Node level performance optimization

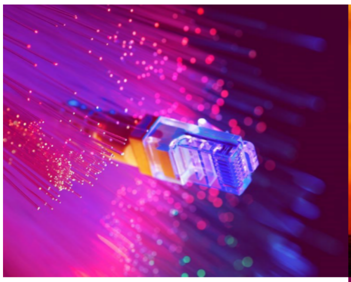
May 18 – 20, 2021

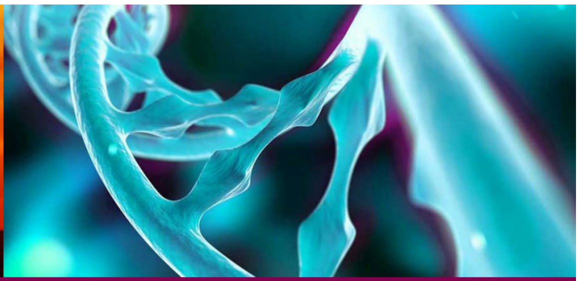
CSC – IT Center for Science Ltd., Espoo

Jussi Enkovaara, CSC
Mikko Byckling, Intel
Michael Klemm, AMD

CSC
Training for
Brilliant Minds

PRACE





Introduction to Application Performance

CSC Training, 2021-05

CSC – Finnish expertise in ICT for research, education and public administration

Course outline

- Analyzing and understanding performance issues
  - Awareness of modern CPUs
- Improving performance through vectorization
- Improving performance through memory optimization
- Improving performance though advanced threading techniques

Why worry about application performance?

- Obvious benefits
  - Better throughput => more science
  - Cheaper than new hardware
  - Save energy, compute quota, money etc.
- ...and some non-obvious ones
  - Potential cross-disciplinary research with computer science
  - Deeper understanding of application

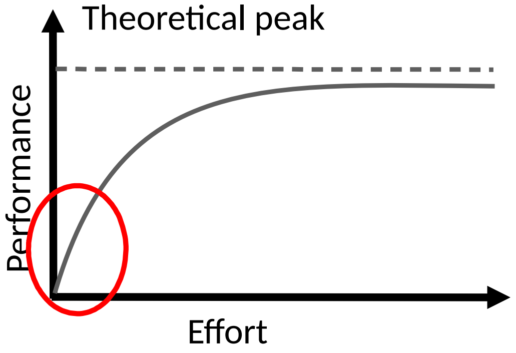
Factors affecting performance in HPC

- Single node performance
  - single core performance
  - threading (and MPI within a node)
- Communication between nodes
- Input/output to disk

How to improve single node performance?

- Choose good algorithm
  - e.g. $O(N\log N)$ vs. $O(N^2)$
  - remember prefactor!
- Use high performance libraries
  - linear algebra (BLAS/LAPACK), FFTs, ...
- Experiment with compilers and compiler options
  - There is no single best compiler and set of options for all use cases
- Experiment with threading options
  - Thread pinning, loop scheduling, ...
- Optimize the program code

```
./fibonacci 20
With loop, Fibonacci number i=20 is 6765
Time elapsed 79 ums
With recursion, Fibonacci number i=20 is 6765
Time elapsed 343773 ums
```



Doesn't the compiler do everything?

- You can make a big difference to code performance with how you express things
- Helping the compiler spot optimisation opportunities
- Using the insight of your application
  - language semantics might limit compiler
- Removing obscure (and obsolescent) “optimizations” in older code
  - Simple code is the best, until otherwise proven
- This is a dark art, mostly: optimize on case-by-case basis
  - First, check what the compiler is already doing

What the compiler is doing?

- Compilers have vast amount of heuristics for optimizing common programming patters
- Most compilers can provide a report about optimizations performed, with various amount of detail
    - See compiler manuals for all options
- Look into assembly code with `-S -fverbose-asm`

Compiler	Opt. report
GNU	`-fopt-info`
Intel	`-qopt-report`
Clang	`-Rpass=.*`

```
...
  vfmadd213pd %ymm0, %ymm2, %ymm10
  vfmadd213pd %ymm0, %ymm2, %ymm9
  vfmadd213pd %ymm0, %ymm2, %ymm8
...
```

Measuring performance

A day in life at CSC

CSC customer

I’m performing simulations with my Fortran code. It seems to perform much worse with MKL library in the new system than with IMSL library in the old system.

No

CSC specialist

Have you profiled your code?

A day in life at CSC

- Profiled the code: 99.9% of the execution time was being spent on these lines:

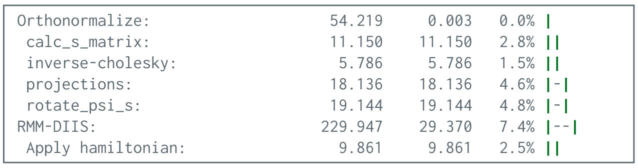
```
do i=1,n          ! Removing these unnecessary loop iterations reduced the
 do j=1,m         ! wall-time of one simulation run from 17 hours to 3 seconds…
    do k=1,fact(x)
      do o=1,nchoosek(x)
         where (ranktypes(:,:)==k)
            ranked(:,:,o)=rankednau(o,k)
         end where
      end do
    end do
  end do
end do
```

Measuring performance

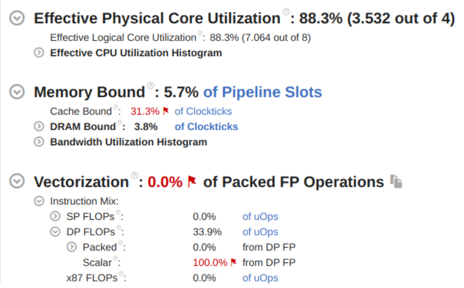
- First step should always be measuring the performance and finding performance critical parts
  - Application can contain hundreds of thousands of lines of code, but typically a small part of the code (~10 %) consumes most (~90%) of the execution time
  - “Premature code optimization is the root of all evil”
- Choose test case which represents a real production run
- Measurements should be carried out on the target platform
  - "Toy" run on laptop may provide only limited information

Profiling application

- Applications own timing information
  - Can be useful for big picture
- Performance analysis tools
  - Provide detailed information about the application
  - Find hot-spots (functions and loops)
  - Identify causes of less-than-ideal performance
  - Information about low-level hardware
  - **Intel VTune**, **AMD uProf**, perf, Tau, Scalasca, PAPI, …
  - http://www.vi-hps.org/tools/tools.html



```
Orthonormalize:                    54.219     0.003    0.0% |
 calc_s_matrix:                    11.150    11.150    2.8% ||
 inverse-cholesky:                  5.786     5.786    1.5% ||
 projections:                      18.136    18.136    4.6% |-|
 rotate_psi_s:                     19.144    19.144    4.8% |-|
RMM-DIIS:                         229.947    29.370    7.4% |--|
 Apply hamiltonian:                 9.861     9.861    2.5% ||
```



Effective Physical Core Utilization: 88.3% (3.532 out of 4)

Effective Logical Core Utilization: 88.3% (7.064 out of 8)

Effective CPU Utilization Histogram

Memory Bound: 5.7% of Pipeline Slots

Cache Bound: 31.3% of Clockticks

DRAM Bound: 3.8% of Clockticks

Bandwidth Utilization Histogram

Vectorization: 0.0% of Packed FP Operations

Instruction Mix:

SP FLOPs:	0.0%	of uOps
DP FLOPs:	33.9%	of uOps
Packed:	0.0%	from DP FP
Scalar:	100.0%	from DP FP
x87 FLOPs:	0.0%	of uOps

Profiling application

- Collecting all possible performance metrics with single run is not practical
  - Simply too much information
  - Profiling overhead can alter application behavior
- Start with an overview!
  - Call tree information, what routines are most expensive?

Sampling vs. tracing

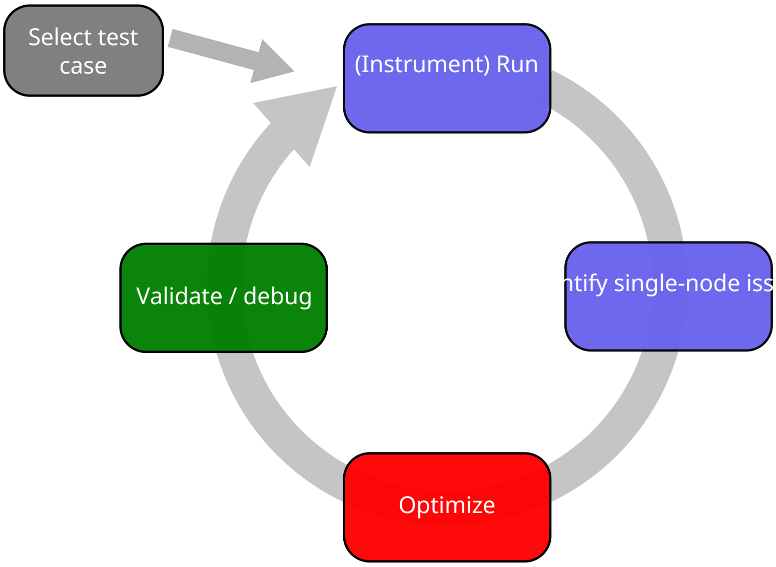
- When application is profiled using sampling, the execution is stopped at predetermined intervals and the state of the application is examined
  - Lightweight, but may give skewed results
- Tracing records every event, e.g. function call
  - Usually requires modification to the executable
    - These modifications are called instrumentation
  - More accurate, but may affect program behavior
  - Generates lots of data

Hardware performance counters

- Hardware performance counters are special registers on CPU that count hardware events
- They enable more accurate statistics and low overhead
  - In some cases they can be used for tracing without any extra instrumentation
- Number of counters is much smaller than the number of events that can be recorded
- Different CPUs have different counters

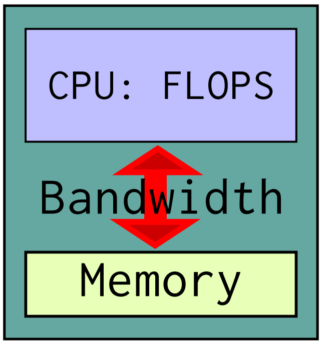
Optimizing program

Code optimization cycle



How to assess application's performance?

- Two fundamental limits
- CPUs peak floating point performance
  - clock frequency
  - number of instructions per clock cycle
  - number of FLOPS per instruction
  - number of cores
  - no real application achieves peak in sustained operation
- Main memory bandwidth
  - How fast data can be fed to the CPU



How to assess application's performance?

- Example: maximum performance of **axpy** `x[i] = a x[i] + y[j]`
  - Two FLOPS (multiply and add) per `i`
  - Three memory references per `i`
  - With double precision numbers arithmetic intensity $I = \frac{\mathrm{FLOPS}}{\mathrm{memorytraffic}} = \frac{2}{3*8} = 0.08$ FLOPS/byte
  - In Puhti, memory bandwidth is ~200 GB/s, so maximum performance is ~16 GFLOPS/s
  - Theoretical peak performance of Puhti node is ~2600 GFLOPS/s

How to assess application's performance?

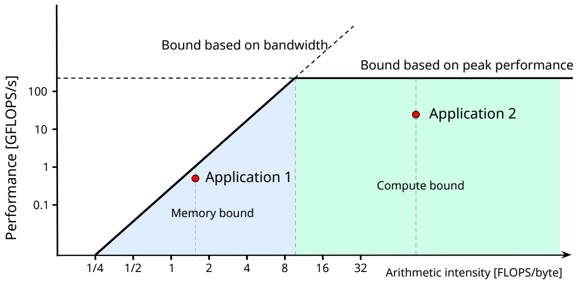
- Example: matrix-matrix multiplication `C[i,j] = C[i,j] + A[i,k] * B[k,j]`
  - $2N^3$ FLOPS
  - $3N^2$ memory references
  - With double precision numbers arithmetic intensity $I = \frac{2N}{3}$ FLOPS/byte
  - With large enough N limited by peak performance

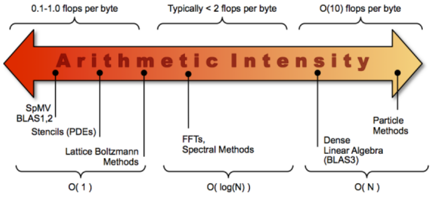
Roofline model

- Simple visual concept for maximum achievable performance
  - can be derived in terms of arithmetic intensity I, peak performance π and peak memory bandwidth β

$$P = min \begin{cases} \pi \\ \beta \times I \end{cases}$$

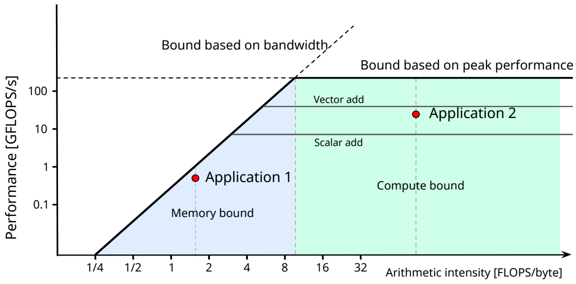
- Machine balance = arithmetic intensity needed for peak performance
  - Typical values 5-15 FLOPS/byte
- Additional ceilings can be included (caches, vectorization, threading)





Roofline model

- Model does not tell if code can be optimized or not
  - Application 1 may not be *fundamentally* memory bound, but only implemented badly (not using caches efficiently)
  - Application 2 may not have *fundamentally* prospects for higher performance (performs only additions and not fused multiply adds)
- However, can be useful for guiding the optimization work



Roofline model

- How to obtain the machine parameters?
  - CPU specs
  - own microbenchmarks
  - special tools (Intel tools, Empirical Roofline Tool)
- How to obtain application GFLOPS/s and arithmetic intensity?
  - Pen and paper and timing measurements
  - Performance analysis tools and hardware counters
  - *True* number of memory references can be difficult to obtain

Take-home messages

- Mind the application performance: it is for the benefit of you, other users and the service provider
- Profile the code and identify the performance issues first, before optimizing anything
  - “Premature code optimization is the root of all evil”
- Optimizing the code should be the last step in performance tuning
- Serial optimization is mostly about helping the compiler to optimize for the target CPU
- Roofline model can work as a guide in optimization

Web resources

- Roofline performance model and Empiral Roofline Tool
    - https://crd.lbl.gov/departments/computer-science/par/research/roofline/
- Web service for looking assembly output from multitude of compilers
    - https://gcc.godbolt.org

A look into modern CPU architecture

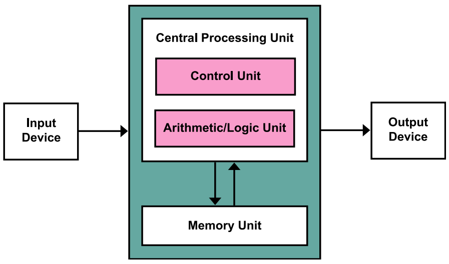
CSC Training, 2021-05

CSC – Finnish expertise in ICT for research, education and public administration

Modern CPU core

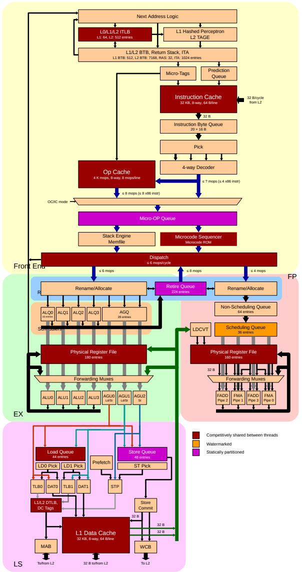
von Neumann architecture

- A CPU core is still largely based on the von Neumann model
  - sequency of operations (instructions) performed on given data
  - instructions and data are fetched from memory into registers in CPU
  - ALU performs operations on data in registers
  - Result is stored back to memory
- From an external point of view, operations are executed sequentially



Modern CPU core

- Internally, each core is highly complex
- **Superscalar out-of-order** instruction execution
- **SIMD** instructions
- Multiple levels of hierarchical **cache memory**

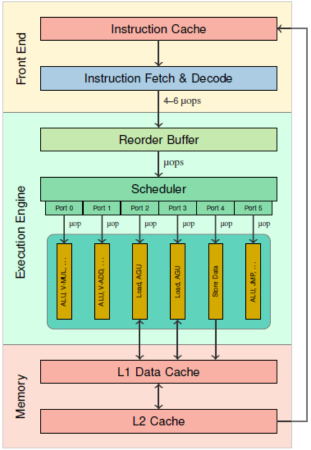


How CPU core operates?

- Clock frequency determines the pace at which CPU works
- Zero to **N** instructions start at each clock cycle
- Instruction latency = number of clock cycles that are required for completing the execution
- Instruction throughput = number of clock cycles to wait before starting same kind of instruction again
  - Throughput can be much smaller than the latency
  - Sometimes given as cycles per instruction (CPI) or its inverse, instructions per cycle (IPC)

Fetch-decode-execute cycle

- Instructions are executed in stages
- Fetch (F): control unit fetches instruction from memory
- Decode (D): decode the instruction and determine operands
  - Instructions are broken into uops
- Execute (E): perform the instruction
  - Utilize ALU or access memory
- Enables simpler logic and **pipelining** the operations

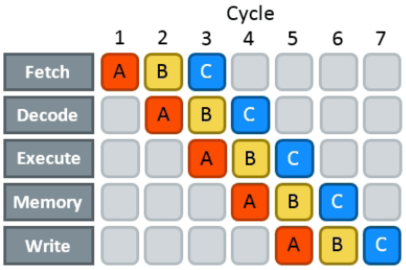


Pipelining

- Instruction execution and arithmetic units can be *pipelined*
  - Instruction execution: work on multiple instructions simultaneously
  - Arithmetic units: execute different stages of a an instruction at the same time in an assembly line fashion
  - Together: one result per cycle after the pipeline is full
- Within the pipeline, hardware can execute instructions in different order than they were issued (**out-of-order** scheduling)
- Requires complicated software (compiler) and hardware to keep the pipeline full
- Conditional branches can cause the pipeline to stall

Pipelining: example

- Wind-up and wind-down phases: no instructions retired
- First result available after 5 cycles, total time 7 cycles compared to 15 cycles without a pipeline
- Real pipeline in modern CPU cores can be much more complex

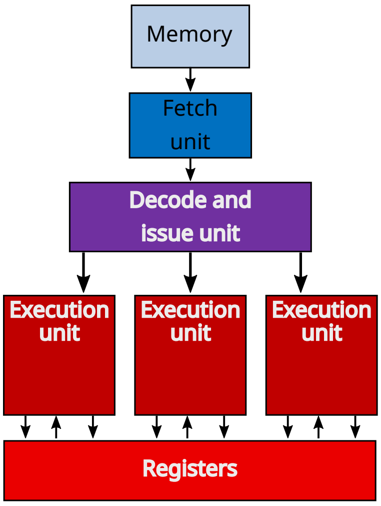


Cycle

	1	2	3	4	5	6	7
Fetch	A	B	C				
Decode		A	B	C			
Execute			A	B	C		
Memory				A	B	C	
Write					A	B	C

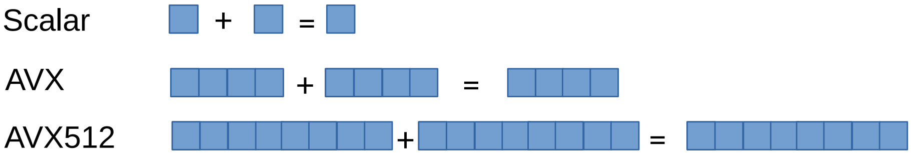
Superscalar execution

- Hardware Instruction Level Parallelism (ILP)
- Multiple instructions per cycle issued to the multiple execution units
- Hardware data dependency resolution preserve sequential execution semantics
  - Actual execution may be out-of-order
- Pipelining and superscalar execution allow instruction throughputs less than one



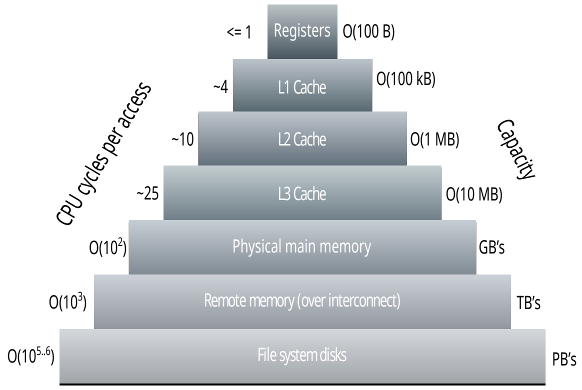
Vectorization

- Modern CPUs have SIMD (Single Instruction, Multiple Data) units and instructions
  - Operate on multiple elements of data with single instructions
- AVX2 256 bits = 4 double precision numbers
- AVX512 512 bits = 8 double precision numbers
  - single AVX512 fused multiply add instruction can perform 16 FLOPS



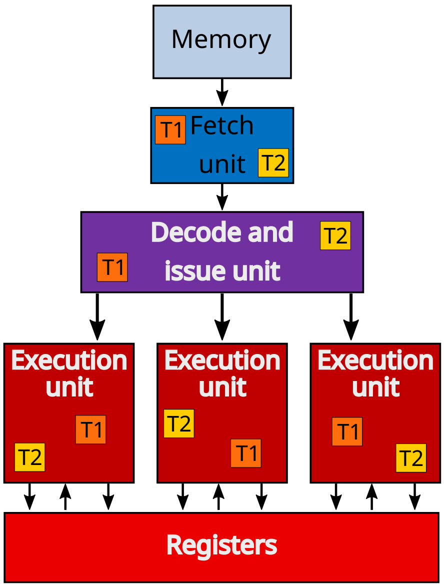
Cache memory

- In order to alleviate the memory bandwidth bottleneck, CPUs have multiple levels of cache memory
  - when data is accessed, it will be first fetched into cache
  - when data is reused, subsequent access is much faster
- L1 cache is closest to the CPU core and is fastest but has smallest capacity
- Each successive level has higher capacity but slower access



Symmetric Multithreading (SMT)

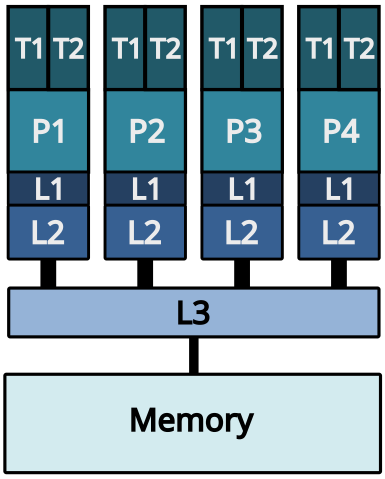
- It is difficult to fill-in all the available hardware resources in a CPU core
  - Pipeline stalls due to main memory latency, I/O, etc.
- To maximize hardware utilization, several hardware threads can be executed on a single core
  - Seen as logical cores by OS
- Benefits depend on the application, and SMT can also worsen the performance



Introduction to modern multicore CPUs

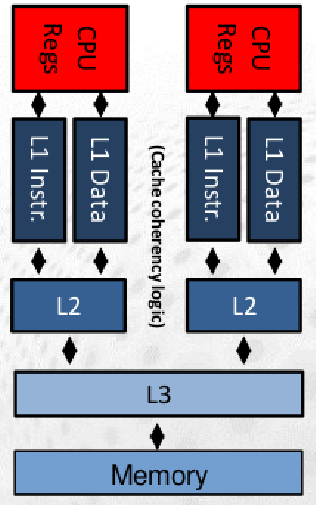
Multicore CPU schematic

- The multicore CPU is packeted in a socket
- Typically, L1 and L2 caches are private per core, and L3 cache is shared between set of cores
- All cores have shared access to the main memory



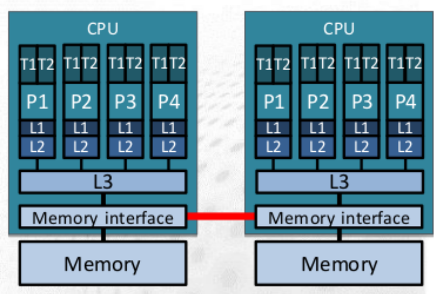
Cache coherency

- With private caches per core, hardware needs to ensure that the data is consistent between the cores
- When a core writes to a cache, CPU may need to update the caches of other cores
  - Possibly expensive operation



NUMA architectures

- A node can have multiple sockets with memory attached to each socket
- Non Uniform Memory Access (NUMA)
  - All memory within a node is accessible, but latencies and bandwidths vary
- Hardware needs to maintain cahce coherency also between different NUMA nodes (ccNUMA)



Summary

- Modern multicore CPUs are complex beasts
- In order to maximally utilize the CPU, application needs to:
  - use multiple threads (or processes)
  - utilize caches for feeding data to CPU at fastest possible pace
  - keep the pipeline full and utilize instruction level parallelism
  - use vector instructions for maximizing FLOPS per instruction

Web resources

- Detailed information about processor microarchitectures:
  - https://en.wikichip.org/wiki/WikiChip
  - https://uops.info/
- Agner's optimization resources https://www.agner.org/optimize/

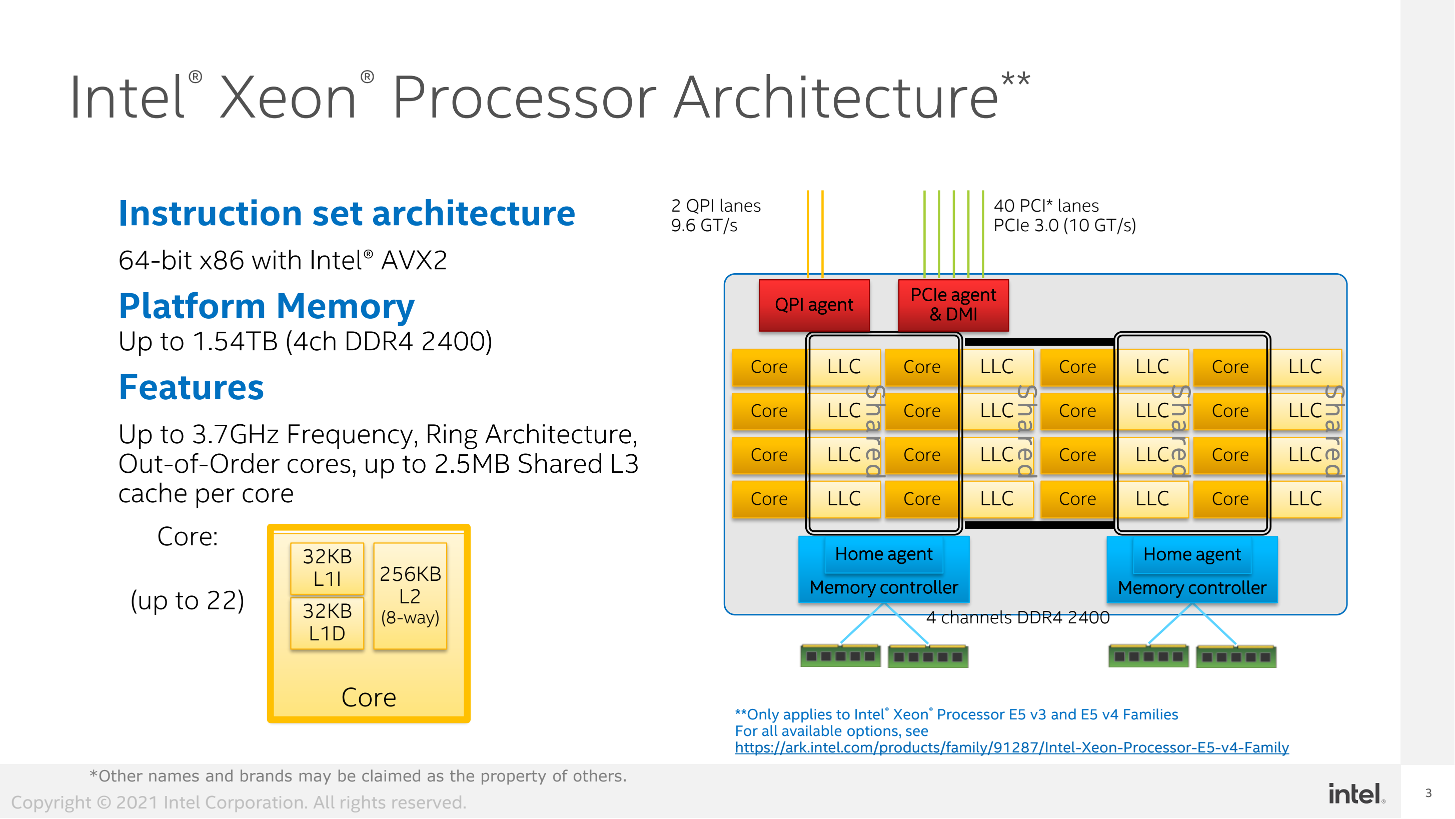
[ONLINE] Node Level Performance Optimization @ CSC, 18-20.5.2021

Performance optimization for Intel® Xeon® Processor architecture

Dr. Mikko Byckling, IAGS DEE XCSS

Contents

- Intel® microarchitectures
  - Intel® Xeon® Processors (codename “Broadwell”, BDW)
  - 2nd generation Intel® Xeon® Scalable Processors (codename “Cascade Lake-SP”, CLX)
- Introduction to SIMD ISA for Intel® processors
  - Intel® AVX and Intel® AVX2
  - Intel® AVX-512 and AVX-512 VNNI



Intel® Xeon® Processor Architecture**

Instruction set architecture

64-bit x86 with Intel® AVX2

Platform Memory

Up to 1.54TB (4ch DDR4 2400)

Features

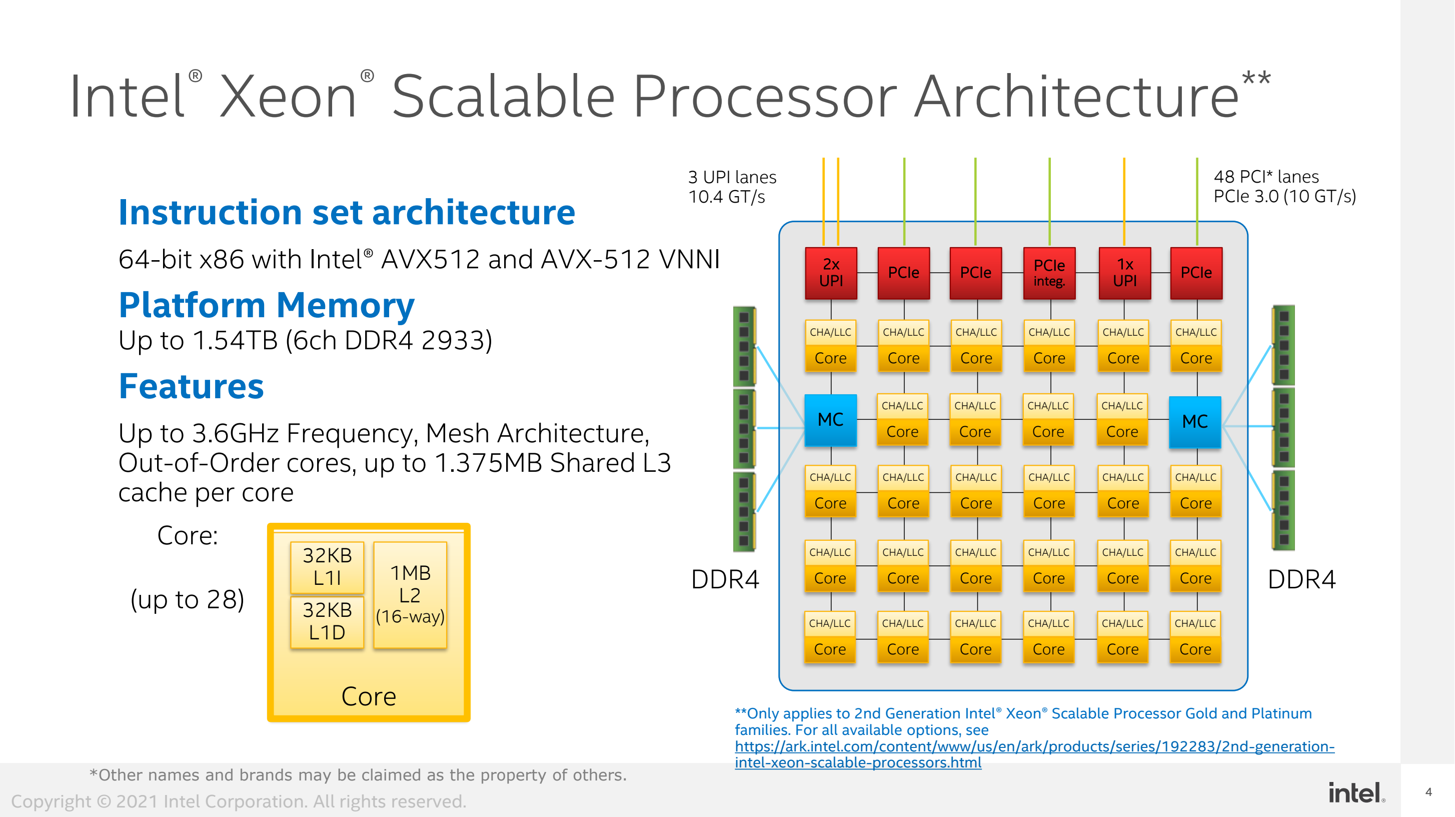
Up to 3.7GHz Frequency, Ring Architecture, Out-of-Order cores, up to 2.5MB Shared L3 cache per core

Core:

(up to 22)

**Only applies to Intel® Xeon® Processor E5 v3 and E5 v4 Families
For all available options, see https://ark.intel.com/products/family/91287/Intel-Xeon-Processor-E5-v4-Family

*Other names and brands may be claimed as the property of others.



Intel® Xeon® Scalable Processor Architecture**

Instruction set architecture

64-bit x86 with Intel® AVX512 and AVX-512 VNNI

Platform Memory

Up to 1.54TB (6ch DDR4 2933)

Features

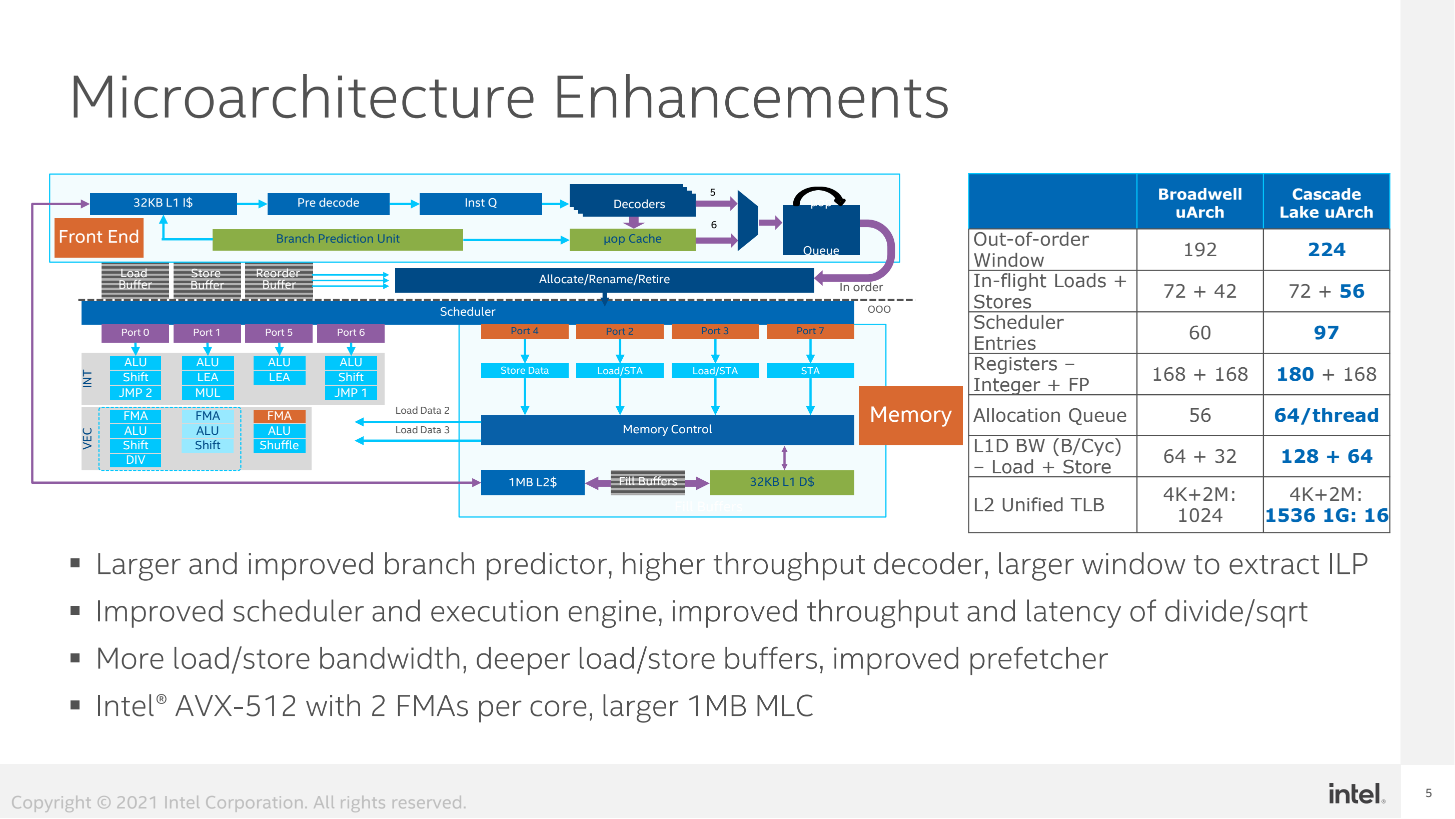
Up to 3.6GHz Frequency, Mesh Architecture, Out-of-Order cores, up to 1.375MB Shared L3 cache per core

Core:

(up to 28)

**Only applies to 2nd Generation Intel® Xeon® Scalable Processor Gold and Platinum families. For all available options, see https://ark.intel.com/content/www/us/en/ark/products/series/192283/2nd-generation-intel-xeon-scalable-processors.html

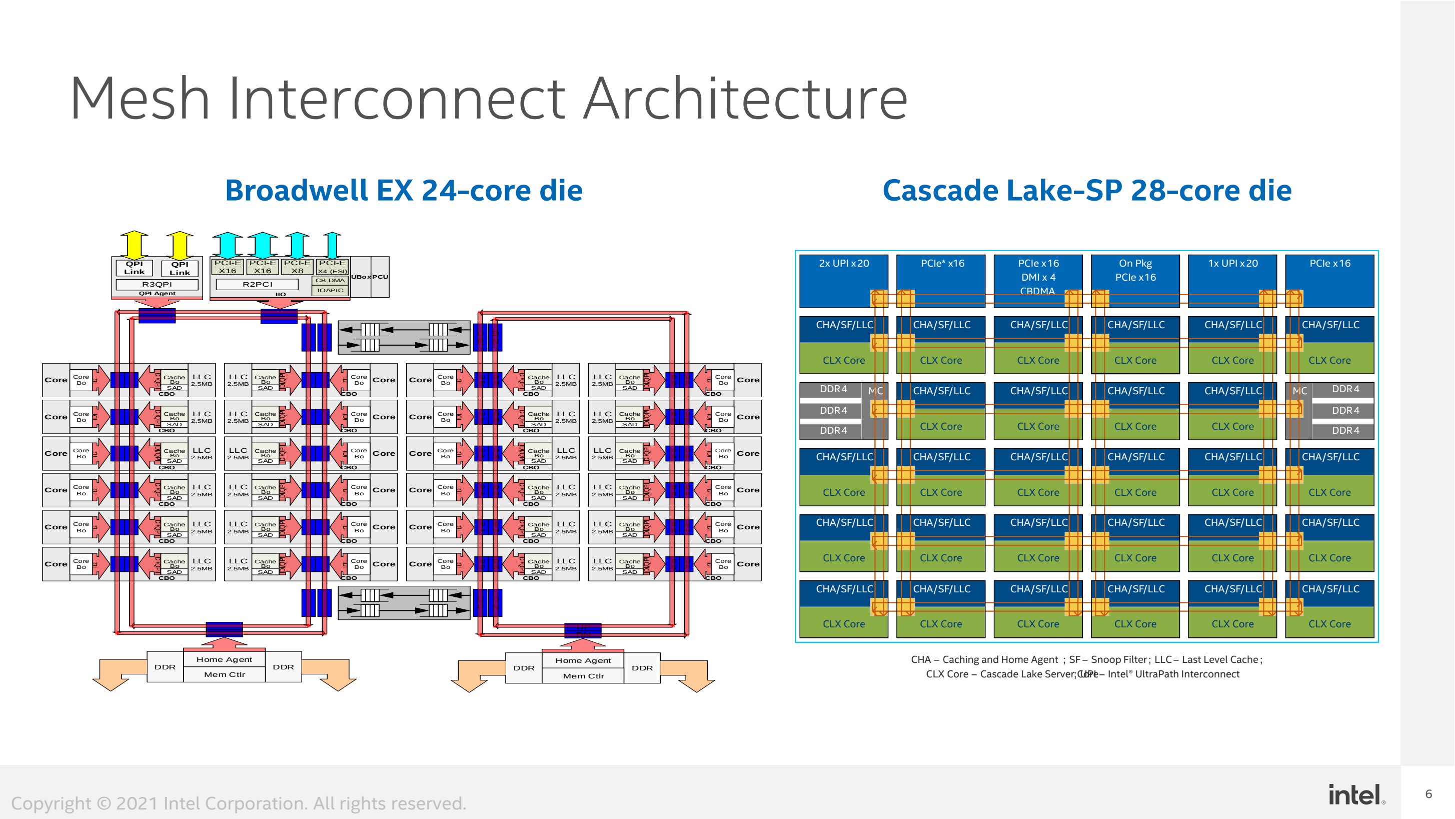
*Other names and brands may be claimed as the property of others.



Microarchitecture Enhancements

	Broadwell uArch	Cascade Lake uArch
Out-of-order Window	192	**224**
In-flight Loads + Stores	72 + 42	72 + **56**
Scheduler Entries	60	**97**
Registers – Integer + FP	168 + 168	**180** + 168
Allocation Queue	56	**64/thread**
L1D BW (B/Cyc) – Load + Store	64 + 32	**128 + 64**
L2 Unified TLB	4K+2M: 1024	4K+2M: **1536 1G: 16**

- Larger and improved branch predictor, higher throughput decoder, larger window to extract ILP
- Improved scheduler and execution engine, improved throughput and latency of divide/sqrt
- More load/store bandwidth, deeper load/store buffers, improved prefetcher
- Intel® AVX-512 with 2 FMAs per core, larger 1MB MLC

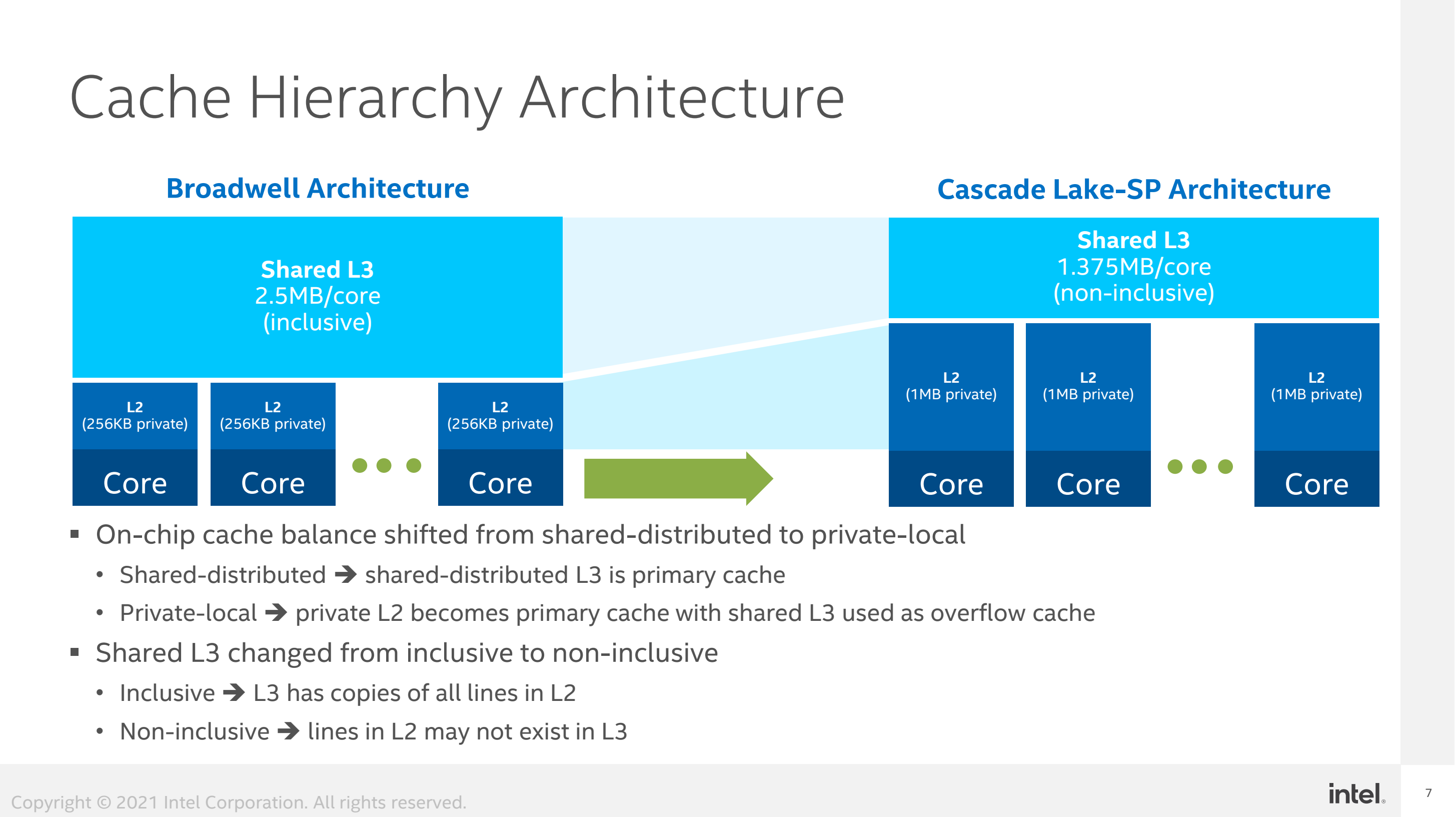


Mesh Interconnect Architecture

Broadwell EX 24-core die

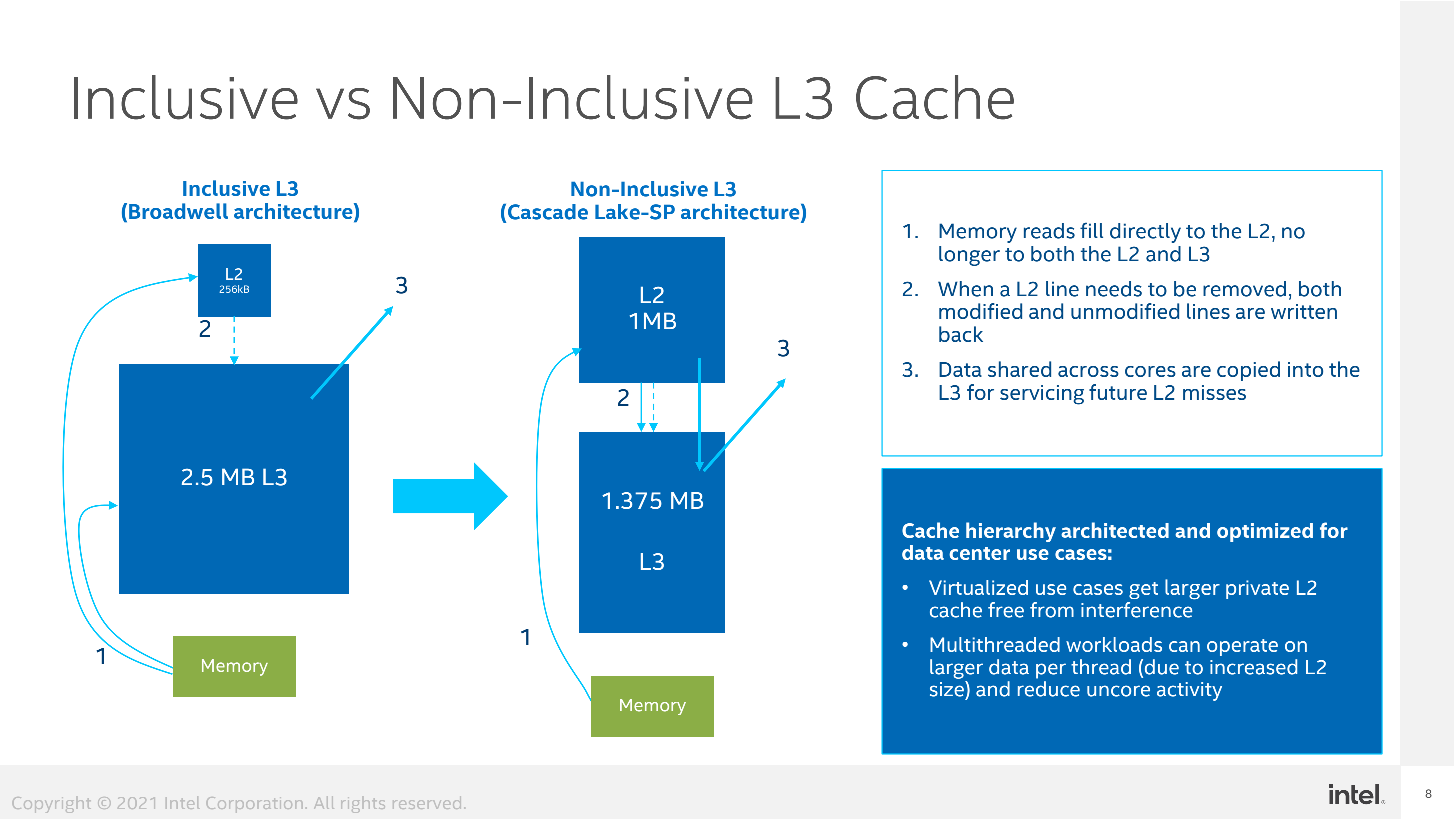
Cascade Lake-SP 28-core die

CHA – Caching and Home Agent ; SF – Snoop Filter ; LLC – Last Level Cache ;
CLX Core – Cascade Lake Server; Core UPI – Intel® UltraPath Interconnect



Cache Hierarchy Architecture

- On-chip cache balance shifted from shared-distributed to private-local
  - Shared-distributed ➔ shared-distributed L3 is primary cache
  - Private-local ➔ private L2 becomes primary cache with shared L3 used as overflow cache
- Shared L3 changed from inclusive to non-inclusive
  - Inclusive ➔ L3 has copies of all lines in L2
  - Non-inclusive ➔ lines in L2 may not exist in L3



Inclusive vs Non-Inclusive L3 Cache

1. Memory reads fill directly to the L2, no longer to both the L2 and L3
2. When a L2 line needs to be removed, both modified and unmodified lines are written back
3. Data shared across cores are copied into the L3 for servicing future L2 misses

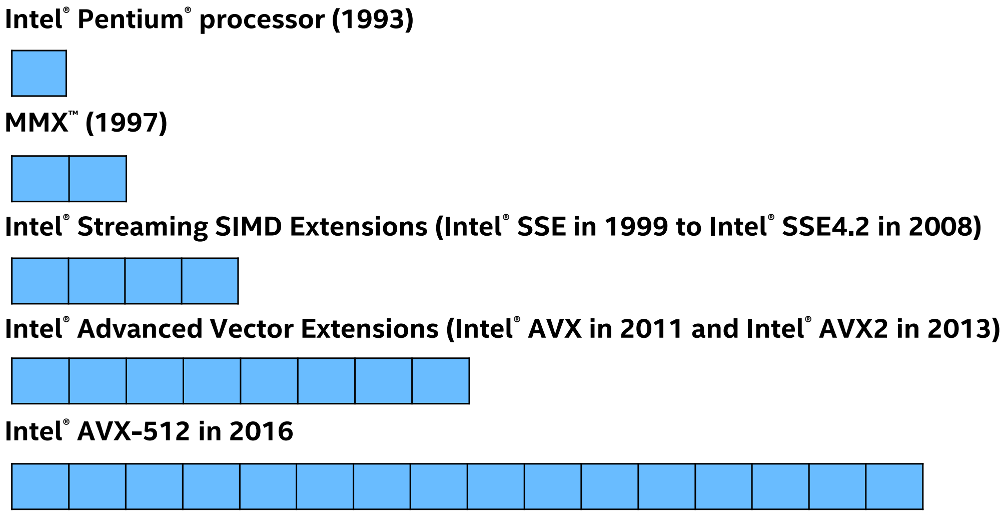
Cache hierarchy architected and optimized for data center use cases:

- Virtualized use cases get larger private L2 cache free from interference
- Multithreaded workloads can operate on larger data per thread (due to increased L2 size) and reduce uncore activity

Introduction to SIMD ISA for Intel® processors

History, features of Intel® AVX, Intel® AVX2 and Intel® AVX-512

History of SIMD ISA extensions*



Intel® Pentium® processor (1993)

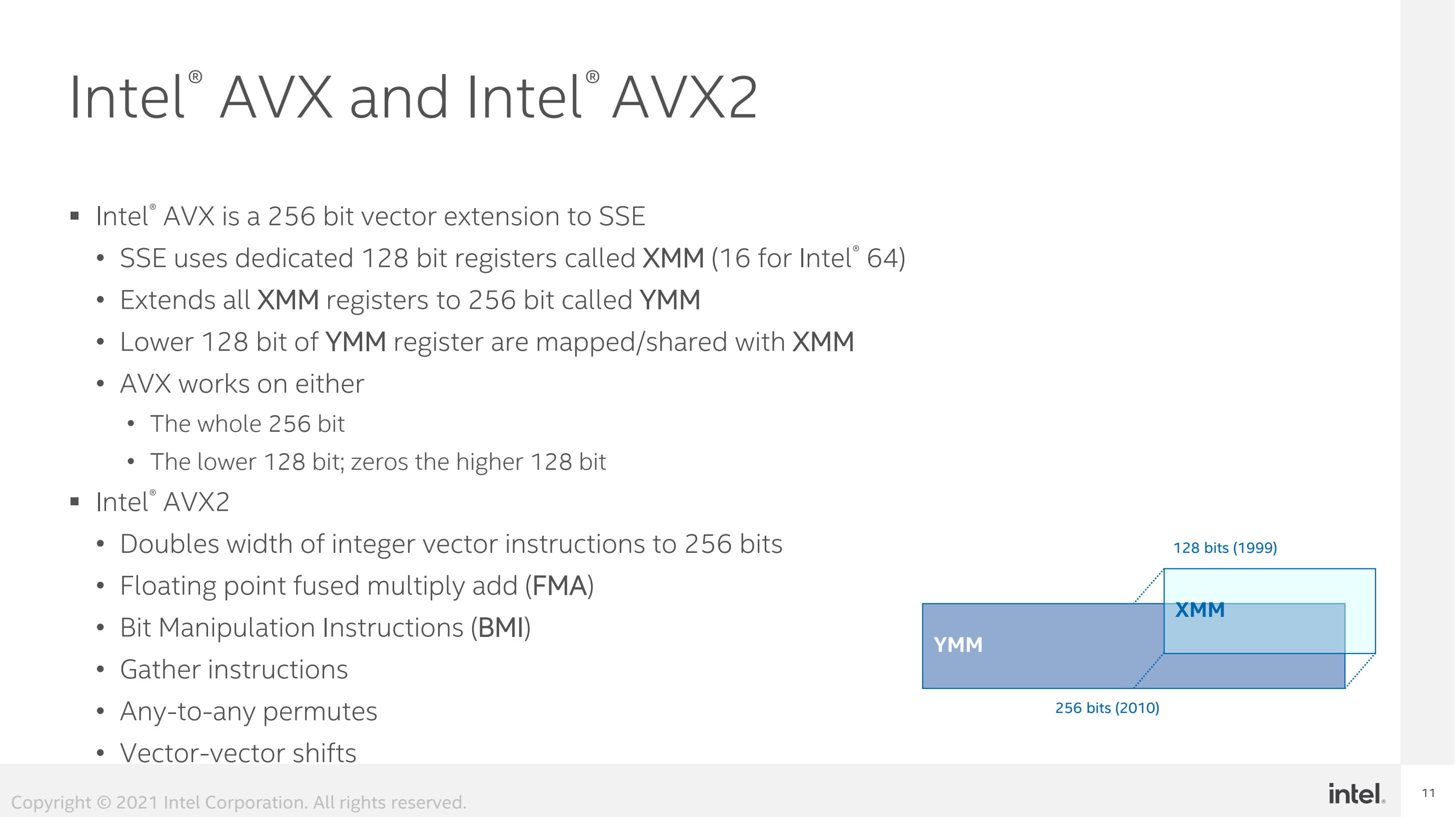
MMX™ (1997)

Intel® Streaming SIMD Extensions (Intel® SSE in 1999 to Intel® SSE4.2 in 2008)

Intel® Advanced Vector Extensions (Intel® AVX in 2011 and Intel® AVX2 in 2013)

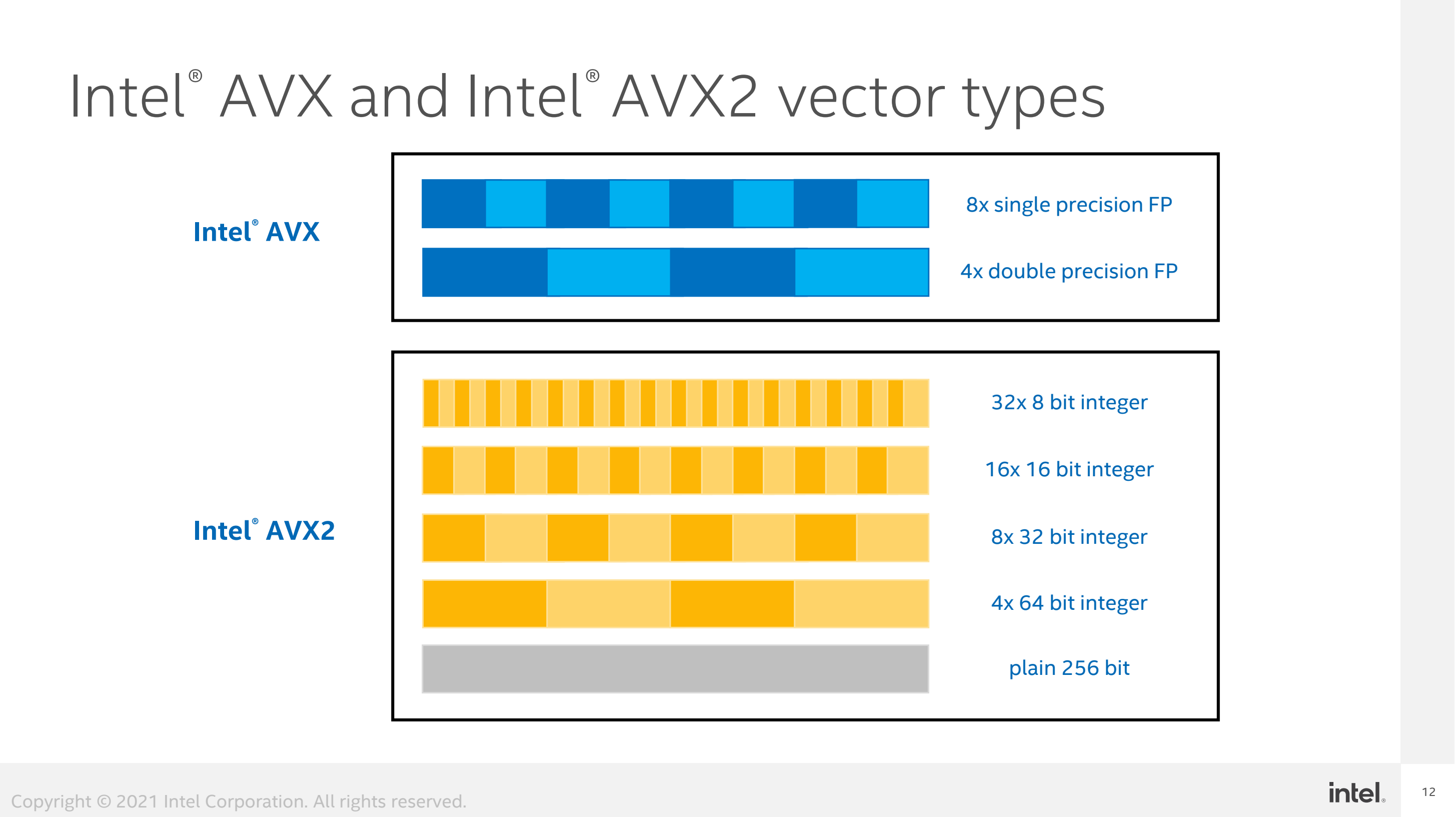
Intel® AVX-512 in 2016

* Illustrated with the number of 32-bit data elements that are processed by one “packed” instruction.



Intel® AVX and Intel® AVX2

- Intel® AVX is a 256 bit vector extension to SSE
  - SSE uses dedicated 128 bit registers called **XMM** (16 for Intel® 64)
  - Extends all **XMM** registers to 256 bit called **YMM**
  - Lower 128 bit of **YMM** register are mapped/shared with **XMM**
  - AVX works on either
    - The whole 256 bit
    - The lower 128 bit; zeros the higher 128 bit
- Intel® AVX2
  - Doubles width of integer vector instructions to 256 bits
  - Floating point fused multiply add (**FMA**)
  - Bit Manipulation Instructions (**BMI**)
  - Gather instructions
  - Any-to-any permutes
  - Vector-vector shifts



Intel® AVX and Intel® AVX2 vector types

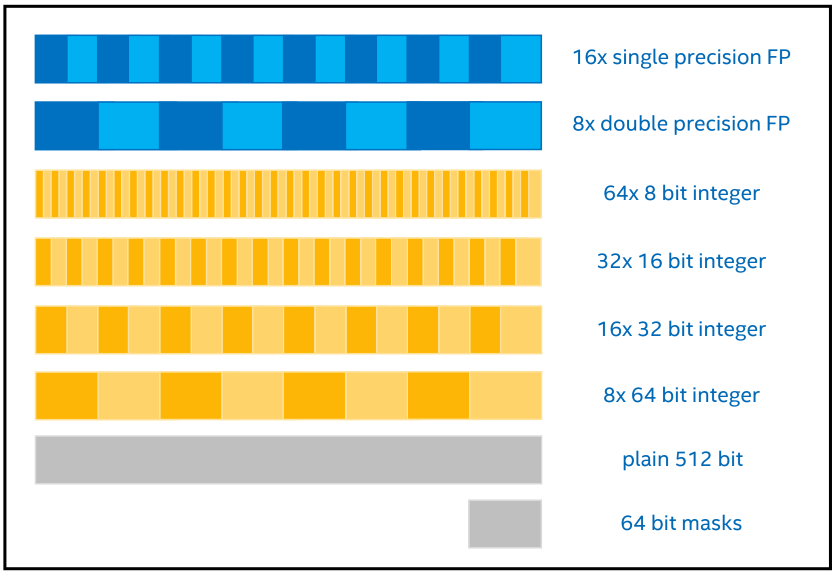
Intel® AVX-512

- 512-bit wide vectors
- 32 operand registers
- 8 64b mask registers
- Embedded broadcast
- Embedded rounding

Microarchitecture	Instruction Set	SP FLOPs / cycle	DP FLOPs / cycle
Intel® Xeon® Processor family	SSE (128b)	8	4
Intel® Xeon® E5 and E5v2 Processor families	Intel AVX (256b)	16	8
Intel® Xeon® E5v3 and E5v4 Processors families	Intel AVX2 & FMA (256b)	32	16
1st and 2nd generation Intel® Xeon® Scalable Processor Gold and Platinum families	AVX-512 & FMA (512b)	64	32

Intel® AVX-512 vector types

Intel® AVX-512



16x single precision FP

8x double precision FP

64x 8 bit integer

32x 16 bit integer

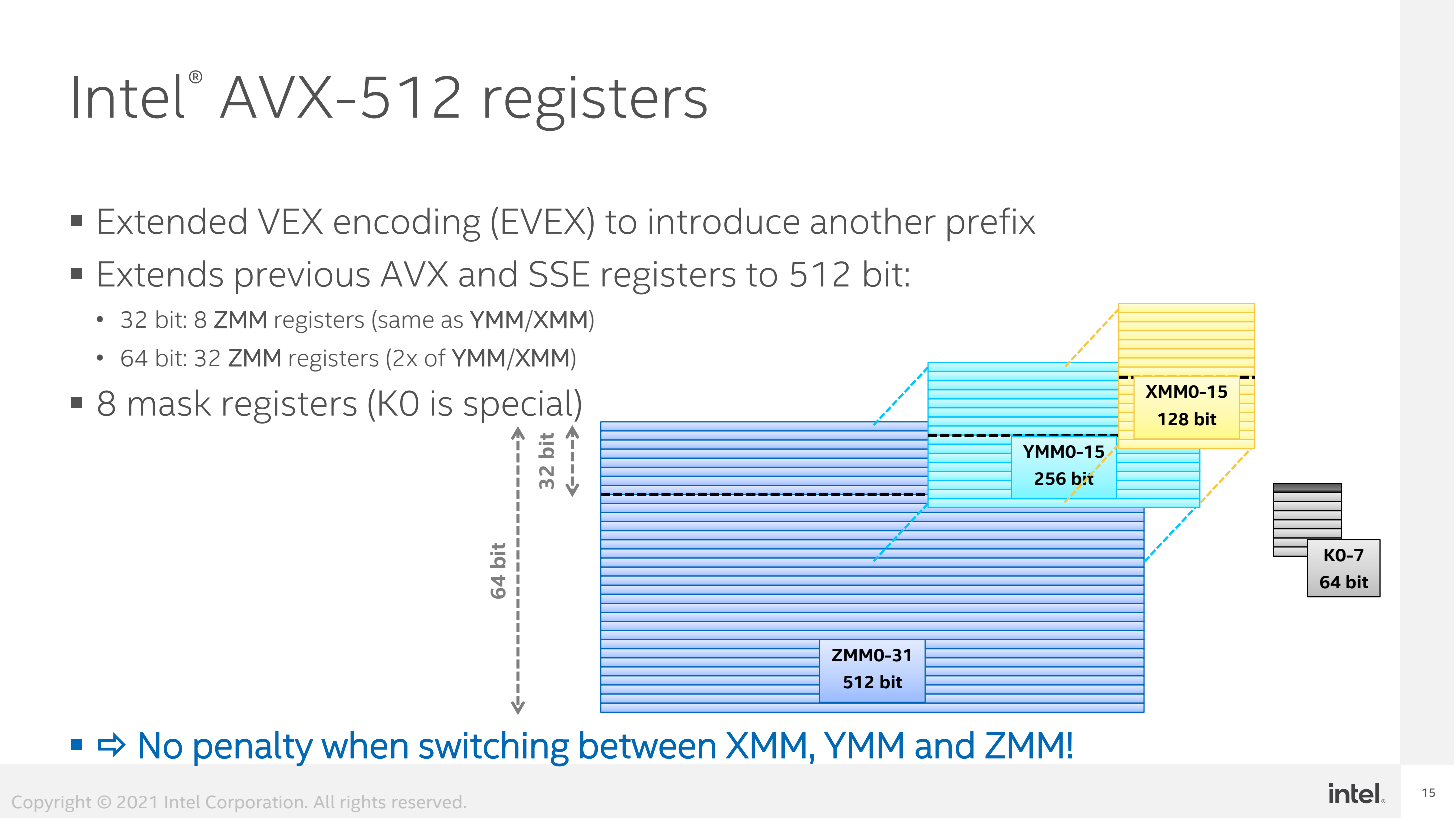
16x 32 bit integer

8x 64 bit integer

plain 512 bit

64 bit masks

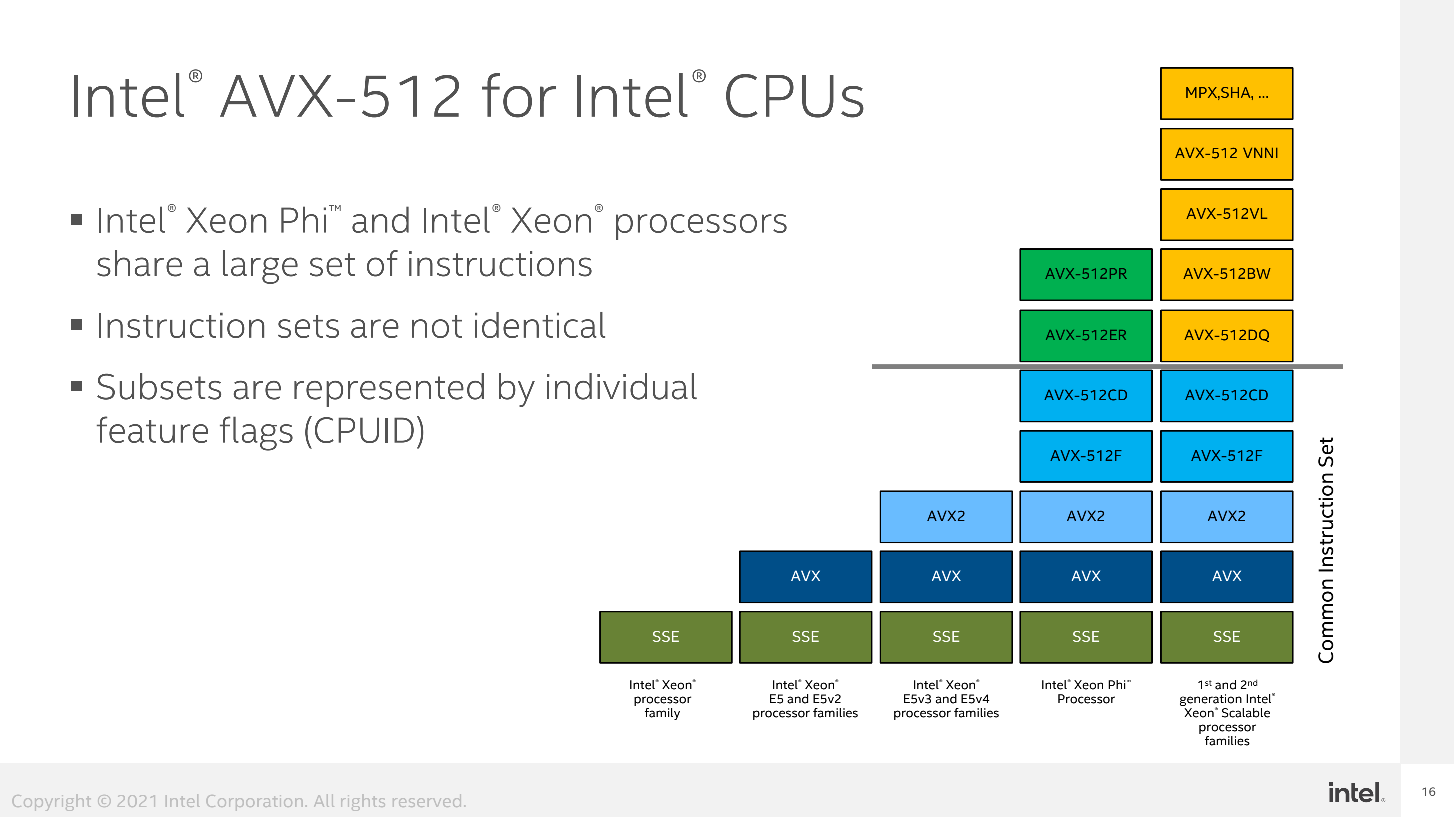
⇨ Includes AVX and AVX2



Intel® AVX-512 registers

- Extended VEX encoding (EVEX) to introduce another prefix
- Extends previous AVX and SSE registers to 512 bit:
  - 32 bit: 8 **ZMM** registers (same as **YMM**/**XMM**)
  - 64 bit: 32 **ZMM** registers (2x of **YMM**/**XMM**)
- 8 mask registers (K0 is special)

- ⇨ No penalty when switching between XMM, YMM and ZMM!



Intel® AVX-512 for Intel® CPUs

- Intel® Xeon Phi™ and Intel® Xeon® processors share a large set of instructions
- Instruction sets are not identical
- Subsets are represented by individual feature flags (CPUID)

Intel® Xeon® processor family	Intel® Xeon® E5 and E5v2 processor families	Intel® Xeon® E5v3 and E5v4 processor families	Intel® Xeon Phi™ Processor	1st and 2nd generation Intel® Xeon® Scalable processor families
				MPX,SHA, ...
				AVX-512 VNNI
				AVX-512VL
			AVX-512PR	AVX-512BW
			AVX-512ER	AVX-512DQ
			AVX-512CD	AVX-512CD
			AVX-512F	AVX-512F
		AVX2	AVX2	AVX2
	AVX	AVX	AVX	AVX
SSE	SSE	SSE	SSE	SSE

Intel® AVX-512

https://software.intel.com/en-us/blogs/additional-avx-512-instructions

Available in all products supporting Intel® AVX-512

- Intel® AVX-512 Foundation (AVX-512F)
  - Extension of AVX instruction sets including mask registers
- Intel® AVX-512 Conflict Detection (AVX-512CD)
  - Check identical values inside a vector (for 32 or 64 bit integers) to finding colliding indexes (32 or 64 bit) before a gather-operation-scatter sequence

Available on Intel® Xeon® processors

- Intel® AVX-512 Vector Length Extension (AVX-512VL)
  - Freely select the vector length (512 bit, 256 bit and 128 bit)
- Intel® AVX-512 Byte/Word (AVX-512BW) and Doubleword/Quadword (AVX-512DQ)
  - Two groups (8 and 16 bit integers and 32 and 64 bit integers/FP)

Available on Intel® Xeon Phi™ processors

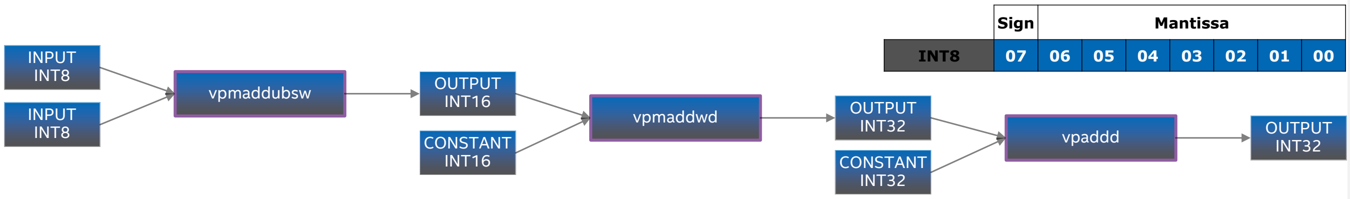
- Intel® AVX-512 Exponential & Reciprocal Instructions (AVX-512ER) and Intel® AVX-512 Prefetch Instructions (AVX-512PF)

Intel® AVX-512 VNNI

Available in selected 2nd Generation Intel® Xeon® Scalable Processors

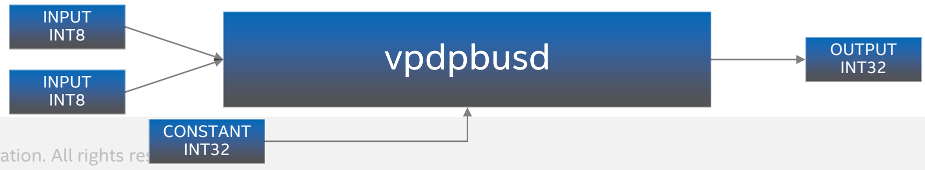
- Intel® AVX-512 Vector Neural Network Instructions (AVX-512 VNNI)
  - Adds **vpdpbusd/vpdpbusds** instructions for 8-bit inputs and **vpdpwssd/vpdpwssds** instructions for 16-bit inputs to accelerate DL convolutions

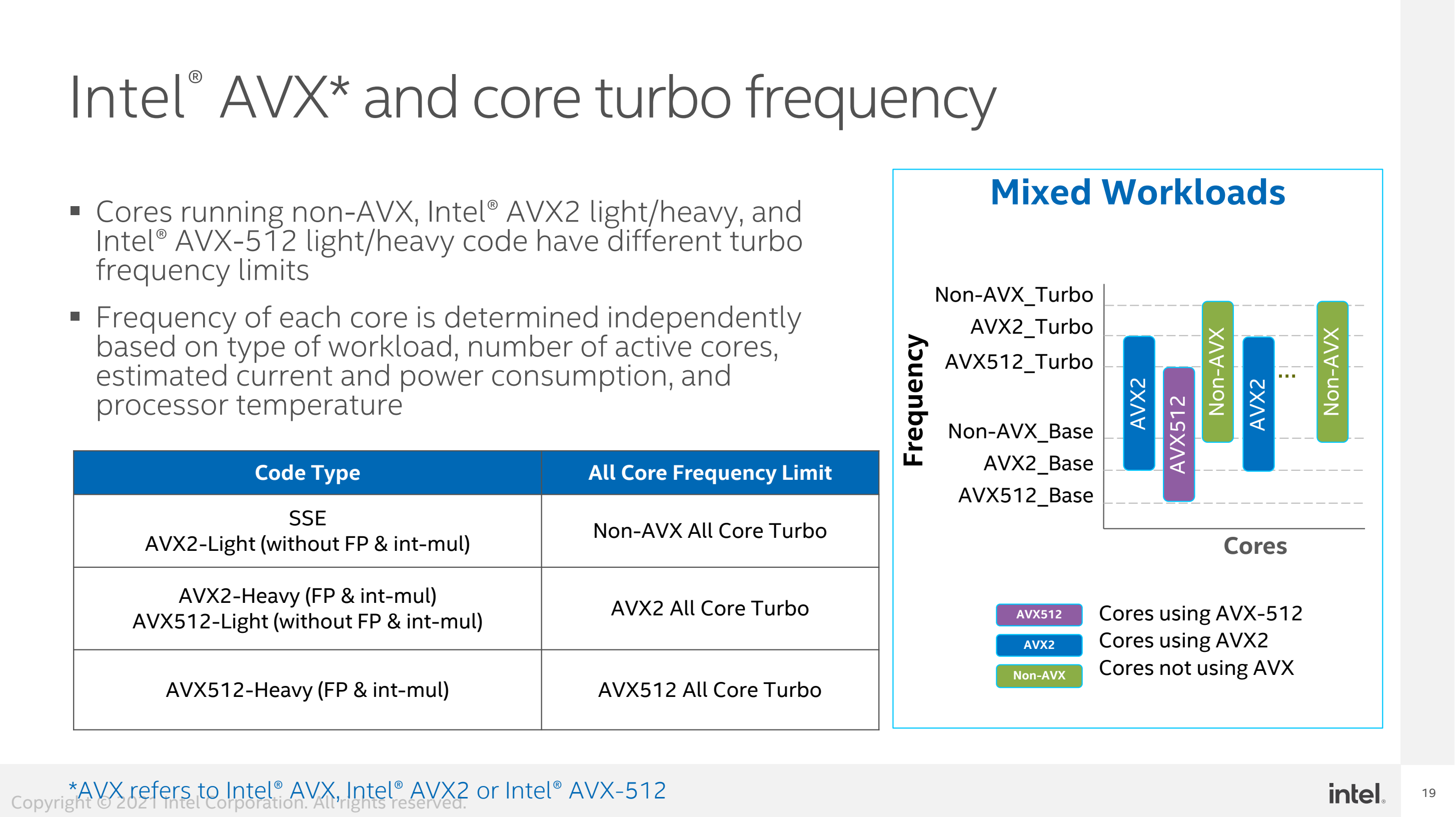
INT8 convolution with AVX-512: **vpmaddubsw, vpmaddwd, vpaddd**



	Sign	Mantissa						
INT8	07	06	05	04	03	02	01	00

INT8 convolution with AVX-512 VNNI: **vpdpbusd**





Intel® AVX* and core turbo frequency

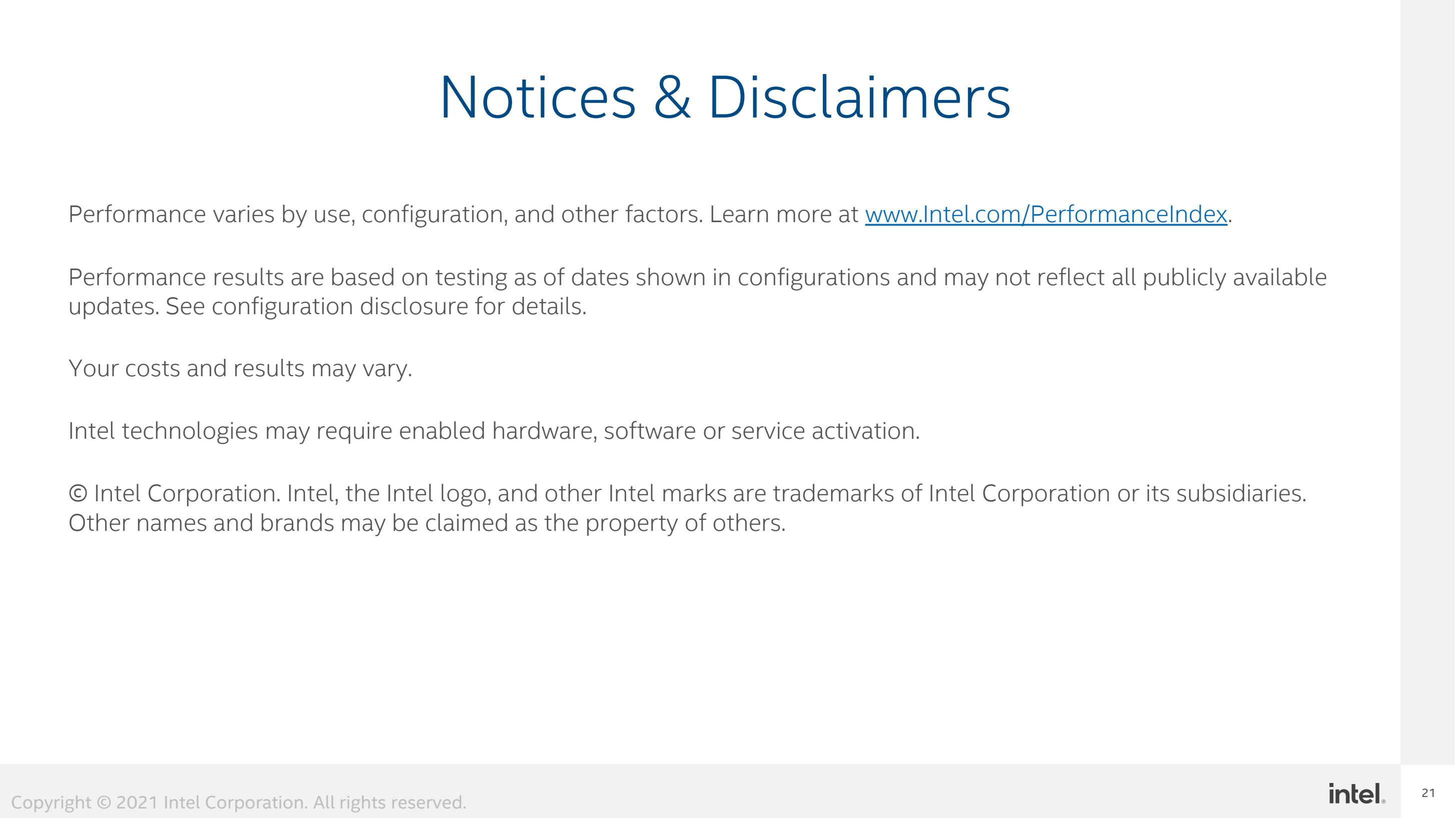
- Cores running non-AVX, Intel® AVX2 light/heavy, and Intel® AVX-512 light/heavy code have different turbo frequency limits
- Frequency of each core is determined independently based on type of workload, number of active cores, estimated current and power consumption, and processor temperature

Code Type	All Core Frequency Limit
SSE AVX2-Light (without FP & int-mul)	Non-AVX All Core Turbo
AVX2-Heavy (FP & int-mul) AVX512-Light (without FP & int-mul)	AVX2 All Core Turbo
AVX512-Heavy (FP & int-mul)	AVX512 All Core Turbo

Mixed Workloads

Cores using AVX-512
Cores using AVX2
Cores not using AVX

*AVX refers to Intel® AVX, Intel® AVX2 or Intel® AVX-512



Notices & Disclaimers

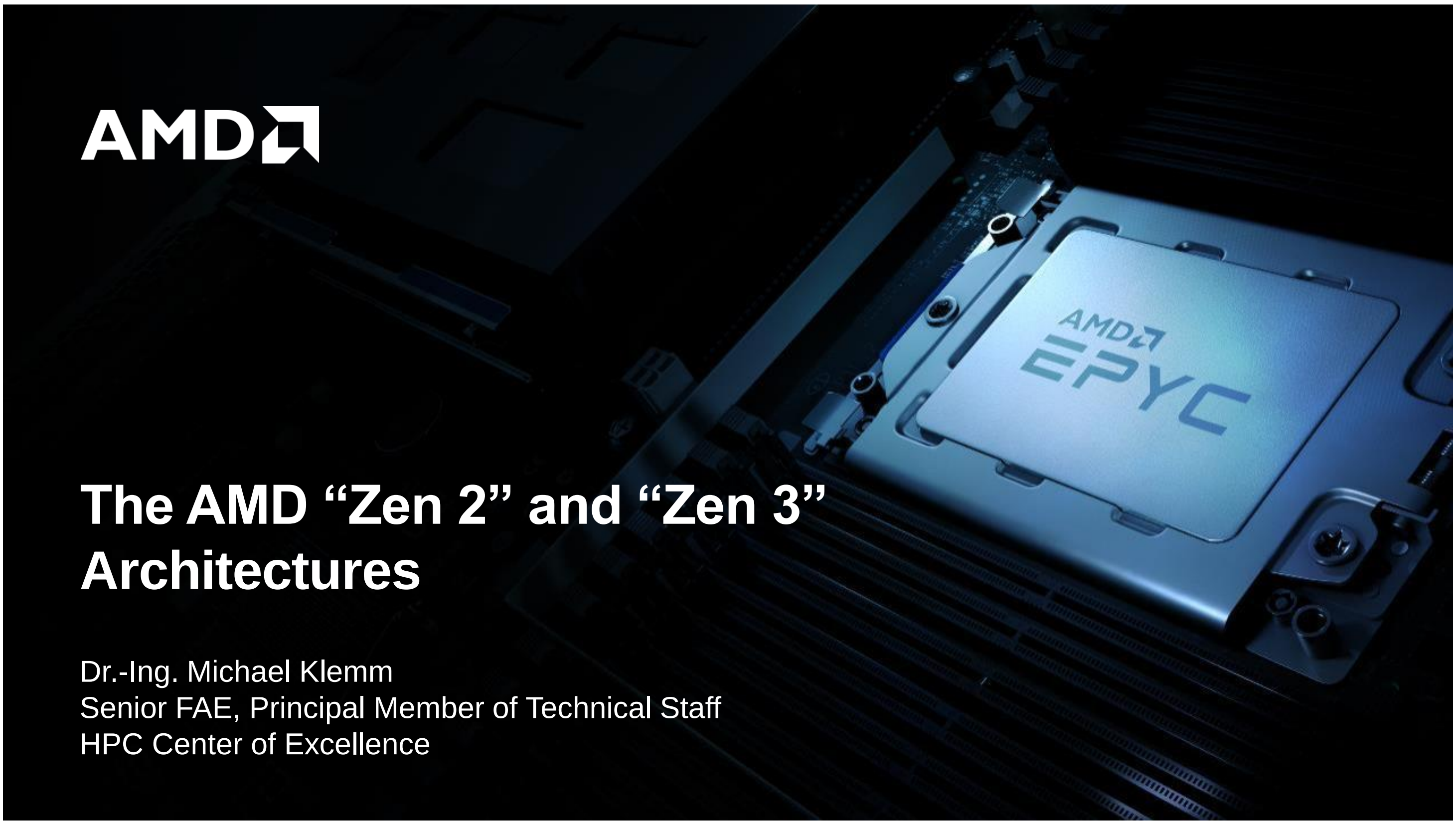
Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details.

Your costs and results may vary.

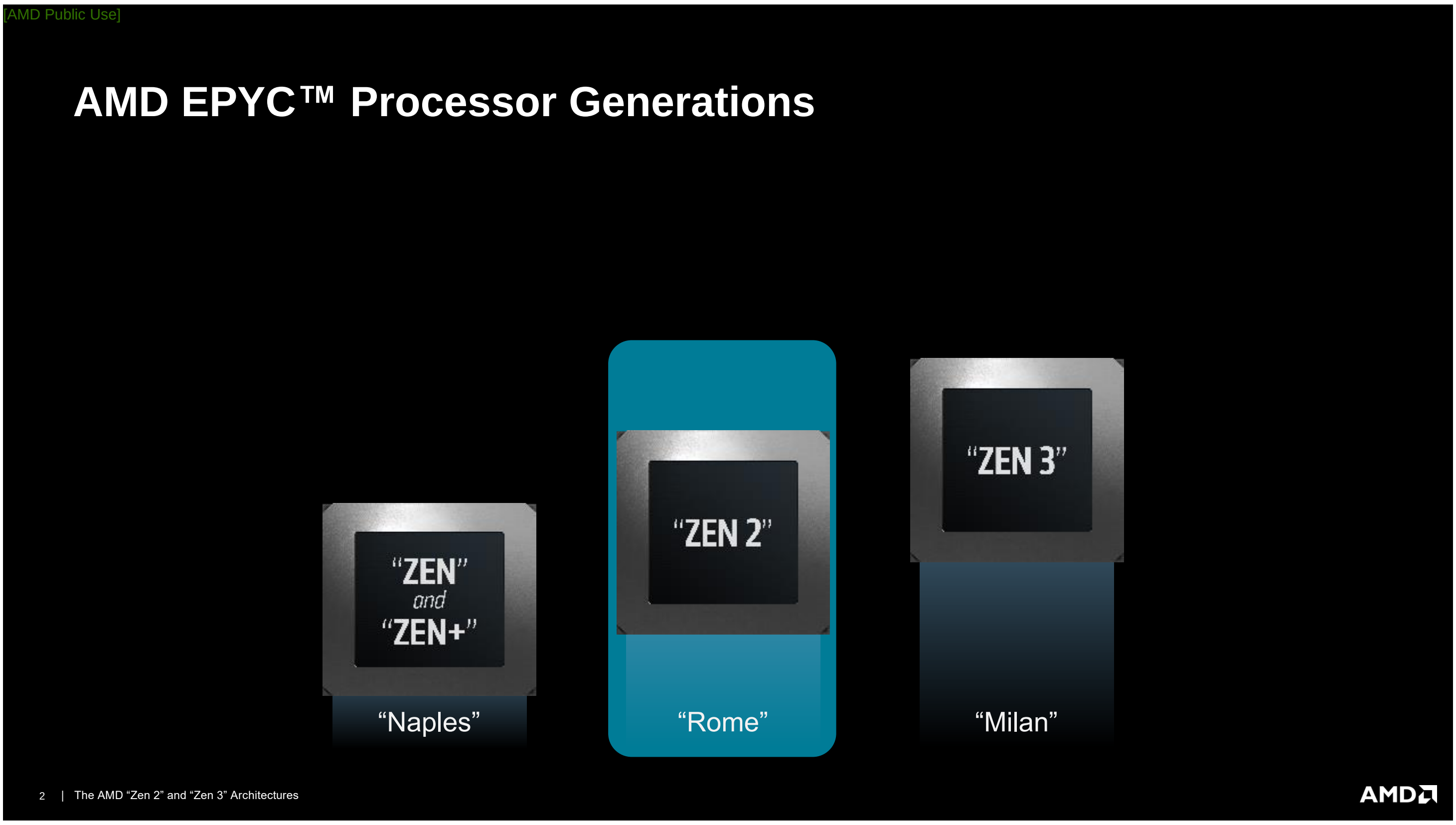
Intel technologies may require enabled hardware, software or service activation.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

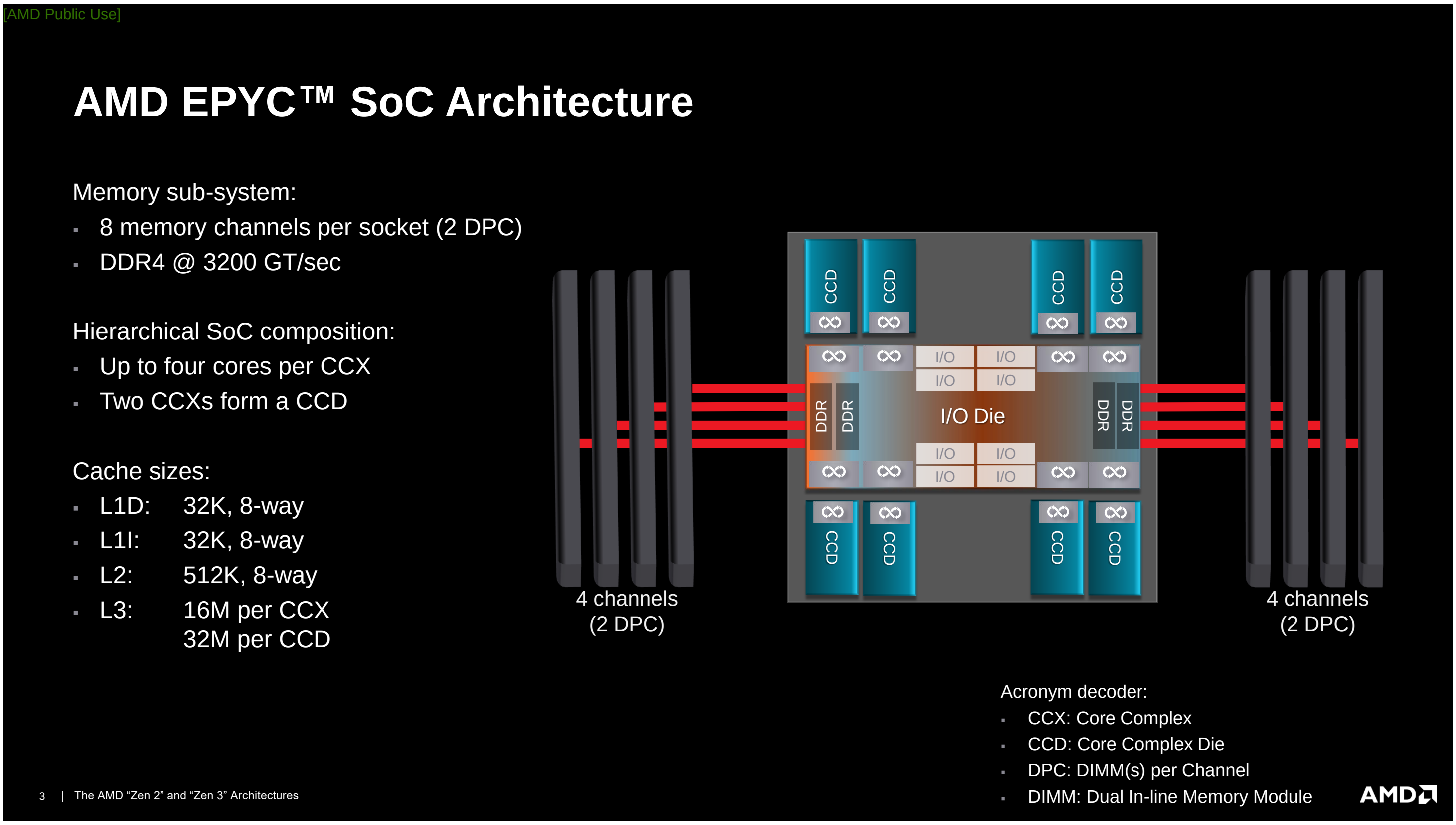


The AMD “Zen 2” and “Zen 3” Architectures

Dr.-Ing. Michael Klemm
Senior FAE, Principal Member of Technical Staff
HPC Center of Excellence



AMD EPYC™ Processor Generations



[AMD Public Use]

AMD EPYC™ SoC Architecture

Memory sub-system:

- 8 memory channels per socket (2 DPC)
- DDR4 @ 3200 GT/sec

Hierarchical SoC composition:

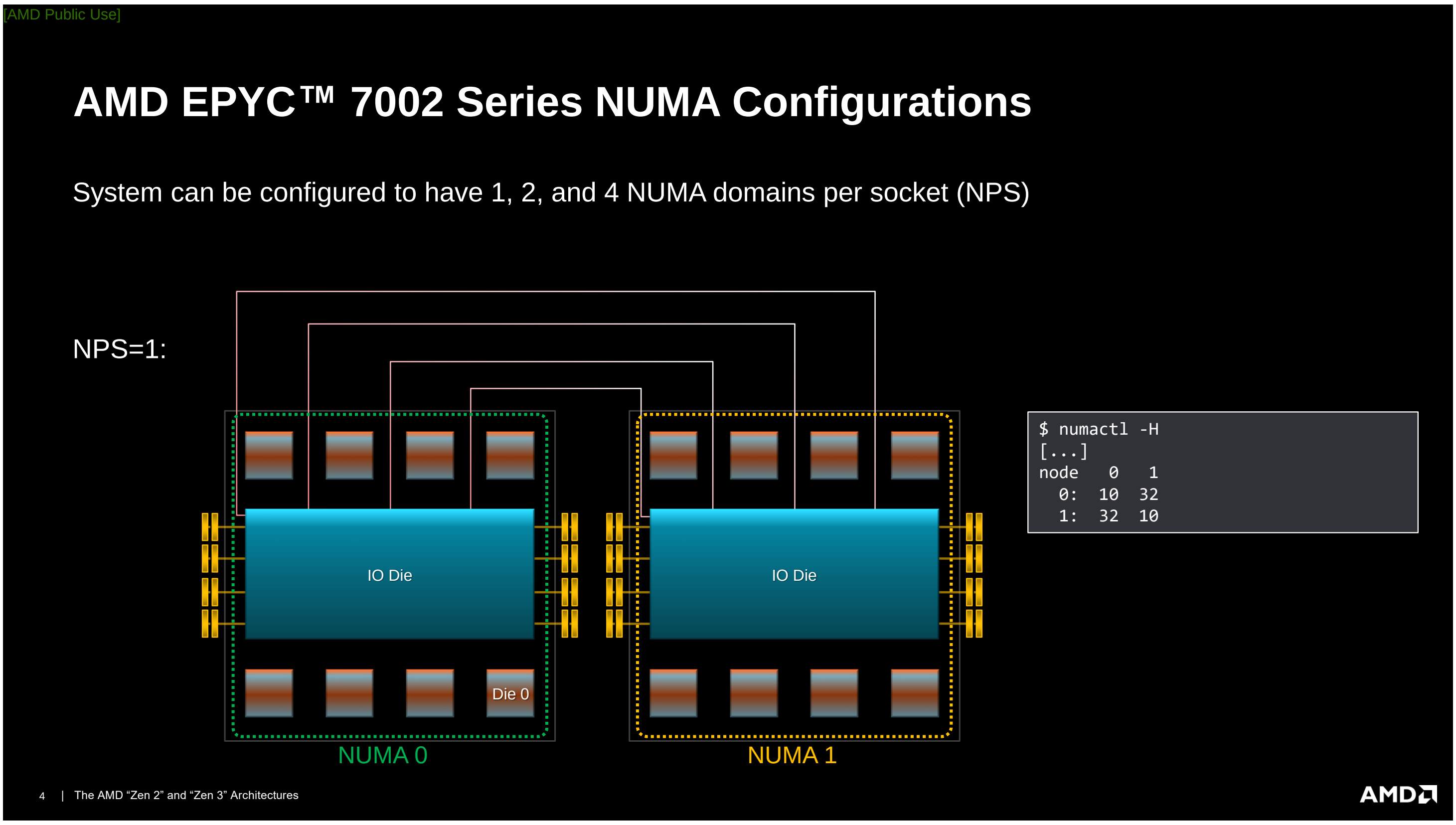
- Up to four cores per CCX
- Two CCXs form a CCD

Cache sizes:

- L1D: 32K, 8-way
- L1I: 32K, 8-way
- L2: 512K, 8-way
- L3: 16M per CCX
  32M per CCD

Acronym decoder:

- CCX: Core Complex
- CCD: Core Complex Die
- DPC: DIMM(s) per Channel
- DIMM: Dual In-line Memory Module



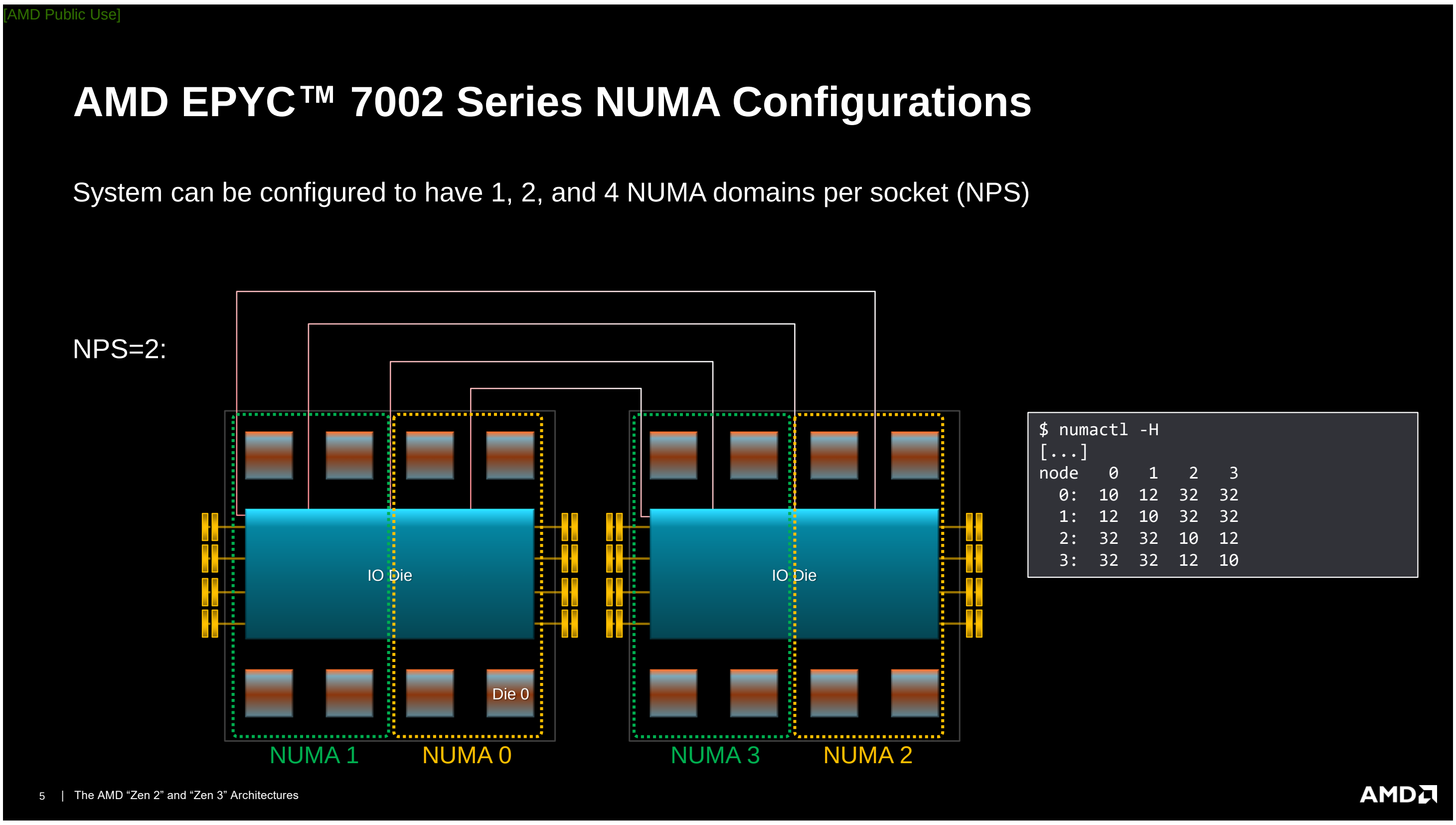
[AMD Public Use]

AMD EPYC™ 7002 Series NUMA Configurations

System can be configured to have 1, 2, and 4 NUMA domains per socket (NPS)

NPS=1:

```
$ numactl -H
[...]
node   0   1
  0:  10  32
  1:  32  10
```



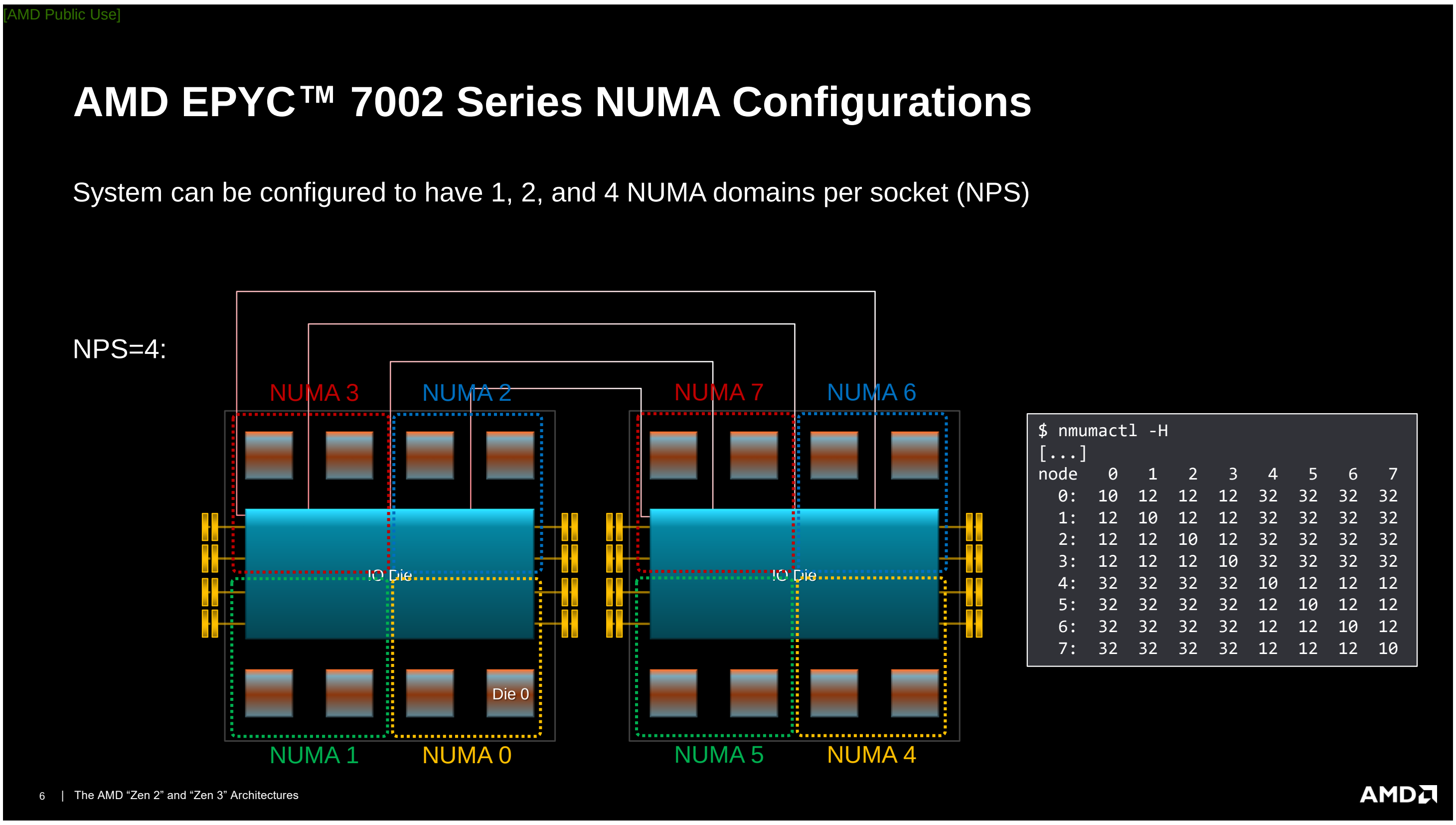
[AMD Public Use]

AMD EPYC™ 7002 Series NUMA Configurations

System can be configured to have 1, 2, and 4 NUMA domains per socket (NPS)

NPS=2:

```
$ numactl -H
[...]
node   0   1   2   3
  0:  10  12  32  32
  1:  12  10  32  32
  2:  32  32  10  12
  3:  32  32  12  10
```



[AMD Public Use]

AMD EPYC™ 7002 Series NUMA Configurations

System can be configured to have 1, 2, and 4 NUMA domains per socket (NPS)

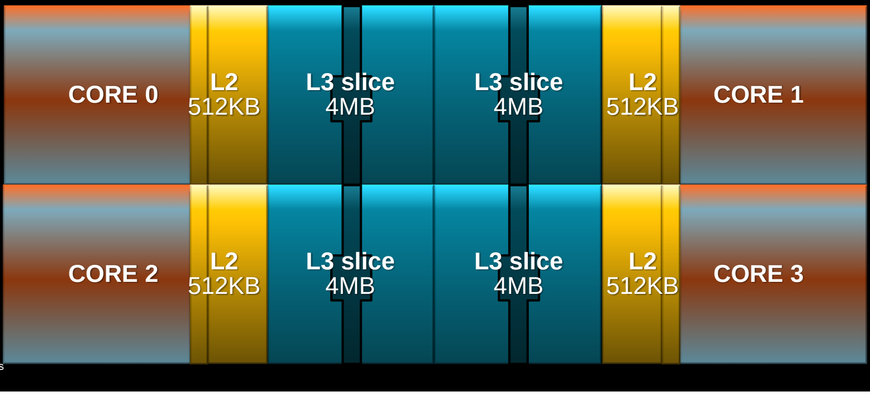
NPS=4:

```
$ nmumactl -H
[...]
node   0   1   2   3   4   5   6   7
  0:  10  12  12  12  32  32  32  32
  1:  12  10  12  12  32  32  32  32
  2:  12  12  10  12  32  32  32  32
  3:  12  12  12  10  32  32  32  32
  4:  32  32  32  32  10  12  12  12
  5:  32  32  32  32  12  10  12  12
  6:  32  32  32  32  12  12  10  12
  7:  32  32  32  32  12  12  12  10
```

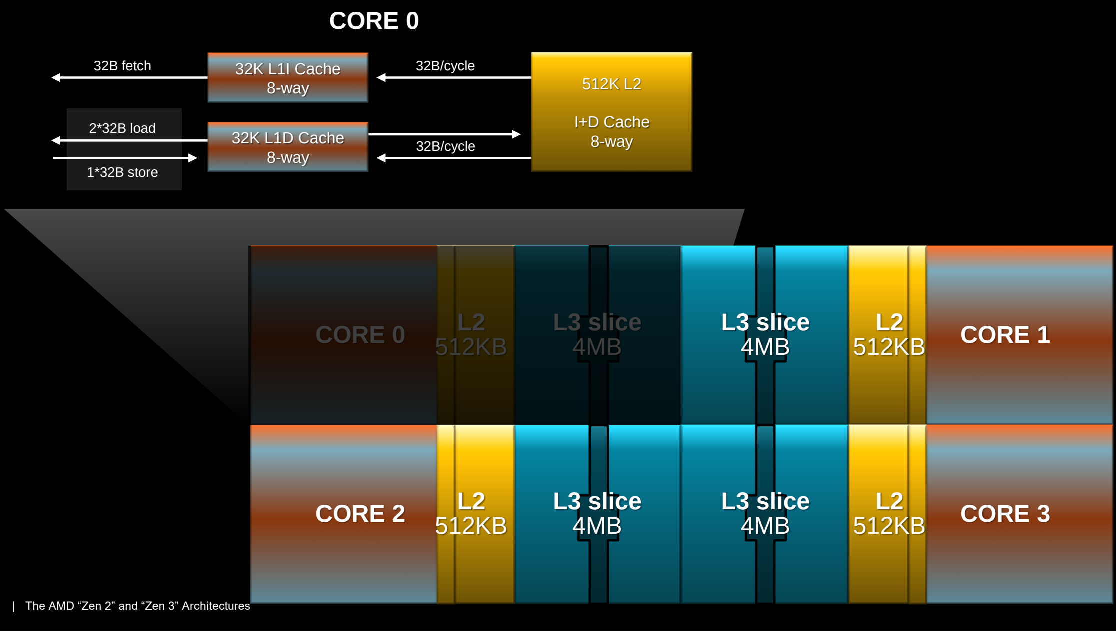
Cache Hierarchy and Core Complex (CCX)

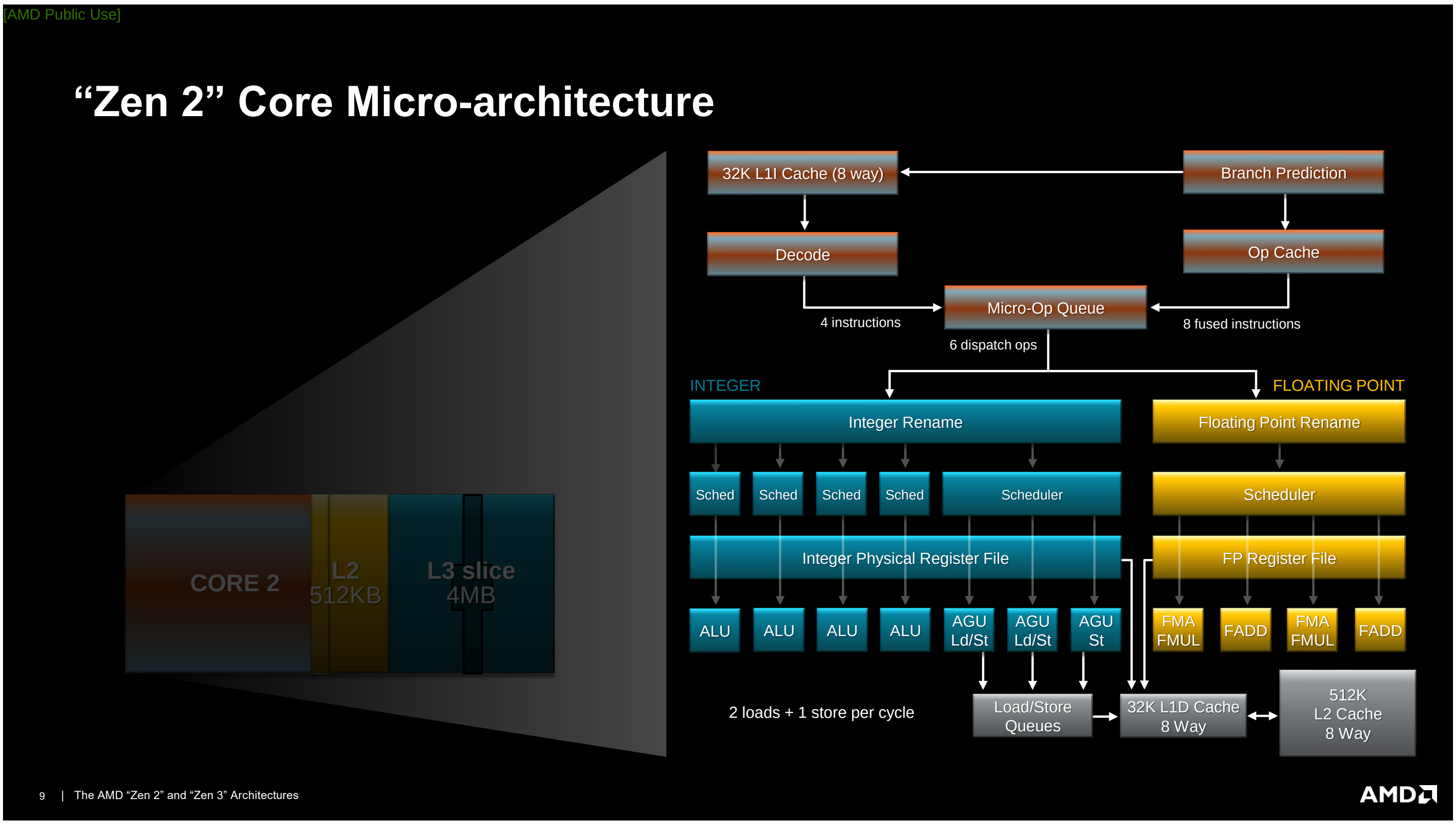
Structure of the CCX consists of

- Four cores with two-way SMT and
  - L1D and L1I cache in the core (32K each, 8-way associative, 64 sets)
  - Core-local L2 cache (512KB, 8-way associative, 1,024 sets)
- Four L3 slides of 4MB that form the 16MB L3 cache
  - 16-way associative, 16,384 sets
  - Used as a victim cache to receive data evicted from the L2 cache



Cache Hierarchy and Core Complex

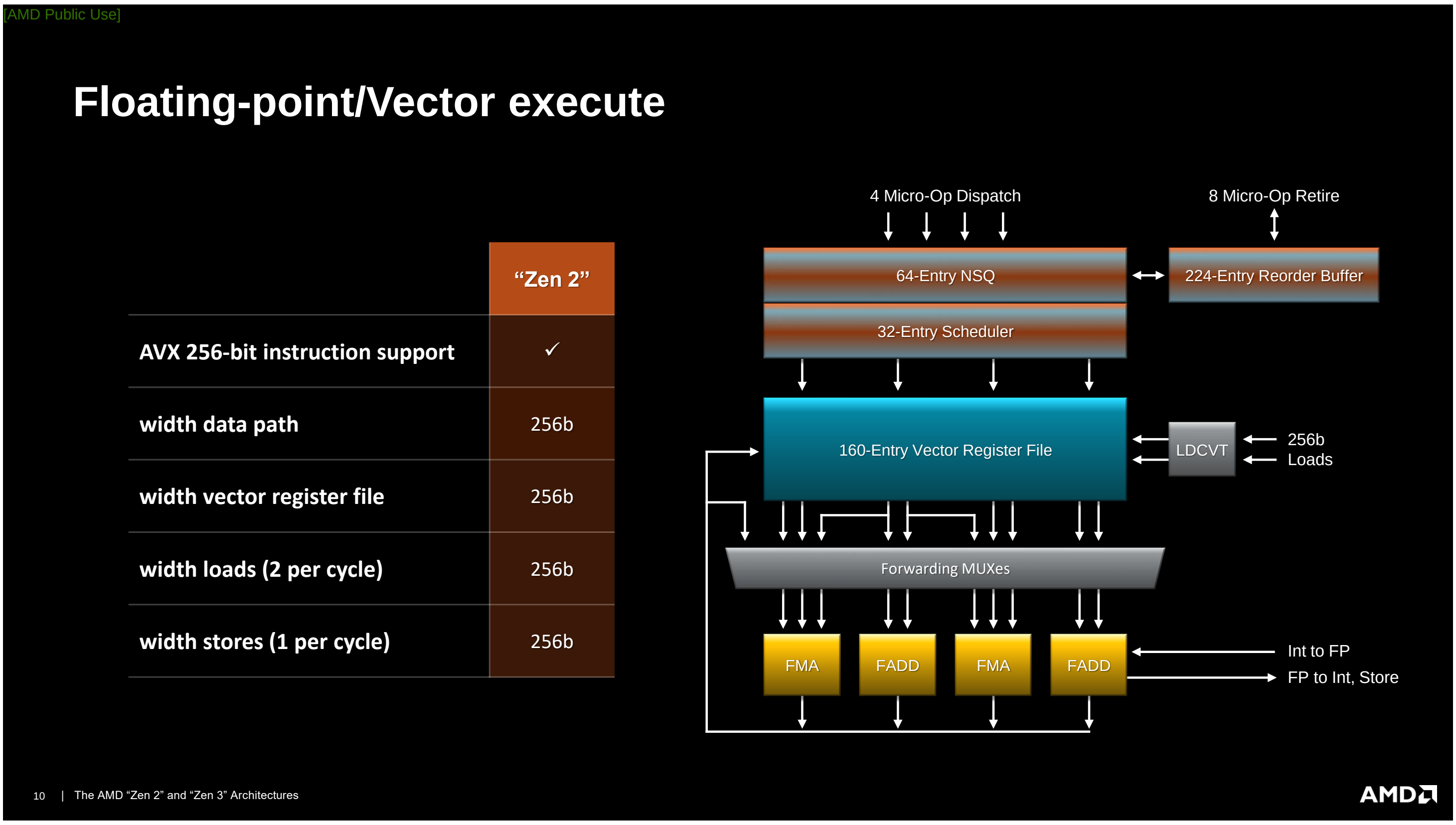




[AMD Public Use]

“Zen 2” Core Micro-architecture

AMD



[AMD Public Use]

Floating-point/Vector execute

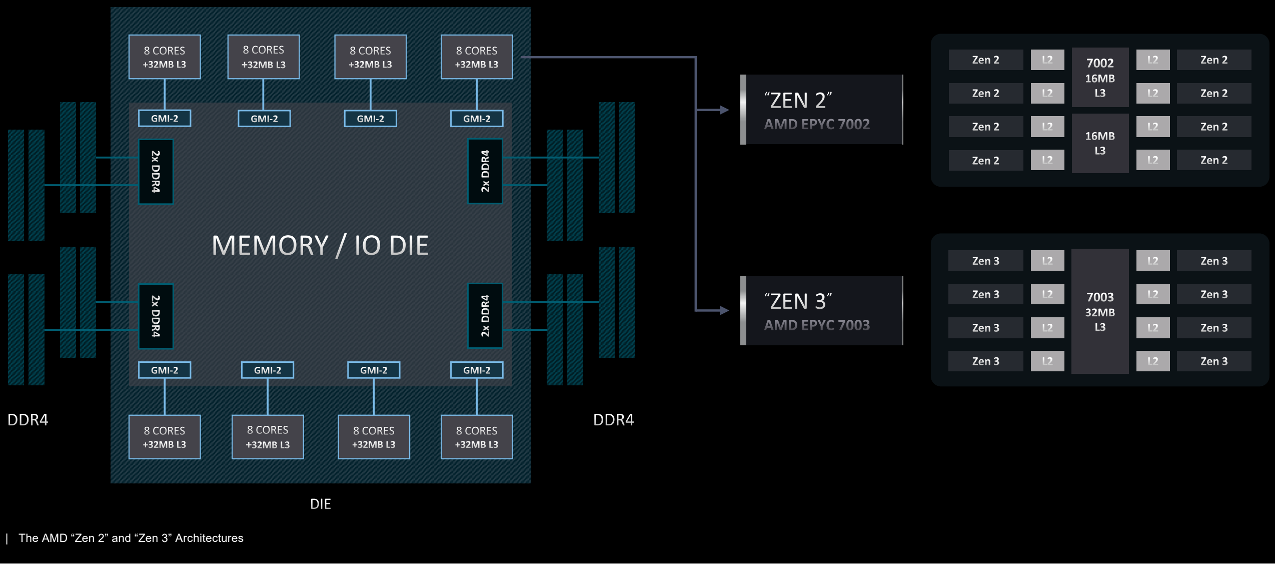
	“Zen 2”
AVX 256-bit instruction support	✓
width data path	256b
width vector register file	256b
width loads (2 per cycle)	256b
width stores (1 per cycle)	256b

AMD

AMD EPYC™ Processor Generations

“Naples” “Rome” “Milan”

AMD EPYC™ 7003 Series – Soc Architecture



8 CORES +32MB L3 (×8)

GMI-2 (×8)

2x DDR4 (×4)

MEMORY / IO DIE

DDR4

DDR4

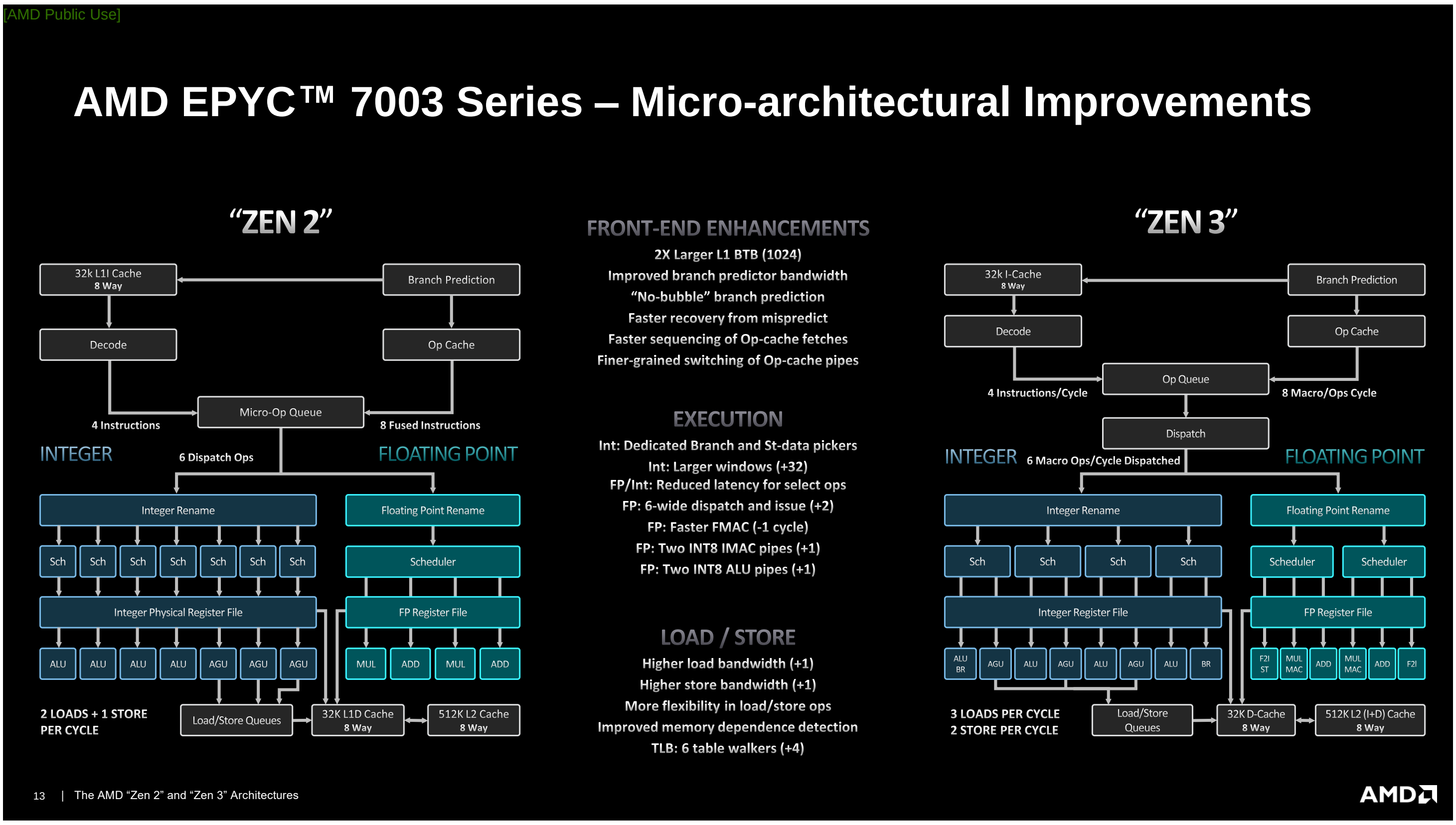
DIE

“ZEN 2” AMD EPYC 7002

“ZEN 3” AMD EPYC 7003

Zen 2	L2	7002 16MB L3	L2	Zen 2
Zen 2	L2		L2	Zen 2
Zen 2	L2	16MB L3	L2	Zen 2
Zen 2	L2		L2	Zen 2

Zen 3	L2	7003 32MB L3	L2	Zen 3
Zen 3	L2		L2	Zen 3
Zen 3	L2		L2	Zen 3
Zen 3	L2		L2	Zen 3


[AMD Public Use]

AMD EPYC™ 7003 Series – Micro-architectural Improvements

FRONT-END ENHANCEMENTS

2X Larger L1 BTB (1024)
Improved branch predictor bandwidth
"No-bubble" branch prediction
Faster recovery from mispredict
Faster sequencing of Op-cache fetches
Finer-grained switching of Op-cache pipes

EXECUTION

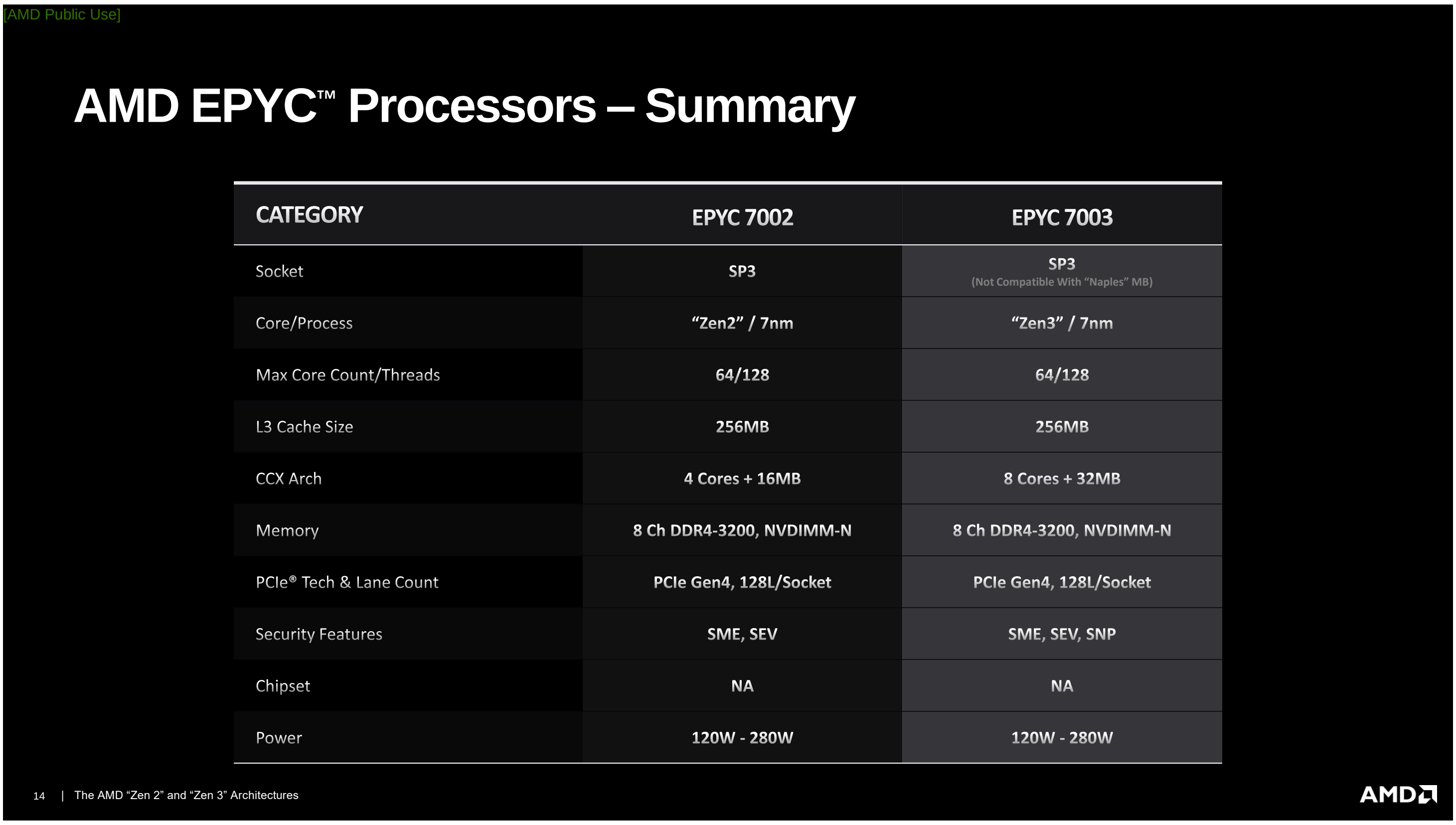
Int: Dedicated Branch and St-data pickers
Int: Larger windows (+32)
FP/Int: Reduced latency for select ops
FP: 6-wide dispatch and issue (+2)
FP: Faster FMAC (-1 cycle)
FP: Two INT8 IMAC pipes (+1)
FP: Two INT8 ALU pipes (+1)

LOAD / STORE

Higher load bandwidth (+1)
Higher store bandwidth (+1)
More flexibility in load/store ops
Improved memory dependence detection
TLB: 6 table walkers (+4)

13 | The AMD "Zen 2" and "Zen 3" Architectures

AMD

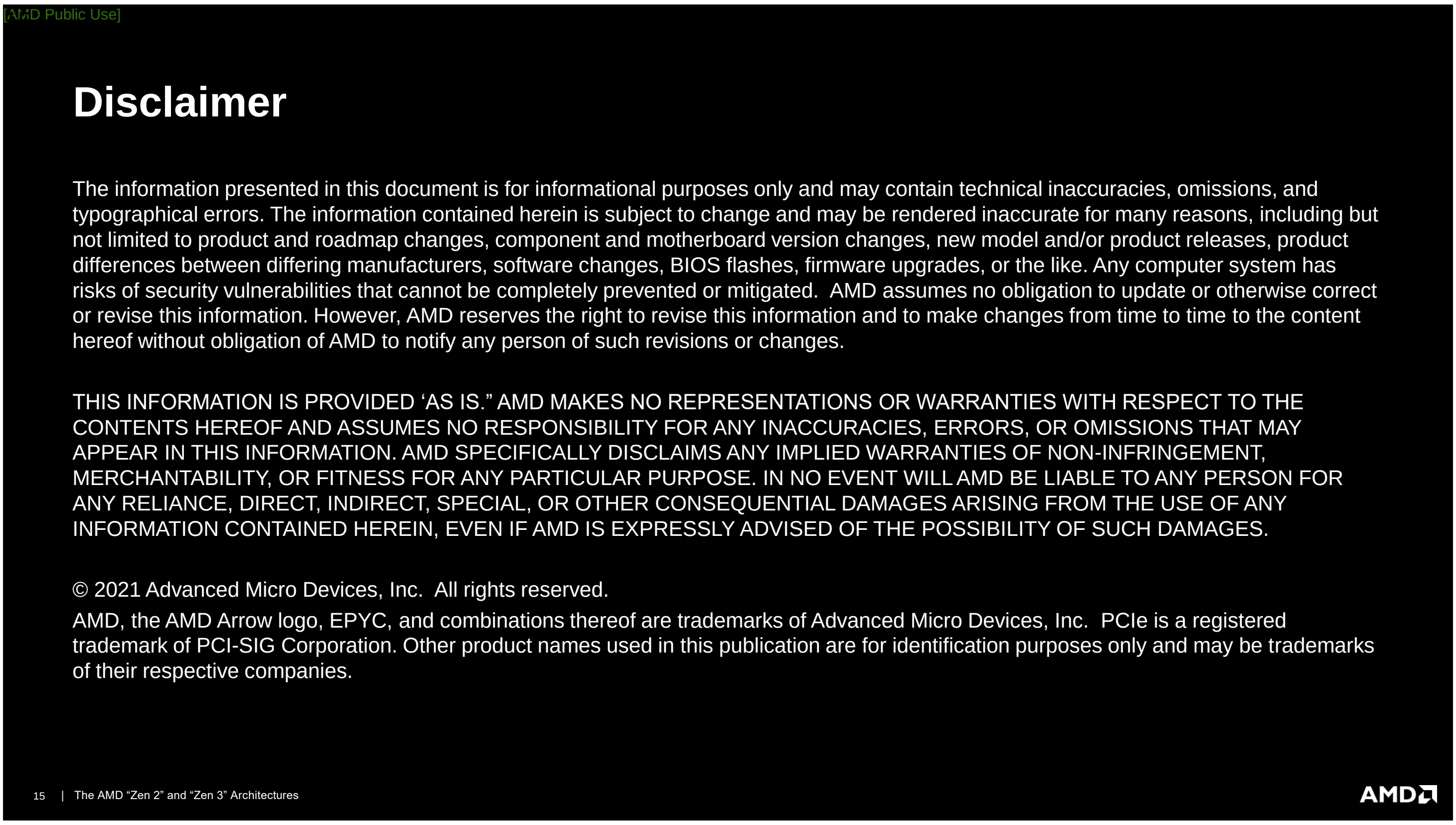

[AMD Public Use]

AMD EPYC™ Processors – Summary

CATEGORY	EPYC 7002	EPYC 7003
Socket	SP3	SP3 (Not Compatible With "Naples" MB)
Core/Process	"Zen2" / 7nm	"Zen3" / 7nm
Max Core Count/Threads	64/128	64/128
L3 Cache Size	256MB	256MB
CCX Arch	4 Cores + 16MB	8 Cores + 32MB
Memory	8 Ch DDR4-3200, NVDIMM-N	8 Ch DDR4-3200, NVDIMM-N
PCIe® Tech & Lane Count	PCIe Gen4, 128L/Socket	PCIe Gen4, 128L/Socket
Security Features	SME, SEV	SME, SEV, SNP
Chipset	NA	NA
Power	120W - 280W	120W - 280W

14 | The AMD "Zen 2" and "Zen 3" Architectures

AMD



Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.” AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD Arrow logo, EPYC, and combinations thereof are trademarks of Advanced Micro Devices, Inc. PCIe is a registered trademark of PCI-SIG Corporation. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

[ONLINE] Node Level Performance Optimization @ CSC, 18-20.5.2021

Performance analysis with Intel® tools

Dr. Mikko Byckling, IAGS DEE XCSS

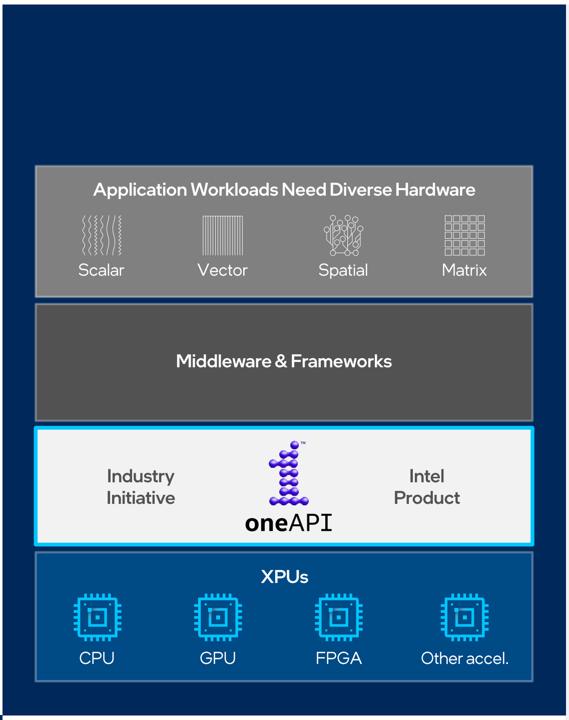
Contents

- Intel® oneAPI performance analysis tools overview
- Application Performance Snapshot
- Introduction to Intel® VTune™ Profiler
  - Features and analysis types
  - Graphical User Interface (GUI)
  - Command Line Interface (CLI)
- Intel® VTune™ Profiler HPC workflow
- Summary

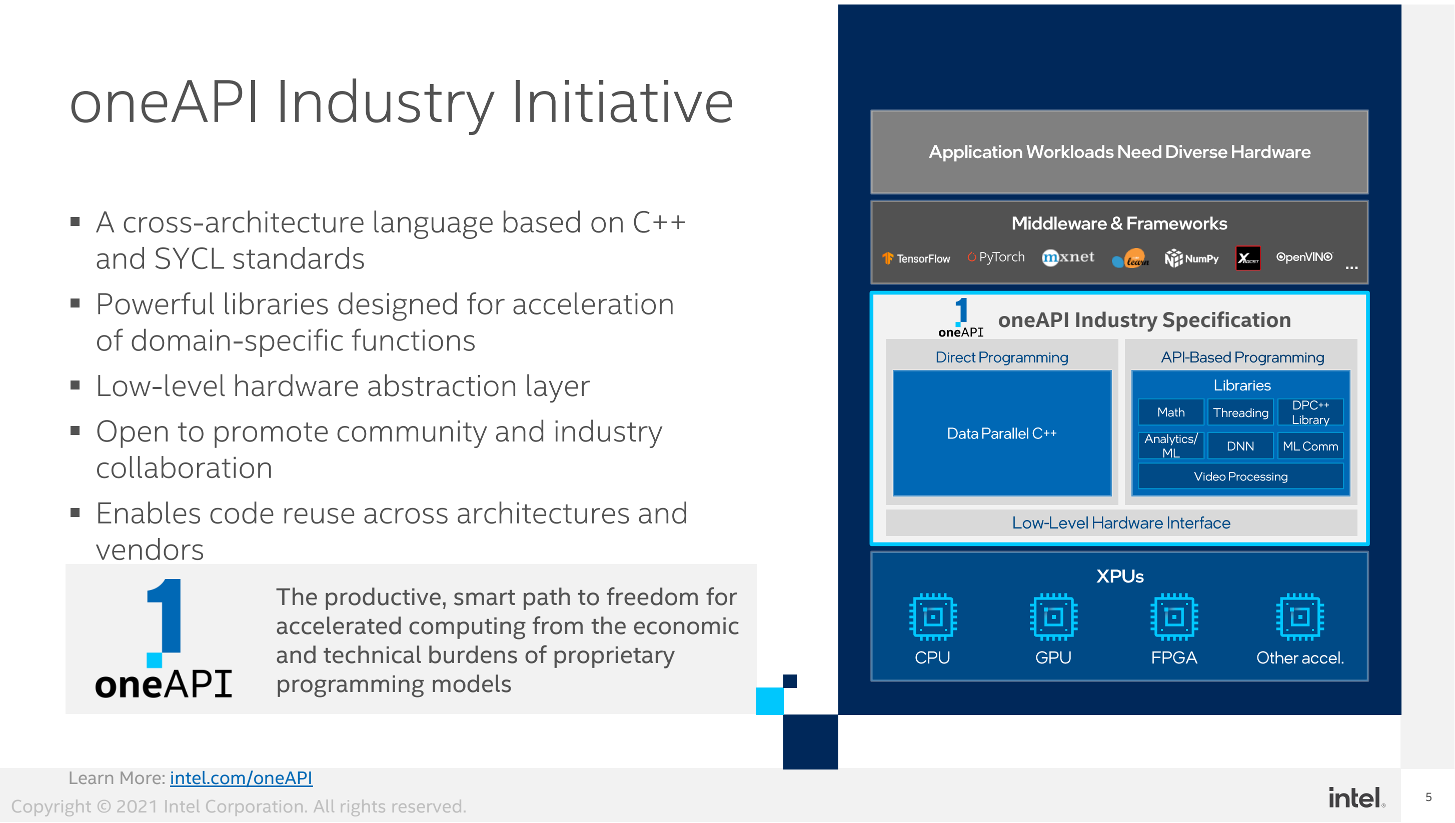
Intel® oneAPI performance analysis tools overview

Introducing oneAPI

- Cross-architecture programming that delivers freedom to choose the best hardware
- Based on industry standards and open specifications
- Exposes cutting-edge performance features of latest hardware
- Compatible with existing high-performance languages and programming models including C++, OpenMP, Fortran, and MPI



Learn More: intel.com/oneAPI



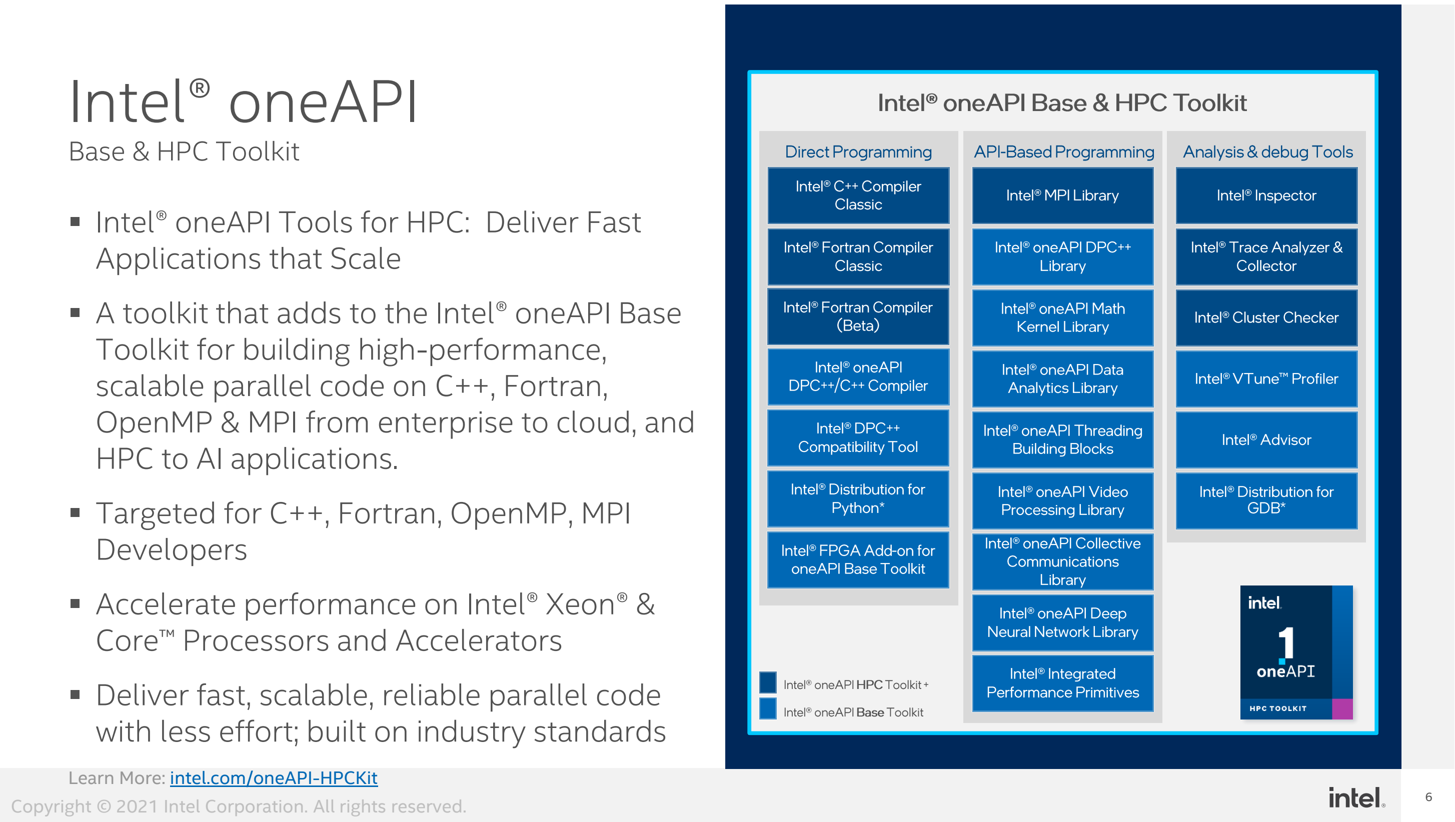
oneAPI Industry Initiative

- A cross-architecture language based on C++ and SYCL standards
- Powerful libraries designed for acceleration of domain-specific functions
- Low-level hardware abstraction layer
- Open to promote community and industry collaboration
- Enables code reuse across architectures and vendors

oneAPI

The productive, smart path to freedom for accelerated computing from the economic and technical burdens of proprietary programming models

Learn More: intel.com/oneAPI



Intel® oneAPI

Base & HPC Toolkit

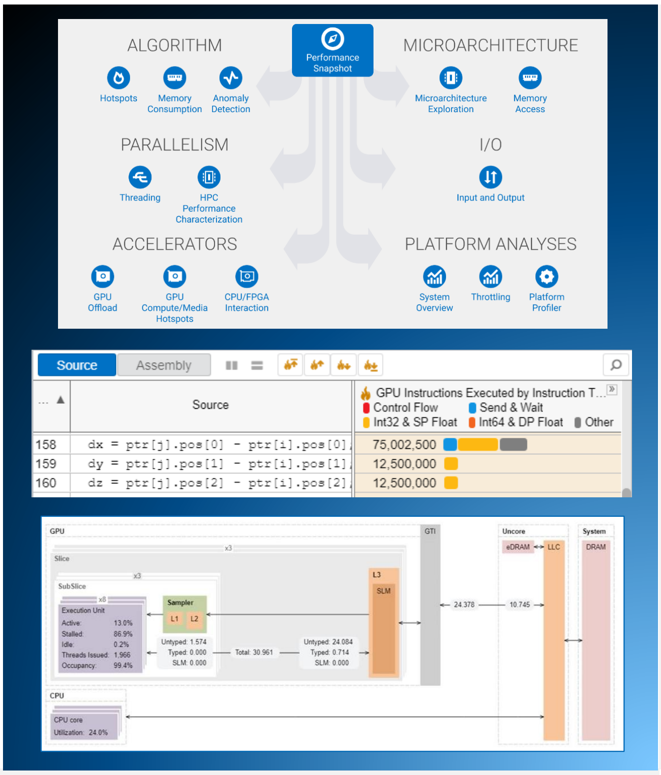
- Intel® oneAPI Tools for HPC: Deliver Fast Applications that Scale
- A toolkit that adds to the Intel® oneAPI Base Toolkit for building high-performance, scalable parallel code on C++, Fortran, OpenMP & MPI from enterprise to cloud, and HPC to AI applications.
- Targeted for C++, Fortran, OpenMP, MPI Developers
- Accelerate performance on Intel® Xeon® & Core™ Processors and Accelerators
- Deliver fast, scalable, reliable parallel code with less effort; built on industry standards

Intel® oneAPI Base & HPC Toolkit
Direct Programming
Intel® C++ Compiler Classic
Intel® Fortran Compiler Classic
Intel® Fortran Compiler (Beta)
Intel® oneAPI DPC++/C++ Compiler
Intel® DPC++ Compatibility Tool
Intel® Distribution for Python*
Intel® FPGA Add-on for oneAPI Base Toolkit
API-Based Programming
Intel® MPI Library
Intel® oneAPI DPC++ Library
Intel® oneAPI Math Kernel Library
Intel® oneAPI Data Analytics Library
Intel® oneAPI Threading Building Blocks
Intel® oneAPI Video Processing Library
Intel® oneAPI Collective Communications Library
Intel® oneAPI Deep Neural Network Library
Intel® Integrated Performance Primitives
Analysis & debug Tools
Intel® Inspector
Intel® Trace Analyzer & Collector
Intel® Cluster Checker
Intel® VTune™ Profiler
Intel® Advisor
Intel® Distribution for GDB*
Intel® oneAPI HPC Toolkit+
Intel® oneAPI Base Toolkit
intel.
oneAPI
HPC TOOLKIT

Learn More: intel.com/oneAPI-HPCKit

Intel® VTune™ Profiler

- Get the Right Data to Find Bottlenecks
  - Profiling for CPU, GPU, FPGA, threading, memory, cache, storage, offload, power…
  - DPC++, C, C++, Fortran, Python*, Go*, Java*, or a mix
  - Linux, Windows, FreeBSD, Android, Yocto and more
- Analyze Data Faster
  - See data on your source, in architecture diagrams, as a histogram, on a timeline…
  - Filter and organize data to find answers
- Work Your Way
  - Graphical user interface or command line
  - Profile locally and remotely
  - Install as an application
  - Install as a server accessible with a web browser



… ▲	Source	GPU Instructions Executed by Instruction T… 🟥 Control Flow 🟦 Send & Wait 🟨 Int32 & SP Float 🟧 Int64 & DP Float ▮ Other
158	`dx = ptr[j].pos[0] - ptr[i].pos[0]`	75,002,500
159	`dy = ptr[j].pos[1] - ptr[i].pos[1]`	12,500,000
160	`dz = ptr[j].pos[2] - ptr[i].pos[2]`	12,500,000

Execution Unit
Active: 13.0%
Stalled: 86.9%
Idle: 0.2%
Threads Issued: 1.966
Occupancy: 99.4%

Untyped: 1.574
Typed: 0.000
SLM: 0.000

Total: 30.961

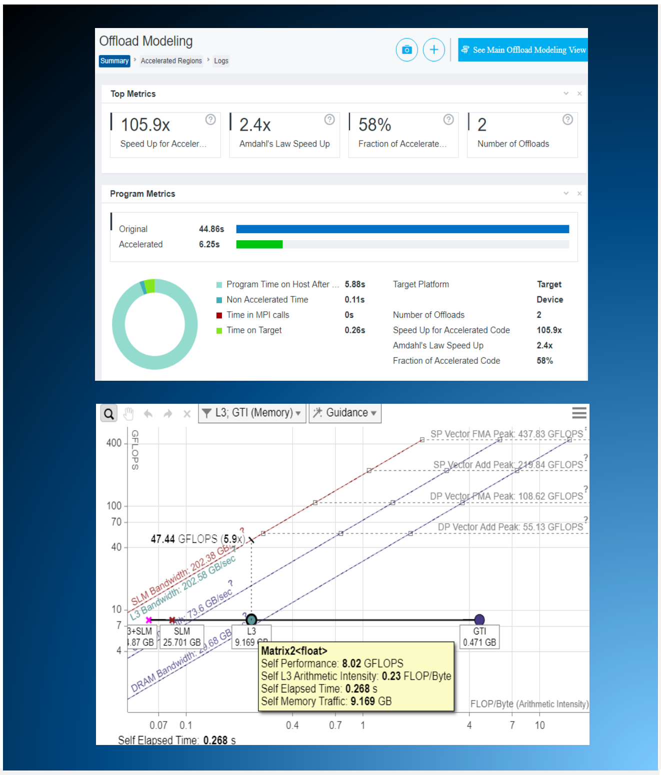
Untyped: 24.084
Typed: 0.714
SLM: 0.000

24.378
10.745

CPU core
Utilization: 24.0%

Intel® Advisor

- Offload Modelling
  - Efficiently offload your code to GPUs even before you have the hardware
- Automated Roofline Analysis
  - Optimize your GPU/CPU code for memory and compute
- Vectorization Optimization
  - Enable more vector parallelism and improve its efficiency
- Thread Prototyping
  - Add effective threading to unthreaded applications
- Flow Graph Analyzer
  - Create, visualize and analyze task and dependency computation graphs



Offload Modeling
Summary
Accelerated Regions
Logs
See Main Offload Modeling View
Top Metrics
Program Metrics
L3; GTI (Memory)
Guidance
GFLOPS
400
100
70
40
10
7
4
SP Vector FMA Peak: 437.83 GFLOPS
SP Vector Add Peak: 219.84 GFLOPS
DP Vector FMA Peak: 108.62 GFLOPS
DP Vector Add Peak: 55.13 GFLOPS
47.44 GFLOPS (5.9x)
SLM Bandwidth: 202.38 GB/sec
L3 Bandwidth: 202.58 GB/sec
…th: 73.6 GB/sec
DRAM Bandwidth: 20.68 GB…
3+SLM .87 GB
SLM 25.701 GB
L3 9.169 GB
GTI 0.471 GB
FLOP/Byte (Arithmetic Intensity)
0.07
0.1
0.4
0.7
1
4
7
10

105.9x Speed Up for Acceler…
2.4x Amdahl's Law Speed Up
58% Fraction of Accelerate…
2 Number of Offloads

Original	44.86s
Accelerated	6.25s

Program Time on Host After …	5.88s	Target Platform	Target Device
Non Accelerated Time	0.11s		
Time in MPI calls	0s	Number of Offloads	2
Time on Target	0.26s	Speed Up for Accelerated Code	105.9x
		Amdahl's Law Speed Up	2.4x
		Fraction of Accelerated Code	58%

Matrix2<float>
Self Performance: **8.02** GFLOPS
Self L3 Arithmetic Intensity: **0.23** FLOP/Byte
Self Elapsed Time: **0.268** s
Self Memory Traffic: **9.169** GB

Self Elapsed Time: **0.268** s

Performance Analysis Types

Get the big picture first with a Snapshot or Platform Profiler

	Snapshot Quickly size potential performance gain. Run a test "during a coffee break".	**In-Depth** Advanced collection & analysis. Insight for effective optimization.
Application Focus • HPC App developer focus • 1 app running during test	VTune Profiler's **Application Performance Snapshot** L⏲	**VTune Profiler** • Many profiles S-M⏲ **Intel Advisor** • Vectorization S⏲ **ITAC** • MPI Optimization S-L⏲
System Focus • Deployed system focus • Full system load test		**VTune Profiler** **- System-wide sampling** S-M⏲ **- Platform Profiler:** L⏲

Maximum collection times: L⏲=long (hours) M⏲=medium (minutes) S⏲=short (seconds-few minutes)

Application Performance Snapshot

A part of Intel® Intel® VTune™ Profiler

A Fast Way to Discover Untapped Performance

Intel® VTune™ Profiler - Application Performance Snapshot

Quick & easy performance overview

- Install & run a test case during a coffee break

All the data in one place

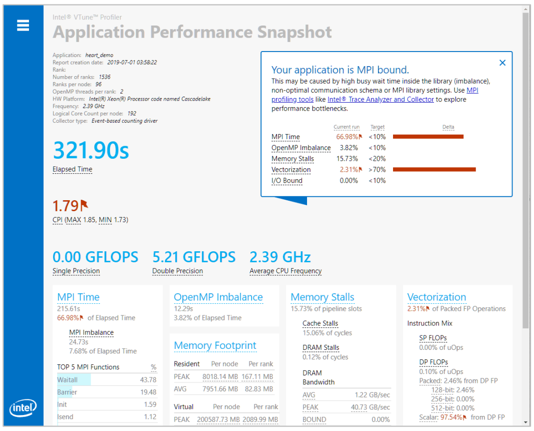
- MPI + OpenMP + Memory + Floating Point

Popular MPI implementations

- Intel® MPI, MPICH, OpenMPI and Cray MPI

New for 2020:

- Communication pattern diagnosis
- See time in high bandwidth, not just average
- Profile large MPI applications >64K ranks



Intel® VTune™ Profiler

Application Performance Snapshot

Application: *heart_demo*
Report creation date: *2019-07-01 03:58:22*
Rank:
Number of ranks: *1536*
Ranks per node: *96*
OpenMP threads per rank: *2*
HW Platform: *Intel(R) Xeon(R) Processor code named Cascadelake*
Frequency: *2.39 GHz*
Logical Core Count per node: *192*
Collector type: *Event-based counting driver*

321.90s
Elapsed Time

1.79
CPI (MAX 1.85, MIN 1.73)

0.00 GFLOPS
Single Precision

5.21 GFLOPS
Double Precision

2.39 GHz
Average CPU Frequency

Your application is MPI bound.

This may be caused by high busy wait time inside the library (imbalance), non-optimal communication schema or MPI library settings. Use MPI profiling tools like Intel® Trace Analyzer and Collector to explore performance bottlenecks.

	Current run	Target	Delta
MPI Time	66.98%	<10%	
OpenMP Imbalance	3.82%	<10%	
Memory Stalls	15.73%	<20%	
Vectorization	2.31%	>70%	
I/O Bound	0.00%	<10%	

MPI Time
215.61s
66.98% of Elapsed Time

MPI Imbalance
24.73s
7.68% of Elapsed Time

TOP 5 MPI Functions	%
Waitall	43.78
Barrier	19.48
Init	1.59
Isend	1.12

OpenMP Imbalance
12.29s
3.82% of Elapsed Time

Memory Footprint

Resident	Per node	Per rank
PEAK	8018.14 MB	167.11 MB
AVG	7951.66 MB	82.83 MB

Virtual	Per node	Per rank
PEAK	200587.73 MB	2089.99 MB

Memory Stalls
15.73% of pipeline slots

Cache Stalls
15.06% of cycles

DRAM Stalls
0.12% of cycles

DRAM Bandwidth	
AVG	1.22 GB/sec
PEAK	40.73 GB/sec
BOUND	0.00%

Vectorization
2.31% of Packed FP Operations

Instruction Mix

SP FLOPs
0.00% of uOps

DP FLOPs
0.10% of uOps
Packed: 2.46% from DP FP
128-bit: 2.46%
256-bit: 0.00%
512-bit: 0.00%
Scalar: 97.54% from DP FP

Linux* only.

Better Snapshots – More Ranks

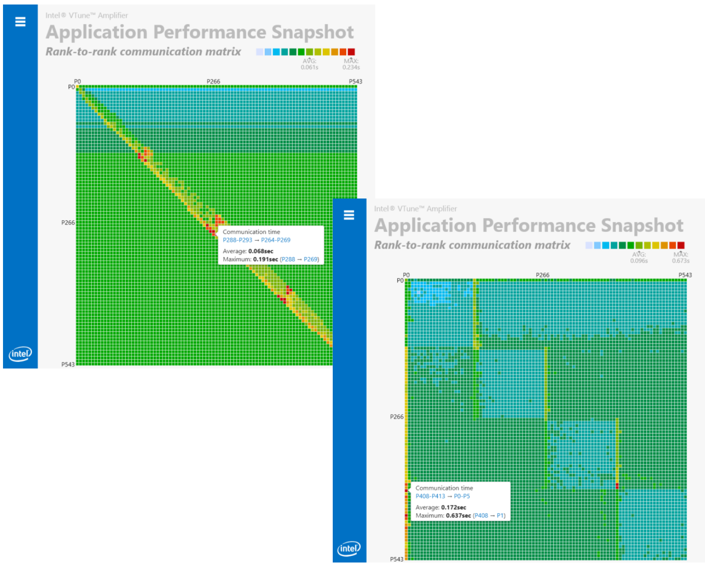
Intel® VTune Profiler – Application Performance Snapshot

Find MPI communication patterns that cause poor MPI scaling

- See rank-to-rank communication by both time and volume
- See time in high bandwidth, not just average

Profile larger MPI applications

- Scales to >64K ranks



Intel® VTune™ Amplifier

Application Performance Snapshot

Rank-to-rank communication matrix

Communication time
P288-P293 → P264-P269
Average: **0.068sec**
Maximum: **0.191sec** (P288 → P269)

Intel® VTune™ Amplifier

Application Performance Snapshot

Rank-to-rank communication matrix

Communication time
P408-P413 → P0-P5
Average: **0.172sec**
Maximum: **0.637sec** (P408 → P1)

Learn More: https://software.intel.com/content/www/us/en/develop/documentation/get-started-with-application-performance-snapshot/top.html

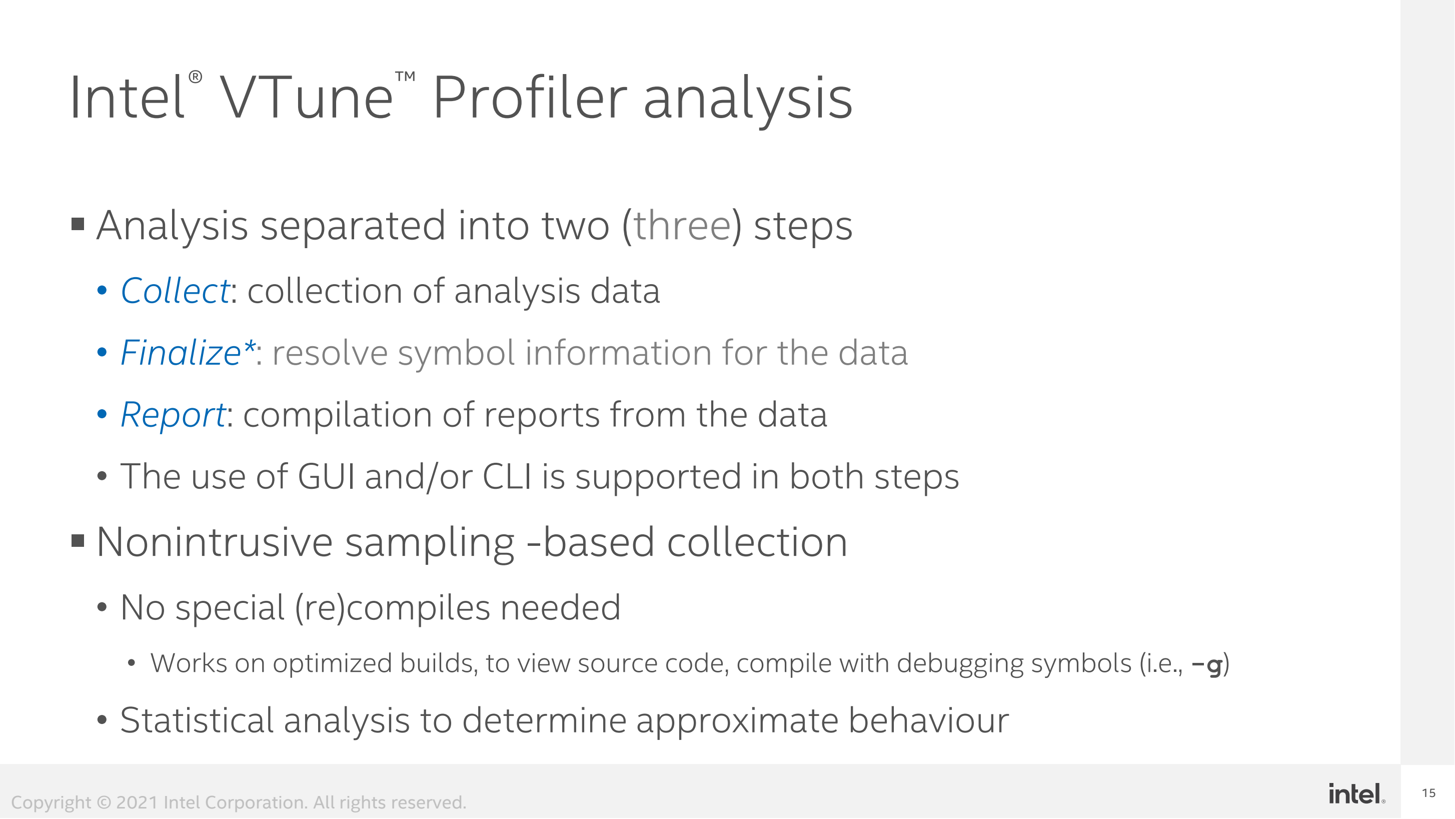
Intel® Application Performance Snapshot

Example

```
# Source Application Performance Snapshot environment
> source /opt/intel/oneapi/vtune/latest/apsvars.sh
# Collect data
> mpirun -np 4 -env OMP_NUM_THREADS=2 aps ./testc
# Generate report
> aps --report aps_result_20210512/ -s
Loading 100.00%
| Summary information
|--------------------------------------------------------------------
  Application                     : testc
  Report creation date            : 2021-05-12 14:02:57
  Number of ranks                 : 4
  Ranks per node                  : 4
  OpenMP threads number per rank: 2
  HW Platform                     : Intel(R) Xeon(R) Processor code named Broadwell
  Frequency                       : 2.19 GHz
  Logical core count per node    : 88
  Collector type                  : Driverless Perf system-wide counting
...
```

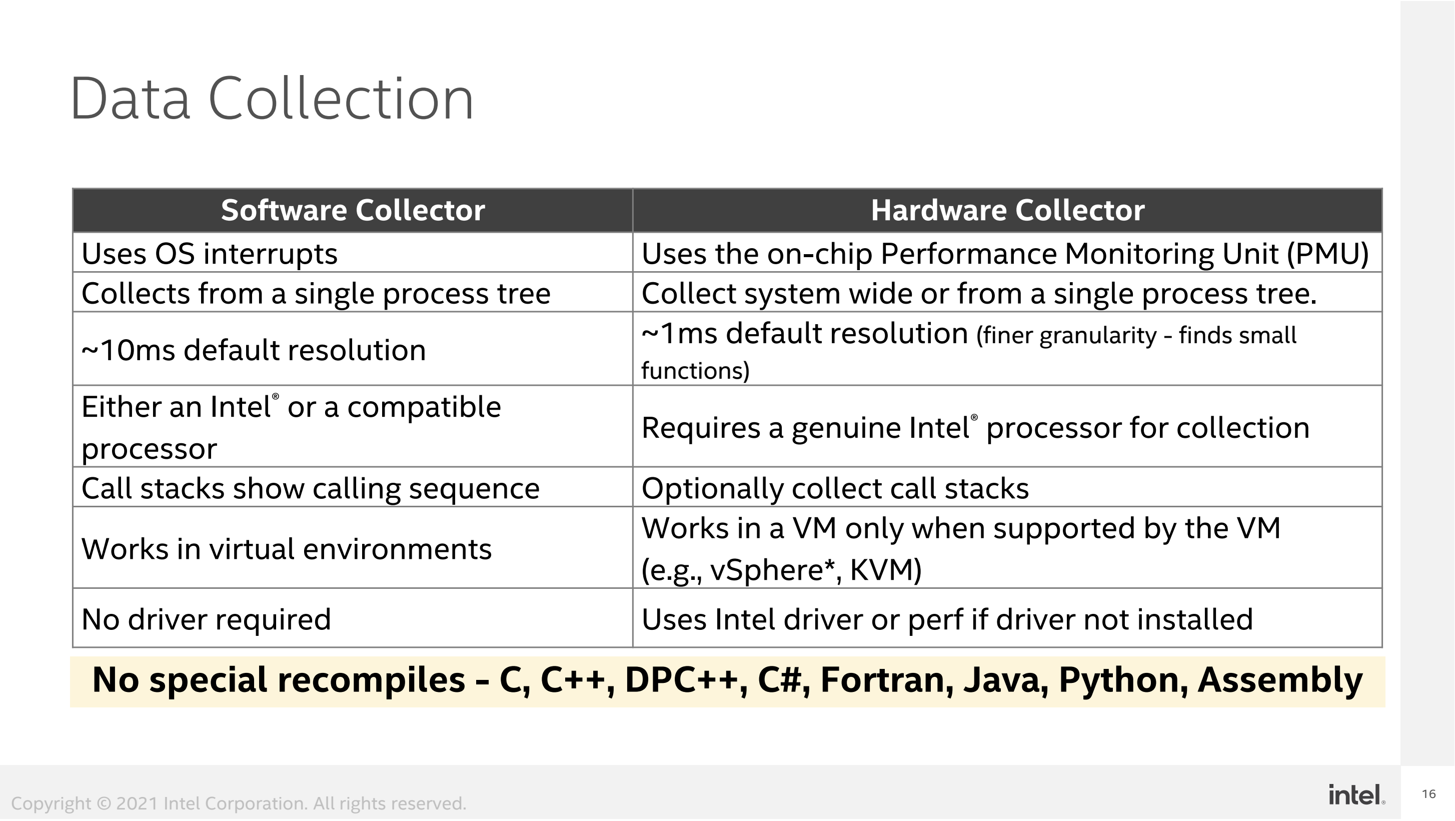
Introduction to Intel® VTune™ Profiler

Features and analysis types, Graphical User Interface (GUI), Command Line Interface (CLI)



Intel® VTune™ Profiler analysis

- Analysis separated into two (three) steps
  - *Collect*: collection of analysis data
  - *Finalize**: resolve symbol information for the data
  - *Report*: compilation of reports from the data
  - The use of GUI and/or CLI is supported in both steps
- Nonintrusive sampling -based collection
  - No special (re)compiles needed
    - Works on optimized builds, to view source code, compile with debugging symbols (i.e., `-g`)
  - Statistical analysis to determine approximate behaviour



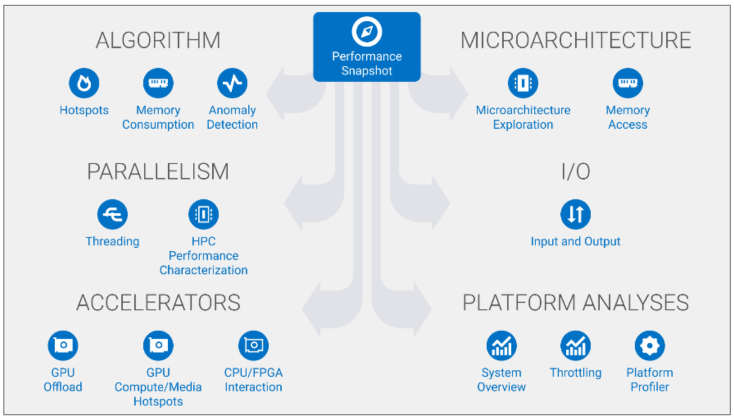
Data Collection

Software Collector	Hardware Collector
Uses OS interrupts	Uses the on-chip Performance Monitoring Unit (PMU)
Collects from a single process tree	Collect system wide or from a single process tree.
~10ms default resolution	~1ms default resolution (finer granularity - finds small functions)
Either an Intel® or a compatible processor	Requires a genuine Intel® processor for collection
Call stacks show calling sequence	Optionally collect call stacks
Works in virtual environments	Works in a VM only when supported by the VM (e.g., vSphere*, KVM)
No driver required	Uses Intel driver or perf if driver not installed

No special recompiles - C, C++, DPC++, C#, Fortran, Java, Python, Assembly

VTune Graphical User Interface (GUI)

- Graphical tool **`vtune-gui`**
  - Default location (Linux): **`/opt/intel/oneapi/vtune/2021.2.0/bin64/vtune-gui`**
- Pure GUI workflow
  - Set up a project
  - Choose analysis type
  - View analysis results



Performance Snapshot

ALGORITHM

Hotspots

Memory Consumption

Anomaly Detection

MICROARCHITECTURE

Microarchitecture Exploration

Memory Access

PARALLELISM

Threading

HPC Performance Characterization

I/O

Input and Output

ACCELERATORS

GPU Offload

GPU Compute/Media Hotspots

CPU/FPGA Interaction

PLATFORM ANALYSES

System Overview

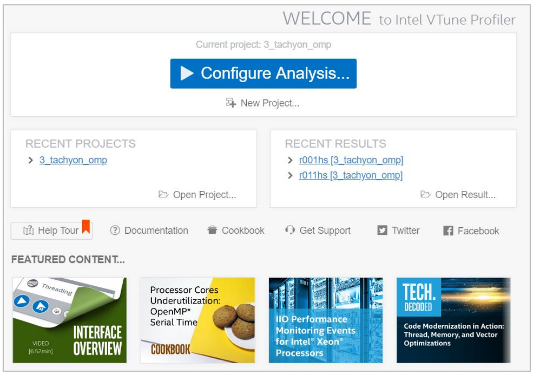
Throttling

Platform Profiler

VTune GUI

Intel® VTune™ Profiler

- Welcome page
  - Quick access to documentation and training
- Built-in sample code, pre-collected results
  - Easy to explore tutorials
- Help tour overlay
  - Quickly learn essential user interface controls



WELCOME to Intel VTune Profiler

Current project: 3_tachyon_omp

▶ Configure Analysis...

New Project...

RECENT PROJECTS

› 3_tachyon_omp

Open Project...

RECENT RESULTS

› r001hs [3_tachyon_omp]

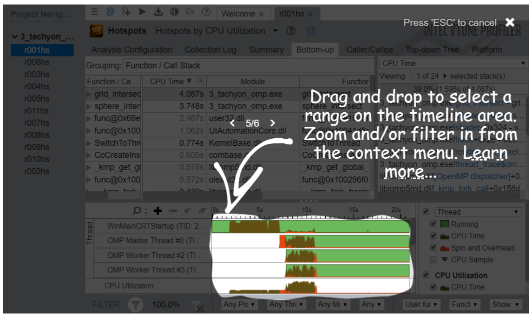
› r011hs [3_tachyon_omp]

Open Result...

Help Tour | Documentation | Cookbook | Get Support | Twitter | Facebook

FEATURED CONTENT...

INTERFACE OVERVIEW
Processor Cores Underutilization: OpenMP* Serial Time
COOKBOOK
IIO Performance Monitoring Events for Intel® Xeon® Processors
TECH. DECODED
Code Modernization in Action: Thread, Memory, and Vector Optimizations



Press 'ESC' to cancel

Drag and drop to select a range on the timeline area. Zoom and/or filter in from the context menu. Learn more...

VTune GUI: Profile Python & Go!

And Mixed Python / C++ / Fortran

Low Overhead Sampling

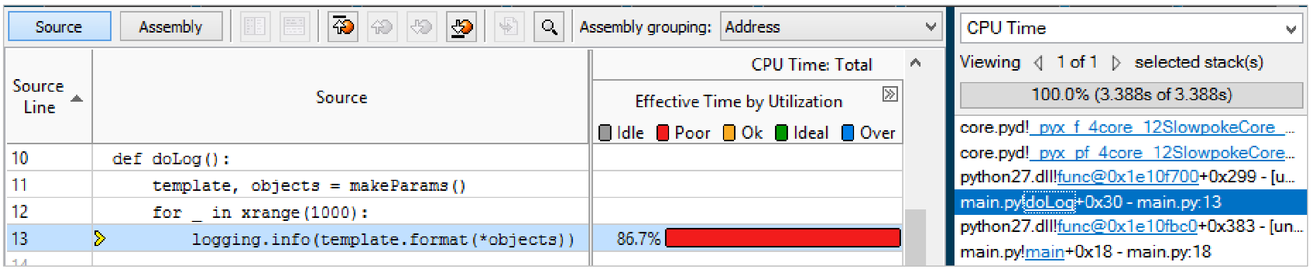
- Accurate performance data without high overhead instrumentation
- Launch application or attach to a running process

Precise Line Level Details

- No guessing, see source line level detail

Mixed Python / native C, C++, Fortran…

- Optimize native code driven by Python



Source Line ▲	Source	CPU Time: Total Effective Time by Utilization ■ Idle ■ Poor ■ Ok ■ Ideal ■ Over
10	`def doLog():`	
11	`    template, objects = makeParams()`	
12	`    for _ in xrange(1000):`	
13	`        logging.info(template.format(*objects))`	86.7%

Viewing ◁ 1 of 1 ▷ selected stack(s)

100.0% (3.388s of 3.388s)

```
core.pyd!_pyx_f_4core_12SlowpokeCore_...
core.pyd!_pyx_pf_4core_12SlowpokeCore...
python27.dll!func@0x1e10f700+0x299 - [u...
main.py!doLog+0x30 - main.py:13
python27.dll!func@0x1e10fbc0+0x383 - [un...
main.py!main+0x18 - main.py:18
```

VTune GUI: Hotspots

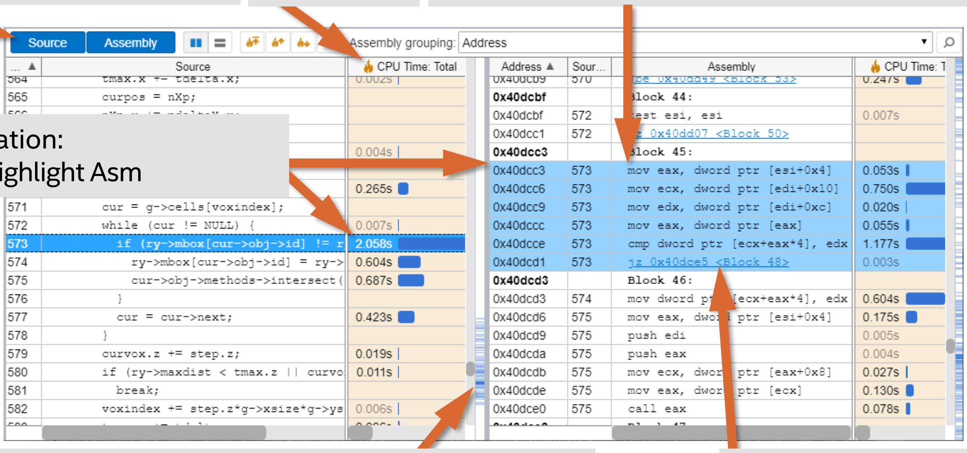
Double Click from Grid or Timeline

View Source / Asm or both

CPU Time

Right click for instruction reference manual

Quick Asm navigation:
Select source to highlight Asm



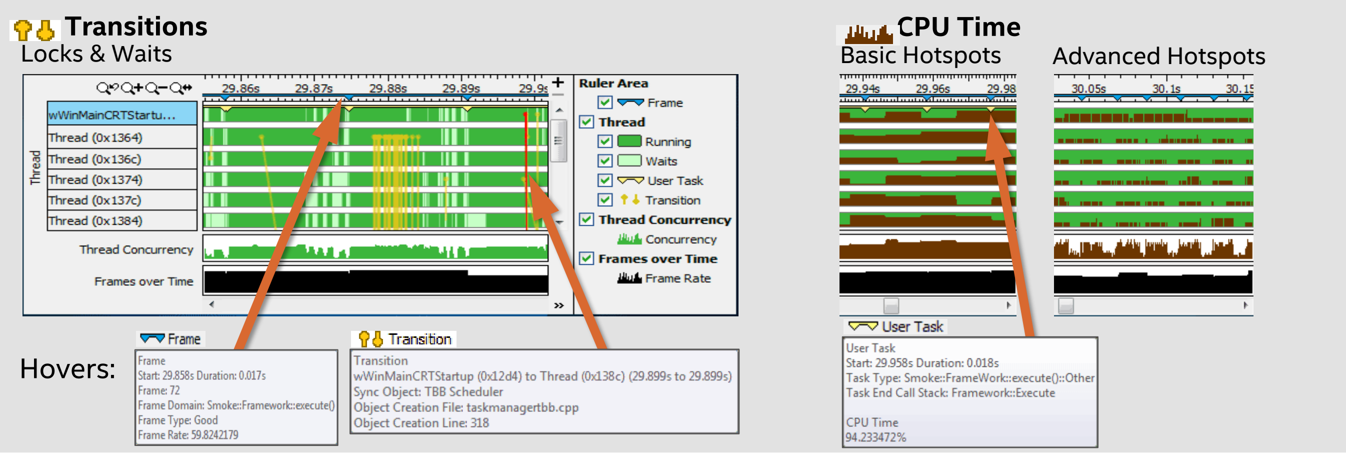
… ▲	Source	CPU Time: Total
564	`tmax.x += tdelta.x;`	0.002s
565	`curpos = nXp;`	
		0.004s
		0.265s
571	`cur = g->cells[voxindex];`	
572	`while (cur != NULL) {`	0.007s
573	`  if (ry->mbox[cur->obj->id] != r`	2.058s
574	`    ry->mbox[cur->obj->id] = ry->`	0.604s
575	`    cur->obj->methods->intersect(`	0.687s
576	`  }`	
577	`  cur = cur->next;`	0.423s
578	`}`	
579	`curvox.z += step.z;`	0.019s
580	`if (ry->maxdist < tmax.z || curvo`	0.011s
581	`  break;`	
582	`voxindex += step.z*g->xsize*g->ys`	0.006s

Address ▲	Sour…	Assembly	CPU Time: T
0x40dcb9	570	`jbe 0x40dd49 <Block 53>`	0.247s
0x40dcbf		`Block 44:`	
0x40dcbf	572	`test esi, esi`	0.007s
0x40dcc1	572	`jz 0x40dd07 <Block 50>`	
0x40dcc3		`Block 45:`	
0x40dcc3	573	`mov eax, dword ptr [esi+0x4]`	0.053s
0x40dcc6	573	`mov ecx, dword ptr [edi+0x10]`	0.750s
0x40dcc9	573	`mov edx, dword ptr [edi+0xc]`	0.020s
0x40dccc	573	`mov eax, dword ptr [eax]`	0.055s
0x40dcce	573	`cmp dword ptr [ecx+eax*4], edx`	1.177s
0x40dcd1	573	`jz 0x40dce5 <Block 48>`	0.003s
0x40dcd3		`Block 46:`	
0x40dcd3	574	`mov dword ptr [ecx+eax*4], edx`	0.604s
0x40dcd6	575	`mov eax, dword ptr [esi+0x4]`	0.175s
0x40dcd9	575	`push edi`	0.005s
0x40dcda	575	`push eax`	0.004s
0x40dcdb	575	`mov ecx, dword ptr [eax+0x8]`	0.027s
0x40dcde	575	`mov eax, dword ptr [ecx]`	0.130s
0x40dce0	575	`call eax`	0.078s

Scroll Bar “Heat Map” is an overview of hot spots

Click jump to scroll Asm

VTune GUI: Threading



Frame
Start: 29.858s Duration: 0.017s
Frame: 72
Frame Domain: Smoke::Framework::execute()
Frame Type: Good
Frame Rate: 59.8242179

Transition
wWinMainCRTStartup (0x12d4) to Thread (0x138c) (29.899s to 29.899s)
Sync Object: TBB Scheduler
Object Creation File: taskmanagertbb.cpp
Object Creation Line: 318

User Task
Start: 29.958s Duration: 0.018s
Task Type: Smoke::FrameWork::execute()::Other
Task End Call Stack: Framework::Execute

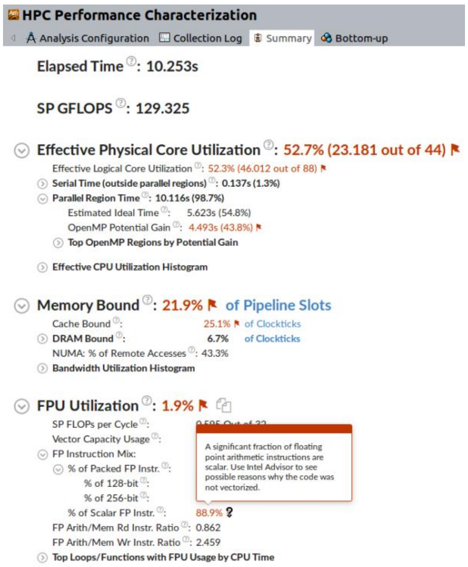
CPU Time
94.233472%

- Optional: Use API to mark frames and user tasks Frame User Task
- Optional: Add a mark during collection

VTune GUI: HPC Performance Characterization

Threading, Memory Access, Vectorization

- Threading: CPU Utilization
  - Serial vs. Parallel time
  - Top OpenMP regions by potential gain
  - Tip: Use hotspot OpenMP region analysis for more detail
- Memory Access Efficiency
  - Stalls by memory hierarchy
  - Bandwidth utilization
  - Tip: Use Memory Access analysis
- Vectorization: FPU Utilization
  - FLOPS † estimates from sampling
  - Tip: Use Intel Advisor for precise metrics and vectorization optimization



Elapsed Time: 10.253s

SP GFLOPS: 129.325

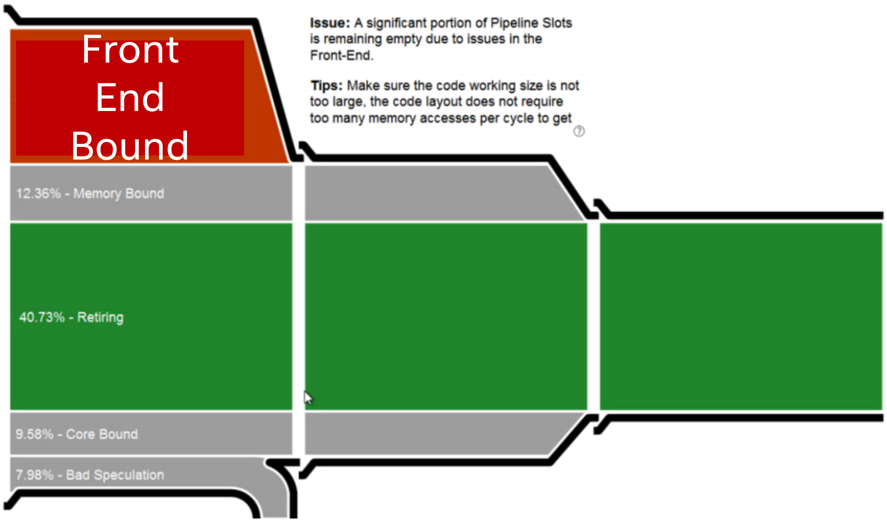
Effective Physical Core Utilization: 52.7% (23.181 out of 44)
Effective Logical Core Utilization: 52.3% (46.012 out of 88)
Serial Time (outside parallel regions): 0.137s (1.3%)
Parallel Region Time: 10.116s (98.7%)
Estimated Ideal Time: 5.623s (54.8%)
OpenMP Potential Gain: 4.493s (43.8%)
Top OpenMP Regions by Potential Gain
Effective CPU Utilization Histogram

Memory Bound: 21.9% of Pipeline Slots
Cache Bound: 25.1% of Clockticks
DRAM Bound: 6.7% of Clockticks
NUMA: % of Remote Accesses: 43.3%
Bandwidth Utilization Histogram

FPU Utilization: 1.9%
SP FLOPs per Cycle: 0.595 Out of 32
Vector Capacity Usage:
FP Instruction Mix:
% of Packed FP Instr.:
% of 128-bit:
% of 256-bit:
% of Scalar FP Instr.: 88.9%
FP Arith/Mem Rd Instr. Ratio: 0.862
FP Arith/Mem Wr Instr. Ratio: 2.459
Top Loops/Functions with FPU Usage by CPU Time

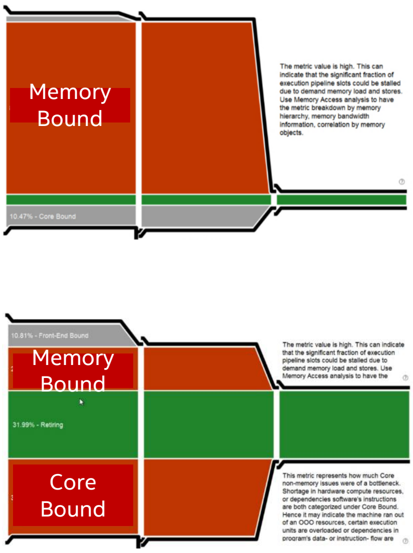
A significant fraction of floating point arithmetic instructions are scalar. Use Intel Advisor to see possible reasons why the code was not vectorized.

VTune GUI: Microarchitecture Exploration



Issue: A significant portion of Pipeline Slots is remaining empty due to issues in the Front-End.

Tips: Make sure the code working size is not too large, the code layout does not require too many memory accesses per cycle to get



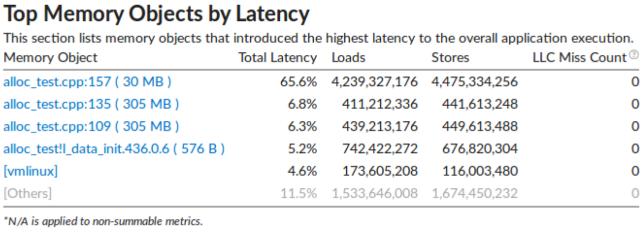
The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the

This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware compute resources, or dependencies software's instructions are both categorized under Core Bound. Hence it may indicate the machine ran out of an OOO resources, certain execution units are overloaded or dependencies in program's data- or instruction- flow are

VTune GUI: Memory Access Analysis

- Tune data structures for performance
  - Attribute cache misses to data structures (not just the code causing the miss)
  - Support for custom memory allocators
- Optimize NUMA latency & scalability
  - True & false sharing optimization
  - Auto detect max system bandwidth
  - Easier tuning of inter-socket bandwidth
- Easier install, Latest processors
  - No special drivers required on Linux*
  - Intel® Xeon Phi™ processor MCDRAM (high bandwidth memory) analysis

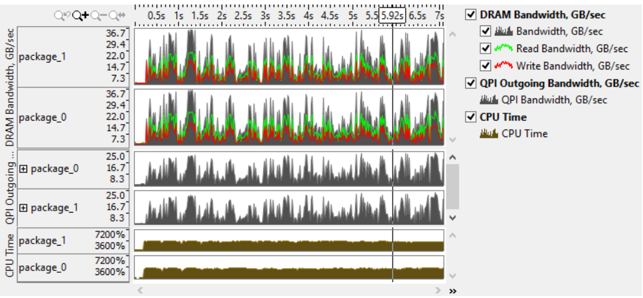


Top Memory Objects by Latency

This section lists memory objects that introduced the highest latency to the overall application execution.

Memory Object	Total Latency	Loads	Stores	LLC Miss Count
alloc_test.cpp:157 (30 MB)	65.6%	4,239,327,176	4,475,334,256	0
alloc_test.cpp:135 (305 MB)	6.8%	411,212,336	441,613,248	0
alloc_test.cpp:109 (305 MB)	6.3%	439,213,176	449,613,488	0
alloc_test!I_data_init.436.0.6 (576 B)	5.2%	742,422,272	676,820,304	0
[vmlinux]	4.6%	173,605,208	116,003,480	0
[Others]	11.5%	1,533,646,008	1,674,450,232	0

*N/A is applied to non-summable metrics.



VTune GUI: Memory Consumption Analysis

See What Is Allocating Memory

- Lists top memory consuming functions and objects
- View source to understand cause
- Filter by time using the memory consumption timeline

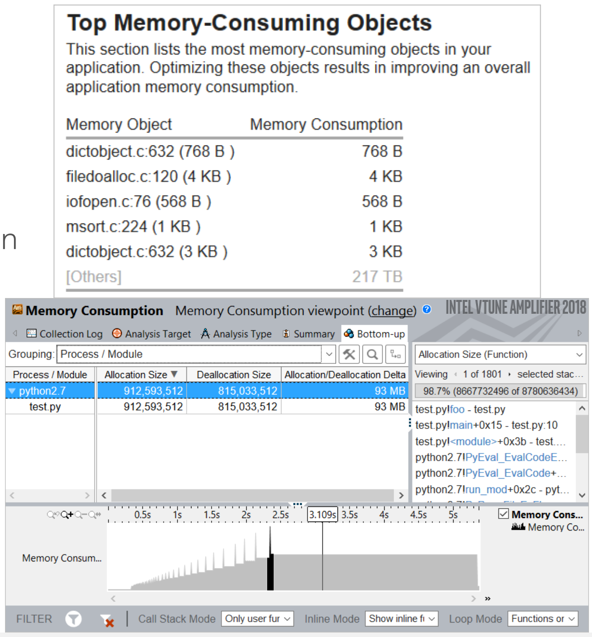
Standard & Custom Allocators

- Recognizes libc malloc/free, memkind and jemalloc libraries
- Use custom allocators after markup with ITT Notify API

Languages

- Python*
- Linux*: Native C, C++, Fortran

Native language support is not currently available for Windows*



Top Memory-Consuming Objects

This section lists the most memory-consuming objects in your application. Optimizing these objects results in improving an overall application memory consumption.

Memory Object	Memory Consumption
dictobject.c:632 (768 B)	768 B
filedoalloc.c:120 (4 KB)	4 KB
iofopen.c:76 (568 B)	568 B
msort.c:224 (1 KB)	1 KB
dictobject.c:632 (3 KB)	3 KB
[Others]	217 TB

Memory Consumption Memory Consumption viewpoint (change)

Grouping: Process / Module

Process / Module	Allocation Size ▼	Deallocation Size	Allocation/Deallocation Delta
▼ python2.7	912,593,512	815,033,512	93 MB
test.py	912,593,512	815,033,512	93 MB

Allocation Size (Function)

Viewing 1 of 1801 selected stac...

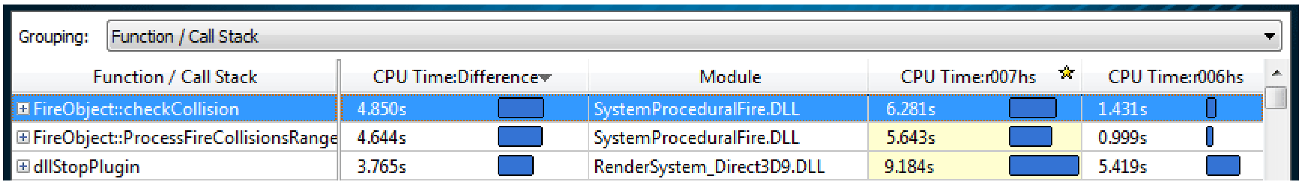
98.7% (8667732496 of 8780636434)

test.py!foo - test.py
test.py!main+0x15 - test.py:10
test.py!<module>+0x3b - test....
python2.7!PyEval_EvalCodeE...
python2.7!PyEval_EvalCode+...
python2.7!run_mod+0x2c - pyt...

FILTER | Call Stack Mode: Only user fur | Inline Mode: Show inline fi | Loop Mode: Functions or

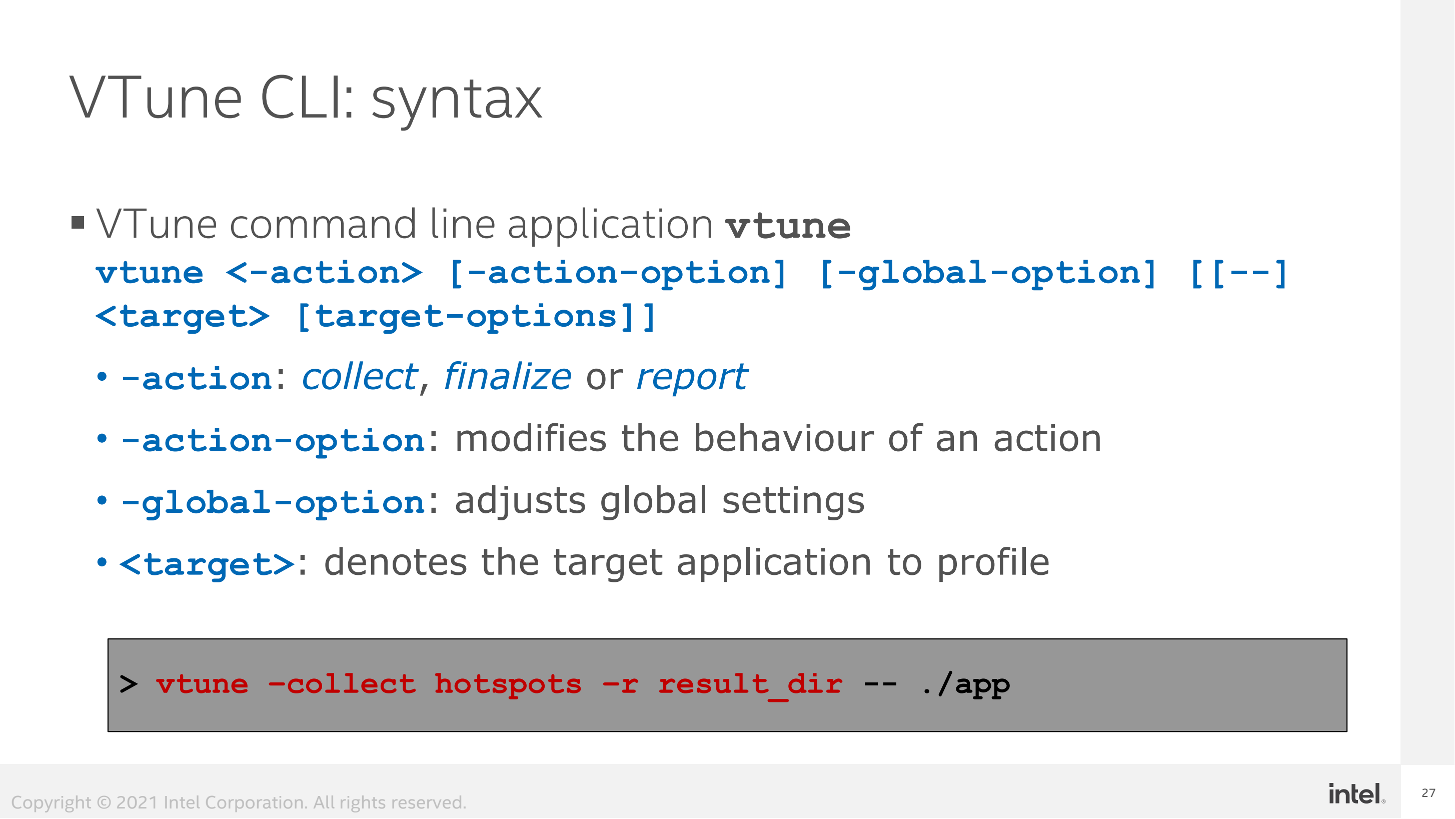
VTune GUI: Results comparison

- Quickly identify cause of regressions.
  - Run a command line analysis daily
  - Identify the function responsible so you know who to alert
- Compare 2 optimizations – What improved?
- Compare 2 systems – What didn't speed up as much?



Grouping: Function / Call Stack

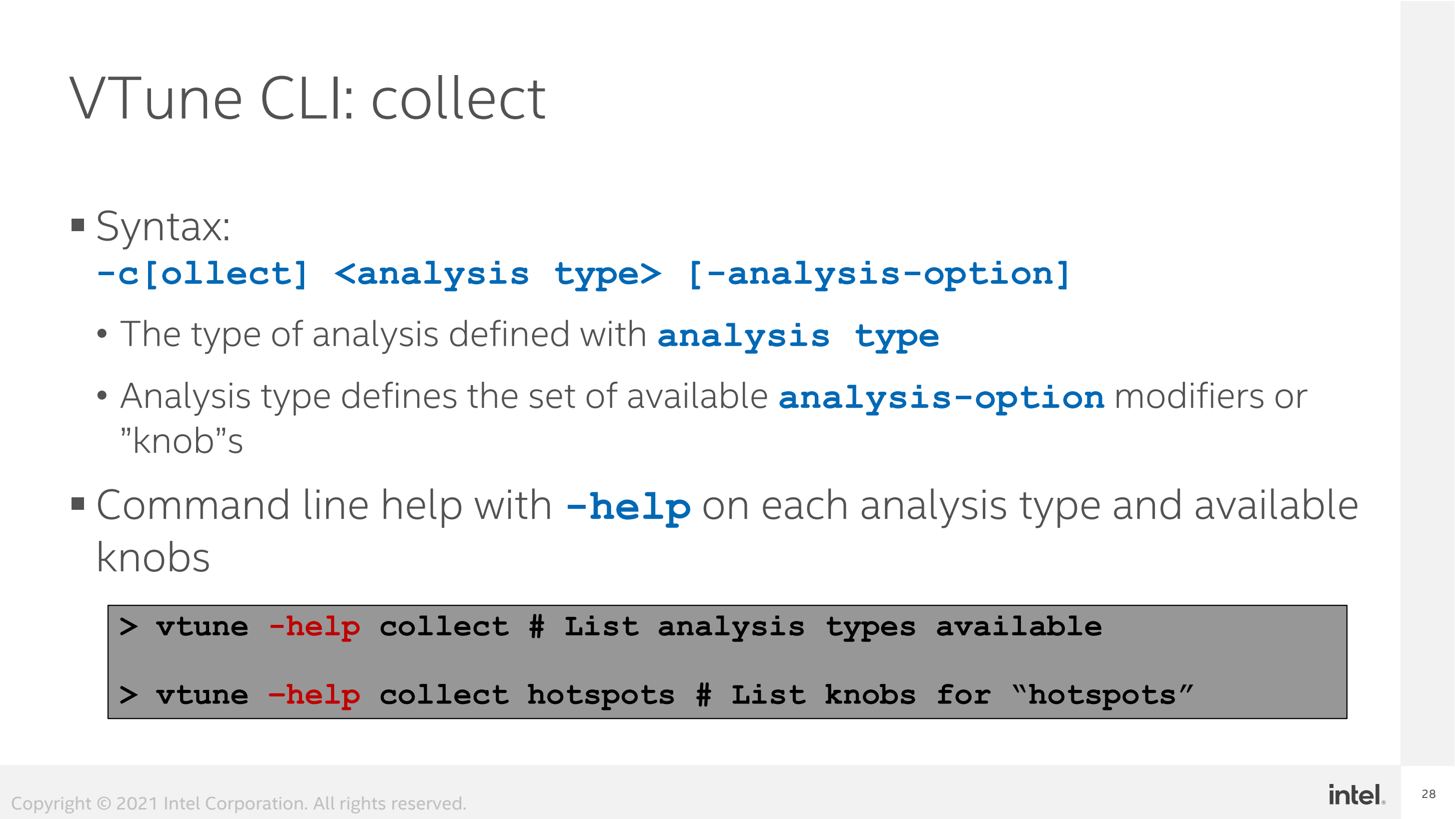
Function / Call Stack	CPU Time:Difference▼	Module	CPU Time:r007hs	CPU Time:r006hs
⊞ FireObject::checkCollision	4.850s	SystemProceduralFire.DLL	6.281s	1.431s
⊞ FireObject::ProcessFireCollisionsRange	4.644s	SystemProceduralFire.DLL	5.643s	0.999s
⊞ dllStopPlugin	3.765s	RenderSystem_Direct3D9.DLL	9.184s	5.419s



VTune CLI: syntax

- VTune command line application **`vtune`**
  `vtune <-action> [-action-option] [-global-option] [[--] <target> [target-options]]`
  - **`-action`**: *collect*, *finalize* or *report*
  - **`-action-option`**: modifies the behaviour of an action
  - **`-global-option`**: adjusts global settings
  - **`<target>`**: denotes the target application to profile

```
> vtune -collect hotspots -r result_dir -- ./app
```



VTune CLI: collect

- Syntax:
  `-c[ollect] <analysis type> [-analysis-option]`
  - The type of analysis defined with **`analysis type`**
  - Analysis type defines the set of available **`analysis-option`** modifiers or ”knob”s
- Command line help with **`-help`** on each analysis type and available knobs

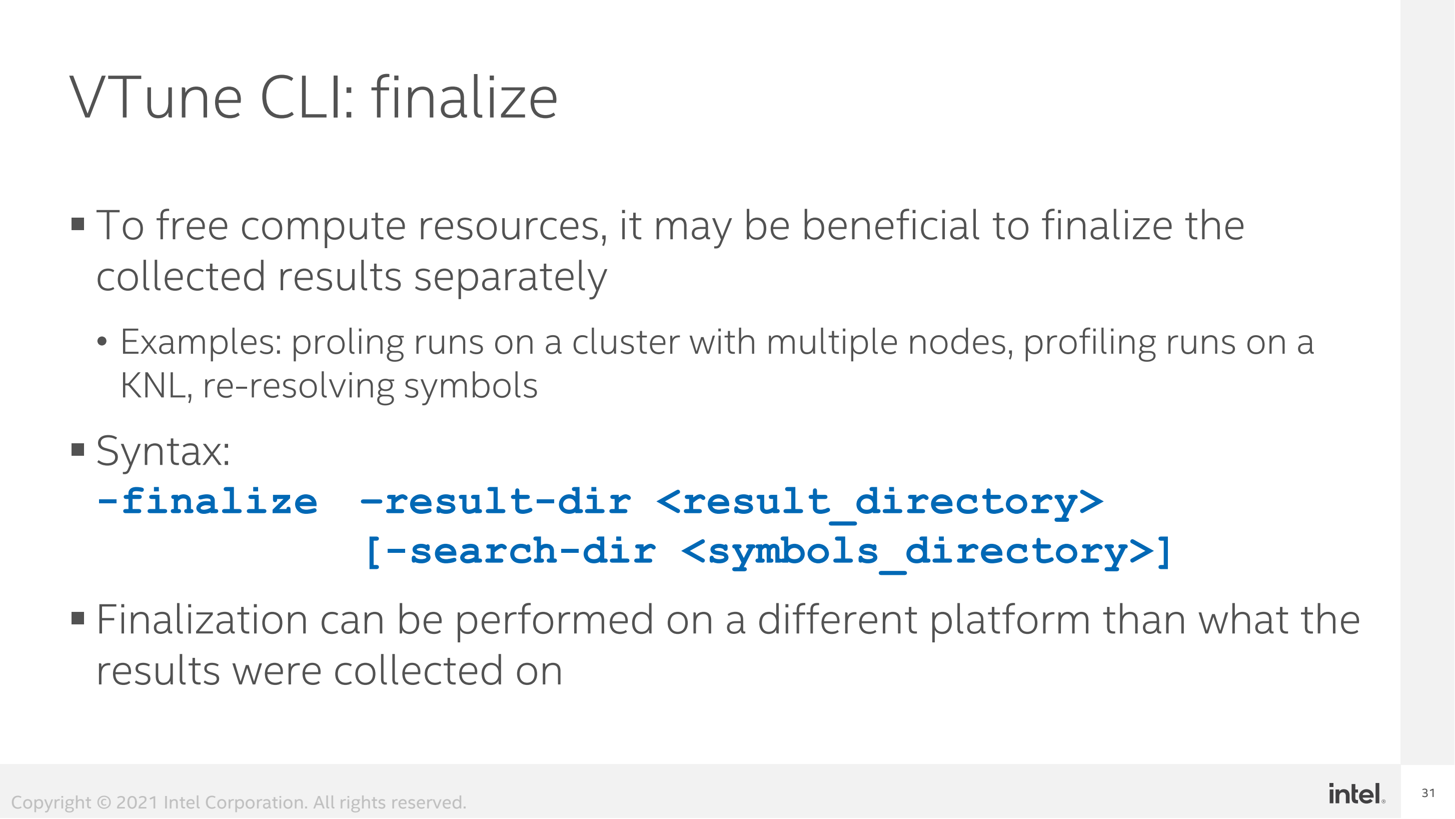
```
> vtune -help collect # List analysis types available

> vtune -help collect hotspots # List knobs for “hotspots”
```

VTune CLI: collect - analysis types

- For HPC, the analysis types of interest are
  - **`hotspots`**: Identify hotspots, collect stacks and call trees
  - **`hpc-performance`**: Analyze CPU and FPU utilization and memory access efficiency
  - **`threading:`** Analyze threading efficiency
  - **`memory-access`**: Identify memory access related issues and estimate memory bandwidth
  - **`memory-consumption`**: Identify memory consumption
  - **`io`**: Analyze processor and disk input and output
  - **`uarch-exploration`**: Identify low-level hardware issues

VTune CLI: collect - global modifiers

- A large number of global modifiers available
  - **`-finalization-mode`**: whether to finalize the result after the collection stops
  - **`-data-limit`**: limit the amount of data collected. The default is 1GB, set to 0 for unlimited
  - **`-quiet`**: limit the amount of information displayed
  - **`-search-dir`**: path where the binary and symbol files are stored
  - **`-result-dir`**: path where the result will be stored

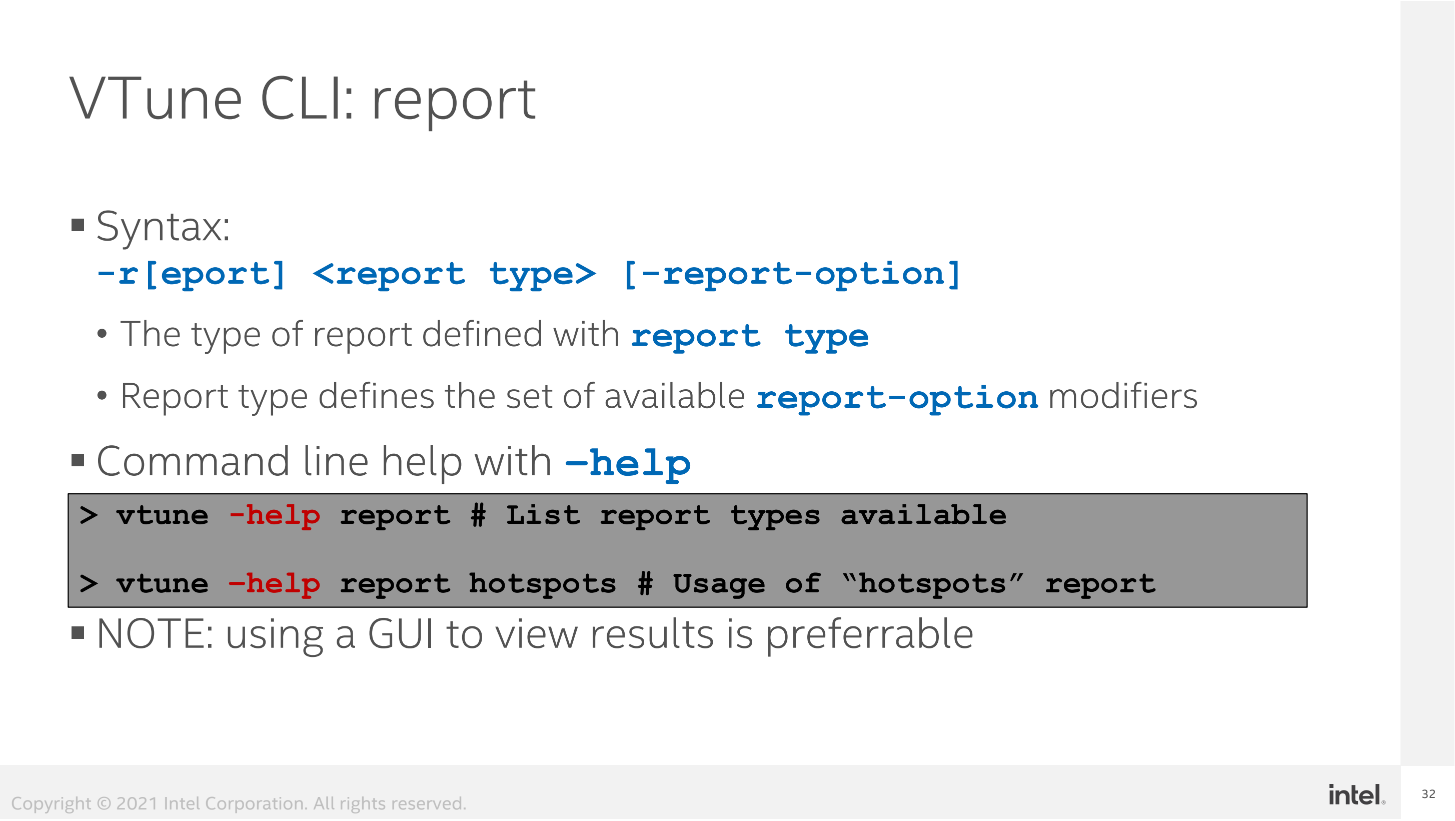


VTune CLI: finalize

- To free compute resources, it may be beneficial to finalize the collected results separately
  - Examples: proling runs on a cluster with multiple nodes, profiling runs on a KNL, re-resolving symbols
- Syntax:

```
-finalize  -result-dir <result_directory>
           [-search-dir <symbols_directory>]
```

- Finalization can be performed on a different platform than what the results were collected on



VTune CLI: report

- Syntax:

```
-r[eport] <report type> [-report-option]
```

  - The type of report defined with **report type**
  - Report type defines the set of available **report-option** modifiers
- Command line help with **-help**

```
> vtune -help report # List report types available

> vtune -help report hotspots # Usage of "hotspots" report
```

- NOTE: using a GUI to view results is preferrable

VTune CLI: report - report types

- For HPC, the report types of interest are
  - `summary`: Report overall application performance
  - `hotspots`: Report CPU time for application
  - `hw-events`: Display the total number of hardware events
- A report is automatically based on the type of data collected!

VTune CLI: report - global modifiers

- A large number of global modifiers available
  - `-column:` Specify which columns to include or exclude
  - `-filter`: Specify which data to include or exclude
  - `-group-by`: Specify grouping in a report
  - `-time-filter`: Specify which time range to include
  - `-source-search-dir`: path where the source code is stored
  - `-result-dir`: path where the result will be stored

VTune CLI: example

- Collect hotspots of application **nbody**, store results to directory **nbody_hs**

```
> vtune -collect hotspots -r nbody_hs -- ./nbody 262144
```

- View available columns in the result and then compile a hotspots report from specific columns

```
> vtune -report hotspots -r nbody_hs column=?

> vtune -report hotspots -r nbody_hs -column="CPU
 Time:Self","Source File"
```

Intel® VTune™ Profiler HPC workflow

Use of Intel® VTune™ Profiler in a cluster environment

Profiling HPC applications

- VTune can profile hybrid MPI+OpenMP applications on a cluster
  - For profiling MPI, use Intel® Trace Analyzer and Collector or Intel® MPI Performance Snapshot
- Recommended workflow:
  - Run **collect** (and **finalize**) with CLI on *a cluster*
  - Run **report** with GUI on *a local workstation* or a cluster login node
    - Finalized collection results can be transferred if needed

VTune with MPI applications (1/3)

- Single node application launch:
  `<vtune_command> [--] <mpi_command> <application>`

```
> vtune -collect advanced-hotspots -r result_dir -- mpirun -np 48
 ./mpi_app
```

- Encapsulates all the ranks to result directory
  - Example: ranks 0-47 in **`result_dir`**
- Works whenever VTune is able to track the processes created
  - Limited to profiling over a single node

VTune with MPI applications (2/3)

- Multiple node application launch:
  <mpi_command> <vtune_command> [--] <application>

```
> aprun -n 48 -ppn 16 vtune -collect hotspots -r result_dir
  ./mpi_app
```

- Results encapsulated to per-node directories suffixed with hostname
  - Example: ranks 0-15 in **result_dir.hostname1**, ranks 16-31 in **result_dir.hostname2**, ranks 32-47 in **result_dir.hostname3**

VTune with MPI applications (3/3)

- Selective rank profiling by modifying the MPI process launch:

```
> mpirun -n 1 ./mpi_app : -n 1 vtune -collect hotspots -r
 result_dir ./mpi_app : -n 14 ./mpi_app
```

- Intel MPI supports **-gtool “<command>:<rank-set>[=mode]”** option:

```
> mpirun -n 16 -gtool “vtune -collect hotspots -r result_dir :1”
  ./mpi_app
```

intel®

Notices & Disclaimers

Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

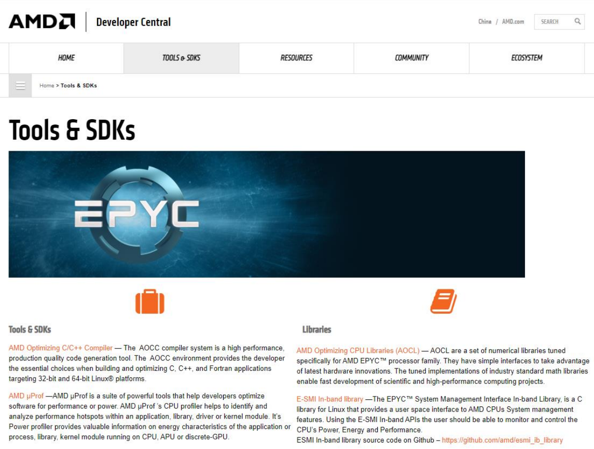
Introduction to AMD µProf Profiler v3.4

Dr.-Ing. Michael Klemm
Senior FAE, Principal Member of Technical Staff
HPC Center of Excellence

AMD offers software development tools optimized for HPC applications on EPYC™ CPUs while supporting developer choice with tools and methods

- AMD Optimizing CPU Compiler (AOCC)
- AMD Optimized CPU Libraries (AOCL)
- AMD µProf profiler
- Spack package support of HPC applications
- Support of open-source tools

developer.amd.com



Tools & SDKs

Tools & SDKs

AMD Optimizing C/C++ Compiler — The AOCC compiler system is a high performance, production quality code generation tool. The AOCC environment provides the developer the essential choices when building and optimizing C, C++, and Fortran applications targeting 32-bit and 64-bit Linux® platforms.

AMD µProf —AMD µProf is a suite of powerful tools that help developers optimize software for performance or power. AMD µProf 's CPU profiler helps to identify and analyze performance hotspots within an application, library, driver or kernel module. It's Power profiler provides valuable information on energy characteristics of the application or process, library, kernel module running on CPU, APU or discrete-GPU.

Libraries

AMD Optimizing CPU Libraries (AOCL) — AOCL are a set of numerical libraries tuned specifically for AMD EPYC™ processor family. They have simple interfaces to take advantage of latest hardware innovations. The tuned implementations of industry standard math libraries enable fast development of scientific and high-performance computing projects.

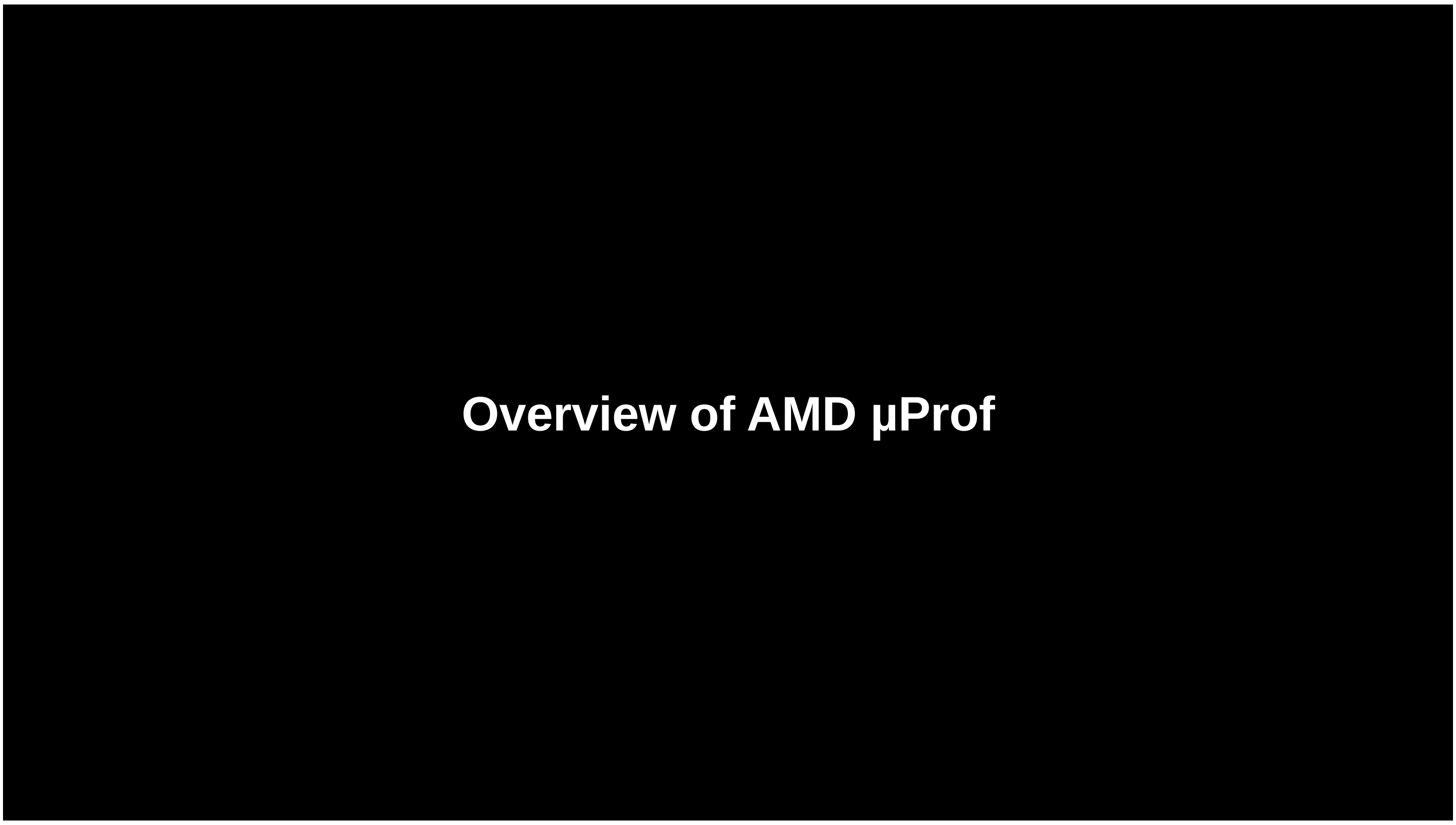
E-SMI In-band library —The EPYC™ System Management Interface In-band Library, is a C library for Linux that provides a user space interface to AMD CPUs System management features. Using the E-SMI In-band APIs the user should be able to monitor and control the CPU's Power, Energy and Performance.
ESMI In-band library source code on Github – https://github.com/amd/esmi_ib_library

µProf vs. uprof usage

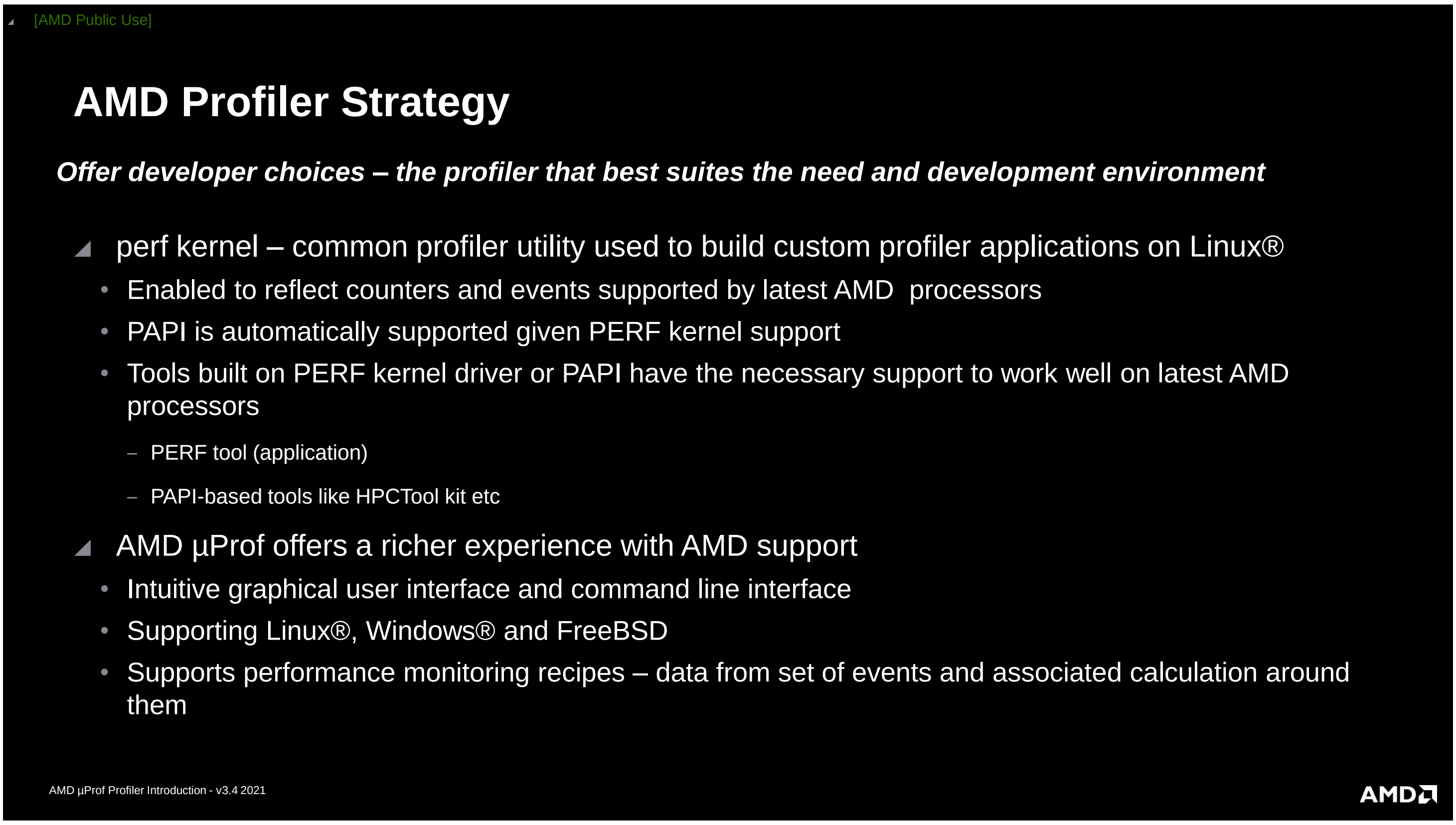
- AMD µProf is pronounced as “MICROprof”
- “uprof” is used for computer-readable form
  - Directory path names
  - Command lines
  - Scripts
  - URLs

AGENDA

- AMD µProf – Overview
- Profiling Overview
- System Analysis
- Application Analysis



Overview of AMD µProf



AMD Profiler Strategy

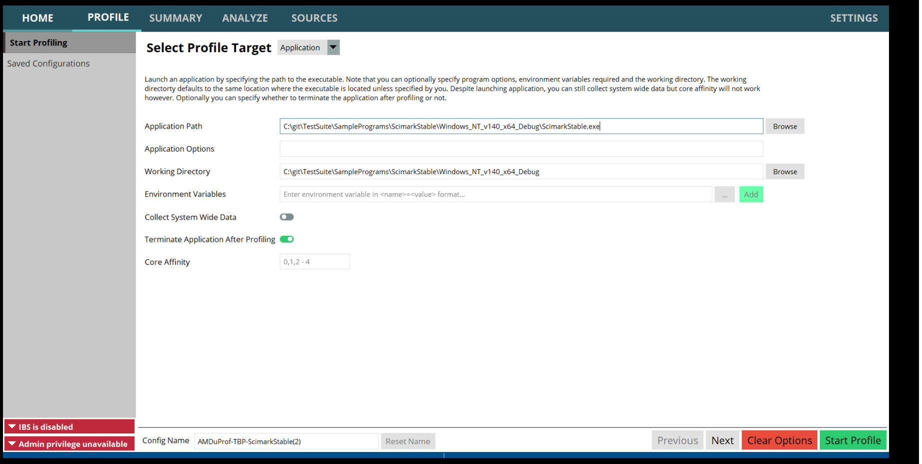
Offer developer choices – the profiler that best suites the need and development environment

- perf kernel – common profiler utility used to build custom profiler applications on Linux®
  - Enabled to reflect counters and events supported by latest AMD processors
  - PAPI is automatically supported given PERF kernel support
  - Tools built on PERF kernel driver or PAPI have the necessary support to work well on latest AMD processors
    - PERF tool (application)
    - PAPI-based tools like HPCTool kit etc
- AMD µProf offers a richer experience with AMD support
  - Intuitive graphical user interface and command line interface
  - Supporting Linux®, Windows® and FreeBSD
  - Supports performance monitoring recipes – data from set of events and associated calculation around them

AMD µProf Profiler Overview

Measure and analyze the performance of an application or the entire system running Linux® or Windows®

- System Analysis
  - Monitors basic core, level 3 cache and data fabric performance metrics
- Application Analysis
  - CPU Profiling to identify runtime performance bottlenecks of an application or the entire system
- Power Profiling
  - Monitors thermal & power characteristics of system
- Energy Analysis
  - Identifies energy hotspots in the application



Select Profile Target Application

Launch an application by specifying the path to the executable. Note that you can optionally specify program options, environment variables required and the working directory. The working directorty defaults to the same location where the executable is located unless specified by you. Despite launching application, you can still collect system wide data but core affinity will not work however. Optionally you can specify whether to terminate the application after profiling or not.

Application Path: C:\git\TestSuite\SamplePrograms\ScimarkStable\Windows_NT_v140_x64_Debug\ScimarkStable.exe

Application Options:

Working Directory: C:\git\TestSuite\SamplePrograms\ScimarkStable\Windows_NT_v140_x64_Debug

Environment Variables: Enter environment variable in <name>=<value> format...

Collect System Wide Data: ○

Terminate Application After Profiling: ●

Core Affinity: 0,1,2 - 4

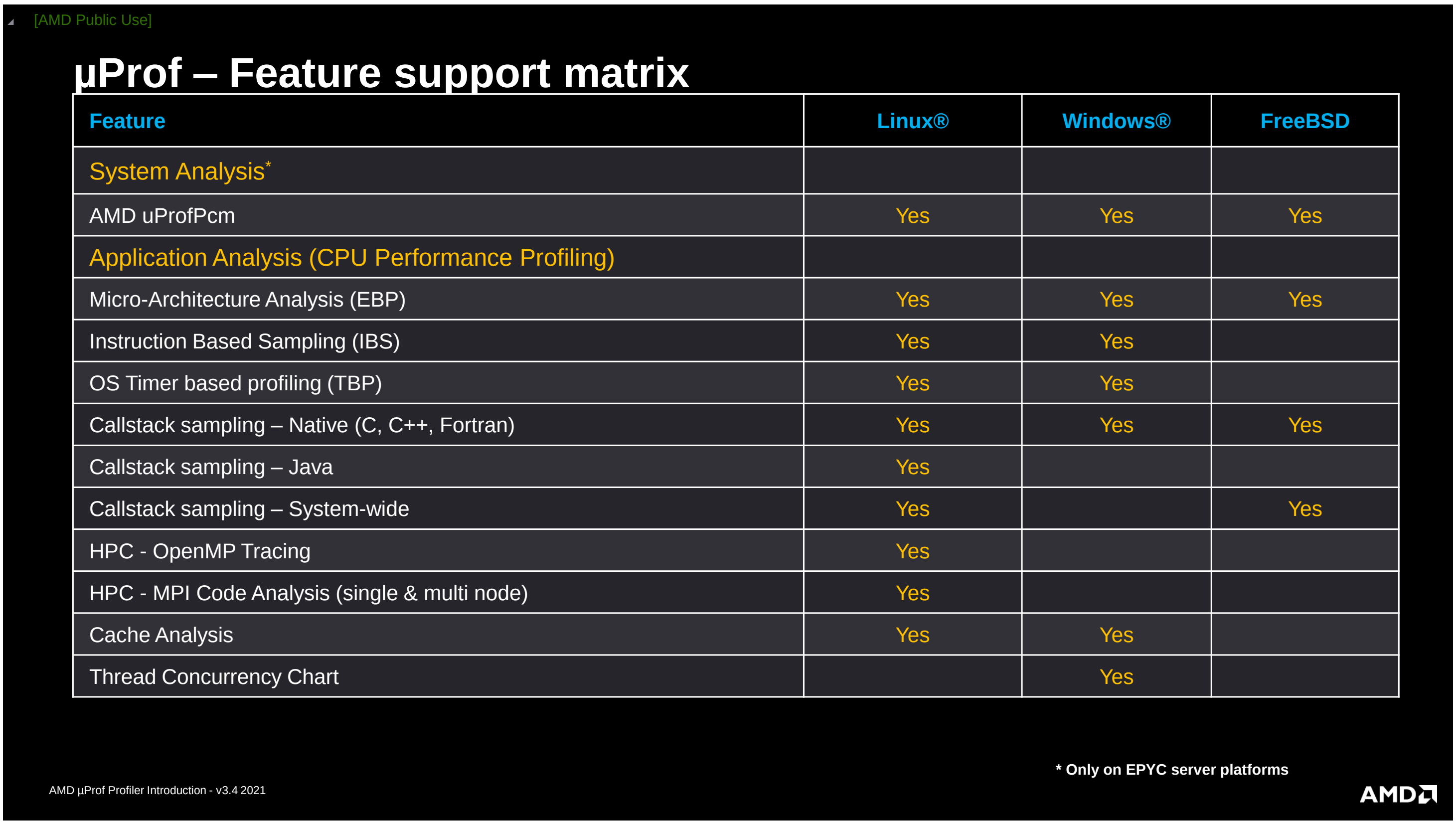
IBS is disabled

Admin privilege unavailable

Config Name: AMDuProf-TBP-ScimarkStable(2)

Broad AMD µProf 3.4 support of Operating Systems & and Compilers

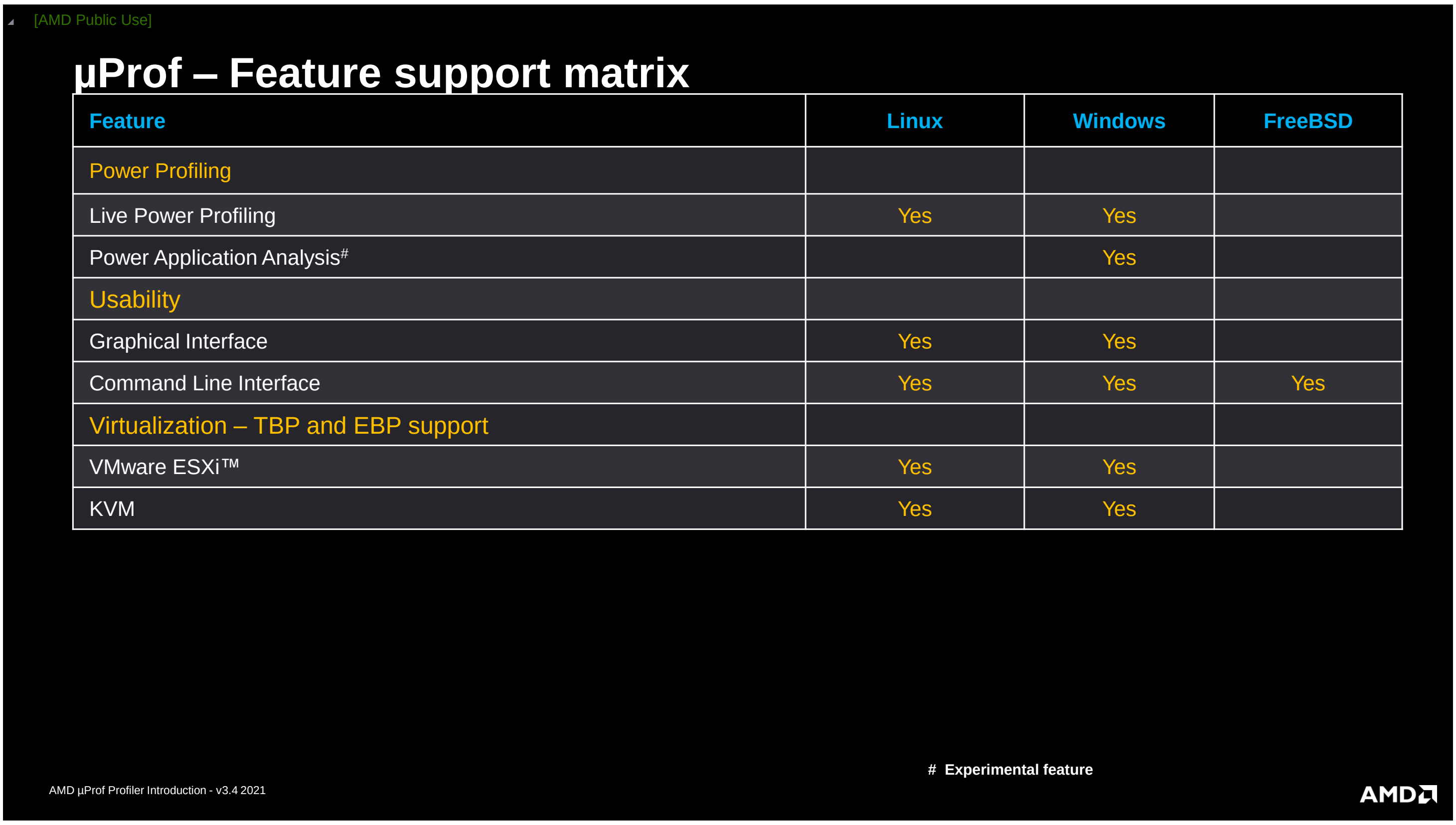
Component	Supported Version	Languages
OpenMP® Spec	• OpenMP® v5.0	
Compiler	• LLVM™ 8 - 12	• C, C++
	• AOCC 2.x, 3.0	• C, C++, Fortran
	• Intel® Compiler Collection (ICC) 19.1	• C, C++, Fortran
OS	• Ubuntu® 18.04 LTS • Ubuntu® 20.04 LTS • Red Hat® Enterprise Linux® 8.x • CentOS™ 8.x • Windows® 10 thru 20H2 • Windows Server® 2019	



µProf – Feature support matrix

Feature	Linux®	Windows®	FreeBSD
System Analysis*			
AMD uProfPcm	Yes	Yes	Yes
Application Analysis (CPU Performance Profiling)			
Micro-Architecture Analysis (EBP)	Yes	Yes	Yes
Instruction Based Sampling (IBS)	Yes	Yes	
OS Timer based profiling (TBP)	Yes	Yes	
Callstack sampling – Native (C, C++, Fortran)	Yes	Yes	Yes
Callstack sampling – Java	Yes		
Callstack sampling – System-wide	Yes		Yes
HPC - OpenMP Tracing	Yes		
HPC - MPI Code Analysis (single & multi node)	Yes		
Cache Analysis	Yes	Yes	
Thread Concurrency Chart		Yes	

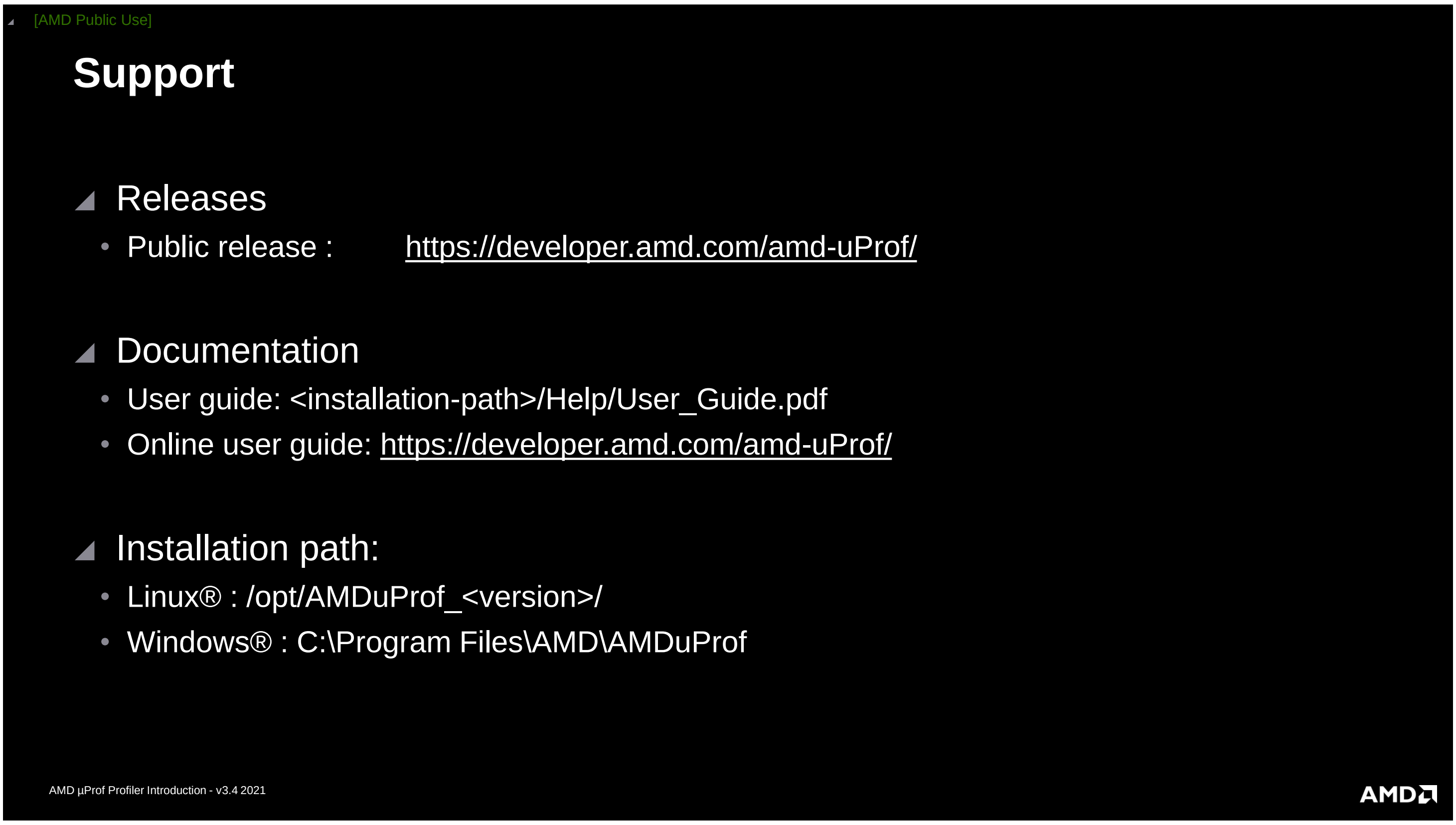
*** Only on EPYC server platforms**



µProf – Feature support matrix

Feature	Linux	Windows	FreeBSD
Power Profiling			
Live Power Profiling	Yes	Yes	
Power Application Analysis#		Yes	
Usability			
Graphical Interface	Yes	Yes	
Command Line Interface	Yes	Yes	Yes
Virtualization – TBP and EBP support			
VMware ESXi™	Yes	Yes	
KVM	Yes	Yes	

# Experimental feature

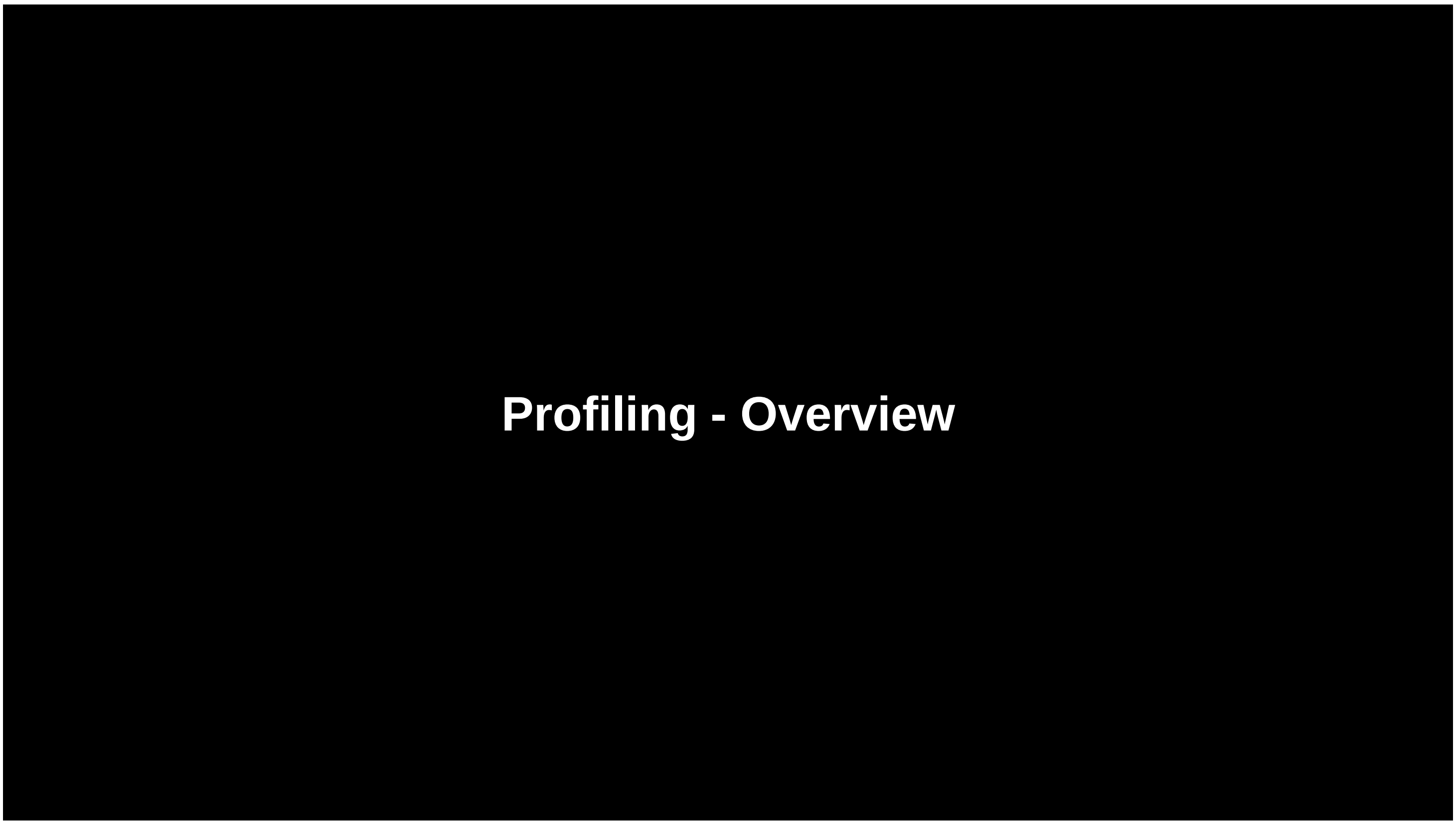


Support

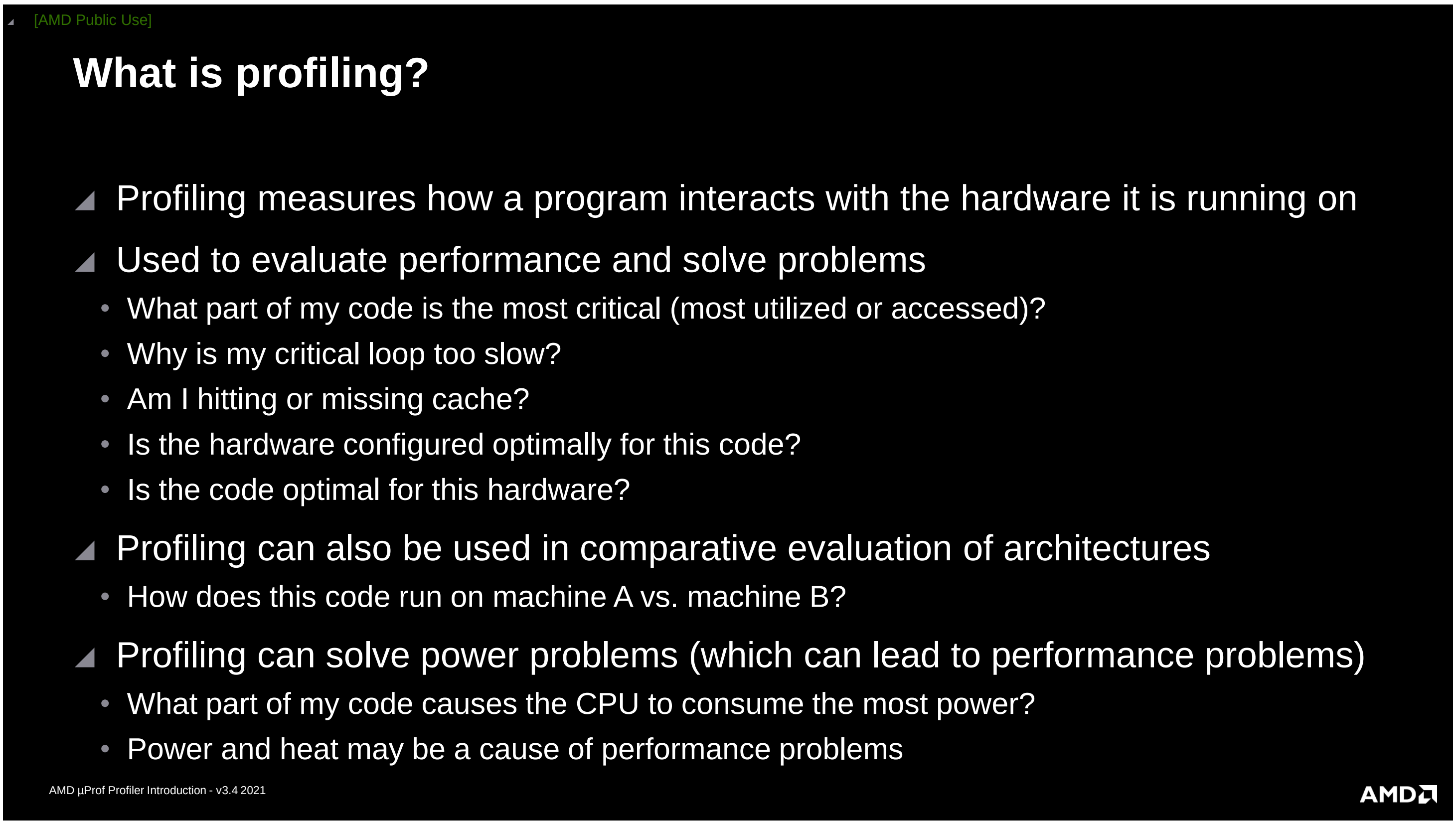
- Releases
  - Public release : https://developer.amd.com/amd-uProf/

- Documentation
  - User guide: <installation-path>/Help/User_Guide.pdf
  - Online user guide: https://developer.amd.com/amd-uProf/

- Installation path:
  - Linux® : /opt/AMDuProf_<version>/
  - Windows® : C:\Program Files\AMD\AMDuProf



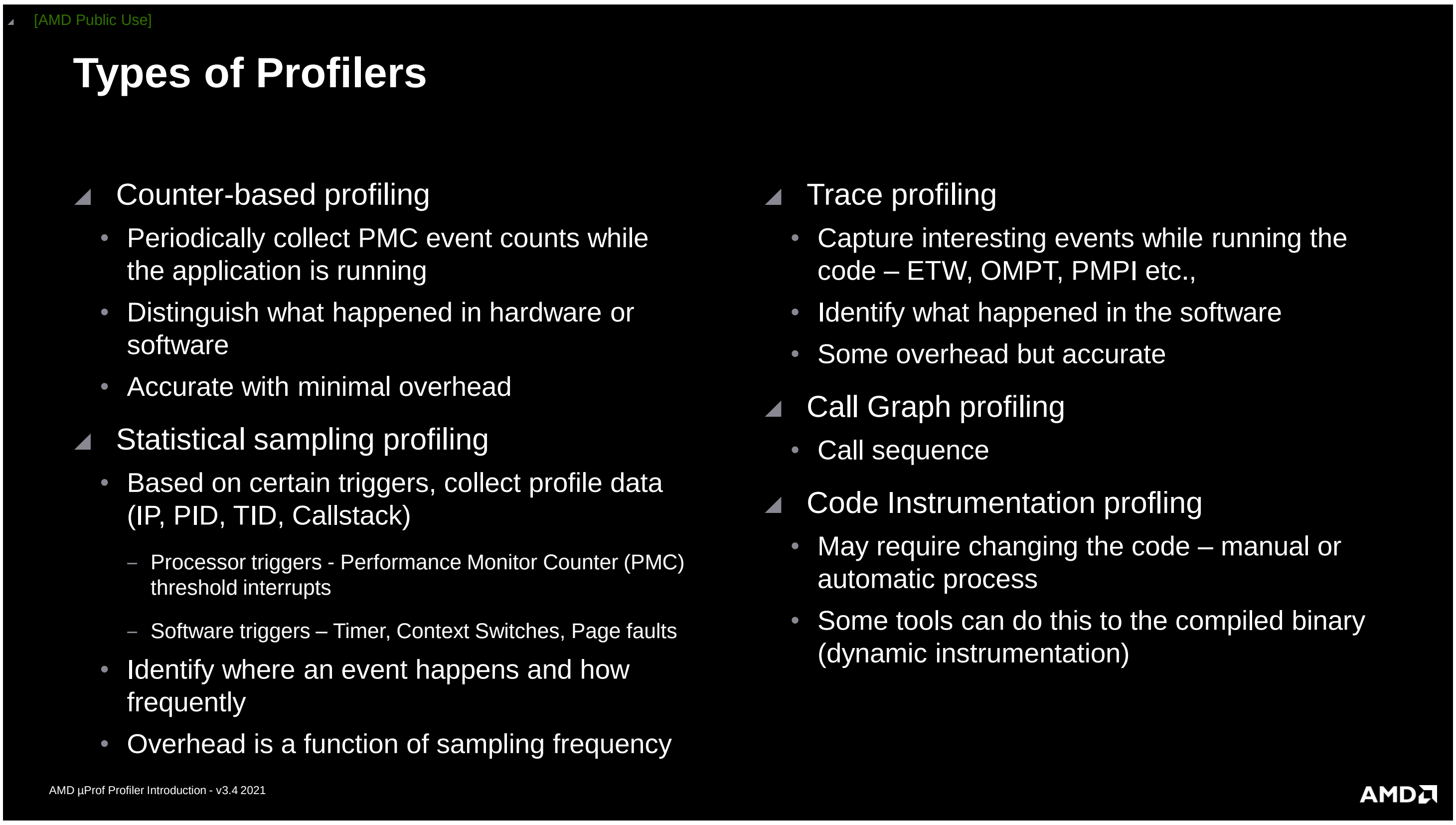
Profiling - Overview



[AMD Public Use]

What is profiling?

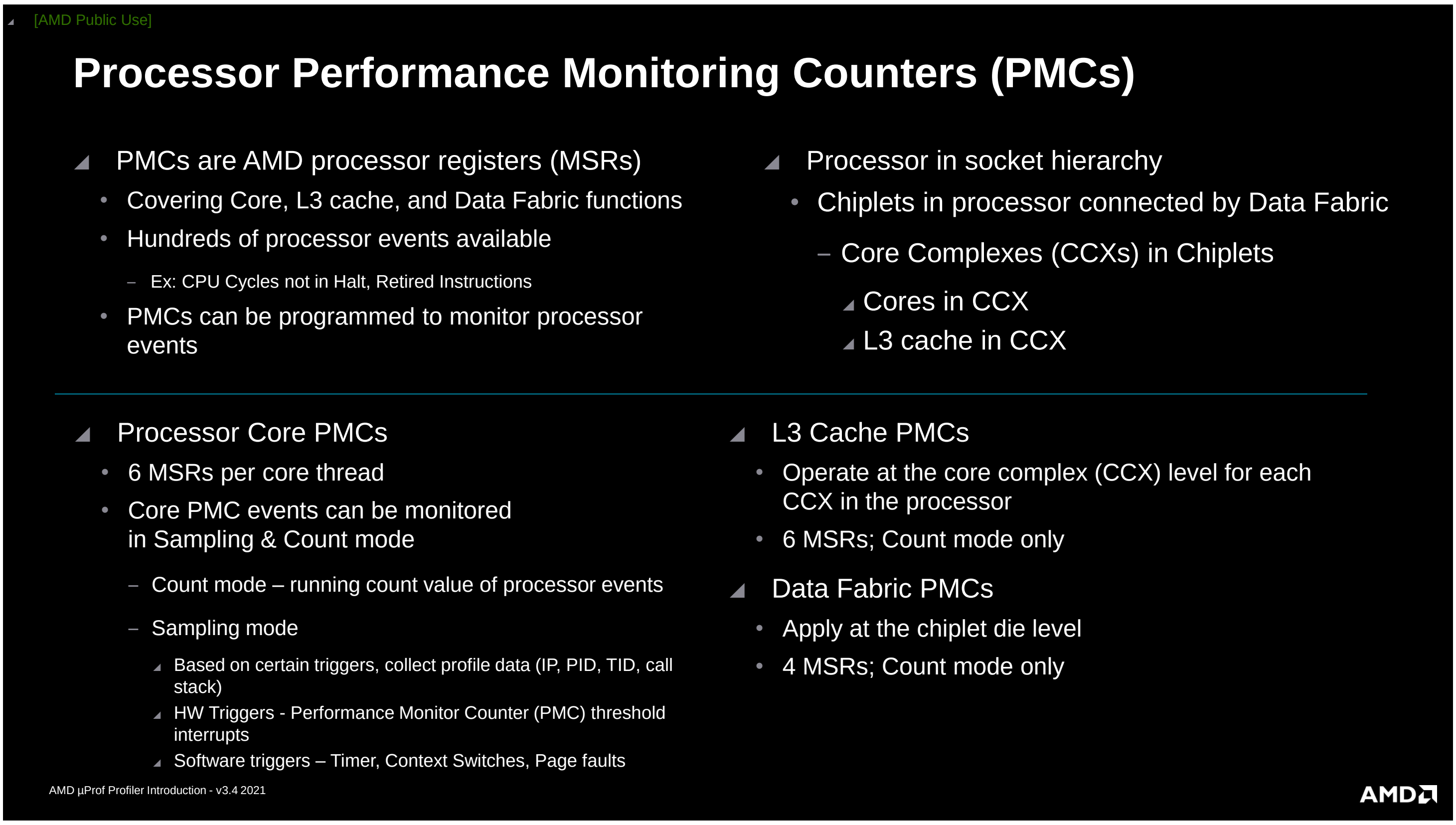
- Profiling measures how a program interacts with the hardware it is running on
- Used to evaluate performance and solve problems
  - What part of my code is the most critical (most utilized or accessed)?
  - Why is my critical loop too slow?
  - Am I hitting or missing cache?
  - Is the hardware configured optimally for this code?
  - Is the code optimal for this hardware?
- Profiling can also be used in comparative evaluation of architectures
  - How does this code run on machine A vs. machine B?
- Profiling can solve power problems (which can lead to performance problems)
  - What part of my code causes the CPU to consume the most power?
  - Power and heat may be a cause of performance problems



[AMD Public Use]

Types of Profilers

- Counter-based profiling
  - Periodically collect PMC event counts while the application is running
  - Distinguish what happened in hardware or software
  - Accurate with minimal overhead
- Statistical sampling profiling
  - Based on certain triggers, collect profile data (IP, PID, TID, Callstack)
    - Processor triggers - Performance Monitor Counter (PMC) threshold interrupts
    - Software triggers – Timer, Context Switches, Page faults
  - Identify where an event happens and how frequently
  - Overhead is a function of sampling frequency
- Trace profiling
  - Capture interesting events while running the code – ETW, OMPT, PMPI etc.,
  - Identify what happened in the software
  - Some overhead but accurate
- Call Graph profiling
  - Call sequence
- Code Instrumentation profling
  - May require changing the code – manual or automatic process
  - Some tools can do this to the compiled binary (dynamic instrumentation)



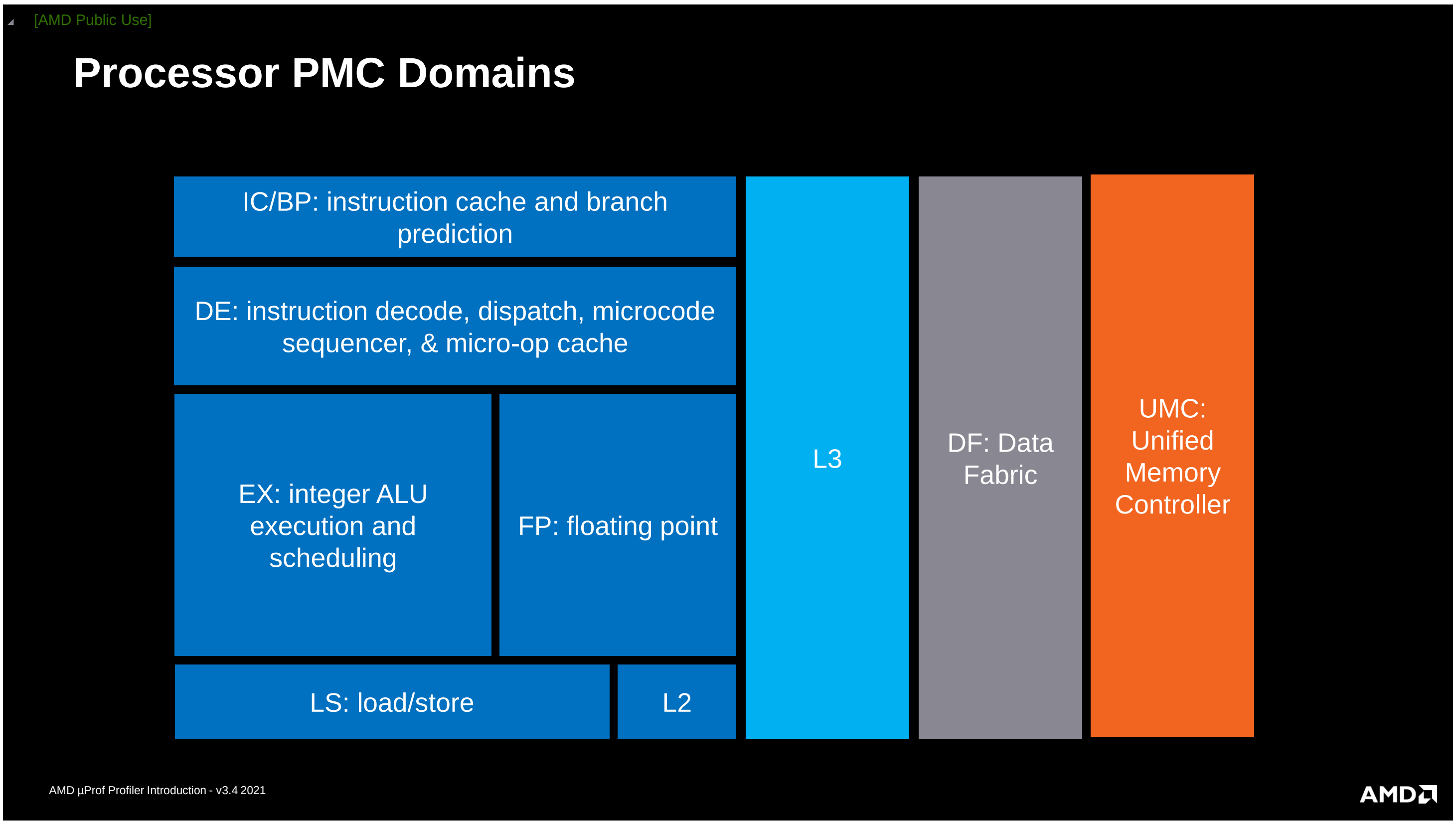
Processor Performance Monitoring Counters (PMCs)

- PMCs are AMD processor registers (MSRs)
  - Covering Core, L3 cache, and Data Fabric functions
  - Hundreds of processor events available
    - Ex: CPU Cycles not in Halt, Retired Instructions
  - PMCs can be programmed to monitor processor events

- Processor in socket hierarchy
  - Chiplets in processor connected by Data Fabric
    - Core Complexes (CCXs) in Chiplets
      - Cores in CCX
      - L3 cache in CCX

- Processor Core PMCs
  - 6 MSRs per core thread
  - Core PMC events can be monitored in Sampling & Count mode
    - Count mode – running count value of processor events
    - Sampling mode
      - Based on certain triggers, collect profile data (IP, PID, TID, call stack)
      - HW Triggers - Performance Monitor Counter (PMC) threshold interrupts
      - Software triggers – Timer, Context Switches, Page faults

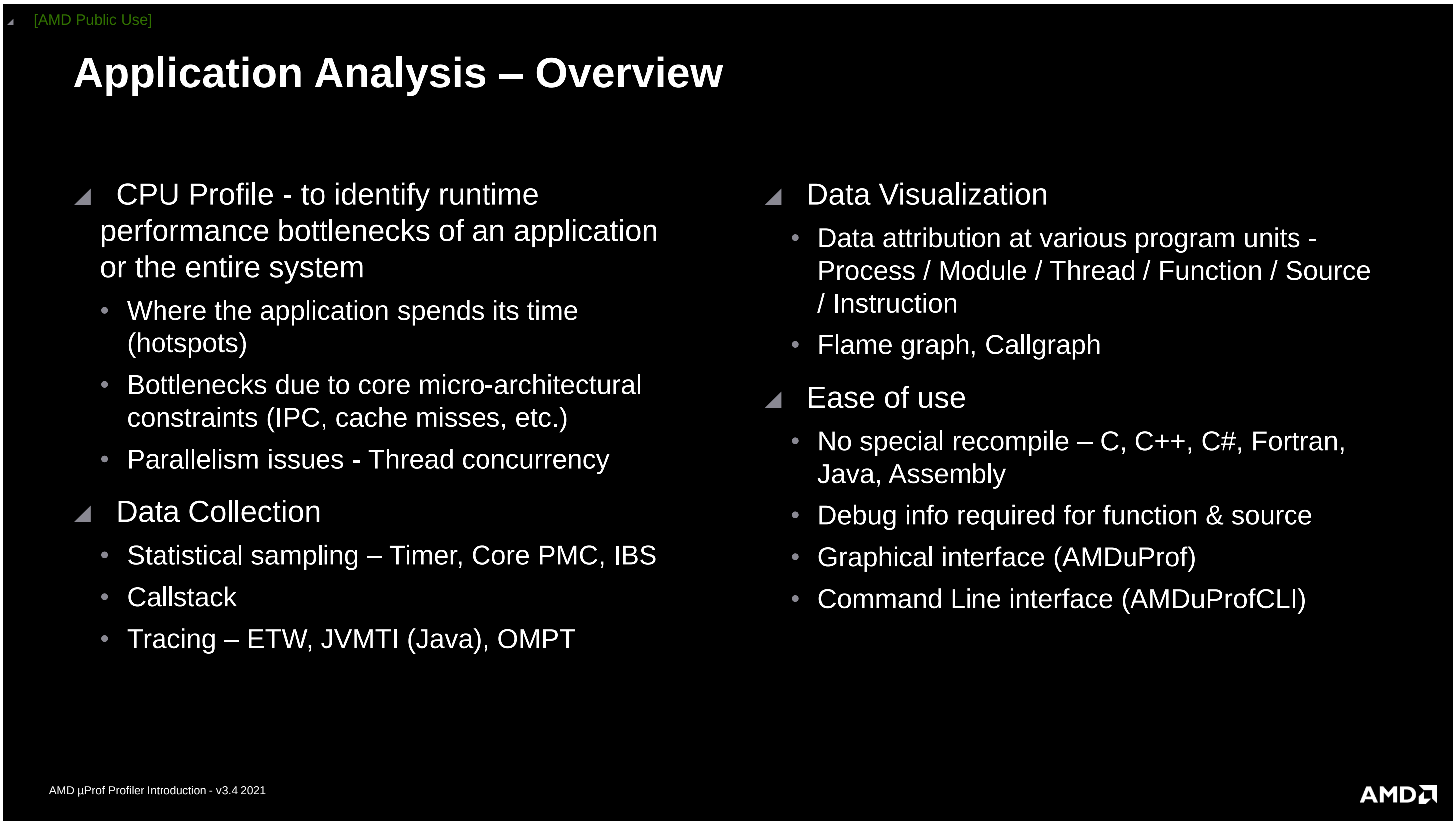
- L3 Cache PMCs
  - Operate at the core complex (CCX) level for each CCX in the processor
  - 6 MSRs; Count mode only
- Data Fabric PMCs
  - Apply at the chiplet die level
  - 4 MSRs; Count mode only



Processor PMC Domains

IC/BP: instruction cache and branch prediction
DE: instruction decode, dispatch, microcode sequencer, & micro-op cache
EX: integer ALU execution and scheduling
FP: floating point
LS: load/store
L2
L3
DF: Data Fabric
UMC: Unified Memory Controller

Application Analysis



Application Analysis – Overview

- CPU Profile - to identify runtime performance bottlenecks of an application or the entire system
  - Where the application spends its time (hotspots)
  - Bottlenecks due to core micro-architectural constraints (IPC, cache misses, etc.)
  - Parallelism issues - Thread concurrency
- Data Collection
  - Statistical sampling – Timer, Core PMC, IBS
  - Callstack
  - Tracing – ETW, JVMTI (Java), OMPT
- Data Visualization
  - Data attribution at various program units - Process / Module / Thread / Function / Source / Instruction
  - Flame graph, Callgraph
- Ease of use
  - No special recompile – C, C++, C#, Fortran, Java, Assembly
  - Debug info required for function & source
  - Graphical interface (AMDuProf)
  - Command Line interface (AMDuProfCLI)

Application Analysis – Performance Data

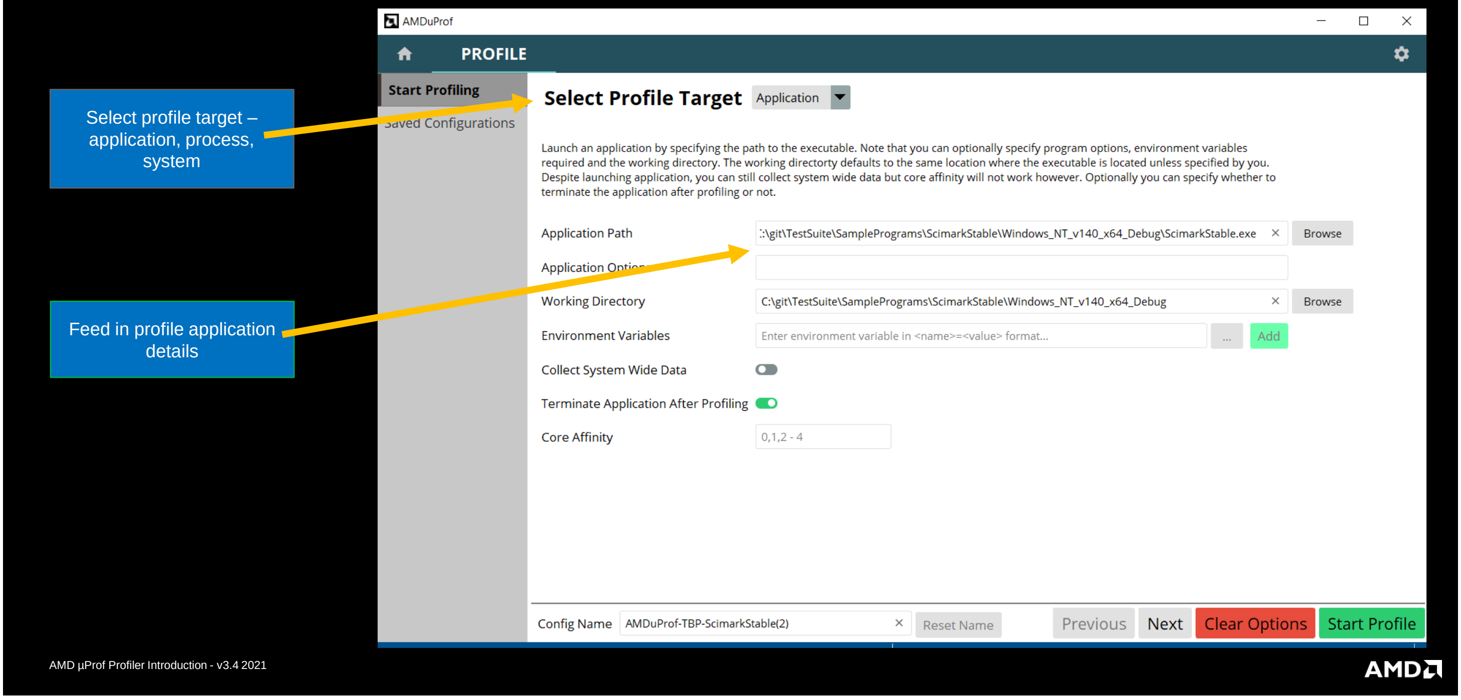
Primary data

- Basic hotspots - Timer based profiling (TBP)
  - Which functions consume most of time?
- Micro-architectural exploration - Core PMC Event based profiling (EBP)
  - Which functions consume most of the cycles?
  - Why - cache misses?, branch mispredictions?
- Memory access - Instruction Based Sampling (IBS)
  - Memory access
  - Potential false cache sharing
- HPC using OMPT
  - OpenMP® parallel region analysis

Secondary data

- Call graph
  - Call sequence
- Thread concurrency
  - Windows® only

Application Analysis – data collection



Select profile target – application, process, system

Feed in profile application details

Select Profile Target Application

Launch an application by specifying the path to the executable. Note that you can optionally specify program options, environment variables required and the working directory. The working directorty defaults to the same location where the executable is located unless specified by you. Despite launching application, you can still collect system wide data but core affinity will not work however. Optionally you can specify whether to terminate the application after profiling or not.

Application Path: :\git\TestSuite\SamplePrograms\ScimarkStable\Windows_NT_v140_x64_Debug\ScimarkStable.exe

Application Options:

Working Directory: C:\git\TestSuite\SamplePrograms\ScimarkStable\Windows_NT_v140_x64_Debug

Environment Variables: Enter environment variable in <name>=<value> format...

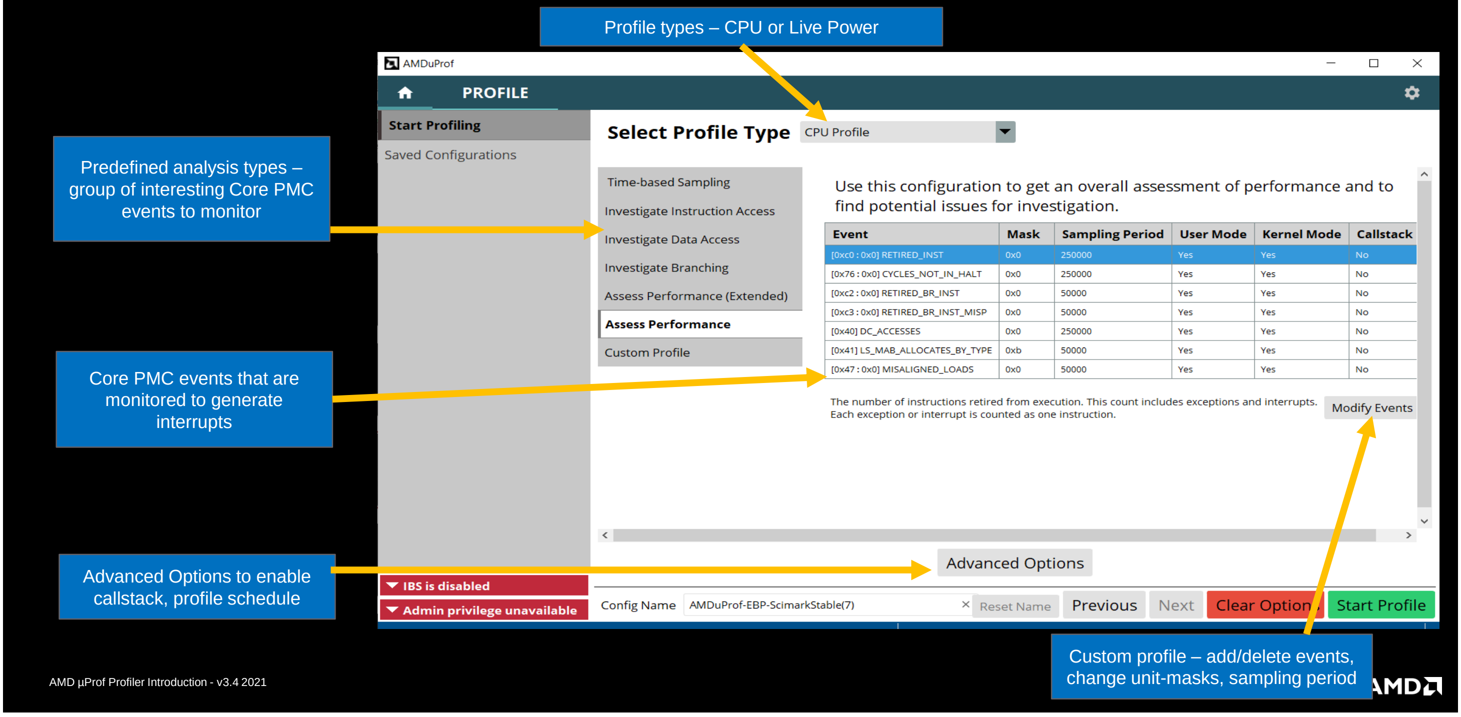
Collect System Wide Data: ○

Terminate Application After Profiling: ●

Core Affinity: 0,1,2 - 4

Config Name: AMDuProf-TBP-ScimarkStable(2)

Application analysis – data collection



Profile types – CPU or Live Power

Predefined analysis types – group of interesting Core PMC events to monitor

Core PMC events that are monitored to generate interrupts

Advanced Options to enable callstack, profile schedule

Custom profile – add/delete events, change unit-masks, sampling period

Use this configuration to get an overall assessment of performance and to find potential issues for investigation.

Event	Mask	Sampling Period	User Mode	Kernel Mode	Callstack
[0xc0 : 0x0] RETIRED_INST	0x0	250000	Yes	Yes	No
[0x76 : 0x0] CYCLES_NOT_IN_HALT	0x0	250000	Yes	Yes	No
[0xc2 : 0x0] RETIRED_BR_INST	0x0	50000	Yes	Yes	No
[0xc3 : 0x0] RETIRED_BR_INST_MISP	0x0	50000	Yes	Yes	No
[0x40] DC_ACCESSES	0x0	250000	Yes	Yes	No
[0x41] LS_MAB_ALLOCATES_BY_TYPE	0xb	50000	Yes	Yes	No
[0x47 : 0x0] MISALIGNED_LOADS	0x0	50000	Yes	Yes	No

The number of instructions retired from execution. This count includes exceptions and interrupts. Each exception or interrupt is counted as one instruction.

Application Analysis – data collection (CLI)

```
Collect assess performance data
$ AMDuProfCLI collect --config assess –o /tmp/namd-assess /tmp/run-namd.sh
Profile completed ...
Generated raw file : /tmp/namd-assess.caperf

Generate Report – this will create /tmp/namd-assess/namd-assess.db & /tmp/namd-
assess/namd-assess.csv
$ AMDuProfCLI report –i /tmp/namd-assess.caperf
Translation started ...
...
Generated report file : /tmp/namd-assess/namd-assess.csv

To only translate – this will create /tmp/namd-assess/namd-assess.db (import in GUI)
$ AMDuProfCLI translate –i /tmp/namd-assess.caperf
Translation started ...
...
Generated db file : /tmp/namd-assess/namd-assess.db

Importing
The rawfile collected or the processed db file can also be imported in GUI for further analysis
```

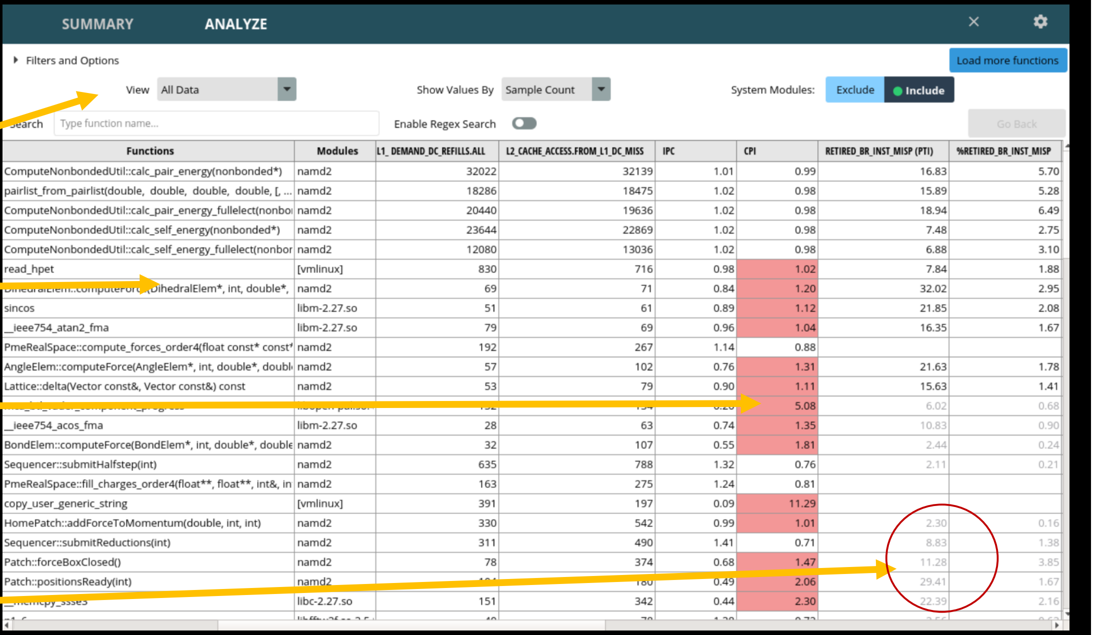
Application analysis – Function hotspots

Filters & Options
View: Select what metric to report;
Show data by: count or %;
Include or exclude system modules;

Double click on a function to view Source

Issue threshold – CPI > 1.0 will be highlighted

Low confidence level due to low number of samples collected – values will be grayed



View: All Data Show Values By: Sample Count System Modules: Exclude ● Include

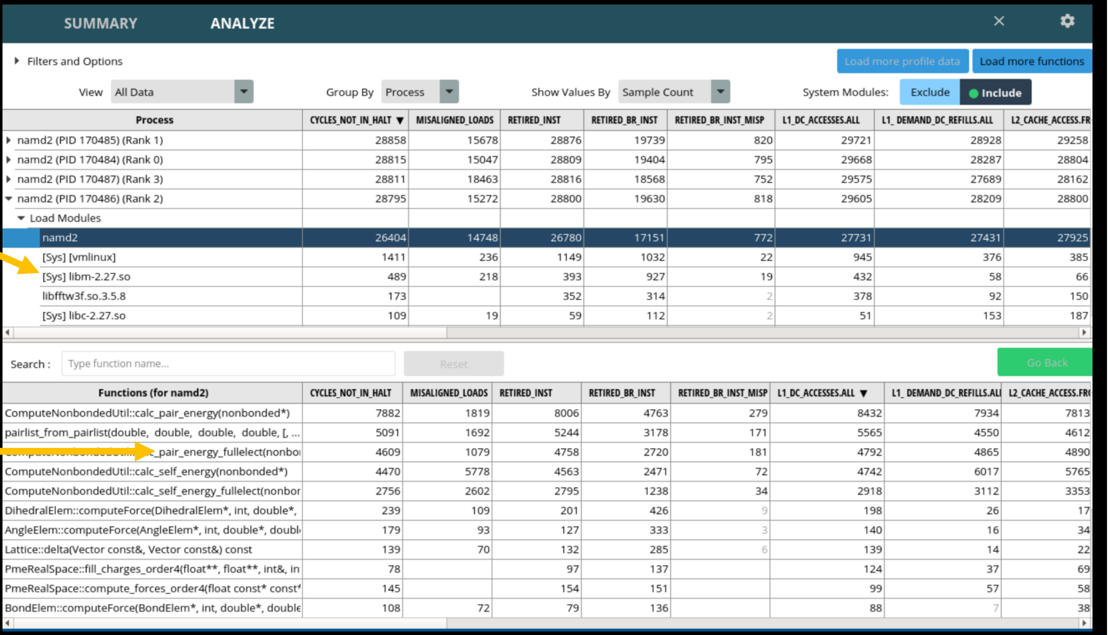
Search: Type function name... Enable Regex Search

Functions	Modules	L1_ DEMAND_DC_REFILLS.ALL	L2_CACHE_ACCESS.FROM_L1_DC_MISS	IPC	CPI	RETIRED_BR_INST_MISP (PTI)	%RETIRED_BR_INST_MISP
ComputeNonbondedUtil::calc_pair_energy(nonbonded*)	namd2	32022	32139	1.01	0.99	16.83	5.70
pairlist_from_pairlist(double, double, double, double, [, ...	namd2	18286	18475	1.02	0.98	15.89	5.28
ComputeNonbondedUtil::calc_pair_energy_fullelect(nonbo	namd2	20440	19636	1.02	0.98	18.94	6.49
ComputeNonbondedUtil::calc_self_energy(nonbonded*)	namd2	23644	22869	1.02	0.98	7.48	2.75
ComputeNonbondedUtil::calc_self_energy_fullelect(nonbor	namd2	12080	13036	1.02	0.98	6.88	3.10
read_hpet	[vmlinux]	830	716	0.98	1.02	7.84	1.88
DihedralElem::computeForce(DihedralElem*, int, double*,	namd2	69	71	0.84	1.20	32.02	2.95
sincos	libm-2.27.so	51	61	0.89	1.12	21.85	2.08
__ieee754_atan2_fma	libm-2.27.so	79	69	0.96	1.04	16.35	1.67
PmeRealSpace::compute_forces_order4(float const* const*	namd2	192	267	1.14	0.88		
AngleElem::computeForce(AngleElem*, int, double*, doubl	namd2	57	102	0.76	1.31	21.63	1.78
Lattice::delta(Vector const&, Vector const&) const	namd2	53	79	0.90	1.11	15.63	1.41
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	5.08	6.02	0.68
__ieee754_acos_fma	libm-2.27.so	28	63	0.74	1.35	10.83	0.90
BondElem::computeForce(BondElem*, int, double*, double	namd2	32	107	0.55	1.81	2.44	0.24
Sequencer::submitHalfstep(int)	namd2	635	788	1.32	0.76	2.11	0.21
PmeRealSpace::fill_charges_order4(float**, float**, int&, in	namd2	163	275	1.24	0.81		
copy_user_generic_string	[vmlinux]	391	197	0.09	11.29		
HomePatch::addForceToMomentum(double, int, int)	namd2	330	542	0.99	1.01	2.30	0.16
Sequencer::submitReductions(int)	namd2	311	490	1.41	0.71	8.83	1.38
Patch::forceBoxClosed()	namd2	78	374	0.68	1.47	11.28	3.85
Patch::positionsReady(int)	namd2	[illegible]	[illegible]	0.49	2.06	29.41	1.67
__memcpy_ssse3	libc-2.27.so	151	342	0.44	2.30	22.39	2.16

Application analysis – Analyze

Program units – load modules and threads

Hot functions for the selected program unit;
Double click function to view Source



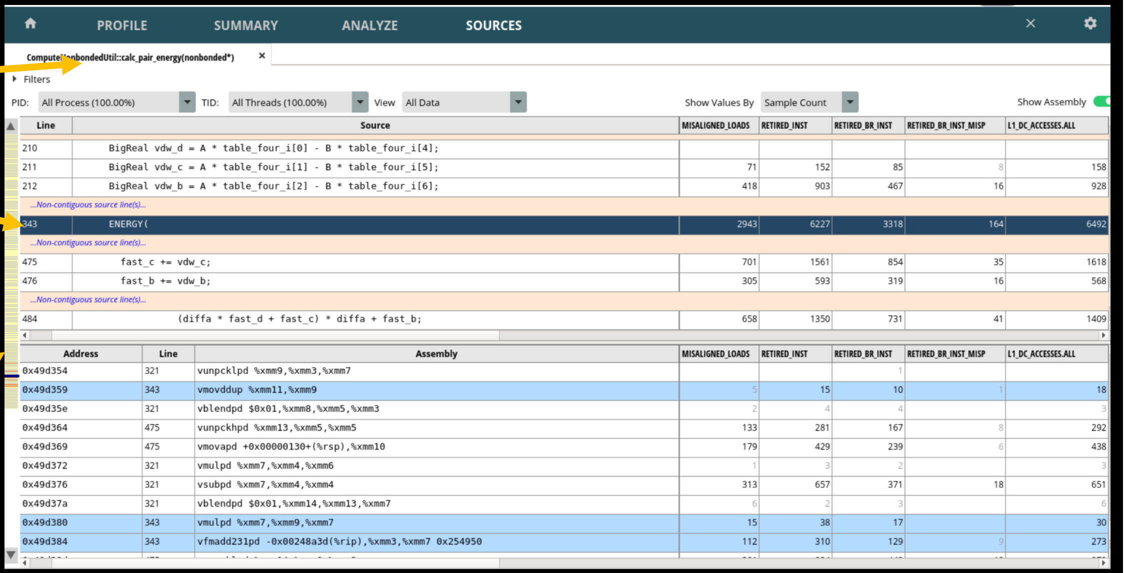
View: All Data Group By: Process Show Values By: Sample Count System Modules: Exclude ● Include

Process	CYCLES_NOT_IN_HALT ▼	MISALIGNED_LOADS	RETIRED_INST	RETIRED_BR_INST	RETIRED_BR_INST_MISP	L1_DC_ACCESSES.ALL	L1_ DEMAND_DC_REFILLS.ALL	L2_CACHE_ACCESS.FR
namd2 (PID 170485) (Rank 1)	28858	15678	28876	19739	820	29721	28928	29258
namd2 (PID 170484) (Rank 0)	28815	15047	28809	19404	795	29668	28287	28804
namd2 (PID 170487) (Rank 3)	28811	18463	28816	18568	752	29575	27689	28162
namd2 (PID 170486) (Rank 2)	28795	15272	28800	19630	818	29605	28209	28800
Load Modules								
namd2	26404	14748	26780	17151	772	27731	27431	27925
[Sys] [vmlinux]	1411	236	1149	1032	22	945	376	385
[Sys] libm-2.27.so	489	218	393	927	19	432	58	66
libfftw3f.so.3.5.8	173		352	314	2	378	92	150
[Sys] libc-2.27.so	109	19	59	112	2	51	153	187

Search : Type function name...

Functions (for namd2)	CYCLES_NOT_IN_HALT	MISALIGNED_LOADS	RETIRED_INST	RETIRED_BR_INST	RETIRED_BR_INST_MISP	L1_DC_ACCESSES.ALL ▼	L1_ DEMAND_DC_REFILLS.ALL	L2_CACHE_ACCESS.FR
ComputeNonbondedUtil::calc_pair_energy(nonbonded*)	7882	1819	8006	4763	279	8432	7934	7813
pairlist_from_pairlist(double, double, double, double, [, ...	5091	1692	5244	3178	171	5565	4550	4612
ComputeNonbondedUtil::calc_pair_energy_fullelect(nonbo	4609	1079	4758	2720	181	4792	4865	4890
ComputeNonbondedUtil::calc_self_energy(nonbonded*)	4470	5778	4563	2471	72	4742	6017	5765
ComputeNonbondedUtil::calc_self_energy_fullelect(nonbor	2756	2602	2795	1238	34	2918	3112	3353
DihedralElem::computeForce(DihedralElem*, int, double*,	239	109	201	426	9	198	26	17
AngleElem::computeForce(AngleElem*, int, double*, doubl	179	93	127	333	3	140	16	34
Lattice::delta(Vector const&, Vector const&) const	139	70	132	285	6	139	14	22
PmeRealSpace::fill_charges_order4(float**, float**, int&, in	78		97	137		124	37	69
PmeRealSpace::compute_forces_order4(float const* const*	145		154	151		99	57	58
BondElem::computeForce(BondElem*, int, double*, double	108	72	79	136		88	7	38

Application analysis – Source view



ComputeNonbondedUtil::calc_pair_energy(nonbonded*)

PID: All Process (100.00%) TID: All Threads (100.00%) View All Data

Show Values By Sample Count

Show Assembly

Line	Source	MISALIGNED_LOADS	RETIRED_INST	RETIRED_BR_INST	RETIRED_BR_INST_MISP	L1_DC_ACCESSES.ALL
210	`BigReal vdw_d = A * table_four_i[0] - B * table_four_i[4];`					
211	`BigReal vdw_c = A * table_four_i[1] - B * table_four_i[5];`	71	152	85	8	158
212	`BigReal vdw_b = A * table_four_i[2] - B * table_four_i[6];`	418	903	467	16	928
	...Non-contiguous source line(s)...					
343	ENERGY (	2943	6227	3318	164	6492
	...Non-contiguous source line(s)...					
475	`fast_c += vdw_c;`	701	1561	854	35	1618
476	`fast_b += vdw_b;`	305	593	319	16	568
	...Non-contiguous source line(s)...					
484	`(diffa * fast_d + fast_c) * diffa + fast_b;`	658	1350	731	41	1409

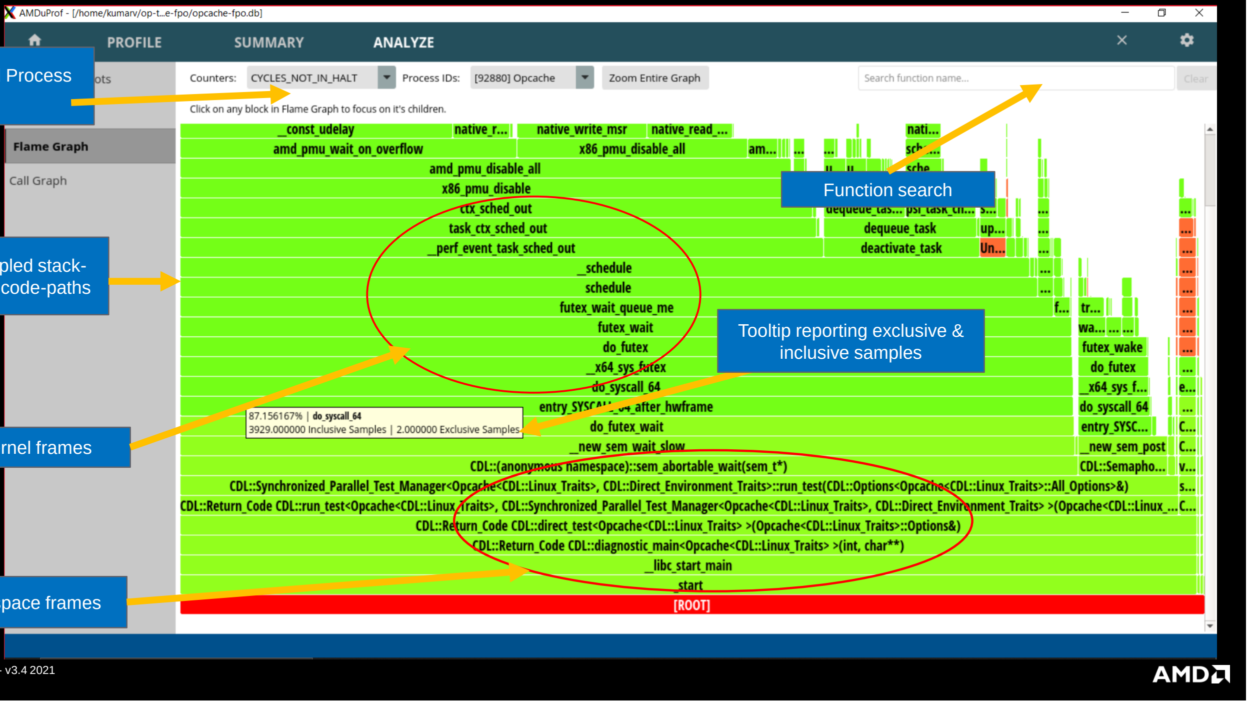
Address	Line	Assembly	MISALIGNED_LOADS	RETIRED_INST	RETIRED_BR_INST	RETIRED_BR_INST_MISP	L1_DC_ACCESSES.ALL
0x49d354	321	`vunpcklpd %xmm9,%xmm3,%xmm7`			1		
0x49d359	343	`vmovddup %xmm11,%xmm9`	5	15	10	1	18
0x49d35e	321	`vblendpd $0x01,%xmm8,%xmm5,%xmm3`	2	4	4		3
0x49d364	475	`vunpckhpd %xmm13,%xmm5,%xmm5`	133	281	167	8	292
0x49d369	475	`vmovapd +0x00000130+(%rsp),%xmm10`	179	429	239	6	438
0x49d372	321	`vmulpd %xmm7,%xmm4,%xmm6`	1	3	2		3
0x49d376	321	`vsubpd %xmm7,%xmm4,%xmm4`	313	657	371	18	651
0x49d37a	321	`vblendpd $0x01,%xmm14,%xmm13,%xmm7`	6	2	3		6
0x49d380	343	`vmulpd %xmm7,%xmm9,%xmm7`	15	38	17		30
0x49d384	343	`vfmadd231pd -0x00248a3d(%rip),%xmm3,%xmm7 0x254950`	112	310	129	9	273

Filter by Process and Thread

Select source line to highlight corresponding assembly

Heatmap – overview of hotspots

Callstack – Combined User & Kernel Callstack (Linux®)


AMDuProf - [/home/kumarv/op-t...e-fpo/opcache-fpo.db]
PROFILE
SUMMARY
ANALYZE
ots
Flame Graph
Call Graph
Zoom Entire Graph
Search function name...
Clear
_const_udelay
native_r...
native_write_msr
native_read_...
nati...
amd_pmu_wait_on_overflow
x86_pmu_disable_all
am...
sch...
amd_pmu_disable_all
sche
x86_pmu_disable
ctx_sched_out
dequeue_tas...
psi_task_ch...
s...
task_ctx_sched_out
dequeue_task
up...
_perf_event_task_sched_out
deactivate_task
Un...
_schedule
schedule
futex_wait_queue_me
f...
tr...
futex_wait
wa...
futex_wake
do_futex
_x64_sys_futex
_x64_sys_f...
e...
do_syscall_64
entry_SYSCALL_64_after_hwframe
do_futex_wait
entry_SYSC...
C...
_new_sem_wait_slow
_new_sem_post
CDL::(anonymous namespace)::sem_abortable_wait(sem_t*)
CDL::Semapho...
v...
CDL::Synchronized_Parallel_Test_Manager<Opcache<CDL::Linux_Traits>, CDL::Direct_Environment_Traits>::run_test(CDL::Options<Opcache<CDL::Linux_Traits>::All_Options>&)
CDL::Return_Code CDL::run_test<Opcache<CDL::Linux_Traits>, CDL::Synchronized_Parallel_Test_Manager<Opcache<CDL::Linux_Traits>, CDL::Direct_Environment_Traits> >(Opcache<CDL::Linux_...
CDL::Return_Code CDL::direct_test<Opcache<CDL::Linux_Traits> >(Opcache<CDL::Linux_Traits>::Options&)
CDL::Return_Code CDL::diagnostic_main<Opcache<CDL::Linux_Traits> >(int, char**)
_libc_start_main
start
[ROOT]

Counters: CYCLES_NOT_IN_HALT Process IDs: [92880] Opcache

Click on any block in Flame Graph to focus on it's children.

87.156167% | do_syscall_64
3929.000000 Inclusive Samples | 2.000000 Exclusive Samples

Sampling event and Process filtering

Function search

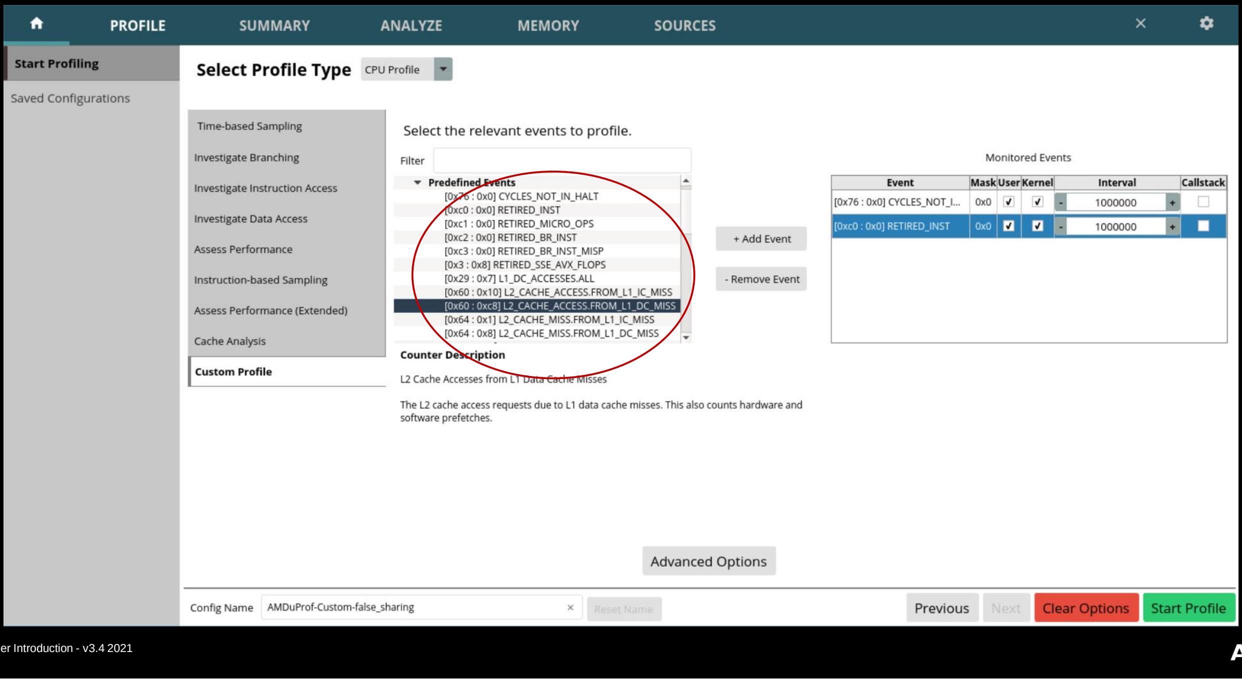
Visualization of sampled stack-traces to identify hot code-paths

Tooltip reporting exclusive & inclusive samples

Kernel frames

User space frames

Predefined Events



Select Profile Type: CPU Profile

Select the relevant events to profile.

Filter

Predefined Events
- [0x76 : 0x0] CYCLES_NOT_IN_HALT
- [0xc0 : 0x0] RETIRED_INST
- [0xc1 : 0x0] RETIRED_MICRO_OPS
- [0xc2 : 0x0] RETIRED_BR_INST
- [0xc3 : 0x0] RETIRED_BR_INST_MISP
- [0x3 : 0x8] RETIRED_SSE_AVX_FLOPS
- [0x29 : 0x7] L1_DC_ACCESSES.ALL
- [0x60 : 0x10] L2_CACHE_ACCESS.FROM_L1_IC_MISS
- [0x60 : 0xc8] L2_CACHE_ACCESS.FROM_L1_DC_MISS
- [0x64 : 0x1] L2_CACHE_MISS.FROM_L1_IC_MISS
- [0x64 : 0x8] L2_CACHE_MISS.FROM_L1_DC_MISS

Counter Description

L2 Cache Accesses from L1 Data Cache Misses

The L2 cache access requests due to L1 data cache misses. This also counts hardware and software prefetches.

Monitored Events

Event	Mask	User	Kernel	Interval	Callstack
[0x76 : 0x0] CYCLES_NOT_I...	0x0	☑	☑	1000000	□
[0xc0 : 0x0] RETIRED_INST	0x0	☑	☑	1000000	□

Config Name: AMDuProf-Custom-false_sharing

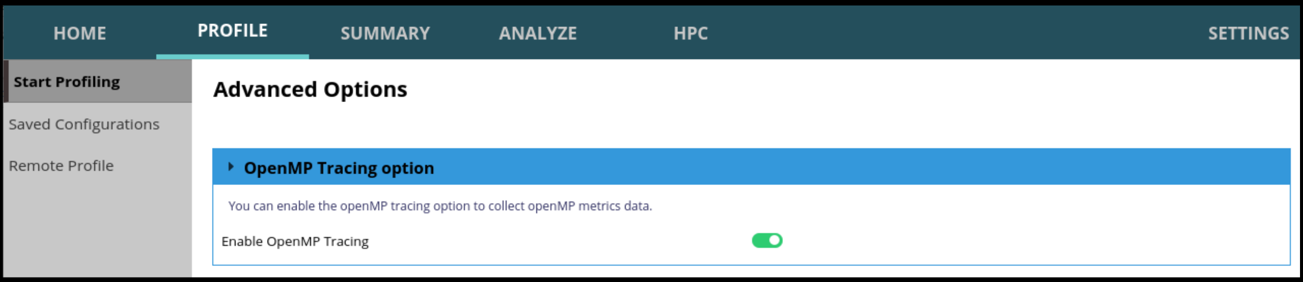
HPC Analysis

- When the threads execute the parallel region code, maximize CPU utilization.
- Due to several reasons the threads wait without doing useful work
  - Idle: A thread finishes it task within the parallel region and waits at the barrier for the other threads to complete.
  - Sync: If locks are used inside the parallel region, threads can wait on synchronization locks to acquire the shared resource.
  - Overhead: Thread management overhead.

- Analysis
  - Parallel Regions: List of all the parallel regions executed with associated metrics.
  - Region Detailed Analysis: thread timeline view – activity of all the threads in a parallel region.
    - Thread spending too much time on non work activity ?
    - Change scheduling, loop chunk size

HPC Analysis – Example

Data Collection



Advanced Options

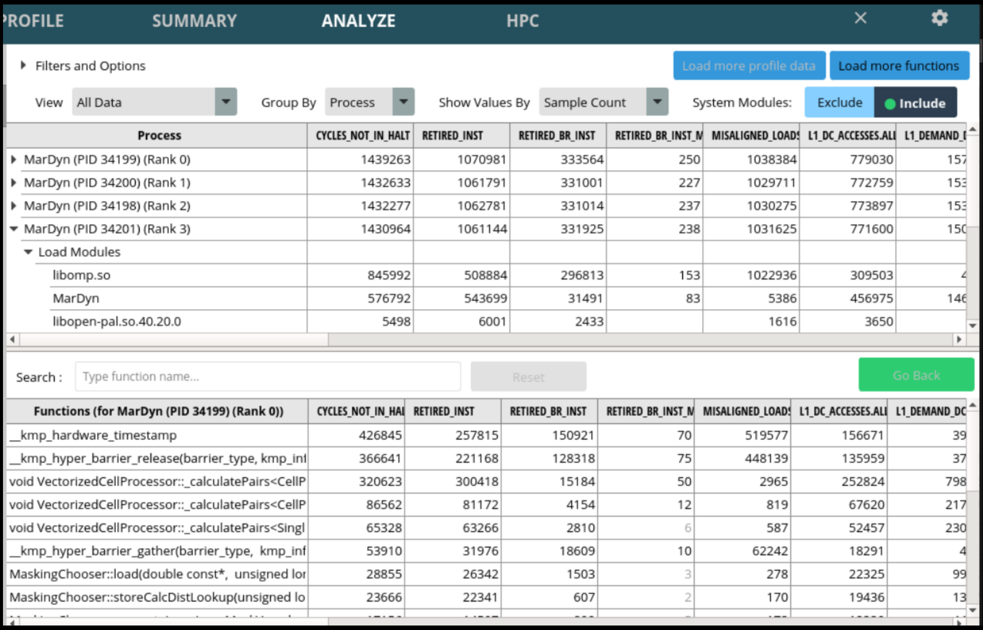
OpenMP Tracing option

You can enable the openMP tracing option to collect openMP metrics data.

Enable OpenMP Tracing ●

```
Collection run using CLI
$ AMDuProfCLI collect --omp --config tbp -o /tmp/myapp_perf <openmp-app>

Report Generation
$ AMDuProfCLI report -i /tmp/myapp_perf.caperf
```

HPC Analysis – Ex) Hotspots



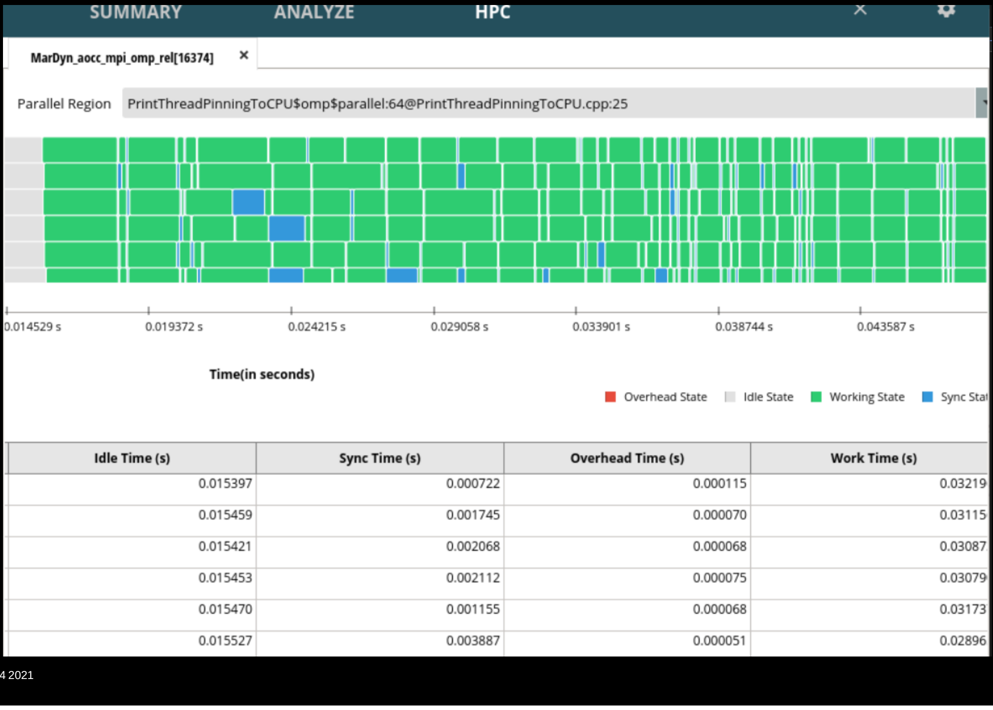
View: All Data | Group By: Process | Show Values By: Sample Count | System Modules: Exclude ● Include

Process	CYCLES_NOT_IN_HALT	RETIRED_INST	RETIRED_BR_INST	RETIRED_BR_INST_M	MISALIGNED_LOADS	L1_DC_ACCESSES.ALI	L1_DEMAND_I
MarDyn (PID 34199) (Rank 0)	1439263	1070981	333564	250	1038384	779030	157
MarDyn (PID 34200) (Rank 1)	1432633	1061791	331001	227	1029711	772759	153
MarDyn (PID 34198) (Rank 2)	1432277	1062781	331014	237	1030275	773897	153
MarDyn (PID 34201) (Rank 3)	1430964	1061144	331925	238	1031625	771600	150
Load Modules							
libomp.so	845992	508884	296813	153	1022936	309503	4
MarDyn	576792	543699	31491	83	5386	456975	146
libopen-pal.so.40.20.0	5498	6001	2433		1616	3650	

Search : Type function name...

Functions (for MarDyn (PID 34199) (Rank 0))	CYCLES_NOT_IN_HAI	RETIRED_INST	RETIRED_BR_INST	RETIRED_BR_INST_M	MISALIGNED_LOADS	L1_DC_ACCESSES.ALI	L1_DEMAND_DC
__kmp_hardware_timestamp	426845	257815	150921	70	519577	156671	39
__kmp_hyper_barrier_release(barrier_type, kmp_inf	366641	221168	128318	75	448139	135959	37
void VectorizedCellProcessor::_calculatePairs<CellP	320623	300418	15184	50	2965	252824	798
void VectorizedCellProcessor::_calculatePairs<CellP	86562	81172	4154	12	819	67620	217
void VectorizedCellProcessor::_calculatePairs<Singl	65328	63266	2810	6	587	52457	230
__kmp_hyper_barrier_gather(barrier_type, kmp_inf	53910	31976	18609	10	62242	18291	4
MaskingChooser::load(double const*, unsigned lor	28855	26342	1503	3	278	22325	99
MaskingChooser::storeCalcDistLookup(unsigned lo	23666	22341	607	2	170	19436	13

HPC Analysis – Ex) Thread State Timeline



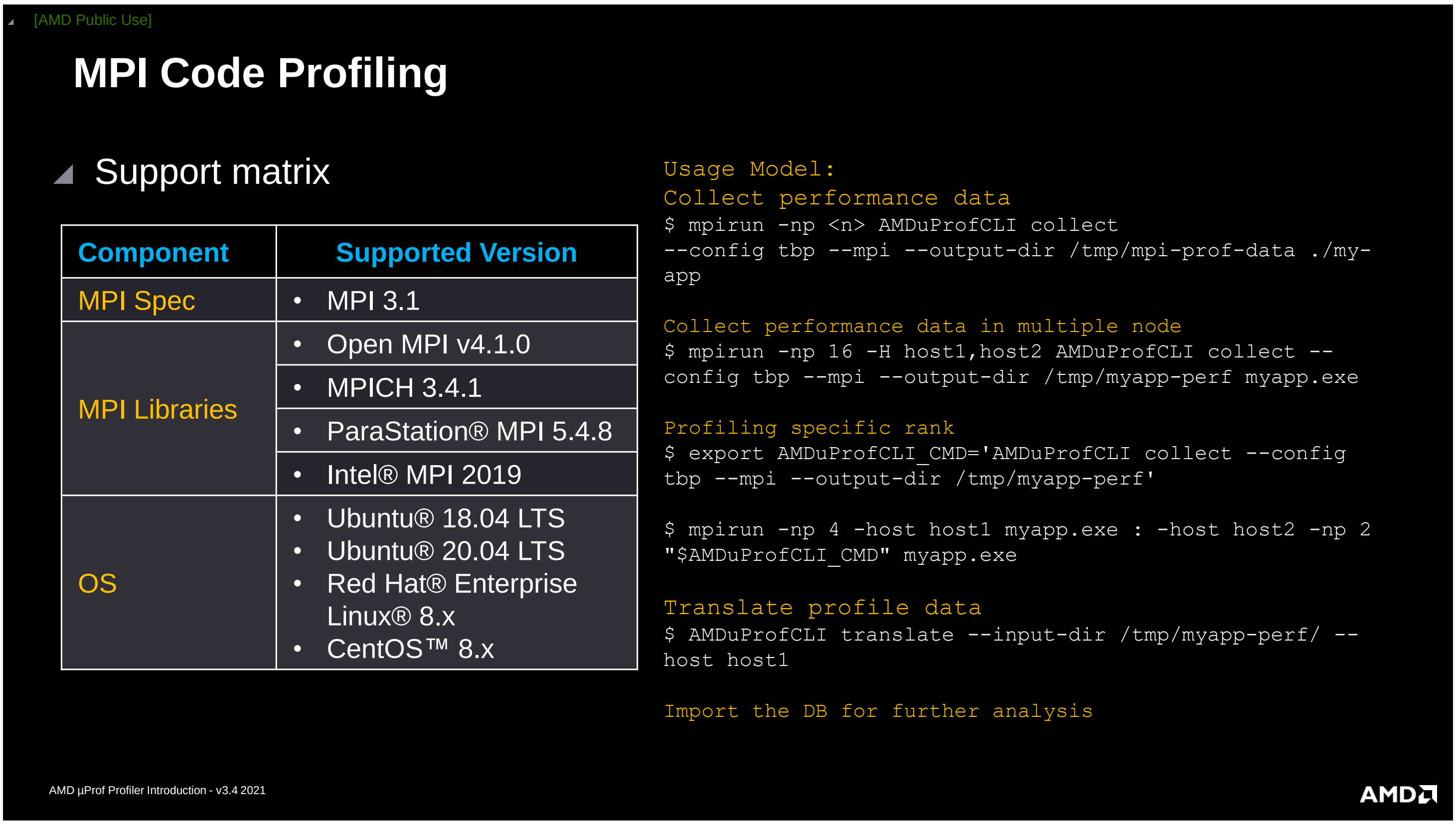
Parallel Region: PrintThreadPinningToCPU$omp$parallel:64@PrintThreadPinningToCPU.cpp:25

Idle Time (s)	Sync Time (s)	Overhead Time (s)	Work Time (s)
0.015397	0.000722	0.000115	0.03219
0.015459	0.001745	0.000070	0.03115
0.015421	0.002068	0.000068	0.03087
0.015453	0.002112	0.000075	0.03079
0.015470	0.001155	0.000068	0.03173
0.015527	0.003887	0.000051	0.02896

HPC Analysis

- Env variables
  - uProf_MAX_PR_INSTANCES - Set the max number of unique parallel regions to be traced. The default value is set to 512
  - uProf_MAX_PR_INSTANCE_COUNT - Set the max number of times one unique parallel region to be traced
- Notes
  - Data processing and loading of HPC page can be slower – depending on number of parallel regions and their instances traced.

- Limitations not supported
  - OpenMP® profiling with system-wide profiling scope.
  - Loop chunk size and schedule type when these parameters are specified using schedule clause. It shows the default values (i.e., ‘1’ & ‘Static’) in this case.
  - Nested parallel regions.
  - GPU offloading and related constructs.
  - Call stack for individual OpenMP threads.
  - OpenMP profiling on Windows® and FreeBSD platforms.
  - Applications with static linkage of OpenMP libraries.



[AMD Public Use]

MPI Code Profiling

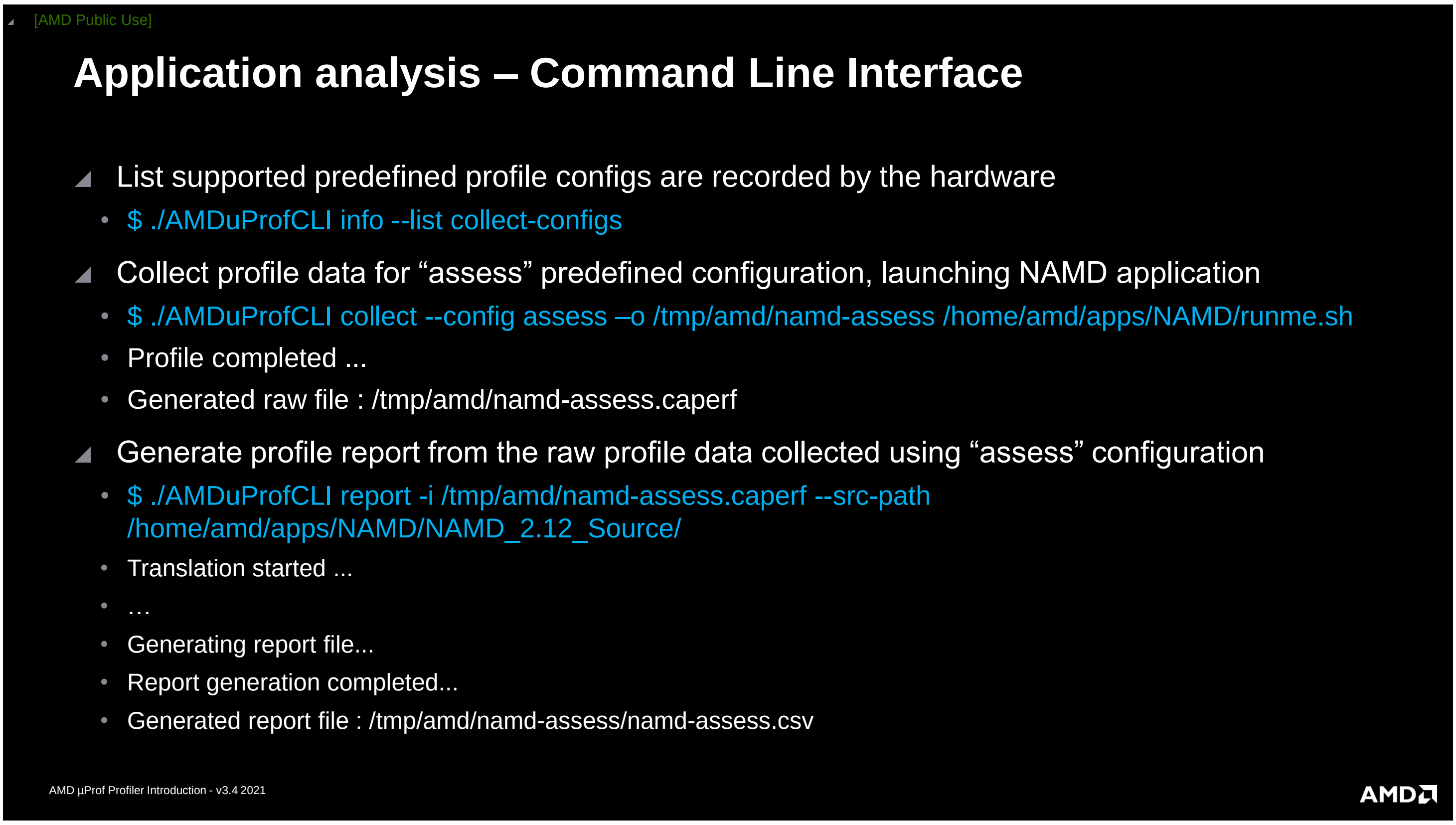
- Support matrix

Component	Supported Version
MPI Spec	• MPI 3.1
MPI Libraries	• Open MPI v4.1.0
	• MPICH 3.4.1
	• ParaStation® MPI 5.4.8
	• Intel® MPI 2019
OS	• Ubuntu® 18.04 LTS • Ubuntu® 20.04 LTS • Red Hat® Enterprise Linux® 8.x • CentOS™ 8.x

```
Usage Model:
Collect performance data
$ mpirun -np <n> AMDuProfCLI collect
--config tbp --mpi --output-dir /tmp/mpi-prof-data ./my-
app

Collect performance data in multiple node
$ mpirun -np 16 -H host1,host2 AMDuProfCLI collect --
config tbp --mpi --output-dir /tmp/myapp-perf myapp.exe

Profiling specific rank
$ export AMDuProfCLI_CMD='AMDuProfCLI collect --config
tbp --mpi --output-dir /tmp/myapp-perf'

$ mpirun -np 4 -host host1 myapp.exe : -host host2 -np 2
"$AMDuProfCLI_CMD" myapp.exe

Translate profile data
$ AMDuProfCLI translate --input-dir /tmp/myapp-perf/ --
host host1

Import the DB for further analysis
```

AMD µProf Profiler Introduction - v3.4 2021

AMD



[AMD Public Use]

Application analysis – Command Line Interface

- List supported predefined profile configs are recorded by the hardware
  - $ ./AMDuProfCLI info --list collect-configs
- Collect profile data for “assess” predefined configuration, launching NAMD application
  - $ ./AMDuProfCLI collect --config assess –o /tmp/amd/namd-assess /home/amd/apps/NAMD/runme.sh
  - Profile completed ...
  - Generated raw file : /tmp/amd/namd-assess.caperf
- Generate profile report from the raw profile data collected using “assess” configuration
  - $ ./AMDuProfCLI report -i /tmp/amd/namd-assess.caperf --src-path /home/amd/apps/NAMD/NAMD_2.12_Source/
  - Translation started ...
  - …
  - Generating report file...
  - Report generation completed...
  - Generated report file : /tmp/amd/namd-assess/namd-assess.csv

AMD µProf Profiler Introduction - v3.4 2021

AMD

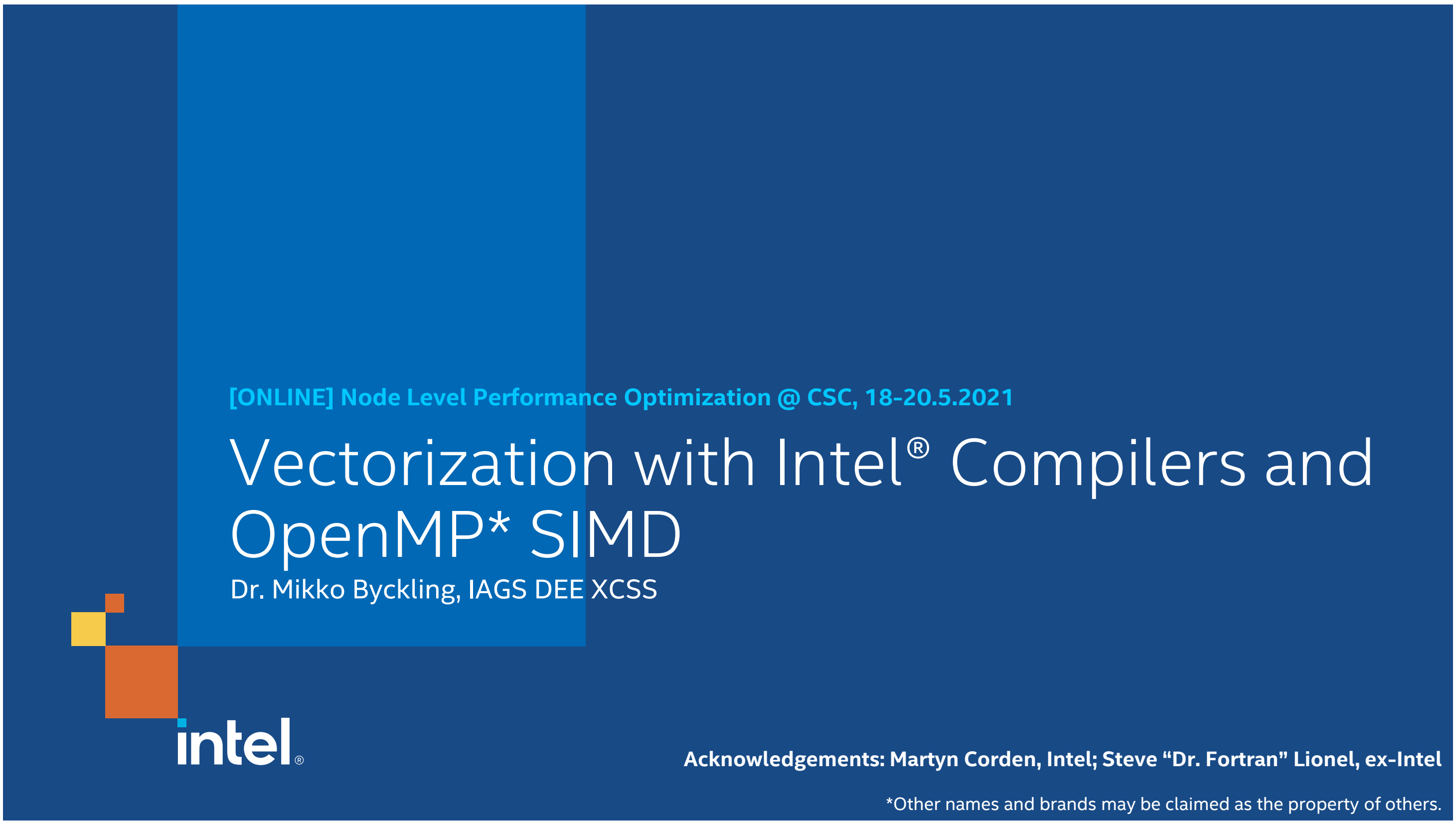
Application analysis – Linux® perf kernel module constraints

- Profiling as non-root user requires /proc/sys/kernel/perf_event_paranoid to be set to -1
- Open file descriptors should be increased to (using “ulimit -n” command)
  - ~100 * number of logical cores
- For Gen2 and Gen3 EPYC™ processors, following distributions are supported:
  - Red Hat Enterprise Linux (RHEL) 8.0.2 with kernel version 4.18.0-80.7.1.el8 or later
  - CentOS® 8.0.1905 with kernel version 4.18.0-80.7.1.el8 or later
  - Ubuntu® 18.04.3 LTS or 19.10 or later
  - SUSE® Linux Enterprise Server (SUSE) 15 SP1 with kernel version 4.12.14-197.26 or later
- On Gen2 and Gen3 EPYC, older Linux® kernels may lead to following error messages:
  - kernel: “Uhhuh. NMI received for unknown reason 3d on CPU 1.”
  - kernel: “Do you have a strange power saving mode enabled?”
  - kernel: “Dazed and confused, but trying to continue”

DISCLAIMER AND TRADEMARKS

DISCLAIMER The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD’s products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale.

AMD



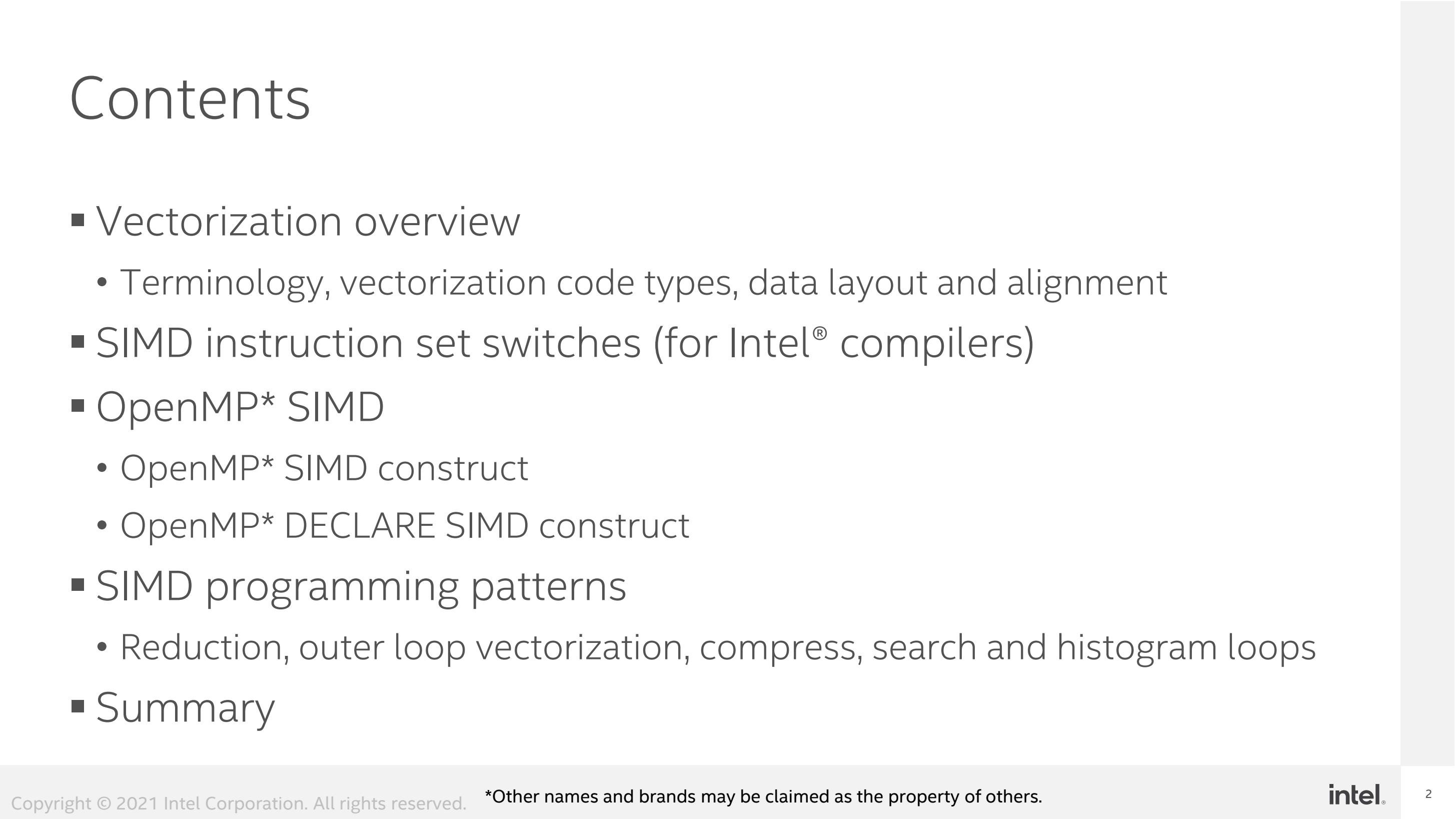
[ONLINE] Node Level Performance Optimization @ CSC, 18-20.5.2021

Vectorization with Intel® Compilers and OpenMP* SIMD

Dr. Mikko Byckling, IAGS DEE XCSS

Acknowledgements: Martyn Corden, Intel; Steve "Dr. Fortran" Lionel, ex-Intel

*Other names and brands may be claimed as the property of others.



Contents

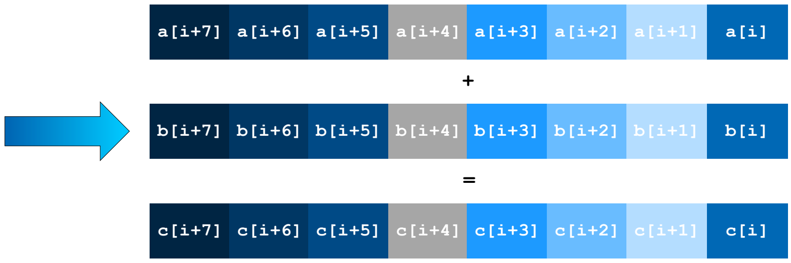
- Vectorization overview
  - Terminology, vectorization code types, data layout and alignment
- SIMD instruction set switches (for Intel® compilers)
- OpenMP* SIMD
  - OpenMP* SIMD construct
  - OpenMP* DECLARE SIMD construct
- SIMD programming patterns
  - Reduction, outer loop vectorization, compress, search and histogram loops
- Summary

*Other names and brands may be claimed as the property of others.

Vectorization of code

- Transform sequential code to exploit SIMD processing capabilities of Intel® processors
  - Calling a vectorized library
  - Automatically by tools like a compiler
  - Manually by explicit syntax

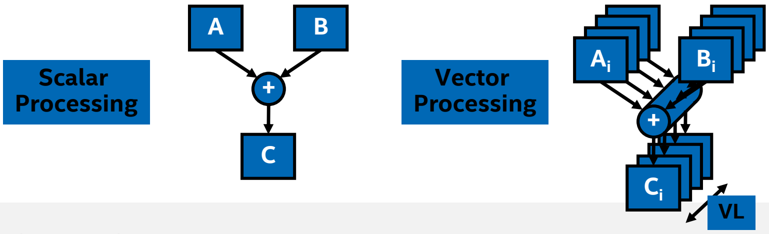
```
for(i = 0; i <= MAX; i++)
   c[i] = a[i] + b[i];
```



a[i+7]	a[i+6]	a[i+5]	a[i+4]	a[i+3]	a[i+2]	a[i+1]	a[i]
			+				
b[i+7]	b[i+6]	b[i+5]	b[i+4]	b[i+3]	b[i+2]	b[i+1]	b[i]
			=				
c[i+7]	c[i+6]	c[i+5]	c[i+4]	c[i+3]	c[i+2]	c[i+1]	c[i]

Vectorization terminology

- Single Instruction Multiple Data (**SIMD**)
  - Processing vector with a single operation
  - Provides data level parallelism (**DLP**)
  - More efficient than scalar processing due to **DLP**
- Vector
  - Consists of more than one element
  - Elements are of same scalar data types (e.g. floats, integers, …)
- Vector length (**VL**), i.e., number of elements in the vector

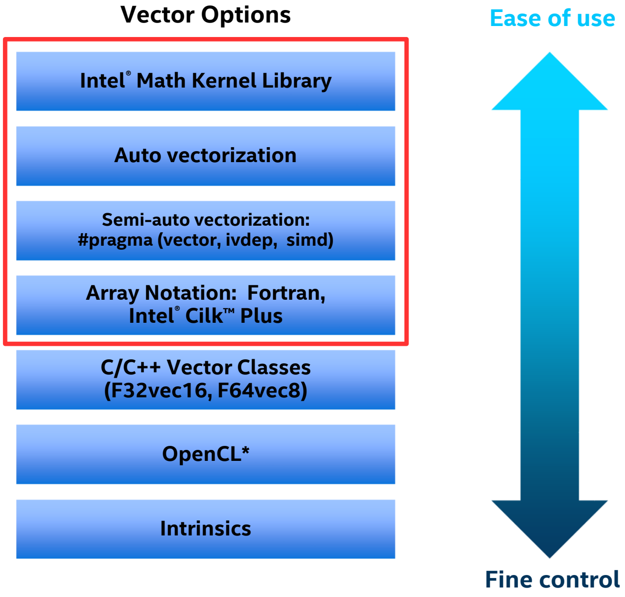


Peel, main and remainder loops

- A vectorized loop consists of
  - Peel loop (optional)
    - Used for the unaligned references in the loop. Uses scalar or slower vector.
  - Main loop body
    - Typically, the fastest part
  - Loop remainder (optional)
    - Used when the number of iterations (trip count) is not divisible by the vector length. Uses Scalar or slower vector.
- Larger vector registers mean more iterations in peel/remainder
- To avoid overhead from peel/remainder loops
  - Avoid loops with a very small trip count
  - Align the data
  - If possible, let the number of iterations be divisible by the vector length

This is where we want our loops to be executing!

Vectorization software architecture



Vector Options

Intel® Math Kernel Library

Auto vectorization

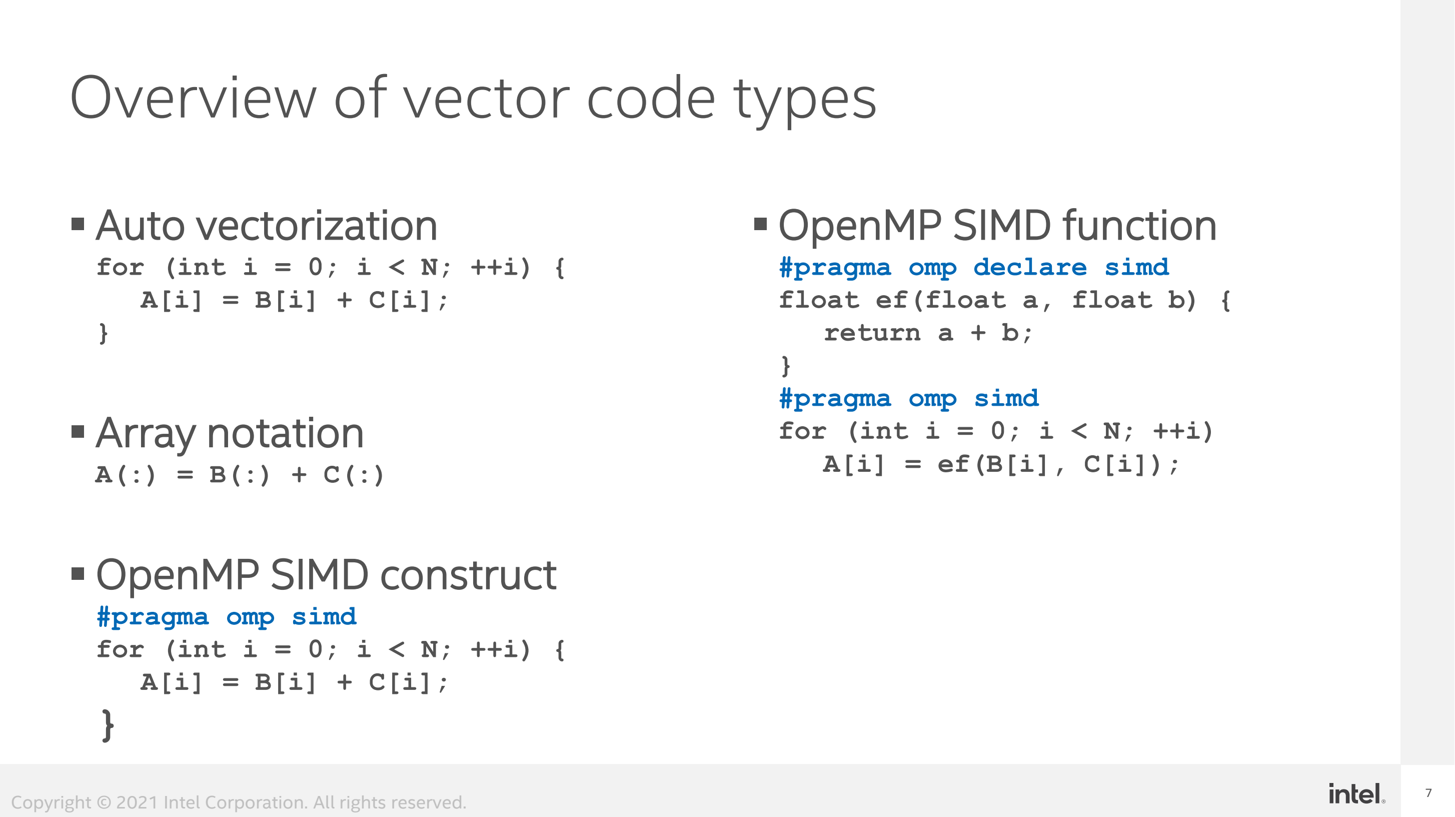
Semi-auto vectorization:
#pragma (vector, ivdep, simd)

Array Notation: Fortran,
Intel® Cilk™ Plus

C/C++ Vector Classes
(F32vec16, F64vec8)

OpenCL*

Intrinsics



Overview of vector code types

- Auto vectorization

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

- Array notation

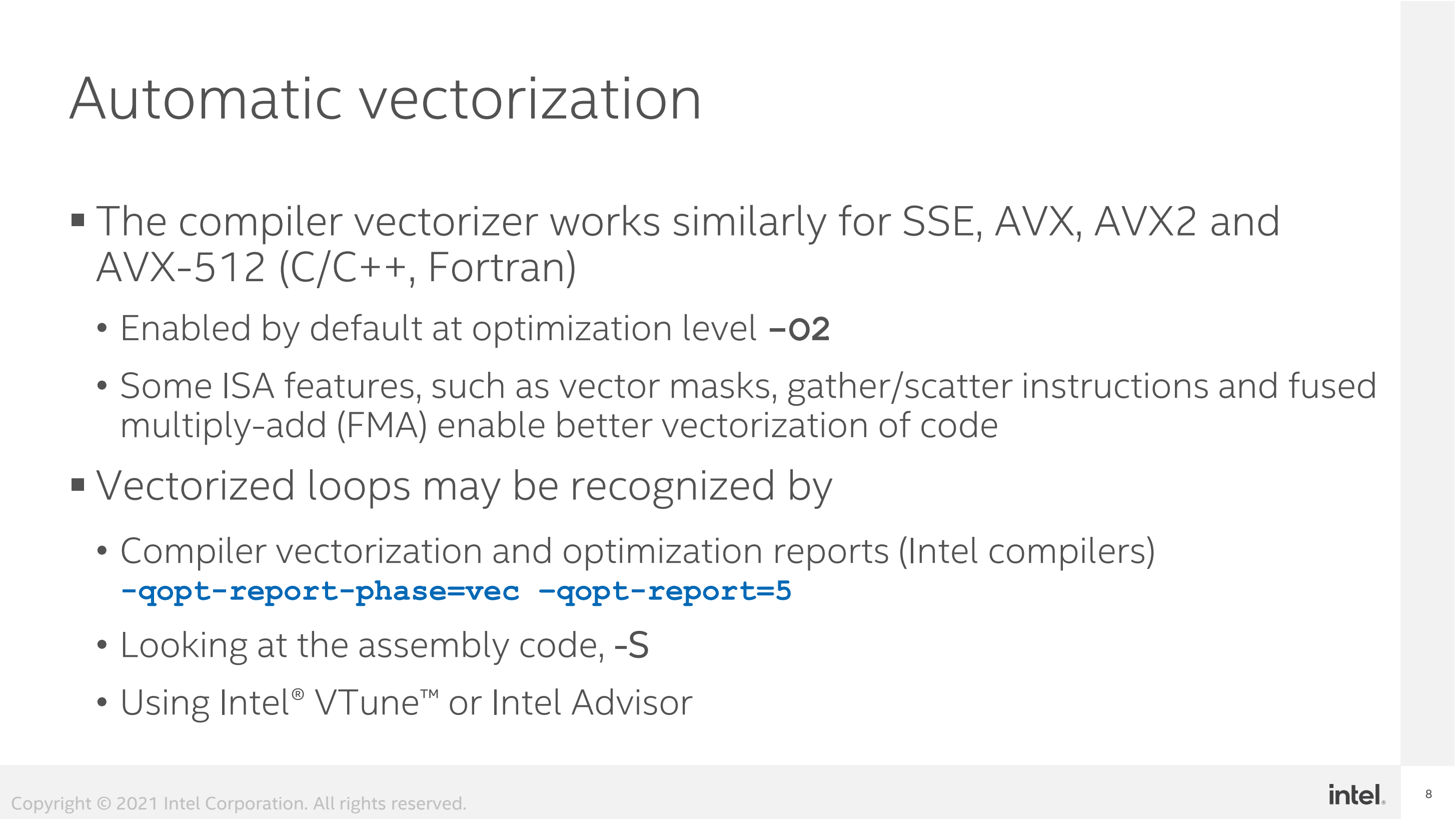
```
A(:) = B(:) + C(:)
```

- OpenMP SIMD construct

```
#pragma omp simd
for (int i = 0; i < N; ++i) {
   A[i] = B[i] + C[i];
}
```

- OpenMP SIMD function

```
#pragma omp declare simd
float ef(float a, float b) {
   return a + b;
}
#pragma omp simd
for (int i = 0; i < N; ++i)
   A[i] = ef(B[i], C[i]);
```



Automatic vectorization

- The compiler vectorizer works similarly for SSE, AVX, AVX2 and AVX-512 (C/C++, Fortran)
  - Enabled by default at optimization level `-O2`
  - Some ISA features, such as vector masks, gather/scatter instructions and fused multiply-add (FMA) enable better vectorization of code
- Vectorized loops may be recognized by
  - Compiler vectorization and optimization reports (Intel compilers) `-qopt-report-phase=vec -qopt-report=5`
  - Looking at the assembly code, -S
  - Using Intel® VTune™ or Intel Advisor

Optimization report: Example

- Example **`novec.f90`**:

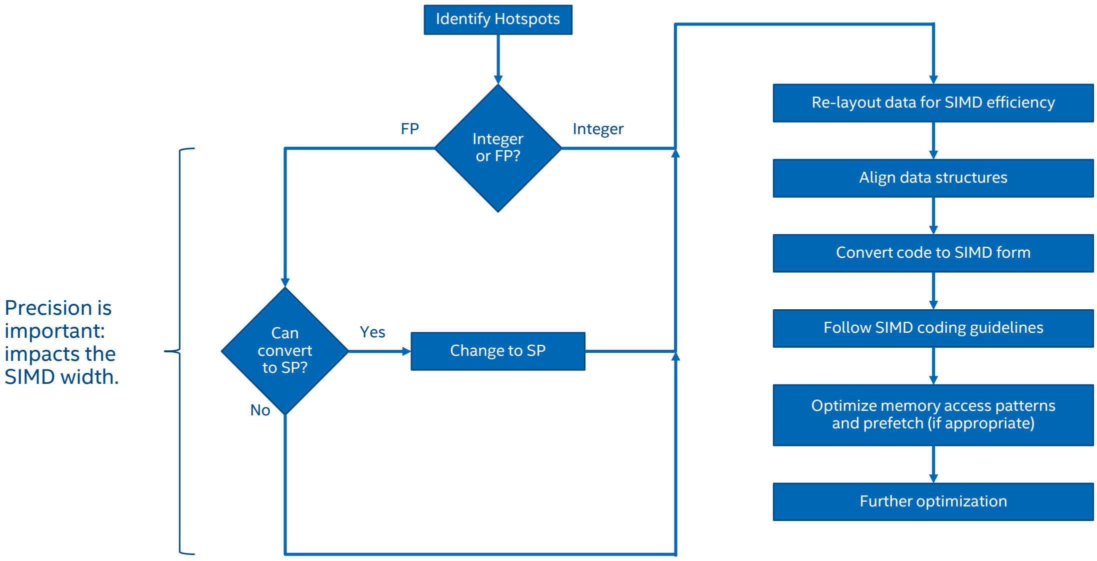
```
1: subroutine fd(y)
2:   integer :: i
3:   real, dimension(10), intent(inout) :: y
4:   do i=2,10
5:     y(i) = y(i-1) + 1
6:   end do
7: end subroutine fd
```

```
$ ifort -c novec.f90 -qopt-report=5
ifort: remark #10397: optimization reports are generated in *.optrpt files in the output location

$ cat novec.optrpt
…
LOOP BEGIN at novec.f90(4,5)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed FLOW dependence between y line 5 and y line 5
   remark #25436: completely unrolled by 9
LOOP END
…
```

Reasons why automatic vectorization fails

- Compiler prioritizes code **correctness**
- Compiler heuristics to estimate vectorization **efficiency**
- Vectorization could lead to incorrect or inefficient code due to
  - Data dependencies
  - Alignment
  - Function calls in loop block
  - Complex control flow / conditional branches
  - Mixed data types
  - Non-unit stride between elements
  - Loop body too complex (register pressure)
  - …

Preparing code for SIMD


Identify Hotspots
Integer or FP?
FP
Integer
Can convert to SP?
Yes
No
Change to SP
Re-layout data for SIMD efficiency
Align data structures
Convert code to SIMD form
Follow SIMD coding guidelines
Optimize memory access patterns and prefetch (if appropriate)
Further optimization
Precision is important: impacts the SIMD width.

Data Layout – why it is important

- Instruction-Level
  - Hardware is optimized for contiguous loads/stores
  - Support for non-contiguous accesses differs with hardware (e.g., AVX2/AVX-512 gather)
- Memory-Level
  - Contiguous memory accesses are cache-friendly
  - Number of memory streams can place pressure on prefetchers

Data layout – common layouts

Array-of-Structs (AoS)

x	y	z	x	y	z
x	y	z	x	y	z
x	y	z	x	y	z

- Pros:
  Good locality of
  {x, y, z},
  1 memory stream
- Cons:
  Potential for gather/scatter

Struct-of-Arrays (SoA)

x	x	x	x	x	x
y	y	y	y	y	y
z	z	z	z	z	z

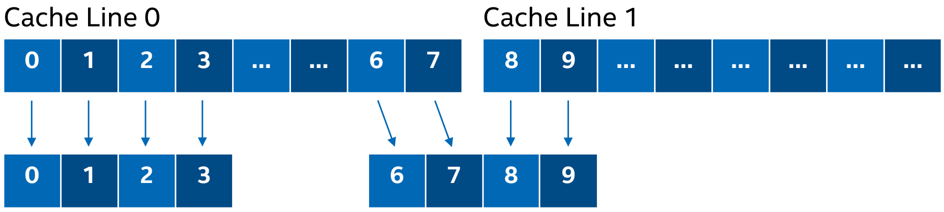
- Pros:
  Contiguous load/store
- Cons:
  Poor locality of
  {x, y, z},
  3 memory streams

Hybrid (AoSoA)

x	x	y	y	z	z
x	x	y	y	z	z
x	x	y	y	z	z

- Pros:
  Contiguous load/store,
  1 memory stream
- Cons:
  Not a “normal” layout

Data alignment – why it is important



Aligned Load

- Address is aligned
- One cache line
- One instruction

Unaligned Load

- Address is not aligned
- Potentially multiple cache lines
- Potentially multiple instructions

Data alignment – sample applications

- 1) Align Memory

```
_mm_malloc(bytes, 64)  /   !dir$ attributes align:64
```

- 2) Access Memory in an Aligned Way

```
for (i = 0; i < N; i++) { array[i] … }
```

- 3) Tell the Compiler   (C\C++ / Fortran)

```
#pragma omp simd aligned(p)    /   !$omp simd aligned(p)
__assume_aligned(p, 16)        /   !dir$ assume_aligned (p, 16)
__assume(i % 16 == 0)          /   !dir$ assume (mod(i,16) .eq. 0)
```

Alignment impact: example

- Unaligned access:

```
void mult(int N, double* a, double* b, double* c)
{
  int i;
#pragma omp simd
  for (i = 0; i < N; i++)
    c[i] = a[i] * b[i];
}
```

```
LOOP BEGIN at mult.c(5,3)
<Peeled loop for vectorization>
   remark #25015: Estimate of max trip count of loop=3
LOOP END

LOOP BEGIN at mult.c(5,3)
   remark #15388: vectorization support: reference c[i] has aligned access   [ mult.c(6,5) ]
   remark #15389: vectorization support: reference a[i] has unaligned access   [ mult.c(6,12) ]
   remark #15389: vectorization support: reference b[i] has unaligned access   [ mult.c(6,19) ]
   remark #15381: vectorization support: unaligned access used inside loop body
...
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15450: unmasked unaligned unit stride loads: 2
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 8
   remark #15477: vector cost: 1.750
   remark #15478: estimated potential speedup: 3.890
   remark #15488: --- end vector cost summary ---
LOOP END
. . .
```

- Aligned access

```
void mult(int N, double* a, double* b, double* c)
{
  int i;
#pragma omp simd aligned(a,b,c)
  for (i = 0; i < N; i++)
    c[i] = a[i] * b[i];
}
```

```
LOOP BEGIN at mult.c(5,3)
   remark #15388: vectorization support: reference c[i] has aligned access   [ mult.c(6,5) ]
   remark #15388: vectorization support: reference a[i] has aligned access   [ mult.c(6,12) ]
   remark #15388: vectorization support: reference b[i] has aligned access   [ mult.c(6,19) ]
...
   remark #15448: unmasked aligned unit stride loads: 2
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 8
   remark #15477: vector cost: 1.250
   remark #15478: estimated potential speedup: 5.260
   remark #15488: --- end vector cost summary ---
LOOP END
. . .
```

Both cases compiled as: **icc -qopenmp -xCORE-AVX2 -qopt-report=5 -c mult.c -o mult.o**

SIMD instruction set switches (for Intel® compilers)

Instruction set architecture switches, instruction set defaults

SIMD instruction set switches (1/3)

For Intel® compilers

- Linux*, OS X*: **`-x<feature>`**, Windows*: **`/Qx<feature>`**
  - Might enable Intel processor specific optimizations
  - Processor-check added to "main" routine:
    Application errors in case SIMD feature missing or non-Intel processor with appropriate/informative message
- Linux*, OS X*: **`-ax<features>`**, Windows*: **`/Qax<features>`**
  - Multiple code paths: baseline and optimized/processor-specific
  - Optimized code paths for Intel processors defined by <features>
  - Multiple SIMD features/paths possible, e.g.: **`-axSSE2, CORE-AVX2`**
  - Baseline code path defaults to **`-msse2`** (**`/arch:sse2`**)
  - The baseline code path can be modified by **`-m<feature>`** or **`-x<feature>`** (**`/arch:<feature>`** or **`/Qx<feature>`**)

SIMD instruction set switches (2/3)

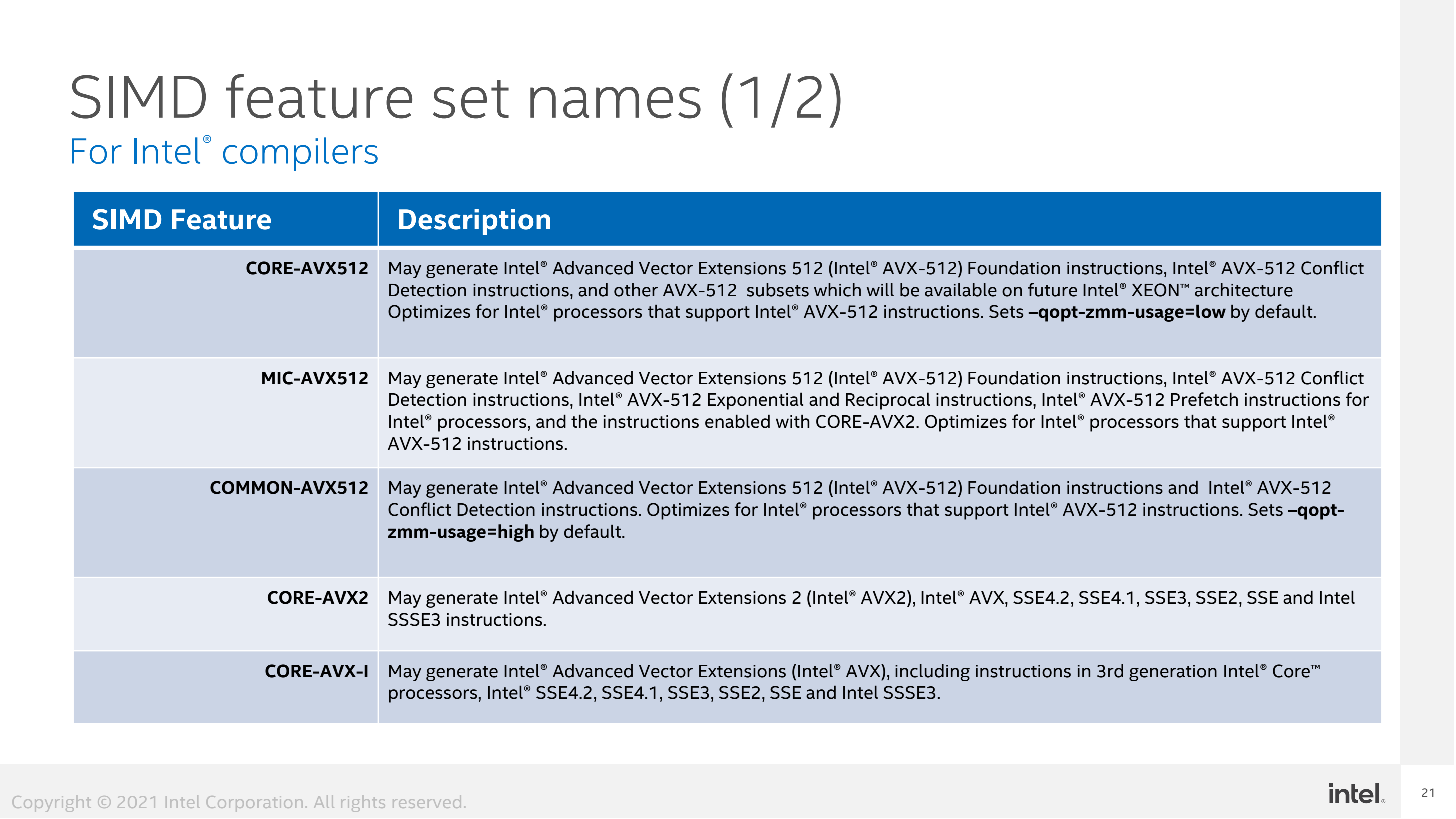
For Intel® compilers

- Linux*, OS X*: **`-m<feature>`**, Windows*: **`/arch:<feature>`**
  - Neither check nor specific optimizations for Intel processors:
    Application optimized for both Intel and non-Intel processors for selected SIMD feature
  - Missing check can cause application to fail in case extension not available
- Default for Linux*: **`-msse2`**, Windows*: **`/arch:sse2`**
  - Activated implicitly
  - Implies the need for a target processor with at least Intel® SSE2
- Default for OS X*: **`-xsse3`** (IA-32), **`-xssse3`** (Intel® 64)

SIMD instruction set switches (3/3)

For Intel® compilers

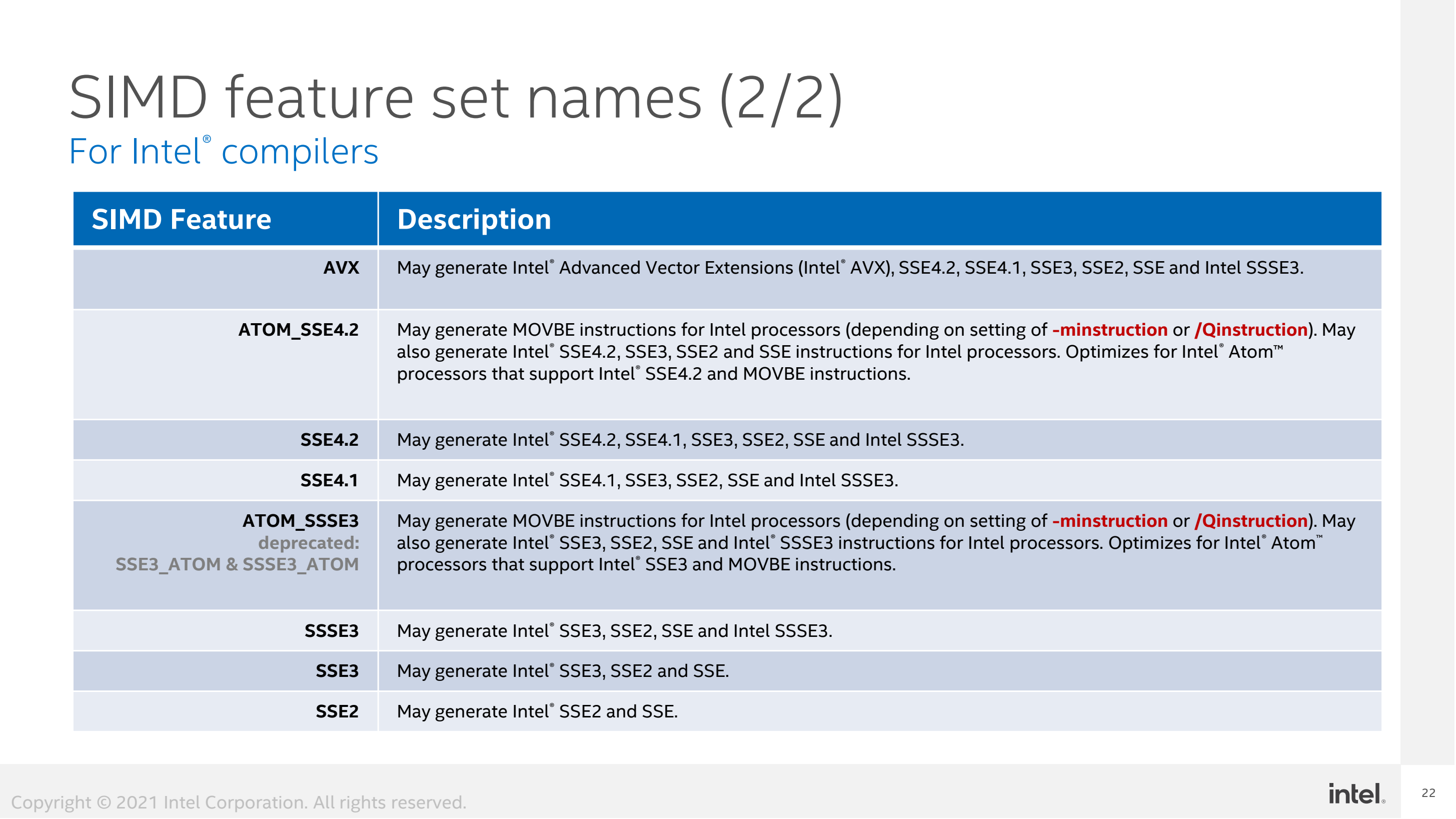
- Special switch for Linux*, OS X*: **`-xHost`**, Windows*: **`/QxHost`**
  - Compiler checks SIMD features of current host processor (where built on) and makes use of latest SIMD feature available
  - Code only executes on processors with same SIMD feature or later as on build host
  - As for **`-x<feature>`** or **`/Qx<feature>`**, if “main” routine is built with
    –xHost or /QxHost the final executable only runs on Intel processors
- Disabling vectorization Linux*, OS X*: **`-no-vec`**, Windows*: **`/Qvec-`**
  - Disables vectorization for the compile unit
  - The compiler can still use some SIMD features



SIMD feature set names (1/2)

For Intel® compilers

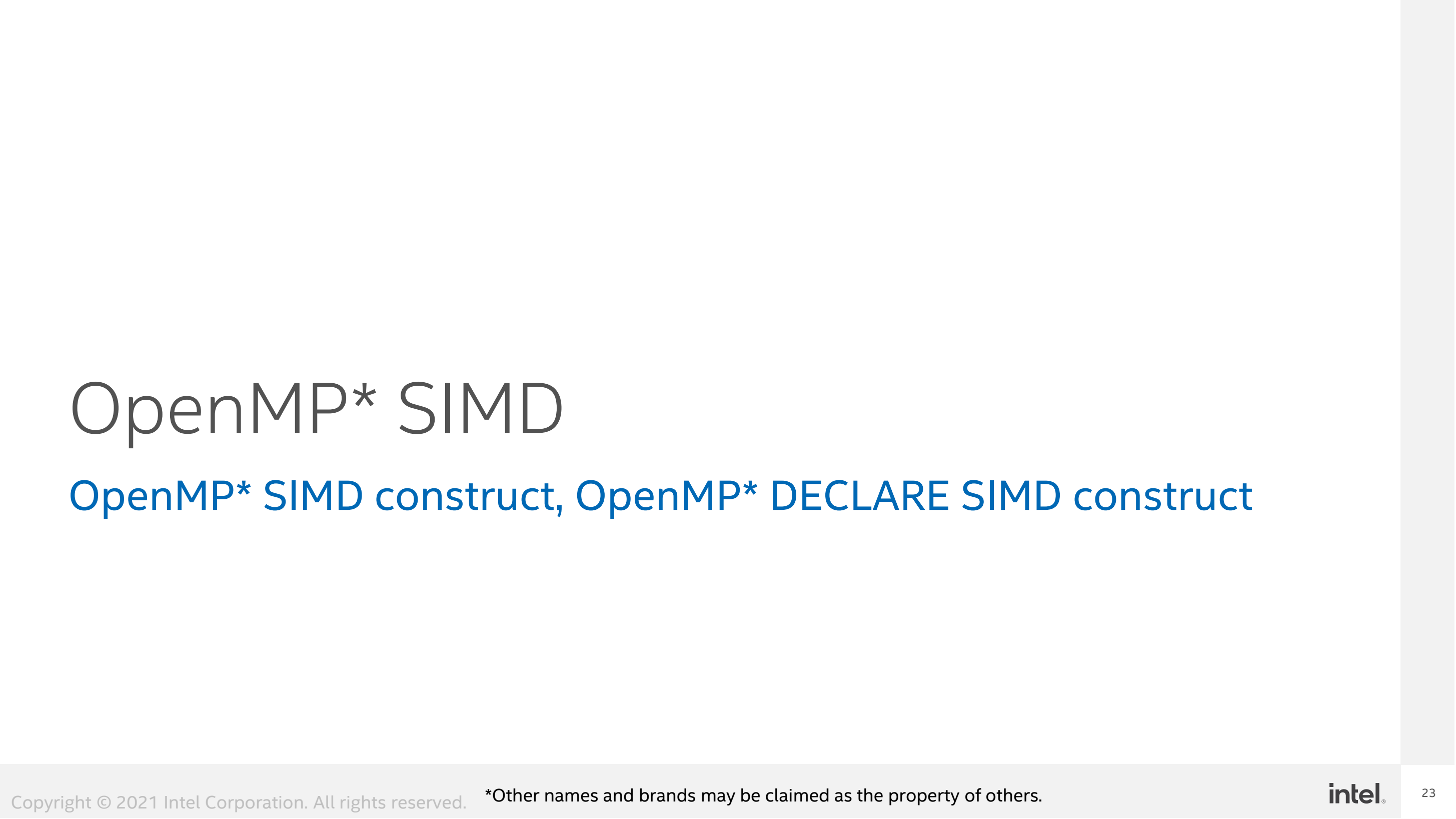
SIMD Feature	Description
CORE-AVX512	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection instructions, and other AVX-512 subsets which will be available on future Intel® XEON™ architecture Optimizes for Intel® processors that support Intel® AVX-512 instructions. Sets **–qopt-zmm-usage=low** by default.
MIC-AVX512	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection instructions, Intel® AVX-512 Exponential and Reciprocal instructions, Intel® AVX-512 Prefetch instructions for Intel® processors, and the instructions enabled with CORE-AVX2. Optimizes for Intel® processors that support Intel® AVX-512 instructions.
COMMON-AVX512	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions and Intel® AVX-512 Conflict Detection instructions. Optimizes for Intel® processors that support Intel® AVX-512 instructions. Sets **–qopt-zmm-usage=high** by default.
CORE-AVX2	May generate Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE and Intel SSSE3 instructions.
CORE-AVX-I	May generate Intel® Advanced Vector Extensions (Intel® AVX), including instructions in 3rd generation Intel® Core™ processors, Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE and Intel SSSE3.



SIMD feature set names (2/2)

For Intel® compilers

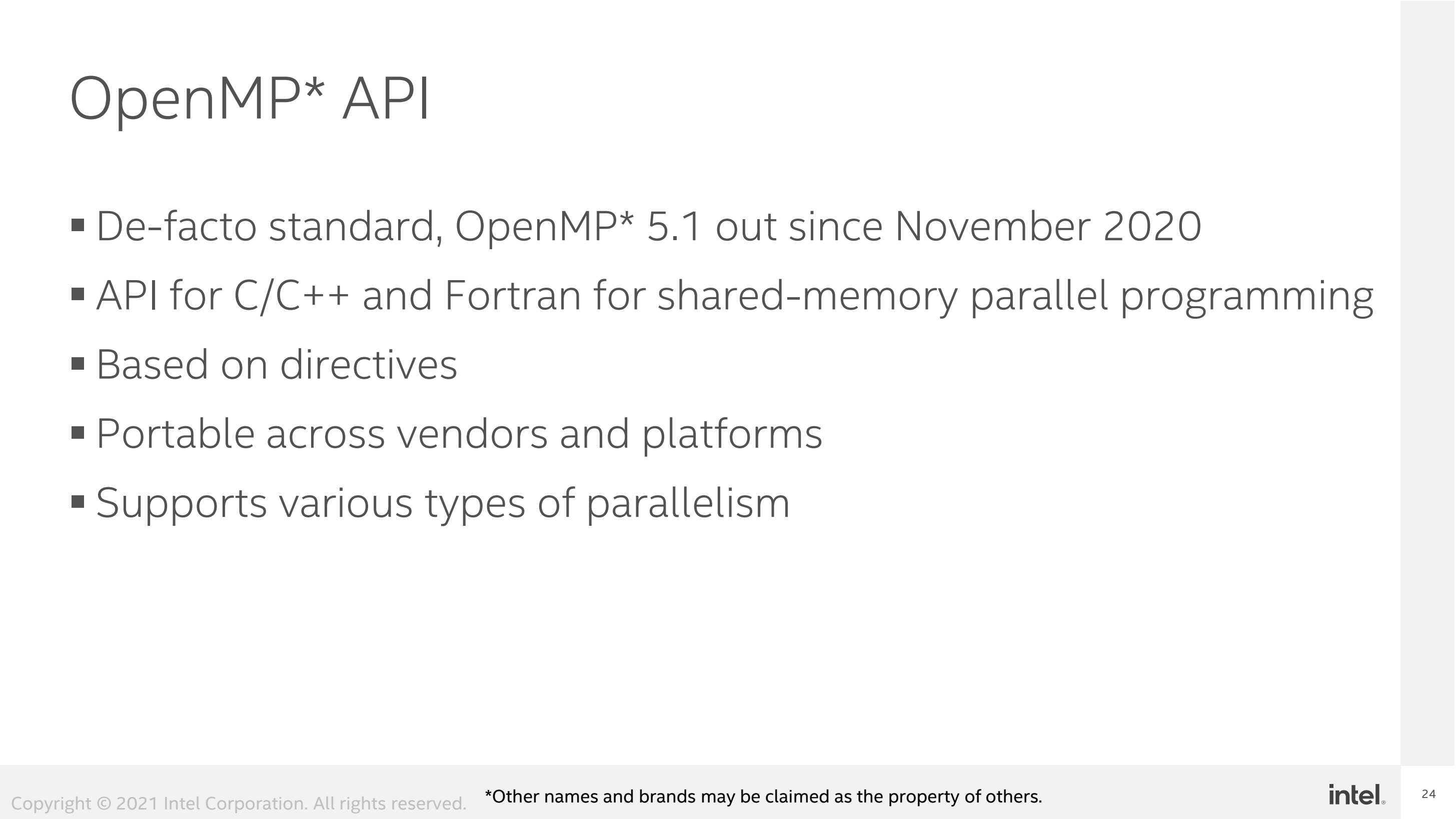
SIMD Feature	Description
AVX	May generate Intel® Advanced Vector Extensions (Intel® AVX), SSE4.2, SSE4.1, SSE3, SSE2, SSE and Intel SSSE3.
ATOM_SSE4.2	May generate MOVBE instructions for Intel processors (depending on setting of **-minstruction** or **/Qinstruction**). May also generate Intel® SSE4.2, SSE3, SSE2 and SSE instructions for Intel processors. Optimizes for Intel® Atom™ processors that support Intel® SSE4.2 and MOVBE instructions.
SSE4.2	May generate Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE and Intel SSSE3.
SSE4.1	May generate Intel® SSE4.1, SSE3, SSE2, SSE and Intel SSSE3.
ATOM_SSSE3 deprecated: SSE3_ATOM & SSSE3_ATOM	May generate MOVBE instructions for Intel processors (depending on setting of **-minstruction** or **/Qinstruction**). May also generate Intel® SSE3, SSE2, SSE and Intel® SSSE3 instructions for Intel processors. Optimizes for Intel® Atom™ processors that support Intel® SSE3 and MOVBE instructions.
SSSE3	May generate Intel® SSE3, SSE2, SSE and Intel SSSE3.
SSE3	May generate Intel® SSE3, SSE2 and SSE.
SSE2	May generate Intel® SSE2 and SSE.



OpenMP* SIMD

OpenMP* SIMD construct, OpenMP* DECLARE SIMD construct

*Other names and brands may be claimed as the property of others.

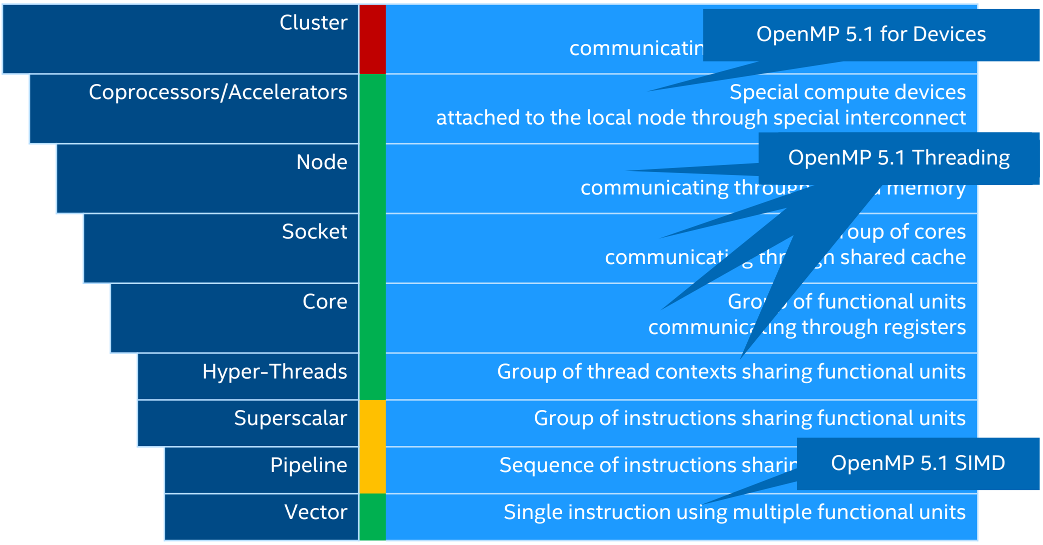


OpenMP* API

- De-facto standard, OpenMP* 5.1 out since November 2020
- API for C/C++ and Fortran for shared-memory parallel programming
- Based on directives
- Portable across vendors and platforms
- Supports various types of parallelism

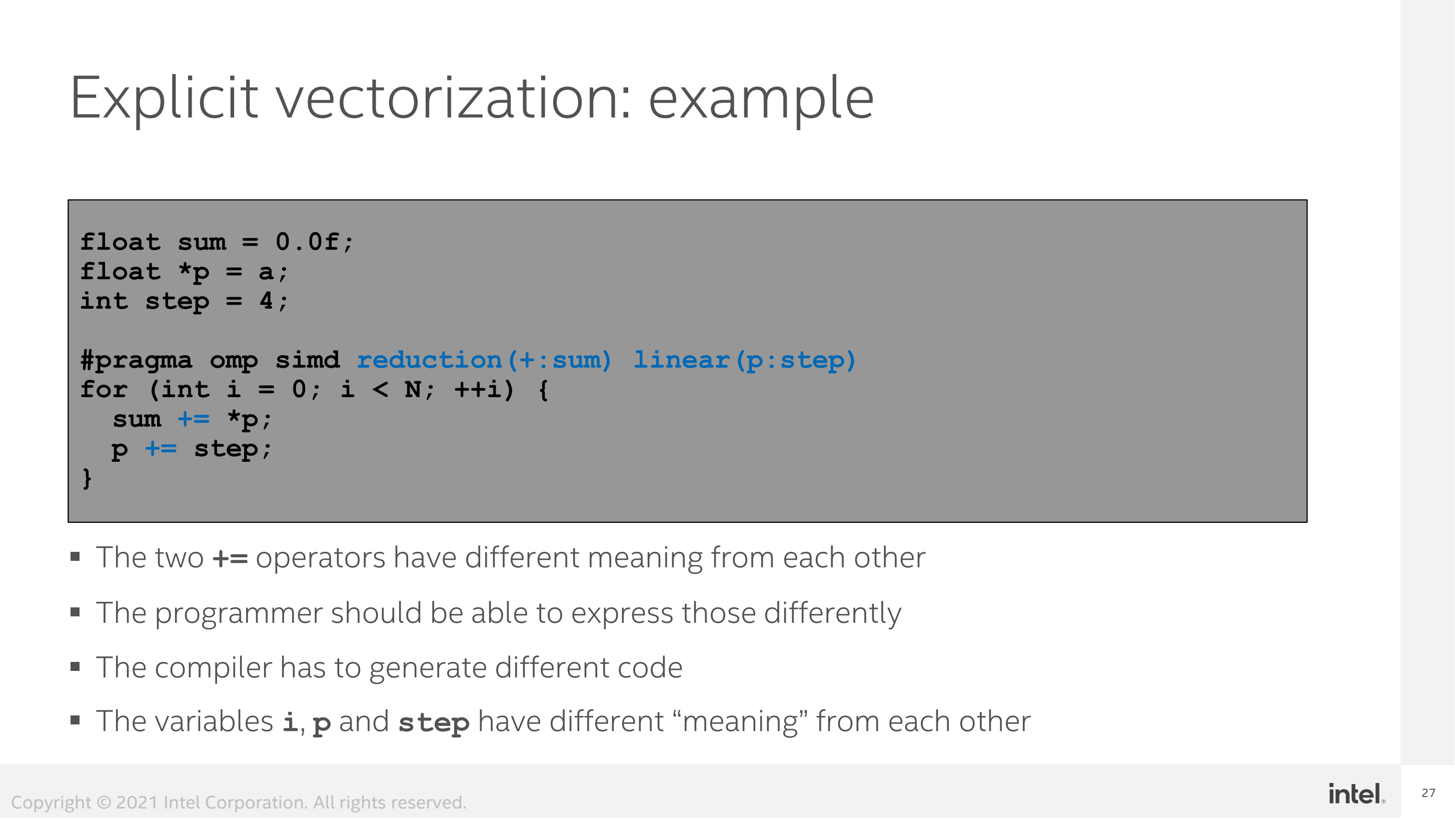
*Other names and brands may be claimed as the property of others.

Levels of parallelism in OpenMP 5.1


Cluster
communicatin
OpenMP 5.1 for Devices
Coprocessors/Accelerators
Special compute devices
attached to the local node through special interconnect
Node
OpenMP 5.1 Threading
communicating throu
memory
Socket
oup of cores
communicati g thr n shared cache
Core
Gro of functional units
communica ing through registers
Hyper-Threads
Group of thread contexts sharing functional units
Superscalar
Group of instructions sharing functional units
Pipeline
Sequence of instructions sharin
OpenMP 5.1 SIMD
Vector
Single instruction using multiple functional units

Explicit vectorization

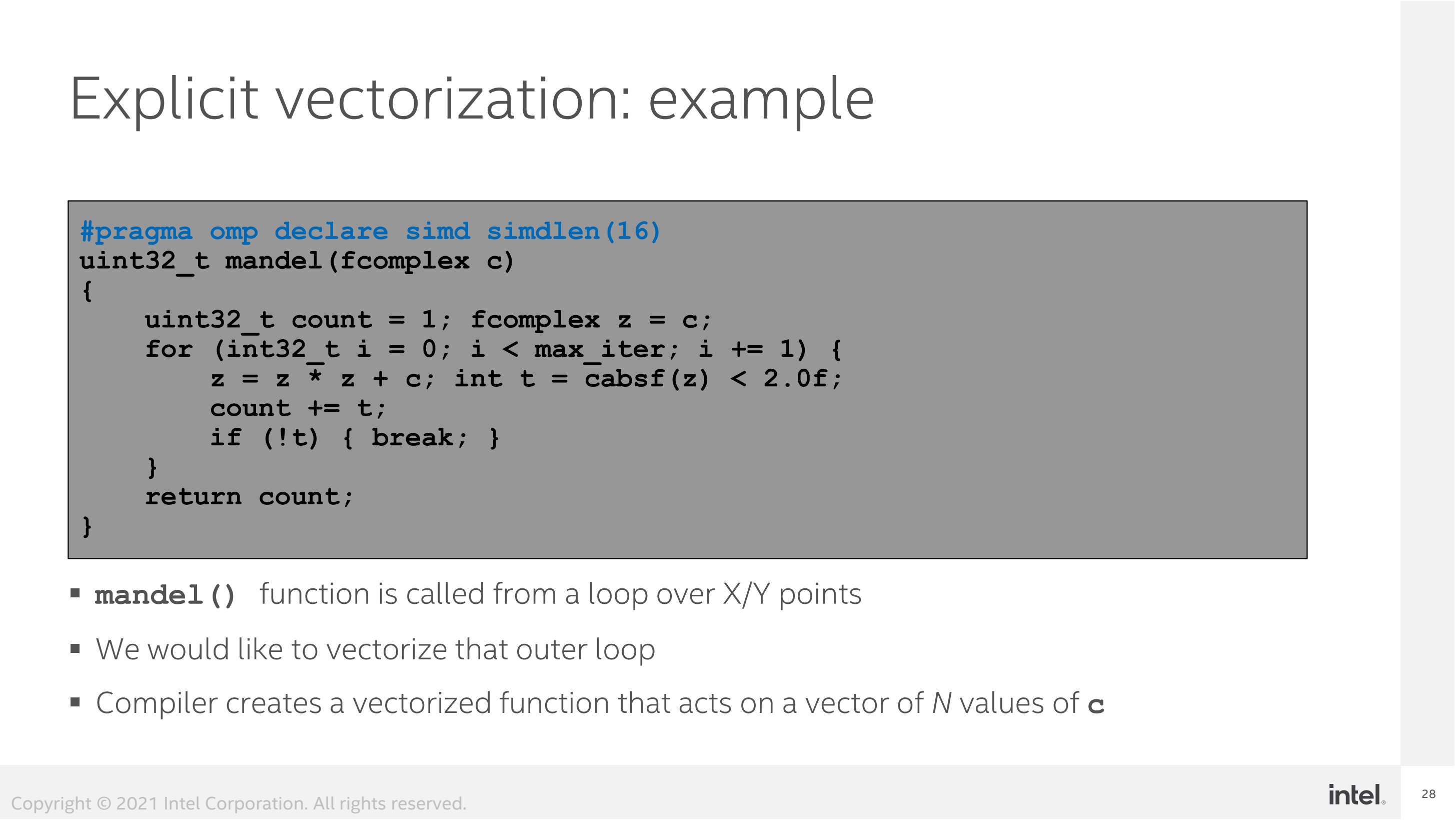
- Compiler Responsibilities
  - Allow programmer to declare that code **can** and **should** be run in SIMD
  - Generate the code the programmer asked for
- Programmer Responsibilities
  - Correctness (e.g., no dependencies, no invalid memory accesses)
  - Efficiency (e.g., alignment, loop order, masking)



Explicit vectorization: example

```
float sum = 0.0f;
float *p = a;
int step = 4;

#pragma omp simd reduction(+:sum) linear(p:step)
for (int i = 0; i < N; ++i) {
  sum += *p;
  p += step;
}
```

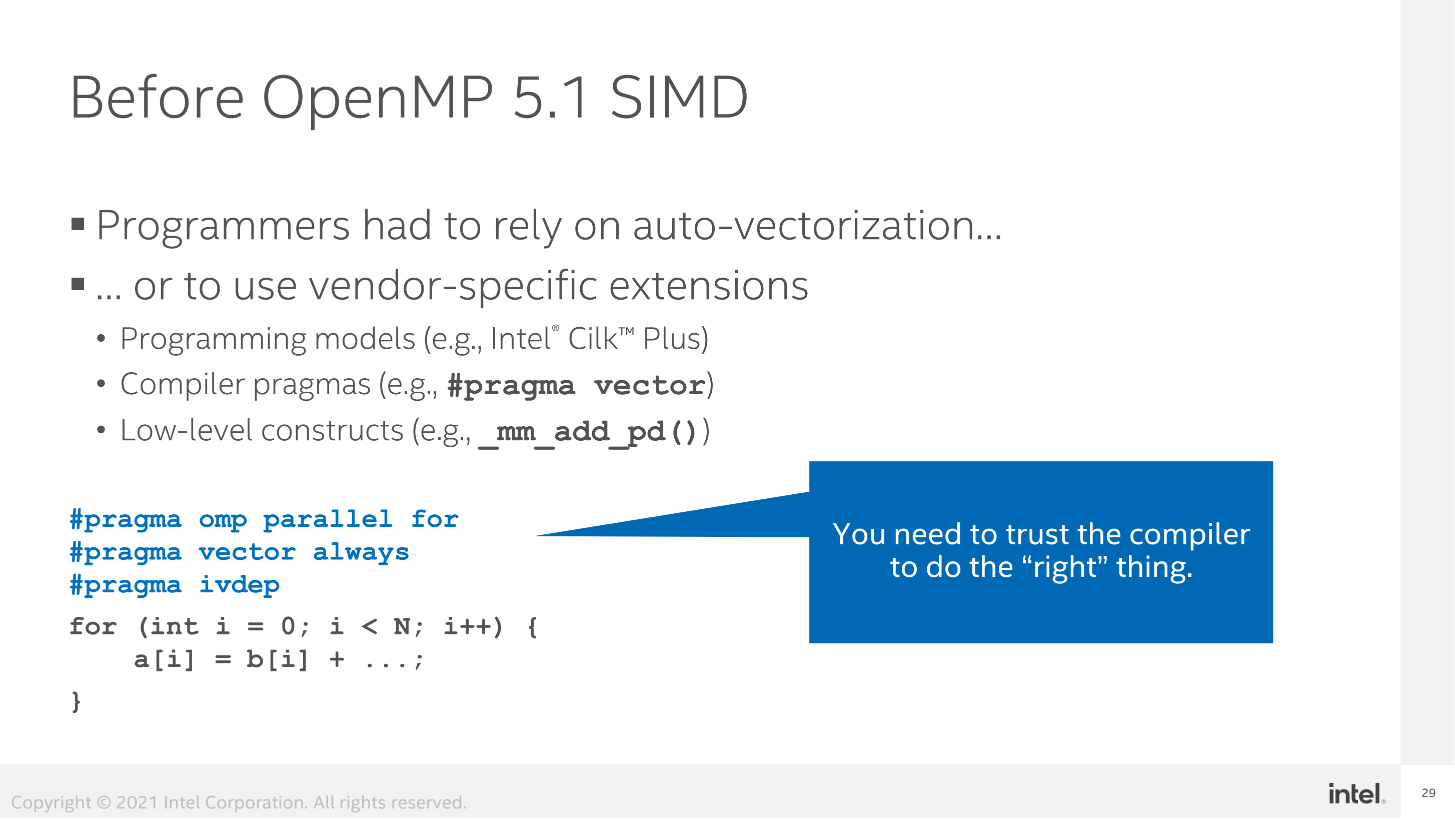
- The two **+=** operators have different meaning from each other
- The programmer should be able to express those differently
- The compiler has to generate different code
- The variables **i**, **p** and **step** have different “meaning” from each other



Explicit vectorization: example

```
#pragma omp declare simd simdlen(16)
uint32_t mandel(fcomplex c)
{
    uint32_t count = 1; fcomplex z = c;
    for (int32_t i = 0; i < max_iter; i += 1) {
        z = z * z + c; int t = cabsf(z) < 2.0f;
        count += t;
        if (!t) { break; }
    }
    return count;
}
```

- **mandel()** function is called from a loop over X/Y points
- We would like to vectorize that outer loop
- Compiler creates a vectorized function that acts on a vector of *N* values of **c**

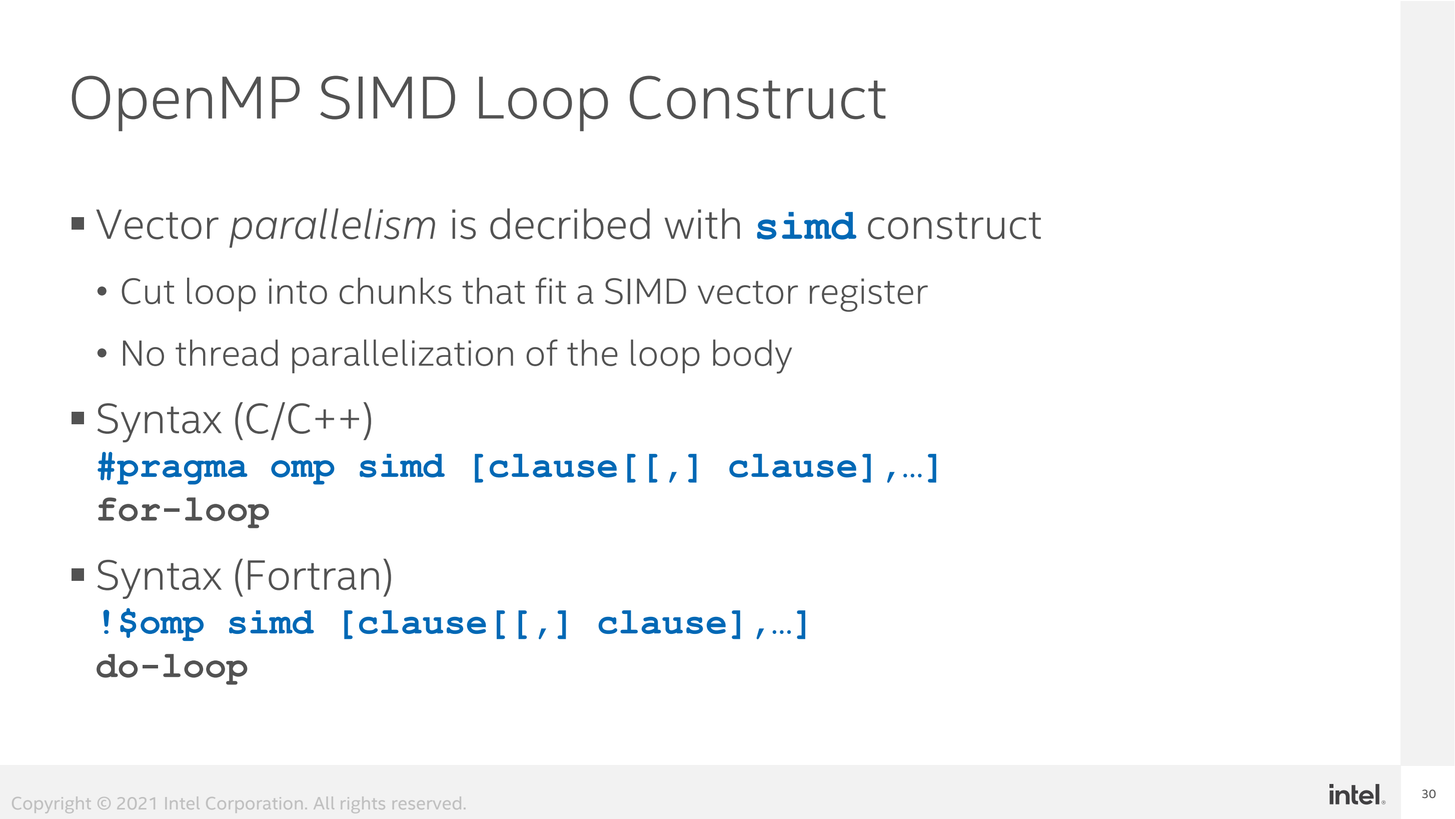


Before OpenMP 5.1 SIMD

- Programmers had to rely on auto-vectorization…
- … or to use vendor-specific extensions
  - Programming models (e.g., Intel® Cilk™ Plus)
  - Compiler pragmas (e.g., **`#pragma vector`**)
  - Low-level constructs (e.g., **`_mm_add_pd()`**)

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + ...;
}
```

You need to trust the compiler to do the “right” thing.



OpenMP SIMD Loop Construct

- Vector *parallelism* is decribed with **`simd`** construct
  - Cut loop into chunks that fit a SIMD vector register
  - No thread parallelization of the loop body
- Syntax (C/C++)

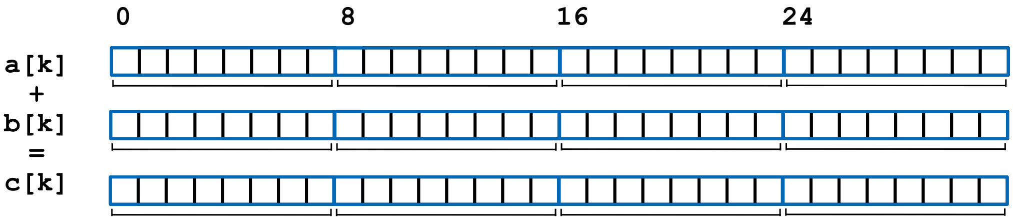
```
#pragma omp simd [clause[[,] clause],…]
for-loop
```

- Syntax (Fortran)

```
!$omp simd [clause[[,] clause],…]
do-loop
```

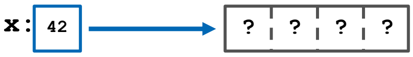
OpenMP SIMD: example

```
void ssum(int n, double *a, double *b, double *c) {
#pragma omp simd
  for (int k=0; k<n; k++)
    c[k] = a[k] + b[k];
}
```

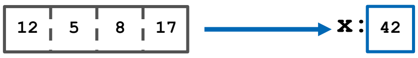


OpenMP SIMD loop clauses

- **private(*var-list*):**
  Uninitialized vectors for variables in *var-list*



- **reduction(*op*:*var-list*):**
  Create private variables for *var-list* and apply reduction operator *op* at the end of the construct



OpenMP SIMD loop clauses

- **`safelen(`*`length`*`)`**
  - Maximum number of iterations that can run concurrently without breaking a dependence
  - in practice, maximum vector length
- **`linear(`*`list`*`[:`*`linear-step`*`])`**
  - The variable’s value is in relationship with the iteration number

    $x_i = x_{orig} + i * \text{linear-step}$
- **`aligned(`*`list`*`[:`*`alignment`*`])`**
  - Specifies that the list items have a given alignment
  - Default is alignment for the architecture
- **`collapse(`*`n`*`)`**
  - Combine the iteration space of the next **`n`** loops

OpenMP SIMD worksharing construct

- Parallelize and vectorize a loop nest
  - Distribute a loop’s iteration space across a thread team
  - Subdivide loop chunks to fit a SIMD vector register
- Syntax (C/C++)

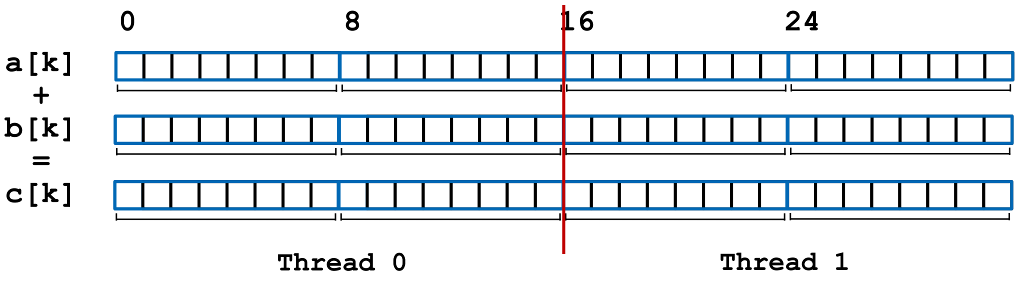
```
#pragma omp for simd [clause[[,] clause],…]
for-loop
```

- Syntax (Fortran)

```
!$omp do simd [clause[[,] clause],…]
do-loop
```

OpenMP SIMD workshare: example

```
void ssum(int n, double *a, double *b, double *c) {
#pragma omp for simd
  for (int k=0; k<n; k++)
    c[k] = a[k] + b[k];
}
```



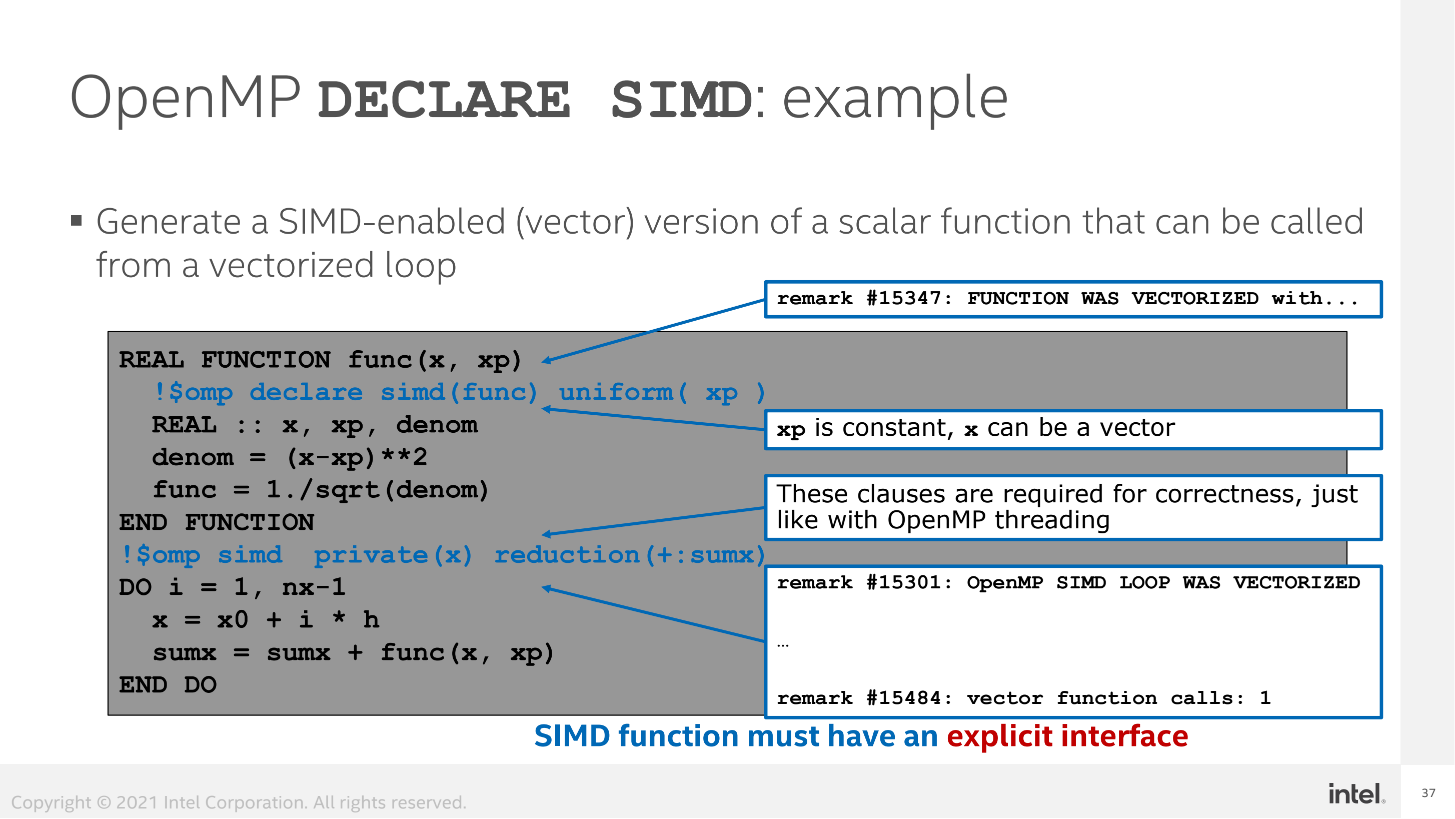
SIMD function vectorization

- Declare one or more functions to be compiled for calls from a SIMD-parallel loop
- Syntax (C/C++):

```
#pragma omp declare simd [clause[[,] clause],…]
[#pragma omp declare simd [clause[[,] clause],…]]
[…]
function-definition-or-declaration
```

- Syntax (Fortran):

```
!$omp declare simd          ! Within function body
!$omp declare simd(proc-name-list) ! At call site
```



OpenMP **DECLARE SIMD**: example

- Generate a SIMD-enabled (vector) version of a scalar function that can be called from a vectorized loop

```
REAL FUNCTION func(x, xp)
  !$omp declare simd(func) uniform( xp )
  REAL :: x, xp, denom
  denom = (x-xp)**2
  func = 1./sqrt(denom)
END FUNCTION
!$omp simd  private(x) reduction(+:sumx)
DO i = 1, nx-1
  x = x0 + i * h
  sumx = sumx + func(x, xp)
END DO
```

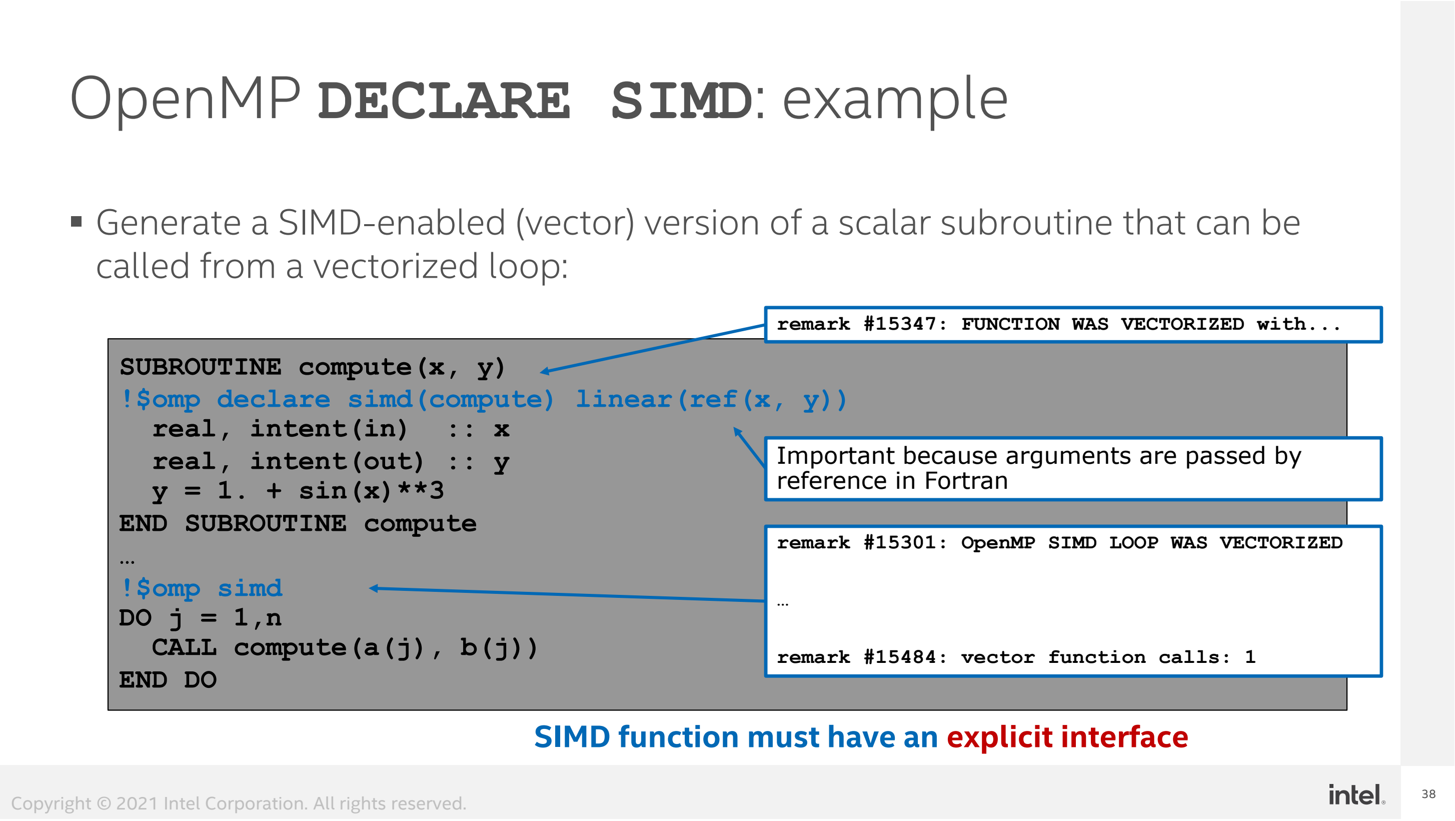
`remark #15347: FUNCTION WAS VECTORIZED with...`

xp is constant, **x** can be a vector

These clauses are required for correctness, just like with OpenMP threading

```
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

…

remark #15484: vector function calls: 1
```

SIMD function must have an explicit interface



OpenMP **DECLARE SIMD**: example

- Generate a SIMD-enabled (vector) version of a scalar subroutine that can be called from a vectorized loop:

```
SUBROUTINE compute(x, y)
!$omp declare simd(compute) linear(ref(x, y))
  real, intent(in)  :: x
  real, intent(out) :: y
  y = 1. + sin(x)**3
END SUBROUTINE compute
…
!$omp simd
DO j = 1,n
  CALL compute(a(j), b(j))
END DO
```

`remark #15347: FUNCTION WAS VECTORIZED with...`

Important because arguments are passed by reference in Fortran

```
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

…

remark #15484: vector function calls: 1
```

SIMD function must have an explicit interface

SIMD function vectorization clauses

- **`simdlen (length)`**
  - Generate function to support a given vector length
- **`uniform (argument-list)`**
  - Argument has a constant value between the iterations of a given loop
- **`inbranch`**
  - Function always called from inside an if statement
- **`notinbranch`**
  - Function never called from inside an if statement
- **`linear(argument-list[:linear-step])`**
- **`aligned(argument-list[:alignment])`**
- **`reduction(operator:list)`**

Same as in **SIMD**

SIMD function arguments and **`LINEAR(REF)`**

- Whenever SIMD function arguments are passed by reference:
  - The compiler places consecutive addresses in a vector register, resulting in a gather from the addresses when the values are needed (=slow)
  - **`LINEAR(REF(…))`** tells the compiler that the addresses are consecutive, resulting to a single dereference and then copy of the consecutive values to a vector register (=fast)
- Recall that Fortran passes **all arguments** by reference
  - **`LINEAR(REF(…))`** is very important for efficient SIMD vectorization of Fortran functions and subroutines

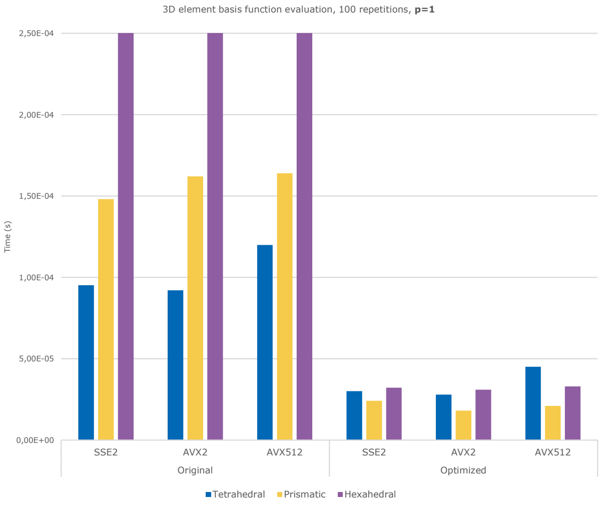
Targeting SIMD functions for CPU ISA

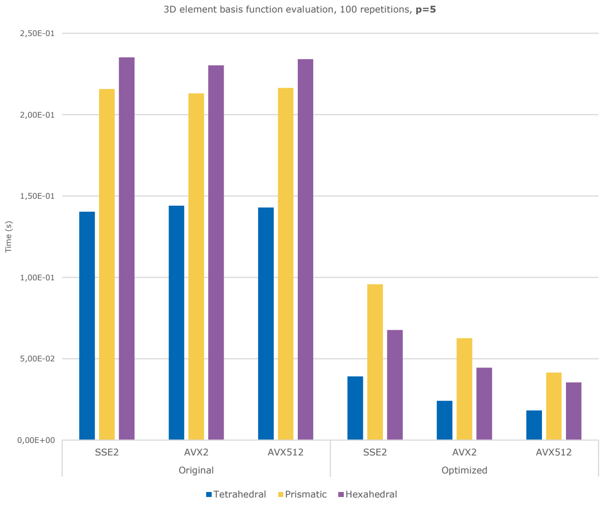
- The default binary ABI requires passing arguments in 128 bit **`xmm`** registers
  - ABI is selected irrespective of **`-xCORE-AVX2`** or **`-xCORE-AVX512`** feature flags
  - Results in inefficient 128 bit code instead of 256 or 512 bit
  - Compiler optimization report:
    `remark #15347: FUNCTION WAS VECTORIZED with xmm, simdlen=4,…`
- Intel® compiler flag **`-vecabi=cmdtarget`**
  - SIMD register width chosen according to the **`-x<feature>`**
  - Compiler optimization report:
    `remark #15347: FUNCTION WAS VECTORIZED with zmm, simdlen=16, …`

Example: OpenMP 4.0 SIMD in Elmer

2S Intel® Xeon® Gold 6148

Results from paper: Byckling, M., Kataja, J., Klemm, M. and Zwinger, T., 2017, September. OpenMP* SIMD Vectorization and Threading of the Elmer Finite Element Software. In International Workshop on OpenMP (pp. 123-137). Springer, Cham.





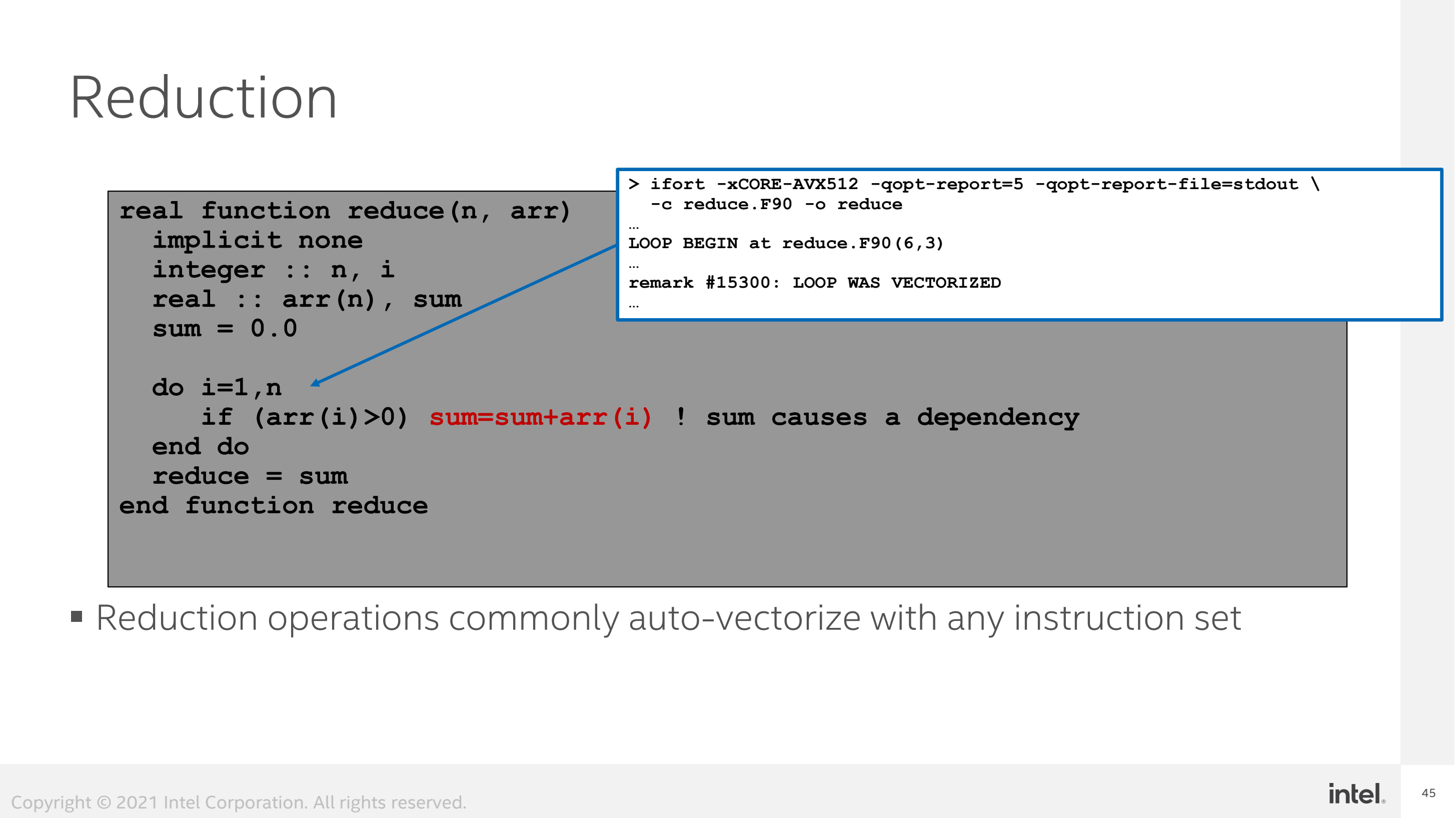
Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex. Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. For configuration info, see System Setup.

SIMD programming patterns

Reduction, outer loop vectorization, compress, search and histogram loops

SIMD programming patterns

- Dependencies can make vectorization unsafe
- Some special patterns can still be handled by the compiler
  - The compiler may recognize a pattern (auto-vectorization)
    - Often works only for simple, ‘clean’ examples
  - The compiler is enforced (explicit vector programming)
    - May work for more complex cases
  - Examples: reduction, compress/expand, search, etc.
- Speed-up can come from vectorizing the rest of a large loop more than from vectorization of the pattern itself

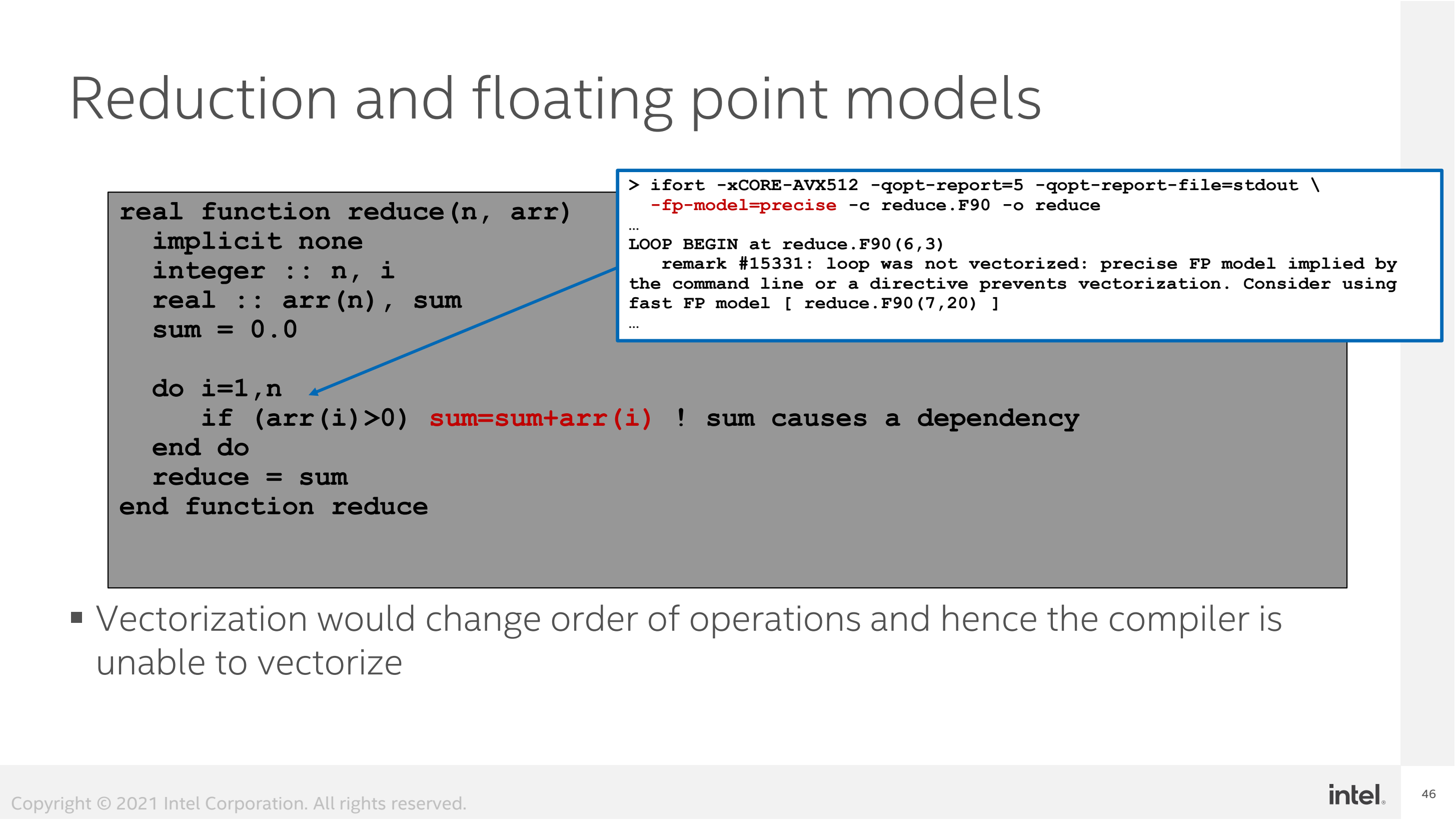


Reduction

```
real function reduce(n, arr)
  implicit none
  integer :: n, i
  real :: arr(n), sum
  sum = 0.0

  do i=1,n
     if (arr(i)>0) sum=sum+arr(i) ! sum causes a dependency
  end do
  reduce = sum
end function reduce
```

```
> ifort -xCORE-AVX512 -qopt-report=5 -qopt-report-file=stdout \
  -c reduce.F90 -o reduce
…
LOOP BEGIN at reduce.F90(6,3)
…
remark #15300: LOOP WAS VECTORIZED
…
```

- Reduction operations commonly auto-vectorize with any instruction set



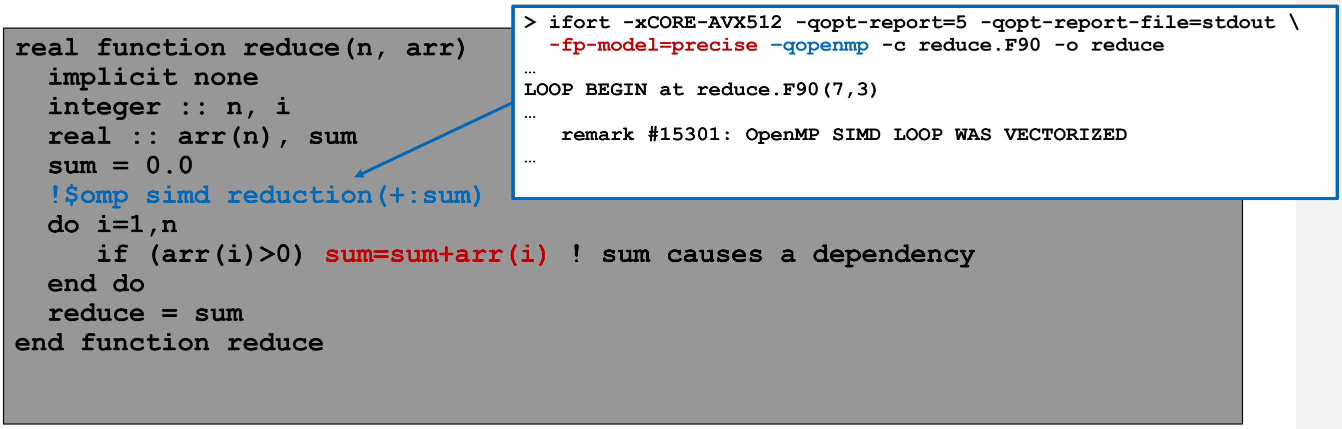
Reduction and floating point models

```
real function reduce(n, arr)
  implicit none
  integer :: n, i
  real :: arr(n), sum
  sum = 0.0

  do i=1,n
     if (arr(i)>0) sum=sum+arr(i) ! sum causes a dependency
  end do
  reduce = sum
end function reduce
```

```
> ifort -xCORE-AVX512 -qopt-report=5 -qopt-report-file=stdout \
  -fp-model=precise -c reduce.F90 -o reduce
…
LOOP BEGIN at reduce.F90(6,3)
   remark #15331: loop was not vectorized: precise FP model implied by
the command line or a directive prevents vectorization. Consider using
fast FP model [ reduce.F90(7,20) ]
…
```

- Vectorization would change order of operations and hence the compiler is unable to vectorize

OpenMP reductions

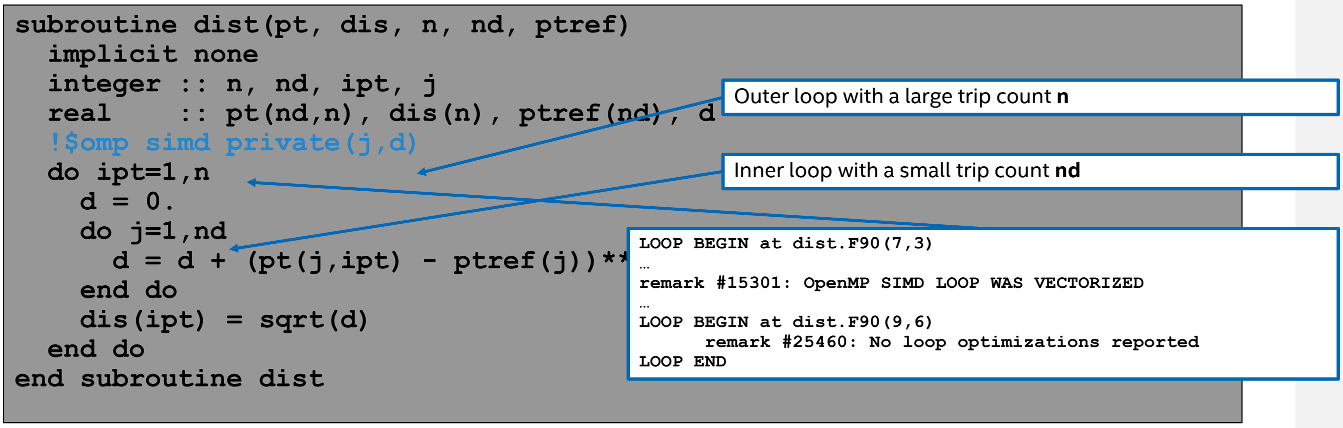


```
real function reduce(n, arr)
  implicit none
  integer :: n, i
  real :: arr(n), sum
  sum = 0.0
  !$omp simd reduction(+:sum)
  do i=1,n
     if (arr(i)>0) sum=sum+arr(i) ! sum causes a dependency
  end do
  reduce = sum
end function reduce
```

```
> ifort -xCORE-AVX512 -qopt-report=5 -qopt-report-file=stdout \
  -fp-model=precise -qopenmp -c reduce.F90 -o reduce
…
LOOP BEGIN at reduce.F90(7,3)
…
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
…
```

- Floating point model can be overridden with explicit vector reduction (OpenMP SIMD reduction)

OpenMP SIMD outer loop vectorization



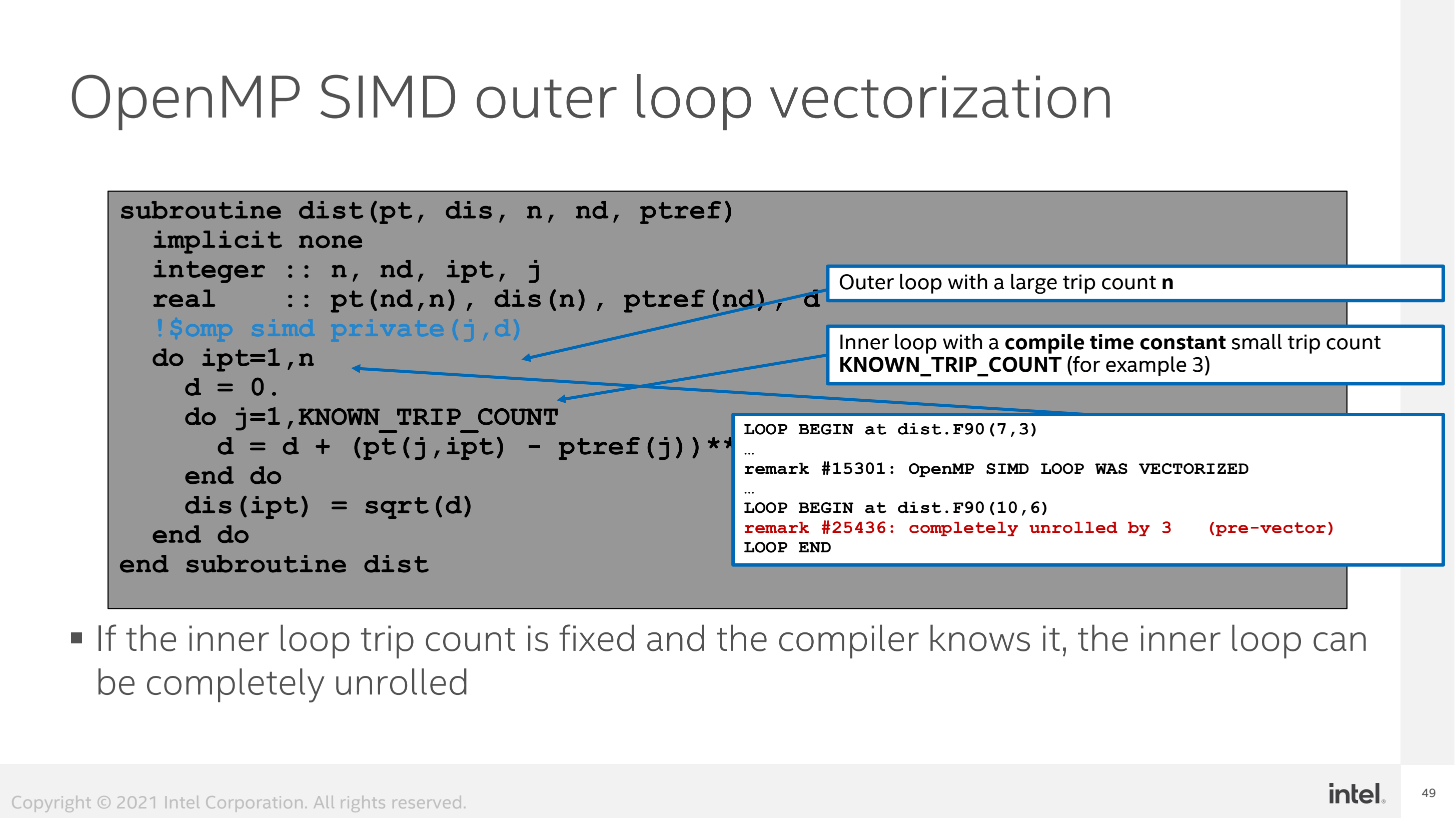
```
subroutine dist(pt, dis, n, nd, ptref)
  implicit none
  integer :: n, nd, ipt, j
  real    :: pt(nd,n), dis(n), ptref(nd), d
  !$omp simd private(j,d)
  do ipt=1,n
    d = 0.
    do j=1,nd
      d = d + (pt(j,ipt) - ptref(j))**2
    end do
    dis(ipt) = sqrt(d)
  end do
end subroutine dist
```

Outer loop with a large trip count **n**

Inner loop with a small trip count **nd**

```
LOOP BEGIN at dist.F90(7,3)
…
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
…
LOOP BEGIN at dist.F90(9,6)
      remark #25460: No loop optimizations reported
LOOP END
```

- When **nd** is small (typically <8), outer loop vectorization may be profitable. Private copies of **j** and **d** needed for correctness



OpenMP SIMD outer loop vectorization

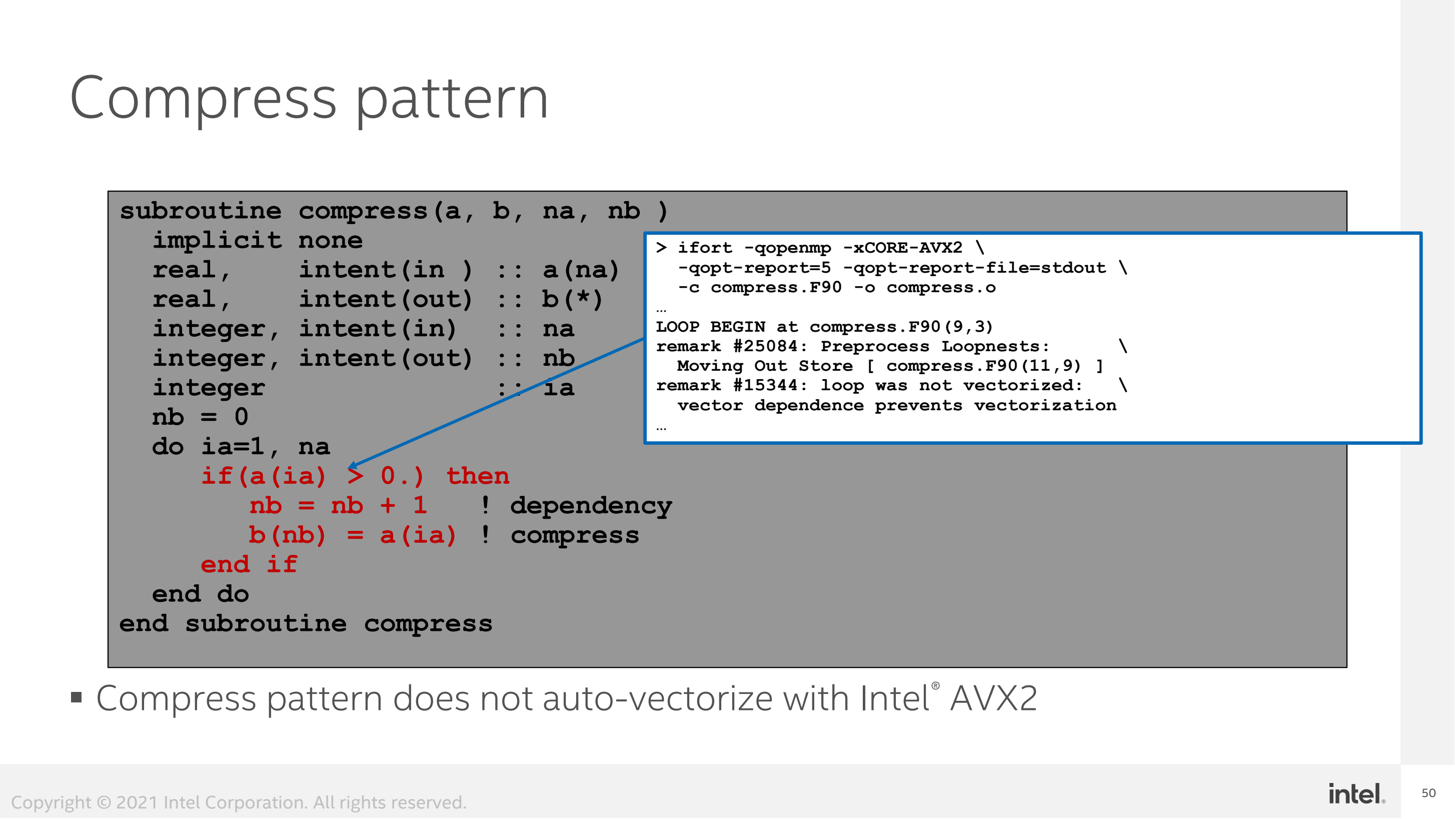
```
subroutine dist(pt, dis, n, nd, ptref)
  implicit none
  integer :: n, nd, ipt, j
  real    :: pt(nd,n), dis(n), ptref(nd), d
  !$omp simd private(j,d)
  do ipt=1,n
    d = 0.
    do j=1,KNOWN_TRIP_COUNT
      d = d + (pt(j,ipt) - ptref(j))**
    end do
    dis(ipt) = sqrt(d)
  end do
end subroutine dist
```

Outer loop with a large trip count **n**

Inner loop with a **compile time constant** small trip count **KNOWN_TRIP_COUNT** (for example 3)

```
LOOP BEGIN at dist.F90(7,3)
…
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
…
LOOP BEGIN at dist.F90(10,6)
remark #25436: completely unrolled by 3   (pre-vector)
LOOP END
```

- If the inner loop trip count is fixed and the compiler knows it, the inner loop can be completely unrolled

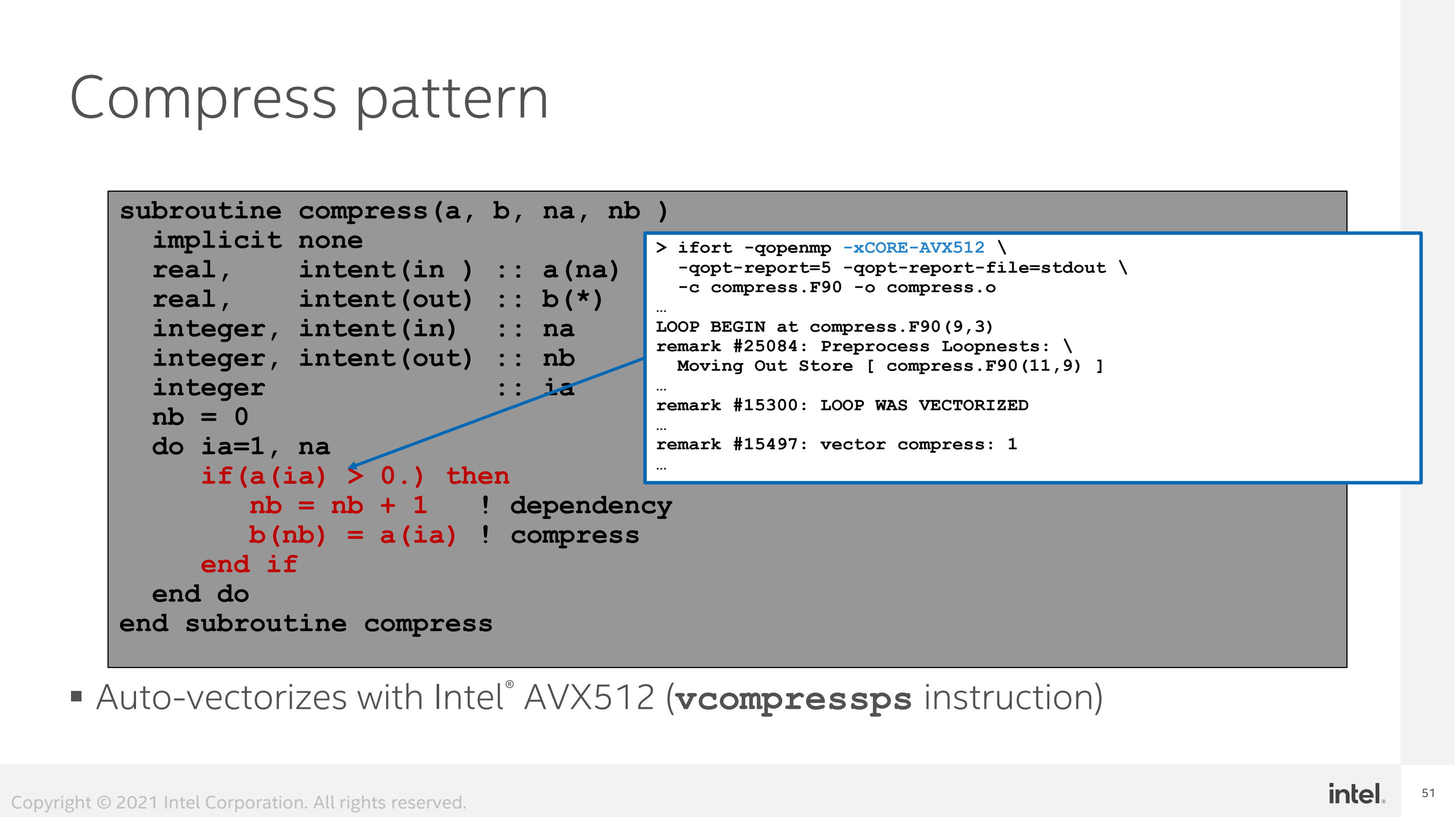


Compress pattern

```
subroutine compress(a, b, na, nb )
  implicit none
  real,    intent(in ) :: a(na)
  real,    intent(out) :: b(*)
  integer, intent(in)  :: na
  integer, intent(out) :: nb
  integer              :: ia
  nb = 0
  do ia=1, na
     if(a(ia) > 0.) then
        nb = nb + 1    ! dependency
        b(nb) = a(ia)  ! compress
     end if
  end do
end subroutine compress
```

```
> ifort -qopenmp -xCORE-AVX2 \
  -qopt-report=5 -qopt-report-file=stdout \
  -c compress.F90 -o compress.o
…
LOOP BEGIN at compress.F90(9,3)
remark #25084: Preprocess Loopnests:      \
  Moving Out Store [ compress.F90(11,9) ]
remark #15344: loop was not vectorized:   \
  vector dependence prevents vectorization
…
```

- Compress pattern does not auto-vectorize with Intel® AVX2

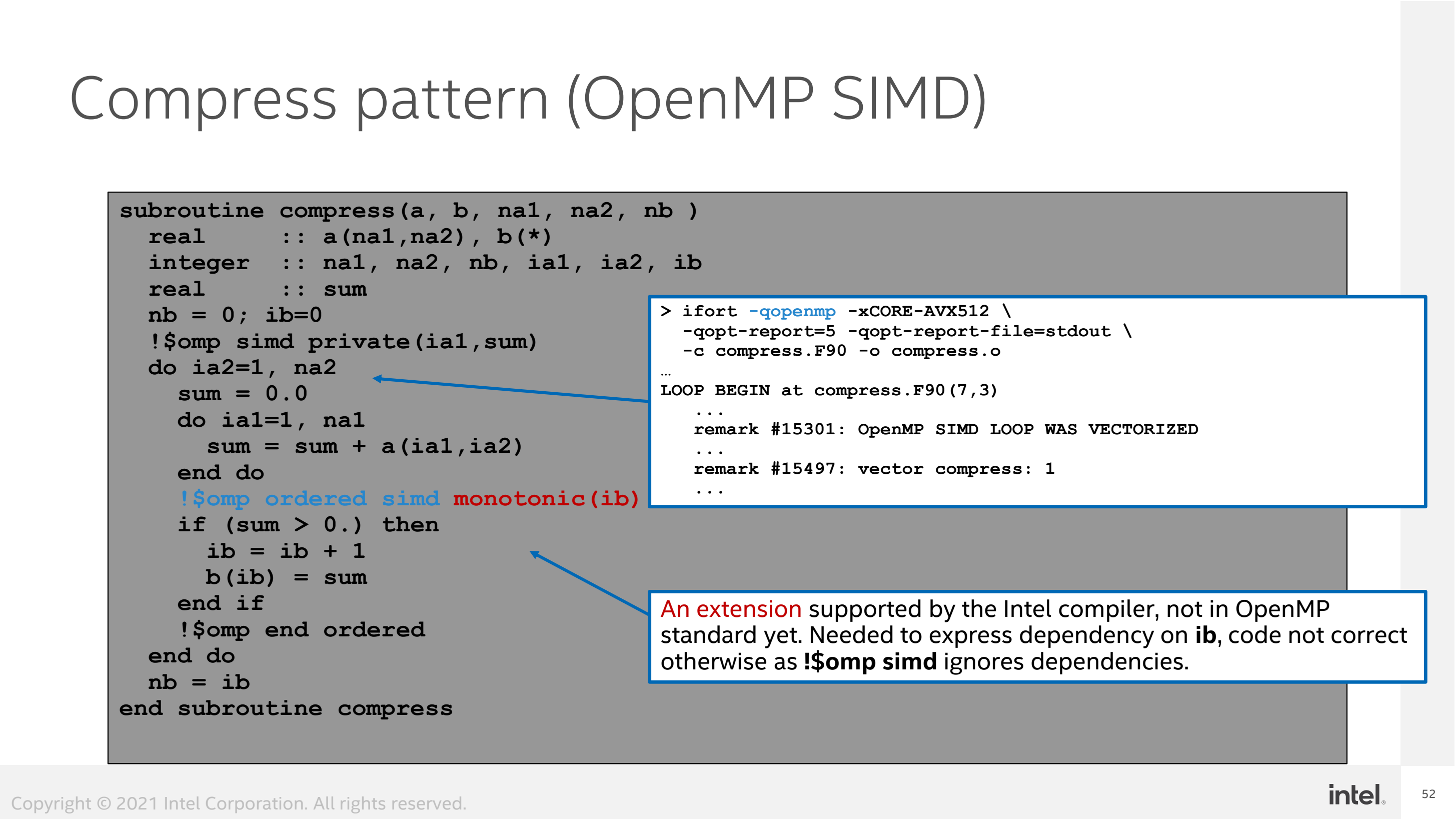


Compress pattern

```
subroutine compress(a, b, na, nb )
  implicit none
  real,    intent(in ) :: a(na)
  real,    intent(out) :: b(*)
  integer, intent(in)  :: na
  integer, intent(out) :: nb
  integer              :: ia
  nb = 0
  do ia=1, na
     if(a(ia) > 0.) then
        nb = nb + 1    ! dependency
        b(nb) = a(ia)  ! compress
     end if
  end do
end subroutine compress
```

```
> ifort -qopenmp -xCORE-AVX512 \
  -qopt-report=5 -qopt-report-file=stdout \
  -c compress.F90 -o compress.o
…
LOOP BEGIN at compress.F90(9,3)
remark #25084: Preprocess Loopnests: \
  Moving Out Store [ compress.F90(11,9) ]
…
remark #15300: LOOP WAS VECTORIZED
…
remark #15497: vector compress: 1
…
```

- Auto-vectorizes with Intel® AVX512 (**`vcompressps`** instruction)



Compress pattern (OpenMP SIMD)

```
subroutine compress(a, b, na1, na2, nb )
  real     :: a(na1,na2), b(*)
  integer  :: na1, na2, nb, ia1, ia2, ib
  real     :: sum
  nb = 0; ib=0
  !$omp simd private(ia1,sum)
  do ia2=1, na2
    sum = 0.0
    do ia1=1, na1
      sum = sum + a(ia1,ia2)
    end do
    !$omp ordered simd monotonic(ib)
    if (sum > 0.) then
      ib = ib + 1
      b(ib) = sum
    end if
    !$omp end ordered
  end do
  nb = ib
end subroutine compress
```

```
> ifort -qopenmp -xCORE-AVX512 \
  -qopt-report=5 -qopt-report-file=stdout \
  -c compress.F90 -o compress.o
…
LOOP BEGIN at compress.F90(7,3)
    ...
    remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
    ...
    remark #15497: vector compress: 1
    ...
```

An extension supported by the Intel compiler, not in OpenMP standard yet. Needed to express dependency on **ib**, code not correct otherwise as **!$omp simd** ignores dependencies.

Search loops

- A vectorizable loop must have a single exit and the iteration count must be known at the start of execution
  - Else a later iteration may have started before an earlier iteration decides the loop should be terminated
- Simple “search” loops are an exception which the compiler recognizes
  - executes special code if an exit occurs during a SIMD iteration
  - only works if no stores back to memory

Search pattern (simple)

```
integer function search(na, target, array)
  implicit none
  integer, intent(in) :: na, target, array(na)
  integer :: i

  do i=1,na
     if (array(i) == target) exit
  end do

  search = i
end function search
```

```
…
LOOP BEGIN at search.F90(6,3)
…
remark #15300: LOOP WAS VECTORIZED
…
```

- Search pattern auto-vectorizes if it contains no stores back to memory

Search pattern (with stores)

```
integer function search(a,b,c,n)
  implicit none
  real, dimension(n) :: a, b, c
  integer            :: n, i

  do i=1,n
     if (a(i) < 0.) exit
     c(i) = sqrt(a(i)) * b(i)
  end do

  search = i-1
end function search
```

```
LOOP BEGIN at search_store.F90(6,3)
remark #15520: loop was not vectorized: loop with multiple \
  exits cannot be vectorized unless it meets search loop   \
  idiom criteria [ search_store.F90(9,3) ]
LOOP END
```

- Search pattern with stores does not auto-vectorize

Search pattern (with stores, vectorized)

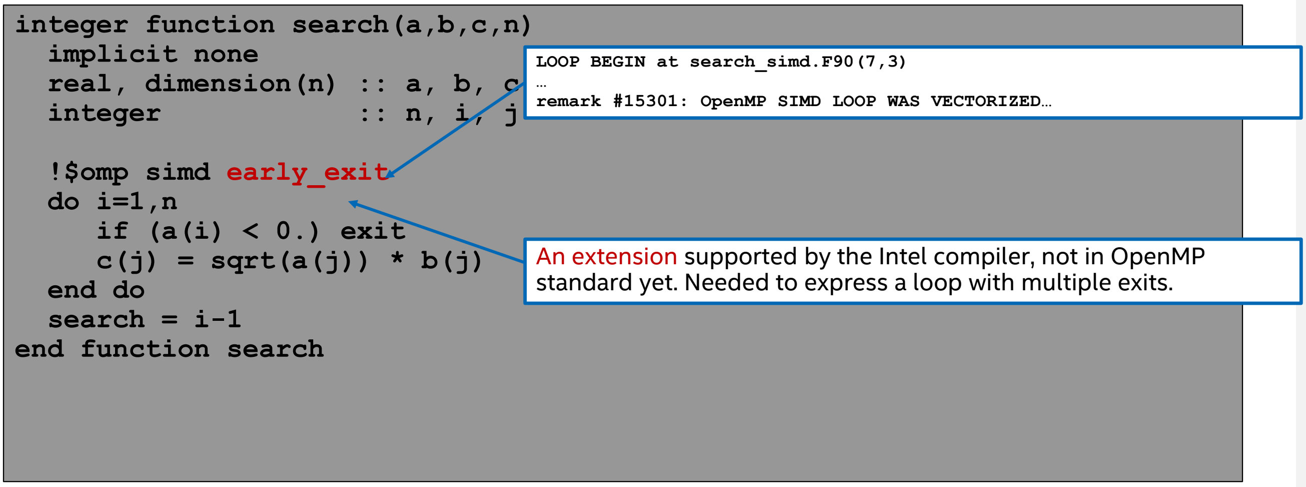
```
integer function search(a,b,c,n)
  implicit none
  real, dimension(n) :: a, b, c
  integer            :: n, i, j

  do i=1,n
     if (a(i) < 0.) exit
  end do
  search = i-1

  do j=1,search
     c(j) = sqrt(a(j)) * b(j)
  end do

end function search
```

```
LOOP BEGIN at search_split.F90(6,3)
…
remark #15300: LOOP WAS VECTORIZED
…
```

```
LOOP BEGIN at search_split.F90(11,3)
…
remark #15300: LOOP WAS VECTORIZED
…
```

- Splitting the loop enables vectorization with the cost of reloading **a**

Search pattern (with stores, OpenMP SIMD)



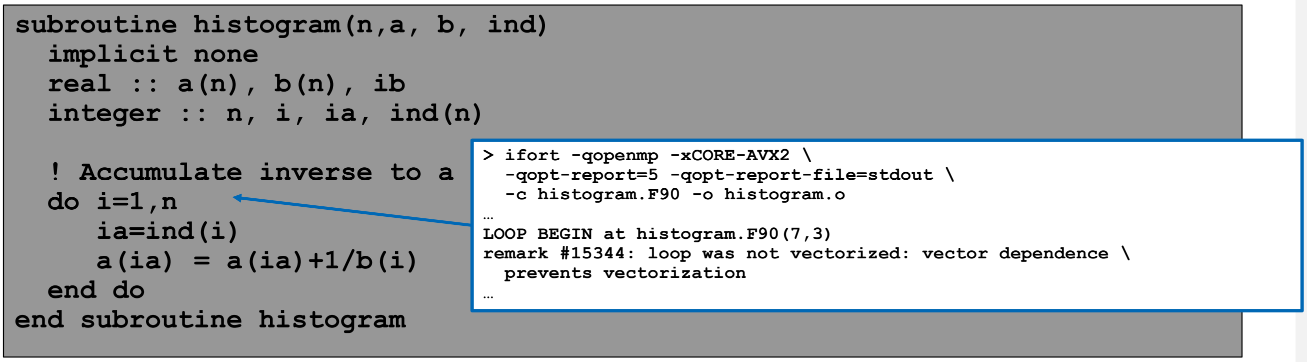
```
integer function search(a,b,c,n)
  implicit none
  real, dimension(n) :: a, b, c
  integer            :: n, i, j

  !$omp simd early_exit
  do i=1,n
     if (a(i) < 0.) exit
     c(j) = sqrt(a(j)) * b(j)
  end do
  search = i-1
end function search
```

```
LOOP BEGIN at search_simd.F90(7,3)
…
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED…
```

An extension supported by the Intel compiler, not in OpenMP standard yet. Needed to express a loop with multiple exits.

- OpenMP SIMD enables vectorization without the cost of reloading **a**

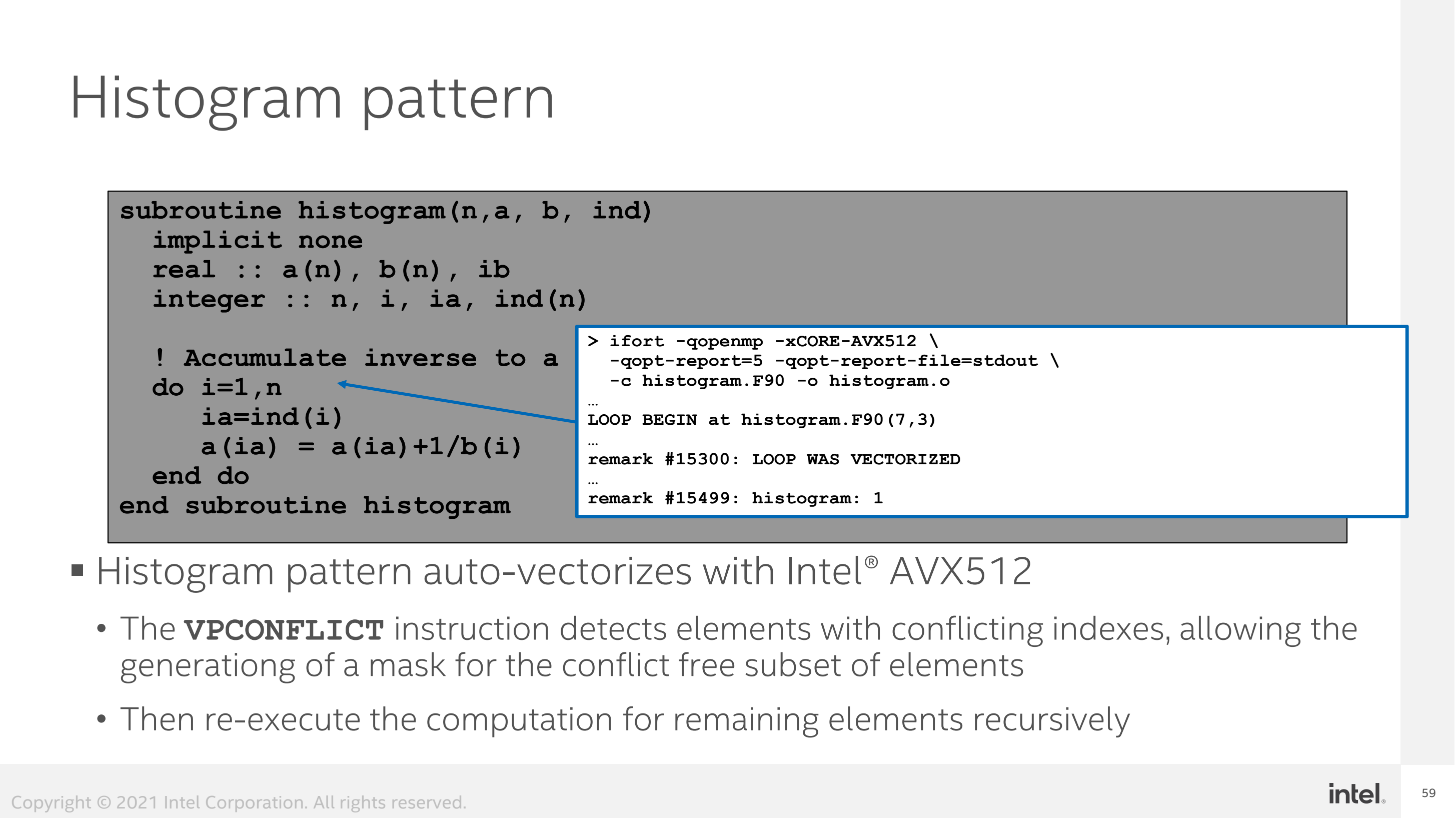
Histogram pattern



```
subroutine histogram(n,a, b, ind)
  implicit none
  real :: a(n), b(n), ib
  integer :: n, i, ia, ind(n)

  ! Accumulate inverse to a
  do i=1,n
     ia=ind(i)
     a(ia) = a(ia)+1/b(i)
  end do
end subroutine histogram
```

```
> ifort -qopenmp -xCORE-AVX2 \
  -qopt-report=5 -qopt-report-file=stdout \
  -c histogram.F90 -o histogram.o
…
LOOP BEGIN at histogram.F90(7,3)
remark #15344: loop was not vectorized: vector dependence \
  prevents vectorization
…
```

- Histogram pattern does not auto-vectorize with Intel® AVX2
  - Store to **a** is a scatter (indirect addressing) and **ia** can have the same value for different values of **i**
  - Vectorization with **!$omp simd** may cause incorrect results

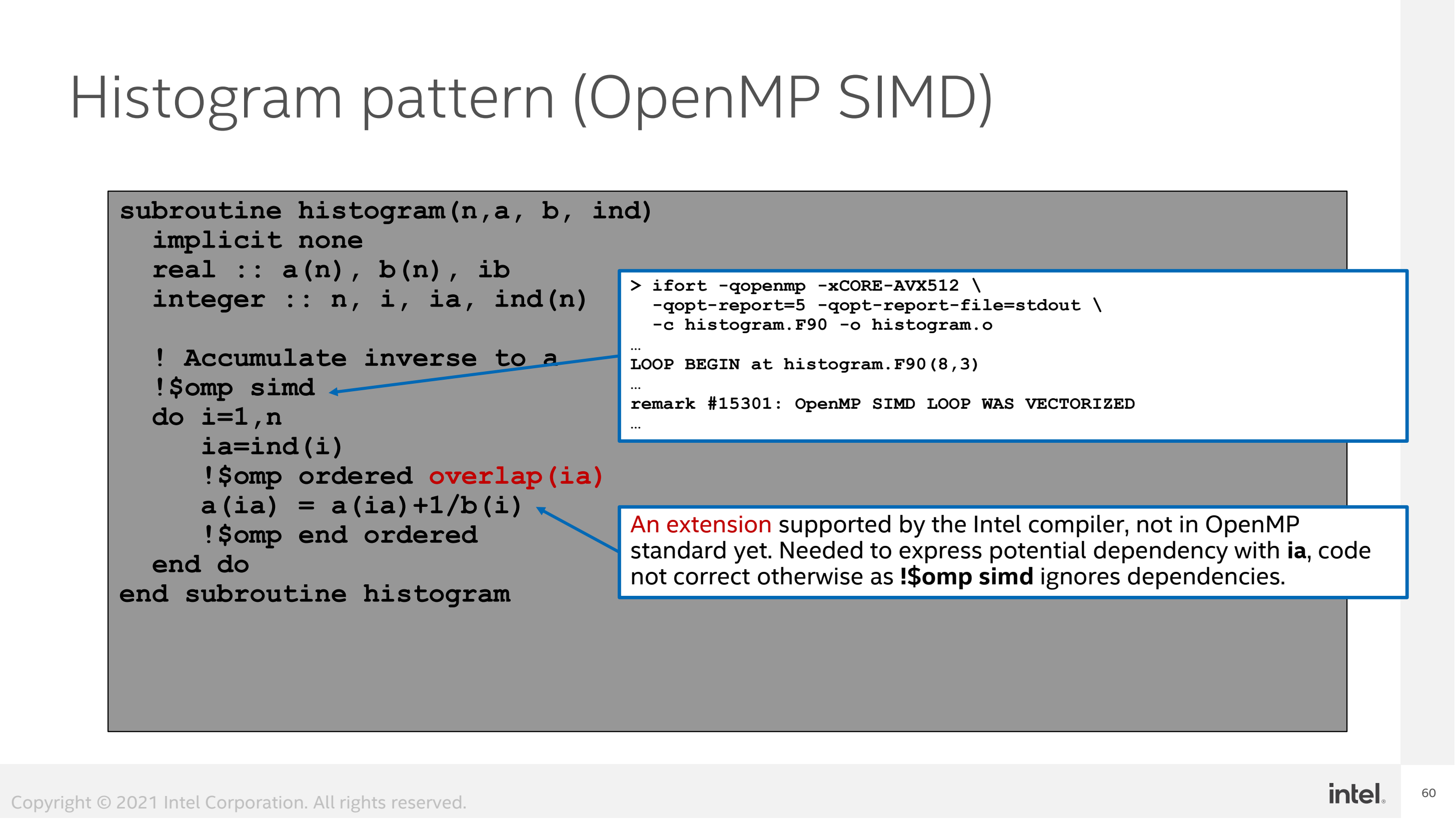


Histogram pattern

```
subroutine histogram(n,a, b, ind)
  implicit none
  real :: a(n), b(n), ib
  integer :: n, i, ia, ind(n)

  ! Accumulate inverse to a
  do i=1,n
     ia=ind(i)
     a(ia) = a(ia)+1/b(i)
  end do
end subroutine histogram
```

```
> ifort -qopenmp -xCORE-AVX512 \
  -qopt-report=5 -qopt-report-file=stdout \
  -c histogram.F90 -o histogram.o
…
LOOP BEGIN at histogram.F90(7,3)
…
remark #15300: LOOP WAS VECTORIZED
…
remark #15499: histogram: 1
```

- Histogram pattern auto-vectorizes with Intel® AVX512
  - The **`VPCONFLICT`** instruction detects elements with conflicting indexes, allowing the generationg of a mask for the conflict free subset of elements
  - Then re-execute the computation for remaining elements recursively



Histogram pattern (OpenMP SIMD)

```
subroutine histogram(n,a, b, ind)
  implicit none
  real :: a(n), b(n), ib
  integer :: n, i, ia, ind(n)

  ! Accumulate inverse to a
  !$omp simd
  do i=1,n
     ia=ind(i)
     !$omp ordered overlap(ia)
     a(ia) = a(ia)+1/b(i)
     !$omp end ordered
  end do
end subroutine histogram
```

```
> ifort -qopenmp -xCORE-AVX512 \
  -qopt-report=5 -qopt-report-file=stdout \
  -c histogram.F90 -o histogram.o
…
LOOP BEGIN at histogram.F90(8,3)
…
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
…
```

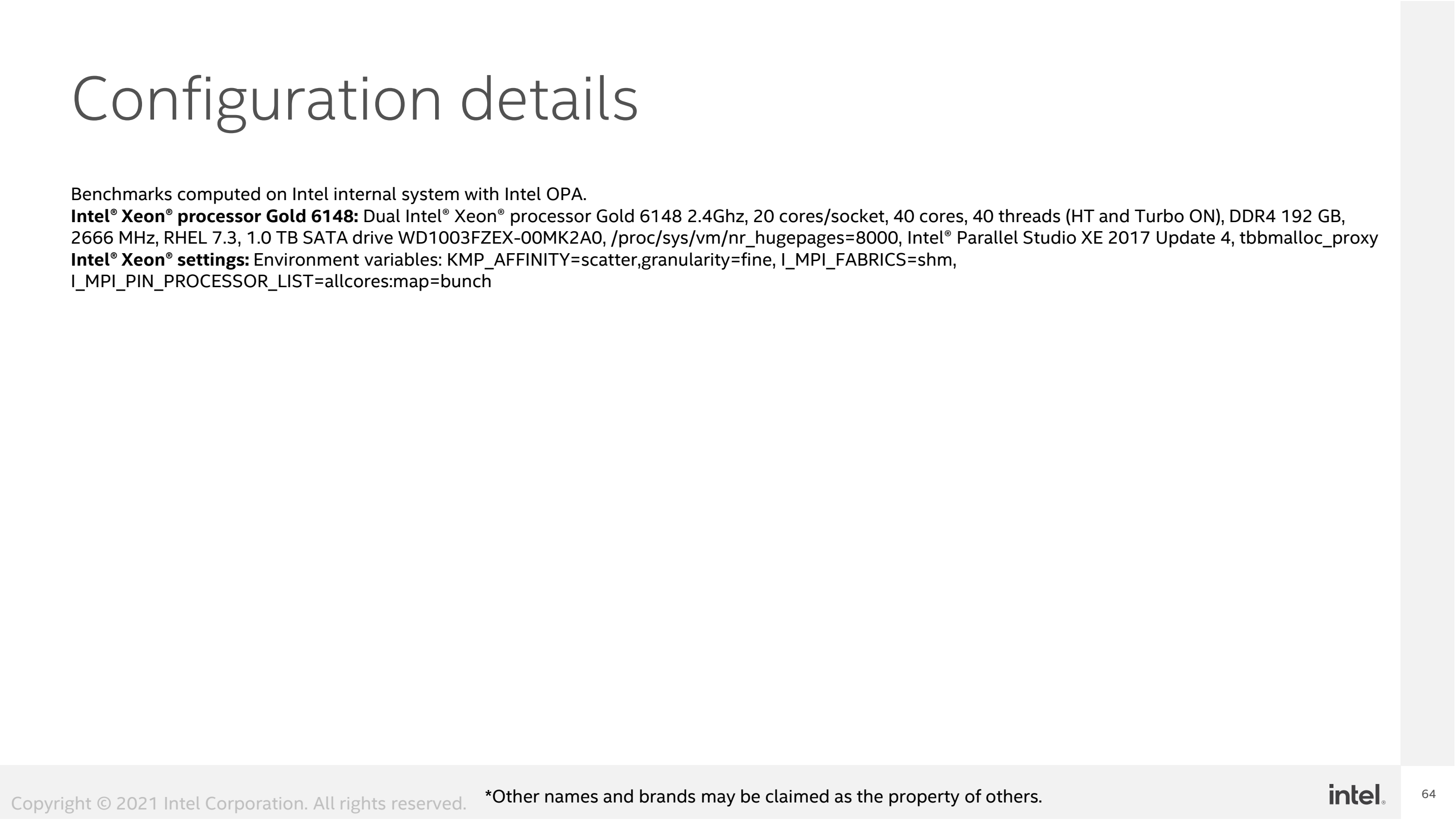
An extension supported by the Intel compiler, not in OpenMP standard yet. Needed to express potential dependency with **ia**, code not correct otherwise as **!$omp simd** ignores dependencies.

Histogram speed-up

- Speed-up depends on the problem details
  - Comes mostly from vectorization of other heavy computation in the loop, not from the scatter itself
  - Speed-up may be (much) less if there are many conflicts, for instance for histograms with a singularity or a narrow spike
  - Speed-up due to vectorization would be considerably higher on Intel® Xeon Phi™ x200 processors because scalar processor is slower.
- Many problems map to histograms
  - For instance: energy deposition in cells in particle transport Monte Carlo simulation, etc.

Summary

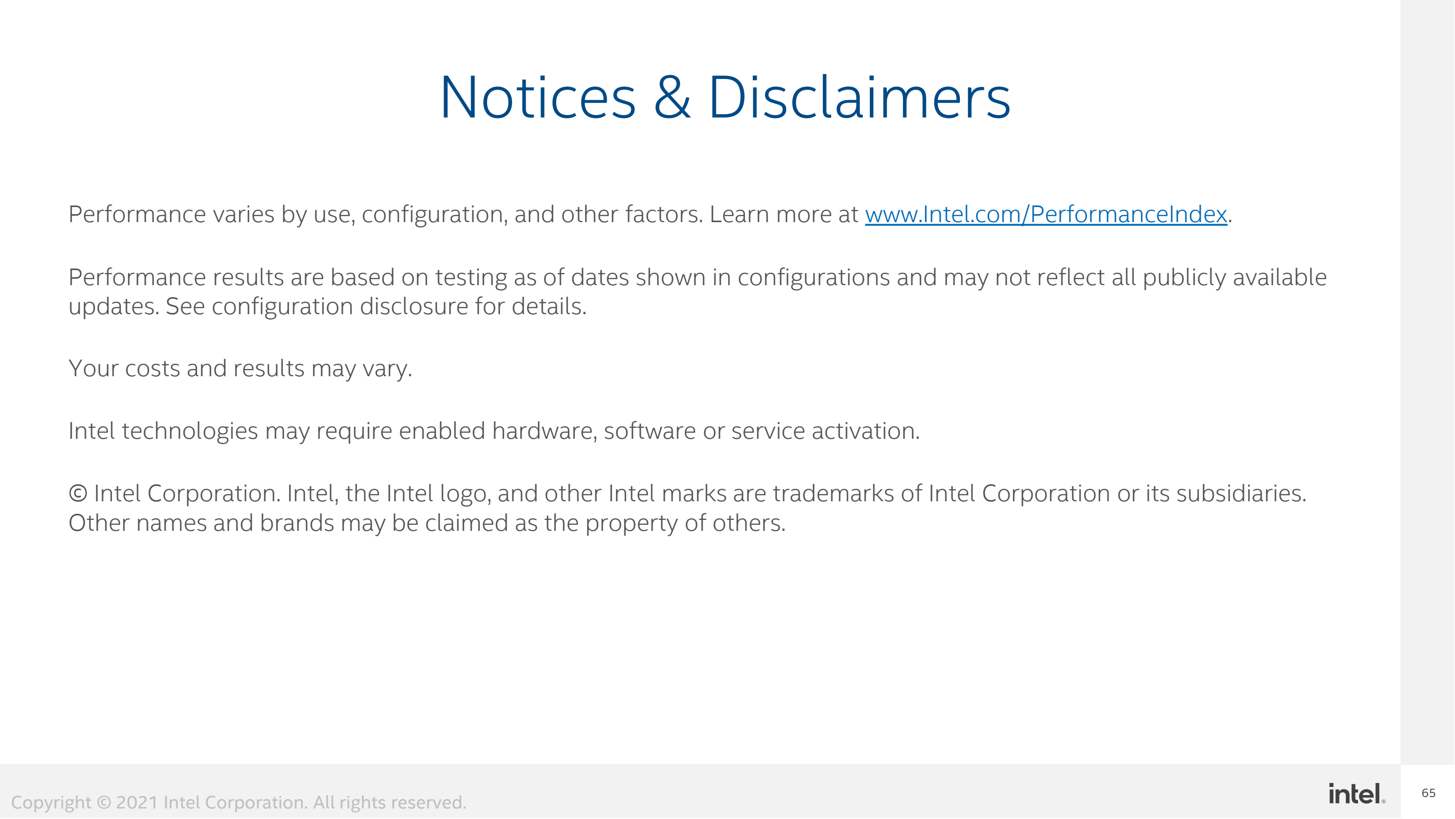
- With Intel® Xeon processors, vectorization (and multithreading) are the keys to good floating point performance
- Application may have to be modified to improve vectorization (and threading) properties
- OpenMP is a standardized way to program vectorized and multithreaded programs



Configuration details

Benchmarks computed on Intel internal system with Intel OPA.
Intel® Xeon® processor Gold 6148: Dual Intel® Xeon® processor Gold 6148 2.4Ghz, 20 cores/socket, 40 cores, 40 threads (HT and Turbo ON), DDR4 192 GB, 2666 MHz, RHEL 7.3, 1.0 TB SATA drive WD1003FZEX-00MK2A0, /proc/sys/vm/nr_hugepages=8000, Intel® Parallel Studio XE 2017 Update 4, tbbmalloc_proxy
Intel® Xeon® settings: Environment variables: KMP_AFFINITY=scatter,granularity=fine, I_MPI_FABRICS=shm, I_MPI_PIN_PROCESSOR_LIST=allcores:map=bunch

*Other names and brands may be claimed as the property of others.



Notices & Disclaimers

Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

Memory optimization

CSC Training, 2021-05

CSC – Finnish expertise in ICT for research, education and public administration

Outline

- Deeper view into data caches
- Basic considerations for cache efficiency
  - Loop traversal and interchange
  - Data structures
- Cache optimization techniques
  - Cache blocking

Deeper view into data caches

Data caches

- Modern CPUs use multilevel caches to access data
- Utilize spatial and temporal locality of data: if data is already in the cache, latency and bandwidth are improved
- For instance, on Intel Cascade lake
  - L1 cache: latency 4-6 cycles, sustained bandwidth 133 B/cycle/core
  - L2 cache: latency 14 cycles, sustained bandwidth 52 B/cycle/core
  - L3 cache: latency 50-70 cycles, sustained bandwidth 16 B/cycle/core
  - Main memory: latency 120-150 ns, bandwidth 128 GB/s per socket

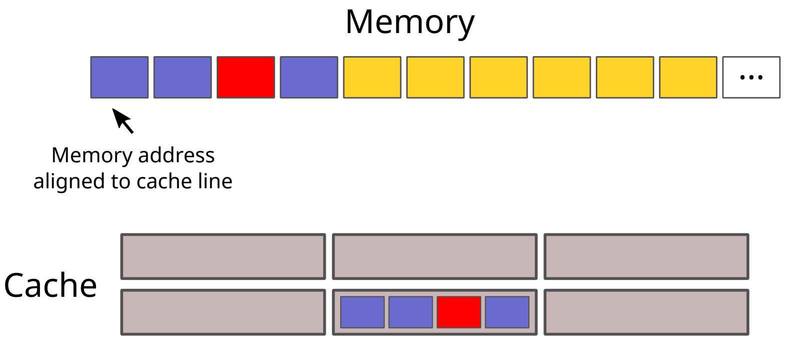
Data caches

- Sizes of the data caches are small compared to the main memory
  - L1 ~32 KiB
  - L2 512-1024 KiB
  - L3 1-4 MiB / core
- Terminology
  - *Cache hit*: the requested data is in the cache
  - *Cache miss*: the requested data is not in the cache
- Optimizing the use of caches is extremely important to leverage the full power of modern CPUs

Cache organization

- Cache is read and written in units of **cache lines**
  - 64 bytes in current x86 CPUs
- Upon *miss*, a line is *evicted* from the cache and replaced by the new line
  - Cache replacement policy determines which line is evicted
- *Inclusive* cache: all the lines in the upper-level cache are also in the lower level
- *Exclusive* cache: lines in the upper-level cache are not in the lower level
- Cache can be also non-inclusive non-exclusive, *i.e.* line may or may not be present in lower-level cache

Cache organization



Write policies

- Most modern CPUs employ a *write-back* cache write policy
  - a changed cache line is updated in the lower level hierarchy only when it is evicted
- Upon write miss, the cache line is typically first read from the main memory (*write-allocate* policy)
- In multicore CPUs with private caches, writes may require updates also in the caches of the other cores

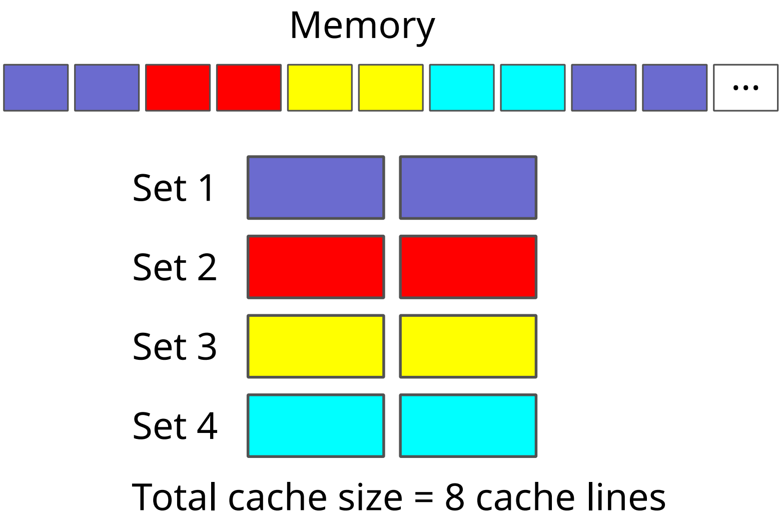
Cache associativity

- A cache with the size of 32 KiB can fit 32 KiB / 64 B = 512 cache lines
- In *fully associate* cache, each of the 512 entries can contain any memory location
  - Each entry needs to be checked for a hit which can be expensive for large caches
- In *direct mapped* cache, each memory location maps into exactly one cache line
  - Part of the cache is not fully utilized if memory addresses are not evenly distributed: some cache lines are evicted repeteadly while others remain empty
- Set associative caches can achieve best of the both worlds: efficient search and good utilization

Set associative cache

- A N-way set associative cache is divided into sets with N cache lines in each
  - 8-way set associative 32 KiB cache has 64 sets with 8 cache line entries per set
- A memory address is mapped into any entry within a **set**
  - need to search only over N entries for a hit
  - better utilization than in a direct mapped cache, but conflict misses still possible
- Fully associative and direct mapped as limiting cases N=∞ and N=1

Example: 2-way set associative cache



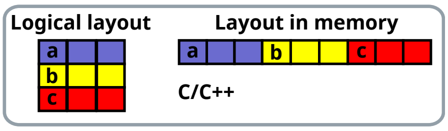
Types of cache misses

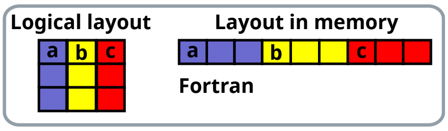
- Compulsory misses: happens the first time a memory address is accessed
  - Prefetching may prevent compulsory misses
- Capacity misses: happens when data the data is evicted due to cache becoming full
  - Can be caused by bad spatial and temporal locality of data in the application (inherent or bad implementation)
- Conflict misses: happens when a set becomes full even when other sets have space
  - Can be caused by particular memory access patterns

Optimizing data access

Accessing multidimensional arrays

- Accessing multidimensional arrays in incorrect order can generate poor cache behaviour
- Loops should written such that the *innermost* loop index matches the *contiguous* array index
  - C/C++ uses row major layout, i.e. last index is contiguous
  - Fortran uses column major layout, i.e. first index is contiguous





- Compiler optimizations may permute the loop indices automatically if possible

Loop interchange example: Fortran

Original loop

```
real :: a(N,M)
real :: sum

do i=1,N
  do j=1,M
     sum = sum + a(i,j)
  end do
end do
```

Interchanged

```
real :: a(N,M)
real :: sum

do j=1,M
  do i=1,N
     sum = sum + a(i,j)
  end do
end do
```

Loop interchange example: C/C++

Original loop

```
float **a;
float sum;

for (int i=0; i < M; i++)
  for (int j=0; j < N; j++)
    sum = sum + a[j][i];
```

Interchanged

```
float **a;
float sum;

for (int j=0; j < N; j++)
  for (int i=0; i < M; i++)
    sum = sum + a[j][i];
```

Data structures

- Data structure choice has an effect on the memory layout
  - Structure of arrays (SoA) vs. Array of Structures (AoS)
- Data should be stored based on its usage pattern
  - Avoid scattered memory access
- Occasionally, use of nonconventional ordering or traversal of data is beneficial
  - Colorings, space filling curves, *etc.*

Data structures: memory layout

Array of Structures

```
type point
  real :: x, y, z
end type point

type(point), allocatable :: points

allocate(points(N))
```

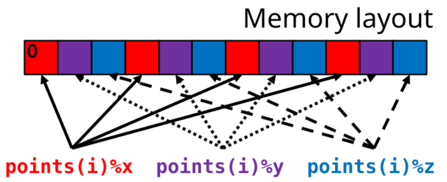
Structure of Arrays

```
type point
  real, allocatable :: x(:)
  real, allocatable :: y(:)
  real, allocatable :: z(:)
end type point

type(point) :: points

allocate(points%x(N), &
         points%y(N), &
         points%z(N))
```

Data structures: memory layout

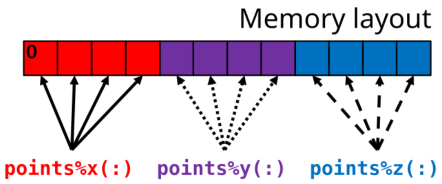
Array of Structures

```
integer :: i, j
real :: dist(4,4)
do i = 1, 4
  do j = i, 4
    dist(i,j) = sqrt( &
      (points(i)%x-points(j)%x)**2 + &
      (points(i)%y-points(j)%y)**2 + &
      (points(i)%z-points(j)%z)**2)
  end do
end do
```



Structure of Arrays

```
integer :: i, j
real :: dist(4,4)
do i = 1, 4
  do j = i, 4
    dist(i,j) = sqrt( &
      (points%x(i)-points%x(j))**2 + &
      (points%y(i)-points%y(j))**2 + &
      (points%z(i)-points%z(j))**2)
  end do
end do
```



Cache blocking

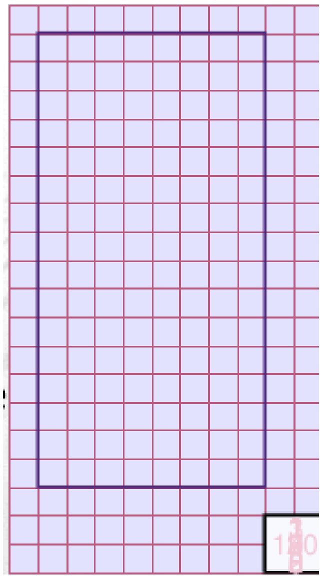
- Multilevel loops can be iterated in blocks in order improve data locality
  - Perform more computations with the data that is already in the cache
- Complicated optimization: optimal block size is hardware dependent (cache sizes, SIMD width, *etc.*)
- Cache oblivious algorithms use recursion to improve performance portability

Cache blocking example

- Consider a 2D Laplacian

```
do j=1, 8
  do i=1, 16
    a(i,j) = u(i-1, j) + u(i+1, j) &
           - 4*u(i,j)              &
           + u(i,j-1) + u(i,j+1)
  end do
end do
```

- (Fictitious) cache structure
  - Each line holds 4 elemets
  - Cache can hold 12 lines of data
- No cache reuse between outer loop iterations

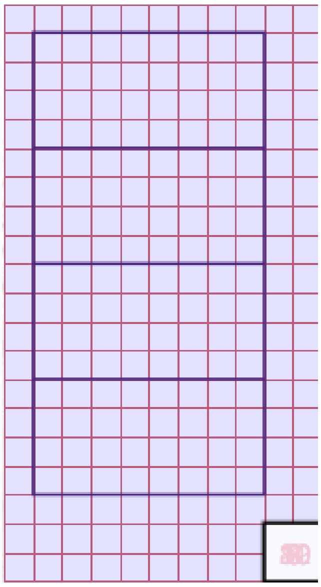


Cache blocking example

- Blocking the inner loop

```
do IBLOCK = 1, 16, 4
  do j=1, 8
    do i=1, IBLOCK, IBLOCK + 3
      a(i,j) = u(i-1, j) + u(i+1, j) &
             - 4*u(i,j)              &
             + u(i,j-1) + u(i,j+1)
    end do
  end do
end do
```

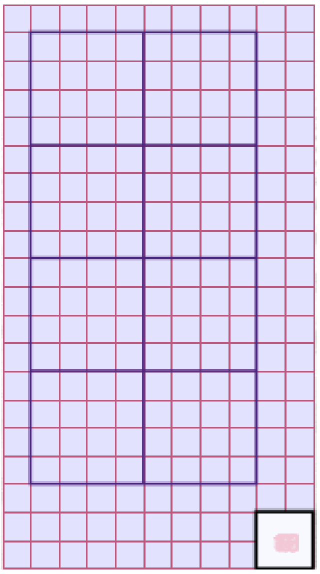
- Better reuse for the j+1 data



Cache blocking example

- Iterate over 4x4 blocks

```
do JBLOCK = 1, 8, 4
  do IBLOCK = 1, 16, 4
    do j=JBLOCK, JBLOCK + 3
      do i=1, IBLOCK, IBLOCK + 3
        a(i,j) = u(i-1, j) + u(i+1, j) &
               - 4*u(i,j)              &
               + u(i,j-1) + u(i,j+1)
      end do
    end do
  end do
end do
```



Cache blocking with OpenMP

- OpenMP 5.1 standard has `tile` construct for blocking
  - Compiler support not necessarily ready yet

```
!$omp tile sizes(4, 4)
do j=1, 8
  do i=1, 16
    a(i,j) = u(i-1, j) + u(i+1, j) &
           - 4*u(i,j)              &
           + u(i,j-1) + u(i,j+1)
  end do
end do
!$omp end tile
```

Array padding

- When data is accessed in strides which are multiple of the cache set size, conflict misses may occur
  - In 8-way associative 32 KiB cache, there are 64 sets
  - Memory address which are 64*64 = 4096 bytes apart map into a same set
  - Example: in `float a[1024][1024]` each column maps into a same set
- Array padding, *i.e.* allocating extra data can in some cases reduce conflict misses
  - `float a[1024 + 16][1024]`
  - Padding should preferably preserve alignment of data

Prefetching

- Modern CPUs try to predict data usage patterns and prefetch data to caches before it is actually needed
  - Can alleviate even compulsory misses
- Prefetching can be requested also by software
  - Compiler
  - Programmer via software directives and intrinsinc functions
  - Difficult optimization:
    - Too early: cache is filled with unnecessary data
    - Too late: CPU has to wait for the data

Non-temporal stores

- With *write-allocate* policy, a write miss incurs a load from main memory
- If data is going to be just written and not reused, some CPUs contain instructions for bypassing the cache by writing directly into the memory with *non-temporal stores*
- Non-temporal stores can be used via pragmas, compiler options, or intrinsincs
  - `omp simd nontemporal(list)` (OpenMP 5.0)
  - Possible benefits depend a lot on application, and misuse can degragade performance
  - Hardware may also recognize access pattern and switch into non-temporal stores

Summary

- Efficient cache usage is on of the most important aspects for achieving good performance
  - Exploite spatial and temporal locality
- Progammer can improve the cache usage by optimizing data layouts and access patterns

Miscellaneous single core optimizations

CSC Training, 2021-05

CSC – Finnish expertise in ICT for research, education and public administration

Outline

- Loop transformations
- Mathematical routines
- Branches
- Function inlining
- Intrincic functions

Loop transformations

Loop transformations

- Loop transformations can provide better vectorization prospects, improve instruction level parallelism, pipeline utilization and cache usage
- Common transformations: interchange, unrolling, fusion, fission, sectioning, unroll and jam
- In many cases compiler can make loop transformations with high enough optimization level
  - Understanding the concepts is still be useful for the programmer
- In some cases manual programming can be useful
  - When misused, transformation can be disadvantageous for performance
  - Readability of code often suffers

Loop unrolling

- If the loop body is very small, overhead from incrementing the loop counter and from the test for the end of the loop can be high
- When vectorizing, loop is implicitly unrolled by the vector length
- May improve pipeline utilization and instruction level parallelism
- Additional logic needed for remainder
- May increase register pressure

```
do i=1,N
  c[i] = a[i] + b[i]
end do
```

```
do i=1,N,4  ! unroll four times
  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]
end do
```

Loop fission

- Loop fission (or loop distribution) splits one loop into sequence of loops
- May improve cache usage and reduce register pressure
- May allow vectorization by moving dependencies
- Some dependencies may prohibit fission

```
do j=1,N
  b(i) = a(i) * a(i)
  d(i) = c(i) - d(i-1)     ! flow dependency
end do
```

```
do j=1,N  ! vectorization possible
  b(i) = a(i) * a(i)
end do
do j=1,N
  d(i) = c(i) - d(i-1)
end do
```

Loop fusion

- Loop fusion (or loop jamming) merges multiple loops into one
- May improve cache usage
- May allow better pipeline utilization and instruction level parallelism
- May cause dependencies which prevent applying the transformation

```
do j=1,N
  b(i) = a(i) * a(i)
end do
do j=1,N
  c(i) = c(i) * a(i)
end do
```

```
do j=1,N
  b(i) = a(i) * a(i)
  c(i) = c(i) * a(i)
end do
```

Loop sectioning

- Loop sectioning (or strip mining) transforms a loop into smaller chunks by creating additional inner loops
- May improve cache usage
- May make the code easier for compiler to vectorize

```
do i=1,N
  process1(data(i))
  process2(data(i))
end do
```

```
do i=1,N,S
  do j=i, min(N, i + S)
     process1(data(i))
  end do
  do j=i, min(N, i + S)
    process2(data(i))
  end do
end do
```

Loop unroll and jam

- Unroll and jam unrolls an outer loop and fuses then the inner loop
- May allow better pipeline utilization and instruction level parallelism
- May potentiate other optimizations

```
do i=1,N
  do j=1,M
    b = 2 * a(i, j)
    c(i,j) = b * b
  end do
end do
```

```
do j=1,N,2
  do i=1,M
    b1 = 2 * a(i, j)
    b2 = 2 * a(i, j + 1)
    c(i, j) = b1*b1
    c(i, j + 1) = b2*b2
  end do
end do
```

Other optimizations

Optimizing mathematical operations

- Due to finite precision of floating point numbers, compilers need to be carefull in some optimizations

  (a + b) + c ≠ a + (b + c)
- Some mathematical routines (`sqrt`, pow, `sin`, cos, ...) can be calculated with different algorithms with different performance and precision
  - In some applications it is possible to compromise precision for speed
- Most compilers have an option for faster mathematics ('`-ffast-math`' for gcc/clang and '`-fp-model fast=2`' for Intel)
  - Important to check that results are valid !

Optimizing mathematical operations

- If *fast math* options cannot be use (*i.e.* part of the application requires higher precision), programmer can make some optimizations by hand
- Examples:
  - Move division out of the loop
  - Replace `pow(x, n)` where n is small integer with multiplications (C/C++)

```
do i=1, n
  do j=1, m
    L(i,j) = (A(i-1,j) - 2.0*A(i,j) + A(i+1,j)) / dx**2 + &
             (A(i,j-1) - 2.0*A(i,j) + A(i,j+1)) / dx**2
  end do
end do
```

vs.

```
idx2 = 1.0 / dx**2
do i=1, n
  do j=1, m
    L(i,j) = (A(i-1,j) - 2.0*A(i,j) + A(i+1,j)) * idx2 + &
             (A(i,j-1) - 2.0*A(i,j) + A(i,j+1)) * idx2
  end do
end do
```

```
double x3 = x*x*x  // instead of pow(x, 3)
```

Optimizing branches

- Branches have the possibility of stalling the CPU pipeline, and can thus be expensive
- When possible, `if` statements should be outside loop bodies
  - manual loop transformations can be helpful
- Hardware branch predictor works well when the branching follows regular pattern
  - performing extra work for improving predictability may be worthwhile

Inline functions

- When inlining, compiler replaces a call to function by the function body
  - Reduces function call overhead
  - If function is called within a loop, may provide additional optimization prospects
- Compiler uses heuristics to decide if inlining is beneficial
  - Might require "interprocedural optimization" options
- In C/C++ `inline` keyword is *hint* for the compiler to inline
- In Fortran, programmer can force inlining only via compiler directives, otherwise compiler makes the decision whether to inline a function
- Overuse of inlining increases the executable size and may hurt performance

Intrinsic functions

- Intrinsic functions are special functions that the compiler replaces with equivalent CPU instruction
  - "high level assembly"
  - Often compiler specific
- Examples:
  - Software prefetch: `_mm_prefetch` (C/C++), `mm_prefetch` (Fortran)
  - Non-temporal stores: `_mm_stream_xxx` (C/C++ only)
  - AVX instructions
- Recommended only in special cases
  - Can make the code non-portable
  - Can also degragade performance - compiler might know better when to use

Summary

- Loops can be transformed in various ways in order to improve performance
  - Often better leave the transformations for the compiler
- Many mathematical operations can be performed faster with some compromise on precision
- Hard to predict branches may stall the CPU pipeline

Web resources

- Intel Intrinsics guide: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Programming OpenMP

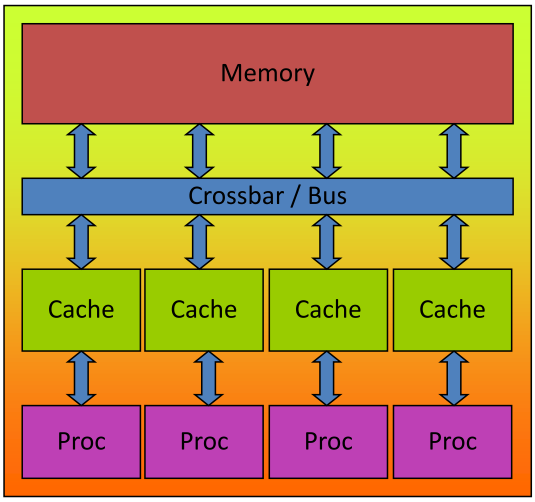
Parallel Region

Michael Klemm

Credit for these slides go to the OpenMP tutorial gang:
Bronis R. de Supinski, Christian Terboven, Ruud van der Pas, Xavier Teruel

OpenMP‘s machine model

- OpenMP: Shared-Memory Parallel Programming Model.



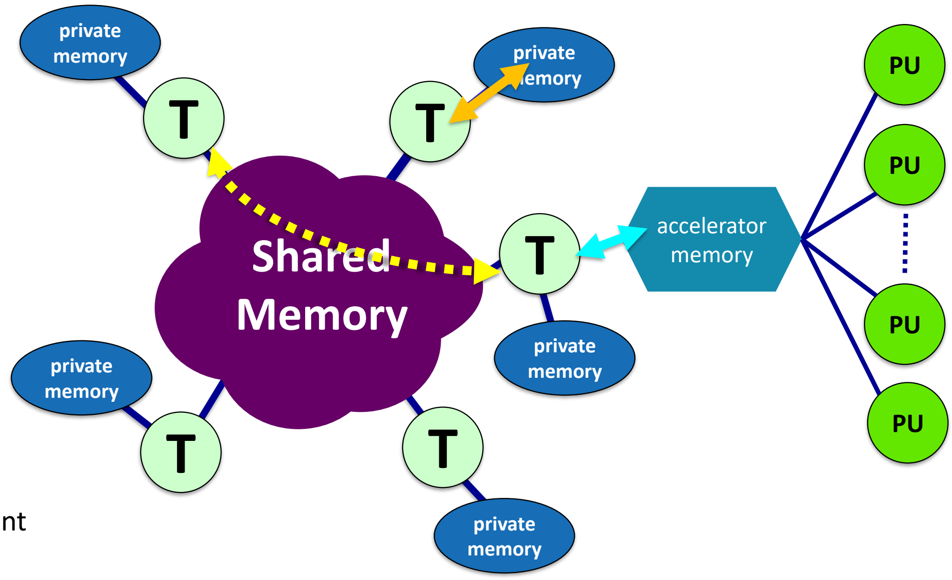
All processors/cores access a shared main memory.

Real architectures are more complex, as we will see later / as we

Parallelization in OpenMP employs multiple threads.

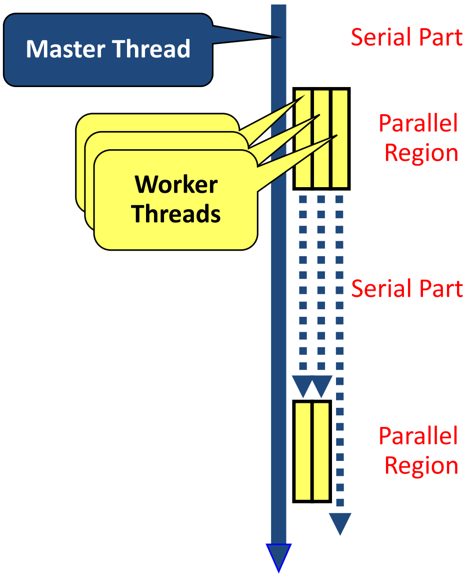
The OpenMP Memory Model

- All threads have access to the same, globally shared memory
- Data in private memory is only accessible by the thread owning this memory
- No other thread sees the change(s) in private memory
- Data transfer is through shared memory and is 100% transparent to the application



The OpenMP Execution Model

- OpenMP programs start with just one thread: The *Master*.
- *Worker* threads are spawned at *Parallel Regions*, together with the Master they form the *Team* of threads.
- In between Parallel Regions the Worker threads are put to sleep. The OpenMP *Runtime* takes care of all thread management work.
- Concept: *Fork-Join*.
- Allows for an incremental parallelization!



Parallel Region and Structured Blocks

- The parallelism has to be expressed explicitly.

C/C++

```
#pragma omp parallel
{
   ...
   structured block
   ...
}
```

Fortran

```
!$omp parallel
   ...
   structured block
   ...
!$omp end parallel
```

- *Structured Block*
  - Exactly one entry point at the top
  - Exactly one exit point at the bottom
  - Branching in or out is not allowed
  - Terminating the program is allowed (abort / exit)

- *Specification of number of threads:*
  - Environment variable: `OMP_NUM_THREADS=`…
  - Or: Via `num_threads` clause: add `num_threads(num)` to the parallel construct

Starting OpenMP Programs on Linux

- From within a shell, global setting of the number of threads:

```
export OMP_NUM_THREADS=4
./program
```

- From within a shell, one-time setting of the number of threads:

```
OMP_NUM_THREADS=4   ./program
```

Programming OpenMP

Tasking Introduction

Sudoko for Lazy Computer Scientists

- Lets solve Sudoku puzzles with brute multi-core force

	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

- (1) Search an empty field
- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field
- Wait for completion

Parallel Brute-force Sudoku

- This parallel algorithm finds all valid solutions

	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

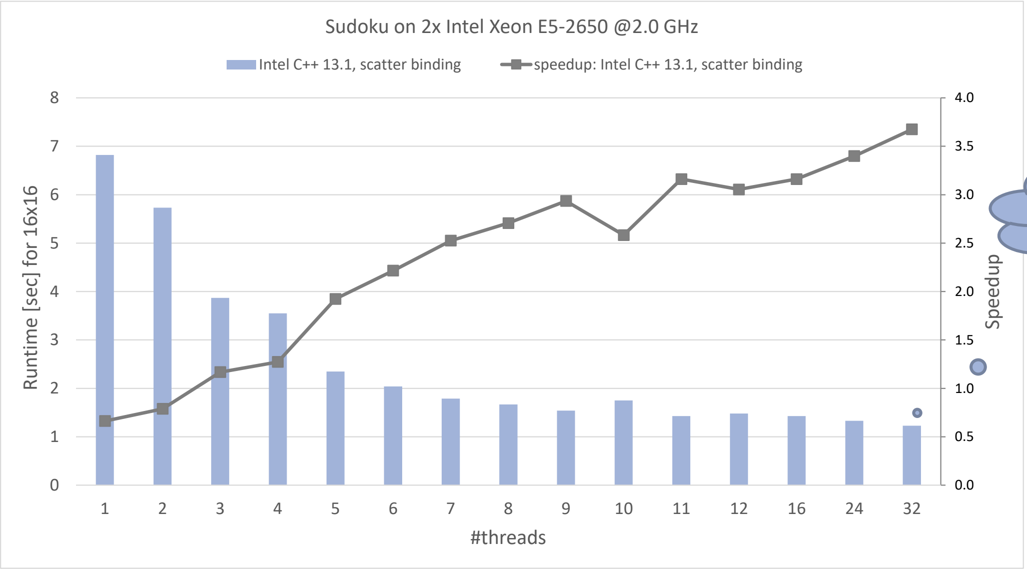
- (1) Search an empty field
- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field
- Wait for completion

first call contained in a
`#pragma omp parallel`
`#pragma omp single`
such that one tasks starts the execution of the algorithm

`#pragma omp task`
needs to work on a new copy of the Sudoku board

`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation



Is this the best we can can do?

Tasking Overview

What is a task in OpenMP?

- Tasks are work units whose execution
  - → may be deferred or…
  - → … can be executed immediately
- Tasks are composed of
  - → **code** to execute, a **data** environment (initialized at creation time), internal **control** variables (ICVs)
- Tasks are created…

  … when reaching a parallel region → implicit tasks are created (per thread)

  … when encountering a task construct → explicit task is created

  … when encountering a taskloop construct → explicit tasks per chunk are created

  … when encountering a target construct → target task is created

Tasking execution model

- Supports unstructured parallelism
  - → unbounded loops

```
while ( <expr> ) {
   ...
}
```

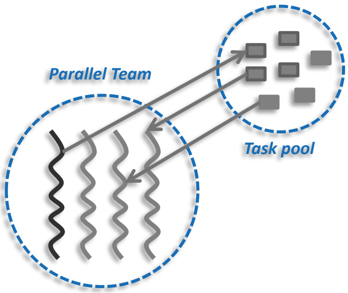
  - → recursive functions

```
void myfunc( <args> )
{
   ...; myfunc( <newargs> ); ...;
}
```

- Several scenarios are possible:
  - → single creator, multiple creators, nested tasks (tasks & WS)
- All threads in the team are candidates to execute tasks

- Example (unstructured parallelism)

```
#pragma omp parallel
#pragma omp master
while (elem != NULL) {
   #pragma omp task
      compute(elem);
   elem = elem->next;
}
```



The task construct

- Deferring (or not) a unit of work (executable for any member of the team)

```
#pragma omp task [clause[[,] clause]...]
{structured-block}
```

```
!$omp task [clause[[,] clause]...]
…structured-block…
!$omp end task
```

- Where clause is one of:

Clause	Category
→ private(list) → firstprivate(list) → shared(list) → default(shared | none) → in_reduction(r-id: list)	Data Environment
→ allocate([allocator:] list) → detach(event-handler)	Miscellaneous
→ if(scalar-expression) → mergeable → final(scalar-expression)	Cutoff Strategies
→ depend(dep-type: list)	Synchronization
→ untied → priority(priority-value) → affinity(list)	Task Scheduling

Task scheduling: tied vs untied tasks

- Tasks are tied by default (when no untied clause present)
  - → tied tasks are executed always by the same thread (not necessarily creator)
  - → tied tasks may run into performance problems
- Programmers may specify tasks to be untied (relax scheduling)

```
#pragma omp task untied
{structured-block}
```

  - → can potentially switch to any thread (of the team)
  - → bad mix with thread based features: thread-id, threadprivate, critical regions...
  - → gives the runtime more flexibility to schedule tasks
  - → but most of OpenMP implementations doesn't “honor” untied ☹

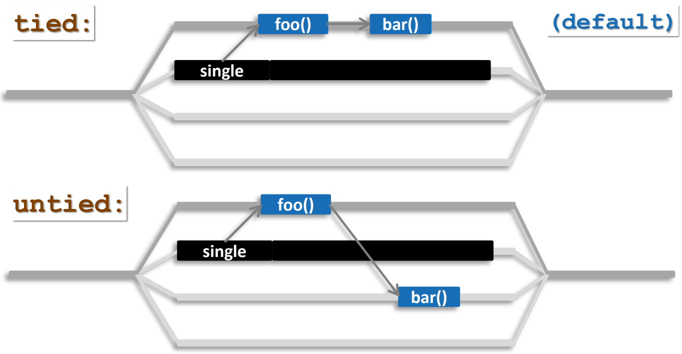
Task scheduling: taskyield directive

- Task scheduling points (and the taskyield directive)
  - → tasks can be suspended/resumed at TSPs → some additional constraints to avoid deadlock problems
  - → implicit scheduling points (creation, synchronization, ...)
  - → explicit scheduling point: the taskyield directive

```
#pragma omp taskyield
```

- Scheduling [tied/untied] tasks: example

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task untied
   {
      foo();
      #pragma omp taskyield
      bar()
   }
}
```



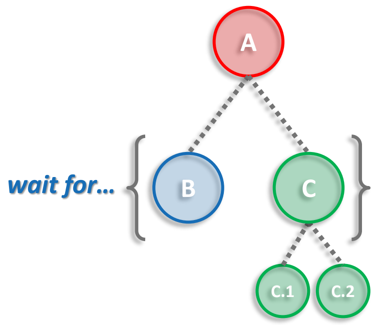
Task synchronization: taskwait directive

- The taskwait directive (shallow task synchronization)
  - → It is a stand-alone directive

```
#pragma omp taskwait
```

  - → wait on the completion of child tasks of the current task; just direct children, not all descendant tasks; includes an implicit task scheduling point (TSP)

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task                     : A
   {
      #pragma omp task         : B
      { … }
      #pragma omp task         : C
      { … #C.1; #C.2; …}
      #pragma omp taskwait
   }
} // implicit barrier will wait for C.x
```



Task synchronization: barrier semantics

- OpenMP barrier (implicit or explicit)
  - → All tasks created by any thread of the current team are guaranteed to be completed at barrier exit

```
#pragma omp barrier
```

  - → And all other implicit barriers at parallel, sections, for, single, etc…

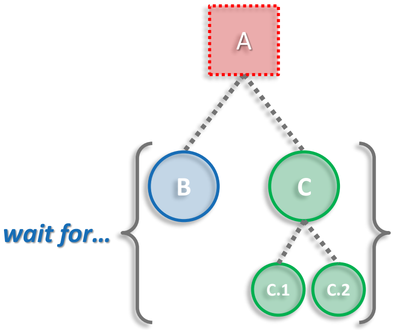
Task synchronization: taskgroup construct

- The taskgroup construct (deep task synchronization)
  - → attached to a structured block; completion of all descendants of the current task; TSP at the end

```
#pragma omp taskgroup [clause[[,] clause]...]
{structured-block}
```

  - → where clause (could only be): reduction(reduction-identifier: list-items)

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp taskgroup                 : A
   {
      #pragma omp task                   :B
      { … }
      #pragma omp task                   :C
      { … #C.1; #C.2; …}

   } // end of taskgroup
}
```



Data Environment

Explicit data-sharing clauses

OpenMP

- Explicit data-sharing clauses (shared, private and firstprivate)

```
#pragma omp task shared(a)
{
  // Scope of a: shared
}
```

```
#pragma omp task private(b)
{
  // Scope of b: private
}
```

```
#pragma omp task firstprivate(c)
{
  // Scope of c: firstprivate
}
```

- If **default** clause present, what the clause says
  - → shared: data which is not explicitly included in any other data sharing clause will be **shared**
  - → none: compiler will issue an error if the attribute is not explicitly set by the programmer (very useful!!!)

```
#pragma omp task default(shared)
{
 // Scope of all the references, not explicitly
 // included in any other data sharing clause,
 // and with no pre-determined attribute: shared
}
```

```
#pragma omp task default(none)
{
 // Compiler will force to specify the scope for
 // every single variable referenced in the context
}
```

Hint: Use default(none) to be forced to think about every variable if you do not see clearly.

Pre-determined data-sharing attributes

OpenMP

- threadprivate variables are threadprivate **(1)**
- dynamic storage duration objects are shared (malloc, new,…) **(2)**
- static data members are shared **(3)**
- variables declared inside the construct
  - → static storage duration variables are shared **(4)**
  - → automatic storage duration variables are private **(5)**
- the loop iteration variable(s)…

5

```
#pragma omp task
{
   int x = MN;
   // Scope of x: private
}
```

4

```
#pragma omp task
{
   static int y;
   // Scope of y: shared
}
```

1

```
int A[SIZE];
#pragma omp threadprivate(A)

// ...
#pragma omp task
{
  // A: threadprivate
}
```

2

```
int *p;

p = malloc(sizeof(float)*SIZE);

#pragma omp task
{
   // *p: shared
}
```

3

```
void foo(void){
   static int s = MN;
}

#pragma omp task
{
   foo(); // s@foo(): shared
}
```

Implicit data-sharing attributes (in-practice)

- Implicit data-sharing rules for the task region
  - → the **shared** attribute is lexically inherited
  - → in any other case the variable is **firstprivate**

```
int a = 1;
void foo() {
   int b = 2, c = 3;
   #pragma omp parallel private(b)
   {
      int d = 4;
      #pragma omp task
      {
         int e = 5;
         // Scope of a:
         // Scope of b:
         // Scope of c:
         // Scope of d:
         // Scope of e:
      }
   }
}
```

→ Pre-determined rules (could not change)

→ Explicit data-sharing clauses (+ default)

→ Implicit data-sharing rules

- (in-practice) variable values within the task:
  - → value of a: 1
  - → value of b: x // undefined (undefined in parallel)
  - → value of c: 3
  - → value of d: 4
  - → value of e: 5

Task reductions (using taskgroup)

- Reduction operation
  - → perform some forms of recurrence calculations
  - → associative and commutative operators
- The (taskgroup) scoping reduction clause

```
#pragma omp taskgroup task_reduction(op: list)
{structured-block}
```

  - → Register a new reduction at [1]
  - → Computes the final result after [3]
- The (task) in_reduction clause [participating]

```
#pragma omp task in_reduction(op: list)
{structured-block}
```

  - → Task participates in a reduction operation [2]

```
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp taskgroup task_reduction(+: res)
    { // [1]
      while (node) {
        #pragma omp task in_reduction(+: res) \
                firstprivate(node)
        { // [2]
          res += node->value;
        }
        node = node->next;
      }
    } // [3]
  }
}
```

Task reductions (+ modifiers)

- Reduction modifiers
  - → Former reductions clauses have been extended
  - → task modifier allows to express task reductions
  - → Registering a new task reduction [1]
  - → Implicit tasks participate in the reduction [2]
  - → Compute final result after [4]
- The (task) in_reduction clause [participating]

```
#pragma omp task in_reduction(op: list)
{structured-block}
```

  - → Task participates in a reduction operation [3]

```
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel reduction(task,+: res)
{ // [1][2]
  #pragma omp single
  {
    #pragma omp taskgroup
    {
      while (node) {
        #pragma omp task in_reduction(+: res) \
                    firstprivate(node)
        { // [3]
          res += node->value;
        }
        node = node->next;
      }
    }
  }
} // [4]
```

Tasking illustrated

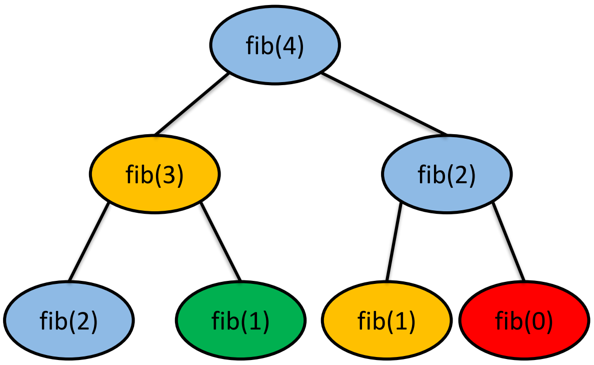
Fibonacci illustrated

```
int main(int argc,
         char* argv[])
{
    [...]
    #pragma omp parallel
    {
        #pragma omp single
        {
            fib(input);
        }
    }
    [...]
}
```

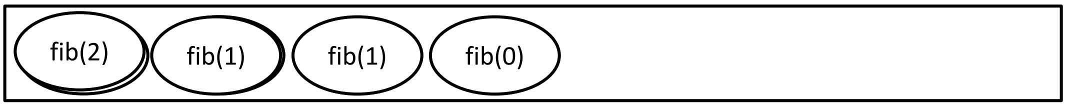
```
int fib(int n)   {
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x)
    {
        x = fib(n - 1);
    }
    #pragma omp task shared(y)
    {
        y = fib(n - 2);
    }
    #pragma omp taskwait
        return x+y;
}
```

- Only one Task / Thread enters fib() from main(), it is responsible for creating the two initial work tasks
- Taskwait is required, as otherwise x and y would get lost

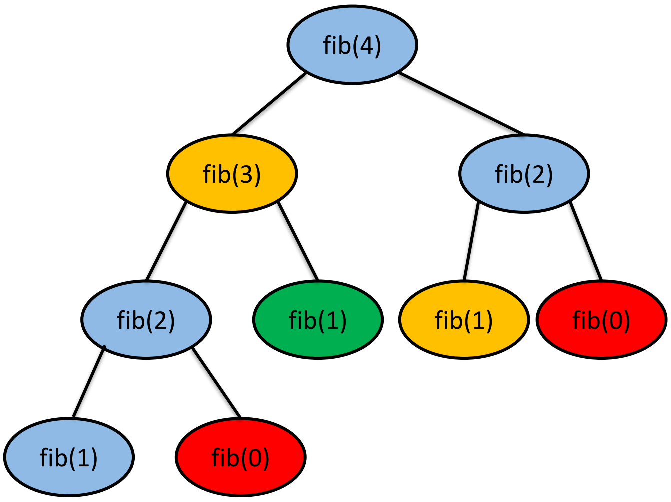
- **T1 enters fib(4)**
- **T1 creates tasks for fib(3) and fib(2)**
- **T1 and T2 execute tasks from the queue**
- **T1 and T2 create 4 new tasks**
- **T1 - T4 execute tasks**



Task Queue



- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
- …



The `taskloop` Construct

Tasking use case: saxpy (taskloop)

```
for ( i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

```
for ( i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS)?SIZE:i+TS;
   for ( ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
   }
}
```

```
#pragma omp parallel
#pragma omp single
for ( i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS)?SIZE:i+TS;
   #pragma omp task private(ii) \
    firstprivate(i,UB) shared(S,A,B)
   for ( ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
   }
}
```

- Difficult to determine grain
  - → 1 single iteration → to fine
  - → whole loop → no parallelism
- Manually transform the code
  - → blocking techniques
- Improving programmability
  - → OpenMP taskloop

```
#pragma omp taskloop grainsize(TS)
for ( i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

- → Hiding the internal details
- → Grain size ~ Tile size (TS) → but implementation decides exact grain size

The taskloop Construct

- Task generating construct: decompose a loop into chunks, create a task for each loop chunk

```
#pragma omp taskloop [clause[[,] clause]…]
{structured-for-loops}
```

```
!$omp taskloop [clause[[,] clause]…]
…structured-do-loops…
!$omp end taskloop
```

- Where clause is one of:

Data Environment
- → shared(list)
- → private(list)
- → firstprivate(list)
- → lastprivate(list)
- → default(sh | *pr* | *fp* | none)
- → reduction(r-id: list)
- → in_reduction(r-id: list)

Chunks/Grain
- → grainsize(grain-size)
- → num_tasks(num-tasks)

Cutoff Strategies
- → if(scalar-expression)
- → final(scalar-expression)
- → mergeable

Scheduler (R/H)
- → untied
- → priority(priority-value)

Miscellaneous
- → collapse(n)
- → nogroup
- → allocate([allocator:] list)

Worksharing vs. taskloop constructs (1/2)

```
subroutine worksharing
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)

!$omp do
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end do

!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 1024

```
subroutine taskloop
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)

!$omp taskloop
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end taskloop

!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 16384

Worksharing vs. taskloop constructs (2/2)

```
subroutine worksharing
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)

!$omp do
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end do

!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 1024

```
subroutine taskloop
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)
!$omp single
!$omp taskloop
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end taskloop
!$omp end single
!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 1024

Taskloop decomposition approaches

- Clause: grainsize(grain-size)
  - → Chunks have at least grain-size iterations
  - → Chunks have maximum 2x grain-size iterations

```
int TS = 4 * 1024;
#pragma omp taskloop grainsize(TS)
for ( i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

- Clause: num_tasks(num-tasks)
  - → Create num-tasks chunks
  - → Each chunk must have at least one iteration

```
int NT = 4 * omp_get_num_threads();
#pragma omp taskloop num_tasks(NT)
for ( i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

- If none of previous clauses is present, the *number of chunks* and the *number of iterations per chunk* is implementation defined
- Additional considerations:
  - → The order of the creation of the loop tasks is unspecified
  - → Taskloop creates an implicit taskgroup region; **nogroup** → no implicit taskgroup region is created

Collapsing iteration spaces with taskloop

- The collapse clause in the taskloop construct

```
#pragma omp taskloop collapse(n)
{structured-for-loops}
```

  - → Number of loops associated with the taskloop construct (n)
  - → Loops are collapsed into one larger iteration space
  - → Then divided according to the **grainsize** and **num_tasks**
- Intervening code between any two associated loops
  - → at least once per iteration of the enclosing loop
  - → at most once per iteration of the innermost loop

```
#pragma omp taskloop collapse(2)
for ( i = 0; i<SX; i+=1) {
   for (  j= 0; i<SY; j+=1) {
      for ( k = 0; i<SZ; k+=1) {
         A[f(i,j,k)]=<expression>;
      }
   }
}
```

```
#pragma omp taskloop
for ( ij = 0; i<SX*SY; ij+=1) {
   for ( k = 0; i<SZ; k+=1) {
      i = index_for_i(ij);
      j = index_for_j(ij);
      A[f(i,j,k)]=<expression>;
   }
}
```

Task reductions (using taskloop)

- Clause: `reduction(r-id: list)`
  - → It defines the scope of a new reduction
  - → All created tasks participate in the reduction
  - → It cannot be used with the **`nogroup`** clause

```
double dotprod(int n, double *x, double *y) {
  double r = 0.0;
  #pragma omp taskloop reduction(+: r)
  for (i = 0; i < n; i++)
    r += x[i] * y[i];

  return r;
}
```

- Clause: `in_reduction(r-id: list)`
  - → Reuse an already defined reduction scope
  - → All created tasks participate in the reduction
  - → It can be used with the **`nogroup`*** clause, but it is user responsibility to guarantee result

```
double dotprod(int n, double *x, double *y) {
  double r = 0.0;
  #pragma omp taskgroup task_reduction(+: r)
  {
    #pragma omp taskloop in_reduction(+: r)*
    for (i = 0; i < n; i++)
      r += x[i] * y[i];
  }
  return r;
}
```

Composite construct: taskloop simd

- Task generating construct: decompose a loop into chunks, create a task for each loop chunk
- Each generated task will apply (internally) SIMD to each loop chunk
  - → C/C++ syntax:

```
#pragma omp taskloop simd [clause[[,] clause]…]
{structured-for-loops}
```

  - → Fortran syntax:

```
!$omp taskloop simd [clause[[,] clause]…]
…structured-do-loops…
!$omp end taskloop
```

- Where clause is any of the clauses accepted by **taskloop** or **simd** directives

Improving Tasking Performance: Task dependences

Motivation

- Task dependences as a way to define task-execution constraints

OpenMP 3.1

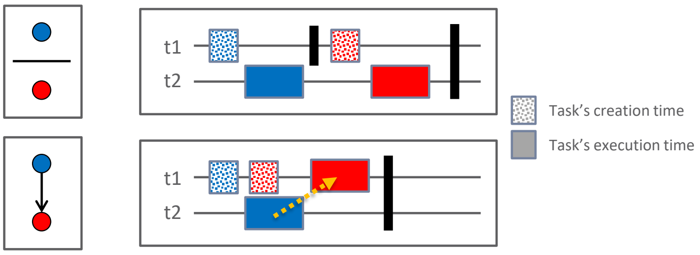
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
🔵 #pragma omp task
  std::cout << x << std::endl;

  #pragma omp taskwait

🔴 #pragma omp task
  x++;
}
```

OpenMP 4.0

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
🔵 #pragma omp task depend(in: x)
  std::cout << x << std::endl;


🔴 #pragma omp task depend(inout: x)
  x++;
}
```

OpenMP 3.1

OpenMP 4.0



Motivation

- Task dependences as a way to define task-execution constraints

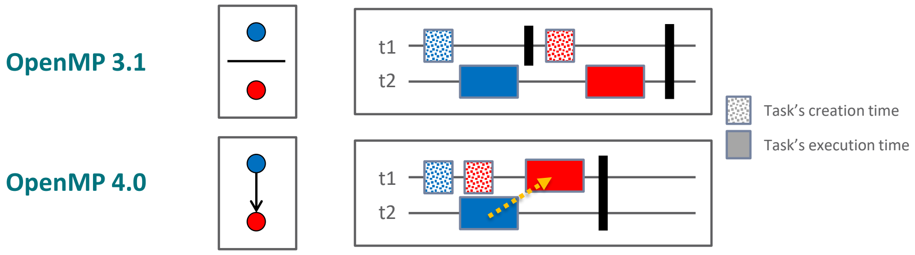
OpenMP 3.1

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
● #pragma omp task
  std::cout << x << std

  #pragma omp taskwait

● #pragma omp task
  x++;
}
```

OpenMP 4.0

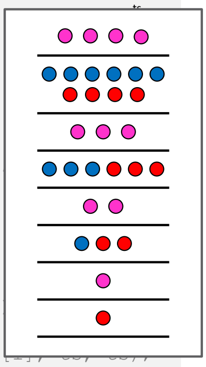
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
● #pragma omp task depend(in: x)
                              d::endl;



                              end(inout: x)
  x++;
}
```

Task dependences can help us to remove "strong" synchronizations, increasing the look ahead and, frequently, the parallelism!!!!



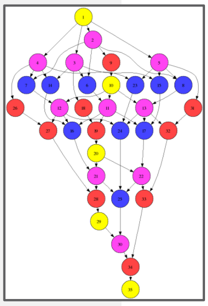
Motivation: Cholesky factorization

```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    potrf(a[k][k], ts, ts);

    // Triangular systems
    for (int i = k + 1; i < nt; i++)
      #pragma omp task
     ●trsm(a[k][k], a[k][i], ts, ts)
    }
    #pragma omp taskwait

    // Update trailing matrix
    for (int i = k + 1; i < nt; i++)
      for (int j = k + 1; j < i; j++
        #pragma omp task
       ●dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
      }
      #pragma omp task
     ●syrk(a[k][i], a[i][i], ts, ts);
    }
    #pragma omp taskwait
  }
}
```



OpenMP 3.1

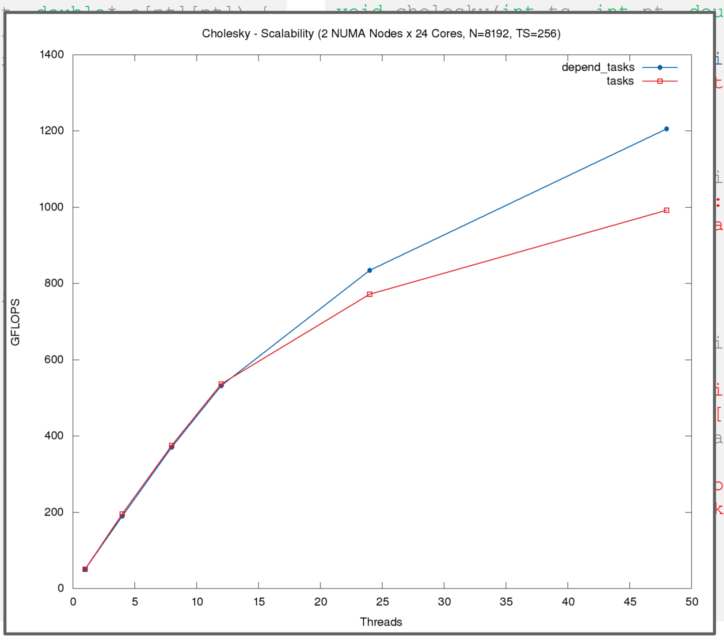
```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    #pragma omp task depend(inout: a[k][k])
   ○potrf(a[k][k], ts, ts);

    // Triangular systems
    for (int i = k + 1; i < nt; i++) {
      #pragma omp task depend(in: a[k][k])
                       depend(inout: a[k][i])
     ●trsm(a[k][k], a[k][i], ts, ts);
    }

    // Update trailing matrix
    for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++) {
        #pragma omp task depend(inout: a[j][i])
                         depend(in: a[k][i], a[k][j])
       ●dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
      }
      #pragma omp task depend(inout: a[i][i])
                       depend(in: a[k][i])
     ●syrk(a[k][i], a[i][i], ts, ts);
    }
  }
}
```



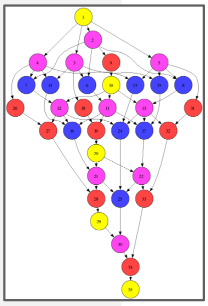
OpenMP 4.0

Motivation: Cholesky factorization

```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k
    // Diagonal Block facto
    potrf(a[k][k], ts, ts);

    // Triangular systems
    for (int i = k + 1; i <
      #pragma omp task
    ● trsm(a[k][k], a[k][i]
    }
    #pragma omp taskwait

    // Update trailing matr
    for (int i = k + 1; i <
      for (int j = k + 1; j
        #pragma omp task
      ● dgemm(a[k][i], a[k]
      }
      #pragma omp task
    ● syrk(a[k][i], a[i][i]
    }
    #pragma omp taskwait
  }
}
```



```
void cholesky(int ts, int nt, double* a[nt][nt]) {
ion
t: a[k][k])

i++) {
: a[k][k])
a[k][i])
 ts);

i++) {
 j++) {
inout: a[j][i])
[k][i], a[k][j])
a[j][i], ts, ts);

out: a[i][i])
k][i])
 ts);
```



OpenMP 4.0

Using 2017 Intel compiler

What's in the spec

What's in the spec: a bit of history

OpenMP 4.0

- The `depend` clause was added to the `task` construct

OpenMP 4.5

- The `depend` clause was added to the target constructs
- Support to doacross loops

OpenMP 5.0

- `lvalue` expressions in the depend clause
- New dependency type: `mutexinoutset`
- Iterators were added to the `depend` clause
- The `depend` clause was added to the `taskwait` construct
- Dependable objects

What's in the spec: syntax depend clause

```
depend([depend-modifier,] dependency-type: list-items)
```

where:

→ `depend-modifier` is used to define iterators

→ `dependency-type` may be: `in`, `out`, `inout`, `mutexinoutset` and `depobj`

→ A `list-item` may be:

- C/C++: A `lvalue` expr or an array section `depend(in: x, v[i], *p, w[10:10])`
- Fortran: A variable or an array section `depend(in: x, v(i), w(10:20))`

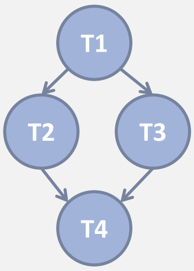
What’s in the spec: sema depend clause (1)

- A task cannot be executed until all its predecessor tasks are completed
- If a task defines an `in` dependence over a list-item
  - → the task will depend on all previously generated sibling tasks that reference that list-item in an `out` or `inout` dependence
- If a task defines an `out`/`inout` dependence over list-item
  - → the task will depend on all previously generated sibling tasks that reference that list-item in an `in`, `out` or `inout` dependence

What’s in the spec: depend clause (1)

- A task cannot be executed until all its predecessor tasks are completed
- If a task defir
  - → the task will c ne of the list items in an `out` or `in`
- If a task defir
  - → the task will c ne of the list items in an `in`, `out` (

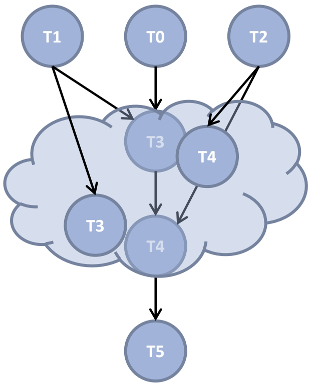
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  { ... }

  #pragma omp task depend(in: x)    //T2
  { ... }

  #pragma omp task depend(in: x)    //T3
  { ... }

  #pragma omp task depend(inout: x) //T4
  { ... }
}
```



What’s in the spec: depend clause (2)

- New dependency type: `mutexinoutset`

```
int x = 0, y = 0, res = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out: res)  //T0
   res = 0;

  #pragma omp task depend(out: x)  //T1
  long_computation(x);

  #pragma omp task depend(out: y)  //T2
  short_computation(y);

  #pragma omp task depend(in: x) depend(mutexinoutset: res) //T3
  res += x;

  #pragma omp task depend(in: y) depend(mutexinoutset: res) //T4
  res += y;

  #pragma omp task depend(in: res)  //T5
  std::cout << res << std::endl;
}
```



1. *inoutset property*: tasks with a `mutexinoutset` dependence create a cloud of tasks (an inout set) that synchronizes with previous & posterior tasks that dependent on the same list item

2. *mutex property*: Tasks inside the inout set can be executed in any order but with mutual exclusion

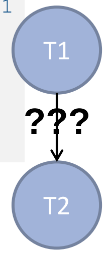
What’s in the spec: depend clause (3)

- Task dependences are defined among **sibling tasks**

- List items used in the depend clauses […] must indicate **identical** or **disjoint storage**

```
//test1.cc
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x)    //T1
  {
    #pragma omp task depend(inout: x)  //T1.1
    x++;

    #pragma omp taskwait
  }
  #pragma omp task depend(in: x)  //T2
  std::cout << x << std::endl;
}
```

```
//test2.cc
int a[100] = {0};
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: a[50:99]) //T1
  compute(/* from */ &a[50], /*elems*/ 50);

  #pragma omp task depend(in: a)    //T2
  print(/* from */ a, /* elem */ 100);
}
```

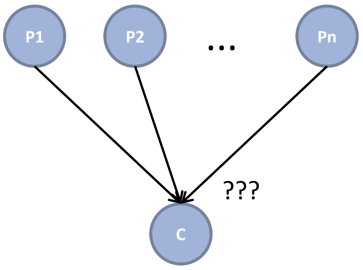


What’s in the spec: depend clause (4)

- Iterators + deps: a way to define a dynamic number of dependences

```
std::list<int> list = ...;
int n = list.size();

#pragma omp parallel
#pragma omp single
{
  for (int i = 0; i < n; ++i)
    #pragma omp task depend(out: list[i])      //Px
     compute_elem(list[i]);

  #pragma omp task depend(iterator(j=0:n), in : list[j]) //C
  print_elems(list);
}
```

It seems innocent but it’s not:
`depend(out: list.operator[](i))`



Equivalent to:
`depend(in: list[0], list[1], …, list[n-1])`

Philosophy

Philosophy: data-flow model

- Task dependences are orthogonal to data-sharings
  - → **Dependences** as a way to define **a task-execution constraints**
  - → Data-sharings as **how the data is captured** to be used inside the task

```
// test1.cc
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) \
                   firstprivate(x) //T1
  x++;

  #pragma omp task depend(in: x)  //T2
  std::cout << x << std::endl;
}
```

OK, but it always prints ‘0’ :(

```
// test2.cc
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  x++;

  #pragma omp task depend(in: x) \
                   firstprivate(x) //T2
  std::cout << x << std::endl;
}
```

We have a data-race!!

Philosophy: data-flow model (2)

- Properly combining dependences and data-sharings allow us to define a **task data-flow model**
  - →Data that is read in the task → input dependence
  - →Data that is written in the task → output dependence

- A task data-flow model
  - →Enhances the **composability**
  - →**Eases the parallelization** of new regions of your code

Philosophy: data-flow model (3)

```
//test1_v1.cc
int x = 0, y = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  {
    x++;
    y++;    // !!!
  }
  #pragma omp task depend(in: x)   //T2
  std::cout << x << std::endl;

  #pragma omp taskwait
  std::cout << y << std::endl;
}
```

```
//test1_v2.cc
```

```
//test1_v3.cc
```

```
//test1_v4.cc
int x = 0, y = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x, y) //T1
  {
    x++;
    y++;
  }
  #pragma omp task depend(in: x)       //T2
  std::cout << x << std::endl;

  #pragma omp task depend(in: y)       //T3
  std::cout << y << std::endl;
}
```

If all tasks are **properly annotated**,
we only have to worry about the
dependendences & data-sharings of the new task!!!

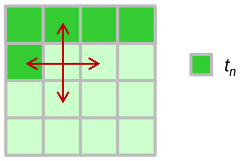
Use case

Use case: intro to Gauss-seidel

```
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
  for (int t = 0; t < tsteps; ++t) {
    for (int i = 1; i < size-1; ++i) {
      for (int j = 1; j < size-1; ++j) {
        p[i][j] = 0.25 * (p[i][j-1] * // left
                          p[i][j+1] * // right
                          p[i-1][j] * // top
                          p[i+1][j]); // bottom
      }
    }
  }
}
```

Access pattern analysis

For a specific t, i and j



Each cell depends on:

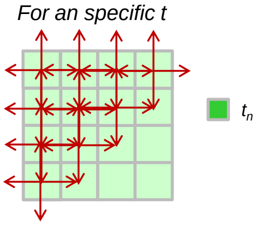
- two cells (north & west) that are computed in the current time step, and
- two cells (south & east) that were computed in the previous time step

Use case: Gauss-seidel (2)

```
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
  for (int t = 0; t < tsteps; ++t) {
    for (int i = 1; i < size-1; ++i) {
      for (int j = 1; j < size-1; ++j) {
        p[i][j] = 0.25 * (p[i][j-1] * // left
                          p[i][j+1] * // right
                          p[i-1][j] * // top
                          p[i+1][j]); // bottom
      }
    }
  }
}
```

1^{st} parallelization strategy

For an specific t



We can exploit the wavefront to obtain parallelism!!

Use case : Gauss-seidel (3)

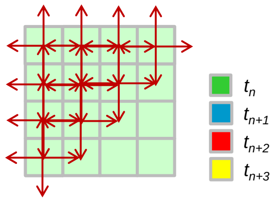
```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;
  #pragma omp parallel
  for (int t = 0; t < tsteps; ++t) {
    // First NB diagonals
    for (int diag = 0; diag < NB; ++diag) {
      #pragma omp for
      for (int d = 0; d <= diag; ++d) {
        int ii = d;
        int jj = diag - d;
         for (int i = 1+ii*TS; i < ((ii+1)*TS); ++i)
           for (int j = 1+jj*TS; i < ((jj+1)*TS); ++j)
             p[i][j] = 0.25 * (p[i][j-1] * p[i][j+1] *
                               p[i-1][j] * p[i+1][j]);
      }
    }
    // Lasts NB diagonals
    for (int diag = NB-1; diag >= 0; --diag) {
      // Similar code to the previous loop
    }
  }
}
```

Use case : Gauss-seidel (4)

```
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
  for (int t = 0; t < tsteps; ++t) {
    for (int i = 1; i < size-1; ++i) {
      for (int j = 1; j < size-1; ++j) {
        p[i][j] = 0.25 * (p[i][j-1] * // left
                          p[i][j+1] * // right
                          p[i-1][j] * // top
                          p[i+1][j]); // bottom
      }
    }
  }
}
```

2nd parallelization strategy

multiple time iterations

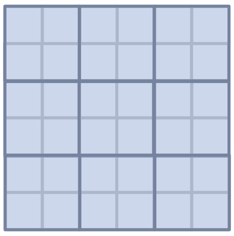


We can exploit the wavefront of multiple time steps to obtain MORE parallelism!!

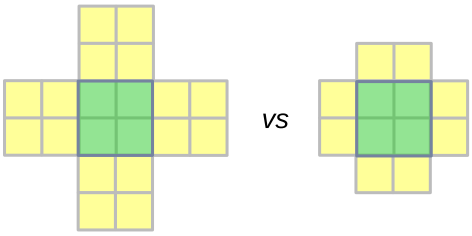
Use case : Gauss-seidel (5)

```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;

  #pragma omp parallel
  #pragma omp single
  for (int t = 0; t < tsteps; ++t)
    for (int ii=1; ii < size-1; ii+=TS)
      for (int jj=1; jj < size-1; jj+=TS) {
        #pragma omp task depend(inout: p[ii:TS][jj:TS])
            depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],
                       p[ii:TS][jj-TS:TS], p[ii:TS][jj:TS])
        {
          for (int i=ii; i<(1+ii)*TS; ++i)
            for (int j=jj; j<(1+jj)*TS; ++j)
               p[i][j] = 0.25 * (p[i][j-1] * p[i][j+1] *
                                 p[i-1][j] * p[i+1][j]);
        }
      }
}
```

inner matrix region



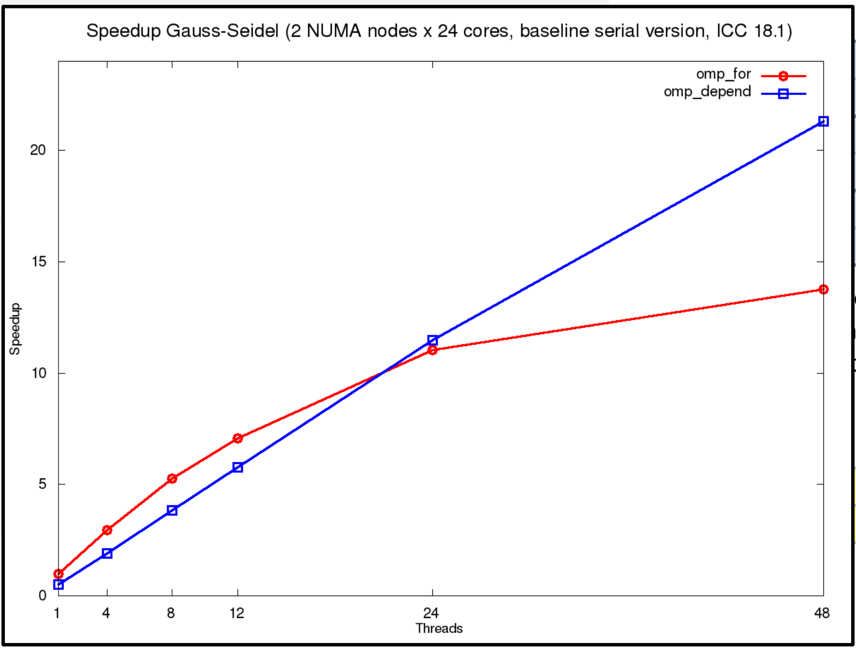
Q: Why do the input dependences depend on the whole block rather than just a column/row?

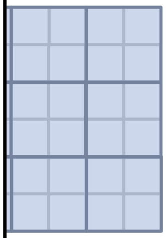


Use case : Gauss-seidel (5)

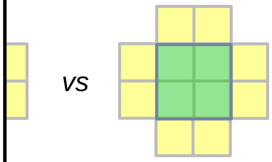
```
void gauss_seidel(i
  int NB = size / T

  #pragma omp paral
  #pragma omp singl
  for (int t = 0; t
    for (int ii=1;
      for (int jj=1
        #pragma omp
            depend(

        {
          for (int
            for (in
               p[i]

        }
      }
}
```

matrix region





e input dependences
e whole block rather
: a column/row?



Improving Tasking Performance: Cutoff clauses and strategies

OpenMP: Memory Access

Example: Loop Parallelization

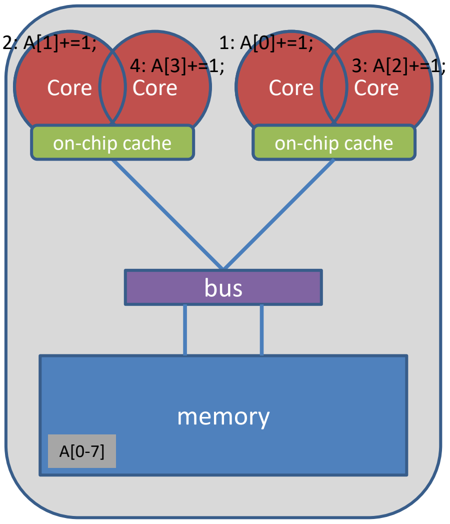
- Assume the following: you have learned that *load imbalances* can severely impact performance and a *dynamic* loop schedule may prevent this:
  - →What is the issue with the following code:

```
double* A;
A = (double*) malloc(N * sizeof(double));
/* assume some initialization of A */

#pragma omp parallel for schedule(dynamic, 1)
for (int i = 0; i < N; i++) {
   A[i] += 1.0;
}
```

  - →How is A accessed? Does that affect performance?

False Sharing

- **False Sharing: Parallel accesses to the same cache line may have a significant performance impact!**



Caches are organized in lines of typically 64 bytes: integer array a[0-4] fits into one cache line.

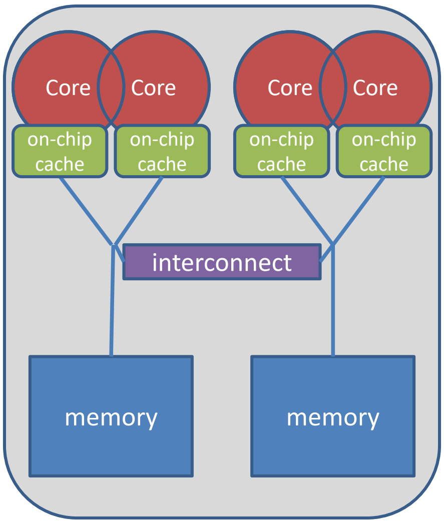
Whenever one element of a cache line is updated, the whole cache line is Invalidated.

Local copies of a cache line have to be re-loaded from the main memory and the computation may have to be repeated.

Non-uniform Memory

How To Distribute The Data ?

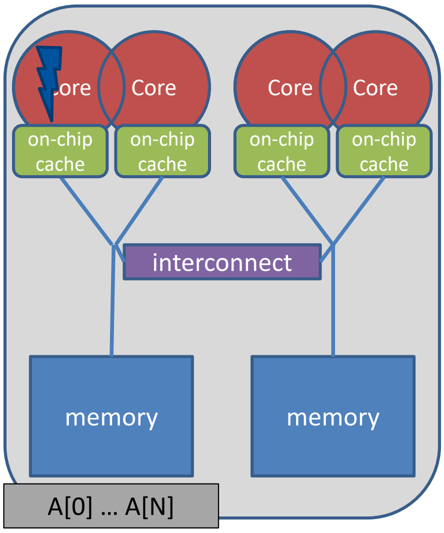
```
double* A;
A = (double*)
    malloc(N * sizeof(double));



for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



Non-uniform Memory

- Serial code: all array elements are allocated in the memory of the NUMA node closest to the core executing the initializer thread (first touch)

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



for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



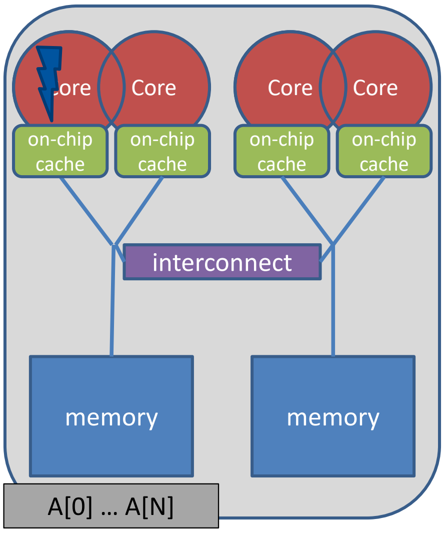
About Data Distribution

- Important aspect on cc-NUMA systems
  - →If not optimal, longer memory access times and hotspots
- Placement comes from the Operating System
  - →This is therefore Operating System dependent
- Windows, Linux and Solaris all use the “First Touch” placement policy by default
  - →May be possible to override default (check the docs)

Non-uniform Memory

- **Serial code: all array elements are allocated in the memory of the NUMA node closest to the core executing the initializer thread (first touch)**

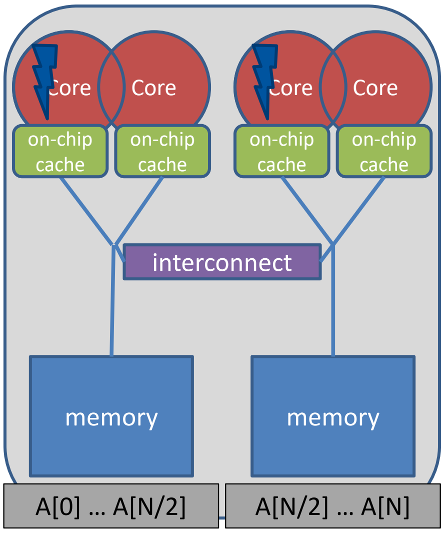
```
double* A;
A = (double*)
     malloc(N * sizeof(double));



for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



First Touch Memory Placement

- **First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node that contains the core that executes the thread that initializes the partition**

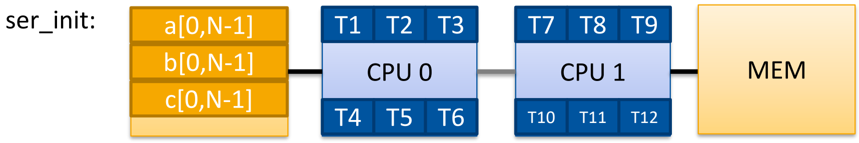
```
double* A;
A = (double*)
    malloc(N * sizeof(double));

omp_set_num_threads(2);

#pragma omp parallel for
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```

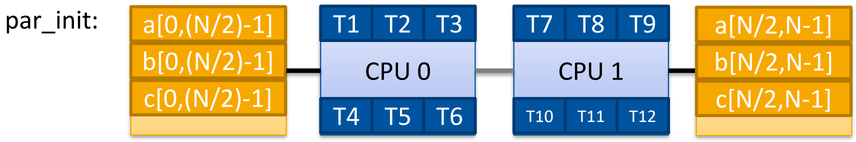


Serial vs. Parallel Initialization

- Stream example on 2 socket sytem with Xeon X5675 processors, 12 OpenMP threads:

	copy	scale	add	triad
ser_init	18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s
par_init	41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s


ser_init:
a[0,N-1]
b[0,N-1]
c[0,N-1]
T1 T2 T3
CPU 0
T4 T5 T6
T7 T8 T9
CPU 1
T10 T11 T12
MEM


par_init:
a[0,(N/2)-1]
b[0,(N/2)-1]
c[0,(N/2)-1]
T1 T2 T3
CPU 0
T4 T5 T6
T7 T8 T9
CPU 1
T10 T11 T12
a[N/2,N-1]
b[N/2,N-1]
c[N/2,N-1]

Get Info on the System Topology

- Before you design a strategy for thread binding, you should have a basic understanding of the system topology. Please use one of the following options on a target machine:
  - →Intel MPI‘s `cpuinfo` tool
    - →`cpuinfo`
    - →Delivers information about the number of sockets (= packages) and the mapping of processor ids to cpu cores that the OS uses.
  - →hwlocs‘ `hwloc-ls` tool
    - →`hwloc-ls`
    - →Displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor ids to cpu cores that the OS uses and additional info on caches.

Decide for Binding Strategy

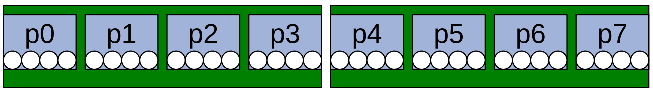
- Selecting the „right“ binding strategy depends not only on the topology, but also on application characteristics.
  - →Putting threads far apart, i.e., on different sockets
    - →May improve aggregated memory bandwidth available to application
    - →May improve the combined cache size available to your application
    - →May decrease performance of synchronization constructs
  - →Putting threads close together, i.e., on two adjacent cores that possibly share some caches
    - →May improve performance of synchronization constructs
    - →May decrease the available memory bandwidth and cache size

Places + Binding Policies (1/2)

- Define OpenMP Places
  - → set of OpenMP threads running on one or more processors
  - → can be defined by the user, i.e. `OMP_PLACES=cores`
- Define a set of OpenMP Thread Affinity Policies
  - → SPREAD: spread OpenMP threads evenly among the places, partition the place list
  - → CLOSE: pack OpenMP threads near master thread
  - → MASTER: collocate OpenMP thread with master thread
- Goals
  - → user has a way to specify where to execute OpenMP threads
  - → locality between OpenMP threads / less false sharing / memory bandwidth

Places

- Assume the following machine:



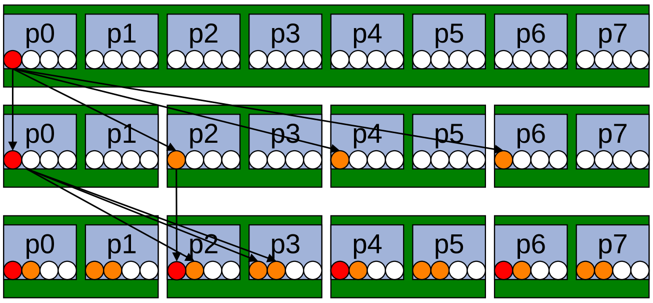
  - → 2 sockets, 4 cores per socket, 4 hyper-threads per core
- Abstract names for OMP_PLACES:
  - → threads: Each place corresponds to a single hardware thread on the target machine.
  - → cores: Each place corresponds to a single core (having one or more hardware threads) on the target machine.
  - → sockets: Each place corresponds to a single socket (consisting of one or more cores) on the target machine.
  - → ll_caches: Each place corresponds to a set of cores that share the last level cache.
  - → numa_domains: Each place corresponds to a set of cores for which their closest memory is: the same memory; and at a similar distance from the cores.

Places + Binding Policies (2/2)

- Example‘s Objective:
  - →separate cores for outer loop and near cores for inner loop
- Outer Parallel Region: proc_bind(spread) num_threads(4)
  Inner Parallel Region: proc_bind(close) num_threads(4)
  - →spread creates partition, compact binds threads within respective partition

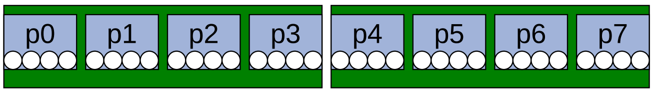
```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-3):8:4   = cores
#pragma omp parallel proc_bind(spread) num_threads(4)
#pragma omp parallel proc_bind(close) num_threads(4)
```

- Example
  - →initial
  - →spread 4
  - →close 4



More Examples (1/3)

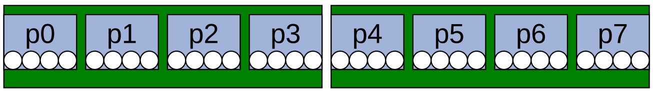
- Assume the following machine:



  - →2 sockets, 4 cores per socket, 4 hyper-threads per core
- Parallel Region with two threads, one per socket
  - →`OMP_PLACES=sockets`
  - →`#pragma omp parallel num_threads(2) proc_bind(spread)`

More Examples (2/3)

- Assume the following machine:



- Parallel Region with four threads, one per core, but only on the first socket
  - →`OMP_PLACES=cores`
  - →`#pragma omp parallel num_threads(4) proc_bind(close)`

More Examples (3/3)

- Spread a nested loop first across two sockets, then among the cores within each socket, only one thread per core
  - →`OMP_PLACES=cores`
  - →`#pragma omp parallel num_threads(2) proc_bind(spread)`
  - →`#pragma omp parallel num_threads(4) proc_bind(close)`

Places API (1/2)

- 1: Query information about binding and a single place of all places with ids 0 … `omp_get_num_places()`:
- `omp_proc_bind_t omp_get_proc_bind()`: returns the thread affinity policy (omp_proc_bind_false, true, master, …)
- `int omp_get_num_places()`: returns the number of places
- `int omp_get_place_num_procs(int place_num)`: returns the number of processors in the given place
- `void omp_get_place_proc_ids(int place_num, int* ids)`: returns the ids of the processors in the given place

Places API (2/2)

- 2: Query information about the place partition:
- `int omp_get_place_num()`: returns the place number of the place to which the current thread is bound
- `int omp_get_partition_num_places()`: returns the number of places in the current partition
- `void omp_get_partition_place_nums(int* pns)`: returns the list of place numbers corresponding to the places in the current partition

Places API: Example

- Simple routine printing the processor ids of the place the calling thread is bound to:

```
void print_binding_info() {
     int my_place = omp_get_place_num();
     int place_num_procs = omp_get_place_num_procs(my_place);

     printf("Place consists of %d processors: ", place_num_procs);

     int *place_processors = malloc(sizeof(int) * place_num_procs);
     omp_get_place_proc_ids(my_place, place_processors)

     for (int i = 0; i < place_num_procs - 1; i++) {
             printf("%d ", place_processors[i]);
     }
     printf("\n");

     free(place_processors);
}
```

OpenMP 5.0 way to do this

- Set `OMP_DISPLAY_AFFINITY=TRUE`
  - →Instructs the runtime to display formatted affinity information
  - →Example output for two threads on two physical cores:

```
nesting_level=   1,   thread_num=   0,   thread_affinity=   0,1
nesting_level=   1,   thread_num=   1,   thread_affinity=   2,3
```

  - →Output can be formatted with `OMP_AFFINITY_FORMAT` env var or corresponding routine
  - →Formatted affinity information can be printed with `omp_display_affinity(const char* format)`

Affinity format specification

t	omp_get_team_num()	a	omp_get_ancestor_thread_num() at level-1
T	omp_get_num_teams()	H	hostname
L	omp_get_level()	P	process identifier
n	omp_get_thread_num()	i	native thread identifier
N	omp_get_num_threads()	A	thread affinity: list of processors (cores)

- Example:

```
OMP_AFFINITY_FORMAT=“Affinity: %0.3L %.8n %.15{A} %.12H“
```

→Possible output:

```
Affinity: 001         0       0-1,16-17       host003
Affinity: 001         1       2-3,18-19       host003
```

A first summary

- Everything under control?
- In principle Yes, but only if
  - →threads can be bound explicitly,
  - →data can be placed well by first-touch, or can be migrated,
  - →you focus on a specific platform (= OS + arch) → no portability
- What if the data access pattern changes over time?
- What if you use more than one level of parallelism?

NUMA Strategies: Overview

- First Touch: Modern operating systems (i.e., Linux >= 2.4) decide for a physical location of a memory page during the first page fault, when the page is first „touched“, and put it close to the CPU causing the page fault.

- Explicit Migration: Selected regions of memory (pages) are moved from one NUMA node to another via explicit OS syscall.

- Next Touch: Binding of pages to NUMA nodes is removed and pages are migrated to the location of the next „touch“. Well-supported in Solaris, expensive to implement in Linux.

- Automatic Migration: No support for this in current operating systems.

User Control of Memory Affinity

- Explicit NUMA-aware memory allocation:
  - →By carefully touching data by the thread which later uses it
  - →By changing the default memory allocation strategy
    - →Linux: `numactl` command
    - →Windows: `VirtualAllocExNuma()` (limited functionality)
  - →By explicit migration of memory pages
    - →Linux: `move_pages()`
    - →Windows: no option

- Example: using numactl to distribute pages round-robin:
  - → `numactl –interleave=all ./a.out`

Improving Tasking Performance:
Task Affinity

Motivation

- Techniques for process binding & thread pinning available
  - →OpenMP thread level: `OMP_PLACES & OMP_PROC_BIND`
  - →OS functionality: `taskset -c`

OpenMP Tasking:

- In general: Tasks may be executed by any thread in the team
  - →Missing task-to-data affinity may have detrimental effect on performance

OpenMP 5.0:

- `affinity` clause to express affinity to data

`affinity` clause

- New clause: `#pragma omp task affinity (list)`
  - →Hint to the runtime to execute task closely to physical data location
  - →Clear separation between dependencies and affinity
- Expectations:
  - →Improve data locality / reduce remote memory accesses
  - →Decrease runtime variability
- Still expect task stealing
  - →In particular, if a thread is under-utilized

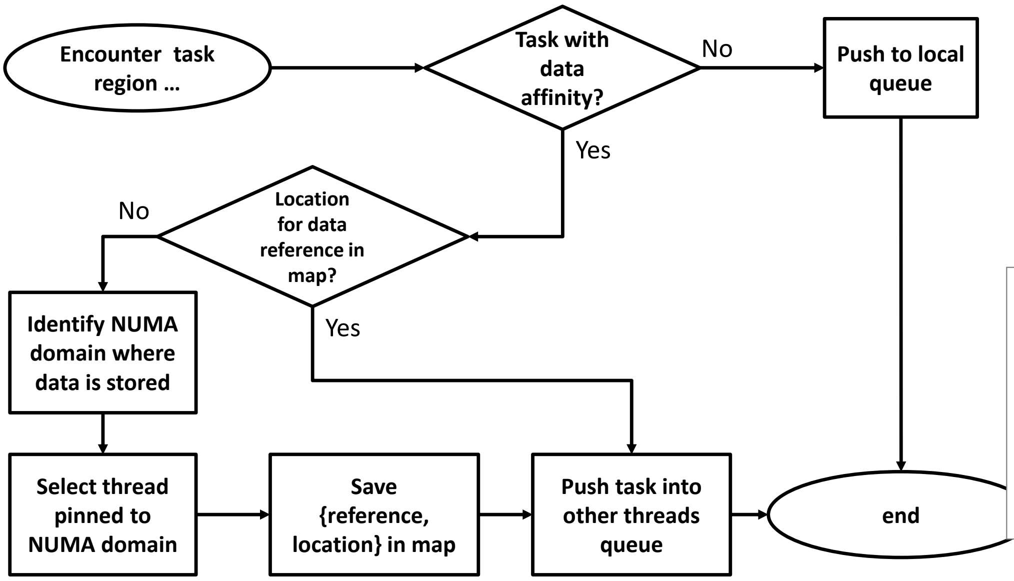
Code Example

- Excerpt from task-parallel STREAM

```
#pragma omp task \
    shared(a, b, c, scalar) \
    firstprivate(tmp_idx_start, tmp_idx_end) \
    affinity( a[tmp_idx_start] )
{
   int i;
   for(i = tmp_idx_start; i <= tmp_idx_end; i++)
        a[i] = b[i] + scalar * c[i];
}
```

  - →Loops have been blocked manually (see `tmp_idx_start/end`)
  - →Assumption: initialization and computation have same blocking and same affinity

Selected LLVM implementation details



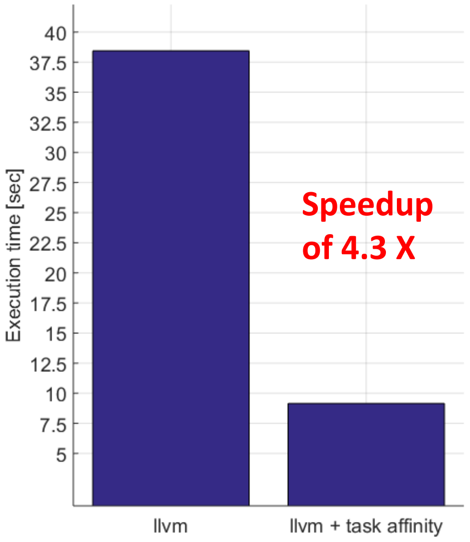
Encounter task region …
Task with data affinity?
No
Push to local queue
Yes
No
Location for data reference in map?
Yes
Identify NUMA domain where data is stored
Select thread pinned to NUMA domain
Save {reference, location} in map
Push task into other threads queue
end

A map is introduced to store location information of data that was previously used

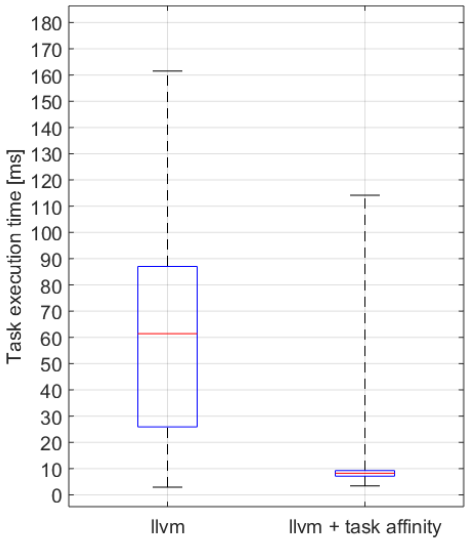
Jannis Klinkenberg, Philipp Samfass, Christian Terboven, Alejandro Duran, Michael Klemm, Xavier Teruel, Sergi Mateo, Stephen L. Olivier, and Matthias S. Müller. **Assessing Task-to-Data Affinity in the LLVM OpenMP Runtime**. Proceedings of the 14th International Workshop on OpenMP, IWOMP 2018. September 26-28, 2018, Barcelona, Spain.

Evaluation

Program runtime
Median of 10 runs



Distribution of single task execution times



LIKWID: reduction of remote data volume from 69% to 13%

Summary

- Requirement for this feature: thread affinity enabled

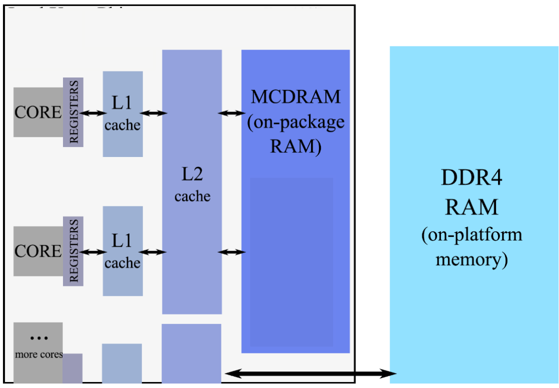
- The `affinity` clause helps, if
  - →tasks access data heavily
  - →single task creator scenario, or task not created with data affinity
  - →high load imbalance among the tasks

- Different from thread binding: task stealing is absolutely allowed

Managing Memory Spaces

Different kinds of memory

- Traditional DDR-based memory
- High-bandwidth memory
- Non-volatile memory
- …



Memory Management

- Allocator := an OpenMP object that fulfills requests to allocate and deallocate storage for program variables
- OpenMP allocators are of type `omp_allocator_handle_t`
- Default allocator for Host
  - →via `OMP_ALLOCATOR` env. var. or corresponding API
- OpenMP 5.0 supports a set of memory allocators

OpenMP Allocators

- Selection of a certain kind of memory

Allocator name	Storage selection intent
omp_default_mem_alloc	use default storage
omp_large_cap_mem_alloc	use storage with large capacity
omp_const_mem_alloc	use storage optimized for read-only variables
omp_high_bw_mem_alloc	use storage with high bandwidth
omp_low_lat_mem_alloc	use storage with low latency
omp_cgroup_mem_alloc	use storage close to all threads in the contention group of the thread requesting the allocation
omp_pteam_mem_alloc	use storage that is close to all threads in the same parallel region of the thread requesting the allocation
omp_thread_local_mem_alloc	use storage that is close to the thread requesting the allocation

Using OpenMP Allocators

- New clause on all constructs with data sharing clauses:
  - → `allocate( [allocator:] list )`
- Allocation:
  - → `omp_alloc(size_t size, omp_allocator_handle_t allocator)`
- Deallocation:
  - → `omp_free(void *ptr, const omp_allocator_handle_t allocator)`
  - → `allocator` argument is optional
- `allocate` directive: standalone directive for allocation, or declaration of allocation stmt.

OpenMP Allocator Traits / 1

- Allocator traits control the behavior of the allocator

sync_hint	contended, uncontended, serialized, private default: contended
alignment	positive integer value that is a power of two default: 1 byte
access	all, cgroup, pteam, thread default: all
pool_size	positive integer value
fallback	default_mem_fb, null_fb, abort_fb, allocator_fb default: default_mem_fb
fb_data	an allocator handle
pinned	true, false default: false
partition	environment, nearest, blocked, interleaved default: environment

OpenMP Allocator Traits / 2

- `fallback`: describes the behavior if the allocation cannot be fulfilled
  - →`default_mem_fb`: return system’s default memory
  - →Other options: null, abort, or use different allocator

- `pinned`: request pinned memory, i.e. for GPUs

OpenMP Allocator Traits / 3

- `partition`: partitioning of allocated memory of physical storage resources (think of NUMA)
  - →`environment`: use system’s default behavior
  - →`nearest`: most closest memory
  - →`blocked`: partitioning into approx. same size with at most one block per storage resource
  - →`interleaved`: partitioning in a round-robin fashion across the storage resources

OpenMP Allocator Traits / 4

- Construction of allocators with traits via
  - →
    ```
    omp_allocator_handle_t   omp_init_allocator(
    omp_memspace_handle_t memspace,
    int ntraits, const omp_alloctrait_t traits[]);
    ```
  - →Selection of memory space mandatory
  - →Empty traits set: use defaults
- Allocators have to be destroyed with `*_destroy_*`
- Custom allocator can be made default with `omp_set_default_allocator(omp_allocator_handle_t allocator)`

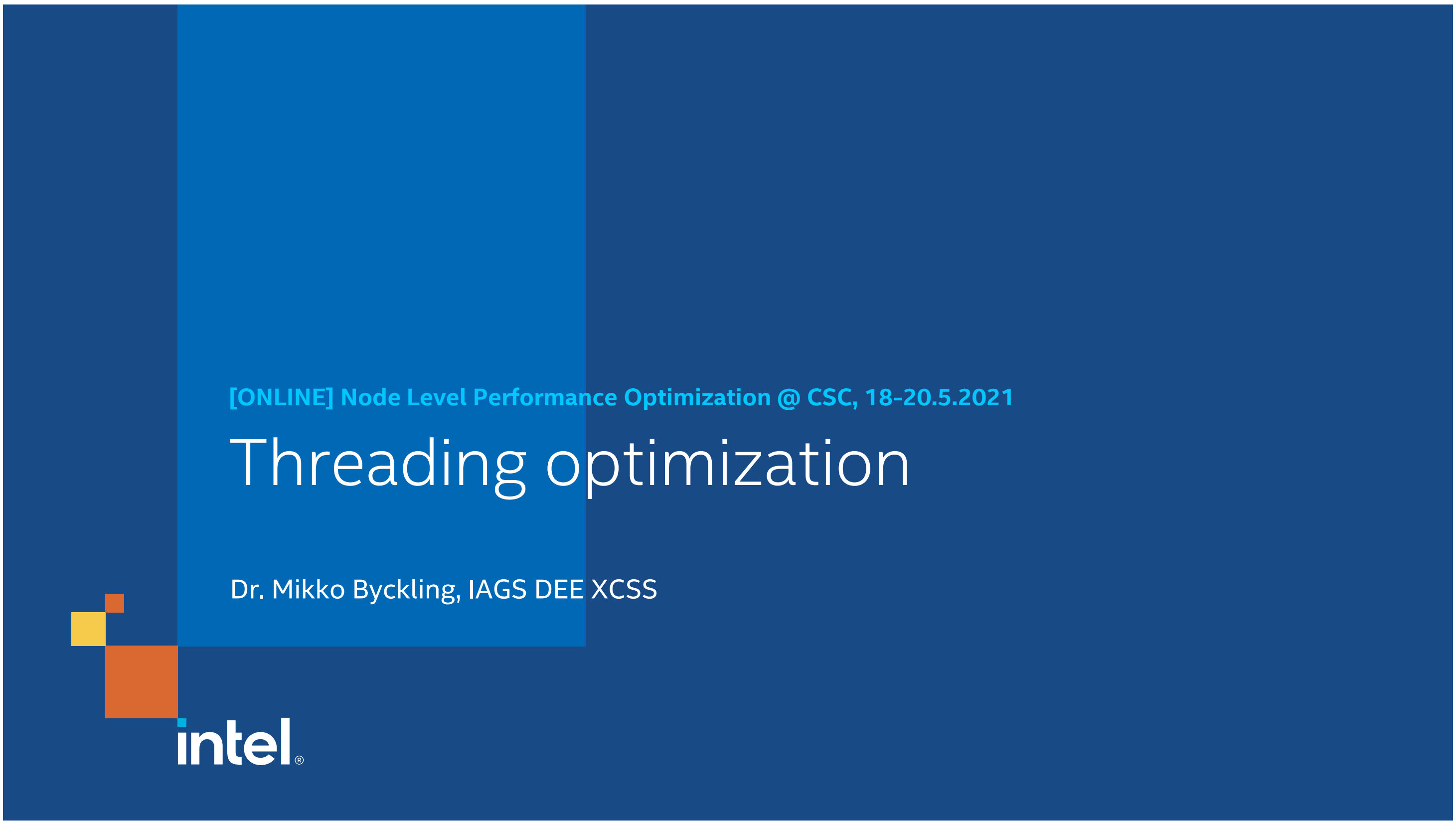
OpenMP Memory Spaces

- Storage resources with explicit support in OpenMP:

omp_default_mem_space	System's default memory resource
omp_large_cap_mem_space	Storage with larg(er) capacity
omp_const_mem_space	Storage optimized for variables with constant value
omp_high_bw_mem_space	Storage with high bandwidth
omp_low_lat_mem_space	Storage with low latency

→Exact selection of memory space is implementation-def.

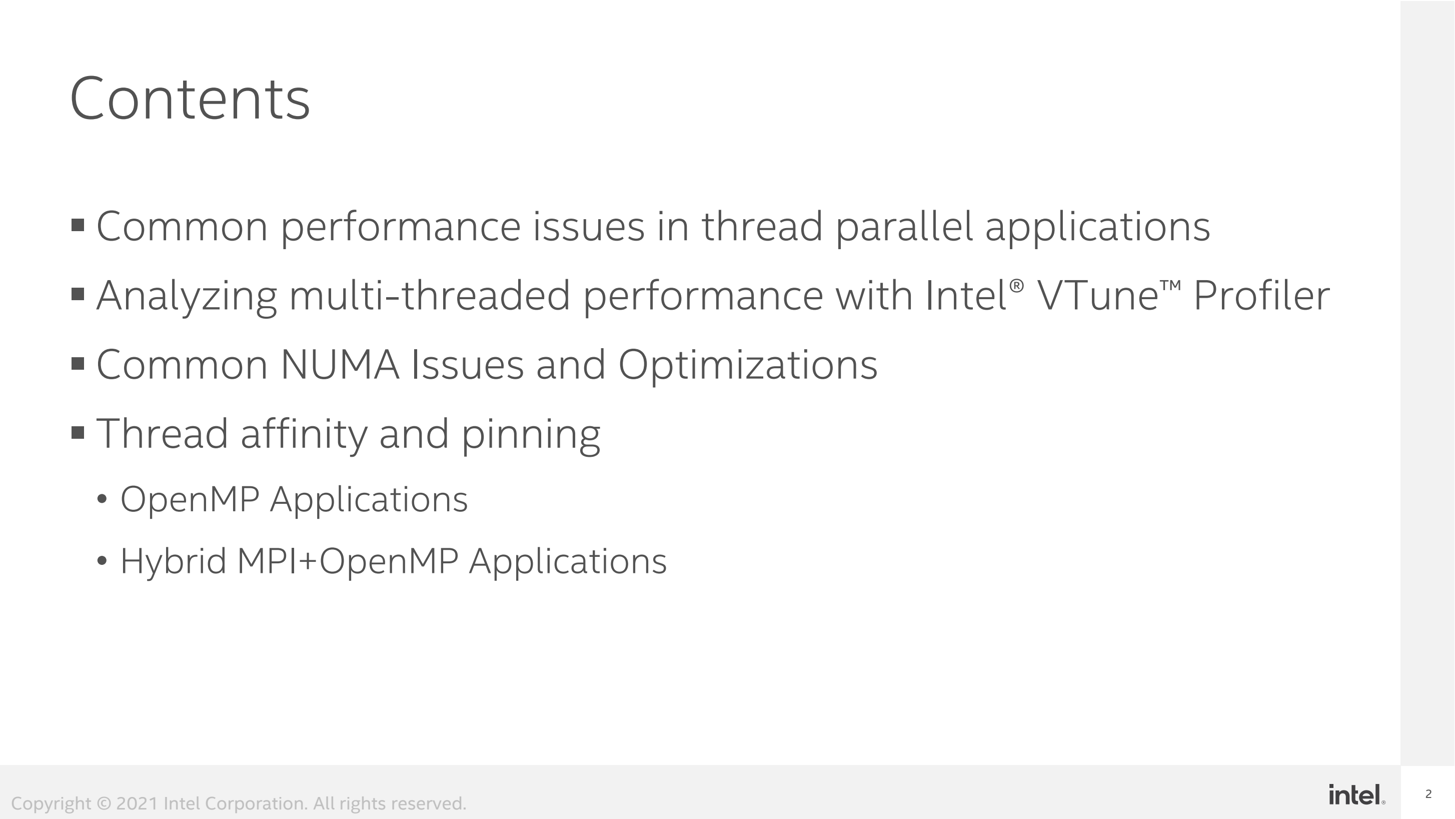
→Pre-defined allocators available to work with these



[ONLINE] Node Level Performance Optimization @ CSC, 18-20.5.2021

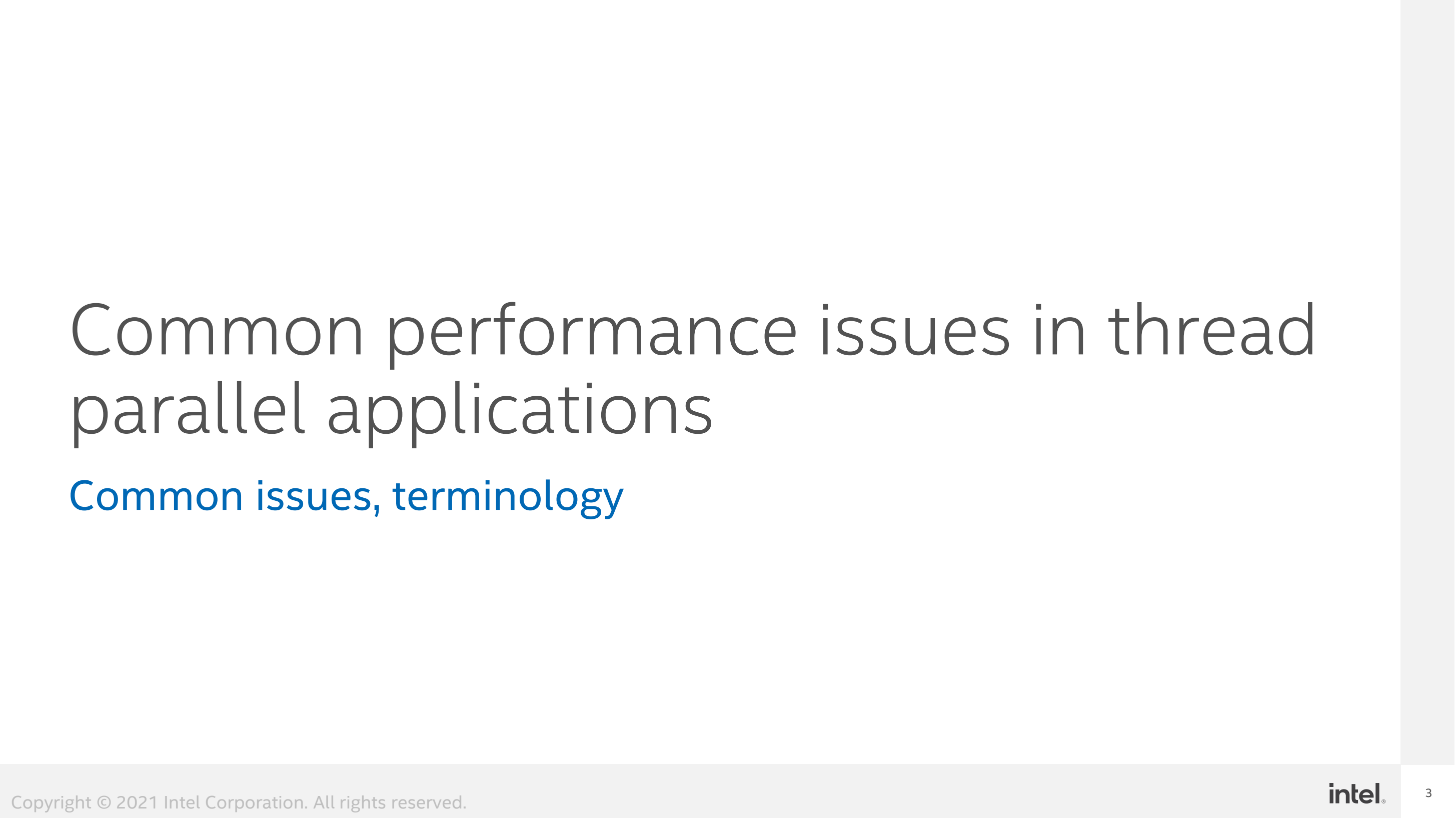
Threading optimization

Dr. Mikko Byckling, IAGS DEE XCSS



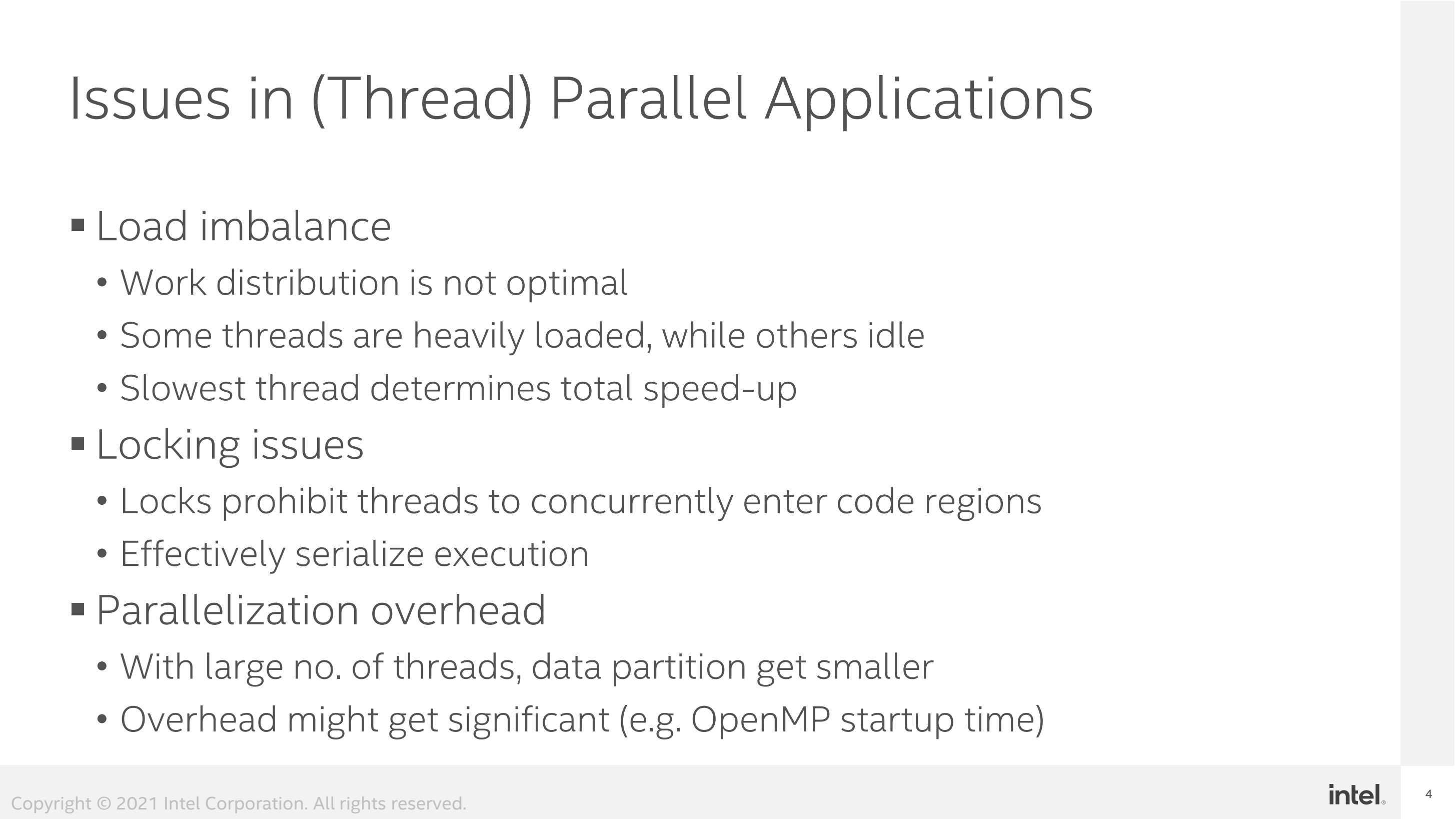
Contents

- Common performance issues in thread parallel applications
- Analyzing multi-threaded performance with Intel® VTune™ Profiler
- Common NUMA Issues and Optimizations
- Thread affinity and pinning
  - OpenMP Applications
  - Hybrid MPI+OpenMP Applications



Common performance issues in thread parallel applications

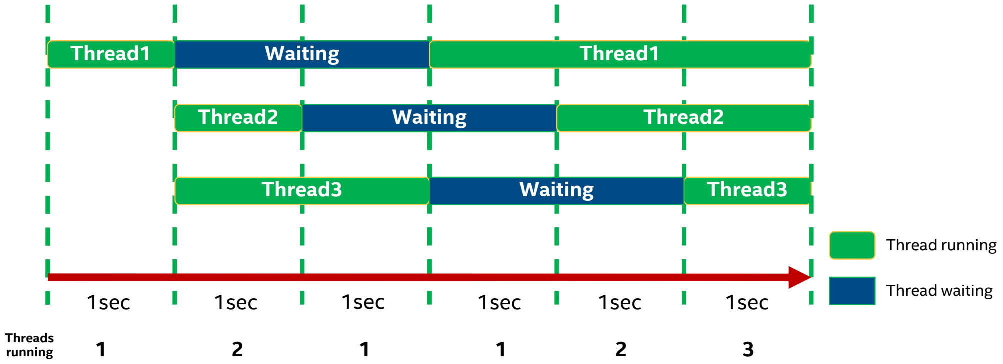
Common issues, terminology



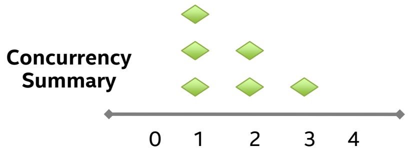
Issues in (Thread) Parallel Applications

- Load imbalance
  - Work distribution is not optimal
  - Some threads are heavily loaded, while others idle
  - Slowest thread determines total speed-up
- Locking issues
  - Locks prohibit threads to concurrently enter code regions
  - Effectively serialize execution
- Parallelization overhead
  - With large no. of threads, data partition get smaller
  - Overhead might get significant (e.g. OpenMP startup time)

Threading Analysis Terminology



- **Elapsed Time**: 6 seconds
- **CPU Time**: T1 (4s) + T2 (3s) + T3 (3s) = 10 seconds
- **Wait Time**: T1(2s) + T2(2s) + T3 (2s) = 6 seconds



Analyzing multi-threaded performance with Intel® VTune™ Profiler

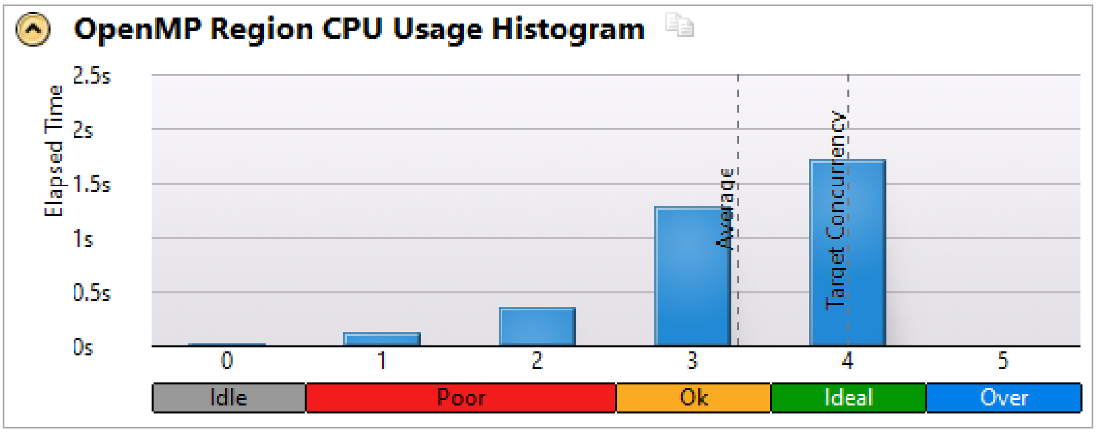
Overview, treading analysis, thread timeline, MPI+OpenMP analysis

VTune GUI: OpenMP analysis

- **Tracing** of OpenMP constructs to provide region/work sharing context and imbalance on barriers
  - Advanced hotspots w/o stacks is recommended to make sampling representative for small regions
- VTune is provided with information by Intel OpenMP RTL
  - Fork-Join points of parallel regions with number of working threads (Intel Compilers version 14 and later)
  - OpenMP construct barrier points with imbalance info and OpenMP loop metadata
    - Embed source file name to an OpenMP region with **`-parallel-source-info=2`** compiler option

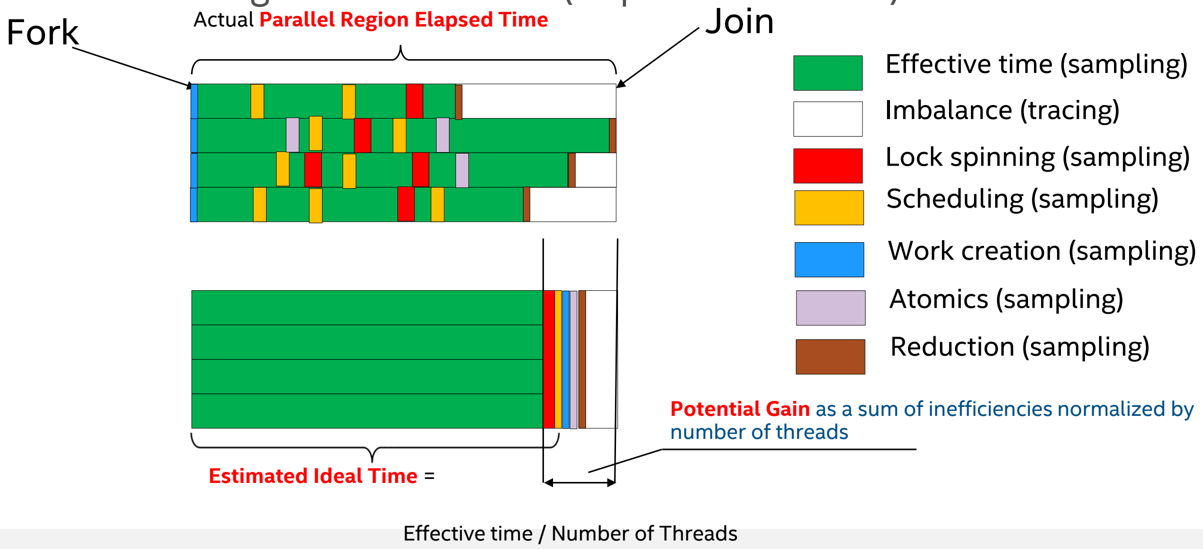
VTune GUI: Thread Concurrency Histogram

Global view of OpenMP concurrency



VTune GUI: OpenMP region view

Definition of Region Potential Gain (elapsed time metric)



Fork

Actual **Parallel Region Elapsed Time**

Join

🟩 Effective time (sampling)

□ Imbalance (tracing)

🟥 Lock spinning (sampling)

🟨 Scheduling (sampling)

🟦 Work creation (sampling)

🟪 Atomics (sampling)

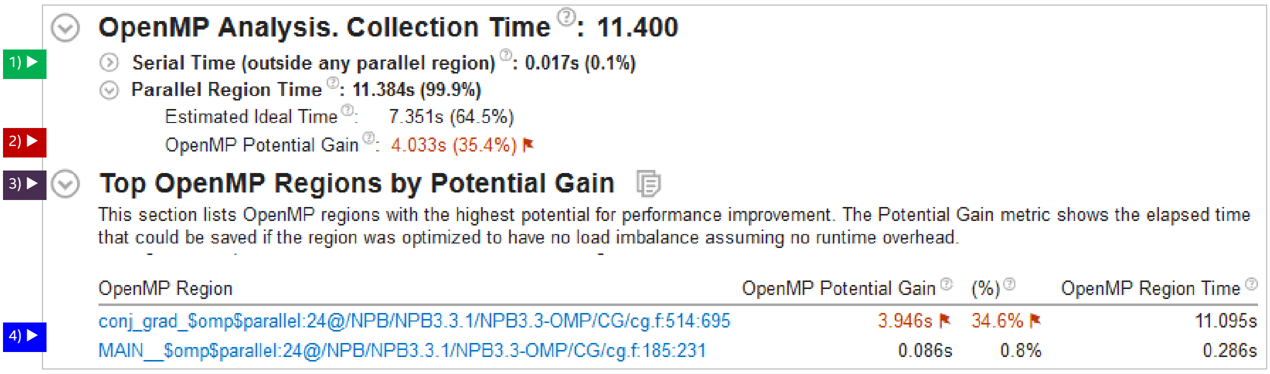
🟫 Reduction (sampling)

Potential Gain as a sum of inefficiencies normalized by number of threads

Estimated Ideal Time =

Effective time / Number of Threads

VTune GUI: Threading Analysis (1/5)



OpenMP Analysis. Collection Time: 11.400

1) ▶ **Serial Time (outside any parallel region): 0.017s (0.1%)**

Parallel Region Time: 11.384s (99.9%)

Estimated Ideal Time: 7.351s (64.5%)

2) ▶ OpenMP Potential Gain: 4.033s (35.4%)

3) ▶ **Top OpenMP Regions by Potential Gain**

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	OpenMP Potential Gain	(%)	OpenMP Region Time
4) ▶ conj_grad_ompparallel:24@/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:514:695	3.946s	34.6%	11.095s
MAIN__ompparallel:24@/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:185:231	0.086s	0.8%	0.286s

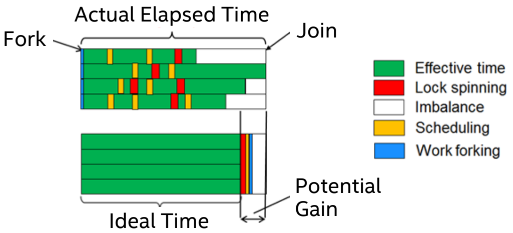
Summary view:

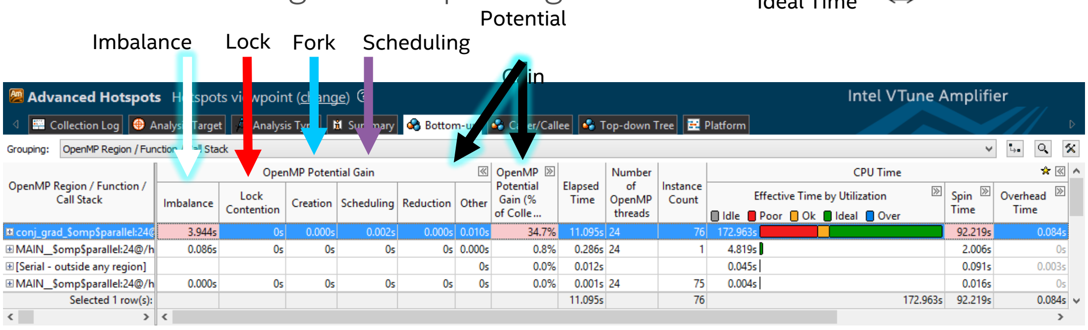
1) Is the **serial** time of my application significant enough to prevent scaling?
2) How much performance can be gained by tuning OpenMP?
3) Which OpenMP regions / loops / barriers will benefit most from tuning?
4) What are the inefficiencies with each region? (click the link to see details)

VTune GUI: Threading Analysis (2/5)

Focus On What's Important

- What region is inefficient?
- Is the potential gain worth it?
- Why is it inefficient?
  Imbalance? Scheduling? Lock spinning?



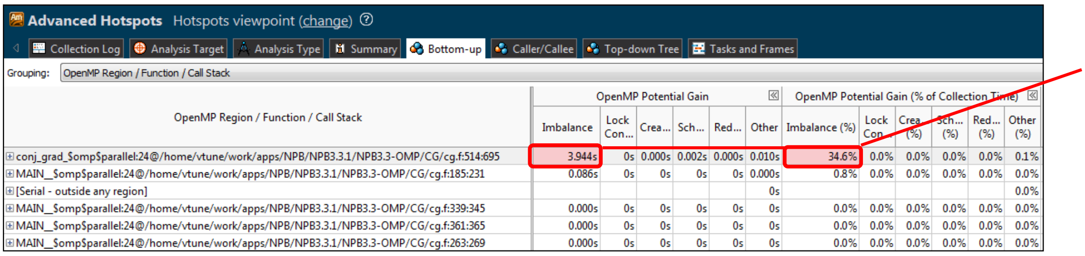


Grouping: OpenMP Region / Function / Call Stack

OpenMP Region / Function / Call Stack	OpenMP Potential Gain						OpenMP Potential Gain (% of Colle…	Elapsed Time	Number of OpenMP threads	Instance Count	CPU Time		
	Imbalance	Lock Contention	Creation	Scheduling	Reduction	Other					Effective Time by Utilization Idle Poor Ok Ideal Over	Spin Time	Overhead Time
conj_grad_ompparallel:24@	3.944s	0s	0.000s	0.002s	0.000s	0.010s	34.7%	11.095s	24	76	172.963s	92.219s	0.084s
MAIN__ompparallel:24@/h	0.086s	0s	0s	0s	0s	0.000s	0.8%	0.286s	24	1	4.819s	2.006s	0s
[Serial - outside any region]						0s	0.0%	0.012s			0.045s	0.091s	0.003s
MAIN__ompparallel:24@/h	0.000s	0s	0s	0s	0s	0s	0.0%	0.001s	24	75	0.004s	0.016s	0s
Selected 1 row(s):								11.095s		76	172.963s	92.219s	0.084s

VTune GUI: Threading Analysis (3/5)

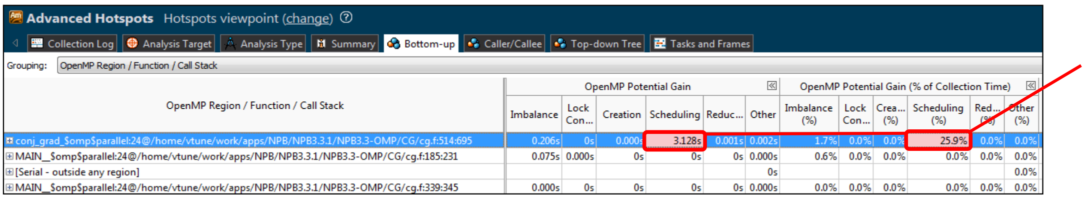
Parallel Region Inefficiencies



Grouping: OpenMP Region / Function / Call Stack

OpenMP Region / Function / Call Stack	OpenMP Potential Gain						OpenMP Potential Gain (% of Collection Time)					
	Imbalance	Lock Con…	Crea…	Sch…	Red…	Other	Imbalance (%)	Lock Con…	Crea… (%)	Sch… (%)	Red… (%)	Other (%)
conj_grad_ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:514:695	3.944s	0s	0.000s	0.002s	0.000s	0.010s	34.6%	0.0%	0.0%	0.0%	0.0%	0.1%
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:185:231	0.086s	0s	0s	0s	0s	0.000s	0.8%	0.0%	0.0%	0.0%	0.0%	0.0%
[Serial - outside any region]						0s						0.0%
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:339:345	0.000s	0s	0s	0s	0s	0s	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:361:365	0.000s	0s	0s	0s	0s	0s	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:263:269	0.000s	0s	0s	0s	0s	0s	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Imbalance



Grouping: OpenMP Region / Function / Call Stack

OpenMP Region / Function / Call Stack	OpenMP Potential Gain						OpenMP Potential Gain (% of Collection Time)					
	Imbalance	Lock Con…	Creation	Scheduling	Reduc…	Other	Imbalance (%)	Lock Con…	Crea… (%)	Scheduling (%)	Red… (%)	Other (%)
conj_grad_ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:514:695	0.206s	0s	0.000s	3.128s	0.001s	0.002s	1.7%	0.0%	0.0%	25.9%	0.0%	0.0%
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:185:231	0.075s	0.000s	0s	0s	0s	0.000s	0.6%	0.0%	0.0%	0.0%	0.0%	0.0%
[Serial - outside any region]						0s						0.0%
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:339:345	0.000s	0s	0s	0s	0s	0.000s	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

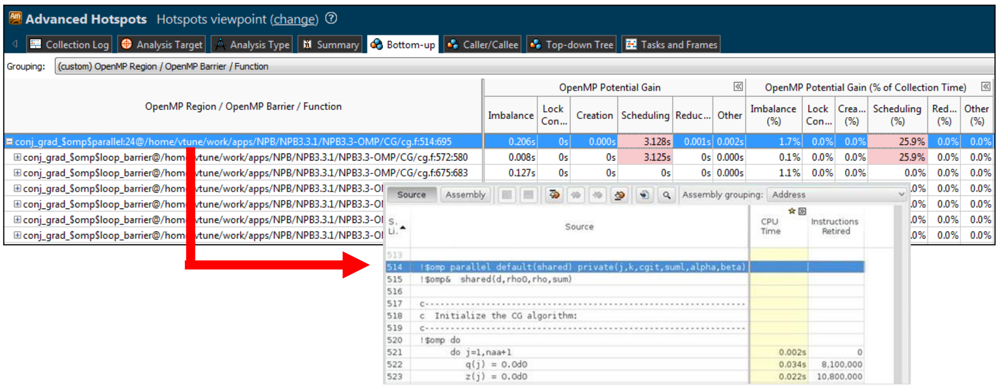
Likely culprit:

Dynamic scheduling overhead

VTune GUI: Threading Analysis (4/5)

Mapping regions to source code

- View data specific to the region at the source code level
- With ‘**-parallel-source-info=2**’ compiler option to embed source file name in region name



Grouping: (custom) OpenMP Region / OpenMP Barrier / Function

OpenMP Region / OpenMP Barrier / Function	OpenMP Potential Gain						OpenMP Potential Gain (% of Collection Time)					
	Imbalance	Lock Con...	Creation	Scheduling	Reduc...	Other	Imbalance (%)	Lock Con...	Crea... (%)	Scheduling (%)	Red... (%)	Other (%)
⊟ conj_grad_ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:514:695	0.206s	0s	0.000s	3.128s	0.001s	0.002s	1.7%	0.0%	0.0%	25.9%	0.0%	0.0%
⊞ conj_grad_omploop_barrier@/hom vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:572:580	0.008s	0s	0s	3.125s	0s	0.000s	0.1%	0.0%	0.0%	25.9%	0.0%	0.0%
⊞ conj_grad_omploop_barrier@/hom vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:675:683	0.127s	0s	0s	0s	0s	0.000s	1.1%	0.0%	0.0%	0.0%	0.0%	0.0%
⊞ conj_grad_omploop_barrier@/hom vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OI										.0%	0.0%	0.0%
⊞ conj_grad_omploop_barrier@/hom vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OI										.0%	0.0%	0.0%
⊞ conj_grad_omploop_barrier@/hom vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OI										.0%	0.0%	0.0%
⊞ conj_grad_omploop_barrier@/hom vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OI										.0%	0.0%	0.0%

S. Li.	Source	CPU Time	Instructions Retired
513			
514	!$omp parallel default(shared) private(j,k,cgit,suml,alpha,beta)		
515	!$omp& shared(d,rho0,rho,sum)		
516			
517	c---		
518	c Initialize the CG algorithm:		
519	c---		
520	!$omp do		
521	do j=1,naa+1	0.002s	0
522	q(j) = 0.0d0	0.034s	8,100,000
523	z(j) = 0.0d0	0.022s	10,800,000

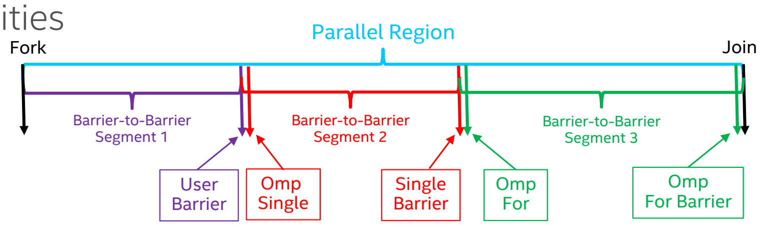
VTune GUI: Threading Analysis (5/5)

Understanding parallel inefficiency

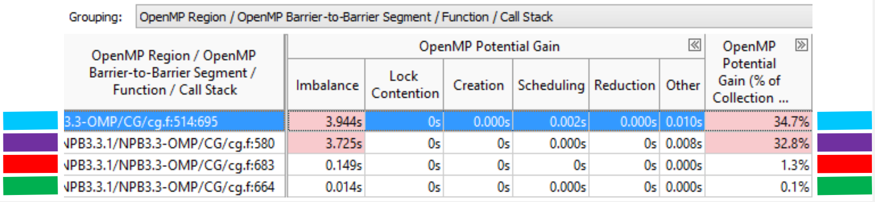
Detailed Barrier to Barrier Analysis

- Tune each segment separately
- Easier to see tuning opportunities

```
#pragma omp parallel
{
      …
#pragma omp barrier
#pragma omp single
        {
              …
        {
#pragma omp for
        {
```



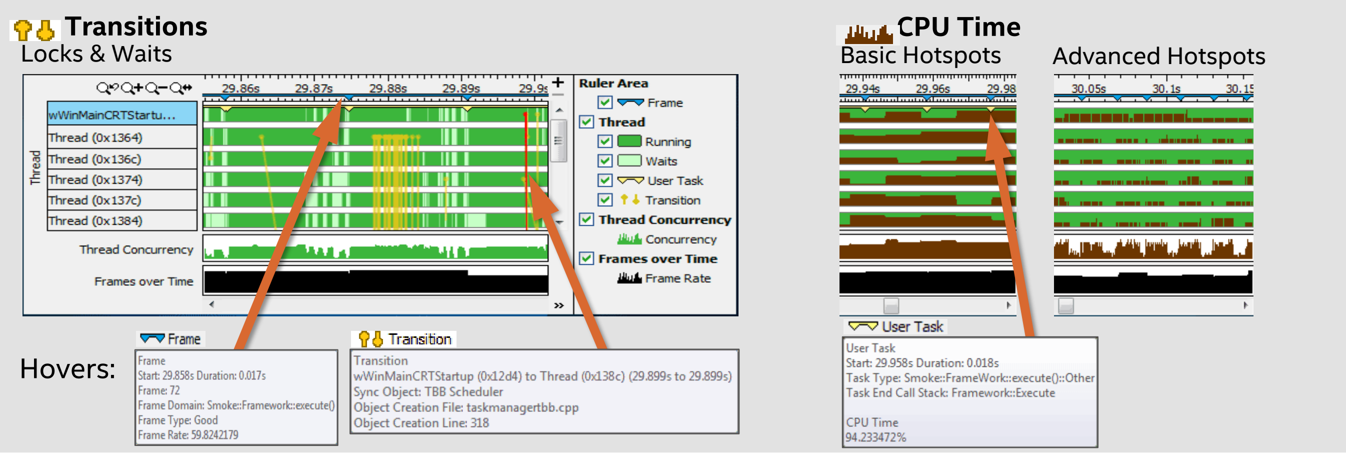
Parallel Region
Fork
Join
Barrier-to-Barrier Segment 1
Barrier-to-Barrier Segment 2
Barrier-to-Barrier Segment 3
User Barrier
Omp Single
Single Barrier
Omp For
Omp For Barrier



Grouping: OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack

OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack	OpenMP Potential Gain						OpenMP Potential Gain (% of Collection ...
	Imbalance	Lock Contention	Creation	Scheduling	Reduction	Other	
3.3-OMP/CG/cg.f:514:695	3.944s	0s	0.000s	0.002s	0.000s	0.010s	34.7%
NPB3.3.1/NPB3.3-OMP/CG/cg.f:580	3.725s	0s	0s	0.000s	0s	0.008s	32.8%
NPB3.3.1/NPB3.3-OMP/CG/cg.f:683	0.149s	0s	0s	0s	0s	0.000s	1.3%
NPB3.3.1/NPB3.3-OMP/CG/cg.f:664	0.014s	0s	0s	0.000s	0s	0.000s	0.1%

VTune GUI: Thread timeline



Frame
Start: 29.858s Duration: 0.017s
Frame: 72
Frame Domain: Smoke::Framework::execute()
Frame Type: Good
Frame Rate: 59.8242179

Transition
wWinMainCRTStartup (0x12d4) to Thread (0x138c) (29.899s to 29.899s)
Sync Object: TBB Scheduler
Object Creation File: taskmanagertbb.cpp
Object Creation Line: 318

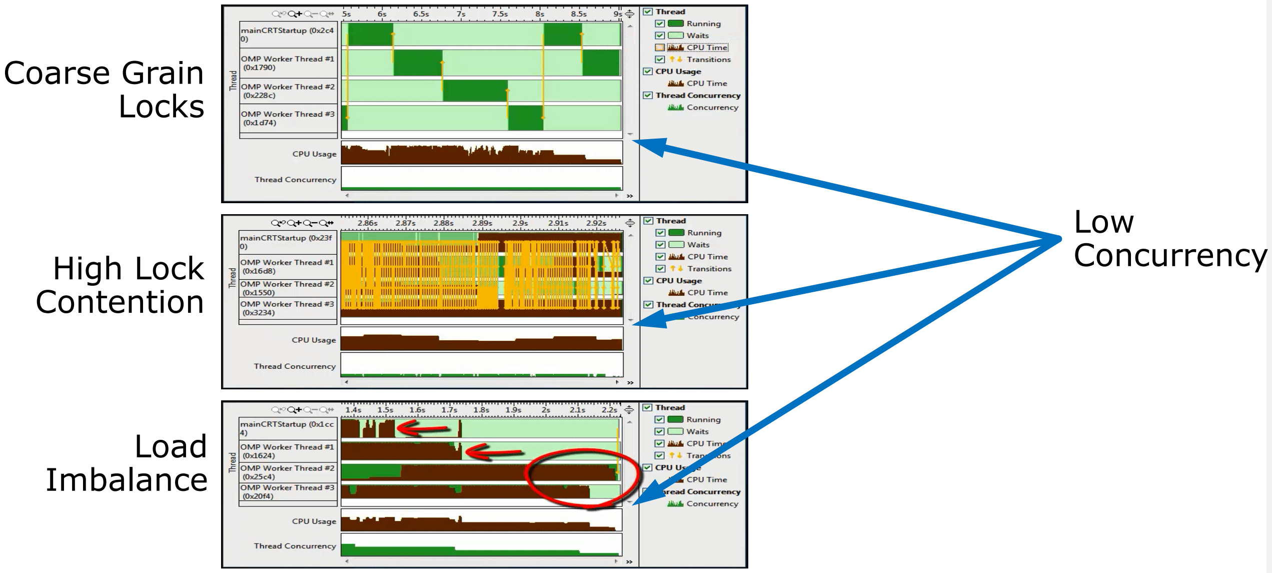
User Task
Start: 29.958s Duration: 0.018s
Task Type: Smoke::FrameWork::execute()::Other
Task End Call Stack: Framework::Execute

CPU Time
94.233472%

- Optional: Use API to mark frames and user tasks Frame User Task
- Optional: Add a mark during collection

VTune GUI: Threading analysis

Common patterns for root causing low concurrency



Coarse Grain Locks

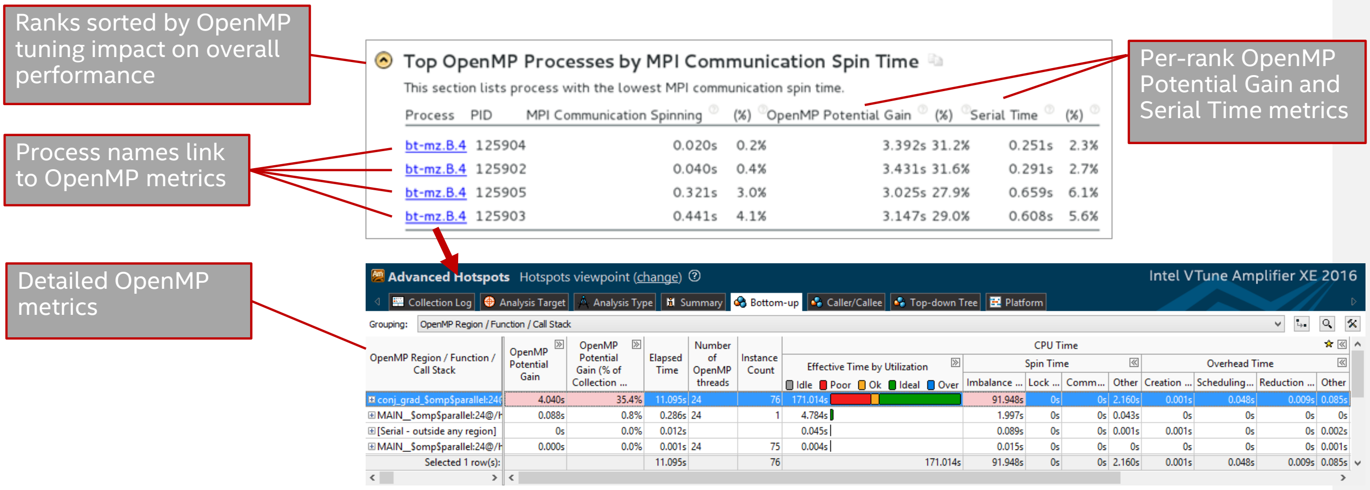
High Lock Contention

Load Imbalance

Low Concurrency

VTune GUI: MPI + OpenMP analysis

Tune OpenMP performance of high impact ranks in VTune Profiler



Ranks sorted by OpenMP tuning impact on overall performance

Process names link to OpenMP metrics

Detailed OpenMP metrics

Per-rank OpenMP Potential Gain and Serial Time metrics

Top OpenMP Processes by MPI Communication Spin Time

This section lists process with the lowest MPI communication spin time.

Process	PID	MPI Communication Spinning	(%)	OpenMP Potential Gain	(%)	Serial Time	(%)
bt-mz.B.4	125904	0.020s	0.2%	3.392s	31.2%	0.251s	2.3%
bt-mz.B.4	125902	0.040s	0.4%	3.431s	31.6%	0.291s	2.7%
bt-mz.B.4	125905	0.321s	3.0%	3.025s	27.9%	0.659s	6.1%
bt-mz.B.4	125903	0.441s	4.1%	3.147s	29.0%	0.608s	5.6%

Grouping: OpenMP Region / Function / Call Stack

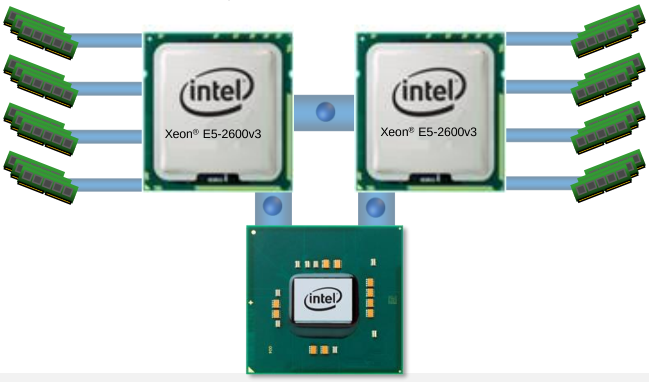
OpenMP Region / Function / Call Stack	OpenMP Potential Gain	OpenMP Potential Gain (% of Collection ...	Elapsed Time	Number of OpenMP threads	Instance Count	CPU Time								
						Effective Time by Utilization	Spin Time				Overhead Time			
						Idle Poor Ok Ideal Over	Imbalance ...	Lock ...	Comm...	Other	Creation ...	Scheduling...	Reduction ...	Other
⊞ conj_grad_ompparallel:24@	4.040s	35.4%	11.095s	24	76	171.014s	91.948s	0s	0s	2.160s	0.001s	0.048s	0.009s	0.085s
⊞ MAIN__ompparallel:24@/h	0.088s	0.8%	0.286s	24	1	4.784s	1.997s	0s	0s	0.043s	0s	0s	0s	0s
⊞ [Serial - outside any region]	0s	0.0%	0.012s			0.045s	0.089s	0s	0s	0.001s	0.001s	0s	0s	0.002s
⊞ MAIN__ompparallel:24@/h	0.000s	0.0%	0.001s	24	75	0.004s	0.015s	0s	0s	0s	0s	0s	0s	0.001s
Selected 1 row(s):			11.095s		76	171.014s	91.948s	0s	0s	2.160s	0.001s	0.048s	0.009s	0.085s

Common NUMA Issues and Optimizations

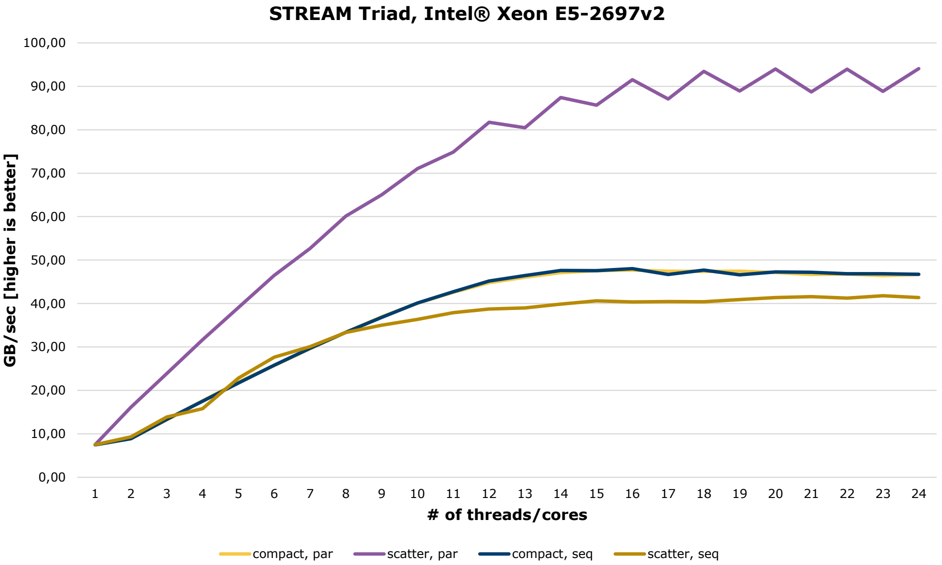
First touch policy, common optimizations

(Almost) all HPC systems are NUMA

- (Almost) all multi-socket compute servers are NUMA systems
  - Different access latencies for different memory locations
  - Different bandwidth observed for different memory locations
- Example: Intel® Xeon E5-2600v3 Series processor



NUMA - Does it matter?

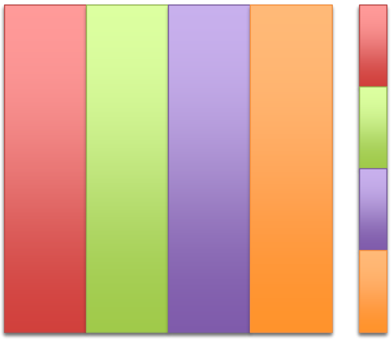


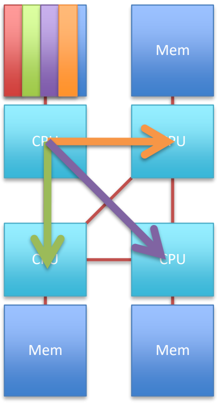
First touch policy

- Modern operating systems all use virtual memory
- The OS typically optimizes memory allocations
  - `malloc()` does not allocate the memory directly
  - Only the memory management “knows” about the memory allocation, but no memory pages are made available
  - At first memory access (*write*), the OS physically allocates the corresponding page (First touch policy)
- On NUMA systems this might lead to performance issues in threaded or multi-process applications

NUMA Optimization with OpenMP

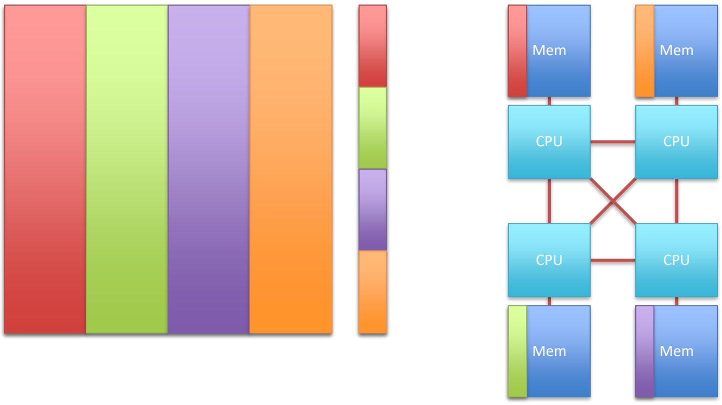
```
// Initialize data

for (size_t i = 0; i < N; i++)
    for (size_t j = 0; j < M; j++)  {…}
// Perform work
#pragma omp parallel for private(j)
for (size_t i = 0; i < N; i++)
    for (size_t j = 0; j < M; j++)  {…}
```





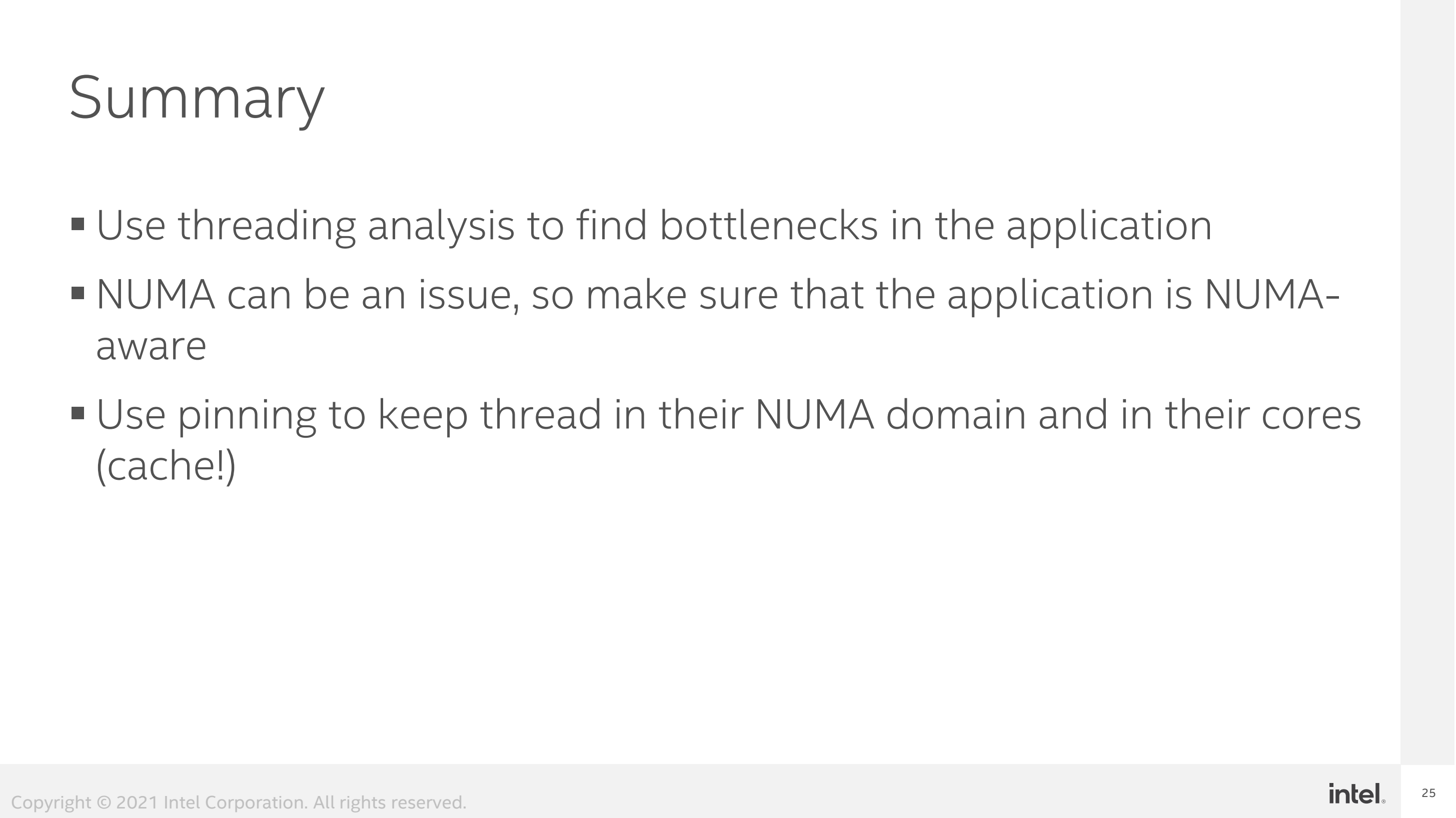
NUMA Optimization with OpenMP

```
// Initialize data
#pragma omp parallel for private(j)
for (size_t i = 0; i < N; i++)
    for (size_t j = 0; j < M; j++) {…}
// Perform work
#pragma omp parallel for private(j)
for (size_t i = 0; i < N; i++)
    for (size_t j = 0; j < M; j++) {…}
```



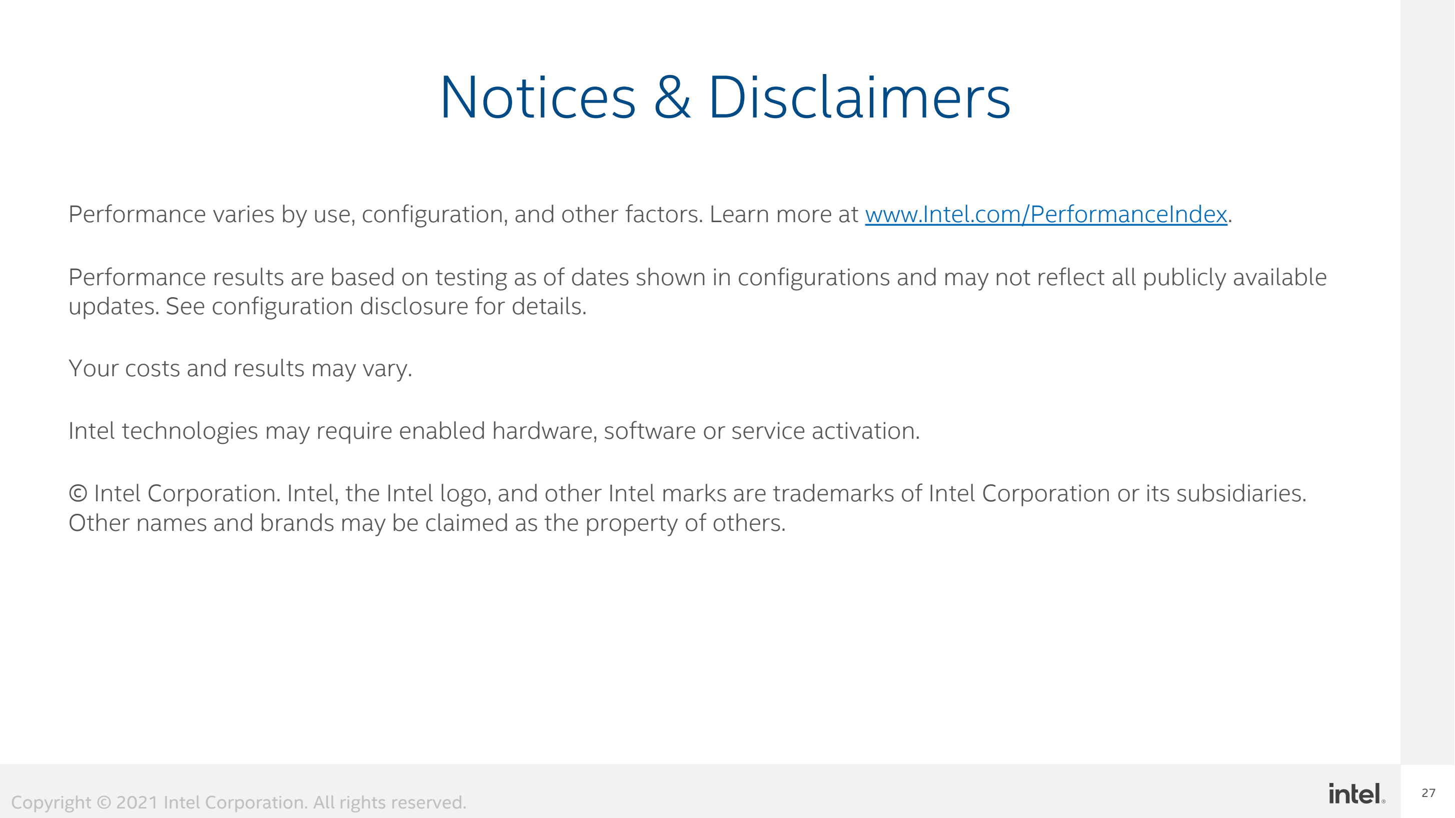
NUMA issues and MPI Applications

- MPI applications might also be affected by NUMA issues:
  - A process allocates memory on one NUMA node…
  - … and is then scheduled to run on another NUMA node.
- Intra-node communication might show different bandwidths and/or latencies to network fabric adapter
- The file system cache
  - Might reserve memory on one NUMA node..
  - ..and thus push out allocations to a remote NUMA node.



Summary

- Use threading analysis to find bottlenecks in the application
- NUMA can be an issue, so make sure that the application is NUMA-aware
- Use pinning to keep thread in their NUMA domain and in their cores (cache!)



Notices & Disclaimers

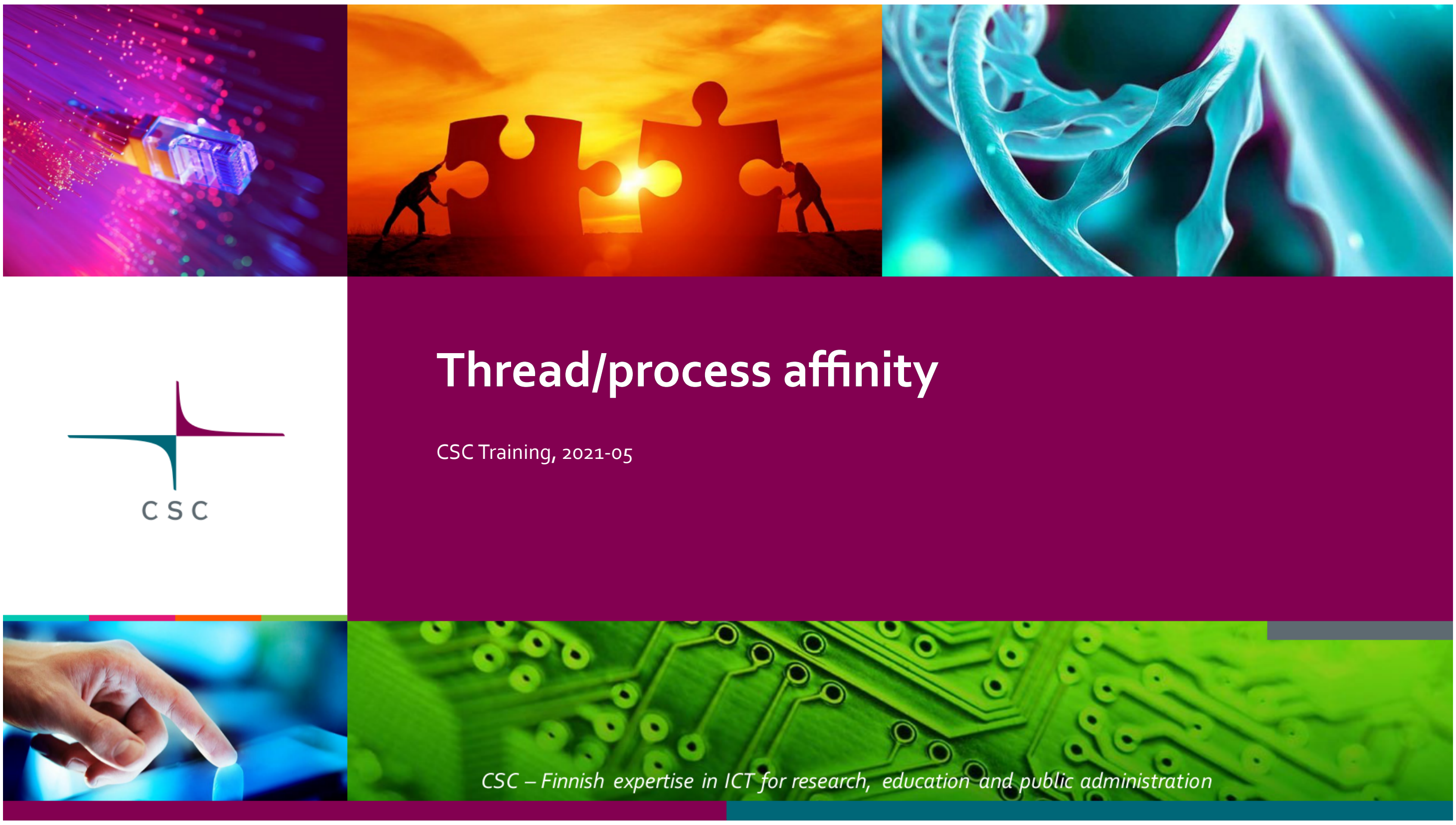
Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.



Thread/process affinity

CSC Training, 2021-05

CSC – Finnish expertise in ICT for research, education and public administration

Thread and process affinity

- Normally, operating system can run threads and processes in any logical core
- Operating system may even move running task from one core to another
  - Can be beneficial for load balancing
  - For HPC workloads often detrimental as private caches get invalidated and NUMA locality is lost
- User can control where tasks are run via affinity masks
  - Task can be *pinned* to a specific logical core or set of logical cores

Controlling affinity

- Affinity for a *process* can be set with a `numactl` command
  - Limit the process to logical cores 0,3,7:

    `numactl --physcpubind=0,3,7 ./my_exe`
  - Threads "inherit" the affinity of their parent process
- Affinity of a thread can be set with OpenMP environment variables
  - `OMP_PLACES=[threads,cores,sockets]`
  - `OMP_PROC_BIND=[true, close, spread, master]`
- OpenMP runtime prints the affinity with `OMP_DISPLAY_AFFINITY=true`

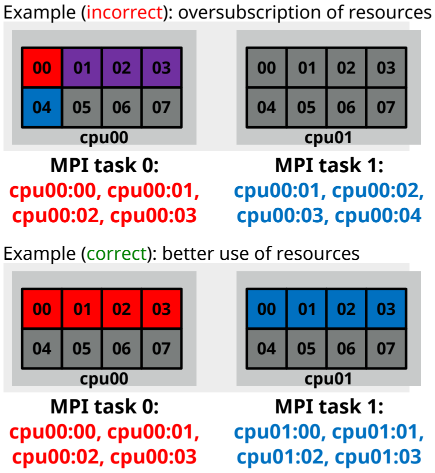
Controlling affinity

```
export OMP_AFFINITY_FORMAT="Thread %0.3n affinity %A"
export OMP_DISPLAY_AFFINITY=true
./test
Thread 000 affinity 0-7
Thread 001 affinity 0-7
Thread 002 affinity 0-7
Thread 003 affinity 0-7
```

```
OMP_PLACES=cores ./test
Thread 000 affinity 0,4
Thread 001 affinity 1,5
Thread 002 affinity 2,6
Thread 003 affinity 3,7
```

MPI+OpenMP thread affinity

- MPI library must be aware of the underlying OpenMP for correct allocation of resources
  - Oversubscription of CPU cores may cause significant performance penalty
- Additional complexity from batch job schedulers
- Heavily dependent on the platform used!


Example (incorrect): oversubscription of resources

MPI task 0:
cpu00:00, cpu00:01, cpu00:02, cpu00:03

MPI task 1:
cpu00:01, cpu00:02, cpu00:03, cpu00:04

Example (correct): better use of resources

MPI task 0:
cpu00:00, cpu00:01, cpu00:02, cpu00:03

MPI task 1:
cpu01:00, cpu01:01, cpu01:02, cpu01:03

Slurm configuration at CSC

- Within a node, `--tasks-per-node` MPI tasks are spread `--cpus-per-task` apart
- Threads within a MPI tasks have the affinity mask for the corresponging `--cpus-per-task` cores

```
export OMP_AFFINITY_FORMAT="Process %P thread %0.3n affinity %A"
export OMP_DISPLAY_AFFINITY=true
srun ... --tasks-per-node=2 --cpus-per-task=4 ./test
Process 250545 thread 000 affinity 0-3
...
Process 250546 thread 000 affinity 4-7
...
```

- Slurm configurations in other HPC centers can be very different
  - Always experiment before production calculations!

Summary

- Performance of HPC applications is often improved when processes and threads are pinned to CPU cores
- MPI and batch system configurations may affect the affinity
    - very system dependent, try to always investigate