Road map

Need for High Performance Computing

Case Study

Real Time Fraud Detection in PayPal

Problem: Detecting fraud in 'real time'.

Complexity: As millions of transactions are processed between disparate systems at volume

- 11 million+ PayPal logins / day.
- 13 million+ financial transactions/day
- 500 variables calculated per event for some models.
- ~4 Billion inserts / day.
- ~8 Billion selects / day

Solution

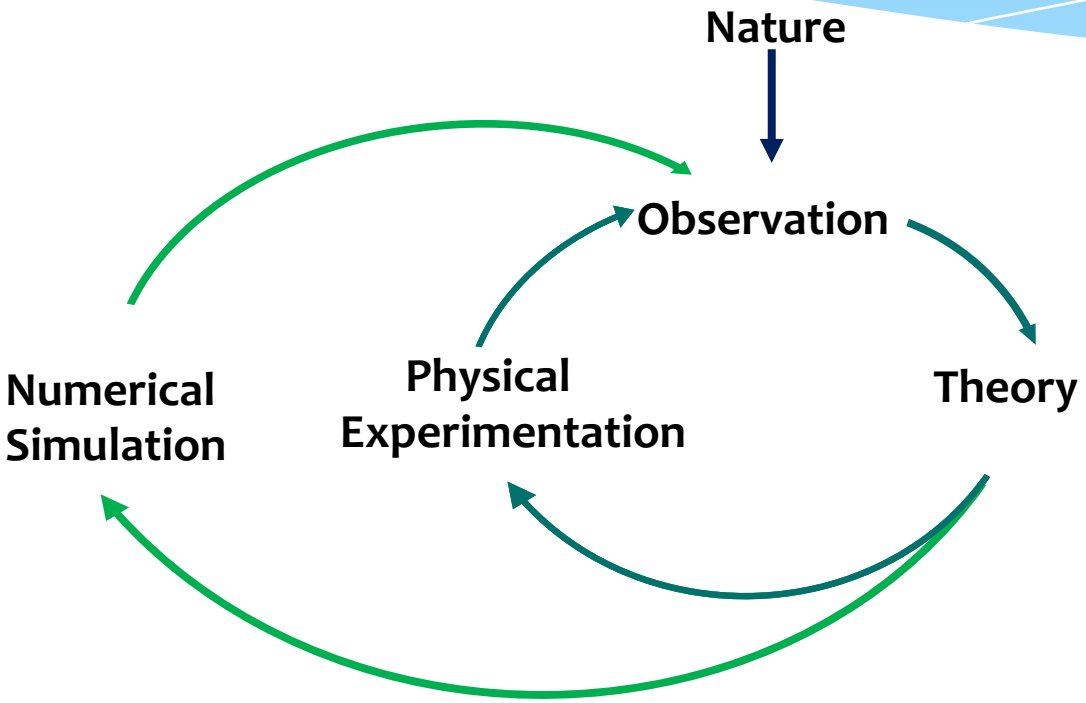
Trinity

- Build a giant graph
- Leverage HPC architecture, Hardware and Software
- Infini Band
- Very high scalability.

HPC Impact – Time & Economy

Airlines	System-wide logistics optimizations evaluated on HPC systems save approx. US$100 million per airline per year.
Automotive Design	Major companies use (500+ CPUs) in CAD and CAM for crash testing, structural integrity and aerodynamics saving over US$1 billion per company per year.
Semiconductor Industries	Semiconductor firms use large systems (500+CPUs) for device electronics simulation and logic validation saving approx. US$1 billion per company per year
Securities Industry	Savings approx. US$15 billion per year for US home mortgages.

Numerical Modelling



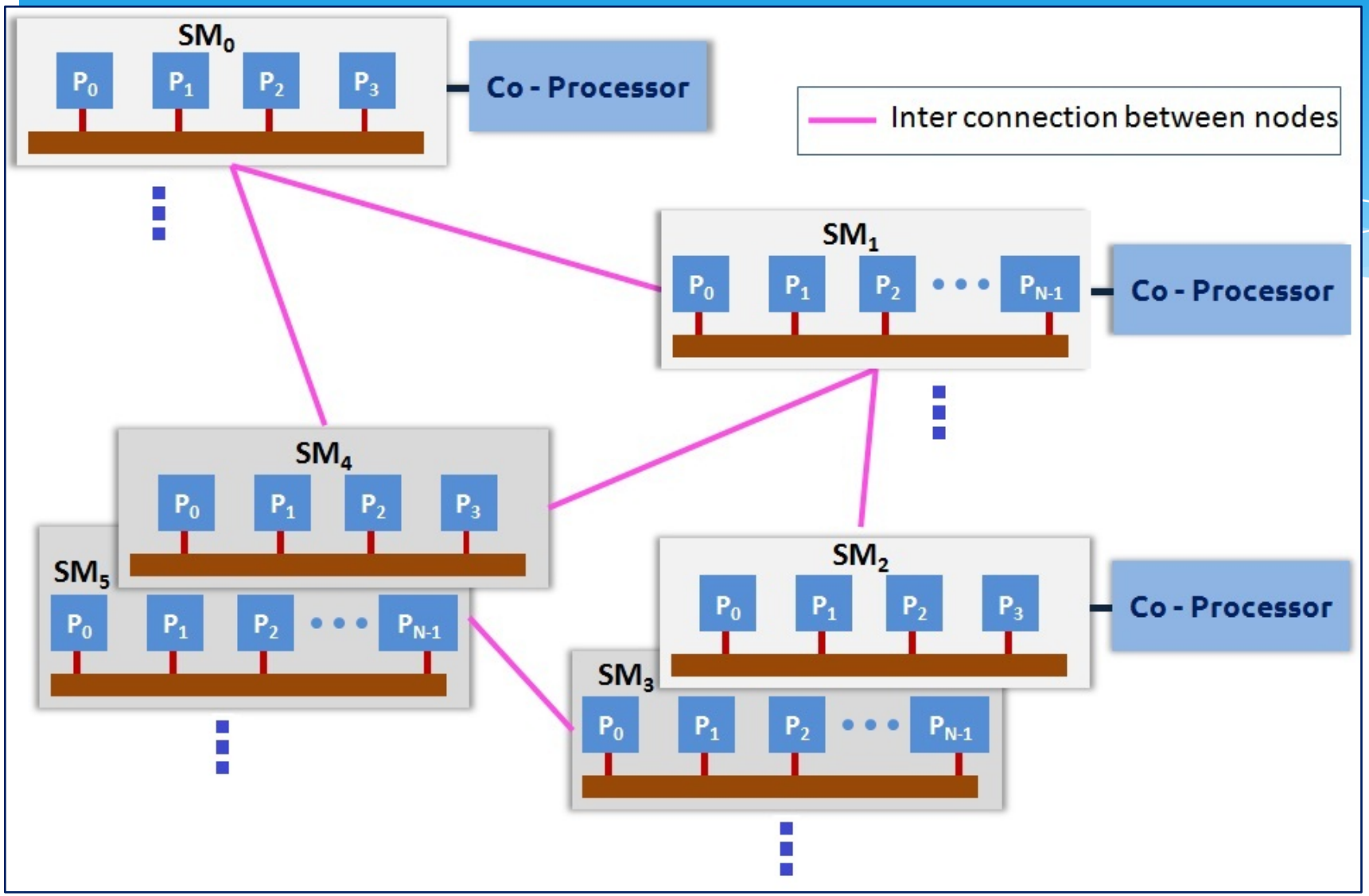
Modelling Algorithm for HPC

* The large amount of computation and data require powerful machine called High Performance Computing (HPC) to produce results in measurable time.
* HPC comprise of a large array / cluster of computing units, connected by a high bandwidth interconnect/fabric communicating to high performance I/O.
* Algorithmic application are coded to take advantage of the various Shared Memory and Distributed Memory system to provide reliability and performance.

Tianhe-2 (Milkyway-2)

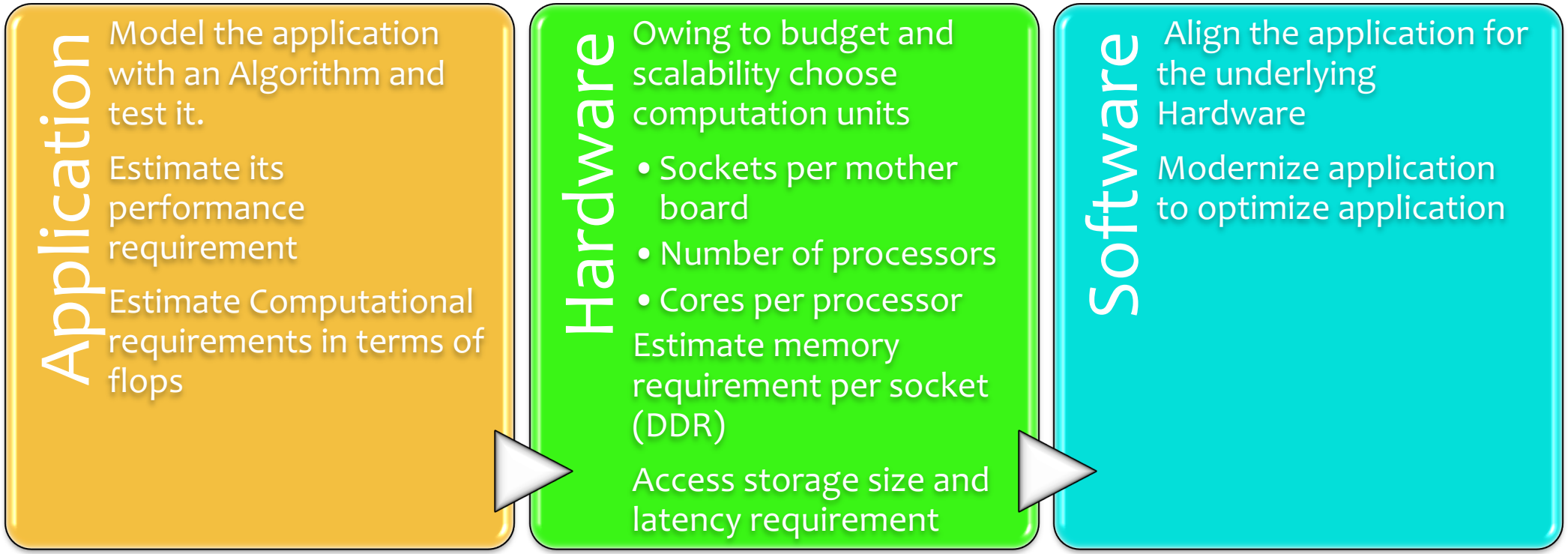
Site:	National Super Computer Center in Guangzhou
Manufacturer:	NUDT
Cores:	3,120,000
Linpack Performance (Rmax)	33,862.7 TFlop/s
Theoretical Peak (Rpeak)	54,902.4 TFlop/s
Nmax	9,960,000
Power:	17,808.00 kW
Memory:	1,024,000 GB
Processor:	Intel Xeon E5-2692v2 12C 2.2GHz (32,000)
Coprocessor:	Intel Xeon-phi 31S1P (48,000)
Interconnect:	TH Express-2
Operating System:	Kylin Linux
Compiler:	icc
Math Library:	Intel MKL-11.0.0
MPI:	MPICH2 with a customized GLEX channel

HPC Cluster

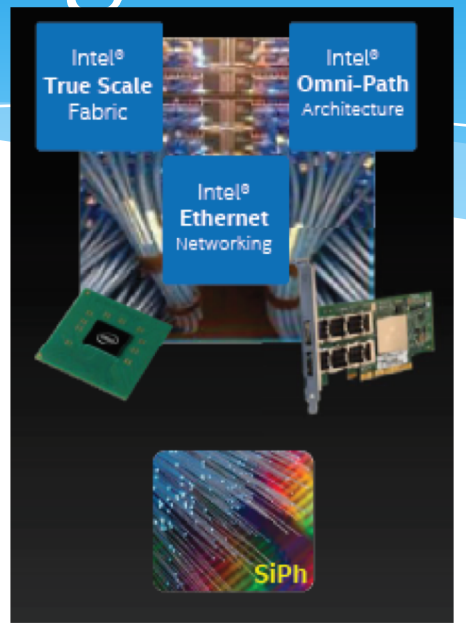


SM Shared Memory
Pn : Processor n
Co-Processor: Intel® Xeon Phi /FPGA based accelerator etc

Application Development



Components of High Performance Computing



Compute Processing

System & Boards

Network & Fabric

I/O Storage

Software & Services

Components of HPC

Compute / Processing:

- Logical circuit that responds to and processes the instruction in a program.

System & Boards

- To bind all the components of a computer Mother board, power supply, its Cooling and etc are essential.

Network & Fabric

- Connecting different nodes.
- Example: Ethernet, Infiniband and Omnipath

I/O Storage

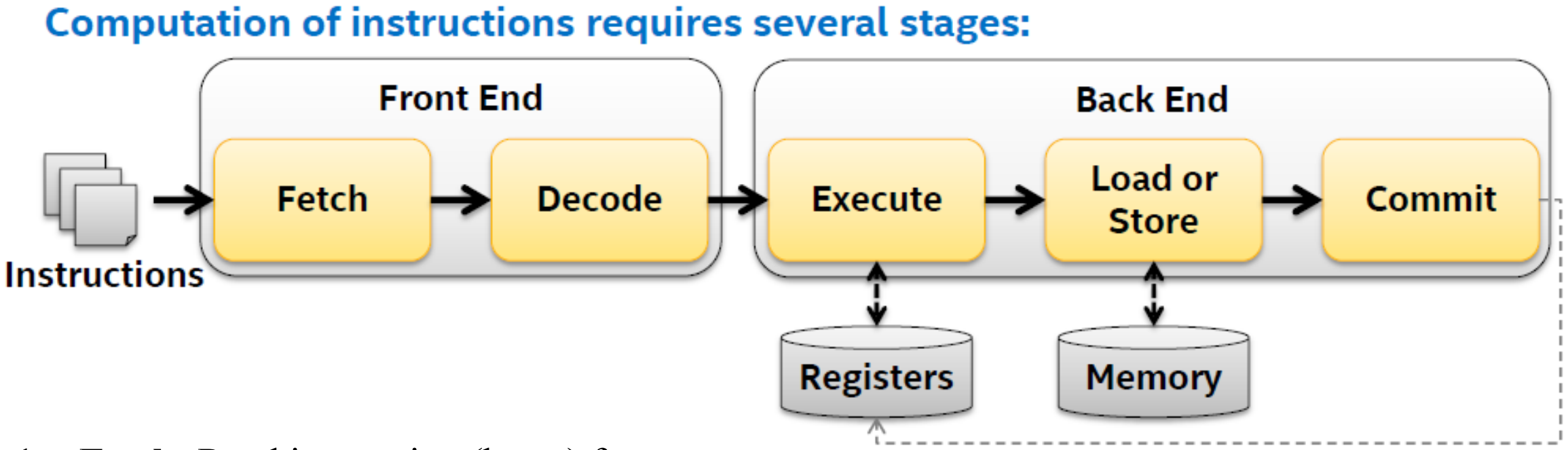
- Categorized as primary(Volatile, RAM) and secondary(Non volatile, ROM) memory.
- For higher data transfer rates which is crucial for HPC, DDR SDRAM (Double data rate synchronous dynamic random-access memory) is used.

Software & Services

- Each node has its own Operating system.
- Compiler and other tools like Intel® Parallel Studio helps us to enhance the performance

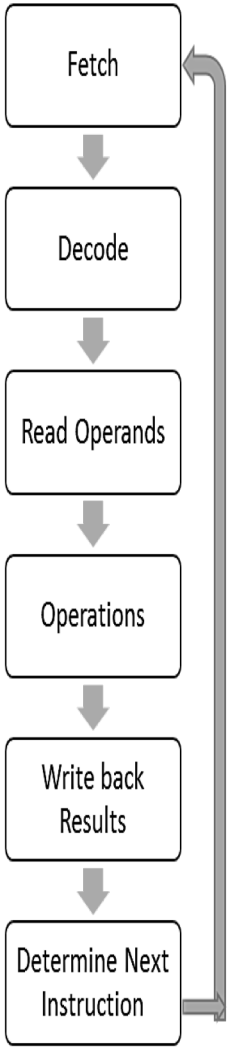
Processor Architecture Basics

Processor Architecture Basics



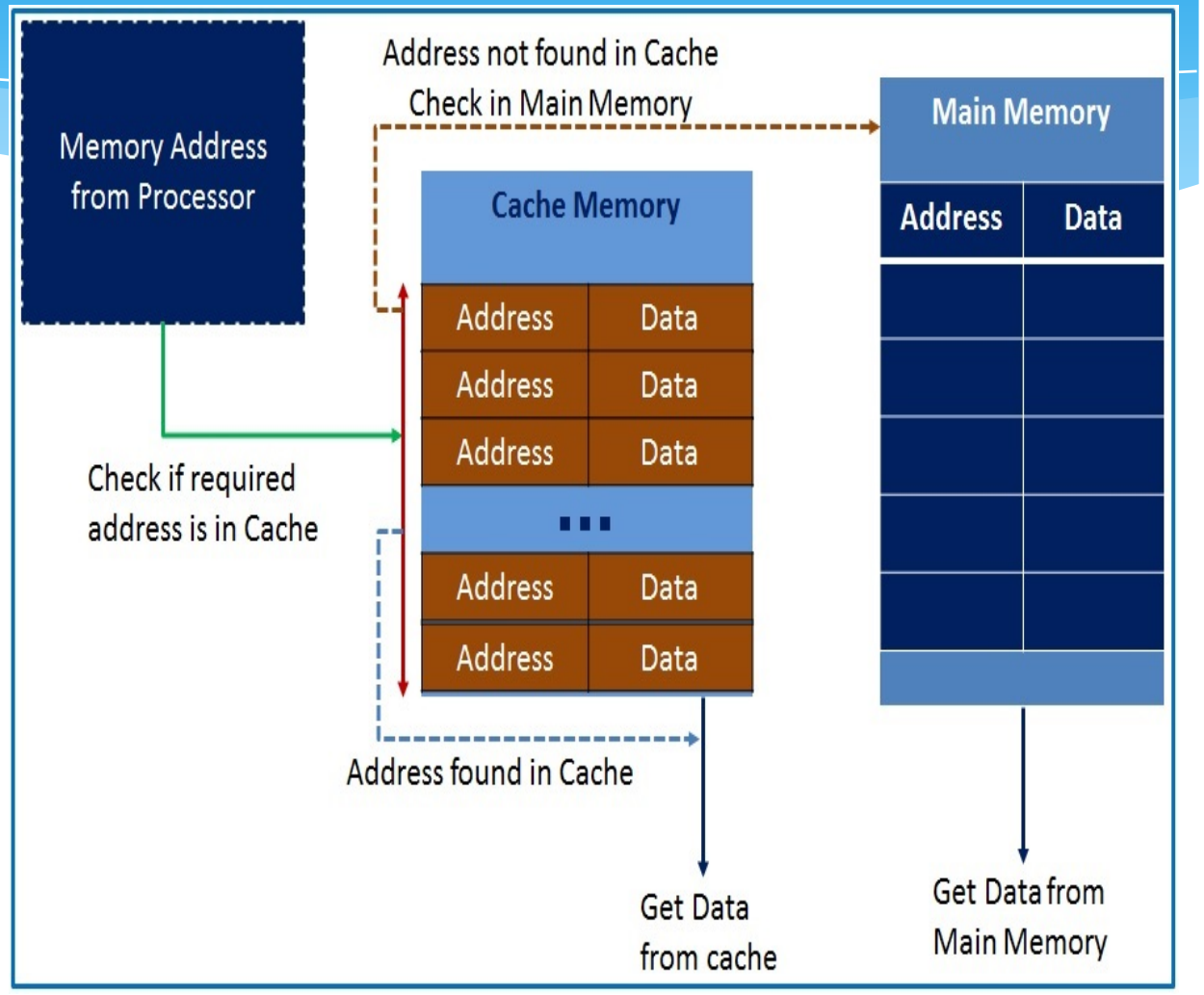
1. **Fetch:** Read instruction (bytes) from memory
2. **Decode:** Translate instruction (bytes) to microarchitecture
3. **Execute:** Perform the operation with a functional unit
4. **Memory:** Load (read) or store (write) data, if required
5. **Commit:** Retire instruction and update micro-architectural state

Instruction Set Execution



Use of cache

* Cache memory, also called as CPU memory.
* Access in cache memory is quicker than access in regular memory.
* Typically integrated directly with the CPU chip or place on a separate chip that has a separate bus inter connect with CPU.
* Used to store program instructions that are frequently re-referenced by a program. Fast access to these increases overall speed of the program.
* When processor needs to access data, it first checks the cache. If it finds, it doesn't have to do more time consuming reading of data from larger memory.
* Granularity of cache is a cache-line of 128-512 length.
* Cache coherency: The consistency of data stored in local caches of a shared resource.



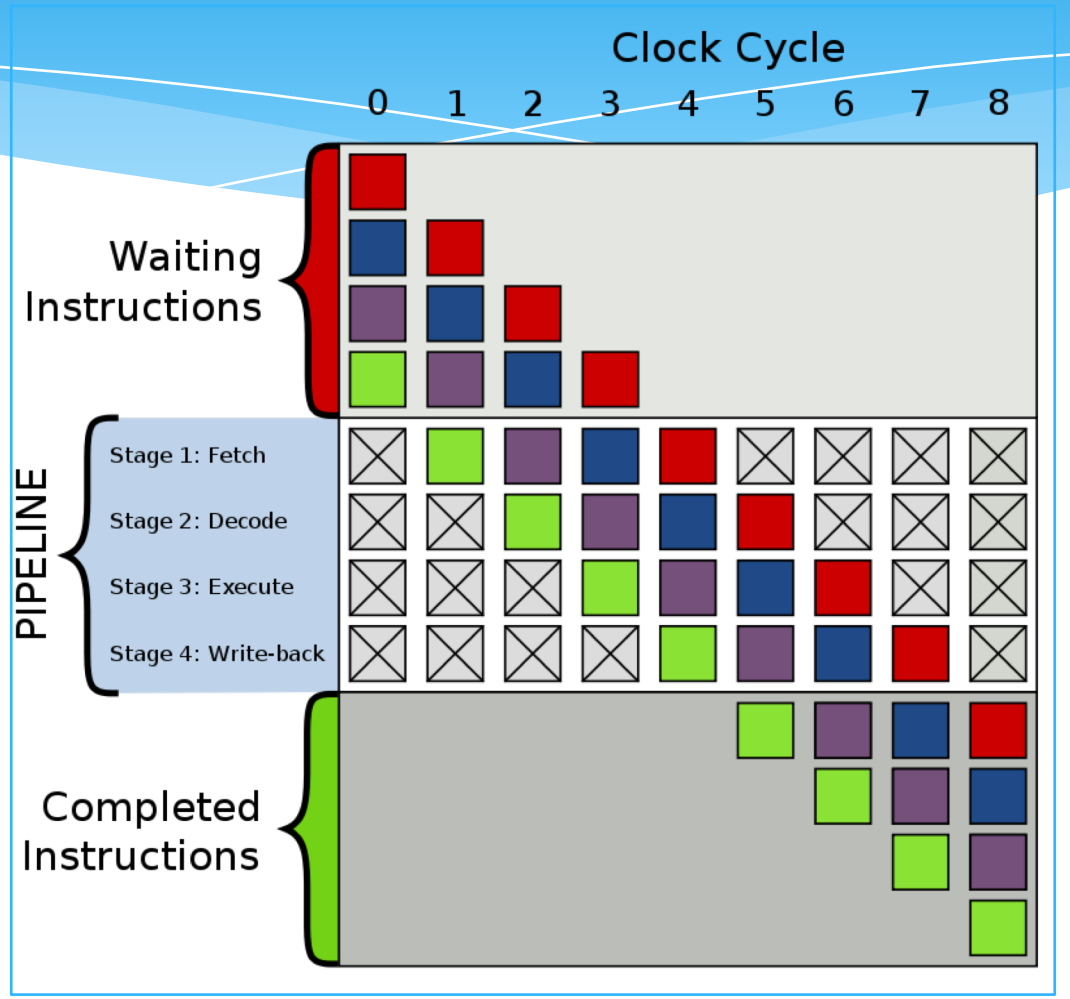
Use of cache

* L1,L2,L3 are different levels of cache.
* Level 1(L1) : Cache is extremely fast but relatively small and usually embedded in the processor chip(CPU).
* Level 2(L2) : It may be located on CPU or separate chip or coprocessor with high speed alternative bus.
* Level 3(L3): Cache is typically specialized memory that works to improve performance of L1 and L2. Significantly slower than former, but usually double the speed of RAM.
* In Multicore system, each core has dedicated L1 and L2, but share a common L3 cache.

* System performance in computers with slower processor but larger cache is better than faster processor with limited cache.

Instruction Level Parallelism

* The potential overlap among instructions is called Instruction Level Parallelism.
* Modern processors employ a technique called pipelining to increase instruction throughput.
* In a pipeline, various dedicated pieces of hardware on the processor each perform particular functions needed to process an instruction, on different instructions at the same time.



Courtesy: wikipedia

Vectorization

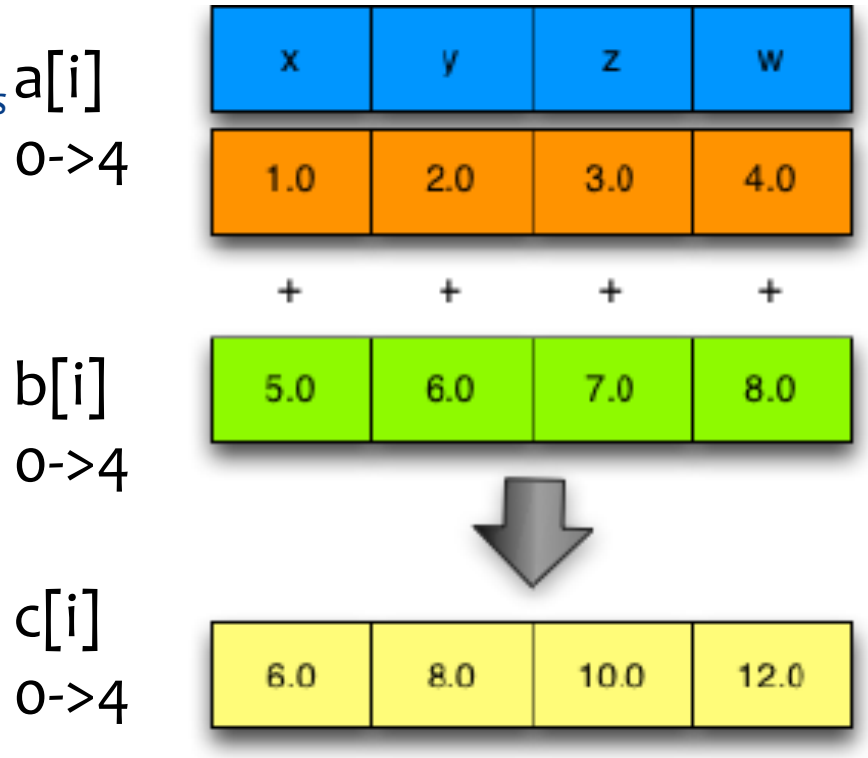
- Data Level Parallelism for single thread.
  - Operate on more than one element at a time
  - Might decrease instruction counts significantly
- Elements are stored on SIMD(Single Instruction Multiple Data) registers or *vectors*
- Code needs to be *vectorized*
  - Vectorization usually on *inner* loops
  - Main and *remainder* loops are generated

Scalar loop

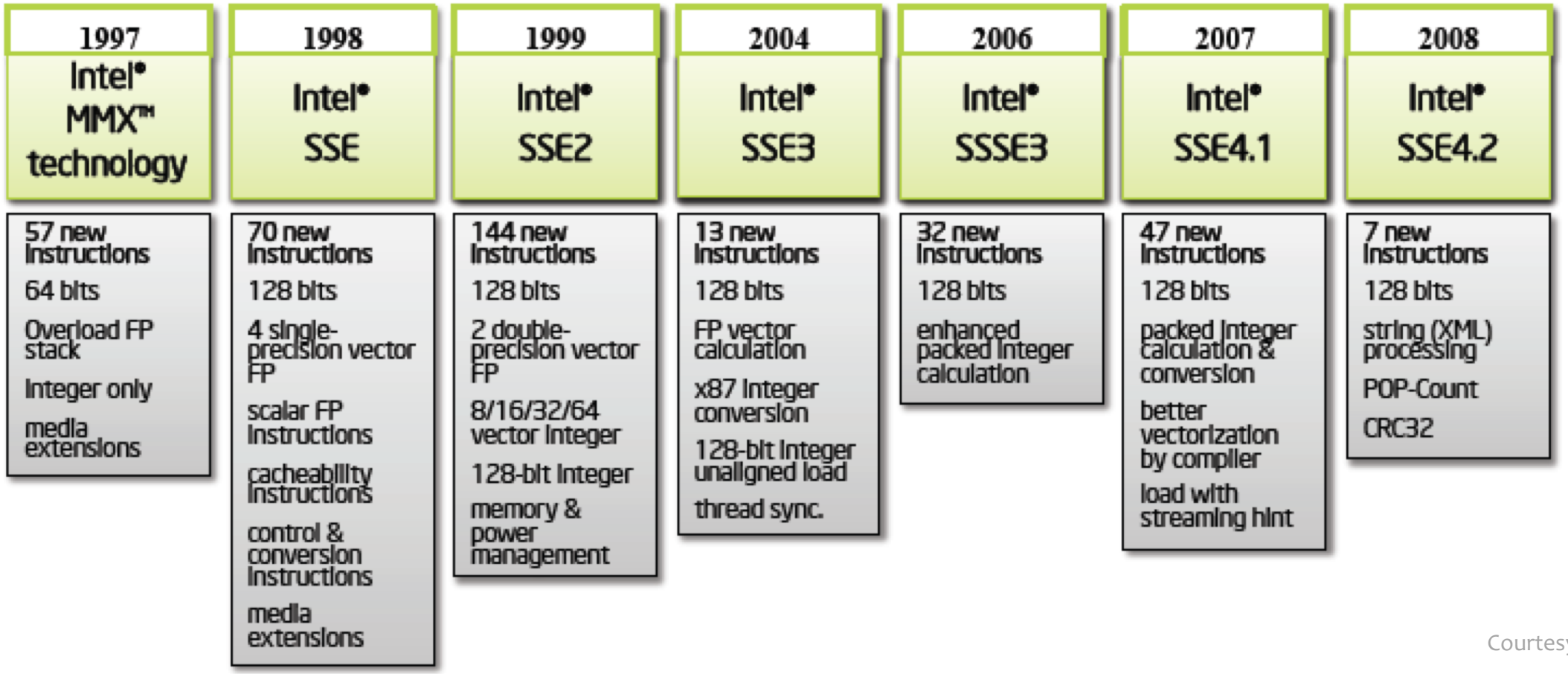
```
for (inti= 0; i< N; i++)
c[i] = a[i] + b[i];
```

Vector loop

```
for (inti= 0; i< N; i+= 4)
c[i] = a[i] + b[i];
c[i+1] = a[i+1] + b[i+1];
c[i+2] = a[i+2] + b[i+2];
c[i+3] = a[i+3] + b[i+3];
```

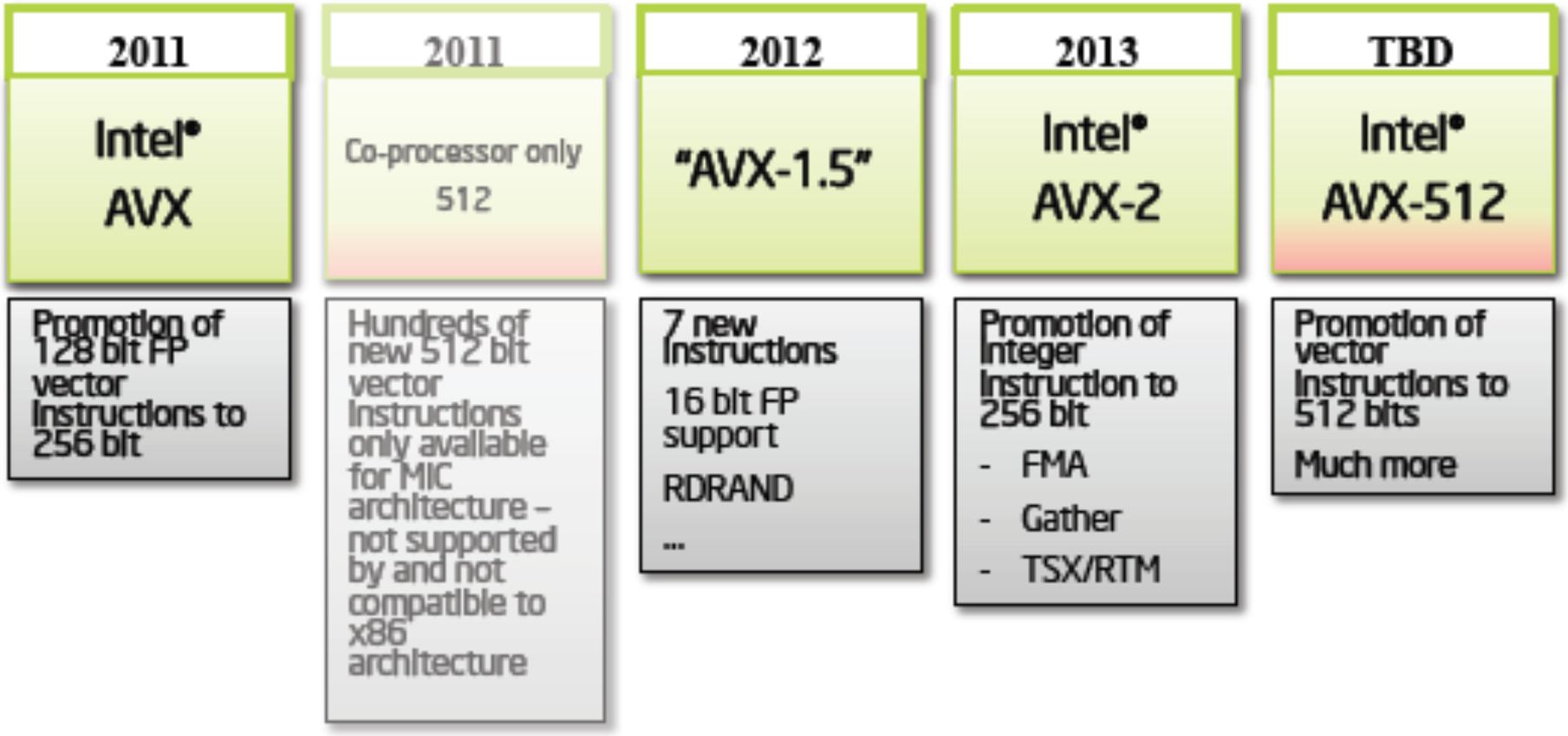


Vectorization Timeline



Courtesy: Intel® Corporation

Vectorization Timeline



Courtesy: Intel® Corporation

Branch Prediction & Hyper Threading

* A branch predictor is part of a processor that determines whether a conditional branch (jump) in the instruction flow of a program is likely to be taken or not. This is called branch prediction.

* Hyper threading:
  * Hyper-Threading Technology is a form of simultaneous multithreading technology introduced by Intel.
  * Architecturally, a processor with Hyper-Threading Technology consists of two logical processors per core, each of which has its own processor architectural state.
  * The main function of hyper-threading is to increase the number of independent instructions in the pipeline; it takes advantage of superscalar architecture, in which multiple instructions operate on separate data in parallel.

Capabilities of Performance Monitoring

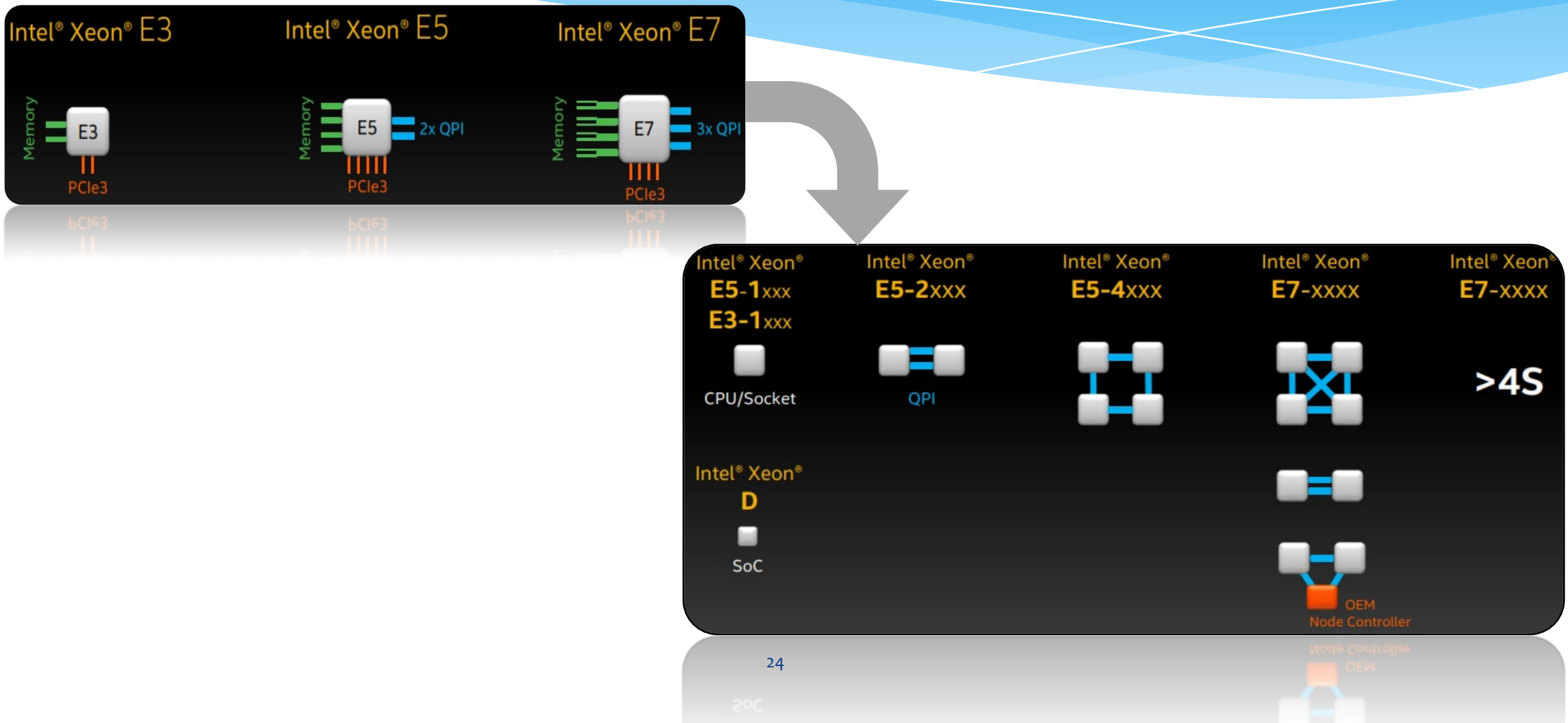
* Software that understands and dynamically adjusts to resource utilization of modern processors has performance and power advantages.
* Hierarchical cache subsystems, non-uniform memory, simultaneous multithreading and out-of-order execution have a huge impact on the performance and compute capacity.
* The Intel® Performance Counter Monitor provides sample C++ routines and utilities to estimate the internal resource utilization of the latest Intel® Xeon® and Core™ processors and gain a significant performance boost
* Unlike PAPI* and Linux* "perf" Intel® support not only core but also uncore PMU(Performance Monitoring Units) of Intel processors
  * Core: instructions retired, elapsed core clock ticks, core frequency including Intel® Turbo boost technology, L2 cache hits and misses, L3 cache misses and hits (including or excluding snoops).
  * Uncore: read bytes from memory controller(s), bytes written to memory controller(s), data traffic transferred by the Intel® QuickPath Interconnect links.

Intel® Xeon Architecture

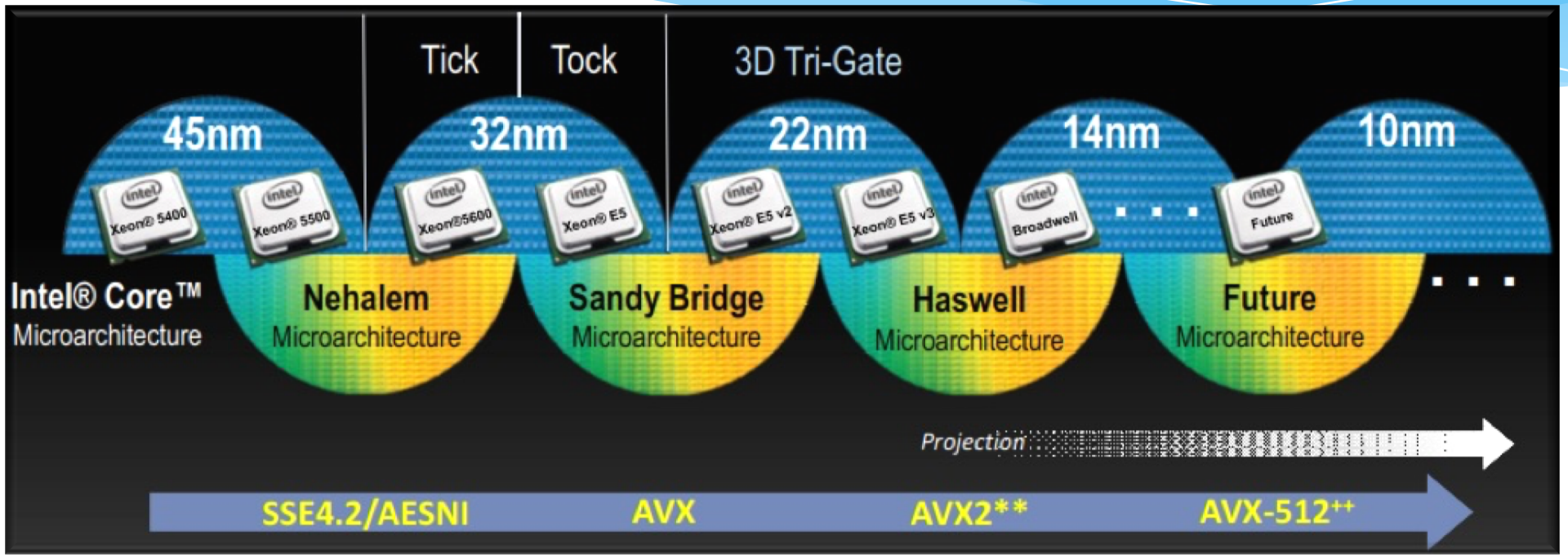
- Xeon is a brand name x86 (and x86-64) Microprocessor manufactured by Intel for the server market.
- Current day Xeon enables creation of Multi-core, Multi-Socket clusters, with ECC memory support for high reliability
- First dual-core Xeon was introduced in 2005. Paxwell DP (2 Cores, 4MB L2 Cache, 1.4MHz) and 7000-Series Paxville MP (2 Cores, 2MB L2 Cache, 2.8GHz)

Note: *DP Implies Dual Processor, can use single or double socket on a mother board. MP Implies Multi Processor, providing room to build mother boards with 4 or more sockets.*

Intel® Xeon Processor family and Platform



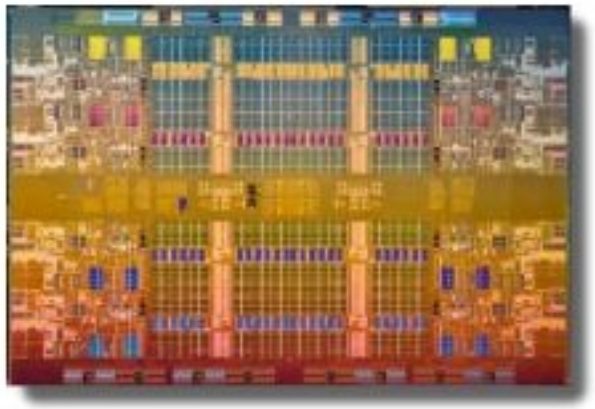
Intel® Tick Tock Model



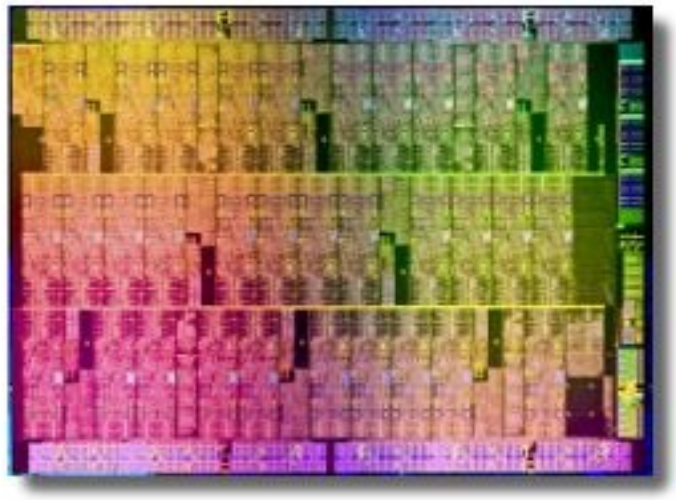
Intel® designs processor on two grounds, one is process technology and the other is microarchitecture.

Intel® Xeon Phi Architecture

- Intel® Xeon Phi is a MIC (Many Integrated Core) architecture
- Designed to be a coprocessor to supplement the Xeon system
- Each core includes a large vector register, vector unit and dedicated cache



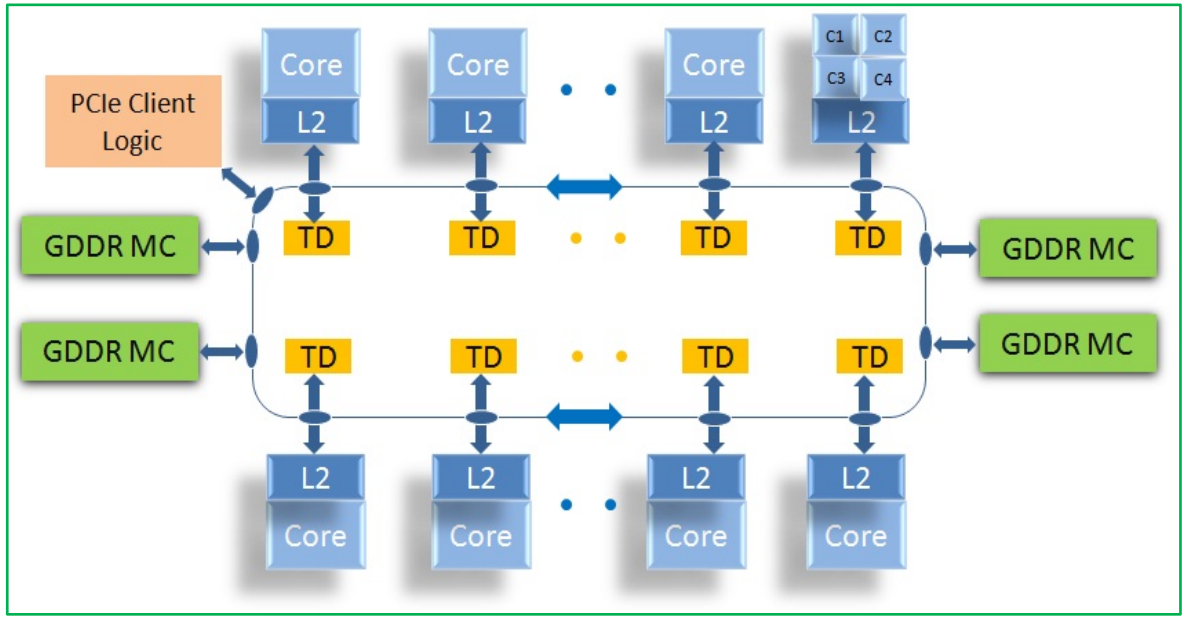
Multi-Core Intel Xeon Processor



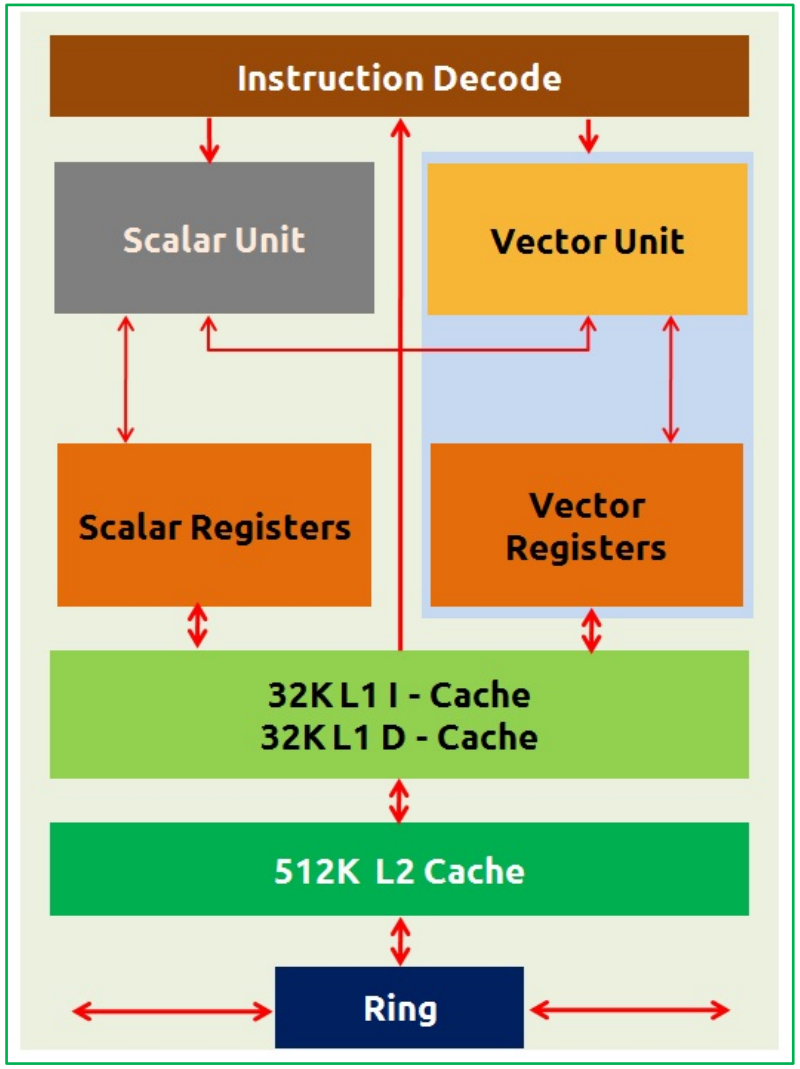
Many-Core Intel Xeon Phi coprocessor

Intel® Xeon Phi Block Diagram

- 50+ Cores with 6+GB of DDR5 Memory
- Each Core Supports 4 way Hyper Threading
- Each Core has its own PMU (Performance Monitoring Unit)
- DDR Bandwidth achieved performance of over 200GB/sec

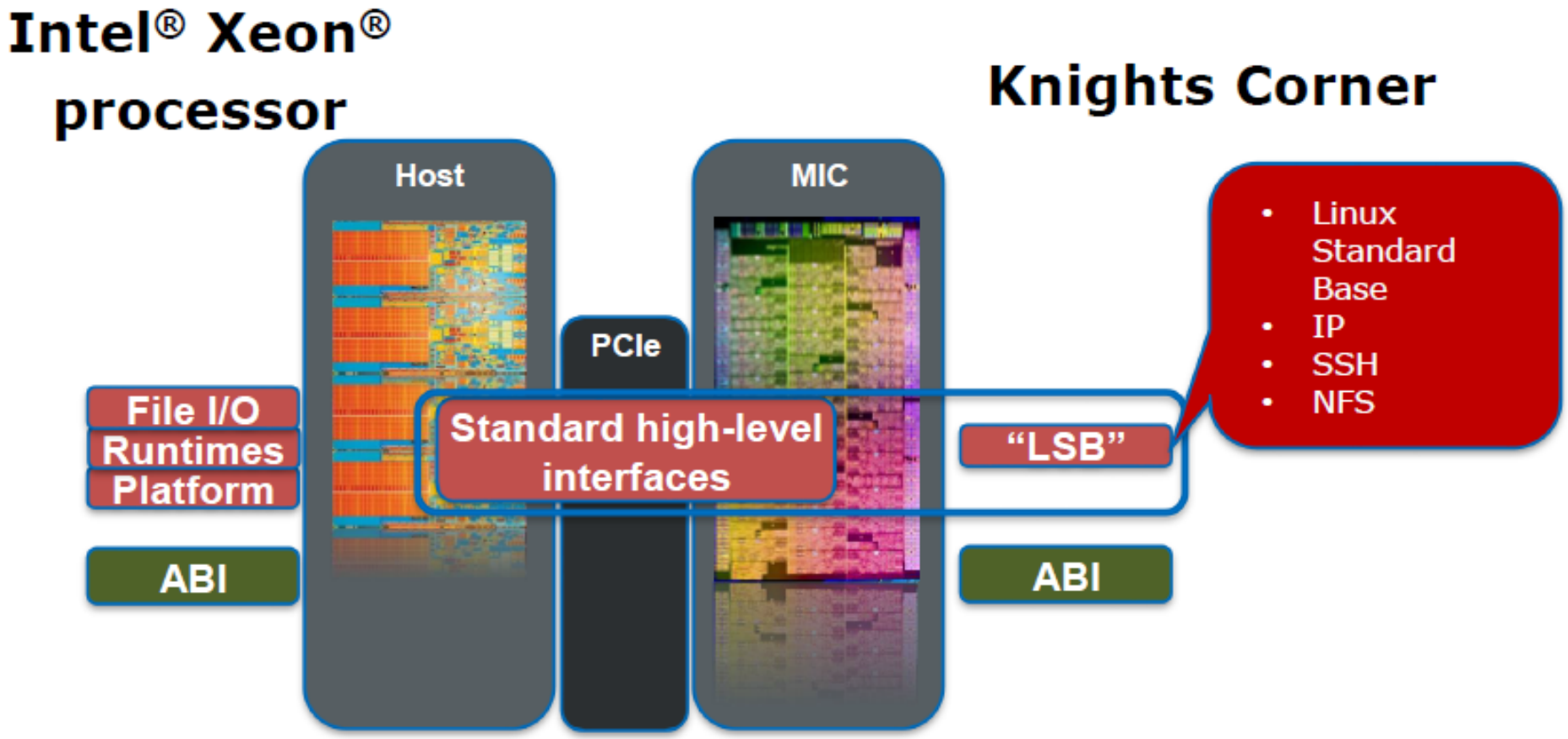


Intel® Xeon Phi Core



* 32KB Data + 32KB Code L1
* 512 bit SIMD, Vector register
* 512KB of dedicated memory for each core, L2

Operating Environment View



A flexible, familiar, compatible operating environment

SIMD

AVX-512

Knights Landing/ Future Xeon 512 bit

FMA

AVX 2

Haswell Architecture, 256 bit

FMA(*,+)

AVX

Sandy Bridge Architecture, 256 bit

Multiplication, Addition

Intel® AVX/AVX2/AVX-512

AVX

- 256-bit basic FP
- 16 registers
- NDS (and AVX128)
- Improved blend
- MASKMOV
- Implicit unaligned

AVX2

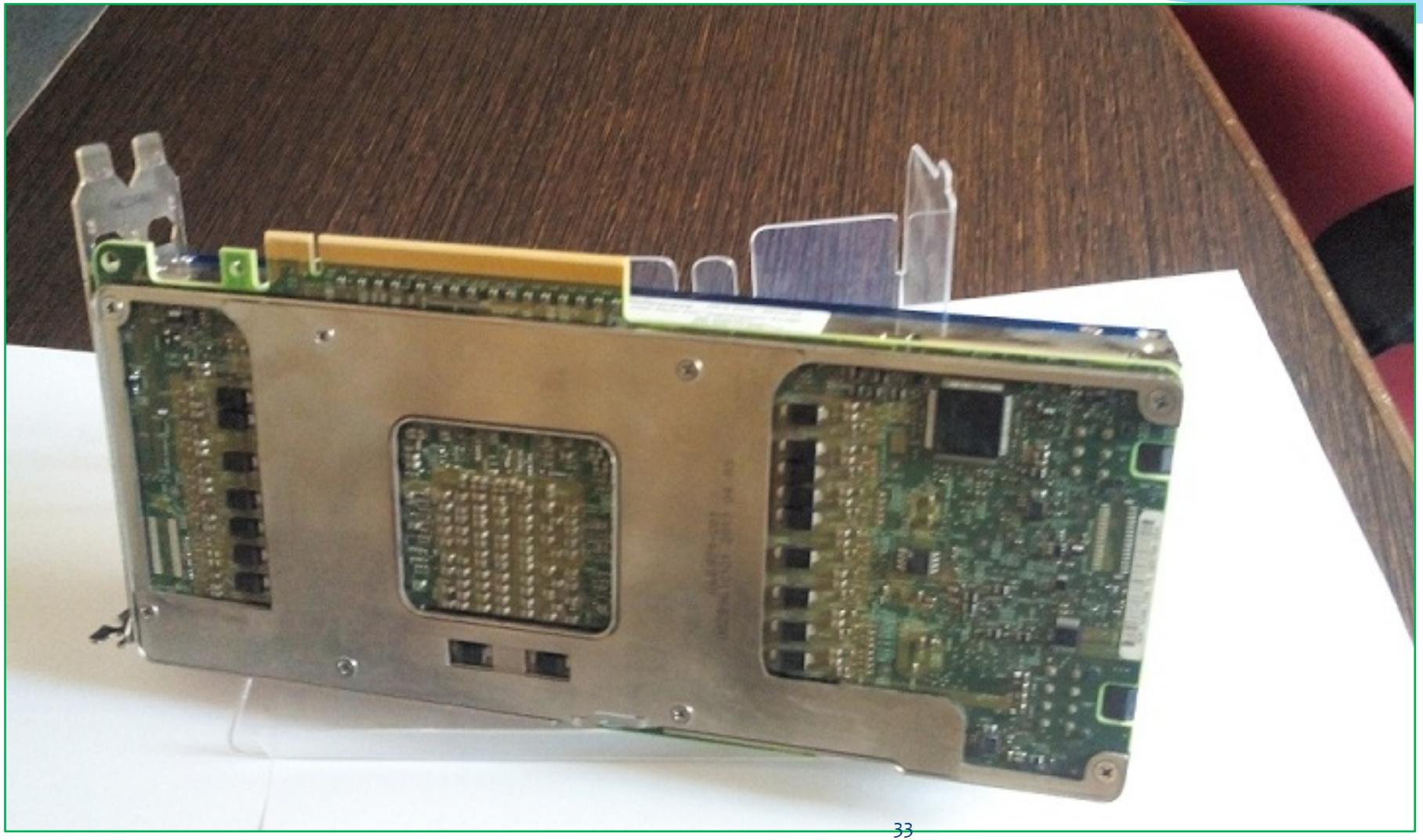
- 256-bit integer
- PERMD
- Gather
- Float16 (IVB 2012)
- 256-bit FP FMA

AVX-512

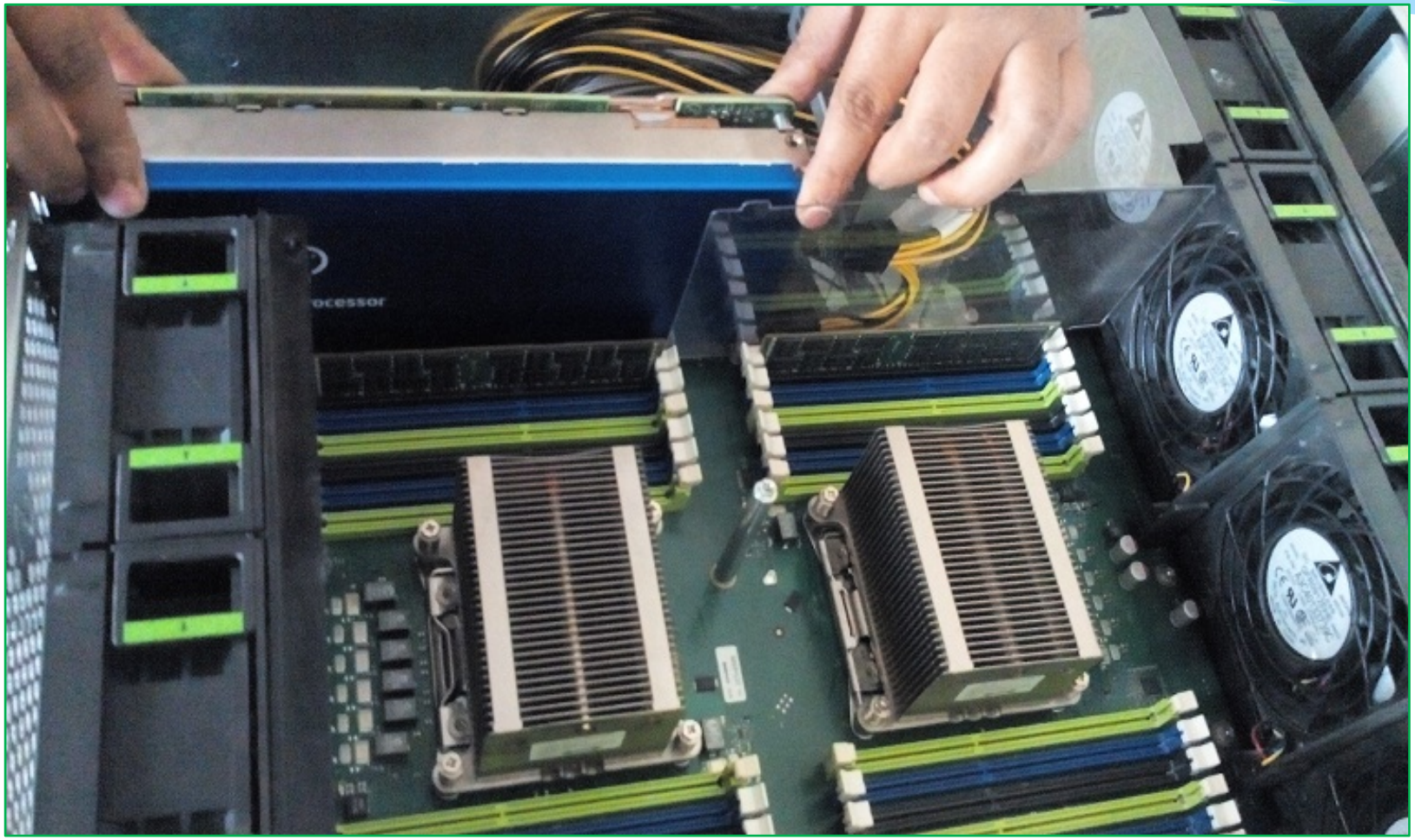
- 512-bit FP and Integer
- 32 registers
- 8 mask registers
- Embedded Rounding
- Embedded broadcast
- Scalar, SSE/AVX promotions
- Native media additions
- HPC additions
- Transcendal support
- Gather/Scatter

Intel® Xeon Phi Card

Intel® Xeon Phi Card



Intel® Xeon Phi Card



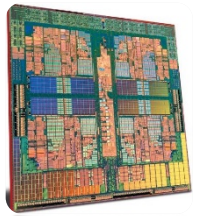
Compute Technology

Parallelism on all Levels:

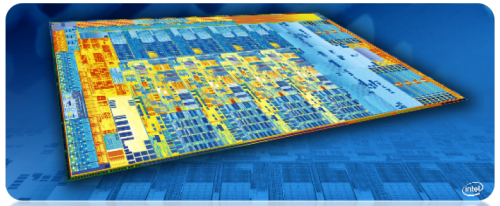
NODES
(Messaging)

Cluster(Nodes+ Fabric)

CORES
(Multi-Threading)

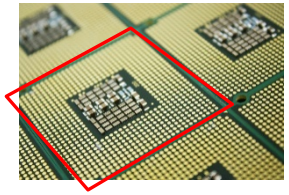


Multi-Core CPU



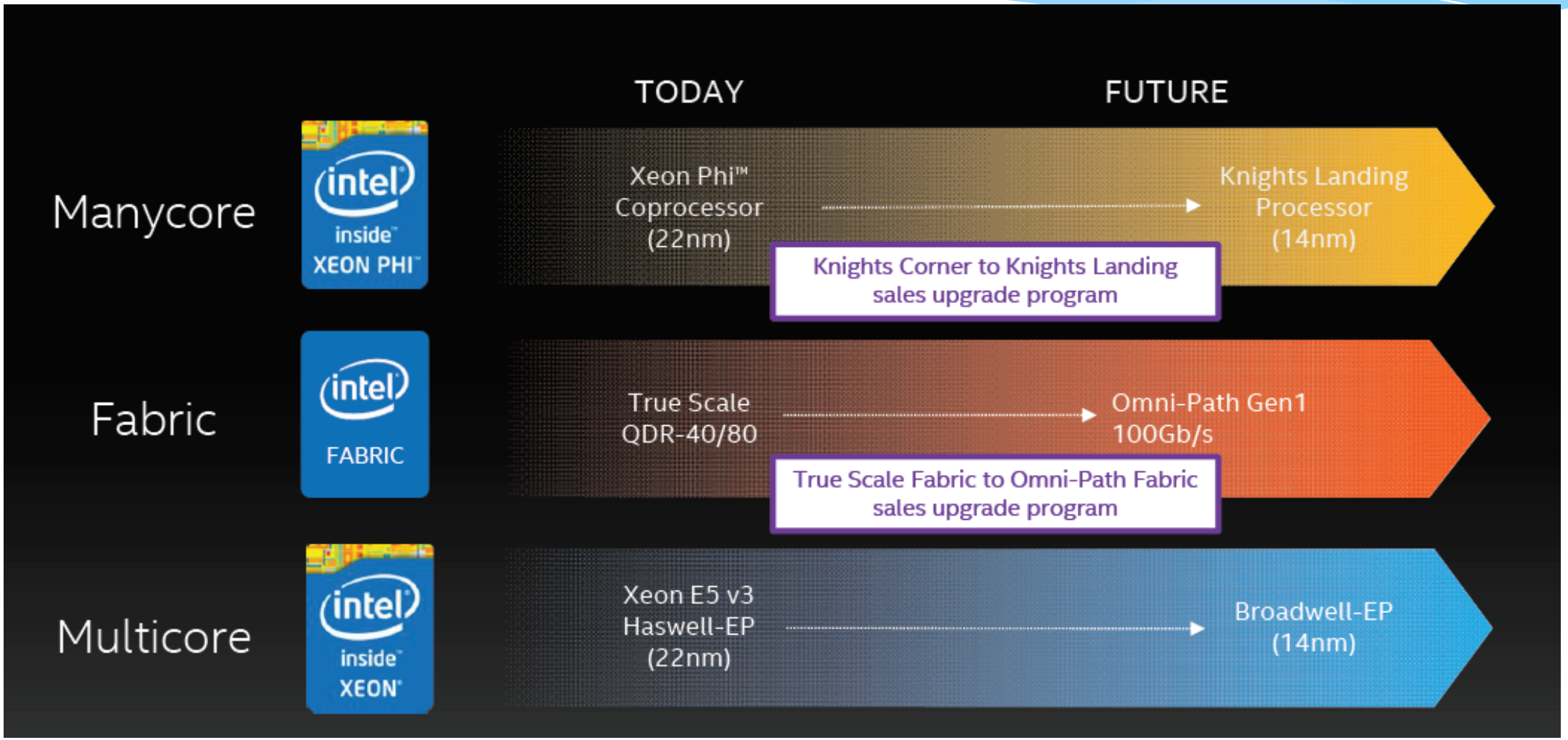
Many-Core CPU

SIMD
(Vectorization)

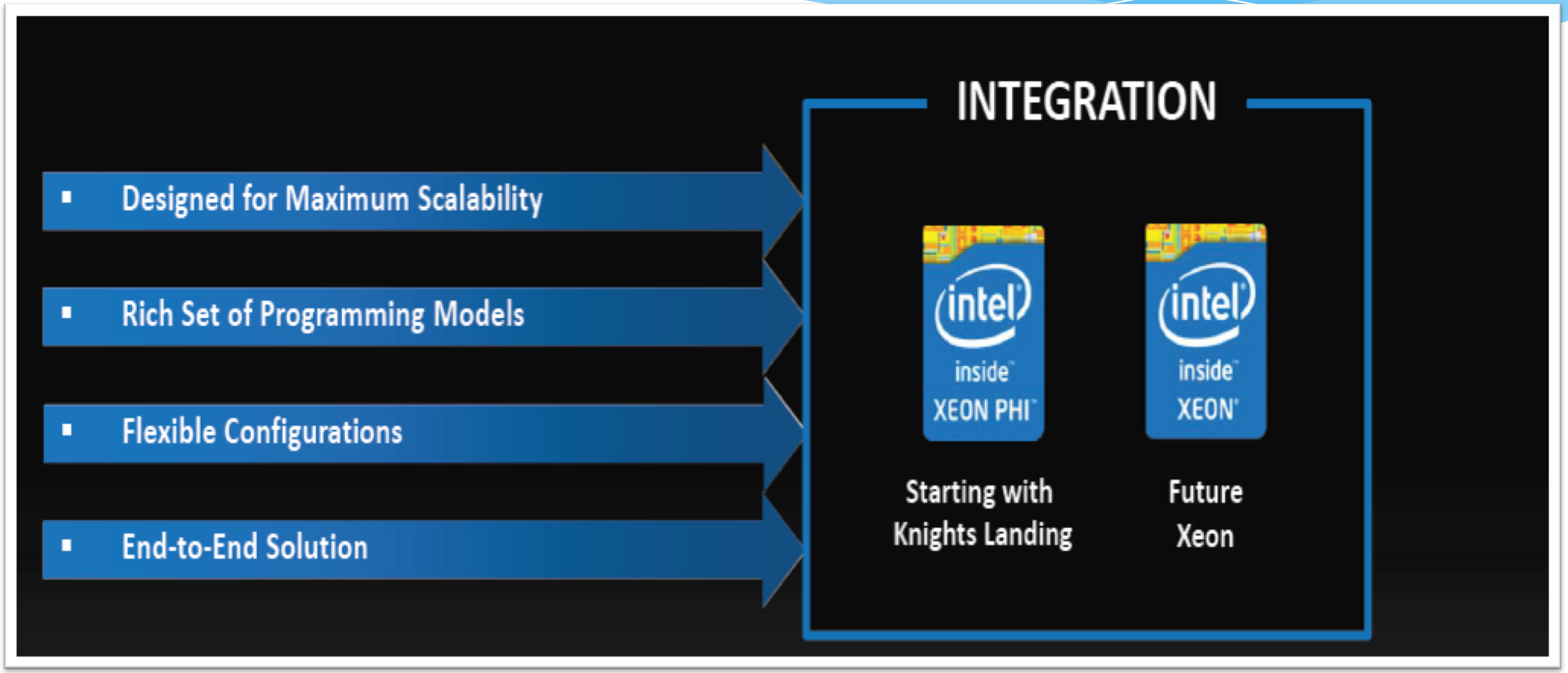


Core

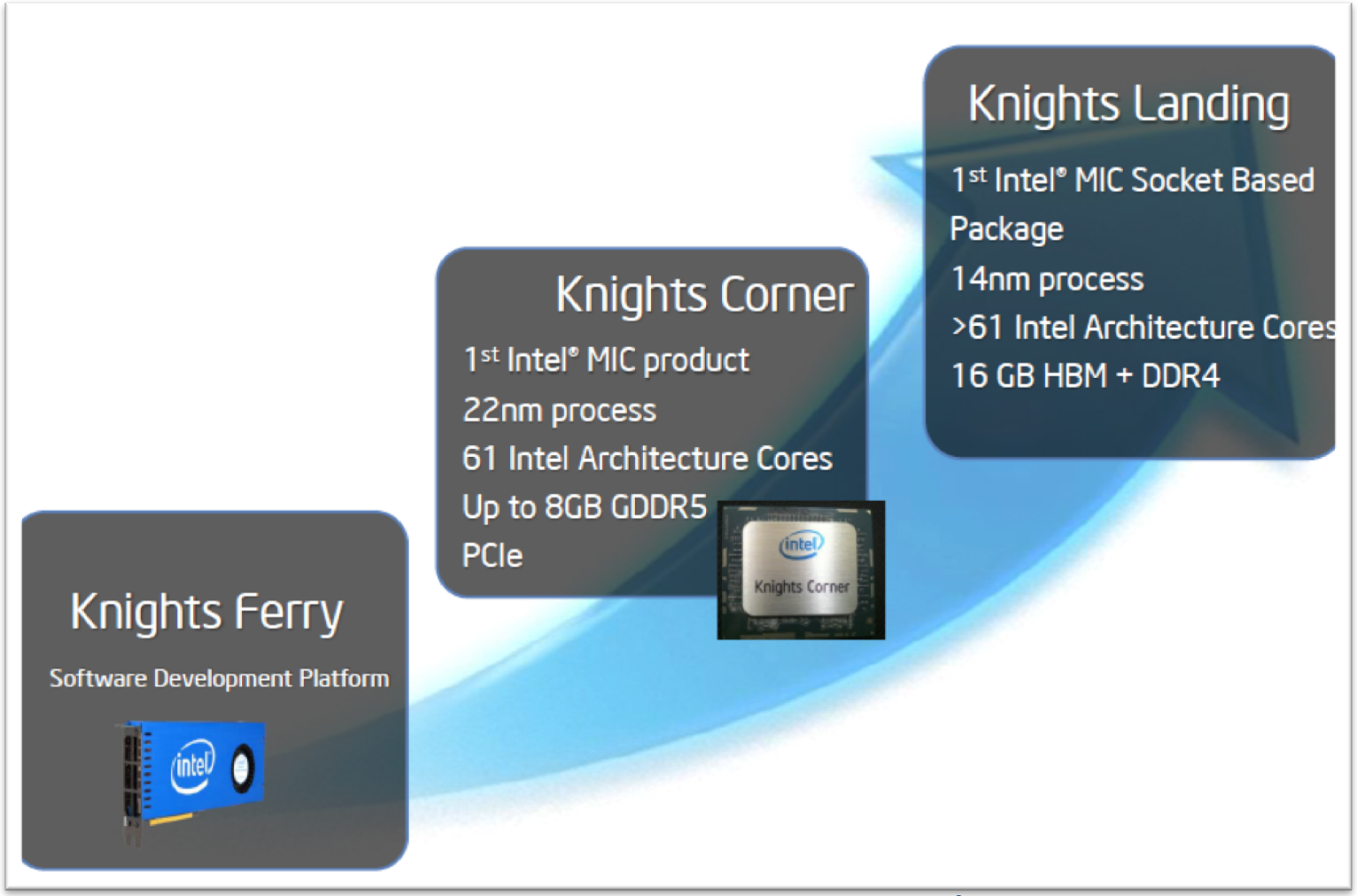
What is Next in HPC?



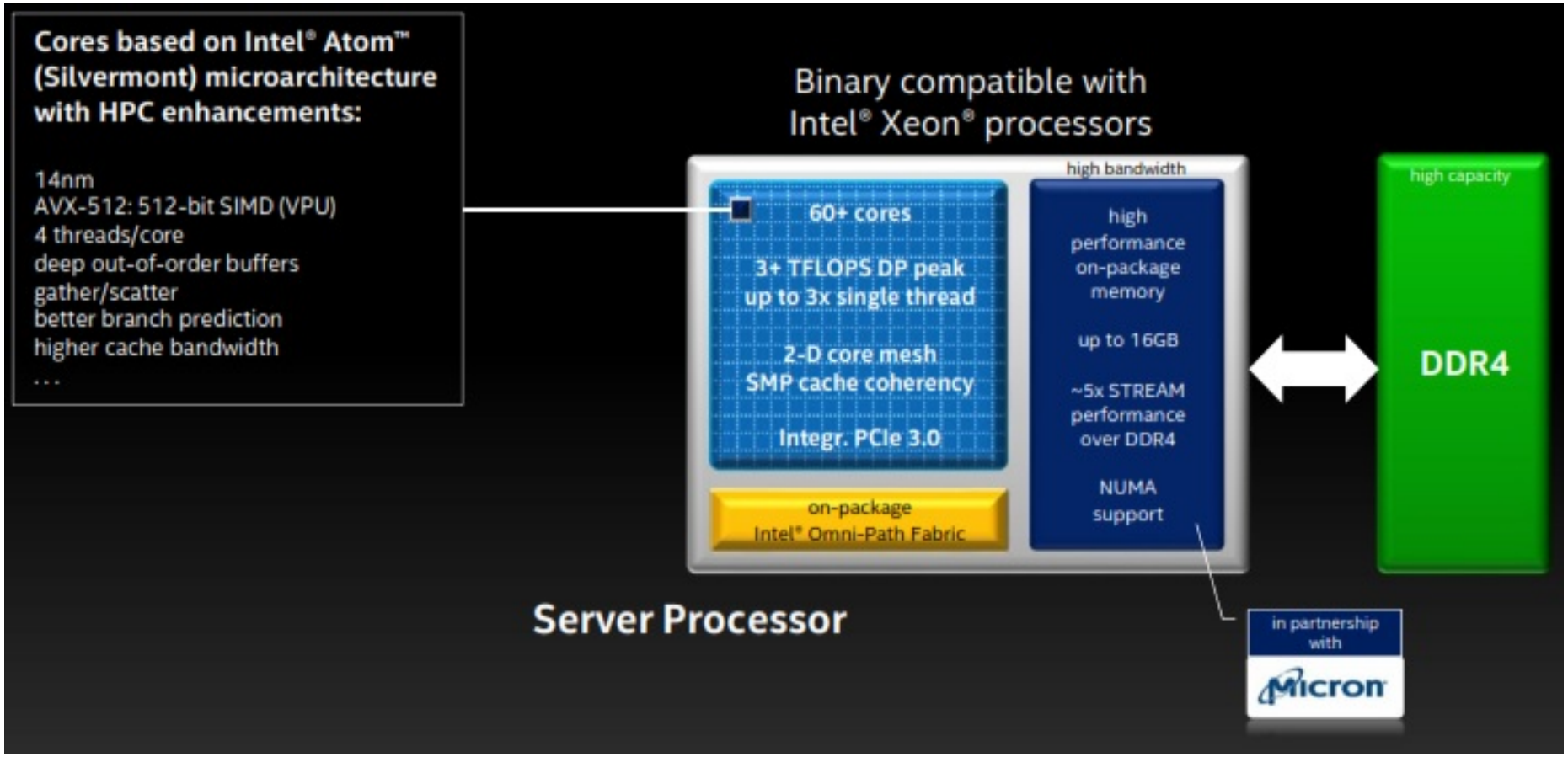
Intel® Omni-Path - The Next-Generation Fabric



The “Knights” Family

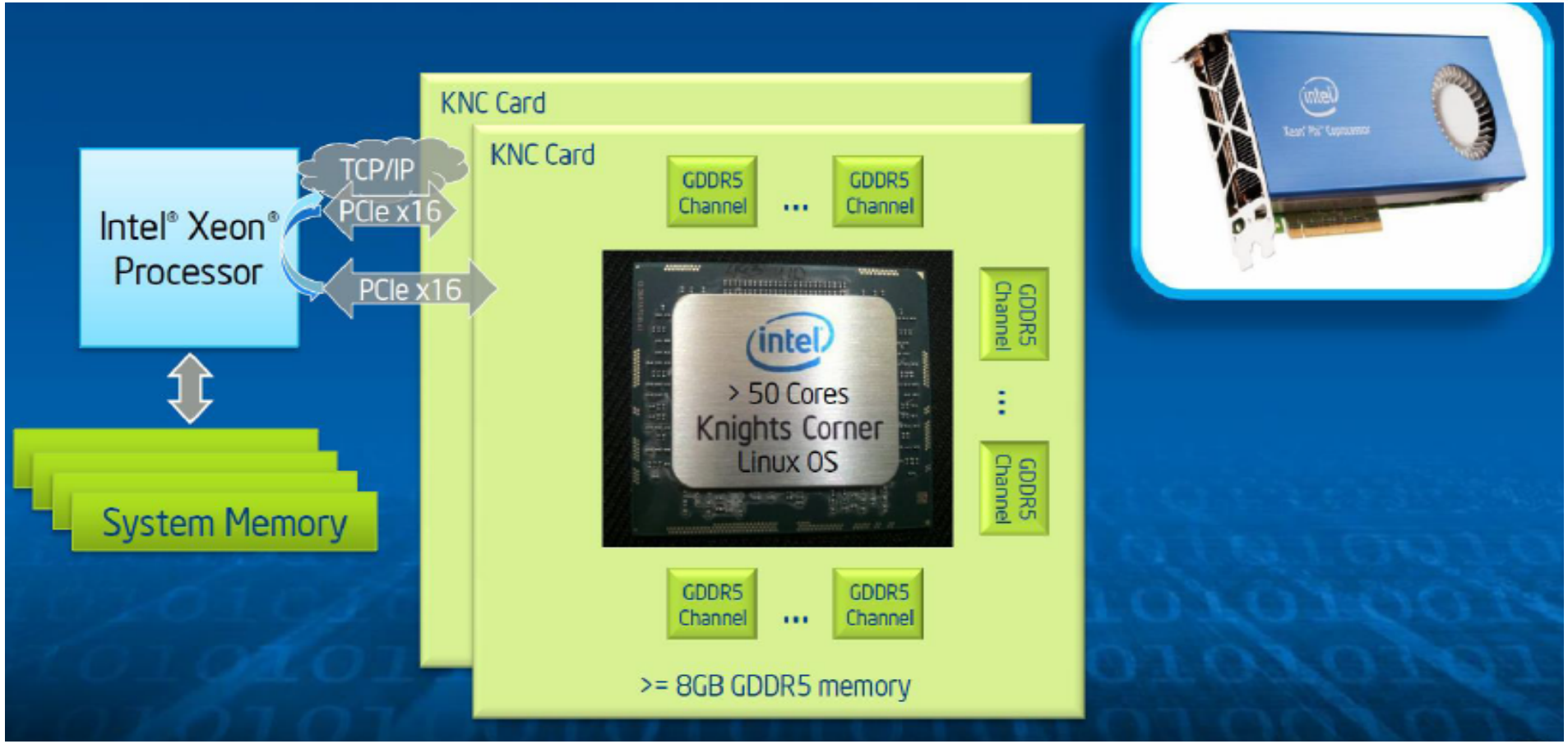


Intel® Knights Landing



Next Generation Intel® Xeon Phi

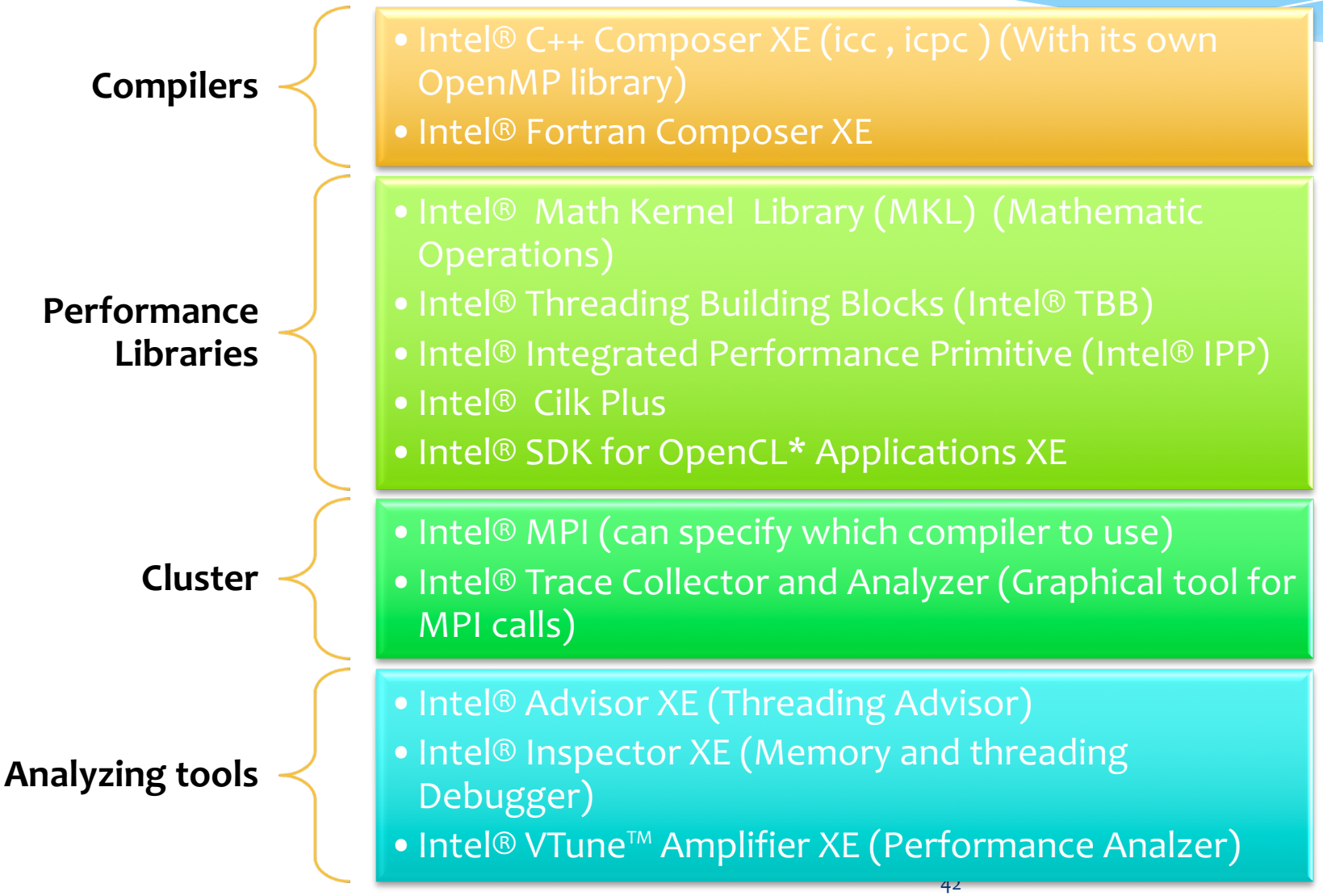
Intel® Knights Landing



Next Generation Intel® Xeon Phi

Intel® Software Technologies

Intel® Software Technologies



* Provides a set of high performance compiler, libraries and analyzing tools for aiding parallel program.

Intel® C/C++ Compiler

* Supports 32 and 64 bits Intel or compatible architecture
* Optimized for Intel Processor
* Levels of optimizations are available.
* Instruction set has evolved with processor and compiler is capable of generating generic or architecture specific code.

Intel® C/C++ Compiler (contd…)

* Levels of optimization
  * -O0 : No optimization
  * -O1 : Optimization for speed over size of the code
  * -O2: (default) Along with basic optimization performs basic loop optimizations, inlining of intrinsic, Intra-file interprocedural optimization
  * -O3: Performs O2 optimizations and enables more aggressive loop transformations. Recommended for applications that have loops that heavily use floating-point calculations and process large data sets.

Intel® Math Kernel Library(MKL)

Math Kernel Library (MKL) is library of optimized math routines for science, engineering, and financial applications.

* Supports 32-bit, 64-bit Intel® or compatible processor and Intel® MIC.
* Fortran and C/C++ are broadly supported language.
* Automatically offloads if Intel® Xeon Phi™ is available.
* Optimized for Intel® Architecture.
* MKL uses OpenMP for threading.

Intel® MKL Contents

BLAS

- Basic Linear Algebra subroutines
- Vector-vector, matrix vector and matrix operations

LAPACK

- Linear Algebra Package
- Solvers and Eigen solvers.

DFTs

- Discrete Fourier transforms
- Mixed radix, multi-dimensional transforms
- Multithreaded

VML

- Vector Math Library
- Set of vectorized transcendental functions
- Most of libm functions, but faster

VSL

- Vector Statistical Library
- Set of vectorized random number generators

Intel® Threading Building Blocks(TBB)

The library consists of data structures and algorithms that allow a programmer to avoid some complications arising from the use of native threading packages in which individual threads of execution are created, synchronized, and terminated manually.

* Supports 32 and 64 bit architecture and also android OS.
* Supports C and C++
* Higher level task based parallelism
* Also available as open source

Intel® Threading Building Blocks(TBB) (contd..)

* Dynamic memory functions are replaced by TBB library functions by loading proxy library at run-time or by linking with the proxy library.
  * These functions include:
    * C library: malloc, calloc, realloc, free
    * Standard POSIX* function: posix_memalign
    * Obsolete functions: valloc, memalign, pvalloc, mallopt
    * Global C++ operators new and delete.

Intel® MPI Library

* Low latency MPI implementation up to 2 times as fast as alternative MPI libraries
* Enable optimized shared memory dynamic connection mode for large SMP nodes
* Increase performance with improved various fabric support
* Accelerate applications using the enhanced tuning utility for MPI

Intel® Integrated Performance Primitives (Intel® IPP)

An extensive library of software functions to help you develop multimedia, data processing, and communications applications.

* Highly optimized using Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX) instruction sets.
* Performance Building - Optimized building blocks perform faster
* Supports Windows, Linux, Android and OS X environments
* Natively supports C/C++ and OpenCL development.

Application Catalog

Questions

Back Up

Portable & Scalable Parallel Programming

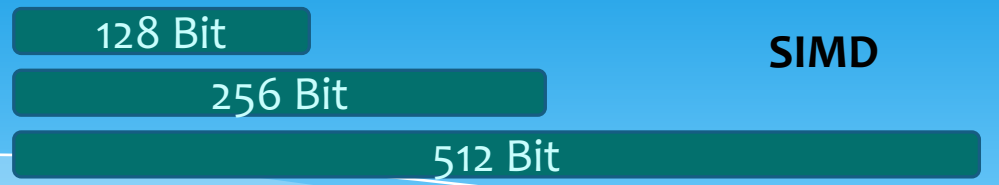
Data Parallelism

Vectorization

Automatic

Directives/Pragmas

Libraries



Thread/ Task Parallelism

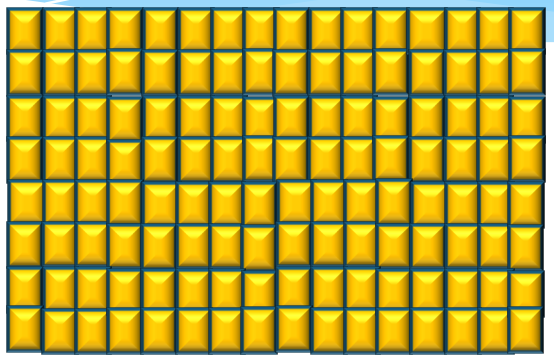
Multi Threading

OpenMP

TBB, OpenCL,

Cilk™ Plus, pthreads

Multi-Core



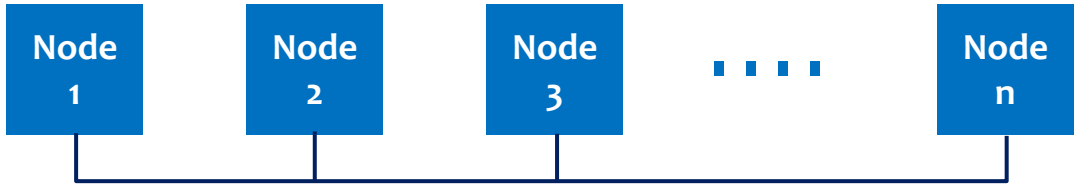
Many-Core

Process Parallelism

Message Passing

MPI

IP-Based

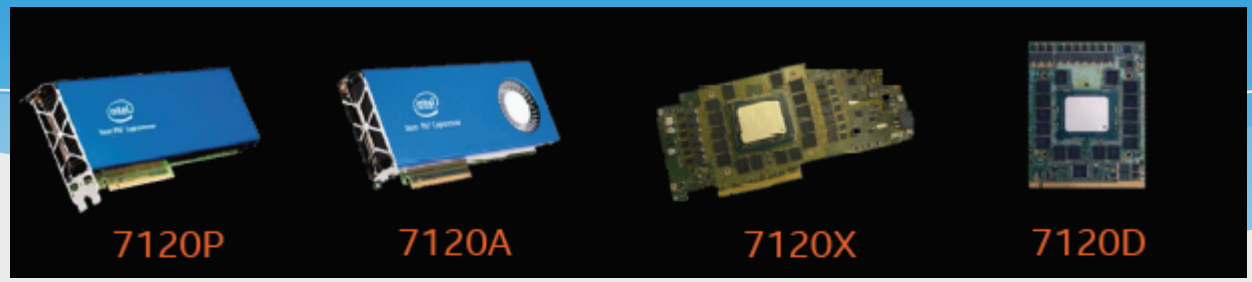
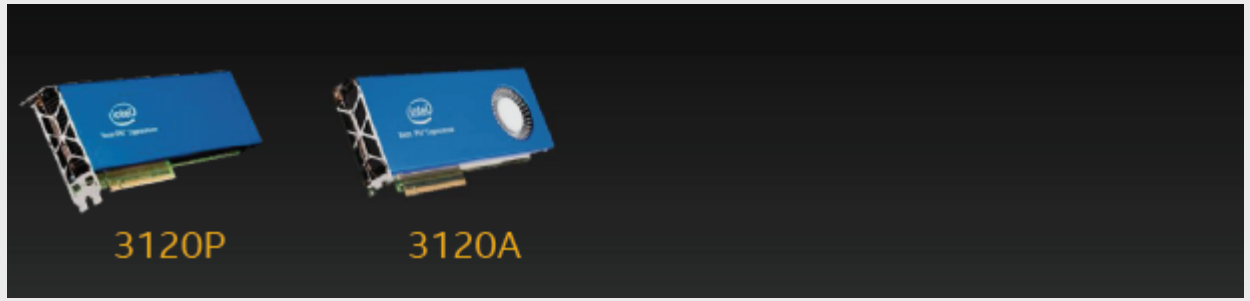


Cluster

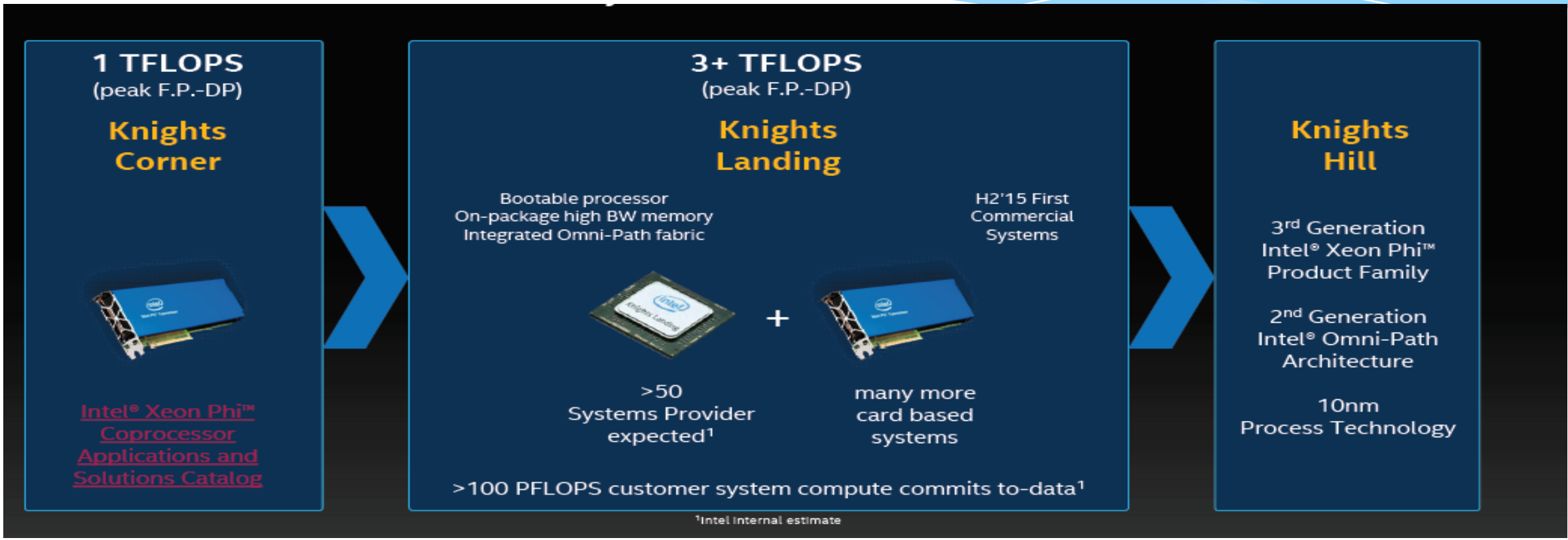
What Intel® Xeon Phi is not

* Xeon phi cores are not directly visible to the host OS, in other words they do not show along with processor.
* Xeon phi card can be booted into uOS from a Window host but still card runs in Linux.
* Not all Linux applications can run on it, core system level commands and network capability are provided on the card but not much else.
* To design a program for better performance with Phi, utilize all the cores and hardware threads dealing with limitation of memory and bandwidth over PCI bus.

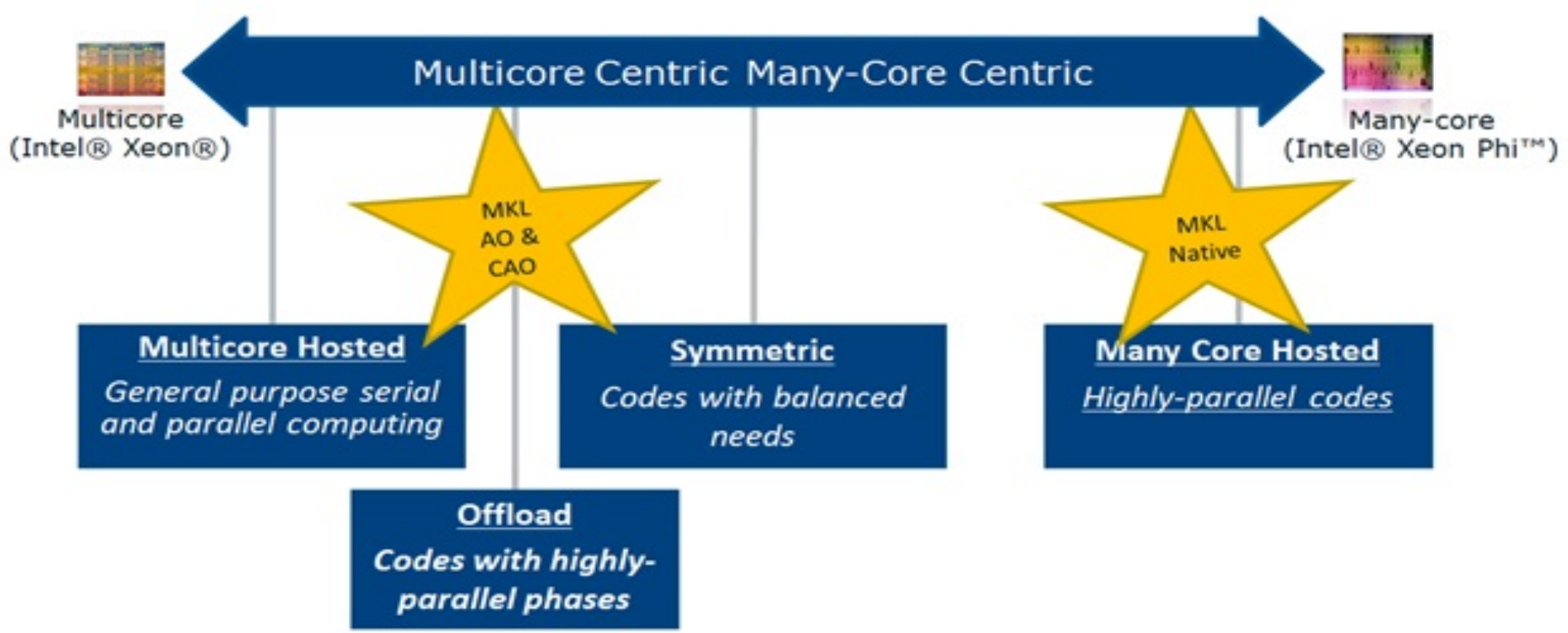
Intel® Xeon Phi™ Coprocessor Product Lineup

7 Family Highest Performance, Most memory Performance leadership	16GB GDDRS 352GB/S >1.2TF DP 270-300W	
5 Family Optimized for High Density Environments Performance/Watt leadership	8GB GDDRS 300GB/S >1TF DP 225-245W	
3 Family Outstanding Parallel Computing Solution Performance/Leadership	6GB GDDRS 240GB/S >1TF DP 300W	

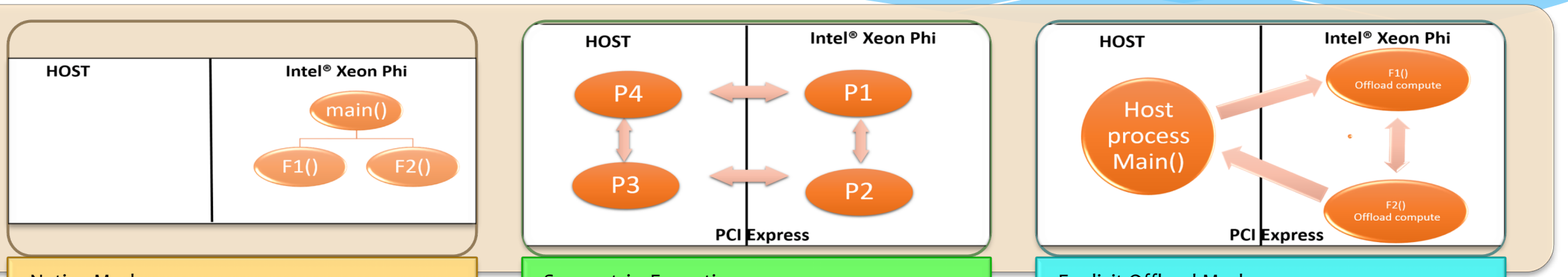
Intel® Xeon Phi Product Family



Execution Models



Programming Modes for Intel® Xeon-Phi



Native Mode

- Running Native Application Directly on Xeon-Phi by logging on to it
- An application has to be cross compiled for Xeon Phi operating environment. Intel® Composer XE provides simple switch to generate cross compiled code

Symmetric Execution:

- Application processes run on both the host and the Intel® Xeon Phi™ coprocessor.
- Communication through MPI.
- This execution environment treats Xeon Phi card as another node in a cluster in a heterogeneous cluster environment

Explicit Offload Mode

- Also known as heterogeneous programming mode
- Selective portions of data/code is transferred to Xeon-Phi at runtime and executed
- Host and the coprocessor may or may not work in parallel.
- Intel® Composer XE provide directives to perform offload computations

Intel® Xeon Phi™ Code Modernization Enablement Program

Goals:

1. Enable developers and end customers to jumpstart code modernization efforts on Intel® Architecture through increased training availability and affordable access to Intel-based hardware and software solutions

2. Prepare for Knights Landing (KNL) by enabling software to be parallelized, vectorized and optimized on today's Intel® Xeon Phi™ coprocessors

Intel® Code Modernization Enablement Program for Developers

Promotion on Intel® Xeon Phi™ Coprocessors 5110P and 5120D

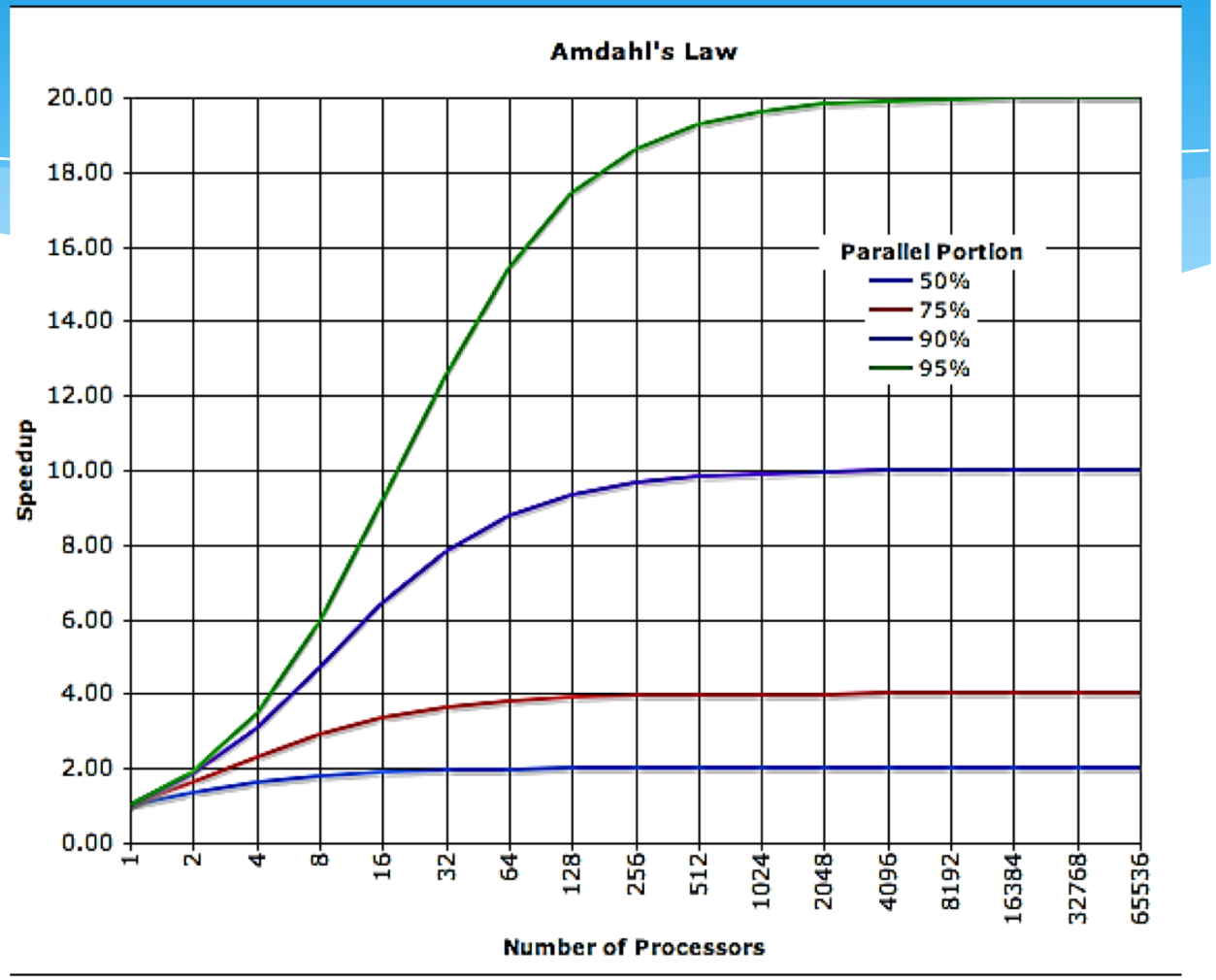
- How to redeem:
- Visit http://software.intel.com/en-us/articles/intel-code-modernizationenablement-program for a list of participating partners
- Contact your preferred OEM sales representative

Promotion on Intel® Parallel Studio XE Cluster Edition – 1 seat (80% off)

- How to redeem:
- Requirement: Attend Intel® Code Modernization workshop or training webinar and purchase Intel® Xeon Phi™ Coprocessor 5110P or 5120D
- Limited to 1 license per Intel® Xeon Phi™ Coprocessor 5110P or 5120D purchased

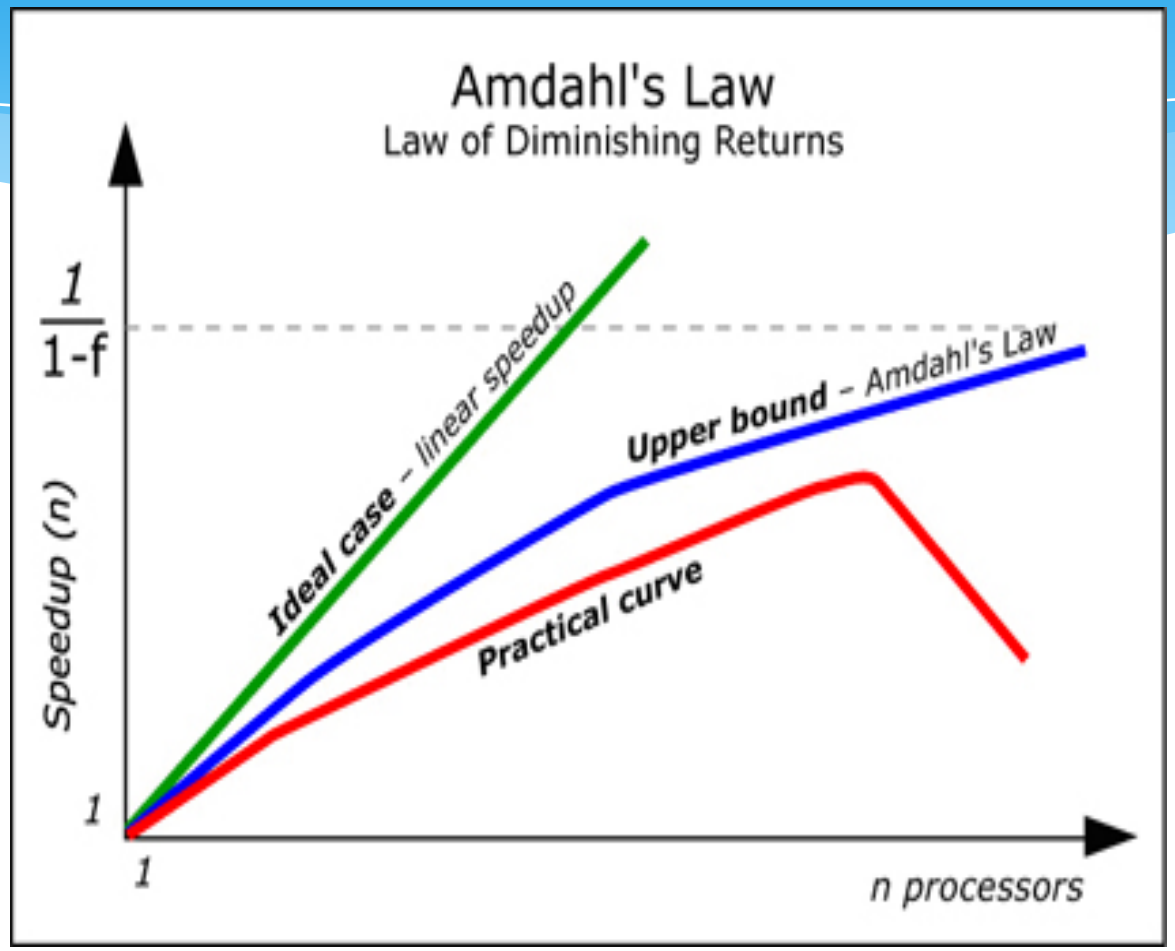
Amdhal's Law

* If F is the fraction of a calculation that is sequential, and (1-F) is the fraction that can be parallelized, then the maximum speed-up that can be achieved by using P processors is 1/(F+(1-F)/P).
* Maximum speedup of parallel computing is limited by the fraction of problem that must be performed sequentially for given dataset.



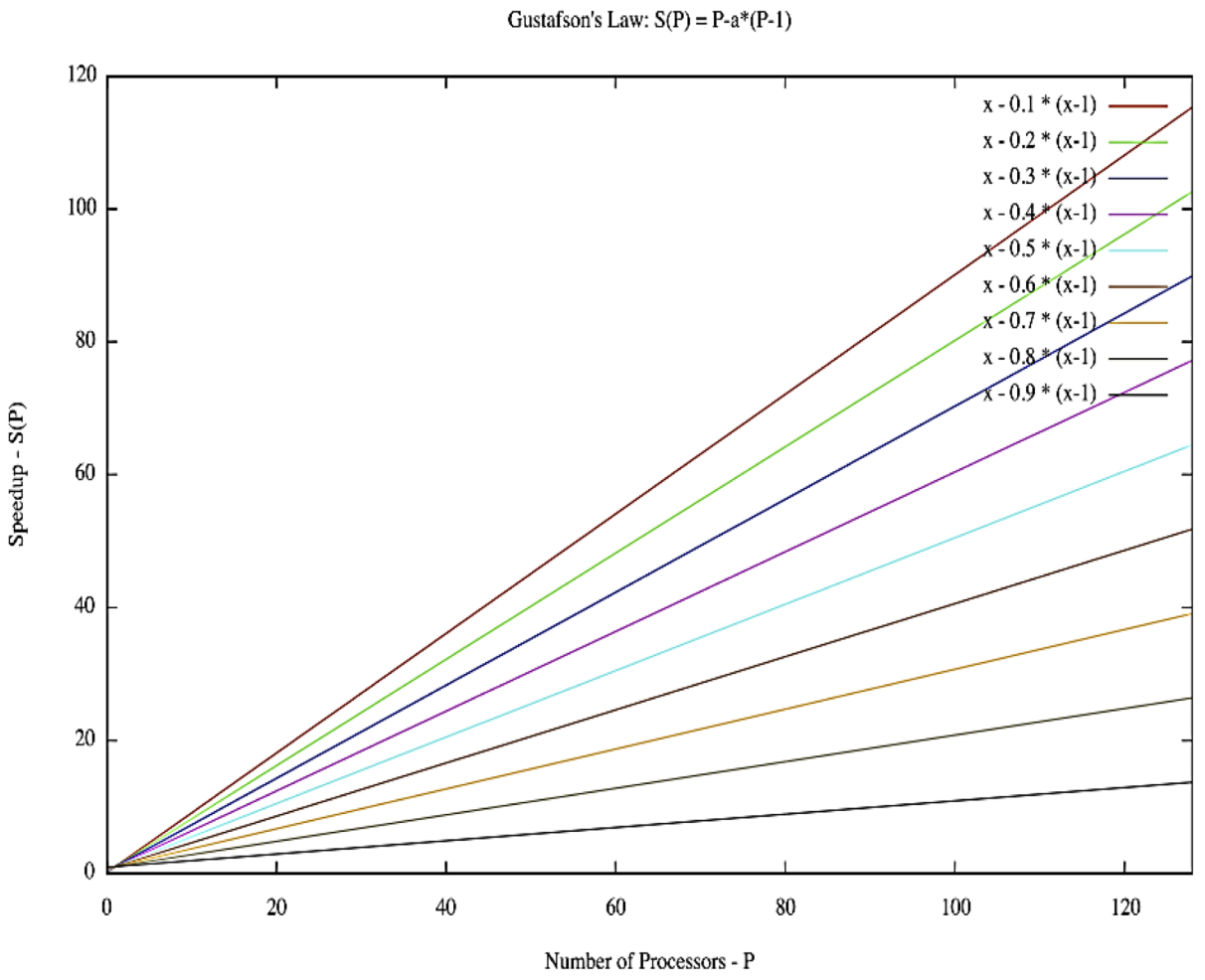
Amdhal's Law

* The actual speed-ups are always less than the speed-up predicted by Amdahl's Law
* Amdahl's law ignores the communication cost in MIMD systems.
  * This term does not occur in SIMD systems, as communications routing steps are deterministic and counted as part of computation cost.
* On communications-intensive applications, it doesn't account to communication slowdown due to network congestion.
* As a result, Amdahl's law usually overestimates speedup achievable



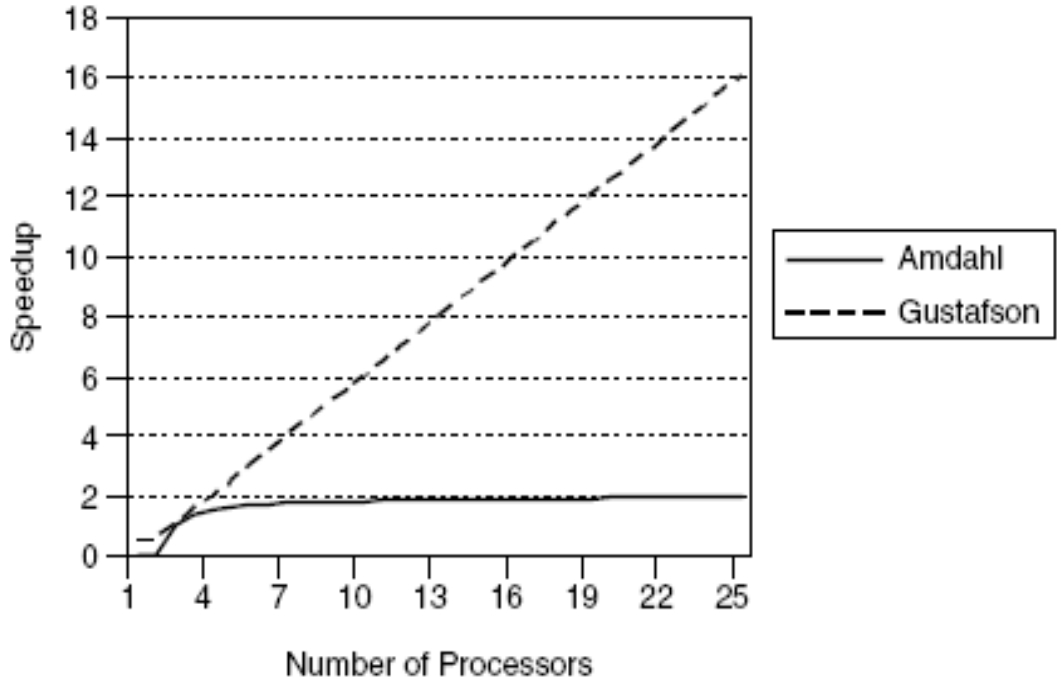
Gustafon's Law

* If α is the largest non-parallelizable fraction of any parallel process, and P is the number of processors, then speed up is S(P), where S(P)=P-α(P-1)
* It infers that if α diminishes then speed up ie S(P) =P. Also, if the problem size is allowed to grow monotonically with *P*, then the sequential fraction of the workload would not ultimately come to dominate



Gustafon's Law

* The true parallel power of a large multiprocessor system is only achievable on arbitrarily large dataset.
* Gustafson's law addresses the shortcomings of Amdahl's law, which does not fully exploit the computing power that becomes available as the number of machines increases



Linear speed of Gustafon's law compared to Amdhal's law with 50% code available for parallelization.

Analogy

* Amdahl's Law:
  "Suppose a car is traveling between two cities 60 miles apart, and has already spent one hour traveling half the distance at 30 mph. No matter how fast you drive the last half, it is impossible to achieve 90 mph average before reaching the second city. Since it has already taken you 1 hour and you only have a distance of 60 miles total; going infinitely fast you would only achieve 60 mph."

* Gustafson's Law :

"Suppose a car has already been traveling for some time at less than 90mph. Given enough time and distance to travel, the car's average speed can always eventually reach 90mph, no matter how long or how slowly it has already traveled. For example, if the car spent one hour at 30 mph, it could achieve this by driving at 120 mph for two additional hours, or at 150 mph for an hour, and so on.

Courtesy: wikipedia

Modes on Xeon Phi

Native Mode

* Application is compiled for Xeon-Phi, the executable is copied to Xeon-Phi and executed on it using SSH.
  * To use icc –mmic option has to be added.
  * Intel SDK tool kit has extended its support to GNU C compiler
  * mico is the default name given to the first xeon-phi in /etc/hosts
  * MPSS transfers all the SSH keys to Xeon-Phi, which enables login
  * micnativeloadex utility can also be used to execute on Xeon-Phi

Native mode Example

```
#include<stdio.h>
#include<unistd.h>
void main()
{


      char hostname[1024];
      gethostname(hostname, 1023);
      int  cores = sysconf(_SC_NPROCESSORS_ONLN);
      printf("Hello world!! @ '%s` co-processor and it has  %d cores\n",
hostname, cores);
}
```

Native Run on Xeon-Phi

```
[test@phi Training]$ icc xeon_phi_lab1.c -mmic
[test@phi Training]$ scp a.out mic0:~
a.out                                         100%   11KB  10.5KB/s   00:00
[test@phi Training]$ ssh mic0
[test@phi-mic0 ~]$ ./a.out
Hello world!! @ phi-mic0 coprocessor and it has 228 cores
[test@phi-mic0 ~]$
```

```
[test@phi Training]$ icc xeon_phi_lab1.c -mmic
[test@phi Training]$ micnativeloadex ./a.out
Hello world!! @ phi-mic0 coprocessor and it has 228 cores
```

Utility from host

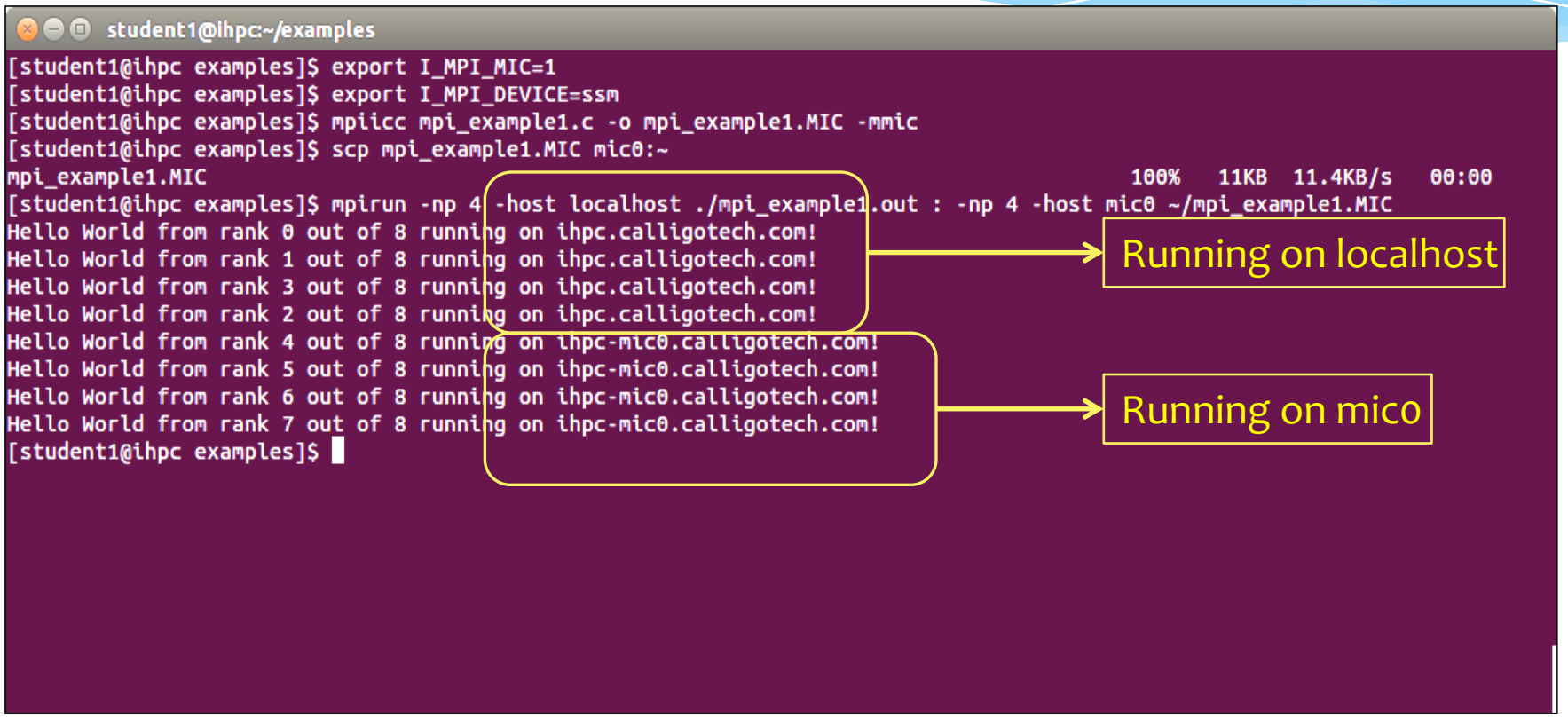
Symmetric mode

* Host and Phi can operate symmetrically as MPI targets
* Steps to build and run in symmetric mode:
  * Compile the program for the processor
  * Compile the program for the host
  * Copy the executable on to the coprocessor
  * Set I_MPI_MIC variable to 1 and I_MPI_DEVICE = ssm
  * And run using mpirun as shown in example
    * e.g: **mpirun –host mic0 –np 4 ~/mic0_exe : -host mic1 –np 8 ~/mic1_exe : -host localhost –np 10 ./localhost_exe**

Symmetric mode Example

```
#include<stdio.h>
#include<unistd.h>
#include<mpi.h>
int main (int argc, char *argv[])
{
        int rank, size;
        MPI_Init (&argc, &argv);
        MPI_Comm_size (MPI_COMM_WORLD, &size);
        MPI_Comm_rank (MPI_COMM_WORLD, &rank);
        char hostname[1024];
        gethostname(hostname, 1023);
        printf("Hello world from rank %d out of %d  running on%s\n”,rank,size,
hostname);
        MPI_Finalize ();
        return 0;
}
```

Symmetric mode on multiple Xeon Phi



Explicit offload mode

* Application can explicitly mark areas of the code to be run on the coprocessor.
  * It's a set of Compiler, C language directives
  * No special options are needed to compile code
  * Compiler produces 2 versions of executable for the offload portion (Xeon and Xeon-Phi)
    - __MIC__ is defined for Xeon-Phi
    - __MIC__ is not defined for Xeon
  * Console Proxy enables IO operations with printf()
  * Environment Variable OFFLOAD_REPORT (0/1/2) enables offload Diagnosis
  * Preprocessor Macro __MIC__ enables writing multi-version (Xeon/Xeon-Phi) offload code

Proxy Console

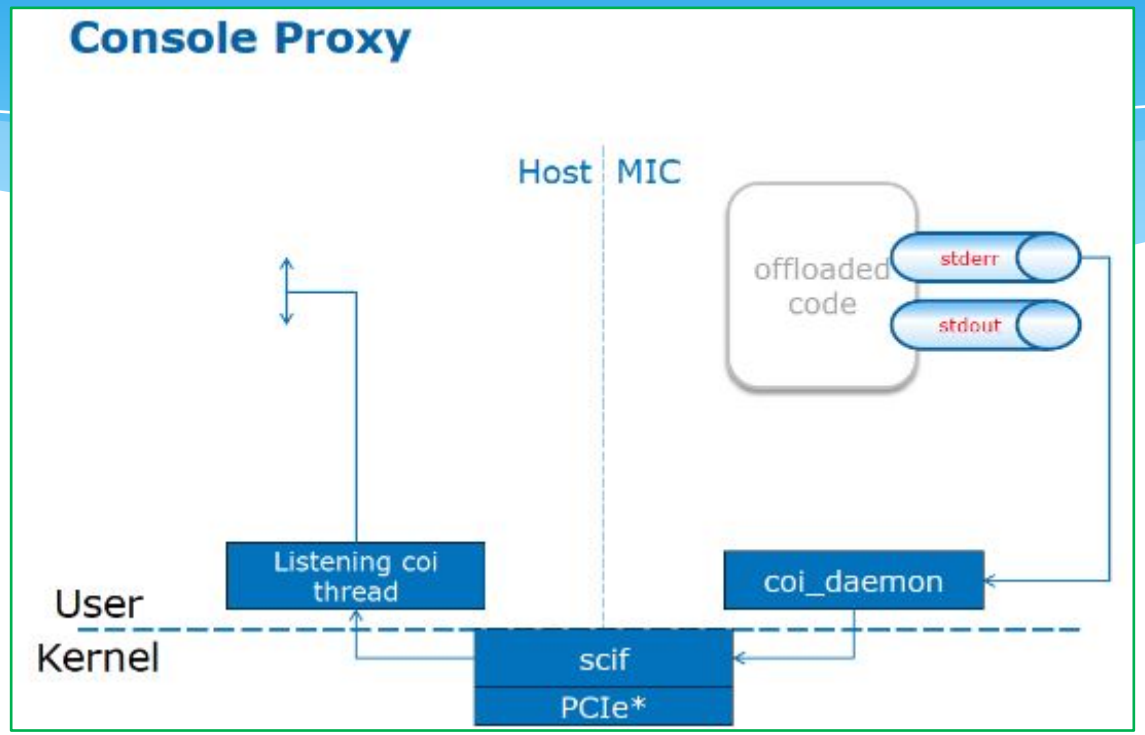


Image Source: *Intel Corporation*

* Coprocessor OS buffers output and sends it to host
* STDOUT and STDERR is supported (No STDIN supported)
* Consistency between Host IO and Xeon-Phi IO can be achieved using fflush(0)
* Environment Variable MIC_PROXY_IO is enabled by default. Can be disabled.
* SCIF, Symmetric Communication Interface between the host processors and the Intel Xeon Phi™ coprocessors in a heterogeneous computing environment.

```
#include<stdio.h>
#include<unistd.h>
void main()
{
        char hostname1[1024], hostname2[1024]; ;
        gethostname(hostname1, 1023);
        int cores2, cores1 = sysconf(_SC_NPROCESSORS_ONLN);
        printf("Hello world! Running on processor = '%s' with %d cores\n",hostname1, cores1);
         #pragma offload target(mic:0)
         {
            gethostname(hostname2, 1023);
            cores2 = sysconf(_SC_NPROCESSORS_ONLN);
            printf("Hello world! Running on co-processor = '%s' with %d logical cores\n",hostname2,
cores2);
        }
 }
```

```
[test@phi Training]$ icc offload_example1.c
[test@phi Training]$ ./a.out
Hello world! Running on processor = 'phi' with 16 cores
Hello world! Running on co-processor = 'phi-mic0' with 228 logical cores
[test@phi Training]$
```

```
[test@phi Training]$ export OFFLOAD_REPORT=1
[test@phi Training]$ icc offload_example1.c
[test@phi Training]$ ./a.out
Hello world! Running on processor = 'phi' with 16 cores
[Offload] [MIC 0] [File]                    offload_example1.c
[Offload] [MIC 0] [Line]                    30
[Offload] [MIC 0] [Tag]                     Tag 0
[Offload] [HOST]  [Tag 0] [CPU Time]        0.179058(seconds)
[Offload] [MIC 0] [Tag 0] [MIC Time]        0.000460(seconds)

Hello world! Running on co-processor = 'phi-mic0' with 228 logical cores
[test@phi Training]$
```

```
[test@phi Training]$ export OFFLOAD_REPORT=2
[test@phi Training]$ icc offload_example1.c
[test@phi Training]$ ./a.out
Hello world! Running on processor = 'phi' with 16 cores
[Offload] [MIC 0] [File]                    offload_example1.c
[Offload] [MIC 0] [Line]                    30
[Offload] [MIC 0] [Tag]                     Tag 0
[Offload] [HOST]  [Tag 0] [CPU Time]        0.190259(seconds)
[Offload] [MIC 0] [Tag 0] [CPU->MIC Data]   1028 (bytes)
[Offload] [MIC 0] [Tag 0] [MIC Time]        0.000436(seconds)
[Offload] [MIC 0] [Tag 0] [MIC->CPU Data]   1028 (bytes)

Hello world! Running on co-processor = 'phi-mic0' with 228 logical cores
[test@phi Training]$
```

```
[test@phi Training]$ export OFFLOAD_REPORT=3
[test@phi Training]$ icc offload_example1.c
[test@phi Training]$ ./a.out
Hello world! Running on processor = 'phi' with 16 cores
[Offload] [HOST]          [State]           Initialize logical card 0 = physical card 0
[Offload] [MIC 0] [File]                    offload_example1.c
[Offload] [MIC 0] [Line]                    30
[Offload] [MIC 0] [Tag]                     Tag 0
[Offload] [HOST]  [Tag 0] [State]           Start target
[Offload] [HOST]  [Tag 0] [State]           Setup target entry: __offload_entry_offload_example1_c_30mainicc102104724074kPeK
[Offload] [HOST]  [Tag 0] [State]           Host->target pointer data 0
[Offload] [HOST]  [Tag 0] [Signal]          signal : none
[Offload] [HOST]  [Tag 0] [Signal]          waits  : none
[Offload] [HOST]  [Tag 0] [State]           Host->target pointer data 1028
[Offload] [HOST]  [Tag 0] [State]           Host->target copyin data 0
[Offload] [HOST]  [Tag 0] [State]           Execute task on target
[Offload] [HOST]  [Tag 0] [State]           Target->host pointer data 1028
[Offload] [MIC 0] [Tag 0] [State]           Start target entry: __offload_entry_offload_example1_c_30mainicc102104724074kPeK
[Offload] [MIC 0] [Tag 0] [Var]             hostname2_V$3  INOUT
[Offload] [MIC 0] [Tag 0] [Var]             cores2_V$4  INOUT
Hello world! Running on co-processor = 'phi-mic0' with 228 logical cores
[Offload] [MIC 0] [Tag 0] [State]           Target->host copyout data   0
[Offload] [HOST]  [Tag 0] [CPU Time]        0.190395(seconds)
[Offload] [MIC 0] [Tag 0] [CPU->MIC Data]   1028 (bytes)
[Offload] [MIC 0] [Tag 0] [MIC Time]        0.000502(seconds)
[Offload] [MIC 0] [Tag 0] [MIC->CPU Data]   1028 (bytes)

[test@phi Training]$
```

Multiple Xeon-Phi

* Current systems can support up to 8 Xeon-Phi
* Program running on the host can offload to multiple Xeon-Phi
  * Single Thread on Host calls non-blocking offload to Xeon-Phi
  * Multiple Threads on Host each call a single blocking offload to Xeon-Phi
* Native Program on Xeon-Phi can use multiple coprocessors in MYO mode

Automatic offload

* Intel® MKL takes advantage of Intel® Xeon phi for computationally intensive Intel® MKL functions.
* No change to the program and compilation way.
* To enable Automatic offload:
  * MKL_MIC_ENABLE=1
    or
  * mkl_mic_enable()
* Default work division can be override by user by setting Environment variables or support functions.
* Automatic offload works better when matrix size is right (>2048) else overhead of data transferring overshadows performance benefit by offloading.
* Can be used with compiler assisted offload.

Automatic offload

```
[test@phi Training]$ export MKL_MIC_ENABLE=1
[test@phi Training]$ export OFFLOAD_REPORT=2
[test@phi Training]$ icc mm_mkl.c -mkl -o mm_mkl.out
[test@phi Training]$ ./mm_mkl.out
[MKL] [MIC --] [AO Function]    DGEMM
[MKL] [MIC --] [AO DGEMM Workdivision]  0.08 0.92
[MKL] [MIC 00] [AO DGEMM CPU Time]      11.708980 seconds
[MKL] [MIC 00] [AO DGEMM MIC Time]      9.261564 seconds
[MKL] [MIC 00] [AO DGEMM CPU->MIC Data] 1626435968 bytes
[MKL] [MIC 00] [AO DGEMM MIC->CPU Data] 3173852160 bytes
Multiplied using Intel(R) MKL
```

Explicit Offload with MPI on Xeon-Phi

```
#include<stdio.h>
#include<unistd.h>
#include<mpi.h>
int main (int argc, char *argv[])
{               int rank, size;
                MPI_Init (&argc, &argv);
                MPI_Comm_size (MPI_COMM_WORLD, &size);
                MPI_Comm_rank (MPI_COMM_WORLD, &rank);
                char hostname1[1024], hostname2[1024];
                gethostname(hostname1, 1023);
                int cores2 ,cores1 = sysconf(_SC_NPROCESSORS_ONLN);
                printf("Hello world! Processor= '%s' %d cores by %d rank / %d size\n",hostname1,cores1,rank,size);
                #pragma offload target(mic:0) in(rank) in(size)
                {
                    gethostname(hostname2, 1023);
                   cores2 = sysconf(_SC_NPROCESSORS_ONLN);
                   printf("Hello world! Running on co-processor = '%s' with %d logical cores by %d rank / %d size
                                                                        \n",hostname2, cores2, rank, size);
                }
                MPI_Finalize ();
                return 0;
}
```

```
[test@phi Training]$ export OFFLOAD_REPORT=1
[test@phi Training]$ mpiicc offload_example2.c
[test@phi Training]$ mpirun -np 2 ./a.out
Hello world! Processor= 'phi' 16 cores by 1 rank / 2 size
Hello world! Processor= 'phi' 16 cores by 0 rank / 2 size
[Offload] [MIC 0] [File]                    offload_example2.c
[Offload] [MIC 0] [Line]                    43
[Offload] [MIC 0] [Tag]                     Tag 0
[Offload] [HOST]  [Tag 0] [CPU Time]        0.193874(seconds)
[Offload] [MIC 0] [Tag 0] [MIC Time]        0.000463(seconds)

[Offload] [MIC 0] [File]                    offload_example2.c
[Offload] [MIC 0] [Line]                    43
[Offload] [MIC 0] [Tag]                     Tag 0
[Offload] [HOST]  [Tag 0] [CPU Time]        0.280292(seconds)
[Offload] [MIC 0] [Tag 0] [MIC Time]        0.000433(seconds)

Hello world! Running on co-processor = 'phi-mic0' with 228 logical cores by 1 rank / 2 size
Hello world! Running on co-processor = 'phi-mic0' with 228 logical cores by 0 rank / 2 size
[test@phi Training]$
```

```
[test@phi Training]$ export OFFLOAD_REPORT=2
[test@phi Training]$ mpiicc offload_example2.c
[test@phi Training]$ mpirun -np 2 ./a.out
Hello world! Processor= 'phi' 16 cores by 1 rank / 2 size
Hello world! Processor= 'phi' 16 cores by 0 rank / 2 size
[Offload] [MIC 0] [File]                    offload_example2.c
[Offload] [MIC 0] [Line]                    43
[Offload] [MIC 0] [Tag]                     Tag 0
[Offload] [HOST]  [Tag 0] [CPU Time]        0.196740(seconds)
[Offload] [MIC 0] [Tag 0] [CPU->MIC Data]   1036 (bytes)
[Offload] [MIC 0] [Tag 0] [MIC Time]        0.000442(seconds)
[Offload] [MIC 0] [Tag 0] [MIC->CPU Data]   1028 (bytes)

[Offload] [MIC 0] [File]                    offload_example2.c
[Offload] [MIC 0] [Line]                    43
[Offload] [MIC 0] [Tag]                     Tag 0
[Offload] [HOST]  [Tag 0] [CPU Time]        0.279399(seconds)
[Offload] [MIC 0] [Tag 0] [CPU->MIC Data]   1036 (bytes)
[Offload] [MIC 0] [Tag 0] [MIC Time]        0.000449(seconds)
[Offload] [MIC 0] [Tag 0] [MIC->CPU Data]   1028 (bytes)

Hello world! Running on co-processor = 'phi-mic0' with 228 logical cores by 1 rank / 2 size
Hello world! Running on co-processor = 'phi-mic0' with 228 logical cores by 0 rank / 2 size
[test@phi Training]$
```

Offloading Functions

* User functions have to be qualified in one of two ways before called within the offload block.
  - Using __attribute__(target(mic)) :

```
__attribute__(target(mic)) foo_1()
{
        printf("I am foo_1\n");
}

__attribute__(target(mic)) foo_2()
{
        printf("I am foo_2\n");
}
```

  - Using #Pragma offload_attribute(push,target(mic)) and #pragma offload_attribute(pop) :

```
#pragma offload_attribute(push, target(mic))
__attribute__(target(mic)) foo_1() {
        printf("I am foo_1\n");
}
__attribute__(target(mic)) foo_2() {
        printf("I am foo_2\n");
}
#pragma offload_attribute(pop)
```

Offloading functions(Cont..)

* Offloading would be done in the following way in sequential program.

```
int main(int argc, char * argv[] )
{
        printf(”On Host. \n”);
        #pragma offload target(mic)
        {
                foo_1();
                foo_2();
        } //end of offload block
printf(”Done\n");
} //end of main()
```

Data transfer to Coprocessor

```
const int size = 100;
int array[size];
#pragma offload target(mic)
for (int ii=0; ii < size; ii++) {
        data[ii] = 0;
}
```

- Local Scalar Variables and Arrays of known size are automatically transferred
  - Dynamically allocated array pointer, has to employ pragma-offload with size specification

```
const int size = 100;
int *array = malloc(size*sizeof(int));
#pragma offload target(mic) inout(array: length(size))
for (int ii=0; ii < size; ii++) {
        data[ii] = 0;
}
```

  - Global scope and static variables have to be declared with attribute qualifier

*int * __attribute__(target(mic)) array*

static __attribute__(target(mic)) int size;

Data transfer to Coprocessor (cont…)

* Direction and attributes of data transfer can be controlled with in pragma-offload
  * In/out – Indicates that date needs to be sent into the coprocessor or out-of the coprocessor

```
#pragma offload target(mic) in(m1, m2 : length(SIZE)) out(k1, k2 : length(SIZE1))
```

  * Inout – data passed to and from the coprocessor
  * Nocopy – data need not be transferred

* Data can be transferred without computation using pragma-offload_transfer

```
#pragma offload_transfer target(mic) inout(m1, m2 : length(SIZE))
```

Data transfer – Data persistence

* Alloc_if and free_if
  * Preserves allocation and data between offloads
  * By default pointer and data reallocated on coprocessor between offload

```
#pragma offload_transfer target(mic) inout(m1, m2 : length(SIZE)) \
                alloc_if(FLAG==0) free_if(FLAG==1)
```

OR


```
#pragma offload target(mic) \
```

Data transfer – Asynchronous Transfer

* signal() wait()
  * Effects a synchronous data transfer between host and coprocessor
  * Its used in conjunction with offload or offload_transfer or offload_wait
  * Effects non-blocking transfer and frees the host for execution

```
#pragma offload_transfer target(mic) inout(m1 : length(SIZE)) \
        signal(m1) alloc_if(FLAG==0)
.
.
#pragma offload target(mic) wait(m1)
{
    .
```

Virtual Shared Memory Model

* Emulates memory sharing between multi and many core, on a single system
* Only available for C and C++
* Eliminates need for data marshaling
* It's a run time user library, enabling virtual-sharing of memory even for complex data types
* Synchronization upon change, allocates same virtual address
* Programmer specifies data to be shared and synchronized
  * Mark Variables/Classes to be shared
  * Same Variable used on multi and many core machines
  * Coherence between the 2 copies of variables automatically maintained

Virtual Shared Memory Model (Cont.)

* _Cilk_shared
  * Function used to mark variables shared between host and co-processor
  * Virtual address of the marked variables on host and co-processor is same

* _Cilk_offload
  * Offload function call, invokes work on co-processor
  * Marks start and end of synchronization for shared variables

Example

```
#include<stdio.h>
#include<unistd.h>
_Cilk_shared char hostname2[1024];
_Cilk_shared int cores2;
_Cilk_shared void get_name() {
#ifdef __MIC__
        gethostname(hostname2, 1023);
cores2 = sysconf(_SC_NPROCESSORS_ONLN);
                printf("Hello world! Running on co-processor = '%s' with %d logical cores\n",hostname2, cores2);
                printf("Address of cores2 on co-processor :  %p/n", &cores2);

#else
         printf("No co-processor avaliable !\n");
#endif
};
void main()
{               char hostname1[1024];
```

Dynamic Memory

```
int * _Cilk_shared data_array;
_Cilk Shared void foo(); {
            \\ Some computation with data_array
               printf("Address of cores2 on co-processor :  %p/n", &data_array[0]);
}
main ()
 {          data_array = (_Cilk_shared int *) _offload_shared_malloc(ARRAYSIZE*sizeof(int));
            printf("Address of cores2 on co-processor :  %p/n", &data_array[0]);
            _Cilk_ofload foo();
            _offload_shared_free(data_array);
};
```

Sharing Class

* Virtual Shared Memory Model allows sharing of complex data types like class and structures, which are not bit-wise copyable
* Regular usage of *new* and calling constructor is not applicable in this case
* Placement version of operator new is to be used with _offload_shared_malloc to create *Virtual Shared Class*
* Include header file <new>

* The placement version of new makes sure only constructor is called and

Sharing Class (Cont.)

```
class _Cilk_shared ExampleClass {

        // Class body
};

ExampleClass * _Cilk_shared sharedClass;

main() {
        int size = sizeof(ExampleClass);
        _Cilk_shared ExampleClass * address = (_Cilk_shared ExampleClass *)
_offload_shared_malloc(size);
        sharedClass = new(address) ExampleClass;

        .
        sharedClass->set(...);
        _Cilk_offload sharedClass->print();
        sharedClass->print();
        .
        .
        _Offload_shared_free(address);

};
```

Virtual Shared Memory Mode

1. _Cilk_shared
2. _Cilk_offload
3. _Offload_shared_malloc and _Offload_shared_free
4. _Offload_shared_aligned_malloc and free
5. new (placement version)

```
__attribute__((target(mic))) int* dummy;

int main()
{
    int num_of_xeonphi = _offload_number_of_devices();
    dummy = (int*) malloc(num_of_xeonphi*sizeof(int));

    #pragma omp parallel for
    for(int ii=0; ii < num_of_xeonphi; ii++)
    {
        #pragma offload target(mic:i)
inout(dummy[ii:1])
        dummy[ii] = 100;
    } //end of parallel for
} //end of main()
```

```
__attribute__((target(mic))) int* dummy;

int main()
{
        int num_of_xeonphi = _offload_number_of_devices();
        dummy = (int*) malloc(num_of_xeonphi*sizeof(int));

        for(int ii=0; ii<num_of_xeonphi; ii++)
        {
           #pragma offload target(mic:ii) inout(dummy[ii:1]) signal\
                        (&dummy[ii])
                dummy[ii] = 100;
        }

        for(int jj=0; jj<num_of_xeonphi; jj++)
        {
         #pragma offload_wait target(mic:jj) wait(&dummy[jj])
        }
} //end of main()
```

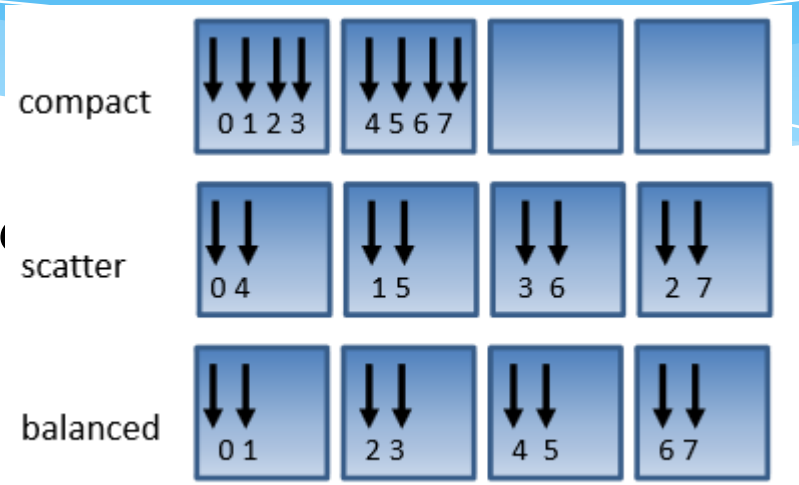
Native on Xeon-Phi with MYO

* _Cilk_offload_to(int i) foo() – Enables offload to the specified co-processor
* _Cilk_spawn _Cilk_offload_to(cnt) foo()
  * Spawn is part of Intel Cilk Library

Thread placement

* To place threads set KMP_AFFINITY=<Type>,<Round Robin Threads Type>
  - By default KMP_AFFINITY=balanced

<type>	Description
compact	Packs threads close to each other
scatter	Round Robin threads to cores
balanced	Keep OMP ids consecutive (MIC only)
explicit	use proclist modifier to pin threads
none	Does not pin threads



```
#include<stdio.h>
#include<unistd.h>
#include<omp.h>
void main()
{          char hostname1[1024],hostname2[1024];
          gethostname(hostname1, 1023);
          int maxthreads2, maxthreads1=omp_get_max_threads();
          printf("Hello world! Running on processor = '%s' with
%d maxthreads\n", hostname1,maxthreads1);
          #pragma omp parallel
          {     int threadid;
                threadid = omp_get_thread_num();
                 printf("%d thread is offloading\n",threadid);
                #pragma offload target(mic:0)
               {  gethostname(hostname2, 1023);
                              maxthreads2=omp_get_max_threads();
     printf("Hello world! Running on co-processor = '%s' with
%d threads by %d thread\n",hostname2,maxthreads2, threadid);
}
          }
}
```

Thread Offload Example

```
[Training ~]$  icc thread_offload.c -openmp
[Training ~]$ ./a.out

Hello world! Running on processor = 'ihpc.calligotech.com' with 12 maxthreads
0 thread is offloading
4 thread is offloading
6 thread is offloading
3 thread is offloading
5 thread is offloading
1 thread is offloading
2 thread is offloading
7 thread is offloading
8 thread is offloading
9 thread is offloading
10 thread is offloading
11 thread is offloading
Hello world! Running on co-processor = 'ihpc-mic0.calligotech.com' with 224 threads by 0 thread
Hello world! Running on co-processor = 'ihpc-mic0.calligotech.com' with 224 threads by 2 thread
Hello world! Running on co-processor = 'ihpc-mic0.calligotech.com' with 224 threads by 1 thread
Hello world! Running on co-processor = 'ihpc-mic0.calligotech.com' with 224 threads by 10 thread
Hello world! Running on co-processor = 'ihpc-mic0.calligotech.com' with 224 threads by 11 thread
Hello world! Running on co-processor = 'ihpc-mic0.calligotech.com' with 224 threads by 4 thread
Hello world! Running on co-processor = 'ihpc-mic0.calligotech.com' with 224 threads by 6 thread
Hello world! Running on co-processor = 'ihpc-mic0.calligotech.com' with 224 threads by 8 thread
Hello world! Running on co-processor = 'ihpc-mic0.calligotech.com' with 224 threads by 3 thread
Hello world! Running on co-processor = 'ihpc-mic0.calligotech.com' with 224 threads by 7 thread
Hello world! Running on co-processor = 'ihpc-mic0.calligotech.com' with 224 threads by 9 thread
Hello world! Running on co-processor = 'ihpc-mic0.calligotech.com' with 224 threads by 5 thread
```