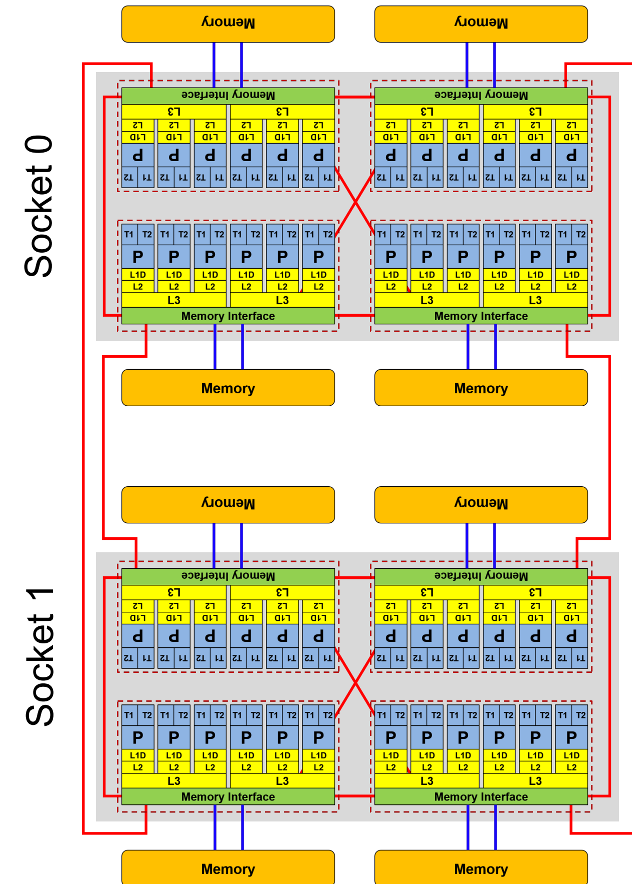


Note: Page placement is implemented in units of OS pages (often 4kB, possibly more)

How much does nonlocal access cost?

Example: AMD “Epyc” 2-socket system
(8 chips, 2 sockets, 48 cores):
STREAM Triad bandwidth measurements [Gbyte/s]

MEM node \ CPU node	0	1	2	3	4	5	6	7
0	32.4	21.4	21.8	21.9	10.6	10.6	10.7	10.8
1	21.5	32.4	21.9	21.9	10.6	10.5	10.7	10.6
2	21.8	21.9	32.4	21.5	10.6	10.6	10.8	10.7
3	21.9	21.9	21.5	32.4	10.6	10.6	10.6	10.7
4	10.6	10.7	10.6	10.6	32.4	21.4	21.9	21.9
5	10.6	10.6	10.6	10.6	21.4	32.4	21.9	21.9
6	10.6	10.7	10.6	10.6	21.9	21.9	32.3	21.4
7	10.7	10.6	10.6	10.6	21.9	21.9	21.4	32.5



ccNUMA default memory locality

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

- **"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
  - This might be a problem, see later
- Caveat: “to touch" means “to write", not “to allocate"
- Example:

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

for(i=0; i<N; i++) // or i+=PAGE_SIZE/sizeof(double)
    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

Coding for ccNUMA data locality

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Most simple case: explicit initialization

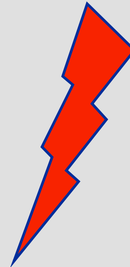
```
integer,parameter :: N=10000000
double precision A(N), B(N)



A=0.d0



!$OMP parallel do
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end parallel do
```



```
integer,parameter :: N=10000000
double precision A(N),B(N)
!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i)=0.d0
end do
!$OMP end do
...
!$OMP do schedule(static)
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

Coding for ccNUMA data locality

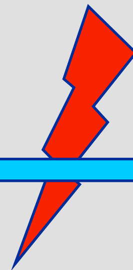
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so "localize" arrays before I/O

```
integer,parameter :: N=10000000
double precision A(N), B(N)

READ(1000) A

!$OMP parallel do
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end parallel do
```



```
integer,parameter :: N=10000000
double precision A(N),B(N)
!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i)=0.d0
end do
!$OMP end do
!$OMP single
READ(1000) A
!$OMP end single
!$OMP do schedule(static)
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

Coding for Data Locality

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

- Required condition: OpenMP loop schedule of initialization must be the same as in all computational loops
  - Only choice: `static`! Specify explicitly on all NUMA-sensitive loops, just to be sure…
  - Imposes some constraints on possible optimizations (e.g. load balancing)
  - Presupposes that all worksharing loops with the same loop length have the same thread-chunk mapping
  - If dynamic scheduling/tasking is unavoidable, the problem cannot be solved completely if a team of threads spans more than one LD
    - Static parallel first touch is still a good idea
    - OpenMP 5.0 will have rudimentary memory affinity functionality
- How about global objects?
  - If communication vs. computation is favorable, might consider properly placed copies of global data
- C++: Arrays of objects and `std::vector<>` are by default initialized sequentially
  - STL allocators provide an elegant solution

Diagnosing bad locality

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

- If your code is cache bound, you might not notice any locality problems
- Otherwise, bad locality limits scalability (whenever a ccNUMA node boundary is crossed)
  - Just an indication, not a proof yet
- Running with **`numactl --interleave`** might give you a hint
  - See later
- Consider using performance counters
  - likwid-perfctr can be used to measure non-local memory accesses
  - Example for Intel dual-socket system (IvyBridge, 2x10-core):

```
likwid-perfctr -g NUMA –C M0:0-4@M1:0-4 ./a.out
```

Using performance counters for diagnosis

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

- Intel Ivy Bridge EP node (running 2x5 threads): measure NUMA traffic per core

```
likwid-perfctr -g NUMA –C M0:0-4@M1:0-4 ./a.out
```

- Summary output:

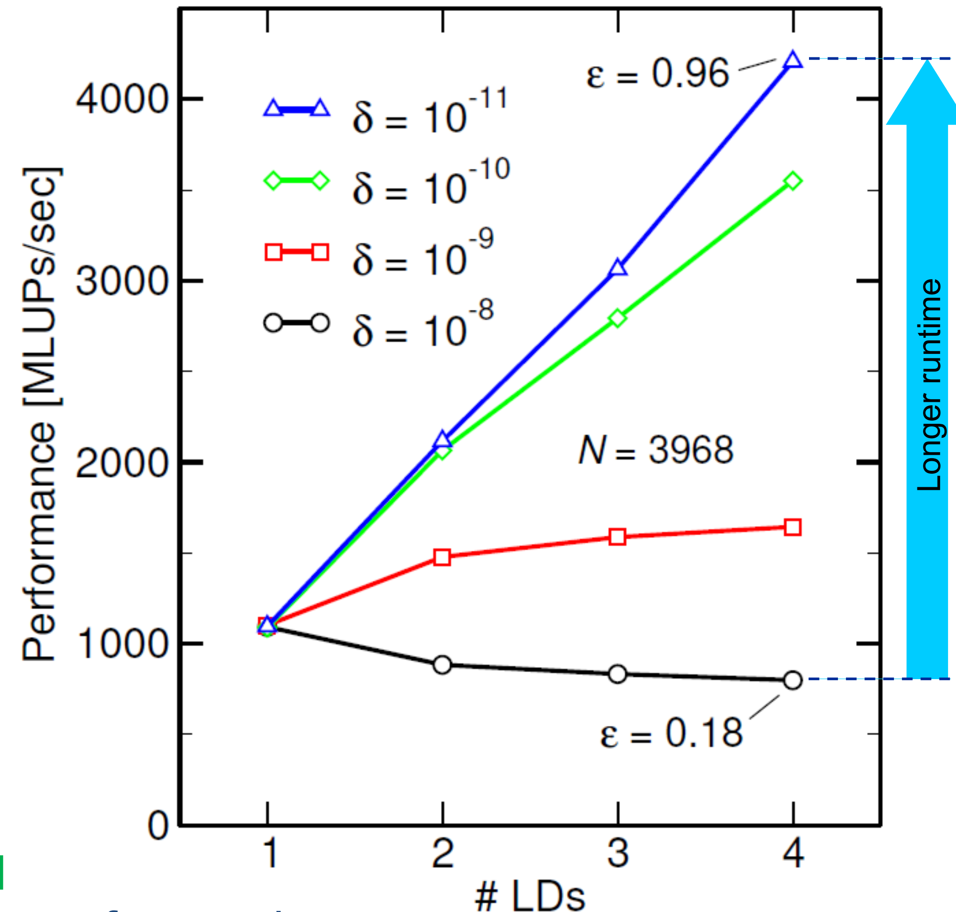
Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	4.050483	0.4050483	0.4050483	0.4050483
Runtime unhalted [s] STAT	3.03537	0.3026072	0.3043367	0.303537
Clock [MHz] STAT	32996.94	3299.692	3299.696	3299.694
CPI STAT	40.3212	3.702072	4.244213	4.03212
Local DRAM data volume [GByte] STAT	7.752933632	0.735579264	0.823551488	0.7752933632
Local DRAM bandwidth [MByte/s] STAT	19140.761	1816.028	2033.218	1914.0761
Remote DRAM data volume [GByte] STAT	9.16628352	0.86682464	0.957811776	0.916628352
Remote DRAM bandwidth [MByte/s] STAT	22630.098	2140.052	2364.685	2263.0098
Memory data volume [GByte] STAT	16.919217152	1.690376128	1.69339104	1.6919217152
Memory bandwidth [MByte/s] STAT	41770.861	4173.27	4180.714	4177.0861

About half of the overall memory traffic is caused by remote domain!

- Caveat: NUMA metrics vary strongly between CPU models

A weird observation

- Experiment: memory-bound Jacobi solver with sequential data initialization
  - No parallel data placement at all!
  - Expect no scaling across LDs
- Convergence threshold δ determines the runtime
  - The smaller δ, the longer the run
- Observation
  - No scaling across LDs for large δ (runtime 0.5 s)
  - Scaling gets better with smaller δ up to almost perfect efficiency ε (runtime 91 s)
- Conclusion
  - Something seems to "heal" the bad access locality on a time scale of tens of seconds



Riddle solved: NUMA balancing

- Linux kernel supports automatic page migration

```
$ cat /proc/sys/kernel/numa_balancing
0
$ echo 1 > /proc/sys/kernel/numa_balancing   # activate
```

- Active on all current Linux distributions
- Parameters control aggressiveness

```
$ ll /proc/sys/kernel/numa*
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_delay_ms
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_period_max_ms
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_period_min_ms
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_size_mb
```

- Default behavior is “take it slow”
- Do not rely on it! Parallel first touch is still a good idea!

Summary on ccNUMA issues

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

- **Identify the problem**
  - Is ccNUMA an issue in your code?
  - Simple test: run with `numactl --interleave`
- **Apply first-touch placement**
  - Look at initialization loops
  - Consider loop lengths and static scheduling
  - C++ and global/static objects may require special care
- **NUMA balancing is active on many Linux systems today**
  - Automatic page migration
  - Slow process, may take many seconds (configurable)
  - Not a silver bullet
  - Still a good idea to do parallel first touch
- **If dynamic scheduling cannot be avoided**
  - Consider round-robin placement as a quick (but non-ideal) fix
  - OpenMP 5.0 will have some data affinity support