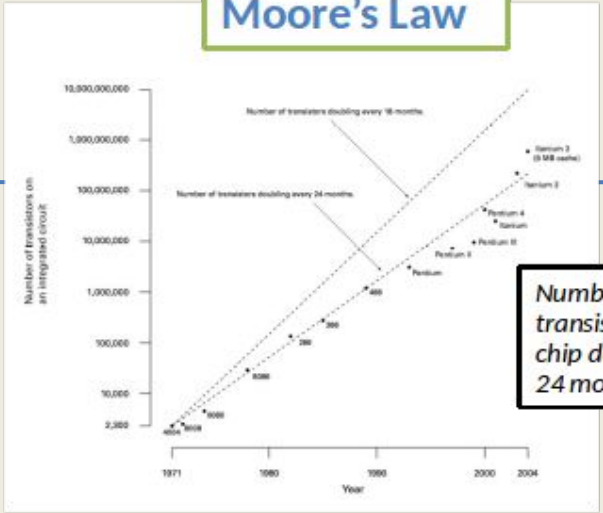
Introduction to Intel scalable architectures

Fabio Affinito (SCAI - Cineca)

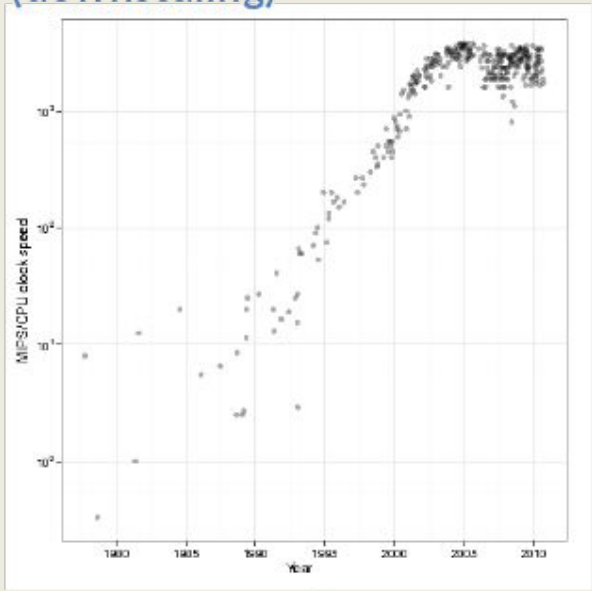
HPC Trends

Moore's Law



Number of transistors per chip double every 24 month

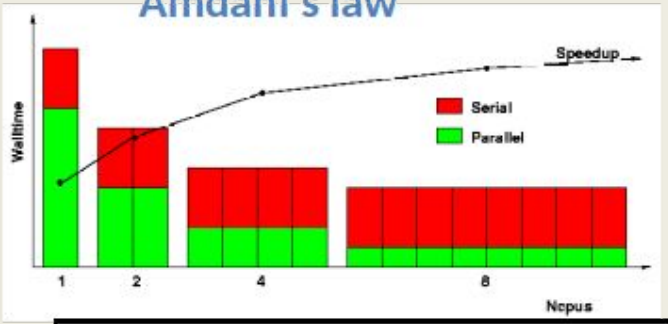
Dennard scaling law (downscaling)



The core frequency and performance do not grow following the Moore's law any longer

Increase the number of cores to maintain the architectures evolution on the Moore's law

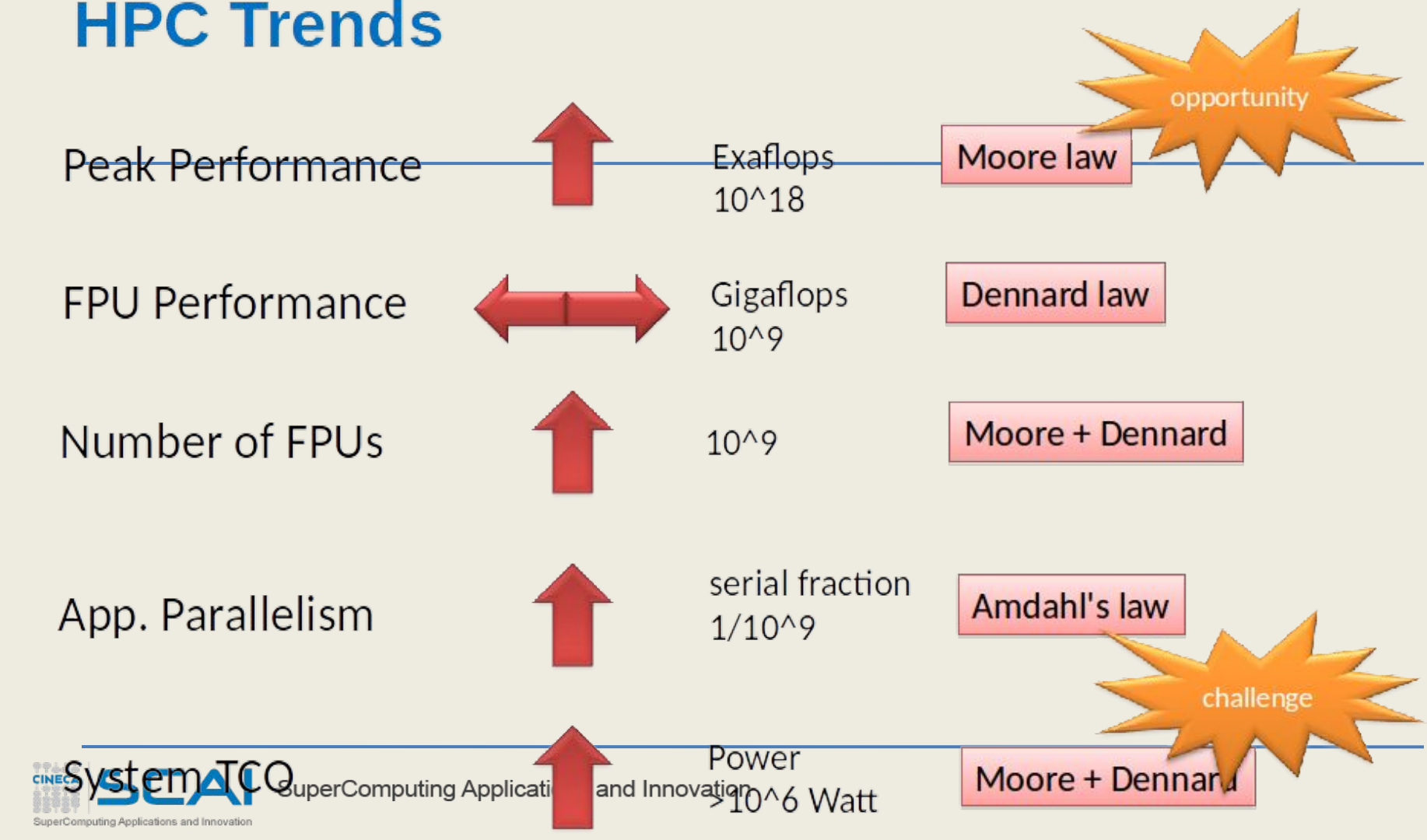
Amdahl's law



maximum speedup tends to $1 / (1 - P)$
P= parallel fraction

The upper limit for the scalability of parallel applications is determined by the fraction of the overall execution time spent in non-parallel operations.

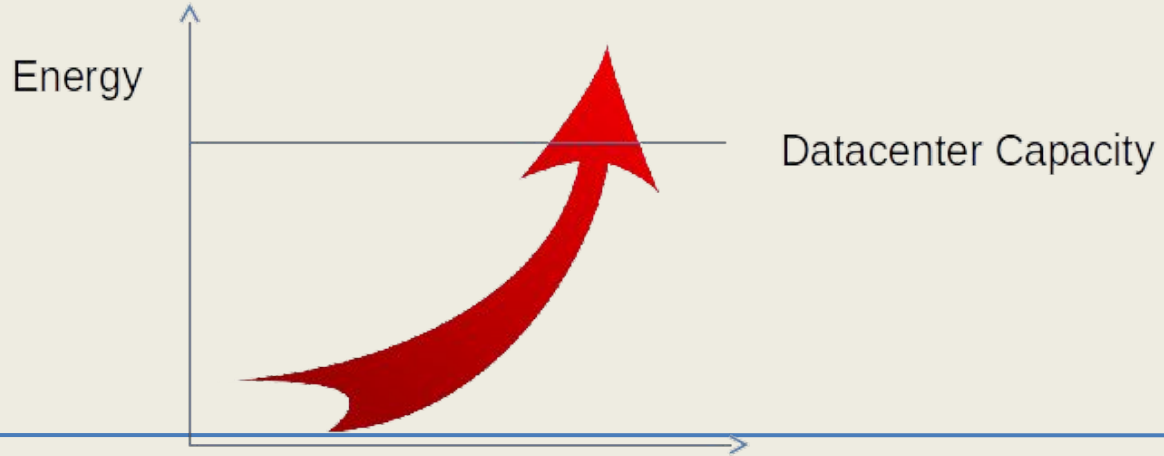
HPC Trends


Peak Performance
Exaflops
10^18
opportunity
Moore law
FPU Performance
Gigaflops
10^9
Dennard law
Number of FPUs
10^9
Moore + Dennard
App. Parallelism
serial fraction
1/10^9
Amdahl's law
challenge
System TCO
Power
>10^6 Watt
Moore + Dennard

Energy trends

"traditional" RISK and CISC chips are designed for maximum performance for all possible workloads

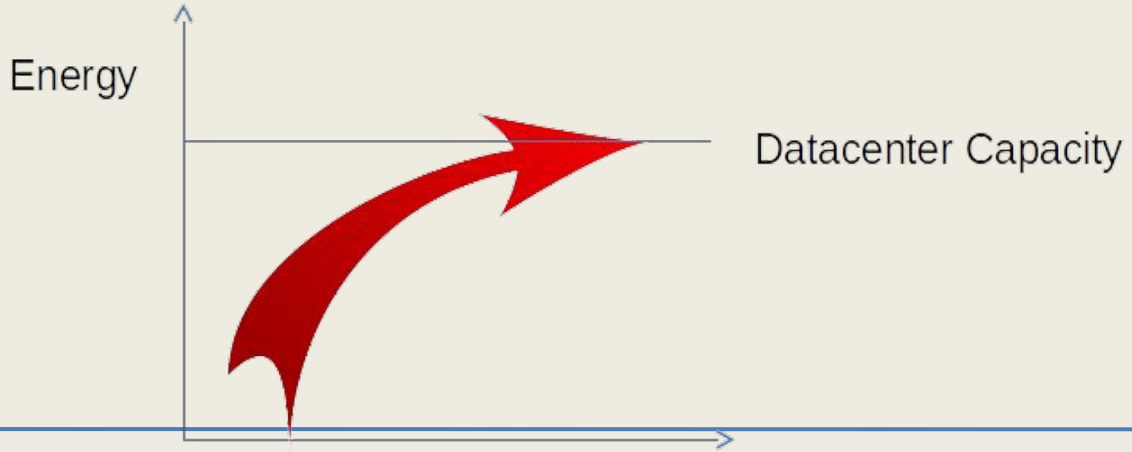
A lot of silicon to maximize single thread performace



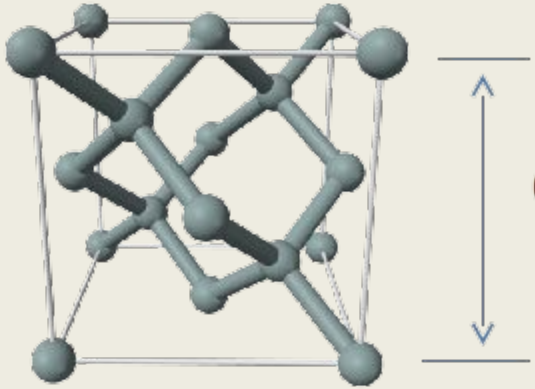
Change of paradigm

New chips designed for maximum performance in a small set of workloads

Simple functional units, poor single thread performance, but maximum throughput

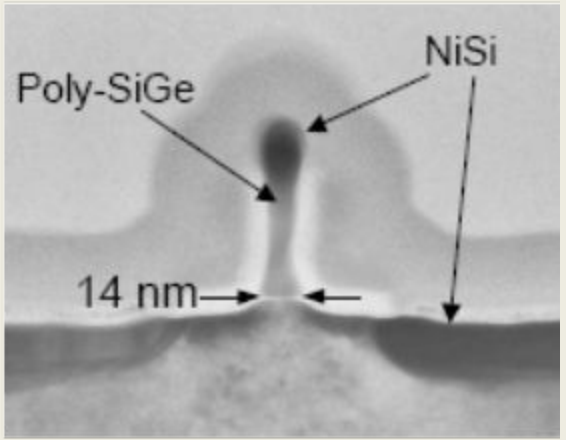


The silicon lattice



0.54 nm

Si lattice



50 atoms!

Exascale architecture

two model

Hybrid — CPU ⟷ ACC — OpenPower/ARM +Nvidia GPU

Homogeneus — SoC — ARM Intel

System attributes	2001	2010	"2015"		"2018"	
System peak	10 Tera	2 Peta	200 Petaflop/sec		1 Exaflop/sec	
Power	~0.8 MW	6 MW	15 MW		20 MW	
System memory	0.006 PB	0.3 PB	5 PB		32-64 PB	
Node performance	0.024 TF	0.125 TF	0.5 TF	7 TF	1 TF	10 TF
Node memory BW		25 GB/s	0.1 TB/sec	1 TB/sec	0.4 TB/sec	4 TB/sec
Node concurrency	16	12	O(100)	O(1,000)	O(1,000)	O(10,000)
System size (nodes)	416	18,700	50,000	5,000	1,000,000	100,000
Total Node Interconnect BW		1.5 GB/s	150 GB/sec	1 TB/sec	250 GB/sec	2 TB/sec
MTTI		day	O(1 day)		O(1 day)	

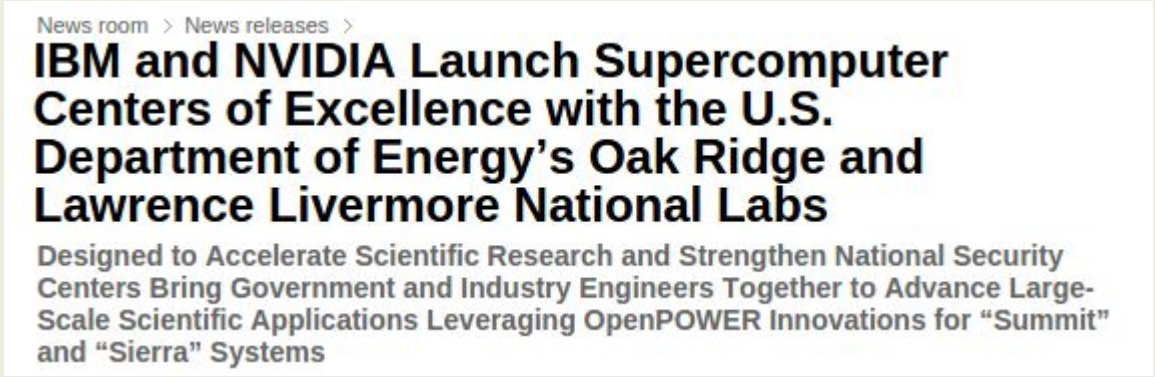
SuperComputing Applications and Innovation

Available options…

Right here, right now… two kind of solutions are available on the market:

- IBM+ nVIDIA (Coral-like)
- Intel-based (Xeon/Xeon Phi)

IBM+NVIDIA



News room > News releases >

IBM and NVIDIA Launch Supercomputer Centers of Excellence with the U.S. Department of Energy's Oak Ridge and Lawrence Livermore National Labs

Designed to Accelerate Scientific Research and Strengthen National Security Centers Bring Government and Industry Engineers Together to Advance Large-Scale Scientific Applications Leveraging OpenPOWER Innovations for "Summit" and "Sierra" Systems

Each node will be based on a Power CPU + 4/6/8 nVIDIA TESLA GPUs connected using an nVIDIA NVlink interconnect

Intel Xeon and Xeon Phi

Intel will keep on with the production of server processors on the Xeon line, together with the introduction of the Xeon Phi many-core chips

Intel Xeon Phi will not be a co-processor anymore, but a self-standing CPU with a very high number of cores

Such systems are integrated by several vendors in many different configurations (Cray, HP, Lenovo, E4..)

MARCONI

FERMI, the IBM BlueGene/Q deployed in Cineca ended its lifecycle in 2016

We needed a new HPC machine that could

- increase the computational power
- respect the agreements with PRACE
- satisfy the needs of the italian computing community

MARCONI

Tender proposal

Partition	Installation	CPU	# nodes	# of Racks	Power
A1 – Broadwell (2.1PFlops)	April 2016	E5-2697 v4	1512	25	700KW
A2 - Knight Landing (11 Pflops)	September 2016	KNL	3600	50	1300KW
A3 – Skylake (4.5PFlops)	June 2017	E5-2680 v5	1512	25	700KW

Network: Intel OmniPath

MARCONI

NeXtScale architecture
nx360M5 nodes:
- Supporting Intel HSW & BDW
- Able to host both IB network Mellanox EDR & Intel Omni-Path
- Twelve nodes are grouped into a Chassis (6 chassis per rack)
- The compute node is made of:
  - 2 x Intel Broadwell (Xeon processor E5-2697 v4) 18 cores, 2,3 HGz
  - 8 x 16GB DIMM memory (RAM DDR4 2400 MHz), 128 GB total
  - 1 x 129 GB SATA MLC S3500 Enterprise Value SSD
- Further details
  - 1 x link OPA 100GBs
  - 2*18*2,3*16 = 1.325 GFs peak
- 24 rack in total:
  - 21 rack compute
  - 1 rack service nodes
  - 2 racks core switch

MARCONI - Network

Network type: new Intel Omnipath
Largest Omnipath cluster of the world

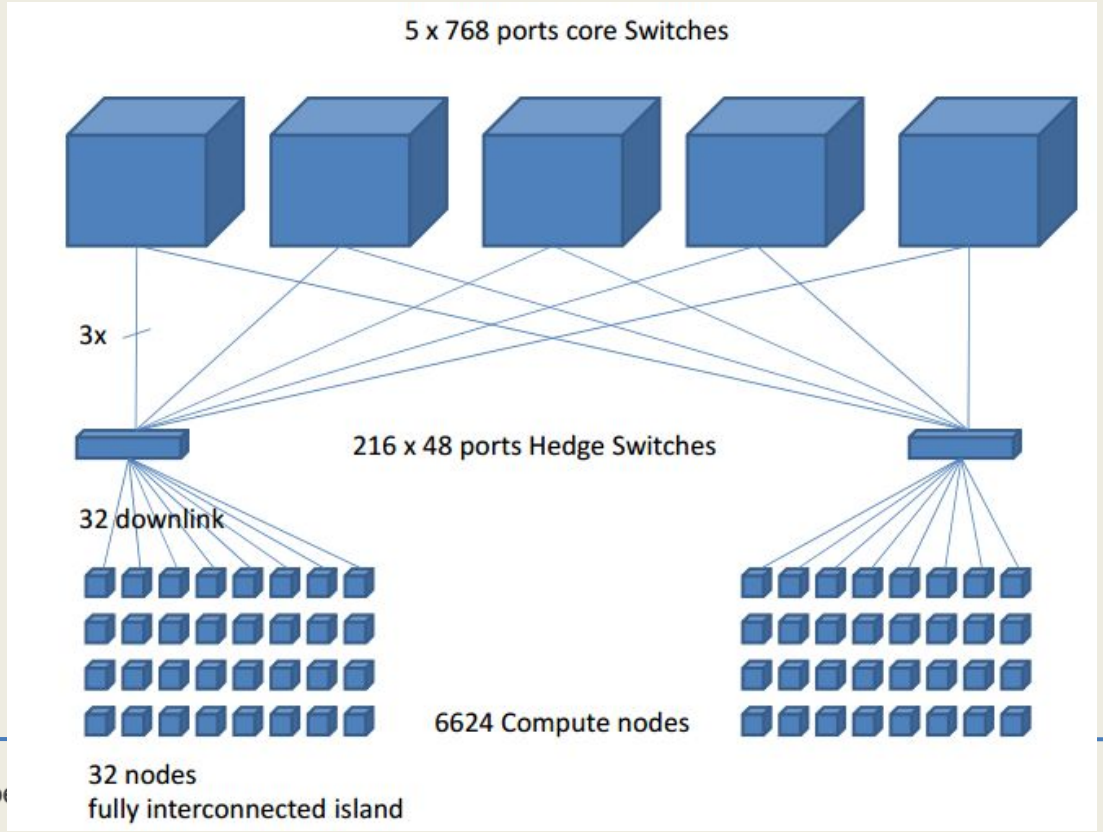
Network topology: Fat-tree
2:1 oversubscription
tapering at the level of the core switches only

Core Switches: 5 x OPA Core Switch “Sawtooth Forest”
768 ports each

Hdge Switch: 216 OPA Edge Switch “Eldorado Forest”
48 ports each

Maximum system configuration:
5(opa) x 768(ports) x 2(tapering) -> 7680 servers

MARCONI - Network



MARCONI - Storage

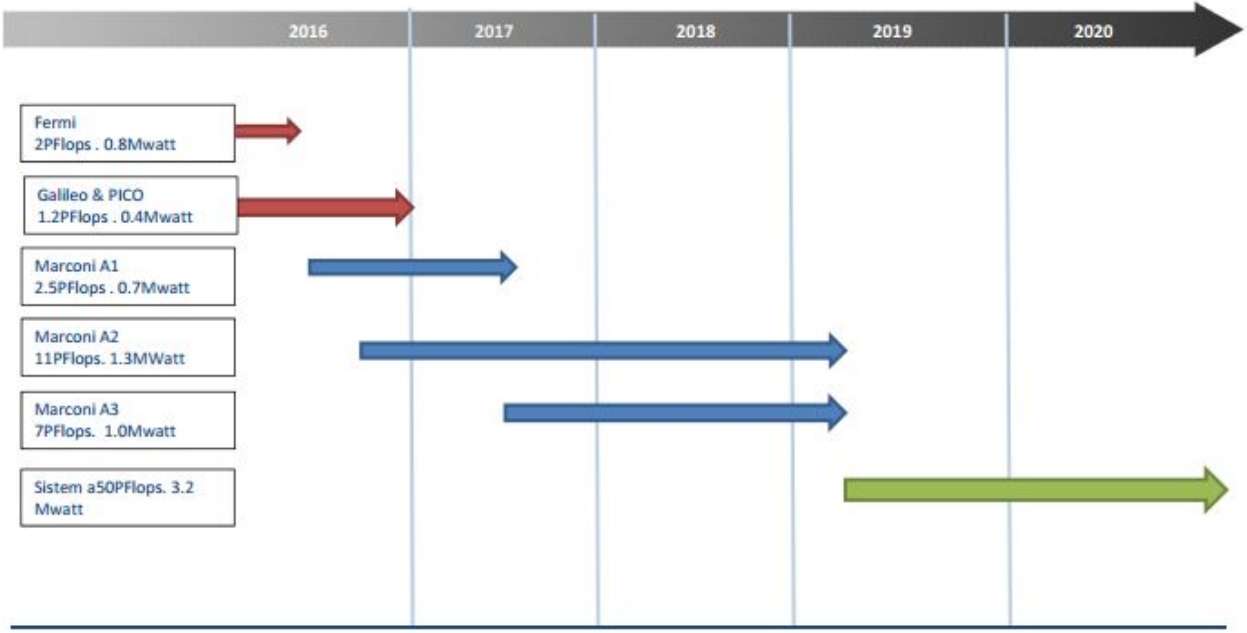
Storage system:
6 x Lenovo GSS-26 Storage

Storage capacity:
17PB(raw capacity)

Storage bandwidth:
100GByte/sec (sustained)

Storage network:
Intel Omnipath (directly attached to the OPA switchs)

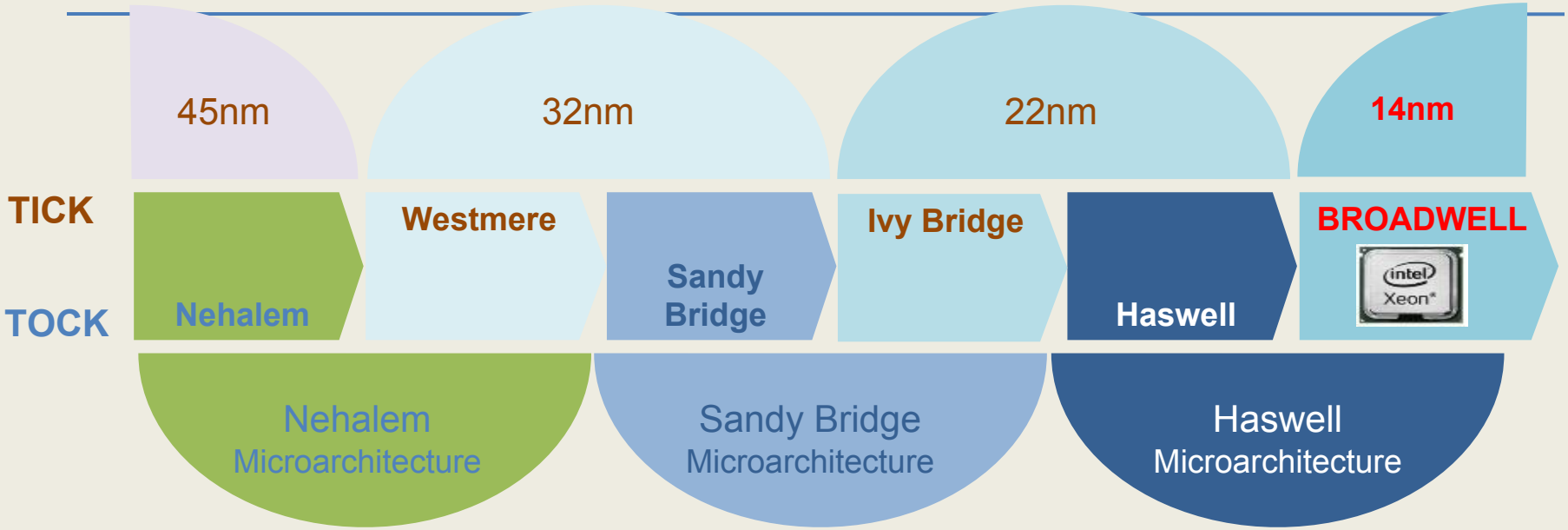
CINECA - Roadmap



1.2Mwatt	2.4Mwatt	2.3Mwatt	2.3Mwatt	3.2Mwatt
50 rack	120 rack	120 rack	120 rack	150 rack
100mq	240mq	240mq	240mq	300mq

Intel® Xeon Processor Architecture

Intel® Xeon® Processor E5-2600 v4 Product Family - TICK



Typically, Increases in Transistor Density Enables New Capabilities, Higher Performance Levels, and Greater Energy Efficiency

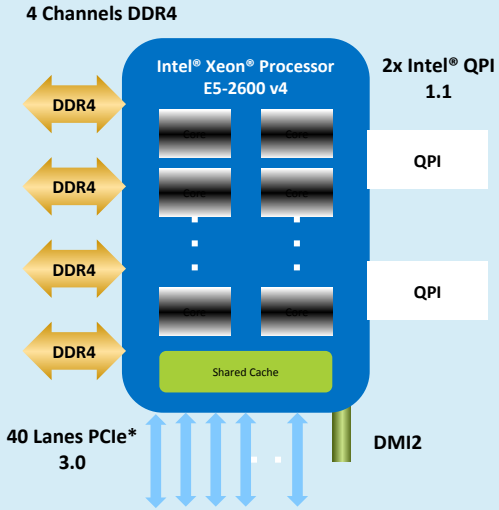
Intel® Xeon® E5-2600 v4 Product Family Overview

New Features:
- Broadwell microarchitecture
- Built on 14nm process technology
- Socket compatible◊ replacement/ upgrade on Grantley-EP platforms

New Performance Technologies:
- Optimized Intel® AVX Turbo mode
- Intel TSX instructions^

Other Enhancements:
- Virtualization speedup
- Orchestration control
- Security improvements

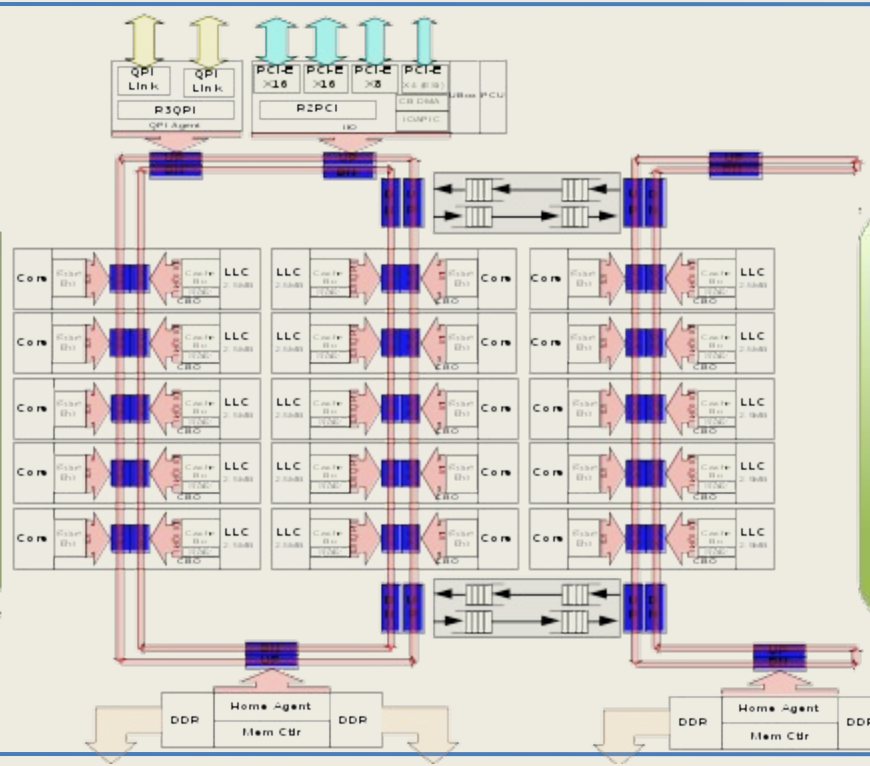
Features	Xeon E5-2600 v3 (Haswell-EP)	Xeon E5-2600 v4 (Broadwell-EP)
Cores Per Socket	Up to 18	**Up to 22**
Threads Per Socket	Up to 36 threads	**Up to 44 threads**
Last-level Cache (LLC)	Up to 45 MB	**Up to 55 MB**
QPI Speed (GT/s)	2x QPI 1.1 channels 6.4, 8.0, 9.6 GT/s	
PCIe* Lanes / Speed(GT/s)	40 / 10 / PCIe* 3.0 (2.5, 5, 8 GT/s)	
Memory Population	4 channels of up to 3 RDIMMs or 3 LRDIMMs	**+ 3DS LRDIMM†**
Memory RAS	ECC, Patrol Scrubbing, Demand Scrubbing, Sparing, Mirroring, Lockstep Mode, x4/x8 SDDC	**+ DDR4 Write CRC**
Max Memory Speed	Up to 2133	**Up to 2400**
TDP (W)	160 (Workstation only), 145, 135, 120, 105, 90, 85, 65, 55	



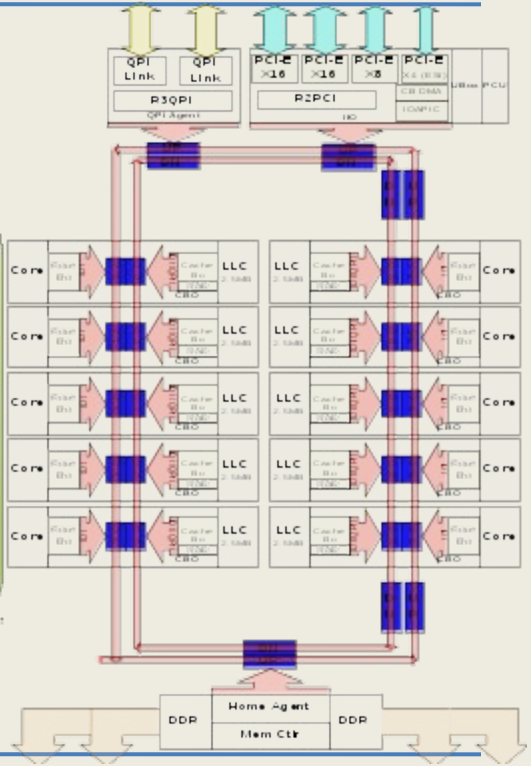
◊ Requires BIOS and firmware update; ^ not available broadly on E5-2600 v3; † Depends on market availability

Intel® Xeon® Processor E5-2600 v4 Product Family MCC/LCC

Medium Core Count (MCC) SKUs
12C to 14C
E5-2690 v4
E5-2680 v4 E5-2660 v4 E5-2650 v4 E5-2650L v4 E5-2687W v4



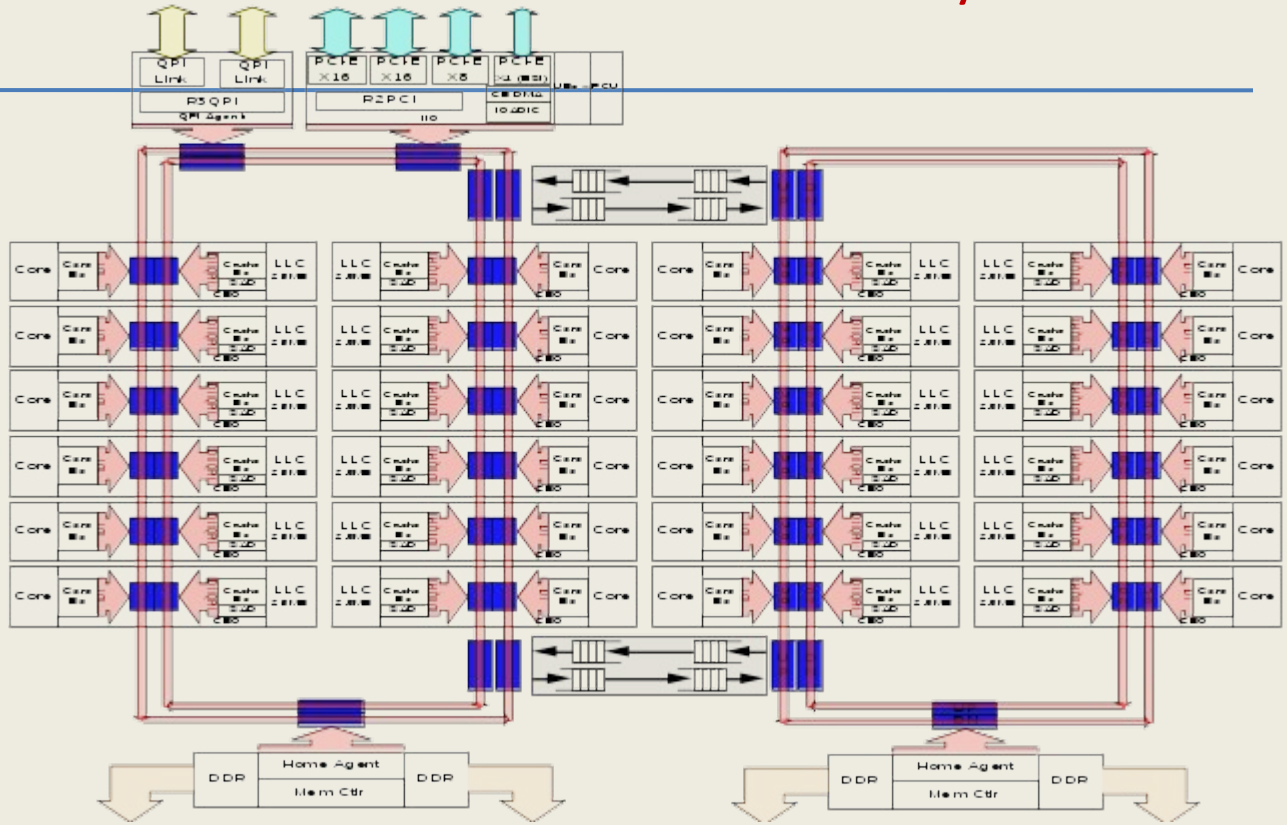
Low Core Count (LCC) SKUs
4C to 10C
E5-2640 v4
E5-2630 v4
E5-2620 v4
E5-2609 v4
E5-2667 v4
E5-2643 v4
E5-2637 v4
E5-2630L v4



Intel® Xeon® Processor E5-2600 v4 Product Family HCC

High core count (HCC) die configuration

- Used by SKUs with 16 to 22 cores
  - E5-2699 v4
  - E5-2698 v4
  - E5-2697 v4
  - E5-2697A v4
  - E5-2695 v4
  - E5-2683 v4
- For each core
  - 2.5M last level cache (LLC)
  - Caching agent (CBO)
- For each ring
  - Home agent (HA)
  - Memory Controller with 2 DDR4 channels



What's next …

- Broadwell (code name) E7 (4-socket server processor models)
- Skylake (code name) server (E5 and E7)
  - Micro-architecture launched in client processors Sep. 2015
  - Intel® AVX-512 (only for server)
  - Expect a lot of additional, key changes
- FPGA and Xeon server integration
- NVM (non-volatile memory) - 3D XPoint™ Technology

Intel® Many Integrated Core Architecture
Intel® Xeon Phi™ Coprocessor

Intel® Xeon Phi™ Product Family

based on Intel® Many Integrated Core (MIC) Architecture

2013:

Intel® Xeon Phi™ Coprocessor x100 Product Family

"Knights Corner"

22 nm process

Up to 61 Cores

Up to 16GB Memory

2016:

Second Generation Intel® Xeon Phi™

"Knights Landing"

14 nm

Processor & Coprocessor

+60 cores

On Package, High-Bandwidth Memory

Future Knights:

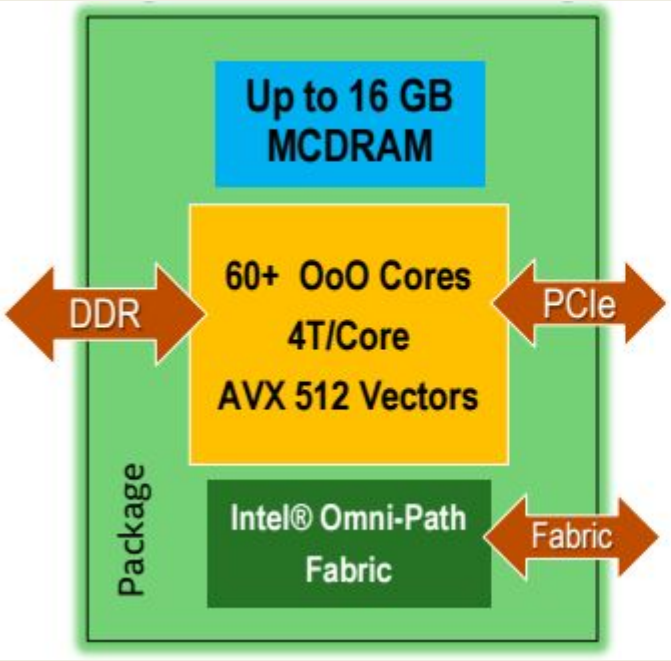
Upcoming Gen of the Intel® MIC Architecture

In planning

Continued roadmap commitment

*Per Intel's announced products or planning process for future products

Knights Landing: Next-Generation Intel® Xeon Phi™



Stand-alone, Self-boot CPU

60+ new Silvermont-based cores

4 Threads per core

AVX 512 vector units

Binary Compatible[1] with Intel® Xeon® processor

2-dimensional Mesh on-die interconnect

MCDRAM: On-Package memory: 400+ GB/s of BW[2]

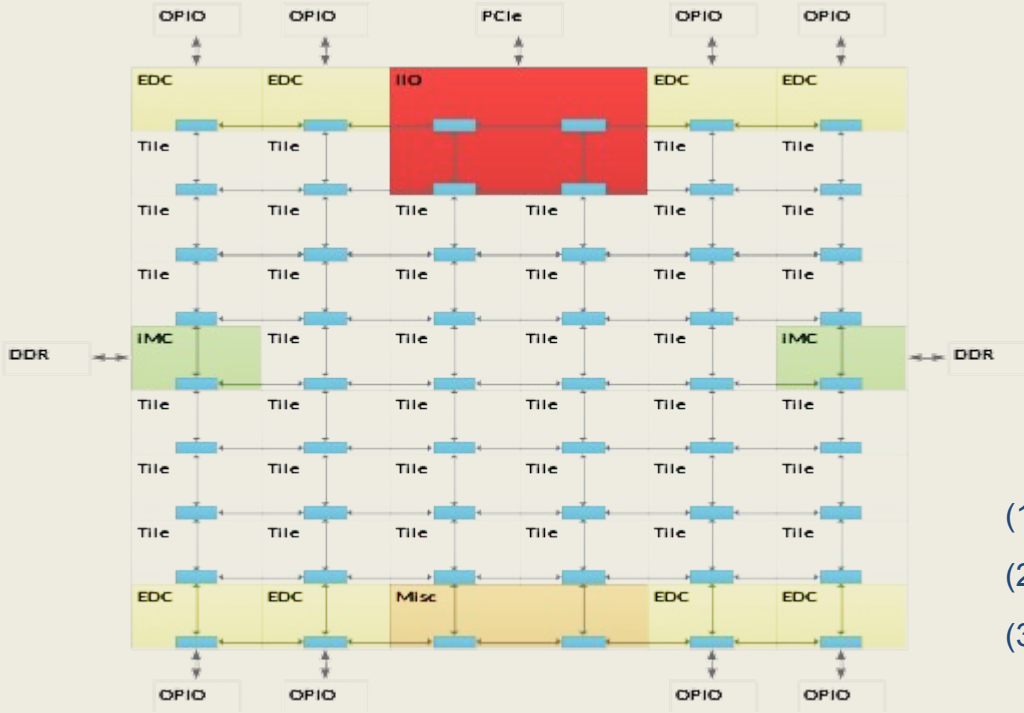
DDR memory

Intel® Omni-path Fabric

3+ TFLops (DP) peak per package

~3x ST performance over KNC

KNL Mesh Interconnect



Mesh of Rings

- Every row and column is a (half) ring
- YX routing: Go in Y → Turn → Go in X
- Messages arbitrate at injection and on turn

Cache Coherent Interconnect

- MESIF protocol (F = Forward)
- Distributed directory to filter snoops

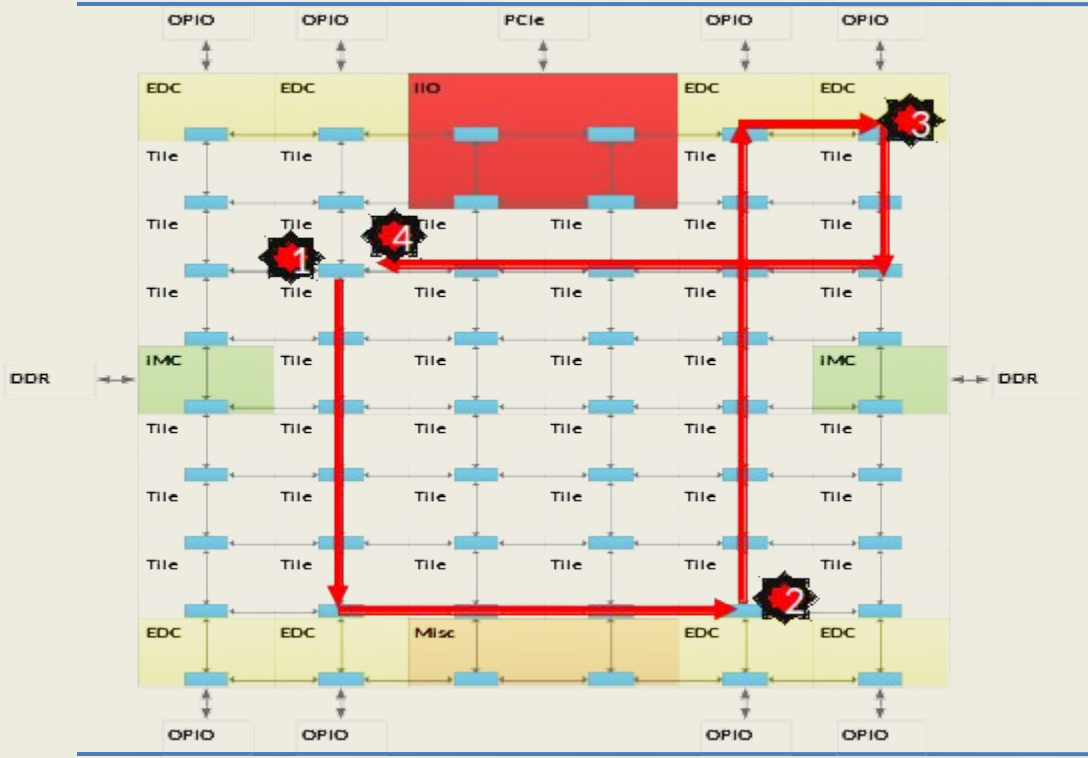
Three Cluster Modes

(1) All-to-All

(2) Quadrant

(3) Sub-NUMA Clustering (SNC)

Cluster Mode: All-to-All



Address uniformly hashed across all distributed directories

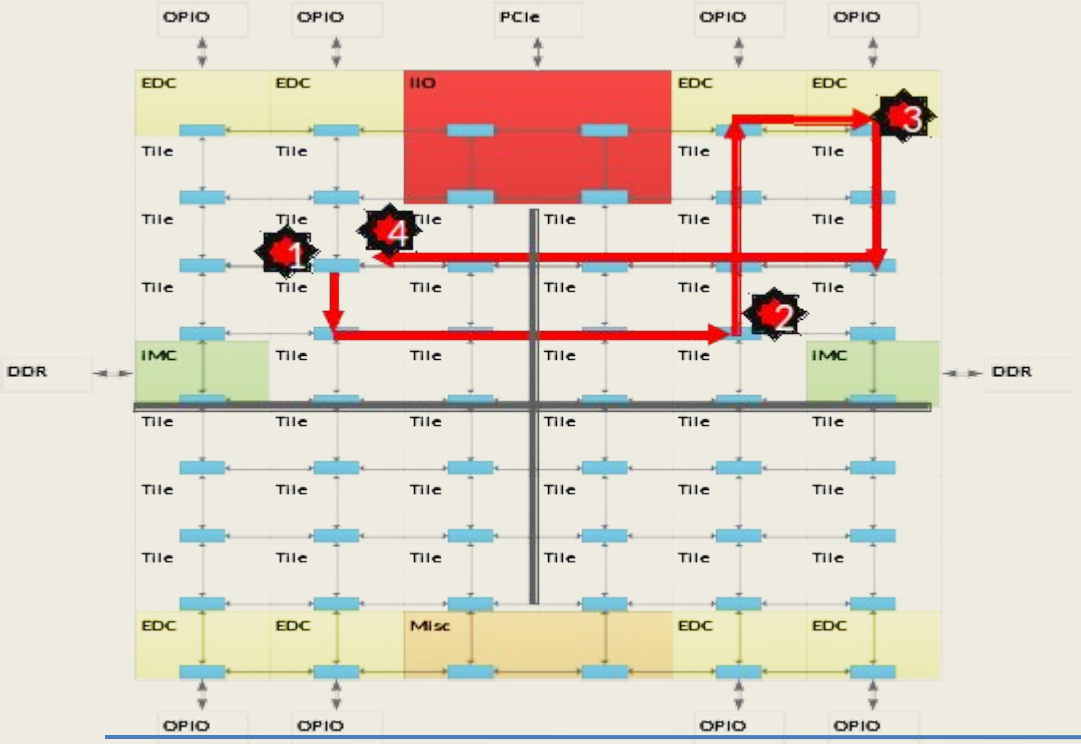
No affinity between Tile, Directory and Memory

Lower performance mode, compared to other modes. Mainly for fall-back

Typical Read L2 miss

1. L2 miss encountered
2. Send request to the distributed directory
3. Miss in the directory. Forward to memory
4. Memory sends the data to the requestor

Cluster Mode: Quadrant



Chip divided into four virtual Quadrants

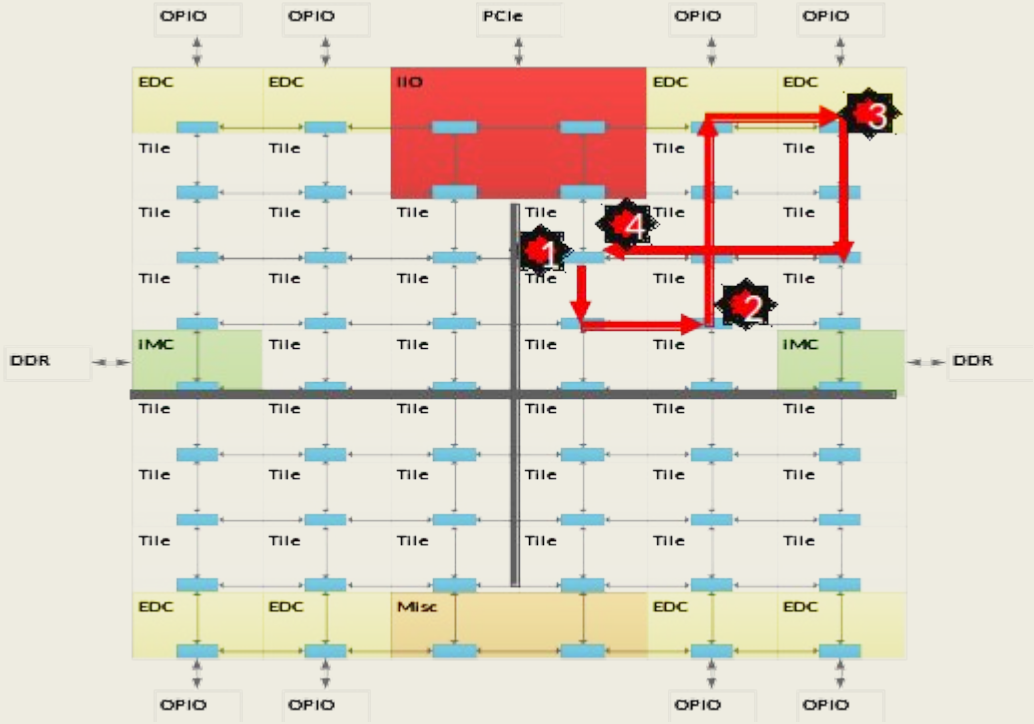
Address hashed to a Directory in the same quadrant as the Memory

Affinity between the Directory and Memory

Lower latency and higher BW than all-to-all. Software transparent.

1. L2 miss, 2. Directory access, 3. Memory access, 4. Data return

Cluster Mode: Sub-NUMA Clustering (SNC)



Each Quadrant (Cluster) exposed as a separate NUMA domain to OS

Looks analogous to 4-Socket Xeon

Affinity between Tile, Directory and Memory

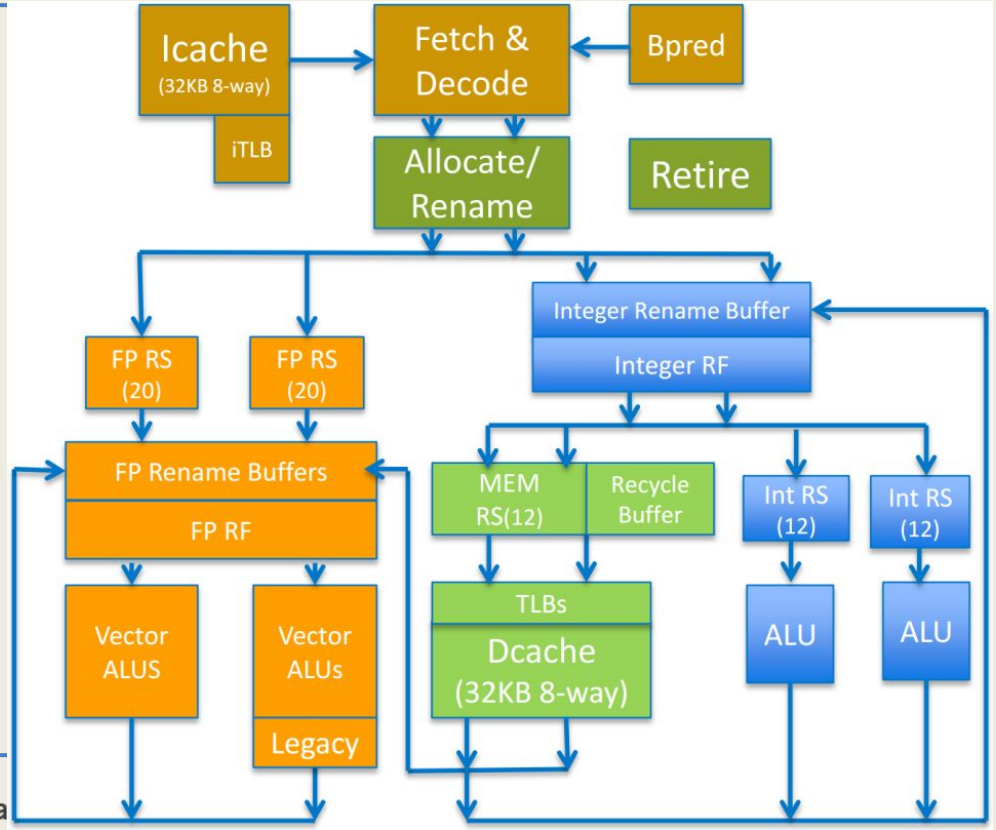
Local communication. Lowest latency of all modes

Software needs to be NUMA-aware to get benefit

1. L2 miss, 2. Directory access, 3. Memory access, 4. Data return

KNL Core and VPU

Out-of-order core w/ 4 SMT threads
VPU tightly integrated with core pipeline
2-wide decode/rename/retire
2x 64B load & 1 64B store port for D$
L1 prefetcher and L2 prefetcher
Fast unaligned and cache-line split support
Fast gather/scatter support



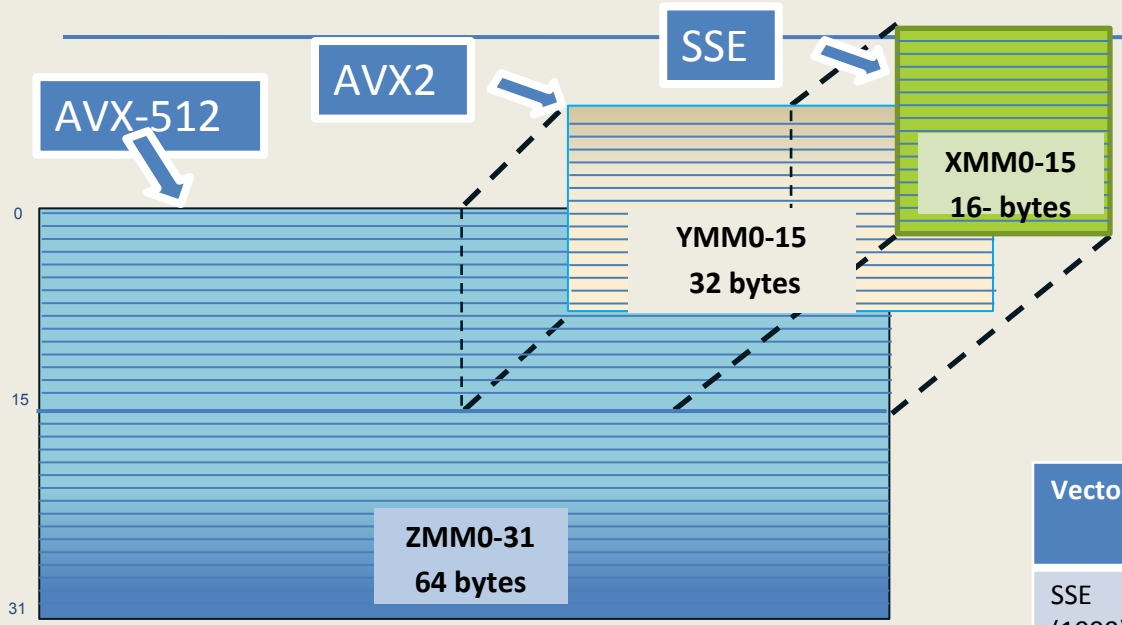
Software Adaption for KNL – Key New Features

Large impact: Intel® AVX-512 instruction set
- Slightly different from future Intel® Xeon™ architecture AVX-512 extensions
- Includes SSE, AVX, AVX-2
- Apps built for HSW and earlier can run on KNL (few exceptions like TSX)
- Incompatible with 1st Generation Intel® Xeon™ Phi (KNC)

Medium impact: New, on-chip high bandwidth memory (MCDRAM) creates heterogeneous (NUMA) memory access
- can be used transparently too however

Minor impact: Differences in floating point execution / rounding due to FMA and new HW-accelerated transcendental functions - like exp()

AVX-512 - Greatly increased Register File



Vector Registers	IA32 (32bit)	Intel64 (64bit)
SSE (1999)	8 x 128bit	16 x 128bit
AVX and AVX-2 (2011 / 2013)	8 x 256bit	16 x 256bit
AVX-512 (2014 – KNL)	8 x 512bit	32 x 512bit

The Intel® AVX-512 Subsets [1]

AVX-512 F

AVX-512 F: 512-bit Foundation instructions common between MIC and Xeon

- ❑ Comprehensive vector extension for HPC and enterprise
- ❑ All the key AVX-512 features: masking, broadcast…
- ❑ 32-bit and 64-bit integer and floating-point instructions
- ❑ Promotion of many AVX and AVX2 instructions to AVX-512
- ❑ Many new instructions added to accelerate HPC workloads

AVX-512CD

AVX-512 CD (Conflict Detection instructions)

- ❑ Allow vectorization of loops with possible address conflict
- ❑ Will show up on Xeon

AVX-512ER

AVX-512PR

AVX-512 extensions for exponential and prefetch operations

- ❑ fast (28 bit) instructions for exponential and reciprocal and transcendentals (as well as RSQRT)
- ❑ New prefetch instructions: gather/scatter prefetches and PREFETCHWT1

The Intel® AVX-512 Subsets [2]

AVX-512DQ

AVX-512 Double and Quad word instructions

- ❑ All of (packed) 32bit/64 bit operations AVX-512F doesn't provide
- ❑ Close 64bit gaps like VPMULLQ : packed 64x64 ➔ 64
- ❑ Extend mask architecture to word and byte (to handle vectors)
- ❑ Packed/Scalar converts of signed/unsigned to SP/DP

AVX-512BW

AVX-512 Byte and Word instructions

- ❑ Extent packed (vector) instructions to byte and word (16 and 8 bit) datatype
- ❑ MMX/SSE2/AVX2 re-promoted to AVX512 semantics
- ❑ Mask operations extended to 32/64 bits to adapt to number of objects in 512bit
- ❑ Permute architecture extended to words (VPERMW, VPERMI2W, …)

AVX-512VL

AVX-512 Vector Length extensions

- ❑ Vector length orthogonality
- ❑ Support for 128 and 256 bits instead of full 512 bit
- ❑ Not a new instruction set but an attribute of existing 512bit instructions

Other New Instructions

MPX

Intel® MPX – Intel Memory Protection Extension

- ❑Set of instructions to implement checking a pointer against its bounds
- ❑Pointer Checker support in HW (today a SW only solution of e.g. Intel compilers)
- ❑Debug and security features

SGX

Intel® SGX – Intel® Software Guard Extensions

- ❑ Intel® Software Guard Extensions enables applications to execute code and protect secrets from within their own protected execution environment, giving developers direct control over their application security

CLFLUSHOPT

Single Instruction – Flush a cache line

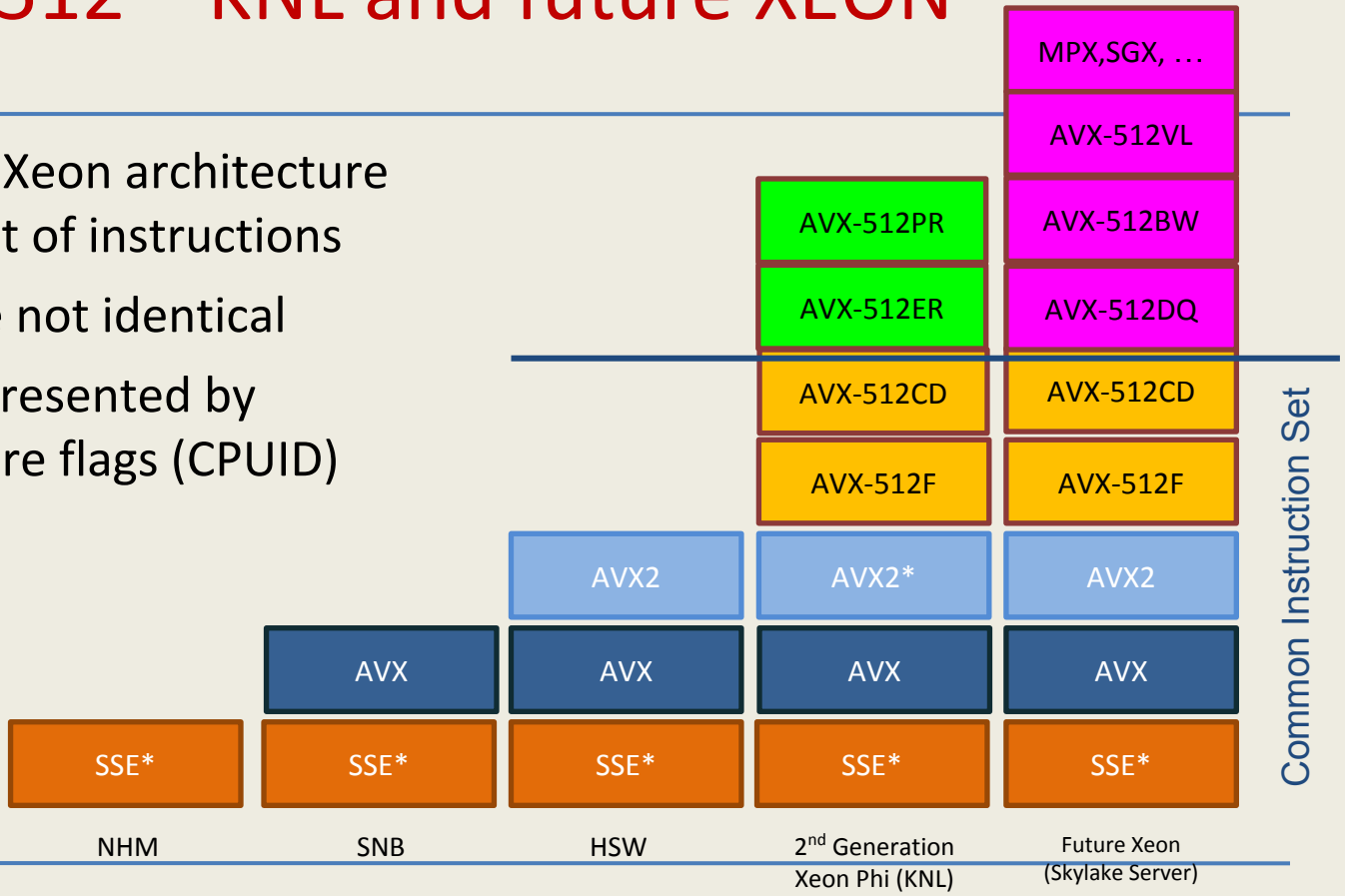
- ❑ needed for future memory technologies

XSAVE{S,C}

Save and restore extended processor state

AVX-512 – KNL and future XEON

- KNL and future Xeon architecture share a large set of instructions
  - but sets are not identical
- Subsets are represented by individual feature flags (CPUID)



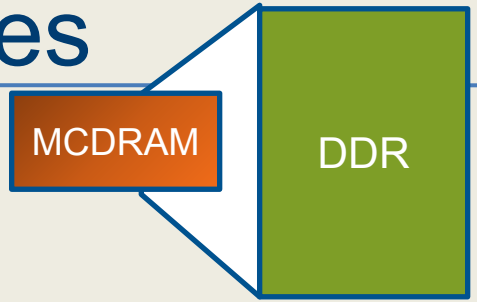
Intel® Compiler Processor Switches

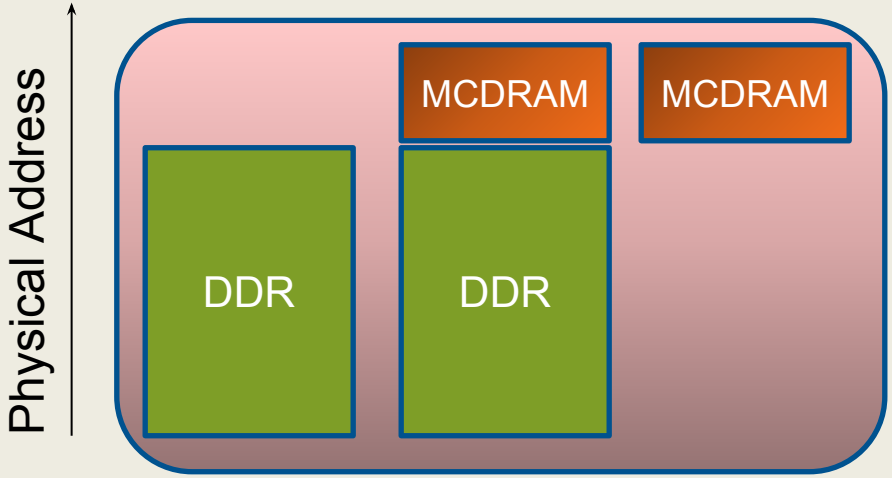
Switch	Description
`-xmic-avx512`	KNL only; already in 14.0
`-xcore-avx512`	Future XEON only, already in 15.0.1
`-xcommon-avx512`	AVX-512 subset common to both, already in 15.0.2
`-m, -march, /arch`	Not yet !
`-ax<…-avx512>`	Same as for “-x<…-avx512>”
`-mmic`	No – not for KNL

KNL Memory Modes

- Mode selected at boot
- MCDRAM-Cache covers all DDR

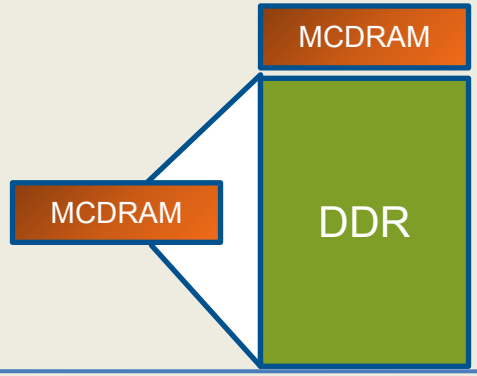
Cache Model



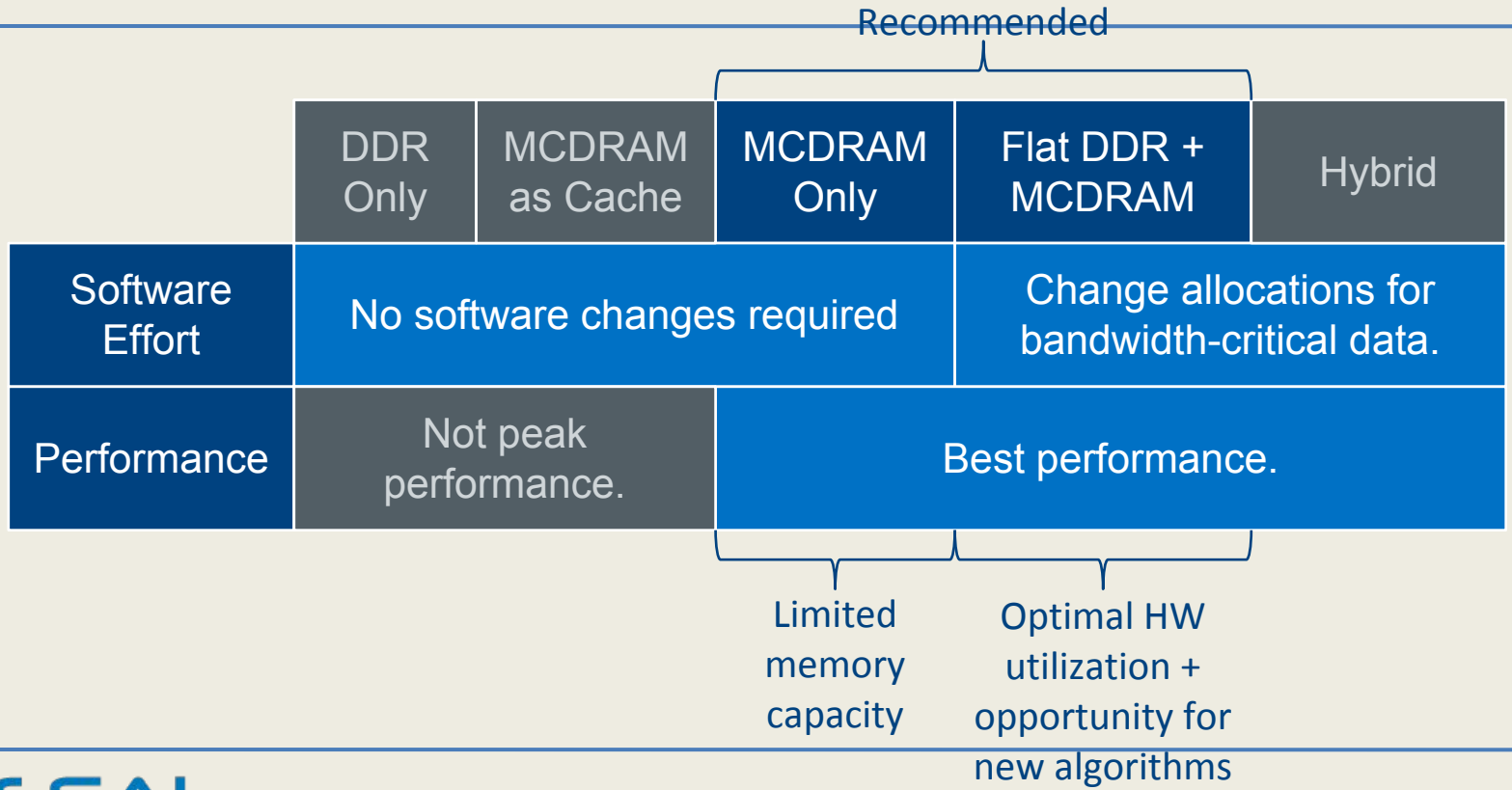


Flat Models

Hybrid Model



MCDRAM: Cache vs Flat Mode



High Bandwidth On-Chip Memory API

- API is open-sourced (BSD licenses)
  - https://github.com/memkind
  - Uses jemalloc API underneath
    - http://www.canonware.com/jemalloc/
    - https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919

Malloc replacement:

```
#include <memkind.h>

  hbw_check_available()
  hbw_malloc, _calloc, _realloc,… (memkind_t kind, …)
  hbw_free()
  hbw_posix_memalign()
  hbw_get_size(), _psize()

ld …  -ljemalloc -lnuma -lmemkind -lpthread
```

HBW API for Fortran, C++

Fortran:

!DIR$ ATTRIBUTES FASTMEM :: data_object1, data_object2

- All Fortran data types supported
- Global, local, stack or heap; scalar, array, …
- Support in compiler 15.0 update 1 and later versions

C++:

standard allocator replacement for e.g. STL like

```
#include <hbwmalloc.h>
std::vector<int, hbwmalloc::hbw_allocator>
```

Porting codes on Knights Landing

SuperComputing Applications and Innovation

Trends that are here to stay

Data parallelism

- Lots of threads, spent on MPI ranks or OpenMP/TBB/pthreads
- Improving support for both peak tput and modest/single thread

Bigger, better, faster memory

- High capacity, high bandwidth, low latency DRAM
- Effective caching and paging
- Increasing support for irregular memory refs, modest tuning

ISA innovation

- Increasing support for vectorization, new usages

Evolution or revolution?

Incremental changes, significant gains

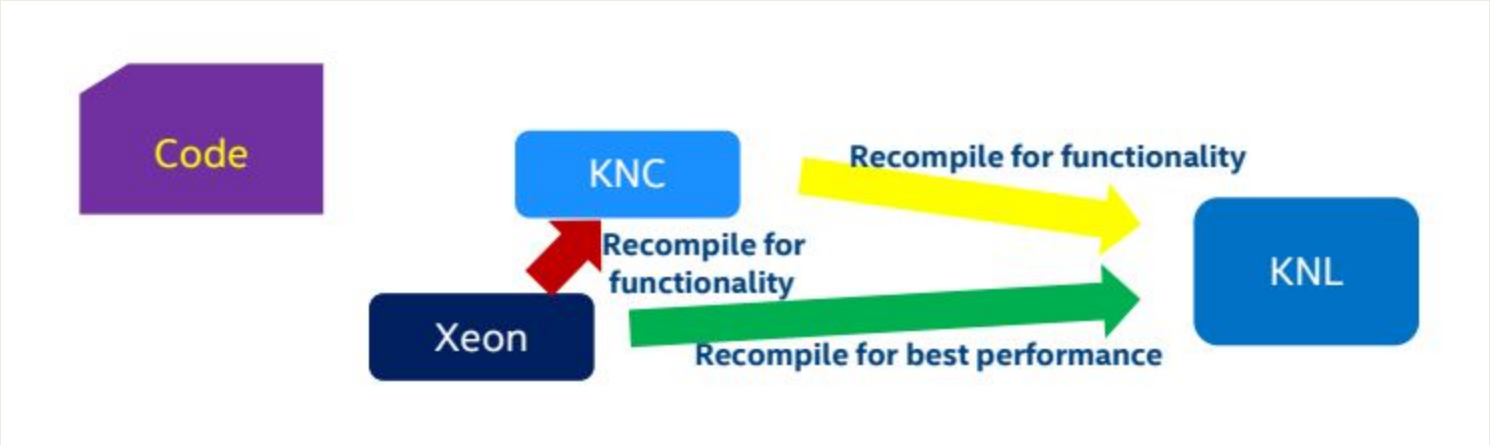
Parallelization – consistent strategy

- MPI vs. OpenMP – already needed to tune and tweak
- Less thread-level parallelism required
- Vectorization – more opportunity, more profitable

Enable new features with memory tuning

- Access MCDRAM with special allocation
- Blocking for MCDRAM vs. just cache

Compatibility



KNL specific enabling

- Recompilation, with –xMIC-AVX512
- Threading: more MPI ranks, 1 thread/core
- Vectorization: increased efficiency
- MCDRAM and memory tuning: tile, 1GB pages

What is needed?

- Building

Change compiler switches in make files

- Coding

Parallelization: vectorization, offload

Memory management: MCDRAM enumeration and memory allocation

- Tuning

Potentially fewer threads: more cores but less need for SMT

More memory more MPI ranks

Take-aways

Keep doing what you were doing for KNC and Xeon

Some goodness comes for free with a recompile

With some extra enabling, use new MCDRAM feature

Programming paradigms for the future?

Fabio Affinito (SCAI - Cineca)

What tools are there?

Differently from hetereogenous architectures (i.e. GPUs), changes can be made without being dramaticaly disruptives.

GPUs require to use CUDA, or in alternative, OpenMP4 directives to address the data copy between host and devices.

Some solutions based on PGAS is avalaible but still far to be optimal

So what?

Actually you could use codes already compiled for a x86 arch and they could work (in theory) on a Haswell/Broadwell/KNL processor.

So, at least on principle, the existence of legacy codes is completely preserved.

But…

The fact that a code can run on a KNL architecture doesn't mean that it is really exploiting all the power offered by your machine.

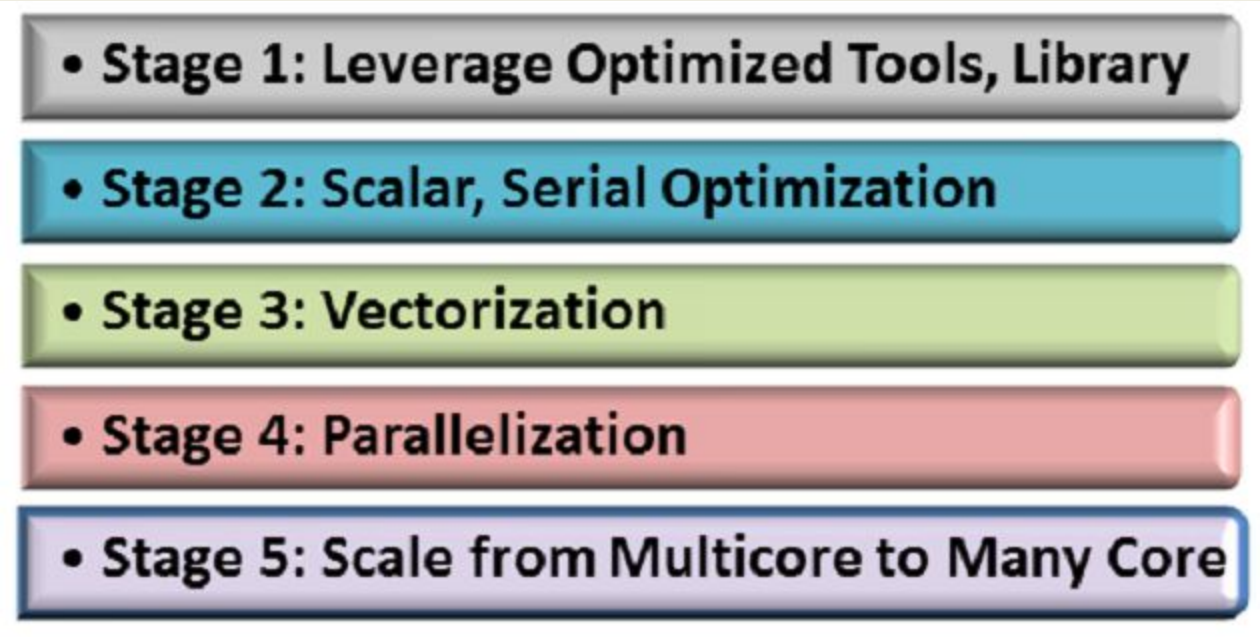
It's like driving a car always with the second gear..

Code modernization: a must

If you don’t want to waste resources (i.e. money and time) you may need to:

- **modernize** your code
- **re-write** your code from scratch (yeah, just like if you were porting it on a GPU!)

Code modernization



Code modernization - 1st stage

At the beginning of your optimization project, select an optimizing development environment. The decision you make at this step will have a profound influence in the later steps. Not only will it affect the results you get, it could substantially reduce the amount of work to do.

The right optimizing development environment can provide you with good compiler tools, optimized, ready-to-use libraries, and debugging and profiling tools to pinpoint exactly what the code is doing at the runtime.

Code modernization - 2nd stage

Before you begin active parallel programming, you need to make sure your application delivers the right results before you vectorize and parallelize it. Equally important, you need to make sure it does the minimum number of operations to get that correct result.

- Choosing the right floating point precision
- Choosing the right approximation method accuracy; polynomial vs. rational
- Avoiding jump algorithms
- Reducing the loop operation strength by using iteration calculations
- Avoiding or minimizing conditional branches in your algorithms
- Avoiding repetitive calculations, using previously calculated results.

Code modernization - 3rd stage

Try **vector level parallelism**. First try to vectorize the inner most loop. For efficient vector loops, make sure that there is minimal control flow divergence and that memory accesses are coherent.

Outer loop vectorization is a technique to enhance performance. By default, compilers attempt to vectorize innermost loops in nested loop structures. But, in some cases, the number of iterations in the innermost loop is small. In this case, inner-loop vectorization is not profitable.

However, if an outer loop contains more work, a combination of elemental functions, strip-mining, and pragma/directive SIMD can force vectorization at this outer, profitable level. (we'll see later…)

Code modernization - 4th step

Now we get to **thread level parallelization**. Identify the outermost level and try to parallelize it. Obviously, this requires taking care of potential data races and moving data declaration to inside the loop as necessary. It may also require that the data be maintained in a cache efficient manner, to reduce the overhead of maintaining the data across multiple parallel paths.

Since the amount of work needs to compensate for the overhead of parallelization, it helps to have as large a parallel effort in each thread as possible. If the outermost level cannot be parallelized due to unavoidable data dependencies, try to parallelize at the next-outermost level that can be parallelized correctly.

4th step: threads

1. If the amount of parallel work achieved at the outermost level appears sufficient for the target hardware and likely to scale with a reasonable increase of parallel resources, you are done. Do not add more parallelism, as the overhead will be noticeable (thread control overhead will negate any performance improvement) and the gains are unlikely.
2. If the amount of parallel work is insufficient, e.g. as measured by core scaling that only scales up to a small core count and not to the actual core count, attempt to parallelize additional layer, as outmost as possible. Note that you don't necessarily need to scale the loop hierarchy to all the available cores, as there may be additional loop hierarchies executing in parallel.
3. If step 2 did not result in scalable code, there may not be enough parallel work in your algorithm. This may mean that partitioning a fixed amount of work among many threads gives each thread too little work, so the overhead of starting and terminating threads swamps the useful work. Perhaps the algorithms can be scaled to do more work, for example by trying on a bigger problem size.
4. Make sure your parallel algorithm is cache efficient. If it is not, rework it to be cache efficient, as cache inefficient algorithms do not scale with parallelism.

Code modernization - 5th step

Lastly we get to **multi-node (Rank) parallelism**. To many developers message passing interface (MPI) is a black box that “just works” behind the scenes, to transfer data from one MPI task (process) to another.

The beauty of MPI for the developer is that the algorithmic coding is hardware independent. The concern that developers have, is that with the many core architecture with 60+ cores, the communication between tasks may create a communication storm either within a node or across nodes. To mitigate these communication bottlenecks, applications should employ hybrid techniques, employing a few MPI tasks and many OpenMP threads.

Leverage your skills!

Some time ago, codes were parallelized using **either** MPI or OpenMP.

Only in some few cases both of them were used.

Nowadays, using MPI+OpenMP is a must!

Two parallelization levels are necessary to exploit both the intra-node (OpenMP) and inter-node (MPI) parallelism.

Leverage your skills!

Plus, a lot of new features of the MPI and OpenMP standards permit to successfully deal with the new HPC architectures, in particular to exploit data-level parallelism, reducing the latencies, etc.

MPI: Non-blocking collectives, RMA, Shred memory parallelism, etc.

OpenMP: SIMD, tasks, etc.

Data-level parallelism

In particular for the new Intel architectures, a lot of performances depend on how much you're able to use the data-level parallelism, in terms of SIMD vectorization.

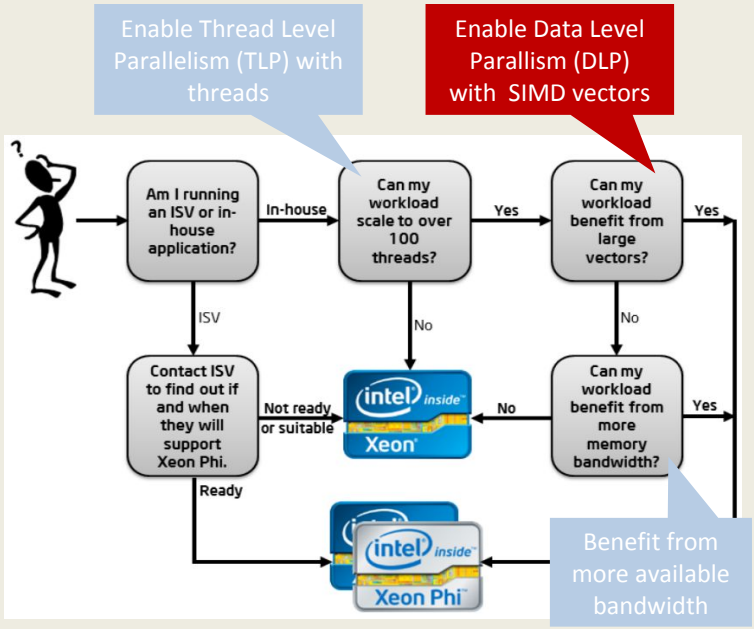
We recall, for example, that in MARCONI A1, a factor 16 of the peak-performance is given by the vectorization...

Overview of the vectorization techniques. Getting ready for AVX-512

Fabio Affinito (SCAI - Cineca)

The need for SIMD vectorization

Is the Intel® Xeon Phi™ coprocessor right for me?



Single thread (ST) performance is limited in today's CPUs

- Clock frequency constraints
- Difficult to discover "near" Instruction level parallelism (ILP) by hardware

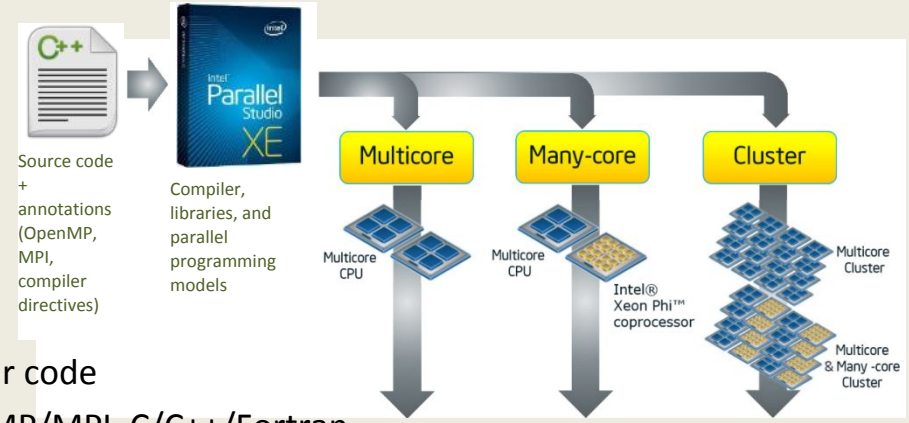
More transistors dedicated to exploit "distant" parallelism

- Task level parallelism (TLP)
  - Improves Multi Thread performance (MT)
- **Data level parallelism (DLP)**
  - Improves Single Thread performance (ST)
  - **Enabled by using SIMD vectors**

"Is the Intel® Xeon Phi™ coprocessor right for me?", by Eric Gardner - https://software.intel.com/en-us/articles/is-the-intel-xeon-phi-coprocessor-right-for-me

How to enable SIMD vectorization?

Enabling parallelism with Intel® Parallel Studio XE 2015 tool suite



Single programming model for all your code

- Based on standards: OpenMP/MPI, C/C++/Fortran
- Programmers/tools responsibility to expose DLP/TLP parallelism

Exposing TLP/DLP in your application will benefit today and future Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors

- Including SIMD vectorization on future Intel® AVX-512 products

Single Instruction Multiple Data (SIMD)

Technique for exploiting DLP on a single thread

- Operate on more than one element at a time
- Might decrease instruction counts significantly

Elements are stored on SIMD registers or *vectors*

Code needs to be *vectorized*

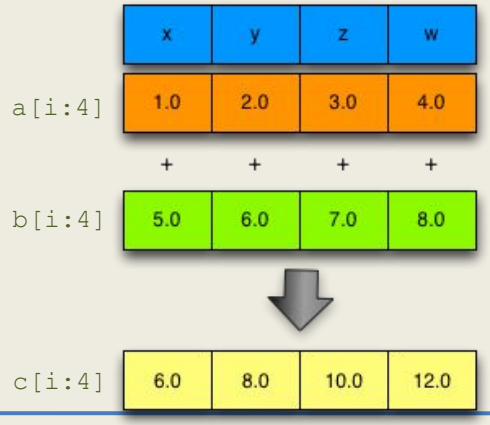
- Vectorization usually on *inner* loops
- Main and *remainder* loops are generated

Scalar loop

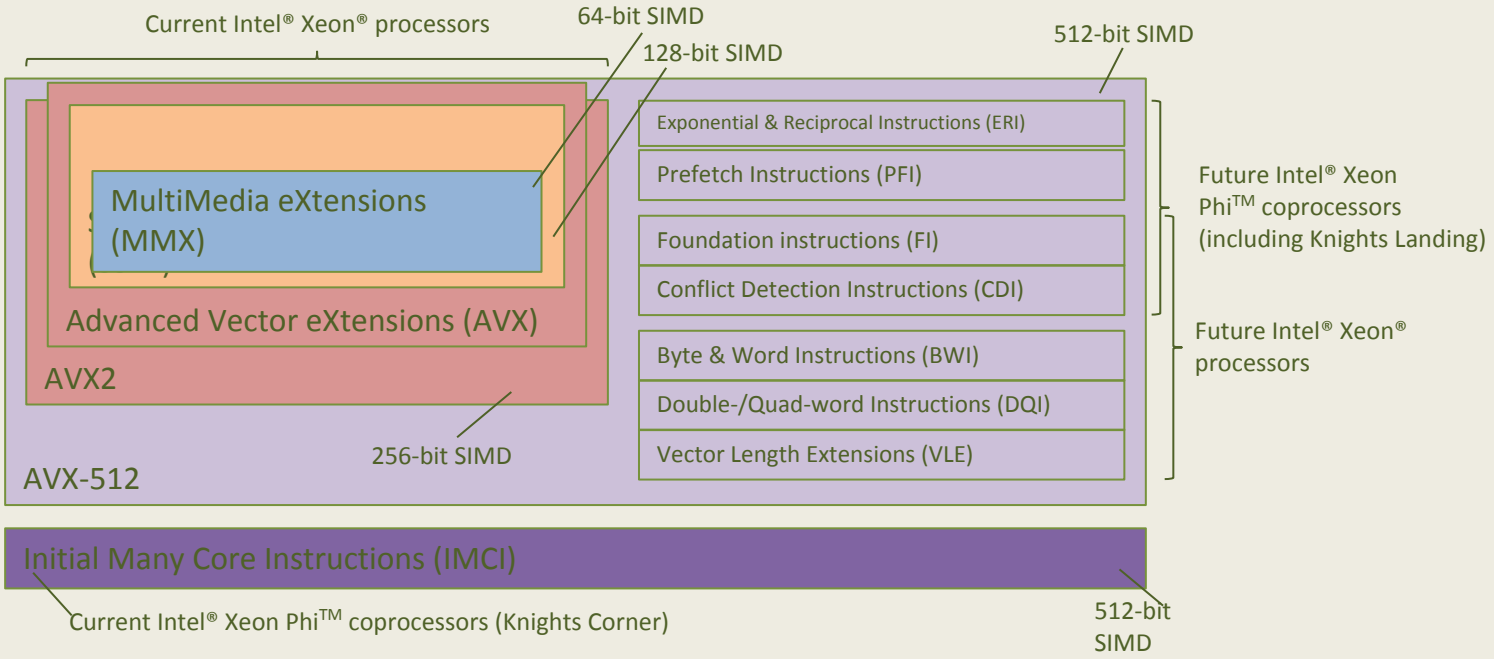
```
for (int i = 0; i < N; i++)
    c[i] = a[i] + b[i];
```

SIMD loop (4 elements)

```
for (int i = 0; i < N; i += 4)
    c[i:4] = a[i:4] + b[i:4];
```



Past, present, and future of Intel SIMD types



For additional information about Intel® AVX-512 instructions, check out James Reinders' initial and updated post for this topic.

Intel® AVX2/IMCI/AVX-512 differences

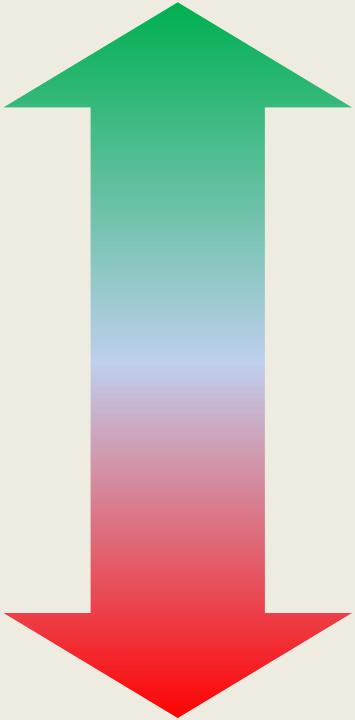
	Intel® Initial Many Core Instructions **IMCI**	Intel® Advanced Vector Extensions 2 **AVX2**	Intel® Advanced Vector Extensions 512 **AVX-512**
Introduction	2012	2013	2015
Products	Knights Corner	Haswell, Broadwell	Knights Landing, future Intel® Xeon® and Xeon® Phi™ products
Register file	SP/DP/int32/int64 data types 32 x 512-bit SIMD registers 8 x 16-bit mask registers	SP/DP/int32/int64 data types 16 x 256-bit SIMD registers No mask registers (instr. blending)	SP/DP/int32/int64 data types 32 x 512-bit SIMD registers 8 x (up to) 64-bit mask
ISA features	Not compatible with AVX*/SSE* No unaligned data support Embedded broadcast/cvt/swizzle MVEX encoding	Fully compatible with AVX/SSE* Unaligned data support (penalty) VEX encoding	Fully compatible with AVX*/SSE* Unaligned data support (penalty) Embedded broadcast/rounding EVEX encoding
Instruction features	Fused multiply-and-add (FMA) Partial gather/scatter Transcendental support	Fused multiply-and-add (FMA) Full gather	Fused multiply-and-add (FMA) Full gather/scatter Transcendental support (ERI only) Conflict detection instructions PFI/BWI/DQI/VLE (if applies)

Intel® AVX-512 is a major step in unifying the instruction set of Intel® MIC and Intel® Xeon® architecture

Vectorization on Intel® compilers

Auto Vectorization	•Compiler knobs
Guided Vectorization	•Compiler hints/pragmas •Array notation
Low level Vectorization	•C/C++ vector classes •Intrinsics/Assembly

Easy of use



Fine control

Auto vectorization

Relies on the compiler for vectorization

- No source code changes
- Enabled with `-vec` compiler knob (default in `-O2` and `-O3` modes)

Option	Description
`-O0`	Disables all optimizations.
`-O1`	Enables optimizations for speed which are know to not cause code size increase.
`-O2/-O` (default)	Enables intra-file interprocedural optimizations for speed, including: • **Vectorization** • **Loop unrolling**
`-O3`	**Performs O2 optimizations and enables more aggressive loop transformations such as:** • **Loop fusion** • **Block unroll-and-jam** • **Collapsing IF statements** This option is recommended for applications that have loops that heavily use floating-point calculations and process large data sets. However, it might incur in slower code, numerical stability issues, and compilation time increase.

Compiler smart enough to apply loop transformations

- It will allow to vectorize more loops

Vectorization: target architecture options

On which architecture do we want to run our program?

Option	Description
`-mmic`	Builds an application that runs natively on Intel® MIC Architecture.
`-xfeature` `-xHost`	Tells the compiler which processor features it may target, referring to which instruction sets and optimizations it may generate (not available for Intel® Xeon Phi™ architecture). Values for *feature* are: • **`COMMON-AVX512` (includes AVX512 FI and CDI instructions)** • **`MIC-AVX512` (includes AVX512 FI, CDI, PFI, and ERI instructions)** • **`CORE-AVX512` (includes AVX512 FI, CDI, BWI, DQI, and VLE instructions)** • **`CORE-AVX2`** • `CORE-AVX-I` (including RDRND instruction) • `AVX` • `SSE4.2, SSE4.1` • `ATOM_SSE4.2, ATOM_SSSE3` (including MOVBE instruction) • `SSSE3, SSE3, SSE2` **When using `-xHost`, the compiler will generate instructions for the highest instruction set available on the compilation host processor.**
`-axfeature`	Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel® processors if there is a performance benefit. Values for *feature* are the same described for `-xfeature` option. Multiple features/paths possible, e.g.: `-axSSE2,AVX`. It also generates a baseline code path for the default case.

Auto vectorization: not all loops will vectorize

Data dependencies between iterations
- Proven Read-after-Write data (i.e., loop carried) dependencies
- Assumed data dependencies
  - Aggressive optimizations (e.g., IPO) might help

Vectorization won't be efficient
- Compiler estimates how better the vectorized version will be
- Affected by data alignment, data layout, etc.

Unsupported loop structure
- While-loop, for-loop with unknown number of iterations
- Complex loops, unsupported data types, etc.
- (Some) function calls within loop bodies
  - Not the case for SVML functions

RaW dependency

```
for (int i = 0; i < N; i++)
    a[i] = a[i-1] + b[i];
```

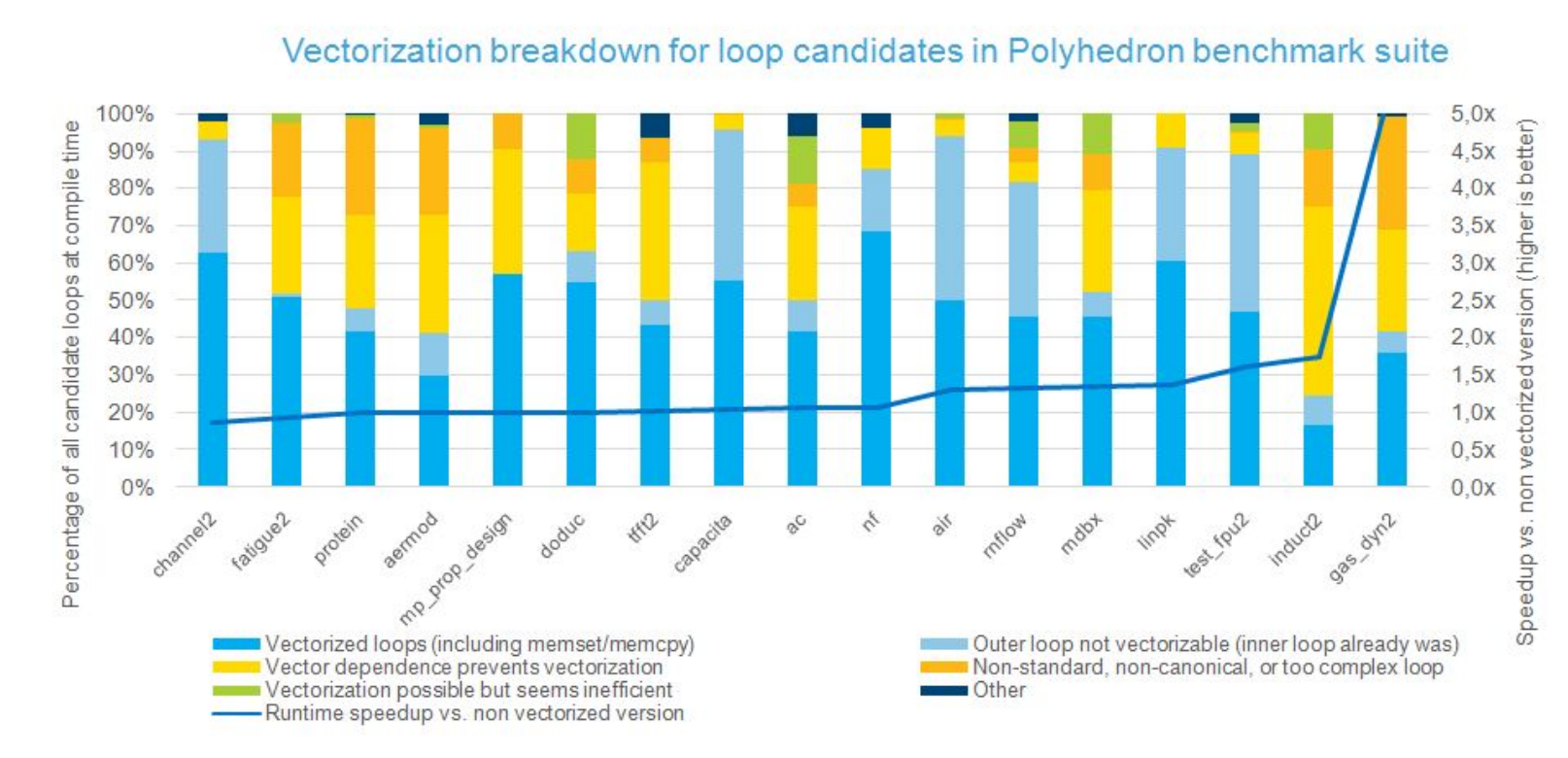
Inefficient vectorization

```
for (int i = 0; i < N; i++)
    a[c[i]] = b[d[i]];
```

Function call within loop body

```
for (int i = 0; i < N; i++)
    a[i] = foo(b[i]);
```

Auto vectorization on Intel® compilers



Validating vectorization success

Generate compiler report about optimizations

`-qopt-report[=n]`	Generate report (level [1..5], default 2)
`-qopt-report-file=<fname>`	Optimization report file (stderr, stdout also valid)
`-qopt-report-phase=<phase>`	Info about opt. phase:

```
LOOP BEGIN at gas_dyn2.f90(193,11) inlined into
gas_dyn2.f90(4326,31)
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 53
   remark #15477: vector loop cost: 14.870
   remark #15478: estimated potential speedup: 2.520
   remark #15479: lightweight vector operations: 19
   remark #15481: heavy-overhead vector operations: 1
   remark #15488: --- end vector loop cost summary ---
   remark #25456: Number of Array Refs Scalar Replaced In Loop: 1
   remark #25015: Estimate of max trip count of loop=4
LOOP END
```

Vectorized loop

`loop`	Loop nest optimizations
`par`	Auto-parallelization
`vec`	**Vectorization**
`openmp`	OpenMP
`offload`	Offload
`ipo`	Interprocedural optimizations
`pgo`	Profile Guided optimizations
`cg`	Code generation optimizations
`tcollect`	Trace analyzer (MPI) collection
`all`	All optimizations (default)

```
LOOP BEGIN at gas_dyn2.f90(2346,15)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed OUTPUT dependence between IOLD line 376 and IOLD line
354
   remark #25015: Estimate of max trip count of loop=3000001
LOOP END
```

Non-vectorized loop

Guided vectorization: disambiguation hints

Get rid of assumed vector dependencies

Assume function arguments won't be aliased

- C/C++: Compile with `-fargument-noalias`

C99 "restrict" keyword for pointers

- Compile with `-restrict` otherwise

```
void v_add(float *restrict c,
           float *restrict a,
           float *restrict b)
{
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

Ignore assumed vector dependencies (compiler directive)

- C/C++: `#pragma ivdep`
- Fortran: `!dir$ ivdep`

```
void v_add(float *c, float *a, float *b)
{
#pragma ivdep
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

Some Intel® compiler directives

Directive	Description
`distribute, distribute_point`	Instructs the compiler to prefer loop distribution at the location indicated.
`inline`	Instructs the compiler to inline the calls in question.
`ivdep`	Instructs the compiler to ignore assumed vector dependencies.
`loop_count`	Indicates the loop count is likely to be an integer.
`optimization_level`	Enables control of optimization for a specific function.
`parallel/noparallel`	Facilitates auto-parallelization of an immediately following loop; using keyword `always` forces the compiler to auto-parallelize; `noparallel` pragma prevents auto-parallelization.
`[no]unroll`	Instructs the compiler the number of times to unroll/not to unroll a loop
`[no]unroll_and_jam`	Prevents or instructs the compiler to partially unroll higher loops and jam the resulting loops back together.
`unused`	Describes variables that are unused (warnings not generated).
`[no]vector`	Specifies whether the loop should be vectorised. In case of forcing vectorization that should be according to the given clauses.

Guided vectorization: #pragma simd

Force loop vectorization ignoring **all** dependencies

- Additional clauses for specify reductions, etc.

SIMD loop

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

SIMD function

```
__declspec(vector)
void v_add(float c, float a, float b)
{
    c = a + b;
}
    …
for (int i = 0; i < N; i++)
    v_add(C[i], A[i], B[i]);
```

Guided vectorization: `#pragma simd`

Also supported in OpenMP

- Almost same functionality/syntax
  - Use `#pragma omp simd [clauses]` for SIMD loops
  - Use `#pragma omp declare simd [clauses]` for SIMD functions
- See OpenMP 4.0 specification for more information

Explicit vectorization with array notation

Express high-level vector parallel array operations

- Valid notation in Fortran since Fortran 90
- Supported in C/C++ by Intel® compiler (Cilk™ Plus) and GCC 4.9
  - Enabled by default on Intel® compiler, use `-fcilkplus` option on GCC
- No additional modifications to source code
- Most arithmetic and logic operations already overloaded
- Also built-in reducers for array sections

Vectorization becomes explicit

- C/C++ syntax: *array-expression[lower-bound:length[:stride]]*

Samples

```
a[:]       // All elements
a[2:6]     // Elements 2 to 7
a[:][5]    // Column 5
a[0:3:2]   // Elements 0,2,4
```

SIMD function invoked with array notation

```
__declspec(vector)
void v_add(float c, float a, float b)
{
    c = a + b;
}
    …
v_add(C[:], A[:], B[:]);
```

Improving vectorization: data layout

Vectorization more efficient with unit strides

- Non-unit strides will generate gather/scatter
- Unit strides also better for data locality
- Compiler might refuse to vectorize

AoS vs SoA

- Layout your data as Structure of Arrays (SoA)

Traverse matrices in the right direction

- C/C++: `a[i][:]`, Fortran: `a(:,i)`
- Loop interchange might help
  - Usually the compiler is smart enough to apply it
  - Check compiler optimization report

Array of Structures vs Structure of Arrays

```
// Array of Structures (AoS)
struct coordinate {
    float x, y, z;
} crd[N];
    …
for (int i = 0; i < N; i++)
    … = … f(crd[i].x, crd[i],y, crd[i].z);
```

Consecutive elements in memory →

x0 y0 z0 x1 y1 z1 … x(n-1) y(n-1) z(n-1)

```
// Structure of Arrays (SoA)
struct coordinate {
    float x[N], y[N], z[N];
} crd;
    …
for (int i = 0; i < N; i++)
    … = … f(crd.x[i], crd.y[i], crd.z[i]);
```

Consecutive elements in memory →

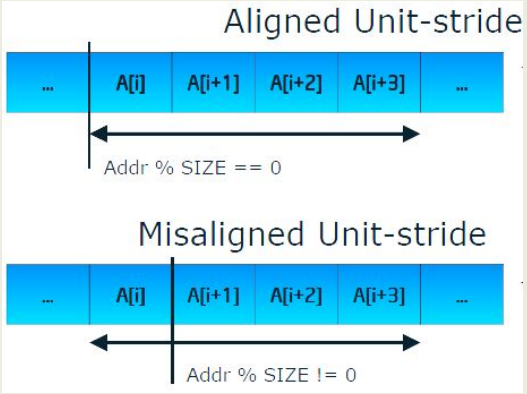
x0 x1 … x(n-1) y0 y1 … y(n-1) z0 z1 … z(n-1)

Improving vectorization: data alignment

Unaligned accesses might cause significant performance degradation
- Two instructions on current Intel® Xeon Phi™ coprocessor
- Might cause “false sharing” problems
  - Consumer/producer thread on the same cache line

Alignment is generally unknown at compile time
- Every vector access is potentially an unaligned access
  - Vector access size = cache line size (64-byte)
- Compiler might “peel” a few loop iterations
  - In general, only one array can be aligned, though

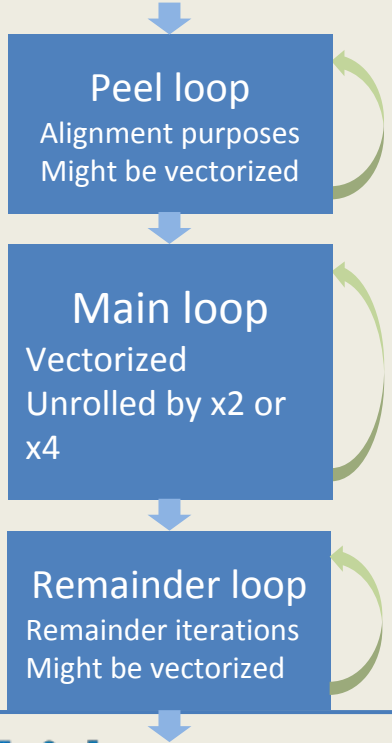
When possible, we have to
- Align our data
- Tell the compiler data is aligned
  - Might not be always the case



Improving vectorization: data alignment

How to…	Language	Syntax	Semantics
…align data	C/C++	`void* _mm_malloc(int size, int n)`	Allocate memory on heap aligned to *n* byte boundary.
	C/C++	`int posix_memalign (void **p, size_t n, size_t size)`	
	C/C++	`__declspec(align(n)) array`	Alignment for variable declarations.
	Fortran (not in common section)	`!dir$ attributes align:n::array`	
	Fortran (compiler option)	`-alignnbyte`	
…tell the compiler about it	C/C++	`#pragma vector aligned`	Vectorize assuming all array data accessed are aligned (may cause fault otherwise).
	Fortran	`!dir$ vector aligned`	
	C/C++	`__assume_aligned(array, n)`	Compiler may assume array is aligned to *n* byte boundary.
	Fortran	`!dir$ assume_aligned array:n`	

Vectorization with multi-version loops




```
LOOP BEGIN at gas_dyn2.f90(2330,26)
<Peeled>
   remark #15389: vectorization support: reference AMAC1U has unaligned access
   remark #15381: vectorization support: unaligned access used inside loop
body
   remark #15301: PEEL LOOP WAS VECTORIZED
LOOP END
LOOP BEGIN at gas_dyn2.f90(2330,26)
   remark #25084: Preprocess Loopnests: Moving Out Store
   remark #15388: vectorization support: reference AMAC1U has aligned access
   remark #15399: vectorization support: unroll factor set to 2
   remark #15300: LOOP WAS VECTORIZED
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 8
   remark #15477: vector loop cost: 0.620
   remark #15478: estimated potential speedup: 15.890
   remark #15479: lightweight vector operations: 5
   remark #15488: --- end vector loop cost summary ---
   remark #25018: Total number of lines prefetched=4
   remark #25019: Number of spatial prefetches=4, dist=8
   remark #25021: Number of initial-value prefetches=6
LOOP END
LOOP BEGIN at gas_dyn2.f90(2330,26)
<Remainder>
   remark #15388: vectorization support: reference AMAC1U has aligned access
   remark #15388: vectorization support: reference AMAC1U has aligned access
   remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
```

Other considerations

Loop tiling/blocking to improve data locality

- Square tiles so elements can be reused

Use streaming loads/stores to save bandwidth

- `#pragma vector [non]temporal(list)`
- `-qopt-streaming-stores=[always|never|auto]`
- `-qopt-streaming-cache-evict[=n]` (Intel® MIC only)

Tune software prefetcher

- `-qopt-prefetch[=n]`
- `-qprefetch-distance=n1[,n2]` (Intel® MIC only)
- `#pragma [no]prefetch [clauses]` (Intel® MIC only)

Low level (explicit) vectorization

A.k.a “ninja programming”

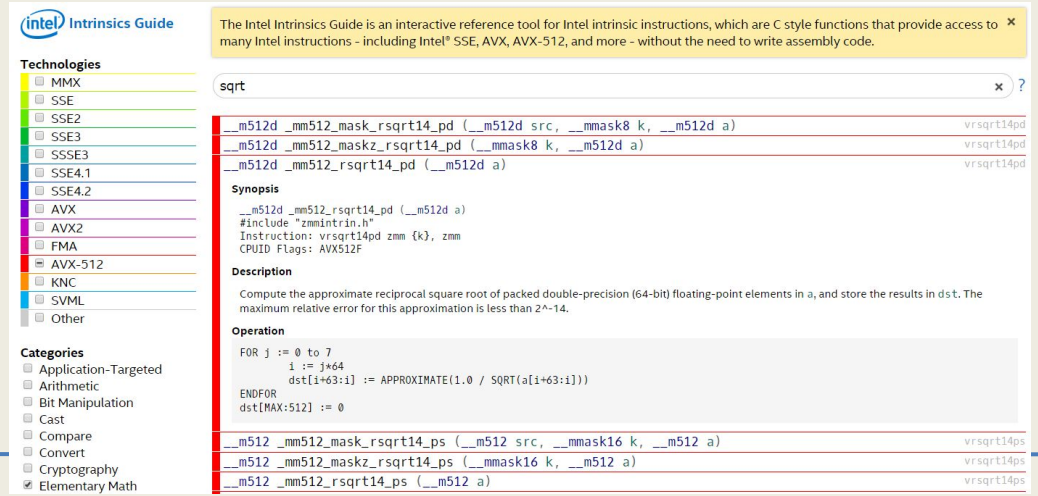
Vectorization relies on the programmer with some help from the compiler

Might be convenient for low level performance tuning of critical hotspots

Not portable among different SIMD architectures

SIMD C++ class	Intrinsics	Assembly

`#include <fvec.h>`
 	
`F32vec4 a,b,c;`
`a = b + c;`
	`#include <xmmintrin.h>`
	
	`__m128 a,b,c;`
	`a = _mm_add_ps(b,c);`
		`__m128 a,b,c;`
		`__asm {`
		`  movaps xmm0,b`
		`  movaps xmm1,c`
		`  addps xmm0,xmm1`
		`  movaps a, xmm0`
		`}`



How to get ready for Intel® AVX-512?

BKM: Start optimizing your application today for current generation of Intel® Xeon® processors and Intel® Xeon™ Phi coprocessors

Tune your AVX-512 kernels on non-existing silicon

- Compile with latest compiler toolchains
  - Intel® compiler (v15.0): `-xCOMMON-AVX512`, `-xMIC-AVX512`, `-xCORE-AVX512`
  - GNU compiler (v4.9): `-mavx512f`, `-mavx512cd`, `-mavx512er`, `-mavx512pf`
- Run Intel® Software Development emulator (SDE)
  - Emulate (future) Intel® Architecture Instruction Set Extensions (e.g. Intel® MPX, …)
  - Tools available for detailed analysis
    - Instruction type histogram
    - Pointer/misalignment checker
  - Also possible to debug the application while emulated

Summary

Programmers are mostly responsible of exposing DLP (SIMD) parallelism

Intel® compilers provide sophisticated/flexible support for vectorization

- Auto, guided (assisted), and low-level (explicit) vectorization
- Based on OpenMP standards and specific directives
- Easily portable across different Intel® SIMD architectures

Fine-tuning of generated code is key to achieve the best performance

- Check whether code is actually vectorized
- Data layout, alignment, remainder loops, etc.

Get ready for Intel® AVX-512 by optimizing your application today on current generation of Intel® Xeon® processors and Intel® Xeon™ Phi coprocessors

Online resources

Intel® Xeon Phi™

- Developer portal — Programming guides, tools, trainings, case studies, etc.
- Solutions catalog — Existing Intel® Xeon Phi™ solutions for known codes

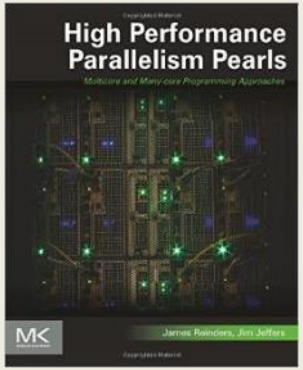
Intel® software development tools, performance tuning, etc.

- Documentation library — All available documentation about Intel software
- Learning lab — Learning material with Intel® Parallel Studio XE
- Performance — Resources about performance tuning on Intel hardware
- Forums — Public discussions about Intel SIMD, threading, ISAs, etc.

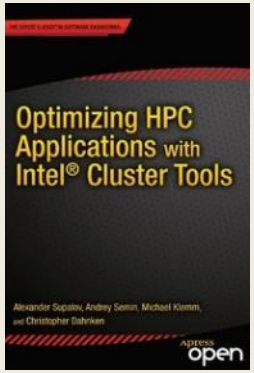
Other resources (white papers, benchmarks, case studies, etc.)

- Go parallel — BKMs for Intel multi- and many-core architectures
- Colfax research — Publications and material on parallel programming
- Bayncore labs — Research and development activities (WIP)

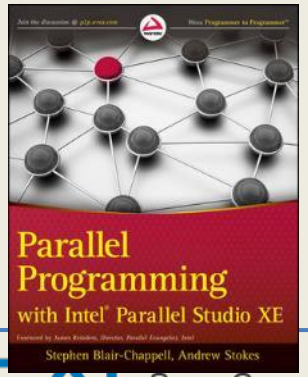
Recommended books



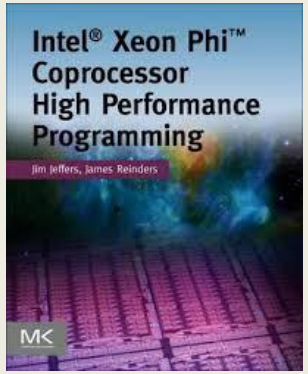
High performance parallelism pearls: multi-core and many-core approaches, by James Reinders and Jim Jeffers, Morgan Kaufmann, 2014



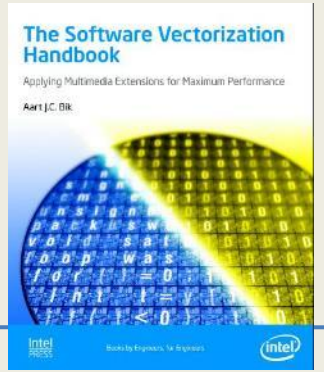
Optimizing HPC applications with Intel® cluster tools, by Alexander Supalov et al, Apress, 2014



Parallel programming with Intel® Parallel Studio XE, by Stephen Blair-Chappell and Andrew Stokes, Wrox press, 2012



Intel® Xeon Phi™ coprocessor high-performance programming, by Jim Jeffers and James Reinders, Morgan Kaufmann, 2013

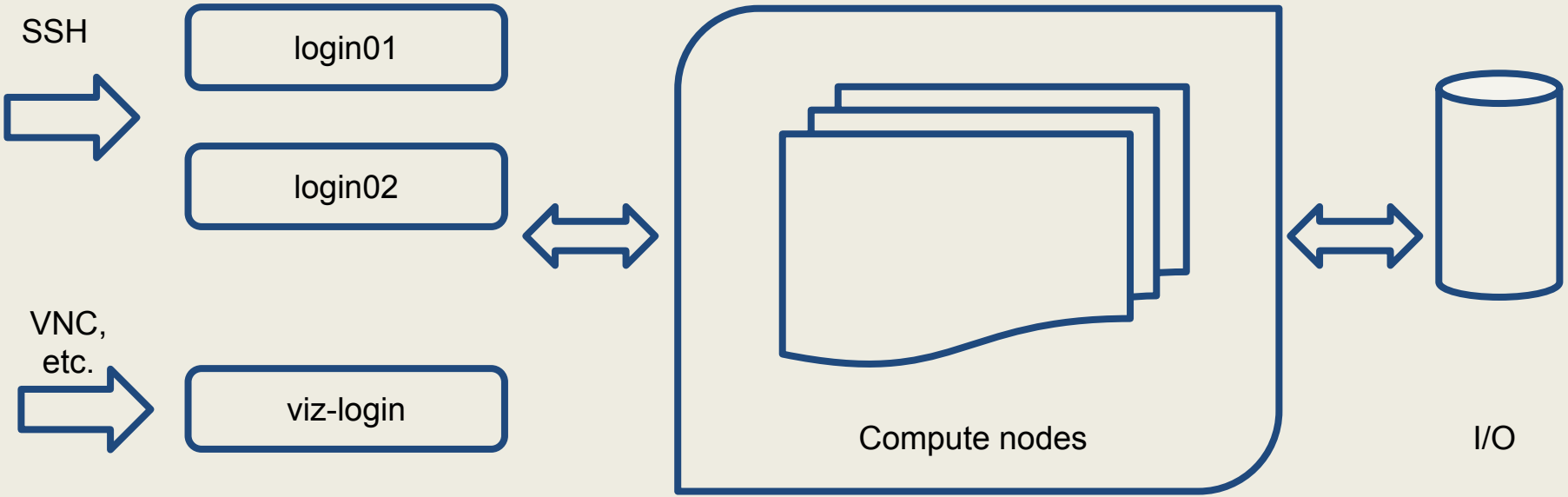


The software optimization handbook, by Aart Bik, Intel® press, 2004

Introduction to the hands-on

SuperComputing Applications and Innovation

General layout of a HPC cluster



Job scheduler

In order to permit to all users to work, a job scheduler is in charge to define the execution of the jobs.

On the Cineca HPC clusters, the job scheduler is PBS.

PBS can work

- with the submission of a script
- or in an interactive session

qsub

```
[faffinit@r000u18l02 ~]$ qsub -I -A cin_priorit -l select=1:ncpus=36:mpiprocs=36 -l walltime=15:00
qsub: waiting for job 62238.r000u17l01 to start
qsub: job 62238.r000u17l01 ready

[faffinit@r041c03s02 ~]$
```

interactive

batch 

```
[faffinit@r000u18l02 ausurf]$ cat job.marconi
#!/bin/bash
#PBS -l walltime=6:00:00
#PBS -l select=2:ncpus=16:mpiprocs=8
#PBS -o job.out
#PBS -e job.err
#PBS -A cin_staff

cd $PBS_O_WORKDIR
export OMP_NUM_THREADS=1

module load profile/phys autoload qe
mpirun -np 16 $QE_HOME/bin/pw.x -input input  > log


#module load autoload openmpi
#module load profile/advanced
#module load fftw/3.3.4--openmpi--1-10.3--gnu--6.1.0
[faffinit@r000u18l02 ausurf]$ qsub job.marconi
```

Workbench: Poisson

We will use as a workbench to test the Intel software tools a small program that simulate the solution of the Poisson equation on a grid.

Exercise: familiarize with the code, understanding (just a little bit) what is it doing, what are the computational kernels, etc.

Exercise: compile the code and run on one single node. Check the execution times changing the number of MPI processes, OpenMP threads, etc

Graphic sessions

In order to work with Intel tools, it can be necessary to work with a graphic session rather than text-only ssh.

A tool is avalaible from Cineca to work with a VNC session: RCM

You can download RCM from:
http://www.hpc.cineca.it/software/rcm

Credentials etc.

GALILEO login: login.galileo.cineca.it

usernames: a08traXX where XX=21..24

account_no = train_cintel16