ERLANGEN REGIONAL COMPUTING CENTER

Single Instruction Multiple Data (SIMD) processing

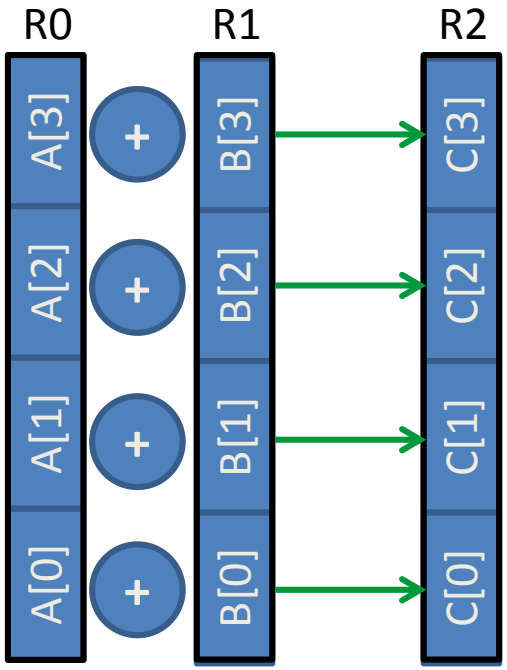
SIMD terminology

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

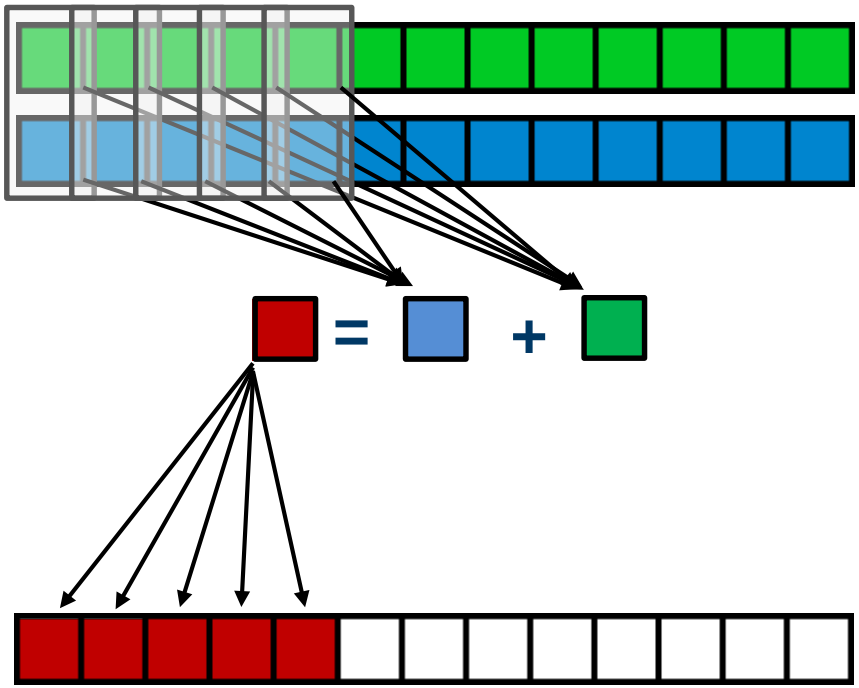
- 1 operand

- 2 operands (SSE)

- 4 operands (AVX)

- 8 operands (AVX512)

Scalar execution



Data-parallel execution units (short vector SIMD)

Single Instruction Multiple Data (SIMD)

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

Register widths

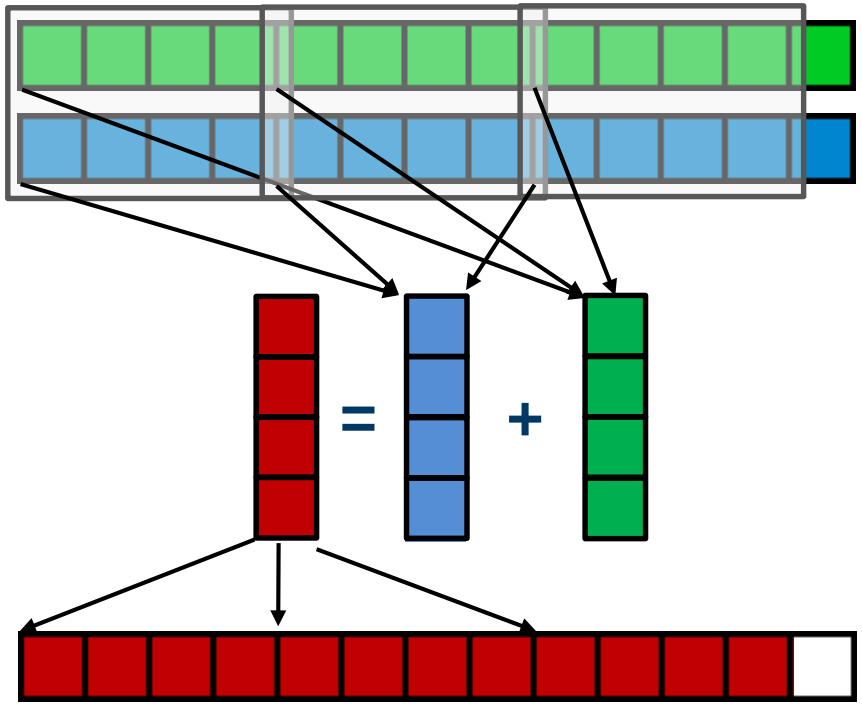
- 1 operand

- 2 operands (SSE)

- 4 operands (AVX)

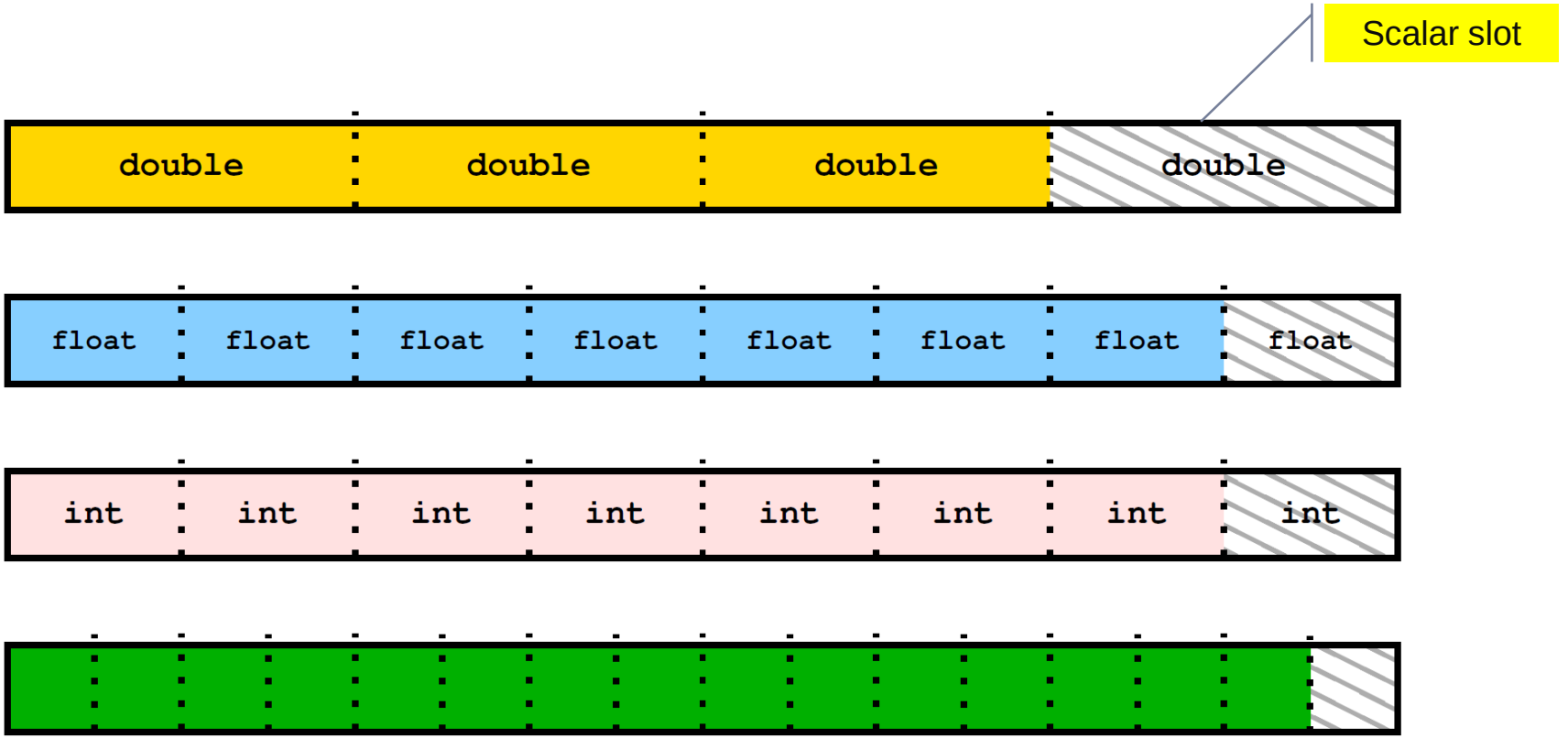
- 8 operands (AVX512)

SIMD execution

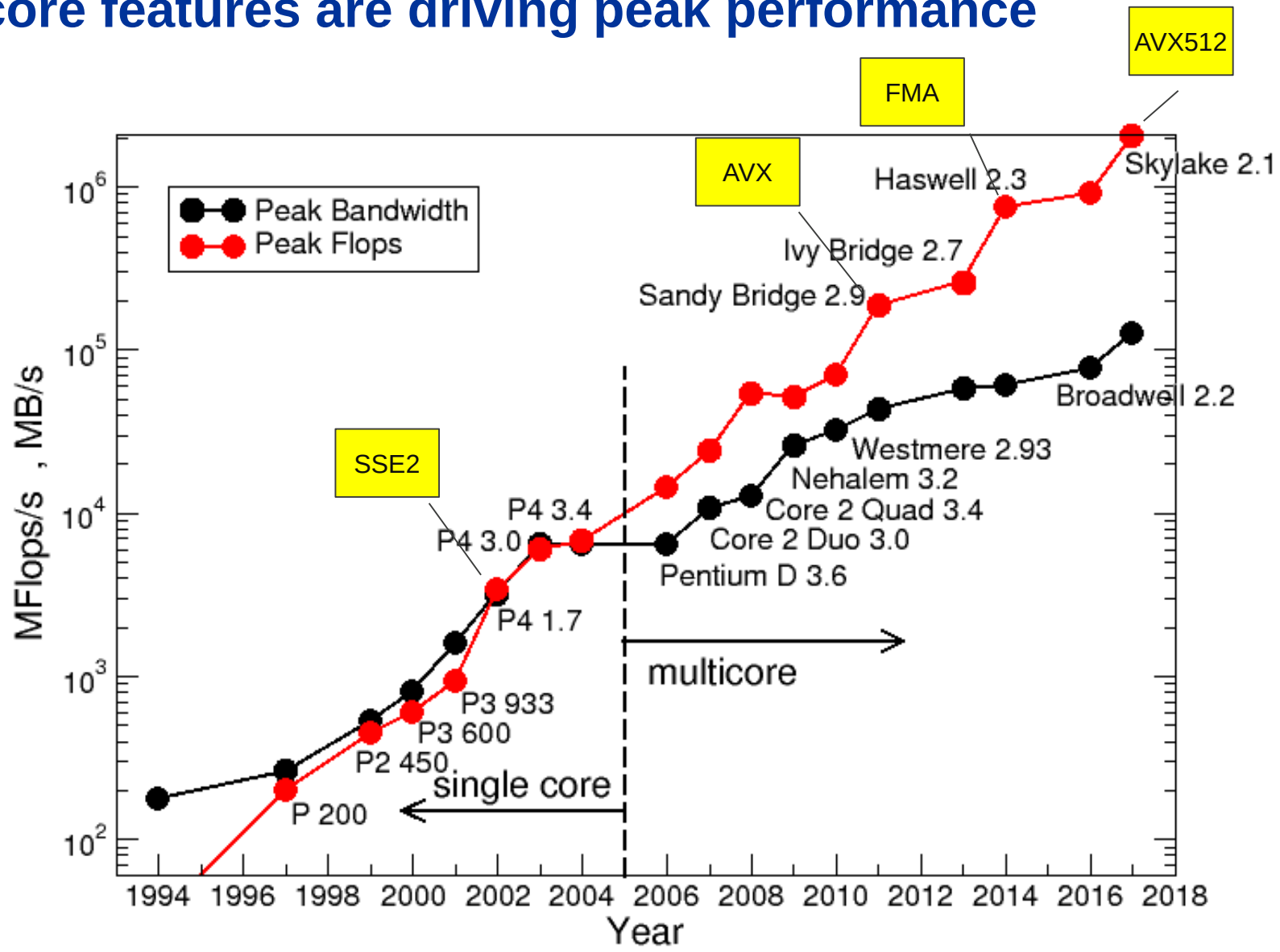


Example: Data types in 32-byte SIMD registers (AVX[2])

- Supported data types depend on actual SIMD instruction set

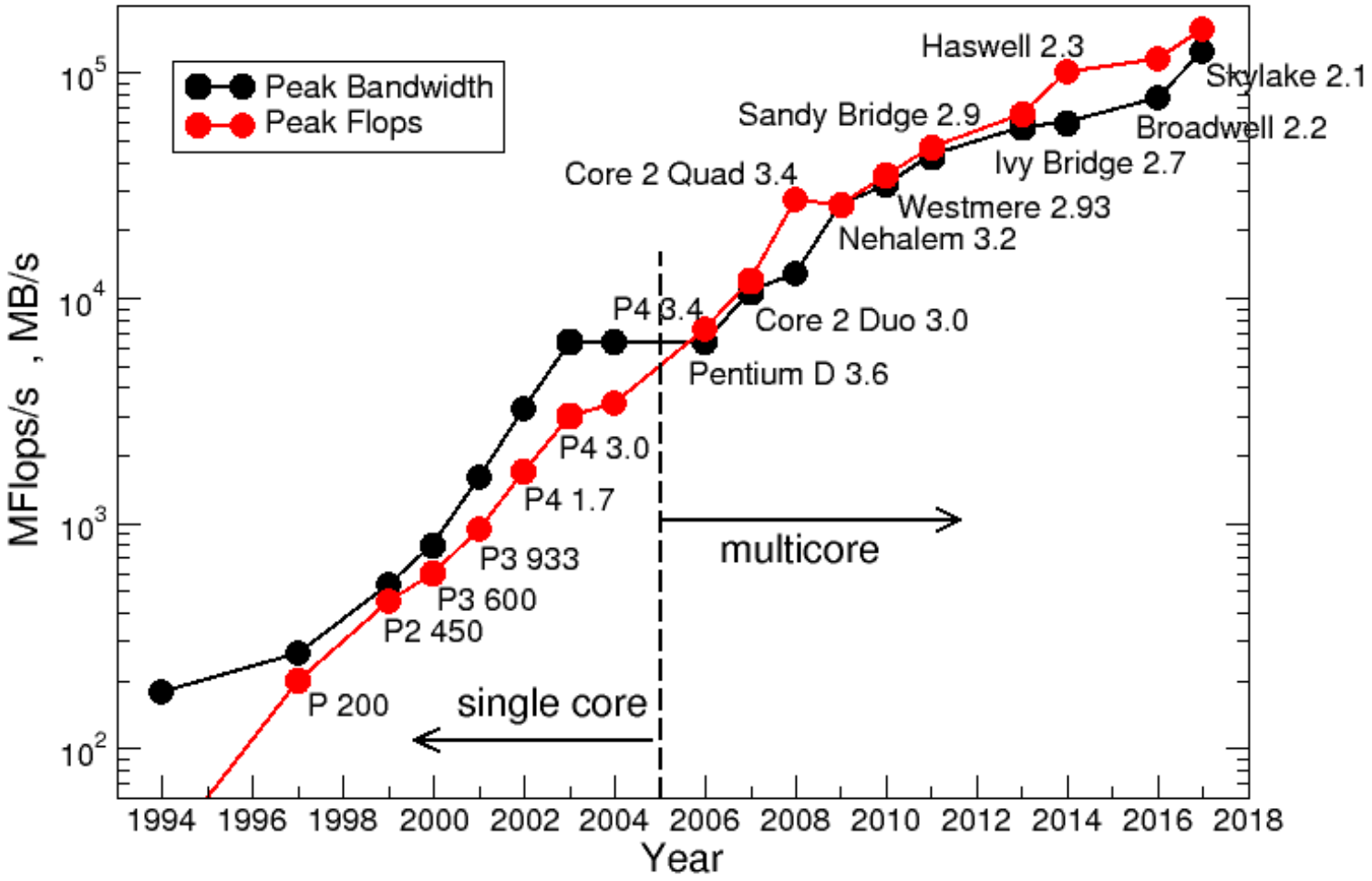


In-core features are driving peak performance

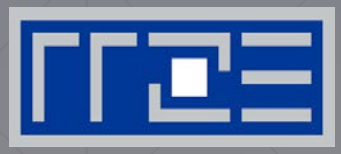


Von Neumann bottleneck reloaded: “DRAM gap”

DP peak performance and peak main memory bandwidth
for a single Intel processor (chip) – without SIMD and FMA



SIMD



The Basics

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
```

This should not be done by hand!

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: roadblocks

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

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)

- 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)

- 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, ...`

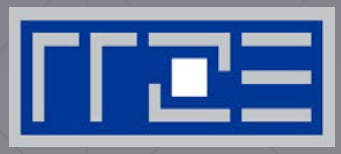
```
for (int j=0; j<n; j++) {
  #pragma omp simd reduction(+:b[j:1])
  for (int i=0; i<n; i++) {
     b[j] += a[j][i];
  }
}
```

x86 Architecture:
SIMD and Alignment

- 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)

Assembler: 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 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

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:

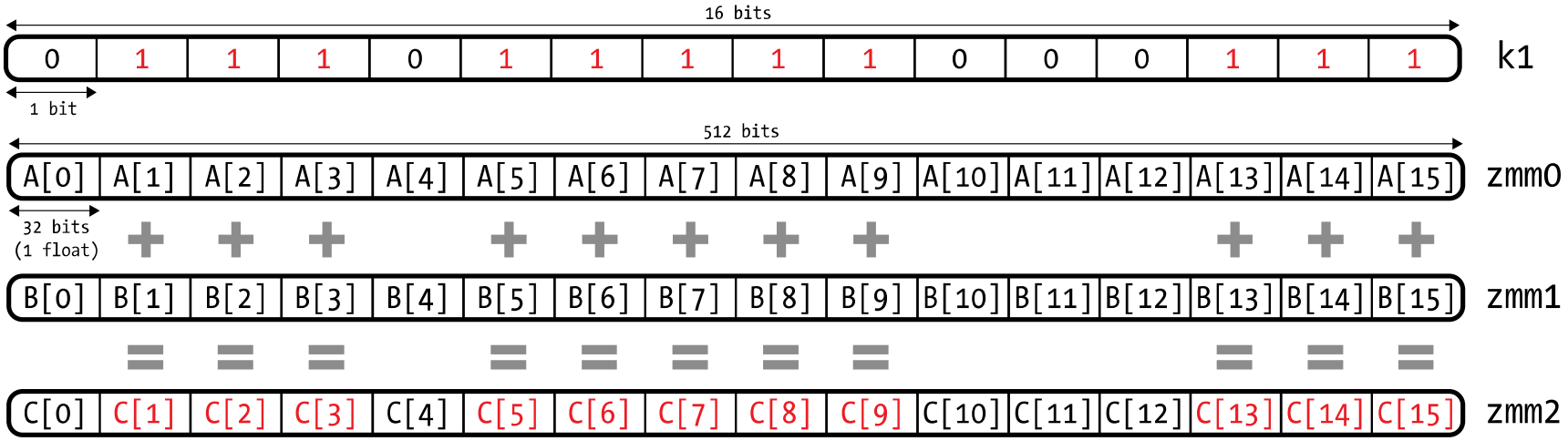
`xmm0-xmm15 (xmm31)`	SSE (128bit) alias with 256-bit and 512-bit registers
`ymm0-ymm15 (xmm31)`	AVX (256bit) alias with 512-bit registers
`zmm0-zmm31`	AVX-512 (512bit)

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

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



ISA support on Intel chips

The current Skylake architecture 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

Case Study: Sum reduction (double precision)

```
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 (double precision) – AVX512 version

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
```

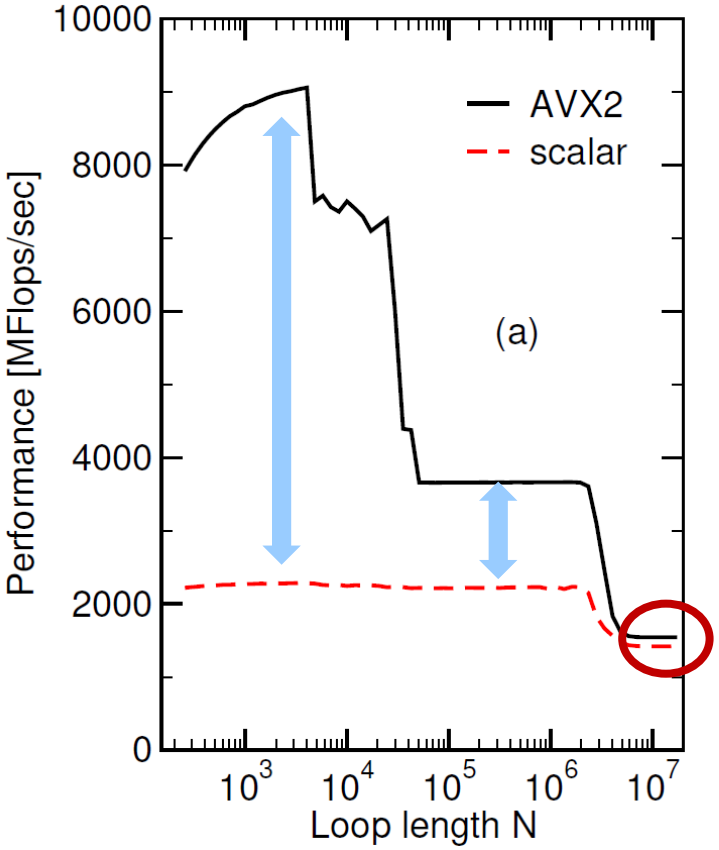
Remainder omitted

Bulk loop code (8x4-way unrolled)

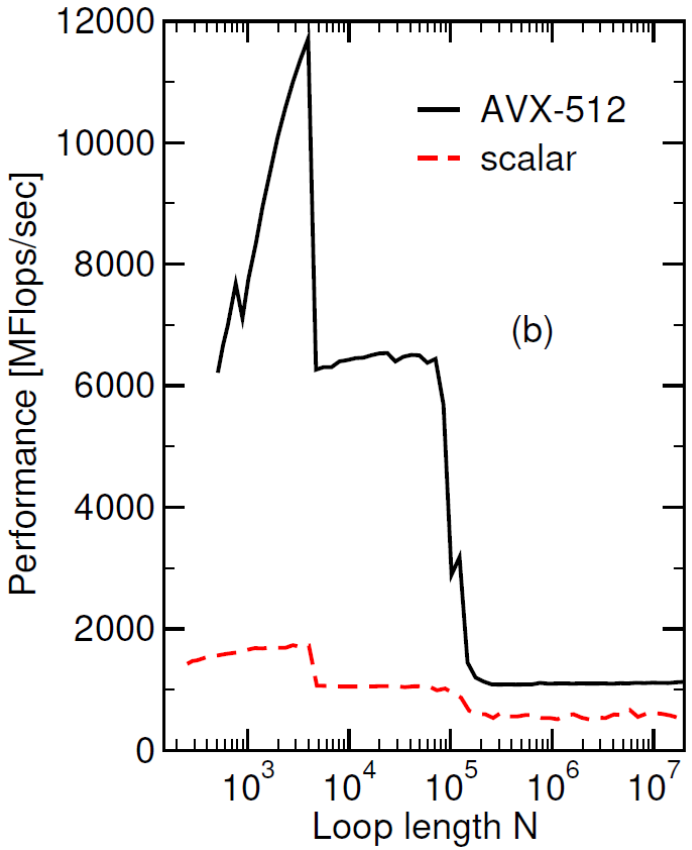
Sum up 32 partial sums into **`xmm0.0`**

Sum reduction (double precision) – sequential performance

Xeon “Broadwell” E5-2697v4



Xeon Phi 7210



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

SIMD with masking

```
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  %zmm4, %zmm5, %k1
        vcmpgtpd  %zmm4, %zmm6, %k2
        vcmpgtpd  %zmm4, %zmm7, %k3
        vcmpgtpd  %zmm4, %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)

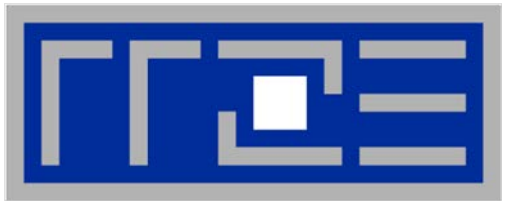
Rules and guidelines for vectorizable loops

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

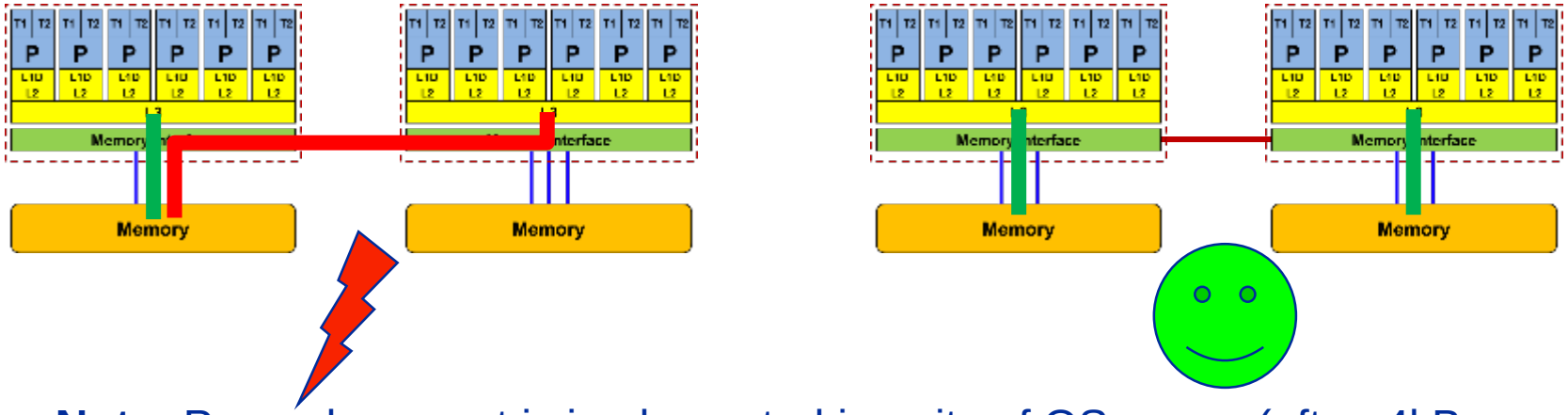
Performance characteristics of ccNUMA nodes

First touch placement policy

ccNUMA performance problems

"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?

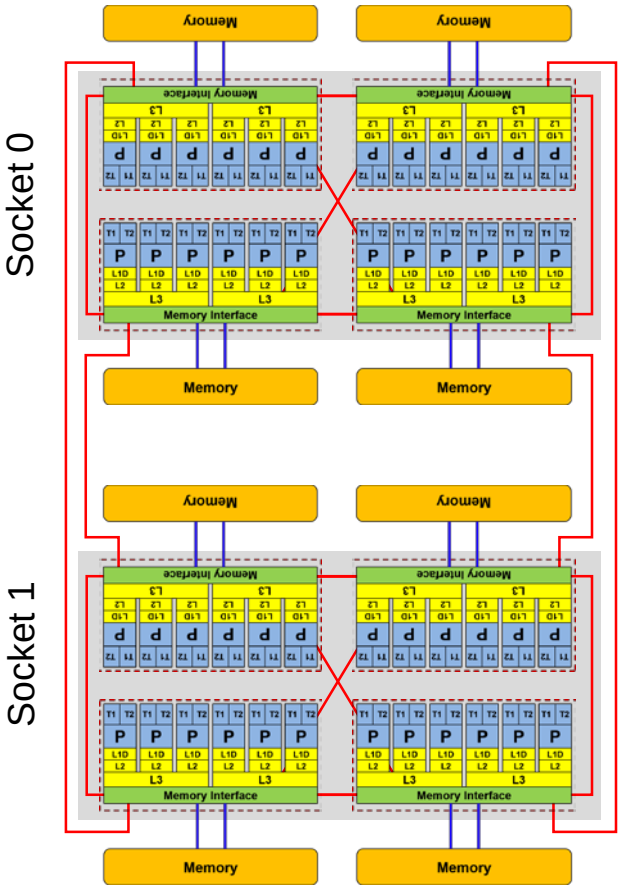


Note: Page placement is implemented in units of OS pages (often 4kB, possibly more)

How much bandwidth does nonlocal access cost?

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

CPU node / MEM 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



numactl as a simple ccNUMA locality tool :
How do we enforce some locality of access?

- **`numactl`** can influence the way a binary maps its memory pages:

```
numactl --membind=<nodes> a.out     # map pages only on <nodes>
        --preferred=<node>  a.out   # map pages on <node>
                                    # and others if <node> is full
        --interleave=<nodes> a.out  # map pages round robin across
                                    # all <nodes>
```

- Examples:

```
for m in `seq 0 7`; do
  for c in `seq 0 7`; do
    env OMP_NUM_THREADS=6 \
      numactl --membind=$m likwid-pin –c M${c}:0-5 ./stream
  done
done
```

ccNUMA map scan for EPYC system

```
numactl --interleave=0-7 likwid-pin -c E:N:8:1:12 ./stream
```

- But what is the default without **`numactl`**?

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
  - 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

Most simple case: explicit initialization

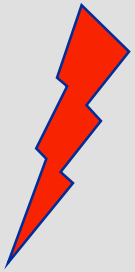
```
integer,parameter :: N=10000000
double precision A(N), B(N)



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
```

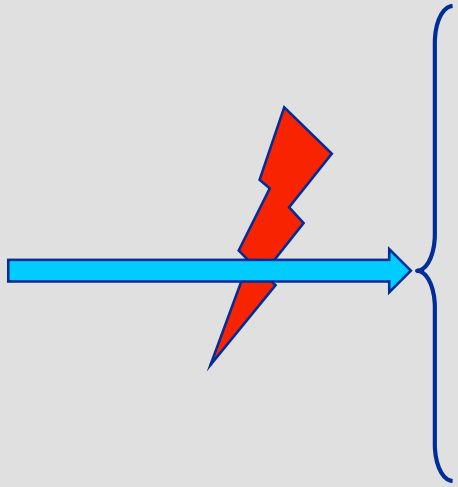
Coding for ccNUMA data locality

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

- 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

- 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 nonlocal 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 diagnosing bad ccNUMA access locality

- 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:

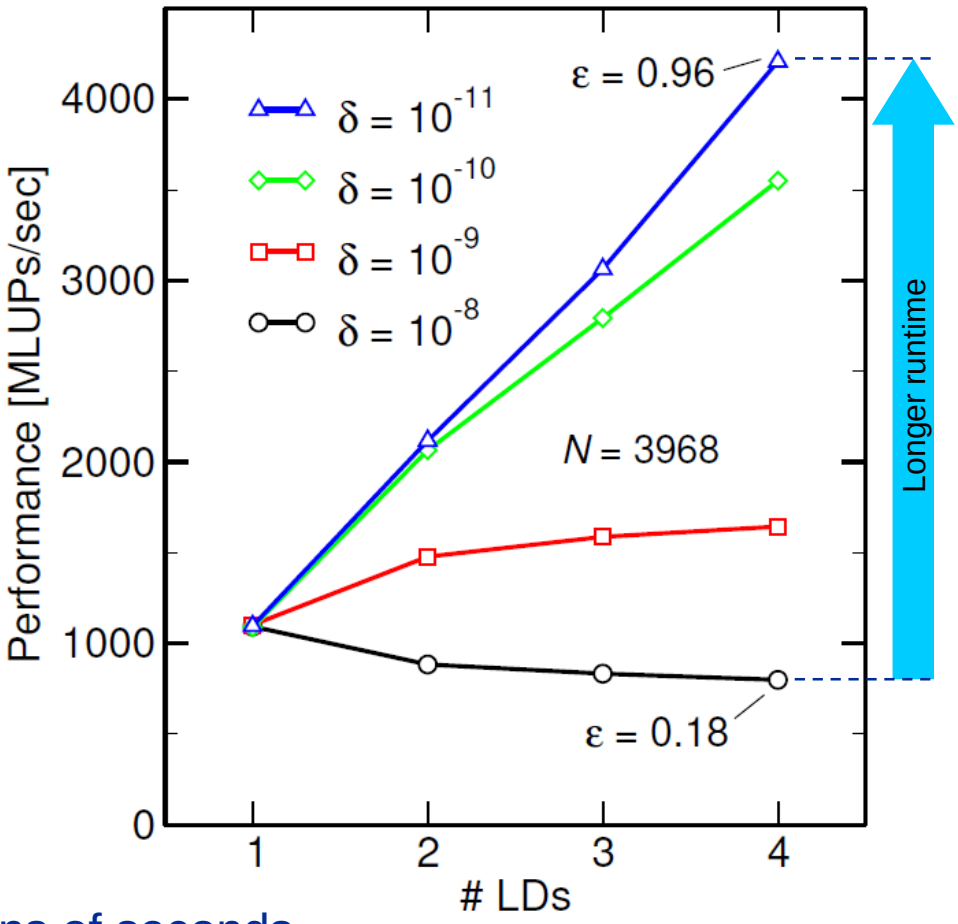
Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	4.050483	0.4050483	0.4050483	0.4050483
Runtime unhalted [s] STAT	3.03537	0.3026072	0.3043367	0.303537
Clock [MHz] STAT	32996.94	3299.692	3299.696	3299.694
CPI STAT	40.3212	3.702072	4.244213	4.03212
Local DRAM data volume [GByte] STAT	7.752933632	0.735579264	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

- Caveat: NUMA metrics vary strongly between CPU models

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

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
```

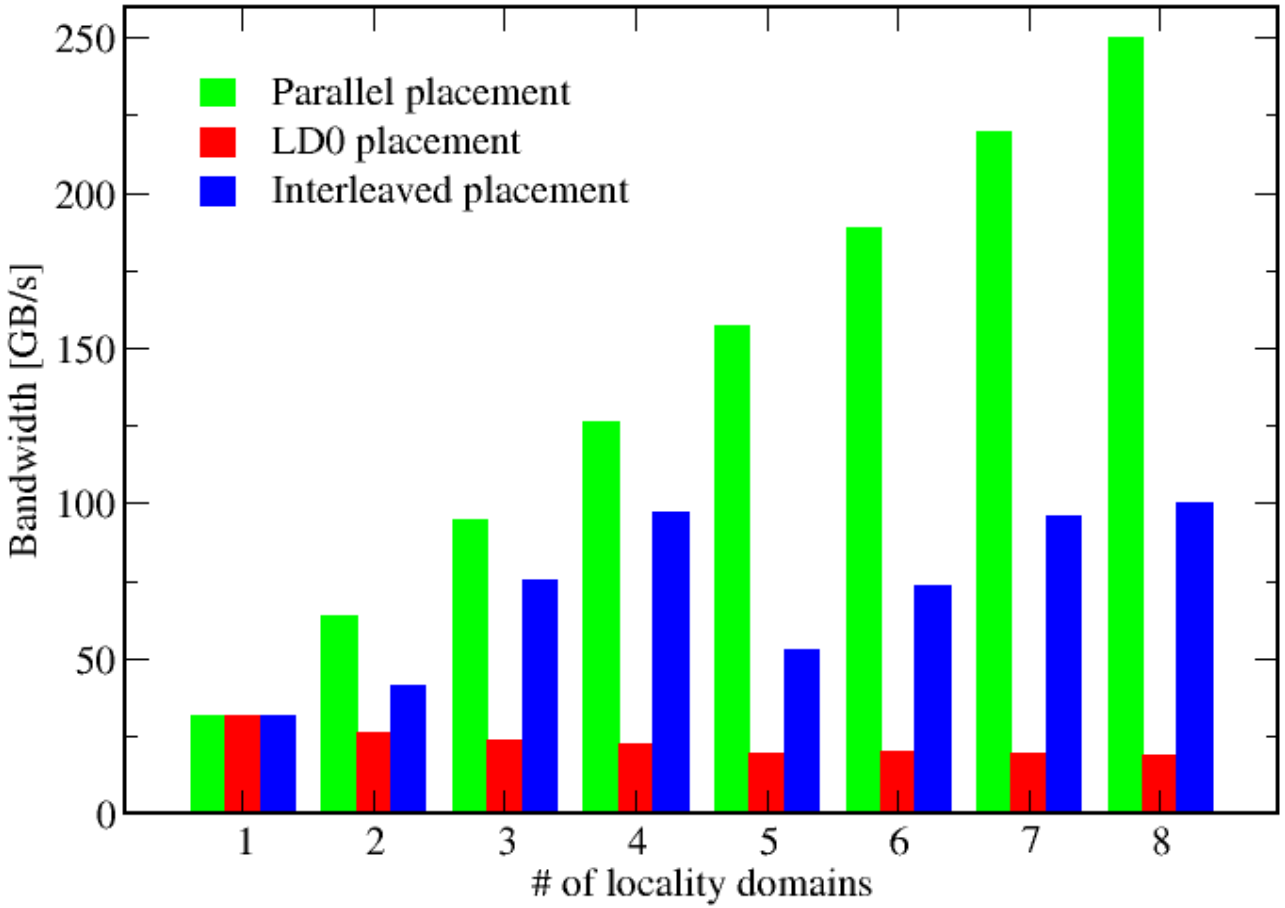
- Active on all current Linux distributions
- Parameters control aggressiveness

```
$ 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
```

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

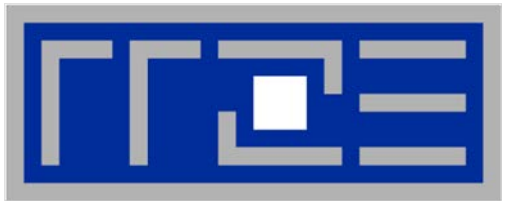
The curse and blessing of interleaved placement:
OpenMP STREAM triad on a dual AMD Epyc 7451 (6 cores per LD)

- Parallel init: Correct parallel initialization
- LD0: Force data into LD0 via **`numactl –m 0`**
- Interleaved: **`numactl --interleave <LD range>`**



Summary on ccNUMA issues

- **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



Simultaneous multithreading (SMT)

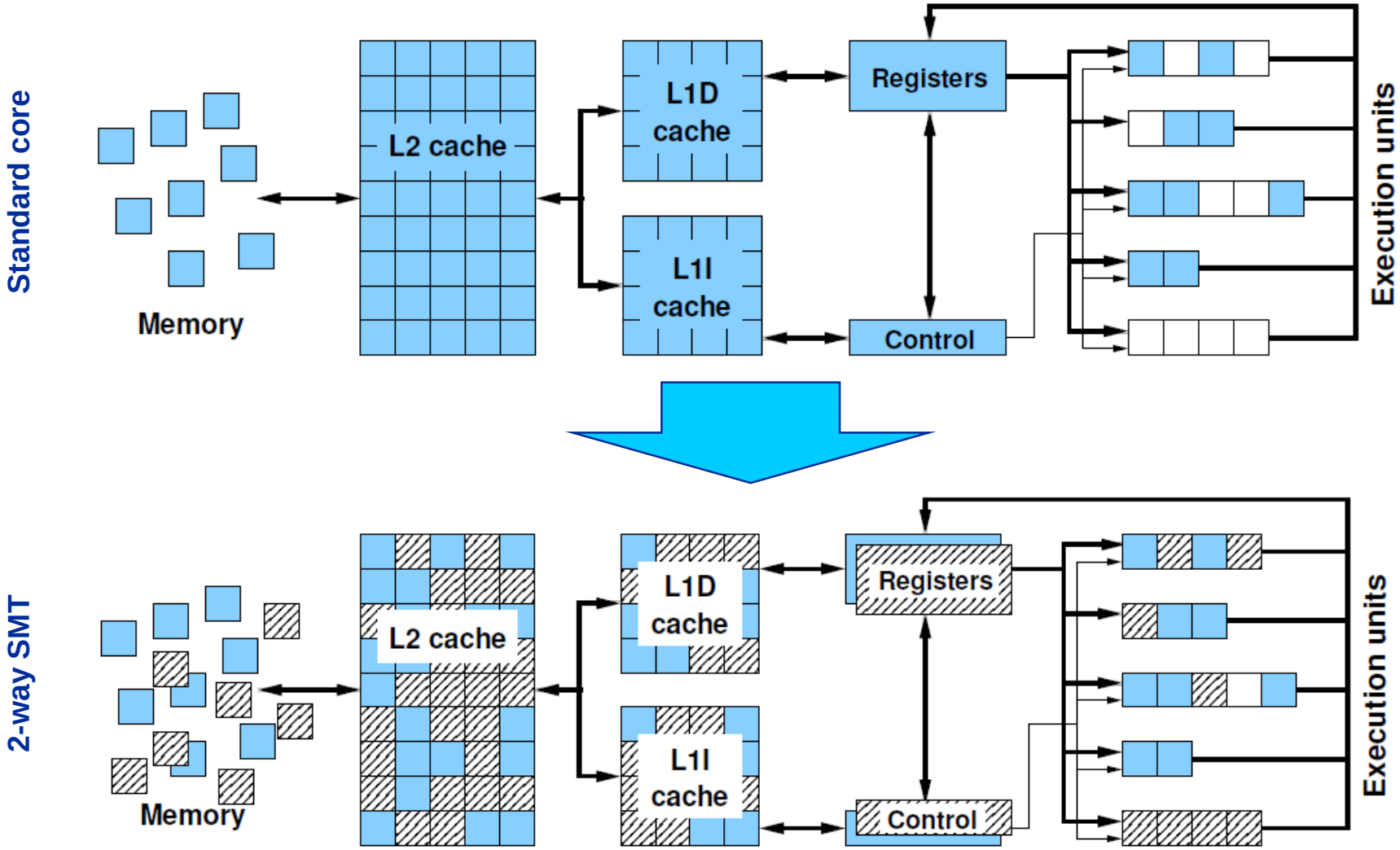
Principles and performance impact

SMT vs. independent instruction streams

Facts and fiction

SMT Makes a single physical core appear as two or more "logical" cores → multiple threads/processes run concurrently

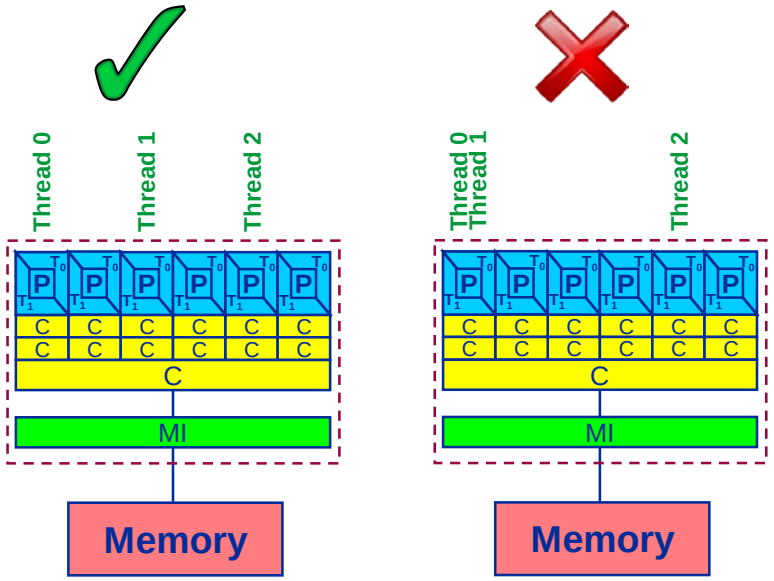
- SMT principle (2-way example):



SMT impact

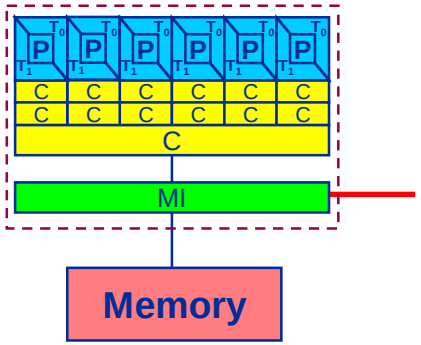
- SMT is primarily suited for increasing processor throughput
  - With multiple threads/processes running concurrently
- Scientific codes tend to utilize chip resources quite well
  - Standard optimizations (loop fusion, blocking, …)
  - High data and instruction-level parallelism
  - Exceptions do exist

- SMT is an important topology issue
  - SMT threads share almost all core resources
    - Pipelines, caches, data paths
  - Affinity matters!
  - If SMT is not needed
    - pin threads to physical cores
    - or switch it off via BIOS etc.



SMT impact

- Caveat: SMT threads share all caches!



- Possible benefit: Better pipeline throughput
  - Filling otherwise unused pipelines
  - Filling pipeline bubbles with other thread's executing instructions:

Thread 0:

```
do i=1,N
  a(i) = a(i-1)*c
enddo
```

Dependency → pipeline stalls until previous MULT is over

Thread 1:

```
do i=1,N
  b(i) = s*b(i-2)+d
enddo
```

Unrelated work in other thread can fill the pipeline bubbles

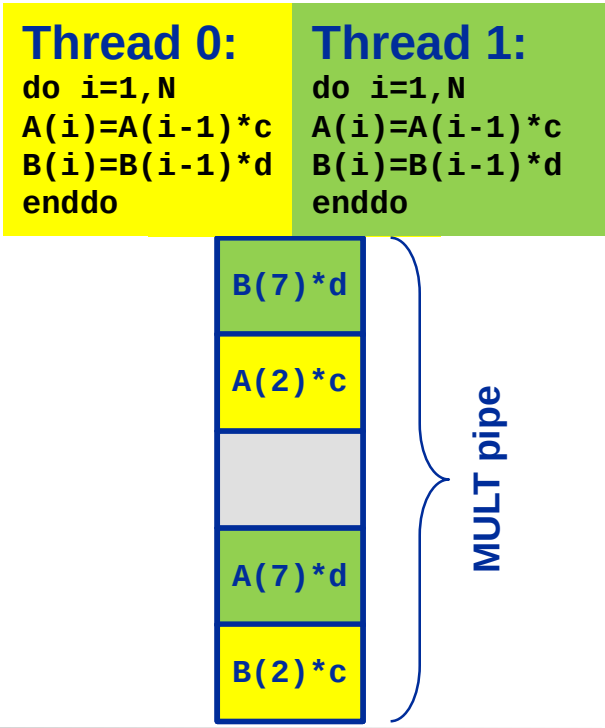
  - Beware: Executing it all in a single thread (if possible) may reach the same goal without SMT:

```
do i=1,N
  a(i) = a(i-1)*c
  b(i) = s*b(i-2)+d
enddo
```

The ideal workload for SMT

Simple loop-carried dependency benchmark `A(i) = s*A(i-1)`

- Bottleneck: MULT pipeline latency
  - Haswell CPU: 5 cy/it best case
- Running 2 threads via SMT: expect 2.5 cy/it if no other bottlenecks turn up
- Further improvement?
  - Multiple independent streams of instructions per thread
- What about the data transfer?

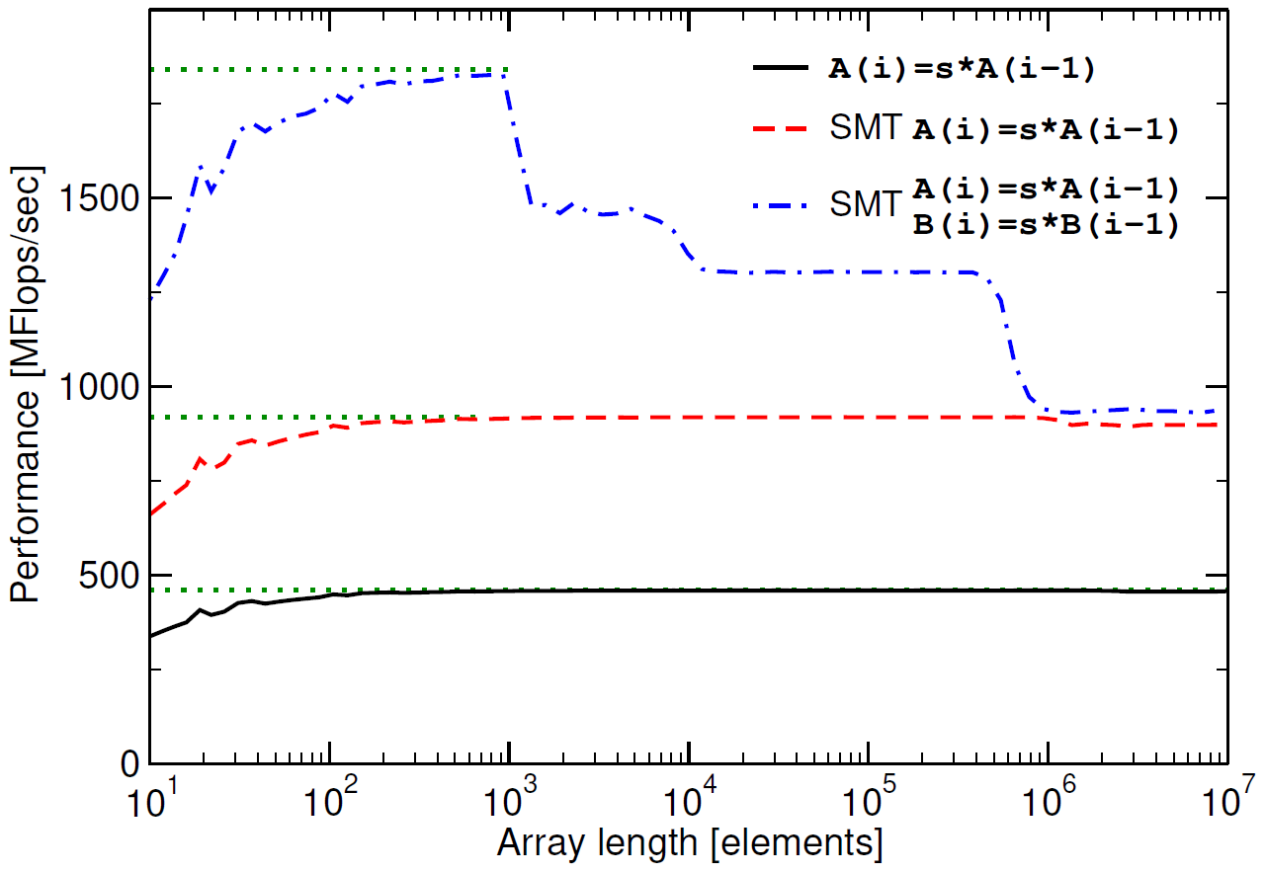


The ideal workload for SMT

Simple loop-carried dependency benchmark `A(i) = s*A(i-1)`

- Bottleneck: MULT pipeline latency
- Performance insensitive to problem size w/o SMT
- Fill bubbles via
  - SMT
  - Multiple indep. streams

Intel Xeon “Haswell” E5-2695v3, 2.3GHz



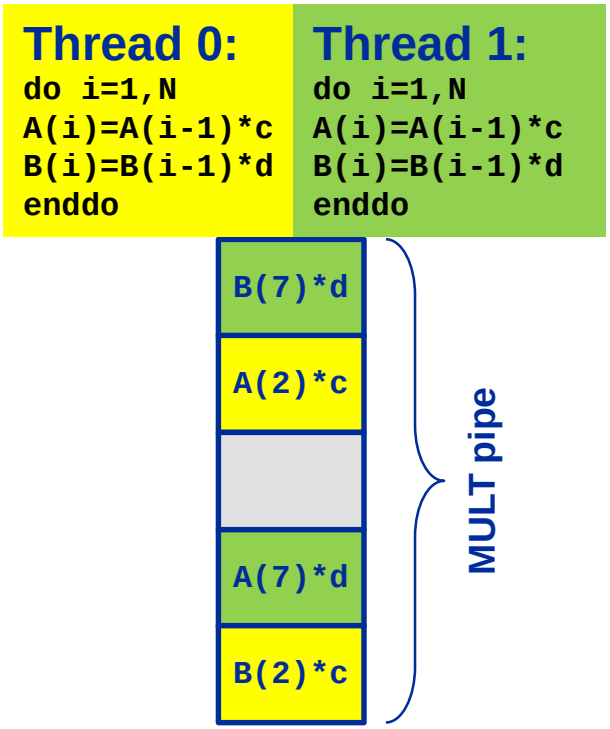
SMT myths: Facts and fiction (1)

Myth: “If the code is compute-bound, then the functional units should be saturated and SMT should show no improvement.”

Truth

1. A compute-bound loop does not necessarily saturate the pipelines; dependencies can cause a lot of bubbles, which may be filled by SMT threads.

2. If a pipeline is already full, SMT will not improve its utilization



SMT myths: Facts and fiction (2)

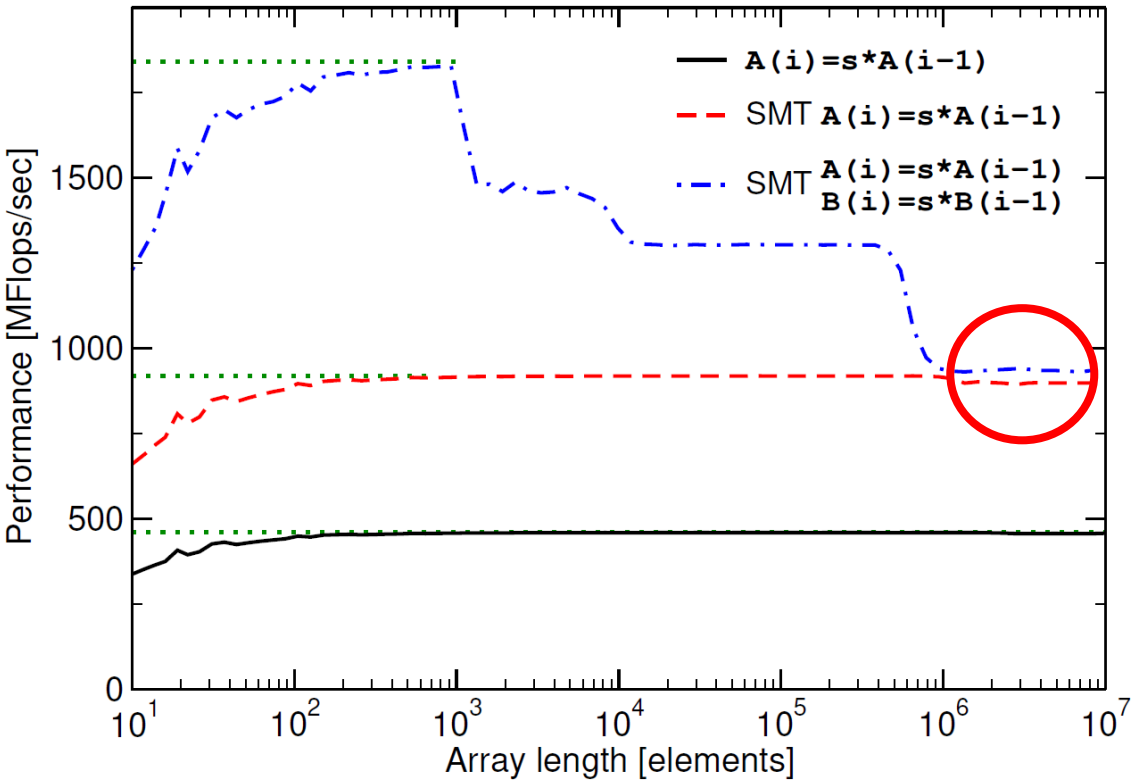
Myth: “If the code is memory-bound, SMT should help because it can fill the bubbles left by waiting for data from memory.”

Truth:

1. If the maximum memory bandwidth is already reached, SMT will not help since the relevant resource (bandwidth) is exhausted.

2. If the relevant bottleneck is not exhausted, SMT may help since it can fill bubbles in the LOAD pipeline.

This applies also to other “relevant bottlenecks!”



SMT myths: Facts and fiction (3)

Myth: “SMT can help bridge the latency to memory (more outstanding references).”

Truth:
Outstanding references may or may not be bound to SMT threads; they may be a resource of the memory interface and shared by all threads. The benefit of SMT with memory-bound code is usually due to better utilization of the pipelines so that less time gets “wasted” in the cache hierarchy.

See also the “ECM Performance Model” later on.