INTRODUCTION TO THE INTEL® XEON PHI™ PROCESSOR (CODENAME "KNIGHTS LANDING")

Dr. Harald Servat - HPC Software Engineer
Data Center Group – Innovation Performing and Architecture Group

Summer School in Advanced Scientific Computing 2016
February 21st, 2016 – Braga, Portugal

LEGAL DISCLAIMERS

Intel technologies features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at [intel.com].

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

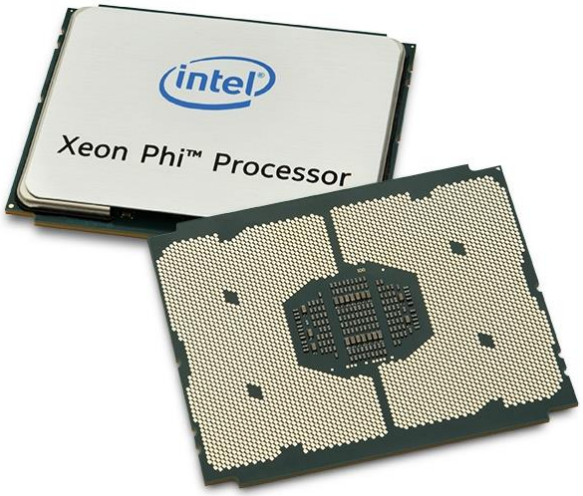
Results have been estimated or simulated using internal Intel analysis or architecture simulation or modeling, and provided to you for informational purposes. Any differences in your system hardware, software or configuration may affect your actual performance.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at https://www-ssl.intel.com/content/www/us/en/high-performance-computing/path-to-aurora.html.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.

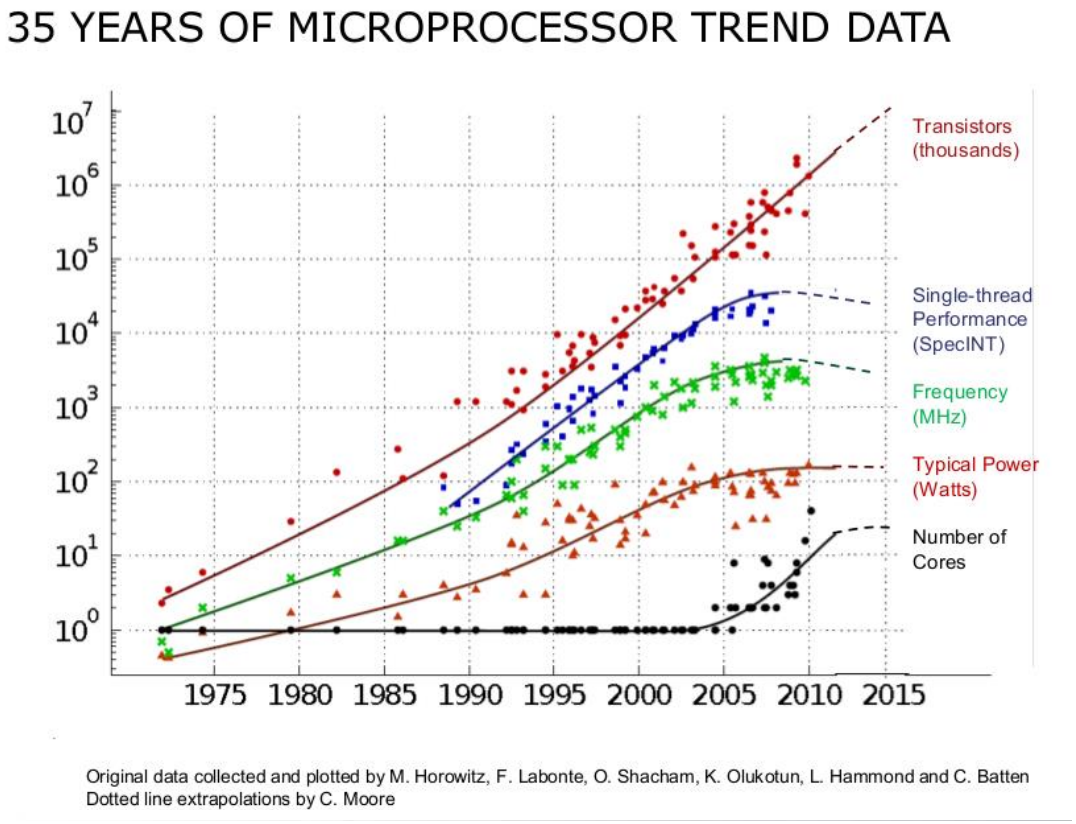
AGENDA

1. Introduction
2. Micro-architecture
   i. Tile architecture
   ii. Untile architecture
3. AVX512 Instruction set
4. High-Bandwidth memory

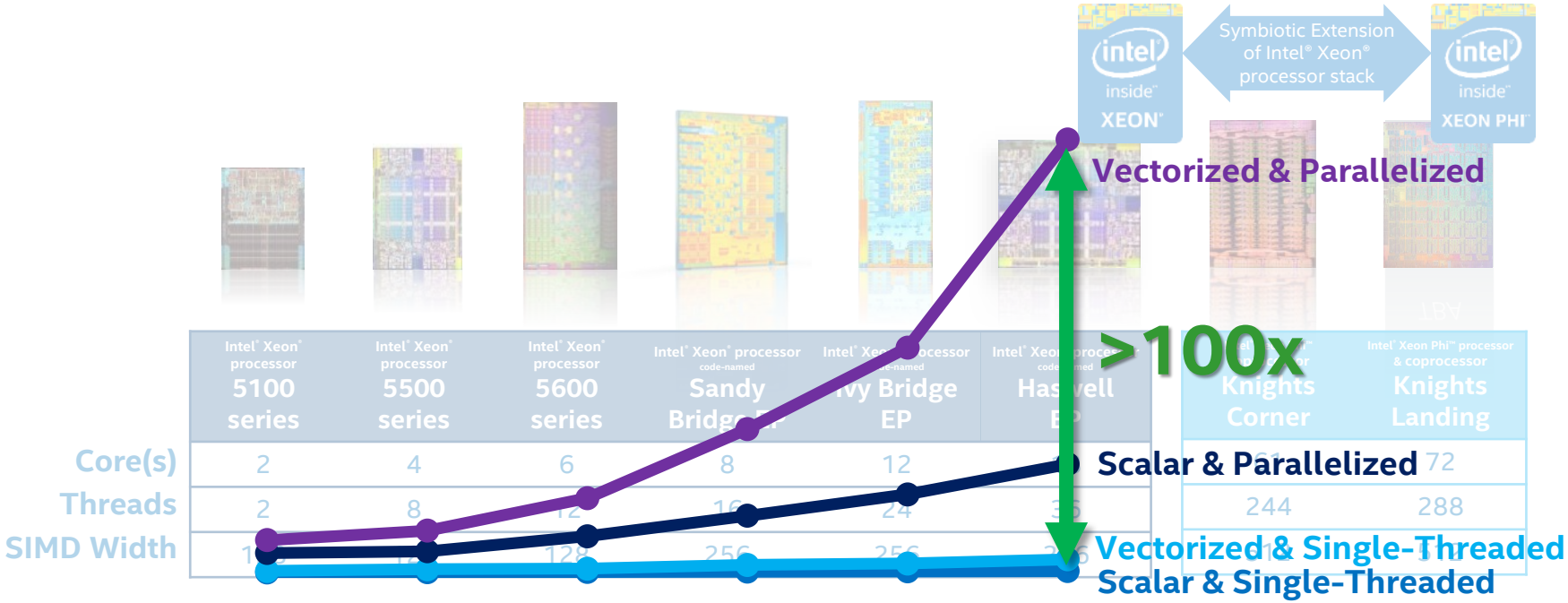


INTRODUCTION

MOORE'S LAW AND PARALLELISM



CPU PARALLELISM IS ALREADY A MUST



Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Configurations: Intel Performance Projections as of Q1 2015. For more information go to http://www.intel.com/performance . Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.
Chart illustrates relative performance of the Binomial Options DP workload running on an Intel® Xeon® processor from the adjacent generation.
Product specification for launched and shipped products available on ark.intel.com.
[1]Not launched

PARALLELISM AND PERFORMANCE

Peak GFLOP/s in Single Precision

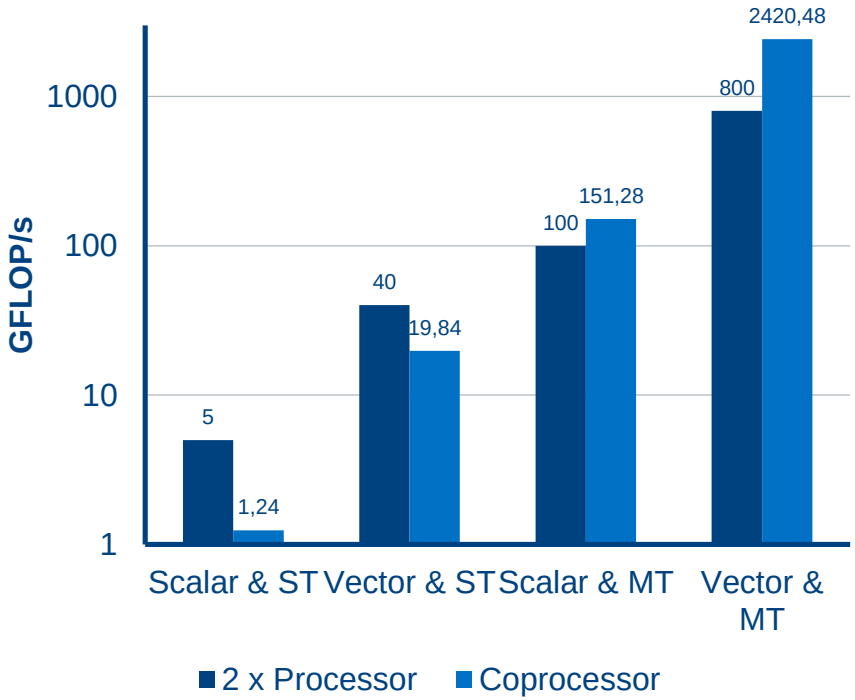
- Clock Rate x Cores x Ops/Cycle x SIMD

2 x Intel® Xeon® Processor E5-2670v2

- 2.5 GHz x 2 x 10 cores x 2 ops x 8 SIMD = 800 GFLOP/s

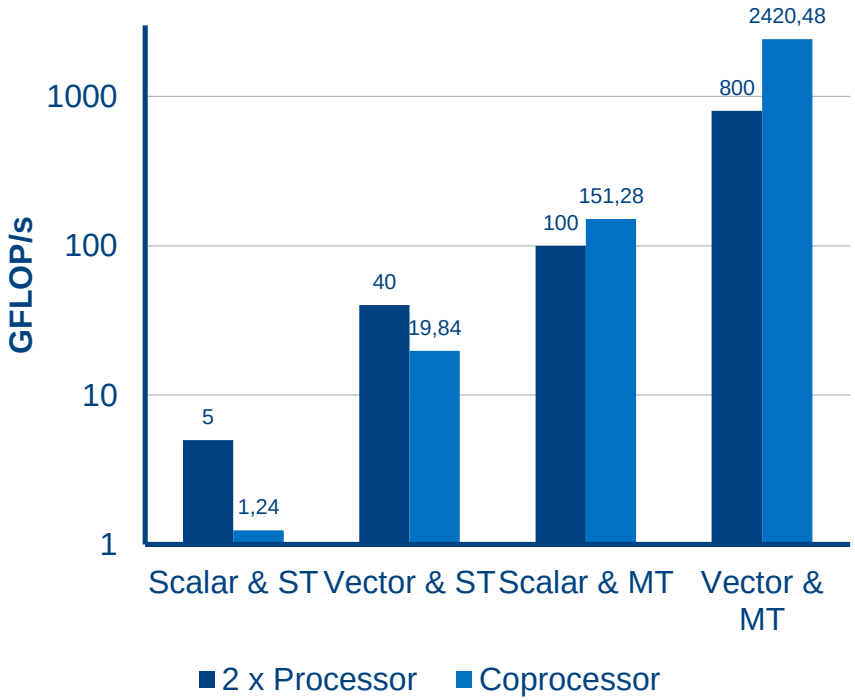
Intel® Xeon Phi™ Coprocessor 7120P

- 1.24 GHz x 61 cores x 2 ops x 16 SIMD = 2420.48 GFLOP/s

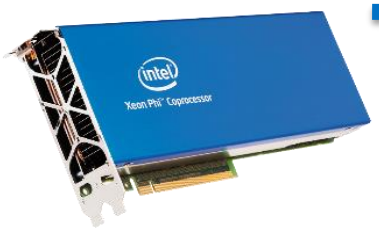


PARALLELISM AND PERFORMANCE

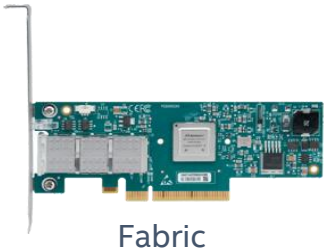
- On modern hardware, Performance = Parallelism
- Flat programming model on parallel hardware is not effective.
- Parallel programming is not optional.
- Codes need to be made parallel ("modernized") before they can be tuned for the hardware ("optimized").



A PARADIGM SHIFT



Coprocessor



Fabric

Memory

Server Processor

Memory Bandwidth
400+ GB/s STREAM

Memory Capacity
Over 25x KNC

Power Efficiency
Over 25% better than card

I/O
200 Gb/s/dir with int fabric

Cost
Less costly than discrete parts

Flexibility
Limitless configurations

Density
3+ KNL with fabric in 1U

KNIGHTS LANDING (HOST OR PCIE)

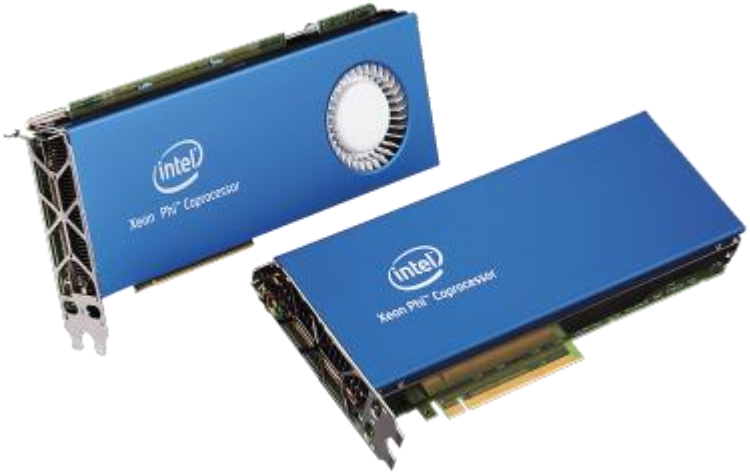
Host Processor

Host Processor
w/ integrated Fabric

Knights Landing Processors

Host Processor for Groveport Platform

Solution for future clusters with both Xeon and Xeon Phi



Knights Landing PCIe Coprocessors

Ingredient of Grantley & Purley Platforms

Solution for general purpose servers and workstations

STAMPEDE-KNL (OR STAMPEDE1.5)

Intel S7200AP Cluster

484x Intel Xeon Phi 7250 68C 1.4GHz

- 32,912 total cores
- 1,474 teraFLOP/s

Intel Omni-Path

INTEL® XEON PHI™ X200 PROCESSOR:
MICRO-ARCHITECTURE

INTEL® XEON PHI™ PROCESSOR FAMILY ARCHITECTURE OVERVIEW

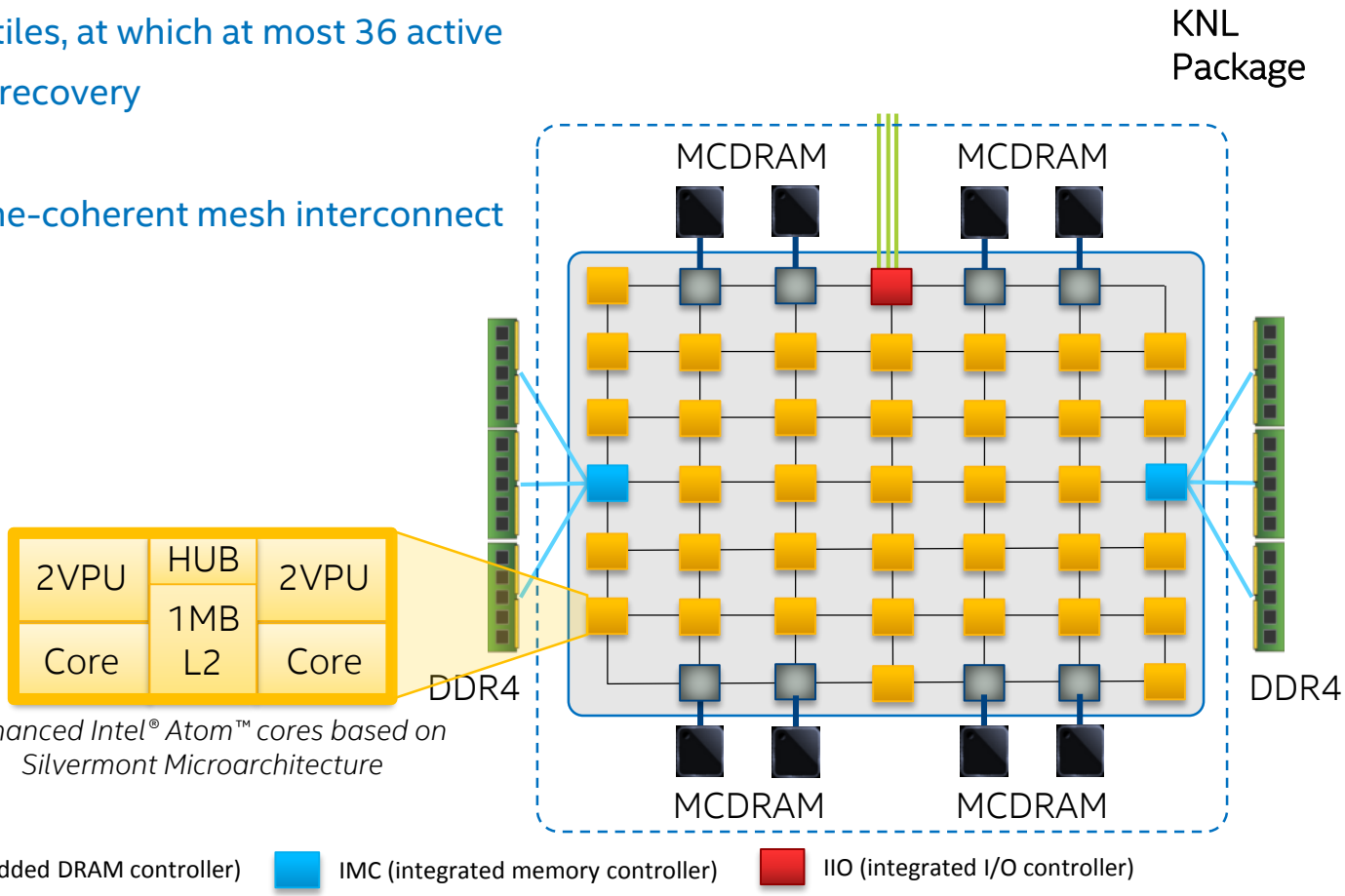
Codenamed "Knights Landing" or KNL

Comprises 38 physical tiles, at which at most 36 active

- Remaining for yield recovery

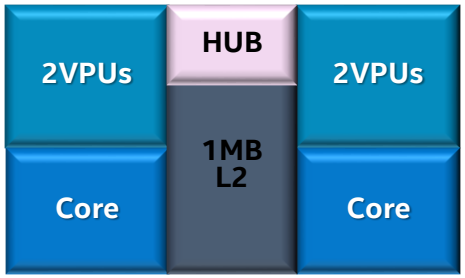
Introduces new 2D cache-coherent mesh interconnect (Untile)

- Tiles
- Memory controllers
- I/O controllers
- Other agents



TILE ARCHITECTURE

KNL PROCESSOR TILE



Tile

- 2 cores, each with 2 vector processing units (VPU)
- 1 MB L2-cache shared between the cores

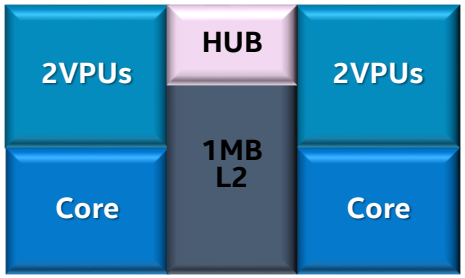
Core

- Binary compatible with Xeon
- Enhanced Silvermont (Atom)-based for HPC w/ 4 threads
- Out-of-order core
- 2-wide decode, 6-wide execute (2 int, 2 fp, 2 mem), 2-wide retire

2 VPU

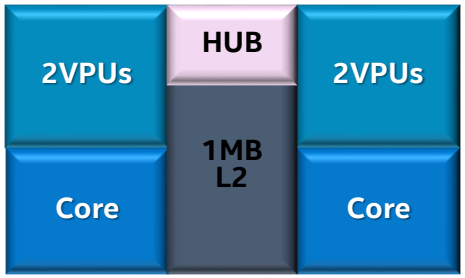
- 512-bit SIMD (AVX512) 32SP/16DP per unit
- Legacy X87, SSE, AVX and AVX2 support

KNL PROCESSOR TILE



Structure	Characteristics	
Cache	L1	32 KB 8-way Icache
		32 KB 8-way Dcache
	L2	1 MB 16-way Unified cache
TLB	L1	48-entry fully-associative ITLB
		64-entry 8-way DTLB 4KB pages
	L2	256-entry 8-way DTLB 4KB pages
		128-entry 8-way DTLB 2/4MB pages
		16-entry fully-associative DTLB 1GB pages

KNL PROCESSOR TILE



CHA Caching/**H**ome **A**gent (or HUB)

- 2D-Mesh connections for Tile
- Distributed Tag Directory to keep L2s coherent
- MESIF protocol

… More to come in the UNTILE section!

INTEL® XEON PHI™ PROCESSOR EARLY SHIP TURBO SPECS

SKU	TDP (W)	Active Tiles	Active Cores	Single Tile Turbo GHz	All Tile Turbo GHz	TDP Freq GHz	AVX Freq GHz	Mesh Freq GHz	OPIO GT/s	DDR MHz
7250	215	34	68	1.6	1.5	1.4	1.2	1.7	7.2	2400
7230	215	32	64	1.5	1.4	1.3	1.1	1.7	7.2	2400
7210	215	32	64	1.5	1.4	1.3	1.1	1.6	6.4	2133

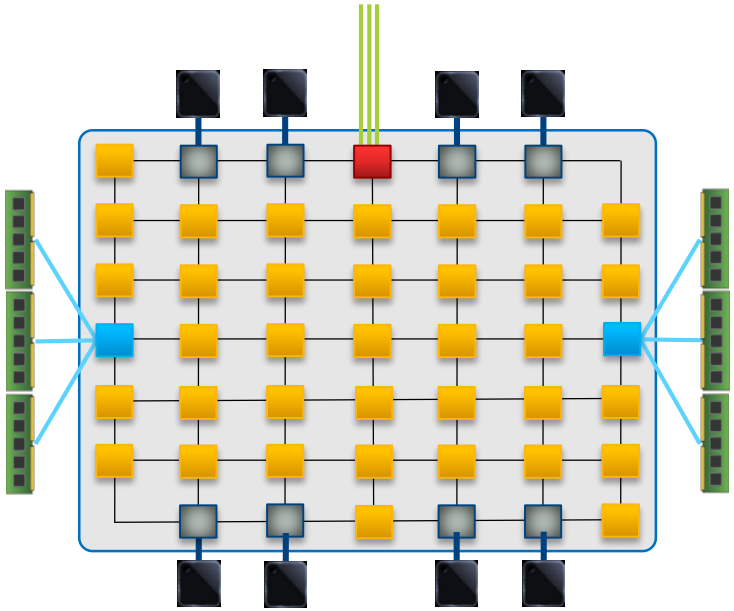
Turbo is an opportunistic increase in frequency over TDP frequency

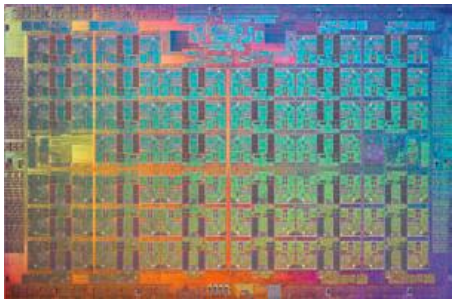
- KNL has two turbo modes
  - Single tile turbo – any one tile increases frequency while all other tiles are in the C6 idle state
  - All tile turbo – all tiles run at an increased frequency
- Frequency varies, depending on the workload, power budget and SKU
- When running AVX intense code frequency may decrease
- UNHALTED_CORE_CYCLES vs UNHALTED_REFERENCE_CYCLES performance counters

OPIO is Intel's On Package IO technology for high speed connections between multiple chips on a single package.

INTEL® XEON PHI™ X200 VS SILVERMONT COMPARISON

FEATURE	SILVERMONT	KNL
Vector ISA	Up to Intel® SSE4.2	Up to Intel® AVX-512
Enhanced (VPU) Vector processing Unit	2x 128-bit VPU / Core	2x 512-bit VPU / Core
Physical/Virtual addressing	36 bits / 48 bits	46 bits / 48 bits
HW based Gather/Scatter	No	Yes
Reorder buffer entries	32	72
Threads / Core	1	4
Memory operations per cycle	1 (16 bytes each)	2 (64 bytes each)
Vector/FP reservation station policy	In-Order	Out-of-Order
L1 cache size	24K	32K
L2 cache to D-cache BW	1X	2X
Micro-TLB	32	64
Data TLB	4K pages: 128 2M pages: 16 1G pages: N/A	4K pages: 256 2M pages: 128 1G pages: 16





KNIGHTS LANDING VS. KNIGHTS CORNER FEATURE COMPARISON

FEATURE	INTEL® XEON PHI™ COPROCESSOR 7120P	KNIGHTS LANDING PRODUCT FAMILY
Processor Cores	Up to 61 enhanced P54C Cores	Up to 72 enhanced Silvermont cores
Key Core Features	In order 4 threads / core (back-to-back scheduling restriction) 2 wide	Out of order 4 threads / core 2 wide
Peak FLOPS[1]	SP: 2.416 TFLOPs • DP: 1.208 TFLOPs	Up to 3x higher
Scalar Performance[1]	1X	Up to 3x higher
Vector ISA	x87, (no Intel® SSE or MMX™), Intel IMIC	x87, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, Intel® AVX, AVX2, AVX-512 (no Intel® TSX)
Interprocessor Bus	Bidirectional Ring Interconnect	Mesh of Rings Interconnect

1- Results have been estimated or simulated using internal Intel analysis or architecture simulation or modeling, and provided to you for informational purposes.
Any differences in your system hardware, software or configuration may affect your actual performance.

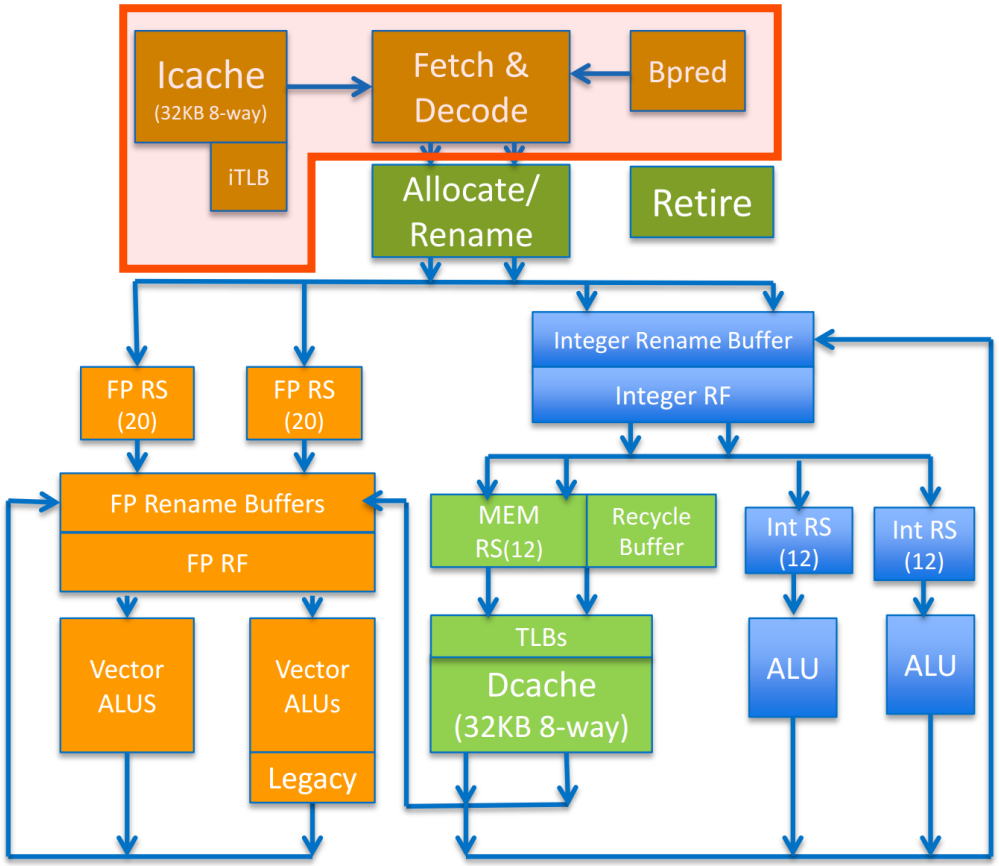
KNL CORE ORGANIZATION

Front-End Unit (FEU)

Decode, allocate 2 instructions/cycle

32-entry instruction queue

Gskew-style branch predictor



KNL CORE ORGANIZATION

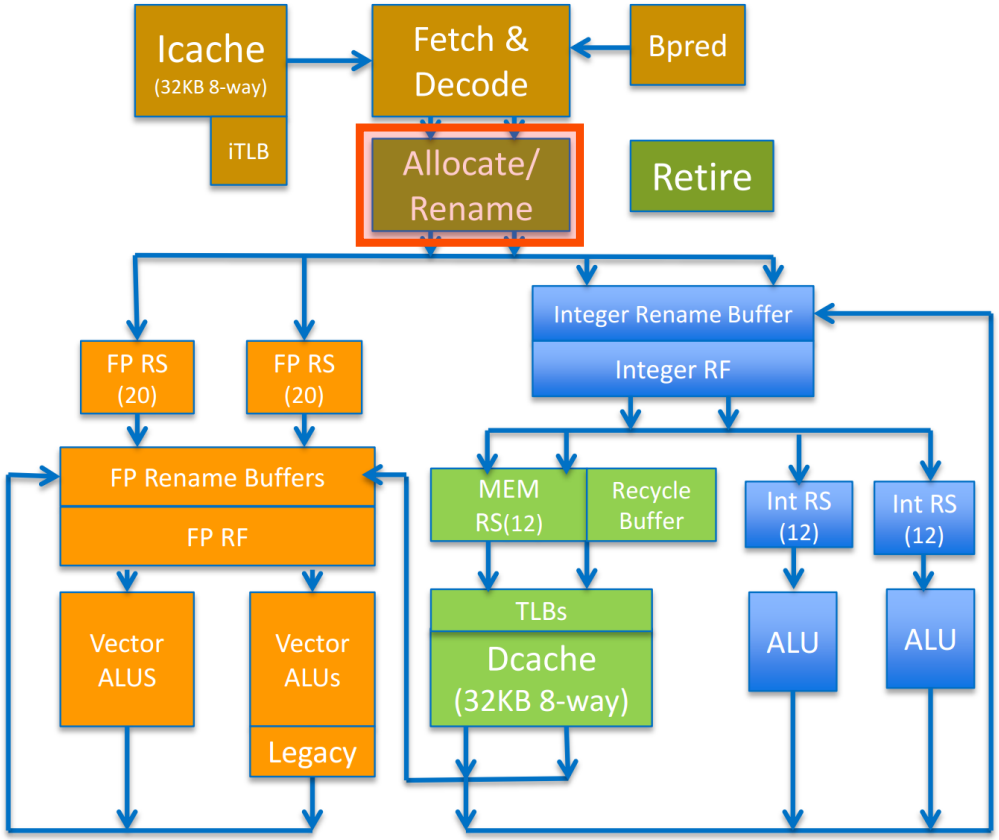
Allocation Unit

72-entry ROB buffer

72-entry rename buffers

16 store data buffers

4 gather scatter data tables



KNL CORE ORGANIZATION

Integer Execution Unit (IEU)

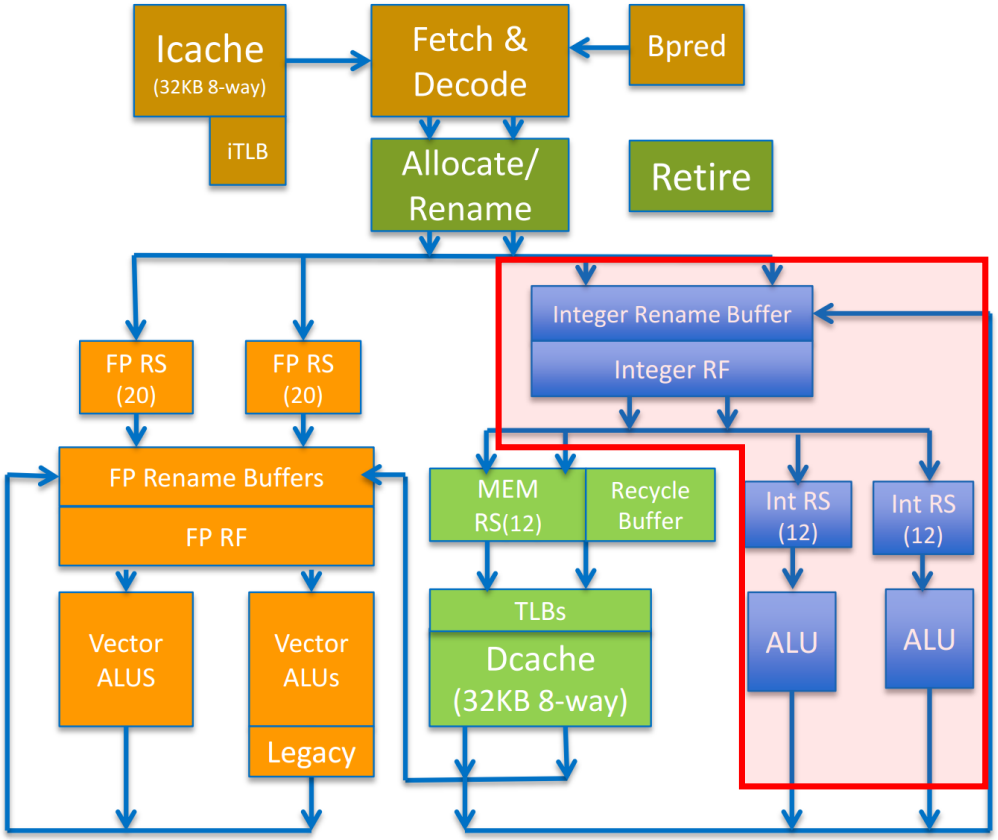
2 IEUs per core

- 2 uops dispatched / cycle
- 12-entries each

Out-of-order

Most operations take 1 cycle

- Some operations take 3-5 and are supported on only one IEU (e.g muls)



KNL CORE ORGANIZATION

Memory Execution Unit (MEU)

Dispatches 2 uops (either LD/ST)

- in-order
- but can complete in any order

2 64B load & 1 64B store port for Dcache

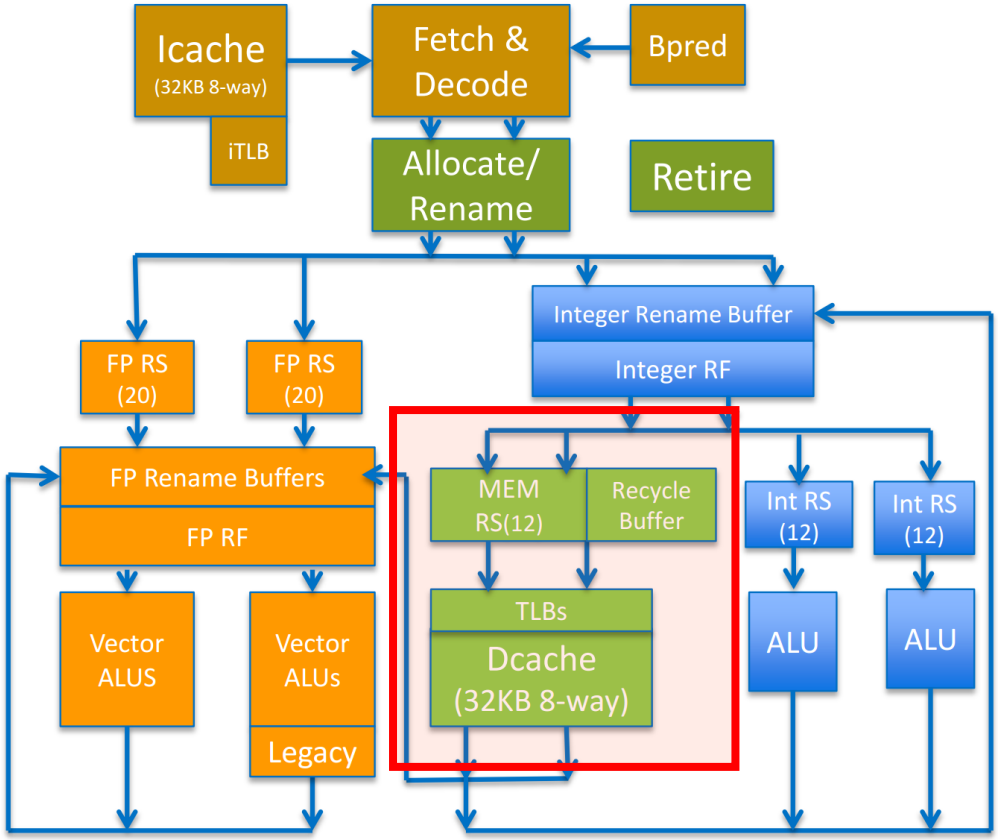
L2 supports 1 Line Read and ½ Line Write per cycle

L1 - L2 prefetcher

- Track up to 48 access patterns

Fast unaligned and cache-line split support

Fast gather/scatter support



KNL CORE ORGANIZATION

Vector Processing Unit (VPU)

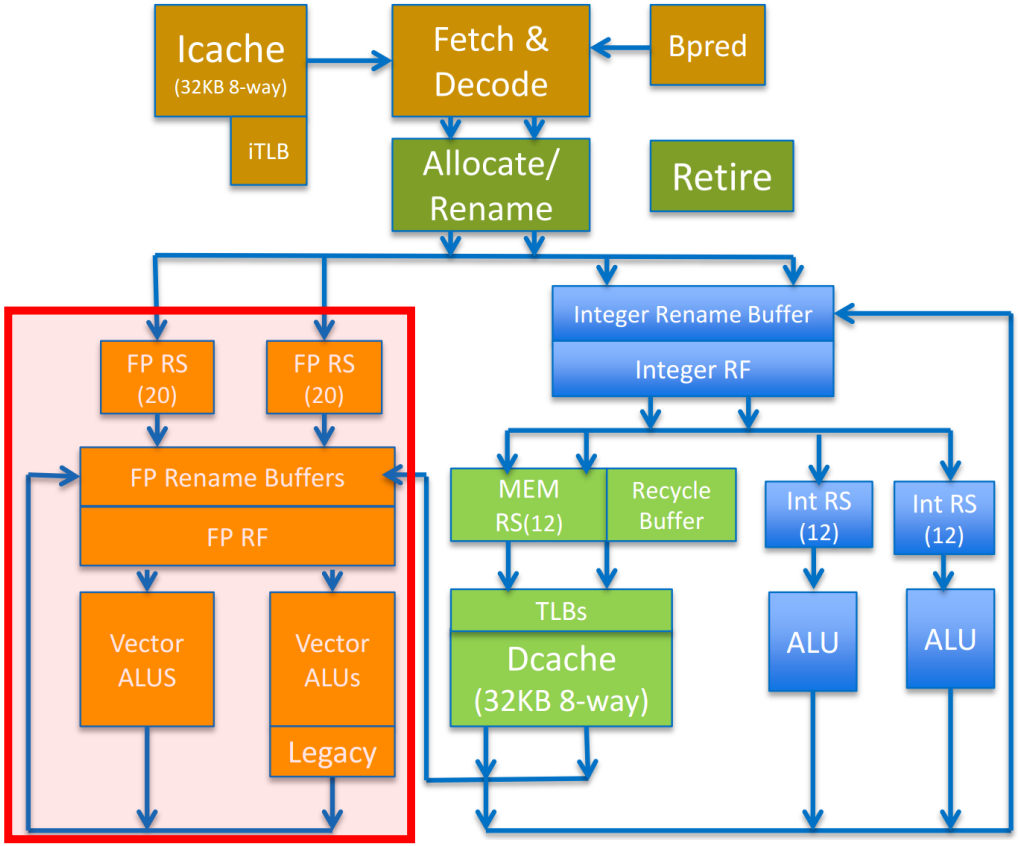
2 VPUs tightly integrated with core pipeline

- 20-entry FP RS
- executed out-of-order

2 512-bit FMA / cycle

Most FP operations take 6 cycles

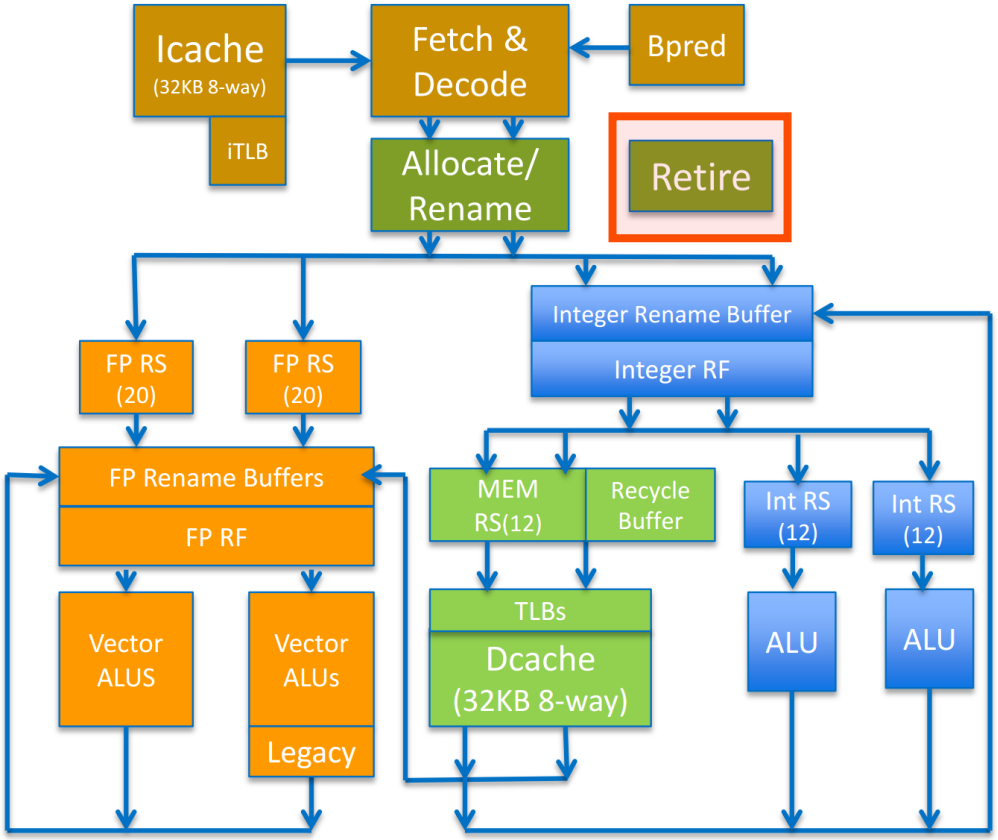
1 VPU provides legacy x87, MMX support a subset of SSE instructions



KNL CORE ORGANIZATION

Retire

2 instructions / cycle



KNL HARDWARE THREADING

4 threads per core SMT

Resources dynamically partitioned

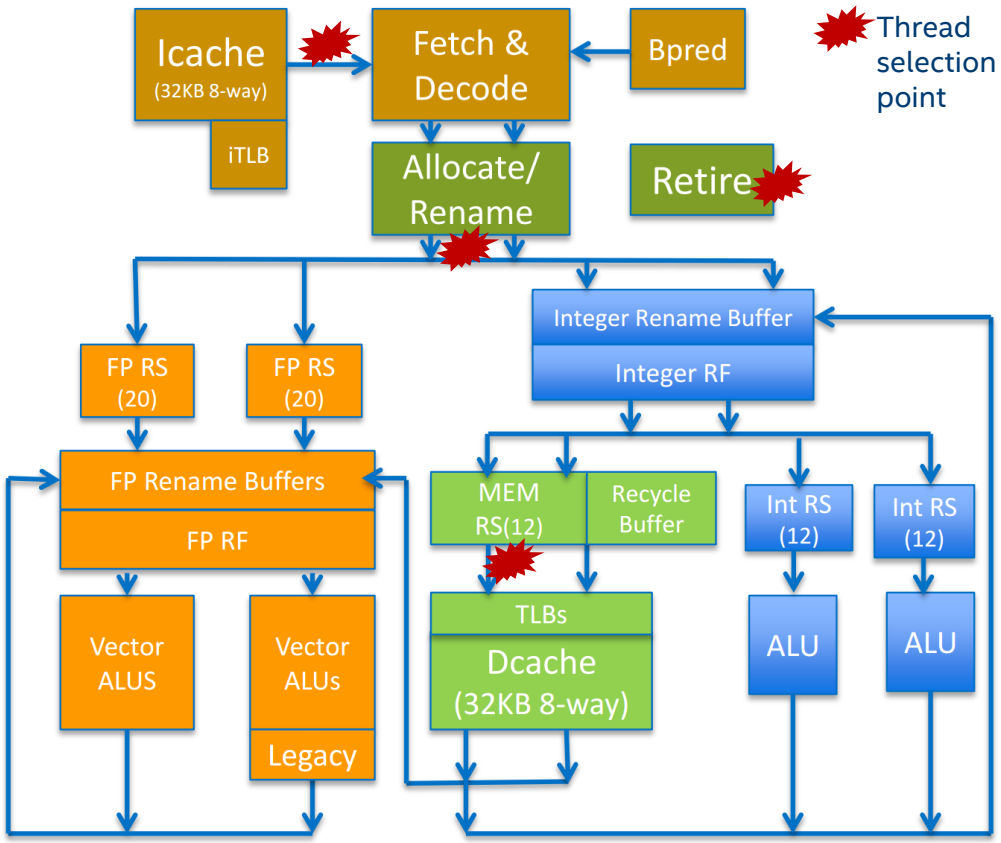
- Re-order buffer, Rename buffers, Reservation station
- Partitioning changes as threads wake up and go to sleep

Resources shared

- Caches
- TLB

Several Thread Selection points in the pipeline (✸)

- Maximize throughput while being fair
- Account for available resources, stalls and forwards progress



TAKING BENEFIT OF THE CORE

Threading

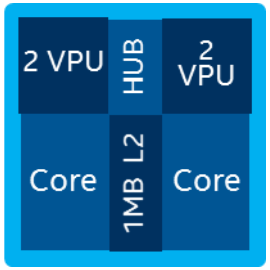
- Ensure that thread affinities are set.
- Understand affinity and how it affects your application (i.e. which threads share data?).
- Understand how threads share core resources.
  - An individual thread has the highest performance when running alone in a core.
  - Running 2 or 4 threads in a core may result in higher per core performance but lower per thread performance.
  - Due to resource partitioning, 3 thread configuration will have fewer aggregative resources than 1, 2 or 4 threads per core. 3 threads in a core is unlikely to perform better than 2 or 4 threads.

Vectorization

- Prefer AVX512 instructions and avoid mixing SSE, AVX and AVX512 instructions.
- Avoid cache-line splits; align data structures to 64 bytes.
- Avoid gathers/scatters; replace with shuffles/permutes for known sequences.
- Use hardware trascendentals (fast-math) whenever possible.
- AVX512 achieves best performance when not using masking
- KNC intrinsic code is unlikely to generate optimal KNL code, recompile from HL language.

DATA LOCALITY: NESTED PARALLELISM

- Recall that KNL cores are grouped into tiles, with two cores sharing an L2.
- Effective capacity depends on locality:
  - 2 cores sharing no data => 2 x 512 KB
  - 2 cores sharing all data => 1 x 1 MB
- Ensuring good locality (e.g. through blocking or nested parallelism) is likely to improve performance.




```
#pragma omp parallel for num_threads(ntiles)
for (int i = 0; i < N; ++i)
{
      #pragma omp parallel for num_threads(8)
      for (int j = 0; j < M; ++j)
      {
            …
      }
}
```

UNTILE ARCHITECTURE

KNL PROCESSOR UNTILE

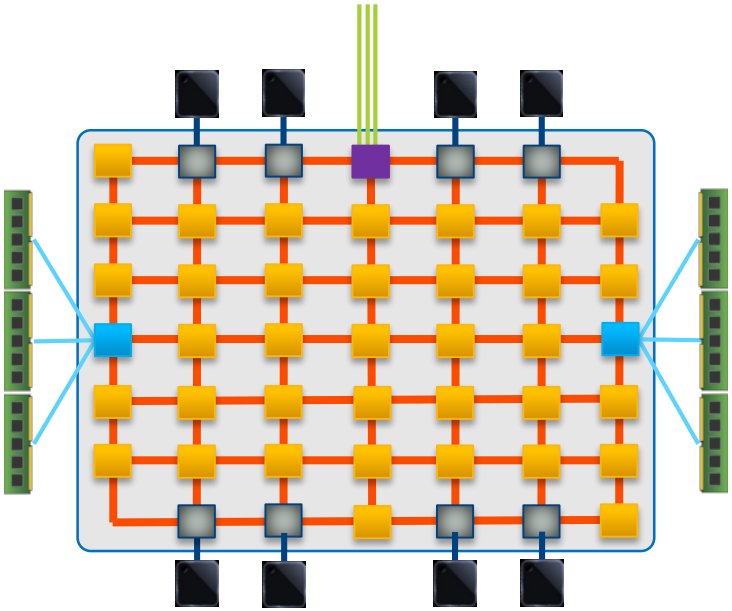
Comprises a mesh connecting the tiles (in red) with the MCDRAM and DDR memories.

- Also with I/O controllers and other agents

Caching Home Agent (CHA) holds portion of the distributed tag directory and serves as connection point between tile and mesh

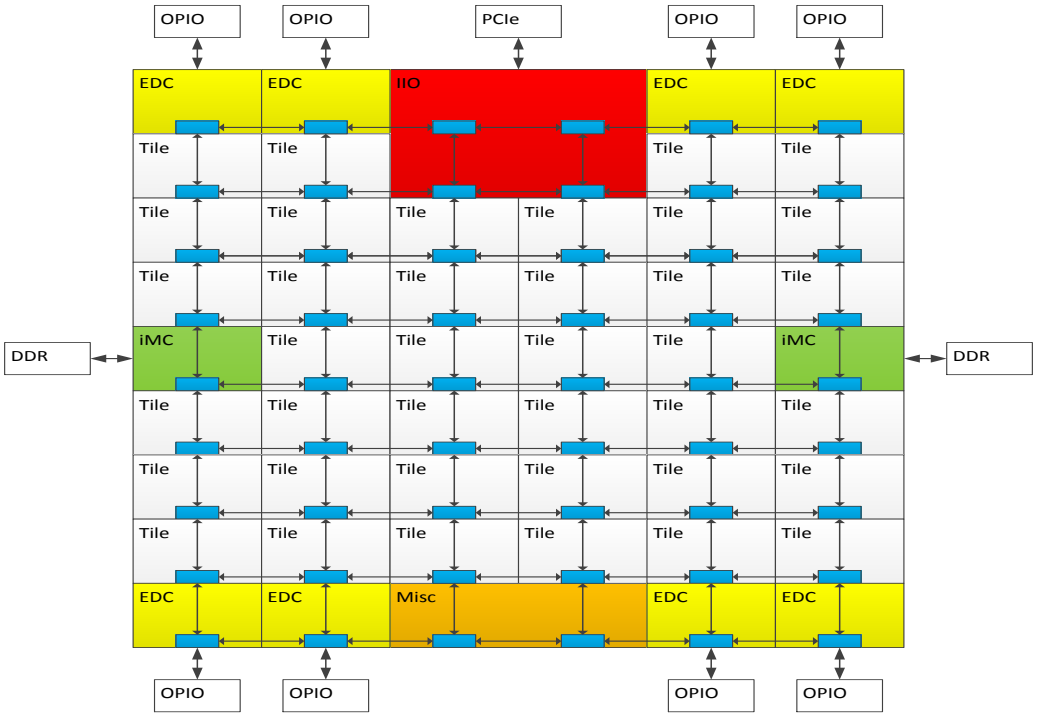
- No L3 cache as in Xeon

Cache coherence uses MESIF protocol (Modified, Exclusive, Shared, Invalid, Forward)



Tile | EDC (embedded DRAM controller) | IMC (integrated memory controller) | IIO (integrated I/O controller)

KNL MESH INTERCONNECT



Mesh of Rings

- Every row and column is a ring
- YX routing: Go in Y → Turn → Go in X
  - 1 cycle to go in Y, 2 cycles to go in X
- Messages arbitrate at injection and on turn

Mesh at fixed frequency of 1.7 GHz

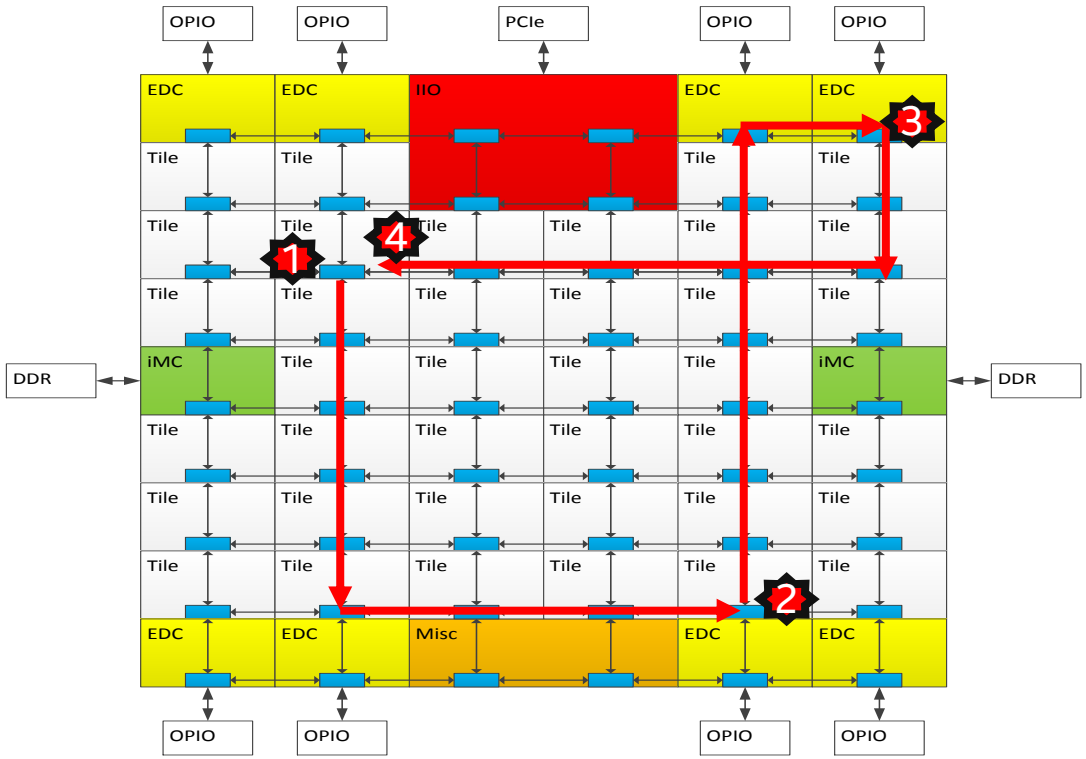
Distributed Directory Coherence protocol

KNL supports Three Cluster Modes

1) All-to-all
2) Quadrant
3) Sub-NUMA Clustering

Selection done at boot time.

CLUSTER MODE: ALL-TO-ALL



Address uniformly hashed across all distributed directories

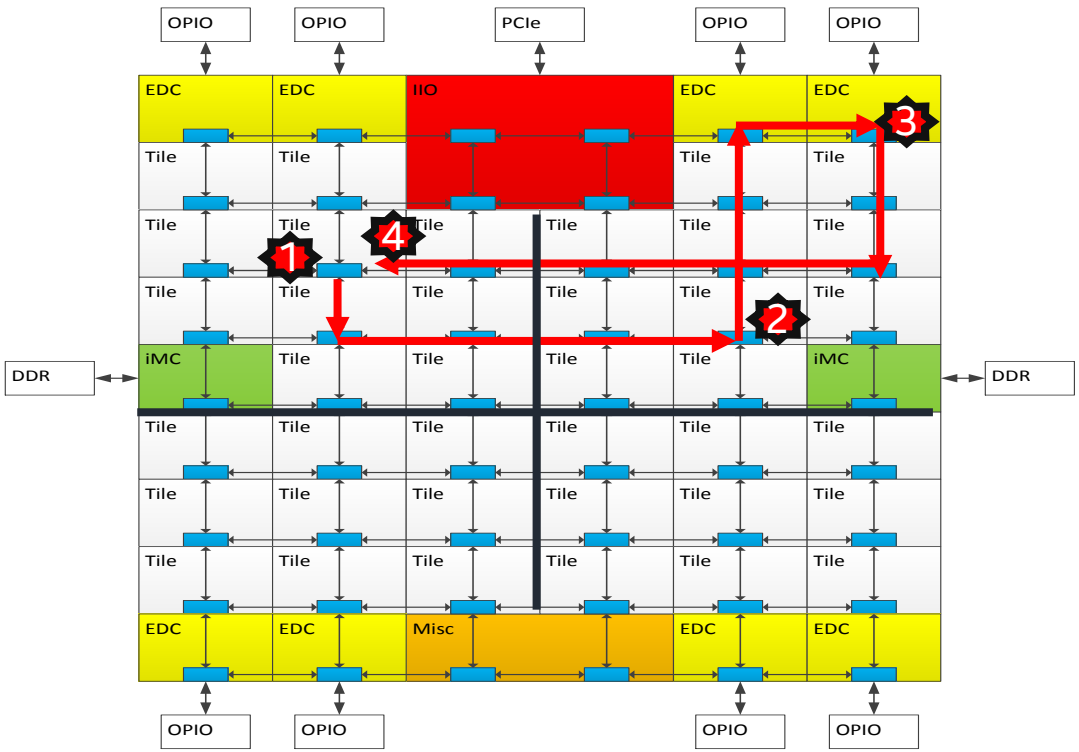
No affinity between Tile, Directory and Memory

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 Quadrants

Affinity between the Directory and Memory

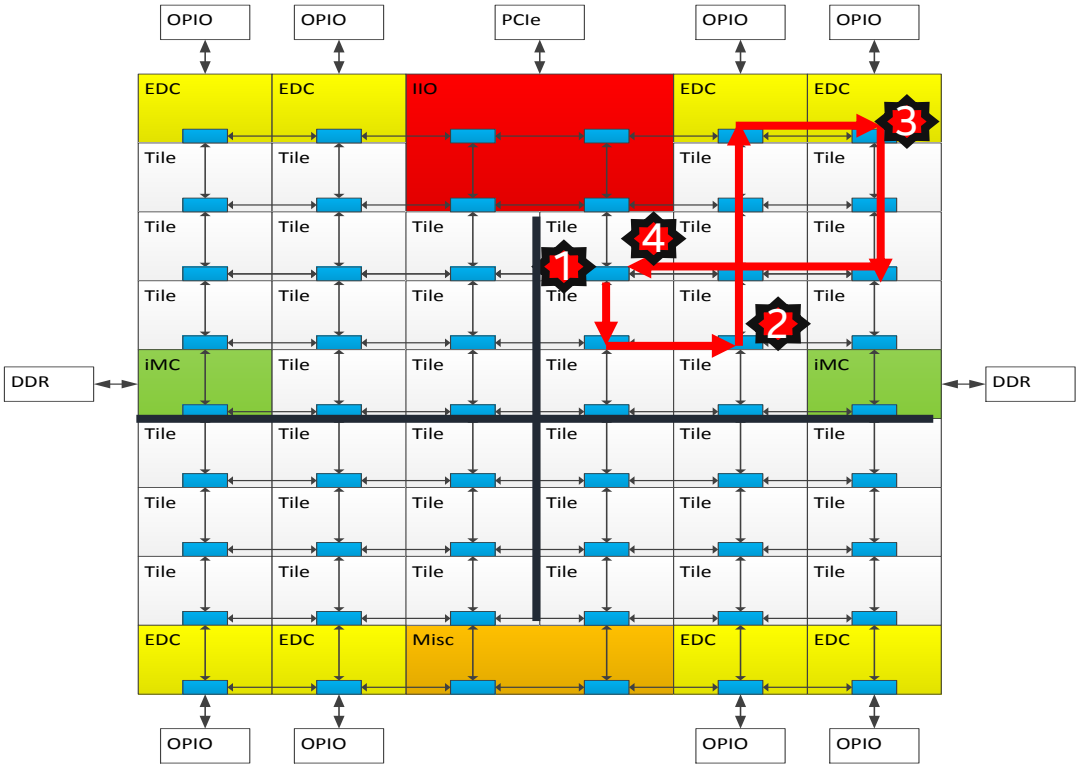
Lower latency and higher BW than all-to-all

SW Transparent

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: SUB-NUMA CLUSTERING (SNC4)



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

Analogous to 4-socket Xeon

SW Visible

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

HOW TO DETECT / USE THE CLUSTER MODES?

Detection

- CPUID instruction
  - `/proc/cpuinfo`
- hwloc command
  - `lstopo -no-io`
- numactl / libnuma

Use

- numactl / libnuma
- Memkind
- MPI/OpenMP

Diagram for Xeon
2-socket 8-core w/ HT and 16GB per socket

IMPLICATIONS FOR PARALLEL RUNTIMES

OpenMP

- No changes for All-2-All or Quadrant modes
- In SNC4 and using multiple MPI ranks per processor, use descriptors
  - compact, scatter
- In SNC4 with no MPI, need to manually handle NUMA bindings

MPI

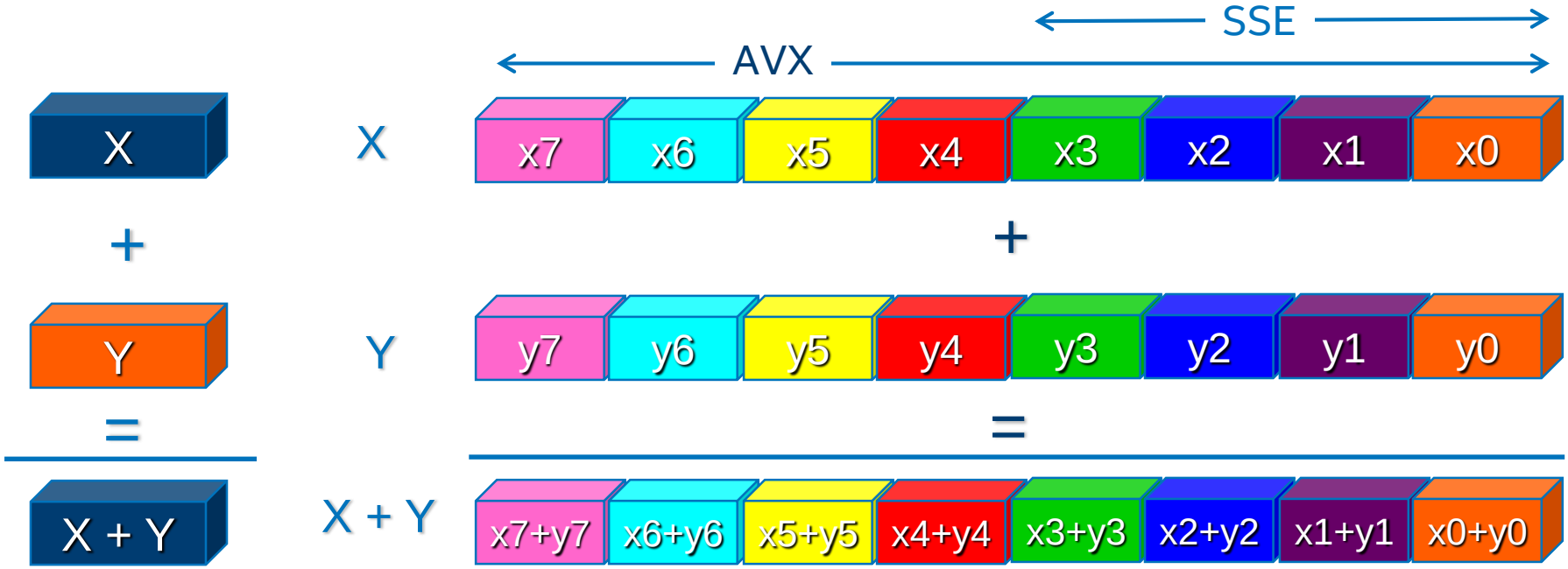
- Use existing (Intel) MPI mechanisms for affinity control
  - I_MPI_PIN, I_MPI_PIN_MODE, I_MPI_PIN_PROCESSOR_LIST, I_MPI_PIN_DOMAIN
- Don't limit yourself to 1 MPI Rank per SNC

INTEL® XEON PHI™ X200 PROCESSOR:
AVX512 INSTRUCTION SET

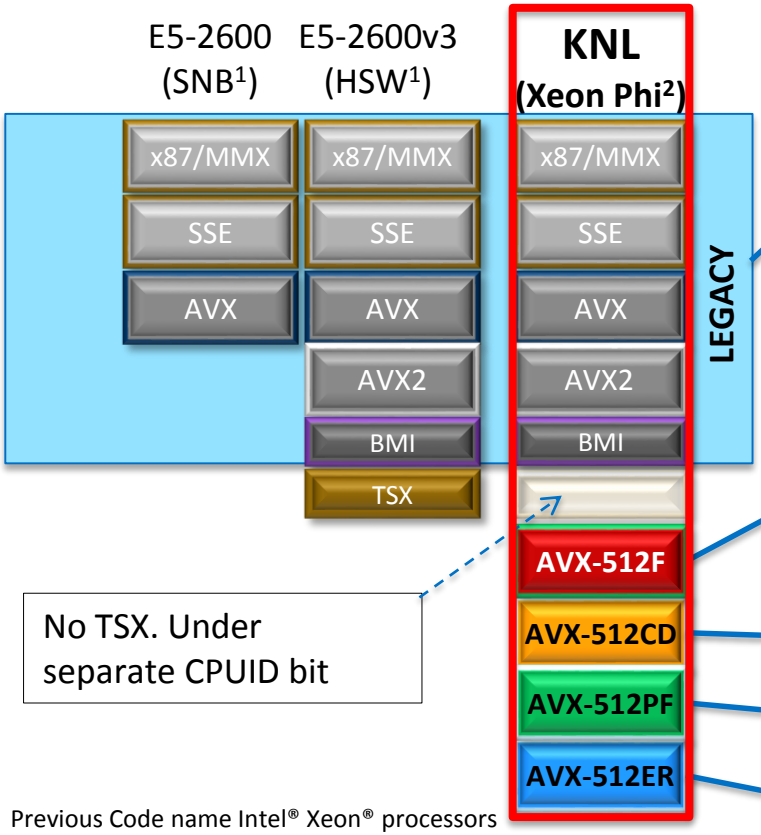
SIMD: SINGLE INSTRUCTION, MULTIPLE DATA

```
for (i=0;  i<n;  i++)
    z[i] = x[i] + y[i];
```

- Scalar mode
  - one instruction produces one result
  - E.g. vaddss, (vaddsd)
- Vector (SIMD) mode
  - one instruction can produce multiple results
  - E.g. vaddps, (vaddpd)



KNL HARDWARE INSTRUCTION SET



No TSX. Under separate CPUID bit

KNL implements all legacy instructions

- Legacy binary runs w/o recompilation
- KNC binary requires recompilation

KNL introduces AVX-512 Extensions

- 512-bit FP/Integer Vectors
- 32 registers & 8 mask registers
- Gather/Scatter

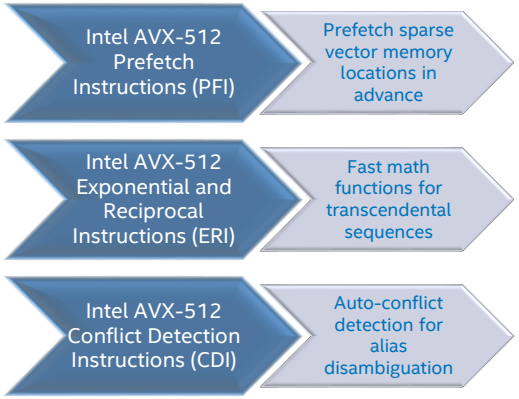
Conflict Detection: Improves Vectorization

Prefetch: Gather and Scatter Prefetch

Exponential and Reciprocal Instructions

1. Previous Code name Intel® Xeon® processors
2. Xeon Phi = Intel® Xeon Phi™ processor

KNL AVX512 INSTRUCTION SET



CPUID	Instructions	Description
AVX512PF	PREFETCHWT1	Prefetch cache line into the L2 cache with intent to write
	VGATHERPF{D,Q}{0,1}PS	Prefetch vector of D/Qword indexes into the L1/L2 cache
	VSCATTERPF{D,Q}{0,1}PS	Prefetch vector of D/Qword indexes into the L1/L2 cache with intent to write
AVX512ER	VEXP2{PS,PD}	Computes approximation of 2^x with maximum relative error of 2^{-23}
	VRCP28{PS,PD}	Computes approximation of reciprocal with max relative error of 2^{-28} before rounding
	VRSQRT28{PS,PD}	Computes approximation of reciprocal square root with max relative error of 2^{-28} before rounding
AVX512CD	VPCONFLICT{D,Q}	Detect duplicate values within a mask and create conflict-free subsets
	VPLZCNT{D,Q}	Count the number of leading zero bits in each element
	VPBROADCASTM{B2Q, W2D}	Broadcast vector mask into vector elements

MOTIVATION FOR CONFLICT DETECTION

Sparse computations are common in HPC, but hard to vectorize due to race conditions

Consider the "scatter" or "histogram" problem:

```
for(i=0; i<16; i++) {  A[B[i]]++; }
```

```
index = vload &B[i]                  // Load 16 B[i] indices
old_val = vgather A, index           // Grab A[B[i]]
new_val = vadd old_val, +1.0         // Compute new values
vscatter A, index, new_val           // Update A[B[i]]
```

- Problem if two vector lanes try to increment the same histogram bin
- Code above is wrong if any values within B[i] are duplicated
  - Only one update from the repeated index would be registered!
- A solution to the problem would be to avoid executing the sequence gather-op-scatter with vector of indexes that contain conflicts

CONFLICT DETECTION INSTRUCTIONS (CDI)

AVX-512 CDI introduces three new instructions:

- `vpconflict{d,q} zmm1 {k1}, zmm2/mem`
  Compares (for equality) each element in zmm2 with "earlier" elements and outputs bit vector.

- `vpbroadcastm{b2q,w2d} zmm1 {k0}, to_do`
- `vplzcnt{d,q}`

- `vptestnm{d,q} k2 {k1}, zmm1, zmm2/mem`
  (from AVX-512F)

Manipulate bit vector from vpconflict to construct a useful mask.

CONFLICT DETECTION INSTRUCTIONS (CDI)

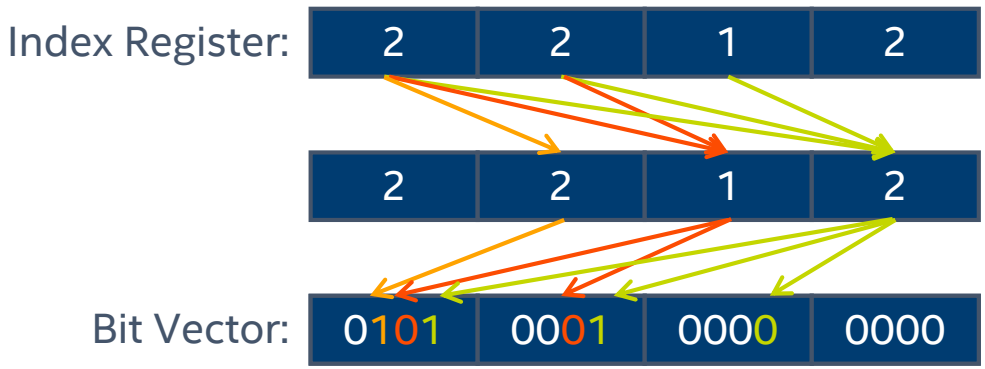
Vectorization with these instructions looks like this:

```
for (int i = 0; i < N; i += 16)
{
        __m512i indices = vload &B[i]
        vpconflictd comparisons, indices // comparisons = __m512i
        __mmask to_do = 0xffff;

        do
        {
                vpbroadcastmd tmp, to_do // tmp = __m512i
                vptestnmd mask {to_do}, comparisons, tmp
                do_work(mask); // gather-compute-scatter
                to_do ^= mask;
        } while(to_do);
}
```

Do work for element if no conflicts on **remaining** earlier elements.

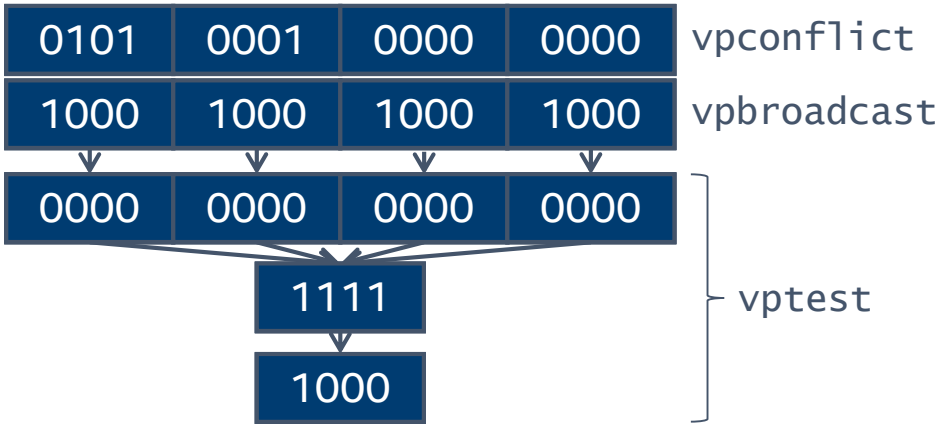
CONFLICT DETECTION INSTRUCTIONS (CDI) - EXAMPLE



1) Compare (for equality) each element in zmm2 with "**earlier**" elements and output bit vector.

2) Combine bit vector and todo to work out which elements can be updated in **this iteration**.

3) Loop until todo is 0000.



CONFLICT DETECTION INSTRUCTIONS (CDI) – COMPILER

The Intel® compiler (15.0 onwards) will recognise potential run-time conflicts and generate vpconflict loops automatically:

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

Such loops would originally have resulted in:

```
remark #15344: loop was not vectorized: vector dependence prevents vectorization
remark #15346: vector dependence: assumed FLOW dependence between histogram line 22 and histogram line 22
remark #15346: vector dependence: assumed ANTI dependence between histogram line 22 and histogram line 22
```

If you know that conflicts cannot occur, you should still specify this:
(e.g. #pragma ivdep, #pragma simd, #pragma omp simd)

GUIDELINES FOR WRITING VECTORIZABLE CODE

Prefer simple “for” or “DO” loops

Write straight line code. Try to avoid:

- function calls (unless inlined or SIMD-enabled functions)
- branches that can’t be treated as masked assignments.

Avoid dependencies between loop iterations

- Or at least, avoid read-after-write dependencies

Prefer arrays to the use of pointers

- Without help, the compiler often cannot tell whether it is safe to vectorize code containing pointers.
- Try to use the loop index directly in array subscripts, instead of incrementing a separate counter for use as an array address.
- Disambiguate function arguments, e.g. -fargument-noalias

Use efficient memory accesses

- Favor inner loops with unit stride
- Minimize indirect addressing a[i] = b[ind[i]]
- Align your data consistently where possible (to 16, 32 or 64 byte boundaries)

PROCESSOR DISPATCH (FAT BINARIES)

Compiler can generate multiple code paths

- Optimized for different processors
  - Only when likely to help performance
- One default code path, one or more optimized paths
- Optimized paths are for Intel processors only

Examples:

- -axavx
  - default path optimized for Intel® SSE2 (Intel or non-Intel) (-msse2)
  - Second path optimized for Intel® AVX (code name Sandy Bridge, etc.)
- -axcore-avx2,avx -xsse4.2
  - Default path optimized for Intel® SSE4.2 (code name Nehalem, Westmere)
  - Second path optimized for Intel® AVX (code name Sandy Bridge, etc)
  - Third path optimized for Intel® AVX2 (code name Haswell)

INTEL® COMPILER SWITCHES TARGETING INTEL® AVX-512

Switch	Description
`-xmic-avx512`	KNL only Not a fat binary.
`-xcore-avx512`	Future Xeon only Not a fat binary.
`-xcommon-avx512`	AVX-512 subset common to both. Not a fat binary.
`-axmic-avx512 etc.`	Fat binaries. Allows to target KNL and other Intel® Xeon® processors

Don't use –mmic with KNL !

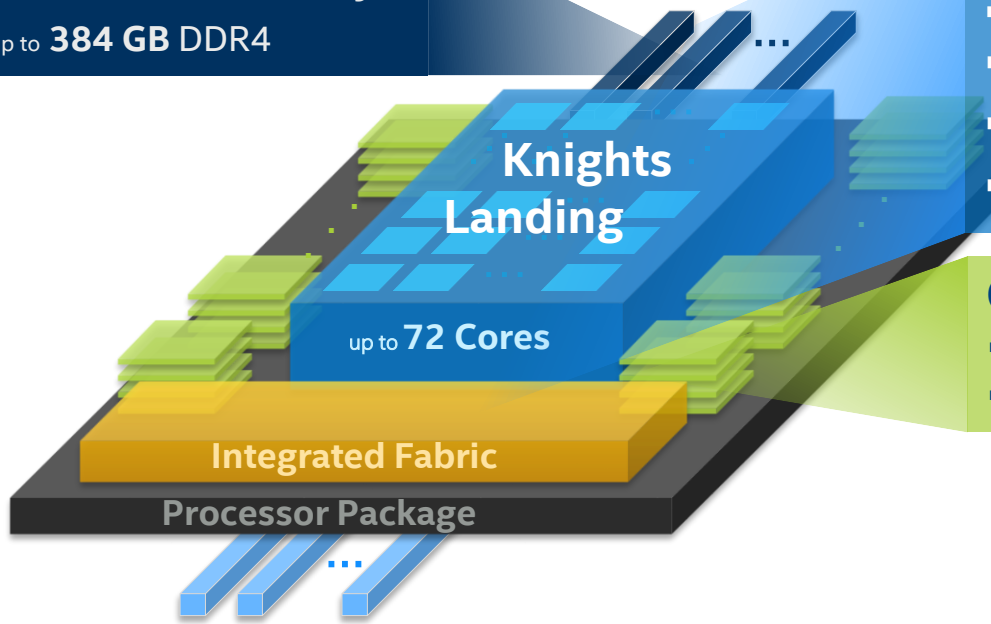
Best would be to use –axcore-avx512,mic-avx512 –xcommon-avx512

All supported in 16.0 and forthcoming 17.0 compilers

Binaries built for earlier Intel® Xeon® processors will run unchanged on KNL
Binaries built for Intel® Xeon Phi™ coprocessors will not.

INTEL® XEON PHI™ X200 PROCESSOR:
HIGH-BANDWIDTH MEMORY

INTEL® XEON PHI™ X200 PROCESSOR OVERVIEW



Platform Memory

Up to **384 GB** DDR4

Compute

- Intel® Xeon® Processor Binary-Compatible
- **3+ TF**LOPS, **3X ST** (single-thread) perf. vs KNC
- **2D Mesh** Architecture
- **Out-of-Order** Cores

On-Package Memory (MCDRAM)

- Up to **16 GB** *at launch*
- Over **5x STREAM** *vs. DDR4 at launch*

HETEROGENOUS MEMORY ARCHITECTURE

Intel® Xeon Phi™ x200 processor uses two types of memory

- standard DDR4 (DIMM)
- high-bandwidth MCDRAM (on-package)

What are the usage models?

How software can benefit from them?

MCDRAM MODES

Cache mode

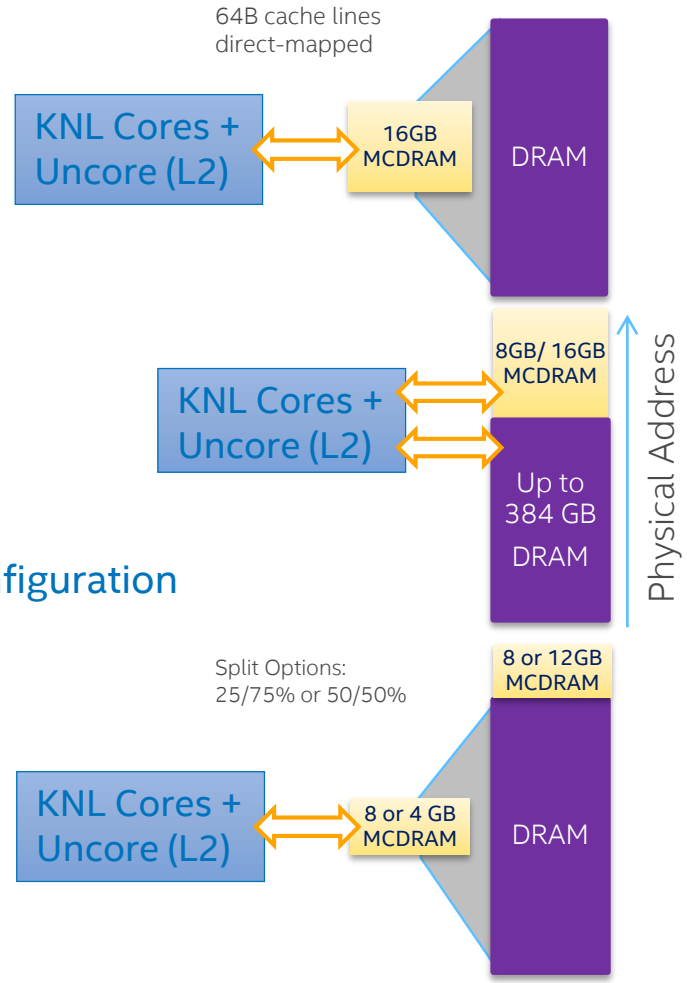
- Direct mapped cache
- Inclusive cache
- Misses have higher latency
  - Needs MCDRAM access + DDR access
- No source changes needed to use, automatically managed by hw as if LLC

Flat mode

- MCDRAM mapped to physical address space
- Exposed as a NUMA node
  - Use `numactl --hardware, lscpu` to display configuration
- Accessed through memkind library or numactl

Hybrid

- Combination of the above two
  - E.g., 8 GB in cache + 8 GB in Flat Mode



MCDRAM AS CACHE

Upside

- No software modifications required
- Bandwidth benefit (over DDR)

Downside

- Higher latency for DDR access
  - i.e., for cache misses
- Sustained misses limited by DDR BW
- All memory is transferred as:
  - DDR -> MCDRAM -> L2
- Less addressable memory

MCDRAM AS FLAT MODE

Upside

- Isolation of MCDRAM for high-performance application use only
  - OS and applications use DDR memory
  - No software modifications required if data fits in MCDRAM
- Lower latency
  - i.e., no MCDRAM cache misses
- Maximum addressable memory

Downside

- Generally, software modifications (or interposer library) required
  - to use DDR and MCDRAM in the same app
- Which data structures should go where?
- MCDRAM is a finite resource and tracking it adds complexity

TAKE AWAY MESSAGE: CACHE VS FLAT MODE

Recommended

	DDR Only	MCDRAM as Cache	MCDRAM Only	Flat DDR + MCDRAM	Hybrid
Software Effort	No software changes required			Change allocations for bandwidth-critical data.	
Performance	Not peak performance.		Best performance.		

Limited memory capacity

Optimal HW utilization + opportunity for new algorithms

HOW TO ACCESS MCDRAM IN FLAT MODE?

New mechanisms proposed by Intel:

- Memkind Library
  - User space library
  - C/C++ language interface
  - Needs source modification
- Fortran FASTMEM compiler directives
  - Internally uses memkind library
  - Ongoing language standardization efforts
- AutoHBW for C/C++
  - interposer library based on memkind
    - No source modification needed (based on size of allocations)
    - No fine control over *individual* allocations

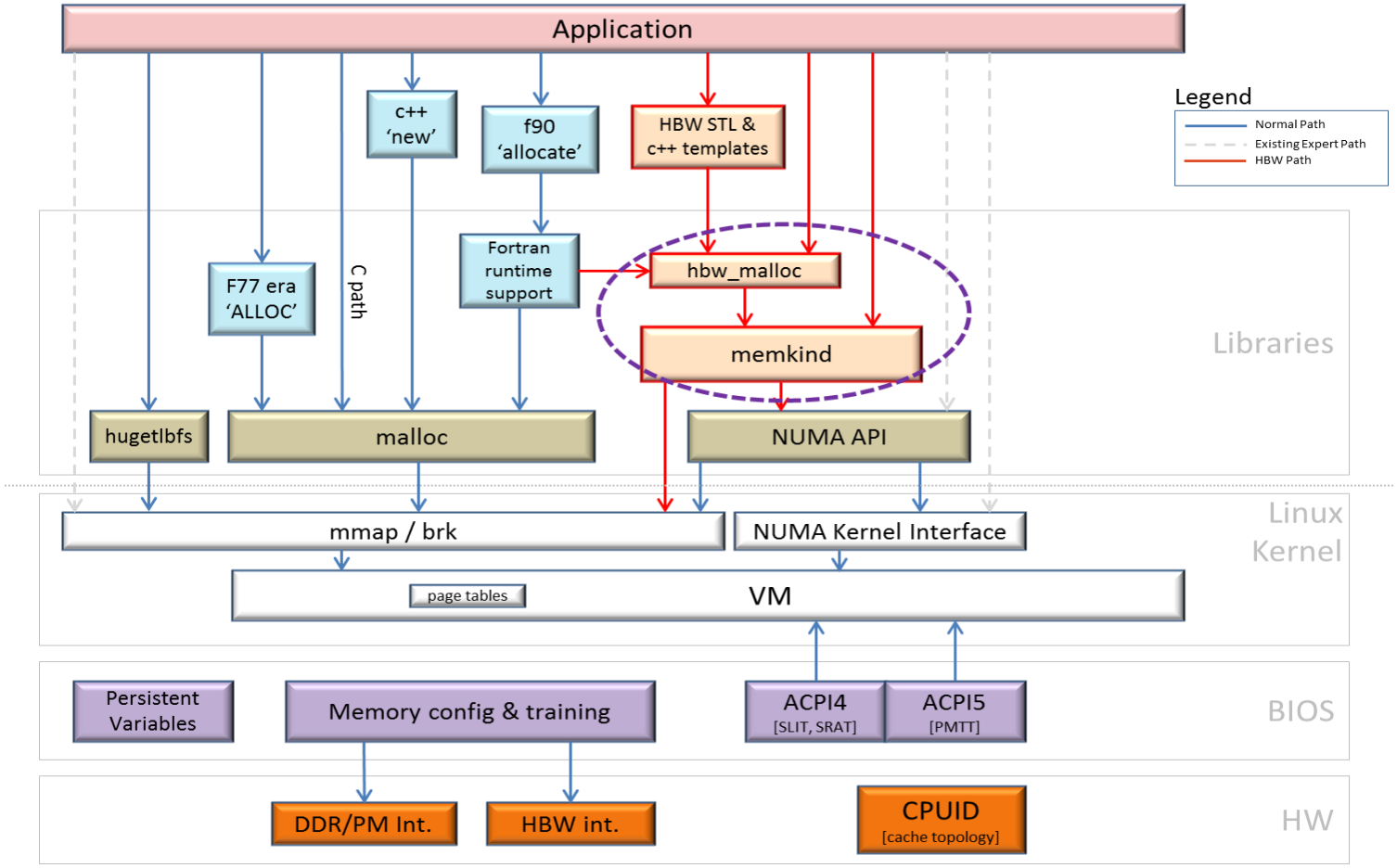
Use standard OS mechanisms

- Using numactl
- Direct OS system calls
  - mmap(1), mbind(1)
  - Not the preferred method
    - Page-only granularity, OS serialization, no pool management

Scope of this presentation

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

MEMKIND LIBRARY ARCHITECTURE



A HETEROGENEOUS MEMORY MANAGEMENT FRAMEWORK

The **memkind** library

- Defines a plug-in architecture
- Each plug-in is called a “kind” of memory
- Built on top of jemalloc
- High level memory management functions can be overridden
- Available via github: https://github.com/memkind

The **hbwmalloc** interface

- The high bandwidth memory interface
- Implemented on top of memkind
- Simplifies memkind plug-in (kind) selection
- Uses all kinds featuring on package memory on the Knights Landing architecture
- Provides support for 2MB and 1GB pages
- Select fallback behavior when on package memory does not exist or is exhausted
- Check for existence of on package memory

MEMKIND – "KINDS" OF MEMORY

Many "kinds" of memory supported by memkind:

- **MEMKIND_DEFAULT**
  Default allocation using standard memory and default page size.
- **MEMKIND_HBW**
  Allocate from the closest high-bandwidth memory NUMA node at time of allocation.
- **MEMKIND_HBW_PREFERRED**
  If there is not enough HBW memory to satisfy the request, fall back to standard memory.
- **MEMKIND_HUGETLB**
  Allocate using 2MB pages.
- **MEMKIND_GBTLB**
  Allocate using GB pages.
- **MEMKIND_INTERLEAVE**
  Allocate pages interleaved across all NUMA nodes.
- **MEMKIND_PMEM**
  Allocate from file-backed heap.

These can all be used with HBW (e.g. MEMKIND_HBW_HUGETLB); all but INTERLEAVE can be used with HBW_PREFERRED.

MEMKIND & HBWMALLOC – EARLY EXPERIMENTS

AutoHBW: Interposer Library that comes with memkind

- Automatically allocates memory from MCDRAM
  - If a heap allocation (e.g., malloc/calloc) is larger than a given threshold

```
LD_PRELOAD=libautohbw.so ./application
```

Run-time configuration options are passed through environment variables:

- **AUTO_HBW_SIZE**=x[:y]
  Any allocation larger than x and smaller than y should be allocated in HBW memory.
- **AUTO_HBW_MEM_TYPE**
  Sets the "kind" of HBW memory that should be allocated (e.g. MEMKIND_HBW)
- **AUTO_HBW_LOG** and **AUTO_HBW_DEBUG** for extra information.

Easy to integrate similar functionality into other libraries, C++ allocators, etc.

MEMKIND – C "HELLO WORLD!" EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <memkind.h>

int main(int argc, char **argv)
{
            const size_t size = 512;
            char *default_str = NULL;
            char *hbw_str = NULL;

            default_str = (char *)memkind_malloc(MEMKIND_DEFAULT, size);
            if (default_str == NULL) {
                        perror("memkind_malloc()");
                        fprintf(stderr, "Unable to allocate default string\n");
                        return errno ? -errno : 1;
            }

            hbw_str = (char *)memkind_malloc(MEMKIND_HBW, size);
            if (hbw_str == NULL) {
                        perror("memkind_malloc()");
                        fprintf(stderr, "Unable to allocate hbw string\n");
                        return errno ? -errno : 1;
            }

            sprintf(default_str, "Hello world from standard memory\n");
            sprintf(hbw_str, "Hello world from high bandwidth memory\n");
            fprintf(stdout, "%s", default_str);
            fprintf(stdout, "%s", hbw_str);

            memkind_free(MEMKIND_DEFAULT, hbw_str);
            memkind_free(MEMKIND_DEFAULT, default_str);

            return 0;
}
```

+ Link w/ memkind library
(otherwise won't link due to unresolved references)

Based on:
https://github.com/memkind/memkind/blob/dev/examples/hello_memkind_example.c

USING MEMKIND LIBRARY TO ACCESS MCDRAM (FORTRAN)

- Unlike C, Fortran does not rely on a malloc –type API to perform allocations of dynamic memory
  - Intrinsic ALLOCATE statement used for all dynamic allocations
  - Intrinsic DEALLOCATE for deallocation of memory
  - NOTE: Fortran 2003 standard requires ALLOCATABLE variables to be automatically deallocated when they go out of scope

```
c       Declare arrays to be dynamic
        REAL, ALLOCATABLE :: A(:), B(:), C(:)

!DEC$ ATTRIBUTES FASTMEM :: A

        NSIZE=1024
c       allocate array 'A' from MCDRAM
        ALLOCATE (A(1:NSIZE))

c       Allocate arrays that will come from DDR
        ALLOCATE  (B(NSIZE), C(NSIZE))
```

+ Link w/ memkind library
(otherwise silently allocated in DDR)

FORTRAN FASTMEM STATUS

- To have ATTRIBUTES FASTMEM, ALLOCATABLE –attribute is required
- In version 16 of the Intel compiler, FASTMEM is not allowed for
  - Variables with the POINTER –attribute
    ```
    REAL, POINTER :: array(:)
    ```
  - Automatic (stack) variables
    ```
    SUBROUTINE SUB1(n)
     INTEGER :: n
     REAL :: A(n,n)
     ...
    END SUBROUTINE
    ```
  - Components of derived types
    ```
    TYPE mytype
      REAL, ALLOCATABLE :: array(:)
    END TYPE mytype
    ```
  - COMMON blocks
    ```
    INTEGER, PARAMETER :: NARR = 1000
    REAL ARRAY(NARR,NARR)
    COMMON /MATRIX/ ARRAY, N
    ```

RUNNING MEMKIND

The following command allocates all the data from the application into DDR (NUMA node 0) except for the MEMKIND allocations on HBW (NUMA node 1)

```
export MEMKIND_HBW_NODES=1
numactl --membind=0 --cpunodebind=0 <binary>
```

HBWMALLOC – C "HELLO WORLD!" EXAMPLE

Fallback policy is controlled with **hbw_set_policy**:
- HBW_POLICY_BIND
- HBW_POLICY_PREFERRED
- HBW_POLICY_INTERLEAVE

Page sizes can be passed to **hbw_posix_memalign_psize**:
- HBW_PAGESIZE_4KB
- HBW_PAGESIZE_2MB
- HBW_PAGESIZE_1GB

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <hbwmalloc.h>

int main(int argc, char **argv)
{
        const size_t size = 512;
        char *default_str = NULL;
        char *hbw_str = NULL;

        default_str = (char *)malloc(size);
        if (default_str == NULL) {
                perror("malloc()");
                fprintf(stderr, "Unable to allocate default string\n");
                return errno ? -errno : 1;
        }

        hbw_str = (char *)hbw_malloc(size);
        if (hbw_str == NULL) {
                perror("hbw_malloc()");
                fprintf(stderr, "Unable to allocate hbw string\n");
                return errno ? -errno : 1;
        }

        sprintf(default_str, "Hello world from standard memory\n");
        sprintf(hbw_str, "Hello world from high bandwidth memory\n");
        fprintf(stdout, "%s", default_str);
        fprintf(stdout, "%s", hbw_str);

        hbw_free(hbw_str);
        free(default_str);

        return 0;
}
```

+ Link w/ memkind library
(otherwise won't link due to unresolved references)

Based on:
https://github.com/memkind/memkind/blob/dev/examples/hello_hbw_example.c

HBWMALLOC – C++ STL ALLOCATOR EXAMPLE

```
#include <iostream>
#include <vector>

#include <hbw_allocator.h>

int main(int argc, char **argv)
{
    const int length = 10;

    std::vector<double, hbw::allocator<double> > data(10);

    for (int i = 0; i < length; ++i) {
        data[i] = (double)(i);
    }

    std::cout << data[length-1] << std::endl;
    return 0;
}
```

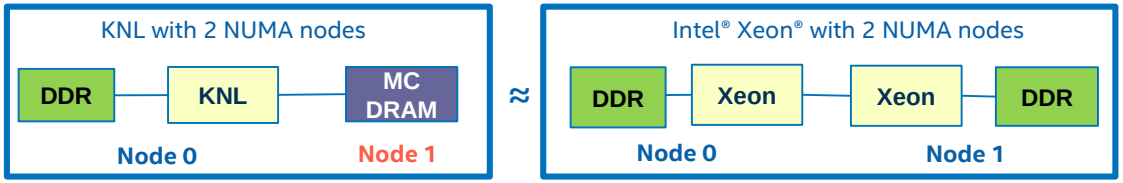
+ Link w/ memkind library
(otherwise won't link due to unresolved references)

STANDARD WAYS OF ACCESSING MCDRAM

MCDRAM is exposed to OS/software **as a NUMA node**

Utility **numactl** is standard utility for NUMA system control

- See **"man numactl"**
- Do **"numactl --hardware"** to see the NUMA configuration of your system



If the total memory footprint of your app is smaller than the size of MCDRAM

- Use **numactl** to allocate all of its memory from MCDRAM
- `numactl --membind=mcdram_id <command>`
  - Where *mcdram_id* is the ID of MCDRAM "node"
- Allocations that don't fit into MCDRAM make application fail

If the total memory footprint of your app is larger than the size of MCDRAM

- You can still use numactl to allocate ***part*** of your app in MCDRAM
  - `numactl --preferred=mcdram_id <command>`
- Allocations that don't fit into MCDRAM spill over to DDR

SOFTWARE VISIBLE MEMORY CONFIGURATION

1. Cache mode / Quadrant

```
[andrey@knl3 ~]$ numactl --hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 1
66 67 68 69 70 71 72 73 74 75 76 77 78 7
 125 126 127 128 129 130 131 132 133 134
175 176 177 178 179 180 181 182 183 184 1
25 226 227 228 229 230 231 232 233 234 235
node 0 size: 98200 MB
node 0 free: 92552 MB
node distances:
node   0
  0:  10
[andrey@knl3 ~]$
```

2. Flat mode / Quadrant

```
[andrey@knl4 ~]$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 1
66 67 68 69 70 71 72 73 74 75 76 77 78 7
 125 126 127 128 129 130 131 132 133 134
175 176 177 178 179 180 181 182 183 184 1
25 226 227 228 229 230 231 232 233 234 23
node 0 size: 98200 MB
node 0 free: 94632 MB
node 1 cpus:
node 1 size: 16384 MB
node 1 free: 15929 MB
node distances:
node   0   1
  0:  10  31
  1:  31  10
[andrey@knl4 ~]$
```

3. Cache mode / **SNC-4**

```
[andrey@knl3 ~]$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 1
202 203 204 205 206 207
node 0 size: 24472 MB
node 0 free: 22989 MB
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26
5 216 217 218 219 220 221 222 223
node 1 size: 24576 MB
node 1 free: 23807 MB
node 2 cpus: 48 49 50 51 52 53 54 55 56 57 58 59
3 244 245 246 247 248 249 250 251 252 253 254 255
node 2 size: 24576 MB
node 2 free: 23835 MB
node 3 cpus: 32 33 34 35 36 37 38 39 40 41 42 43
8 229 230 231 232 233 234 235 236 237 238 239
node 3 size: 24576 MB
node 3 free: 23748 MB
node distances:
node   0   1   2   3
  0:  10  21  21  21
  1:  21  10  21  21
  2:  21  21  10  21
  3:  21  21  21  10
[andrey@knl3 ~]$
```

4. Flat mode with **sub-NUMA clustering (SNC-4)**

```
[andrey@knl3 ~]$ numactl --hardware
available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 1
202 203 204 205 206 207
node 0 size: 24472 MB
node 0 free: 23186 MB
node 1 cpus: 16 17 18 19 20 21 22 23 24 25
5 216 217 218 219 220 221 222 223
node 1 size: 24576 MB
node 1 free: 23807 MB
node 2 cpus: 48 49 50 51 52 53 54 55 56 57 58
3 244 245 246 247 248 249 250 251 252 253 254
node 2 size: 24576 MB
node 2 free: 23741 MB
node 3 cpus: 32 33 34 35 36 37 38 39 40 41 42 43
8 229 230 231 232 233 234 235 236 237 238 239
node 3 size: 24576 MB
node 3 free: 23756 MB
node 4 cpus:
node 4 size: 4096 MB
node 4 free: 3982 MB
node 5 cpus:
node 5 size: 4096 MB
node 5 free: 3981 MB
node 6 cpus:
node 6 size: 4096 MB
node 6 free: 3982 MB
node 7 cpus:
node 7 size: 4096 MB
node 7 free: 3979 MB
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  21  21  21  31  41  41  41
  1:  21  10  21  21  41  31  41  41
  2:  21  21  10  21  41  41  41  31
  3:  21  21  21  10  41  41  31  41
  4:  31  41  41  41  10  41  41  41
  5:  41  31  41  41  41  10  41  41
  6:  41  41  41  31  41  41  10  41
  7:  41  41  31  41  41  41  41  10
[andrey@knl3 ~]$
```

DDR

MCDRAM

OBTAINING MEMKIND LIBRARY

Homepage: http://memkind.github.io/memkind

Download package

- On RHEL* 7
  - `yum install epel-release`; `yum install memkind`
- For other distros: install from http://download.opensuse.org/repositories/home:/cmcantalupo/

Alternatively, you can build from source

- `git clone https://github.com/memkind.git`
- See CONTRIBUTING file for build instructions
- Must use this option to get AutoHBW library
- Requires libnuma (development files and libraries)
  - `yum install numactl-devel`

MKL AND HBM

Intel MKL 2017 memory manager tries to allocate memory to MCDRAM through the Memkind library.

By default the amount of MCDRAM available for Intel MKL is unlimited. To control the amount of MCDRAM available for Intel MK use either of the following:

- Call `mkl_set_memory_limit (MKL_MEM_MCDRAM, <limit in mbytes>)`
- Set the `MKL_FAST_MEMORY_LIMIT=<limit in mbytes>` environment var

MPI AND HBM

The environment variable is to control memory policy for MPI processes. There are three kinds of memory we can control:

- User code memory (emulates “numactl –m” command)
- MPI buffers
- User’s buffers but allocated by IMPI for MPI_Win_allocate_shared/MPI_Win_allocate

PSXE 2017 Beta U1

The suggested format for the environment variable is the following:

I_MPI_HBW_POLICY=<USER BUFFERS POLICY>[,[MPI BUFFERS POLICY][,WIN_ALLOCATE POLICY]]

Where each of comma separated values can be the following:

hbw_preferred	Memory allocation go first to local for a process MCDRAM, then to local DRAM
hbw_bind	Only allocate memory on local for a process MCDRAM
hbw_interleave	Memory will be interleaved between the MCDRAM and DRAM on the local SNC node

TAKE-AWAY MESSAGES

Intel Xeon Phi X200 is a highly capable processor

- Can run your already built applications for Xeon
- w/ Highly parallel system to execute many processes/threads at the same time
- w/ Highly vectorized architecture to generate multiple operations per instruction
- w/ High-Bandwidth Memory to shorten the memory gap
- … with low energy consumption footprint

Intel tools / libraries / compilers are here to help on taking advantage of all these properties.

intel experience what's inside™

THANK YOU!
QUESTIONS?