AMS 250: An Introduction to High Performance Computing

Knights Landing

Shawfeng Dong

shaw@ucsc.edu

(831) 502-7743

Applied Mathematics & Statistics

University of California, Santa Cruz

Outline

- Cori “Knights Landing ” (KNL) Compute Nodes
- Knights Landing Overview
- Knights Landing Configurations
  - Cluster Modes
  - Memory Modes
- Application Porting and Performance
  - MPI + OpenMP
  - Accessing MCDRAM
  - Vectorization

Cori System Overview

- Cray XC40 supercomputer (NERSC-8)
- 2,388 Phase I Haswell Compute Nodes, each with:
  - Two Intel “Haswell” Xeon E5-2698v3 16-core CPU @ 2.3 GHz
  - 128 GB DDR4 2133 MHz memory
- 9,688 Phase II KNL Compute Nodes
- 12 Login Nodes (dual-socket, 16-core) with 512GB memory each
- Cray Aries Interconnect with Dragonfly topology, 5.625 TB/s global BW (Phase I), 45.0 TB/s global BW (Phase II)
- 30 PB of Lustre scratch storage, with an aggregate transfer rate of > 700 GB/s
- 1.8 PB of Burst Buffer

Total # of Phase I Haswell cores = 76,416

Total # of Phase II KNL cores = 658,784

Aggregate memory = 203 TB (Phase I),
1PB (Phase II)

R_{peak} = 1.92 PFLOPS (I) + 29.1 PFLOPS (II)

R_{max} = 14 PFLOPS

Power = 3.9 MW

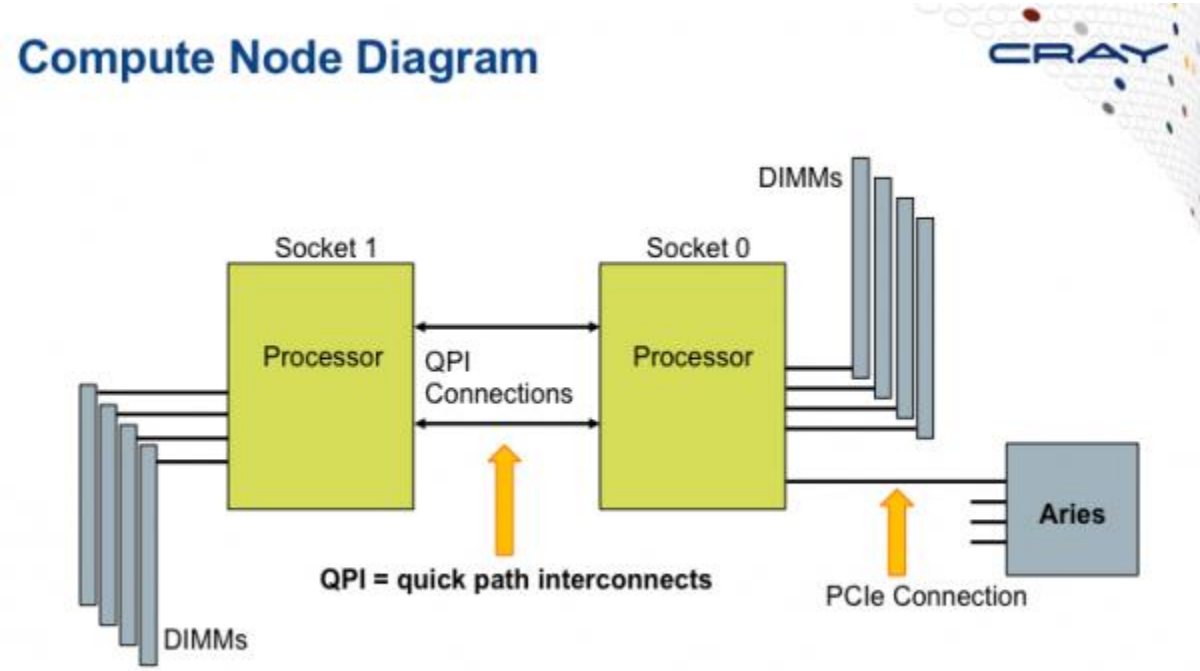
#5 in November 2016

#8 in November 2017

Cori Haswell Compute Nodes

- 2,388 Haswell Compute Nodes, each with:
  - Two Intel “Haswell” Xeon E5-2698 v3 16-core CPU @ 2.3 GHz
  - 128 GB DDR4 2133 MHz memory (four 16 GB DIMMs per socket)
  - An Aries Network Interface Controller (NIC)

    0.25 µs to 3.7 µs MPI latency,
    ~8GB/sec MPI bandwidth
- Compute nodes run a *lightweight* kernel and run-time environment based on SuSE Linux Enterprise Server (SLES) Linux distribution



Cori KNL Compute Nodes

- 9,688 “Knights Landing” (KNL) Compute Nodes, each with:
  - A single Intel Xeon Phi 7250 processor with 68 cores @ 1.4 GHz
  - 16 GB MCDRAM (multi-channel DRAM), > 460 GB/s peak bandwidth
  - 96 GB (6 x 16 GB) DDR4 2400 MHz memory, 102 GB/s peak bandwidth
  - An Aries Network Interface Controller (NIC)

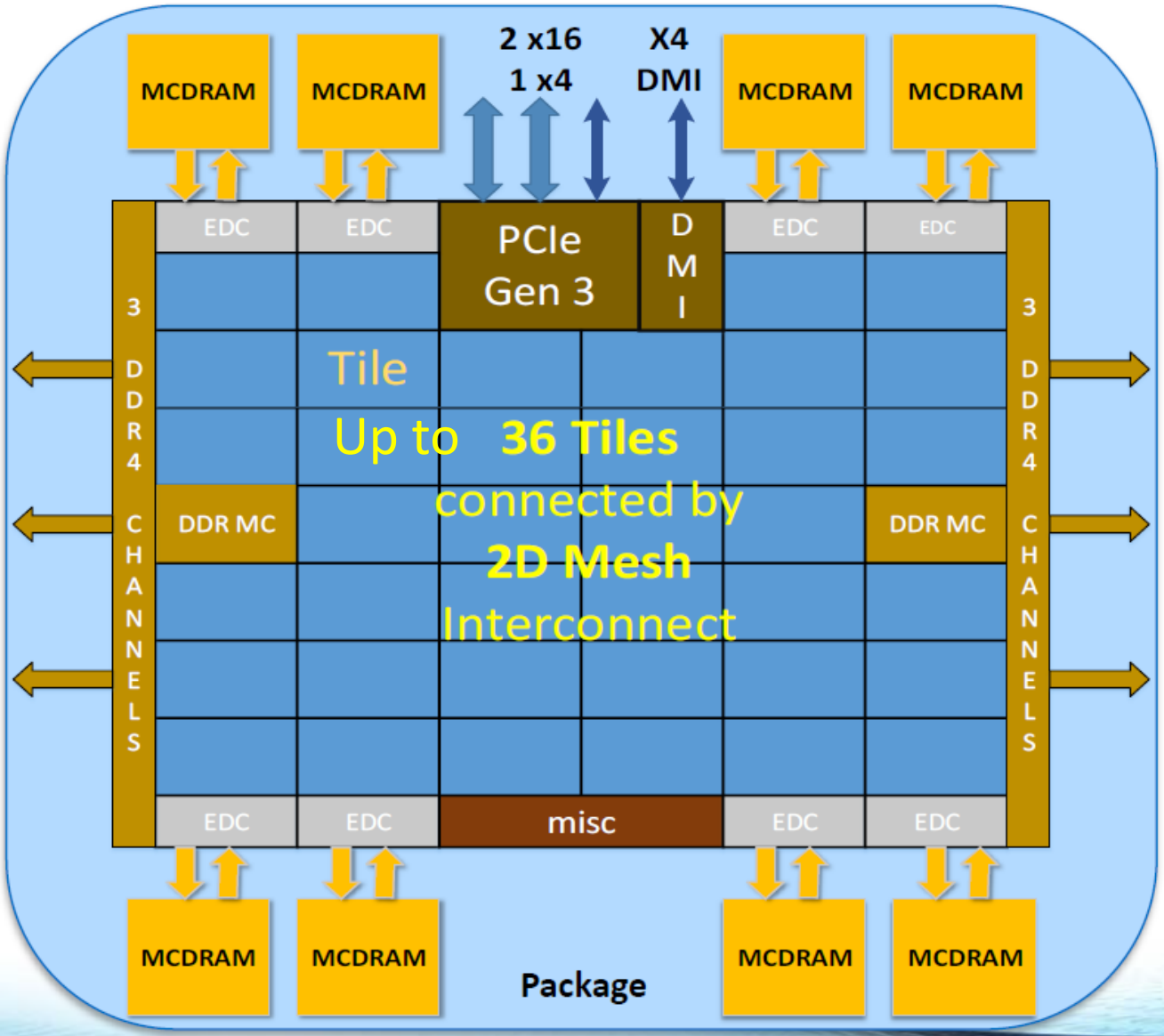
    0.25 µs to 3.7 µs MPI latency, ~8GB/sec MPI bandwidth

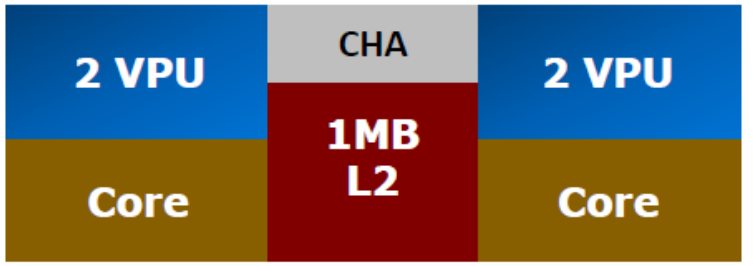
    45.0 TB/s global peak bisection bandwidth
- Compute nodes run a *lightweight* kernel and run-time environment based on SuSE Linux Enterprise Server (SLES) Linux distribution

Note: some of the subsequent slides are borrowed from Avinash Sodani's presentation
http://cgo.org/cgo2016/wp-content/uploads/2016/04/sodani-slides.pdf

```
cori03:> salloc -N 1 -p debug -t 00:10:00 -L SCRATCH -C knl
nid04520:> uname -a
Linux nid04520 3.12.60-52.49.1_2.2-cray_ari_c #1 SMP Wed May 10 20:40:01
UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
nid04520:> lsb_release -a
LSB Version:    n/a
Distributor ID: SUSE LINUX
Description:    SUSE Linux Enterprise Server 12
Release:        12
Codename:       12
nid04520:> cat /proc/cpuinfo
processor       : 0
...
processor       : 271
siblings        : 272
cpu cores       : 68
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good
nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64
monitor ds_cpl est tm2 ssse3 fma cx16 xtpr pdcm sse4_1 sse4_2 x2apic
movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm
3dnowprefetch ida arat epb xsaveopt pln pts dtherm fsgsbase tsc_adjust
bmi1 avx2 smep bmi2 erms avx512f rdseed adx avx512pf avx512er avx512cd
```

Knights Landing Overview





Chip: 36 Tiles interconnected by **2D Mesh**

Tile: 2 Cores + 2 VPU/core + 1 MB L2

Memory: MCDRAM: 16 GB on-package; High BW

DDR4: 6 channels @ 2400 up to 384GB

IO: 36 lanes PCIe Gen3. 4 lanes of DMI for chipset

Node: 1-Socket only

Fabric: Omni-Path on-package (not shown)

Vector Peak Perf: 3+TF DP and 6+TF SP Flops

Scalar Perf: ~3x over Knights Corner

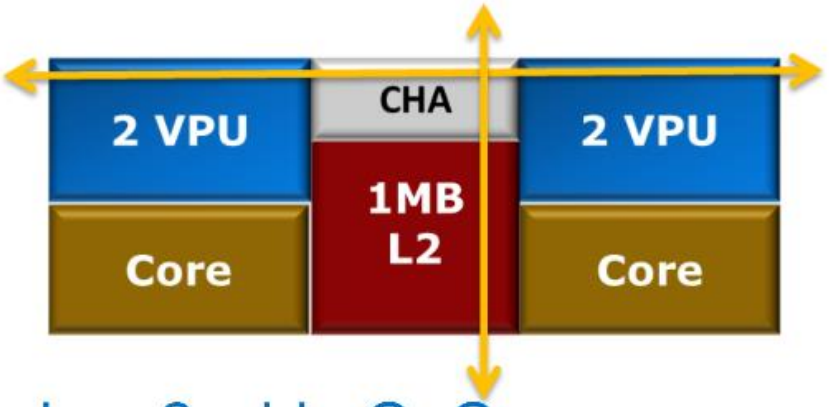
Streams Triad (GB/s): MCDRAM : 400+; DDR: 90+

Source Intel: All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice. KNL data are preliminary based on current expectations and are subject to change without notice. 1Binary Compatible with Intel Xeon processors using Haswell Instruction Set (except TSX). 2Bandwidth numbers are based on STREAM-like memory access pattern when MCDRAM used as flat memory. 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.

KNL Tile:

2 Cores, each with 2 VPU

1M L2 shared between two Cores



Core: Changed from Knights Corner (KNC) to KNL. Based on 2-wide OoO Silvermont™ Microarchitecture, but with *many* changes for HPC.

4 thread/core. Deeper OoO. Better RAS. Higher bandwidth. Larger TLBs.

2 VPU: 2x AVX512 units. 32SP/16DP per unit. X87, SSE, AVX1, AVX2 and EMU

L2: 1MB 16-way. 1 Line Read and ½ Line Write per cycle. Coherent across all Tiles

CHA: **C**aching/**H**ome **A**gent. Distributed Tag Directory to keep L2s coherent. MESIF protocol. 2D-Mesh connections for Tile

Many Trailblazing Improvements in KNL

Improvements	What/Why
Self Boot Processor	No PCIe bottleneck
Binary Compatibility with Xeon	Runs all legacy software. No recompilation.
New Core: Atom™ based	~3x higher ST performance over KNC
Improved Vector density	3+ TFLOPS (DP) peak per chip
New AVX 512 ISA	New 512-bit Vector ISA with Masks
Scatter/Gather Engine	Hardware support for gather and scatter
New memory technology: MCDRAM + DDR	Large High Bandwidth Memory → MCDRAM Huge bulk memory → DDR
New on-die interconnect: Mesh	High BW connection between cores and memory
Integrated Fabric: Omni-Path	Better scalability to large systems. Lower Cost

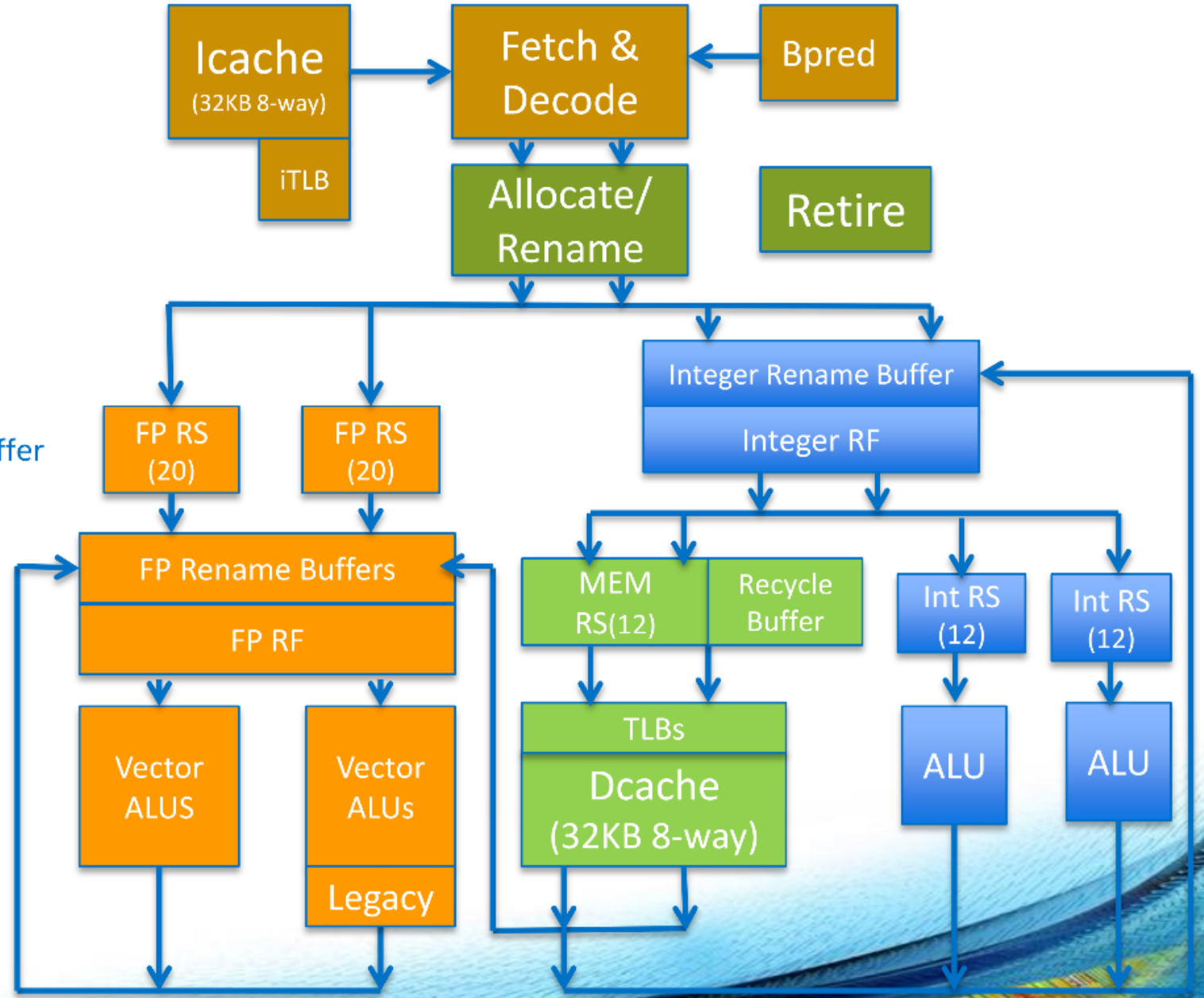
Core & VPU

- Out-of-order core w/ 4 SMT threads
- VPU tightly integrated with core pipeline

- 2-wide Decode/Rename/Retire
- ROB-based renaming. 72-entry ROB & Rename Buffers
- Up to 6-wide at execution
- Int and FP RS OoO.
- MEM RS inorder with OoO completion. Recycle Buffer holds memory ops waiting for completion.
- Int and Mem RS hold source data. FP RS does not.

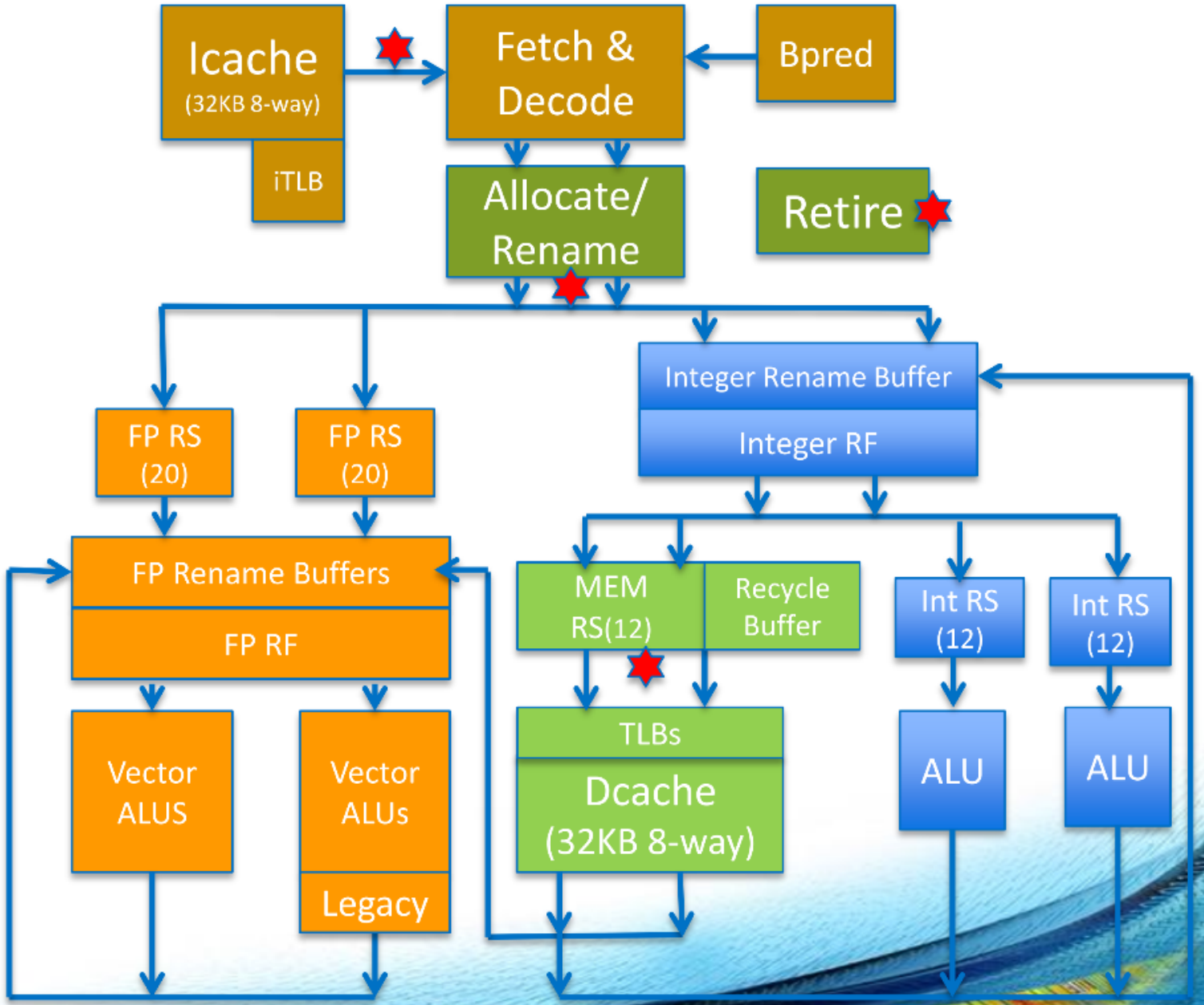
- 2x 64B Load & 1 64B Store ports in Dcache.
- 1st level uTLB: 64 entries
- 2nd level dTLB: 256 4K, 128 2M, 16 1G pages

- L1 Prefetcher (IPP) and L2 Prefetcher.
- 46/48 PA/VA bits
- Fast unaligned and cache-line split support.
- Fast Gather/Scatter support

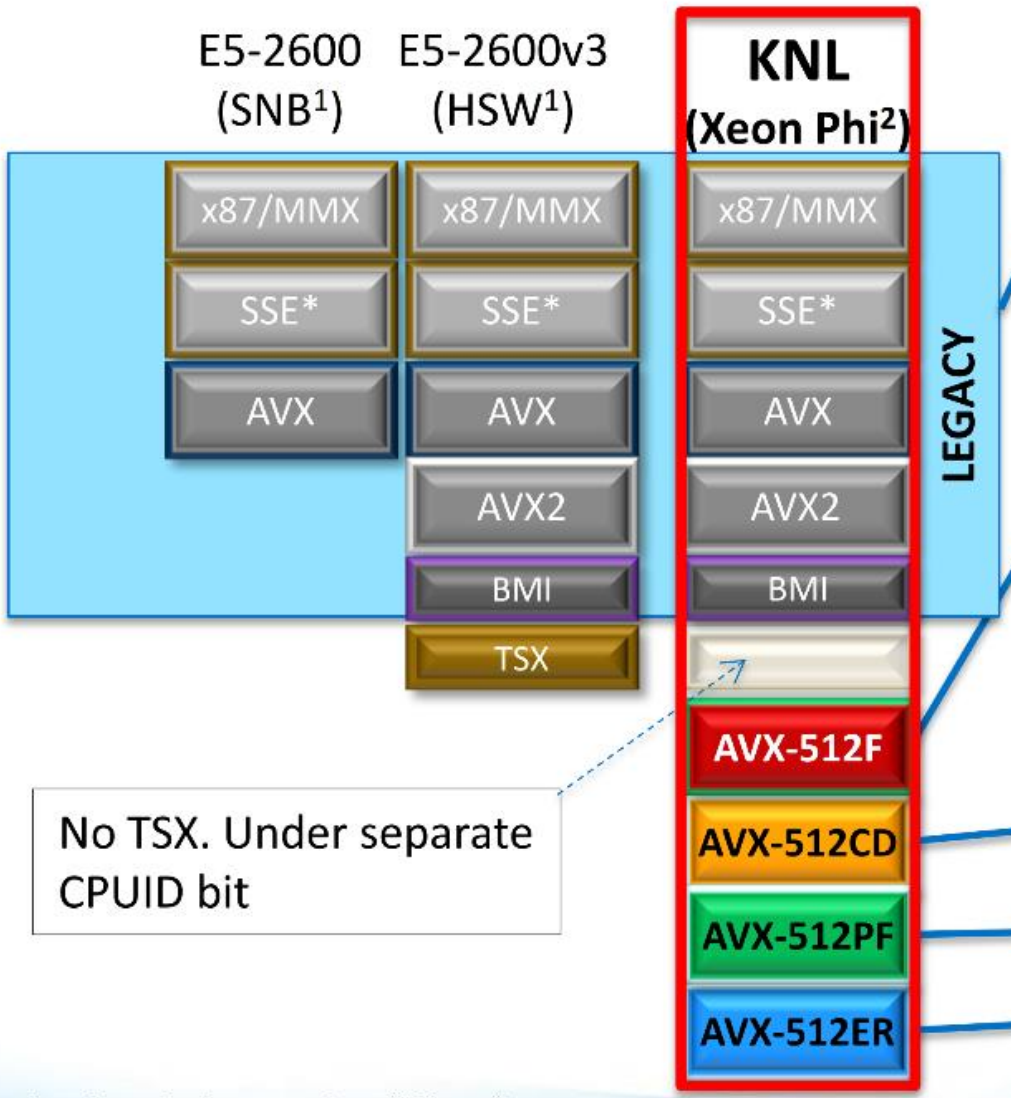


Threading

- 4 Threads per core. Simultaneous Multithreading.
- Core resources **shared** or **dynamically repartitioned** between active threads
  - ROB, Rename Buffers, RS: Dynamically partitioned
  - Caches, TLBs: Shared
  - E.g., 1 thread active → uses full resources of the core
- Several Thread Selection points in the pipeline. (✸)
  - Maximize throughput while being fair.
  - Account for available resources, stalls and forward progress



KNL ISA



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

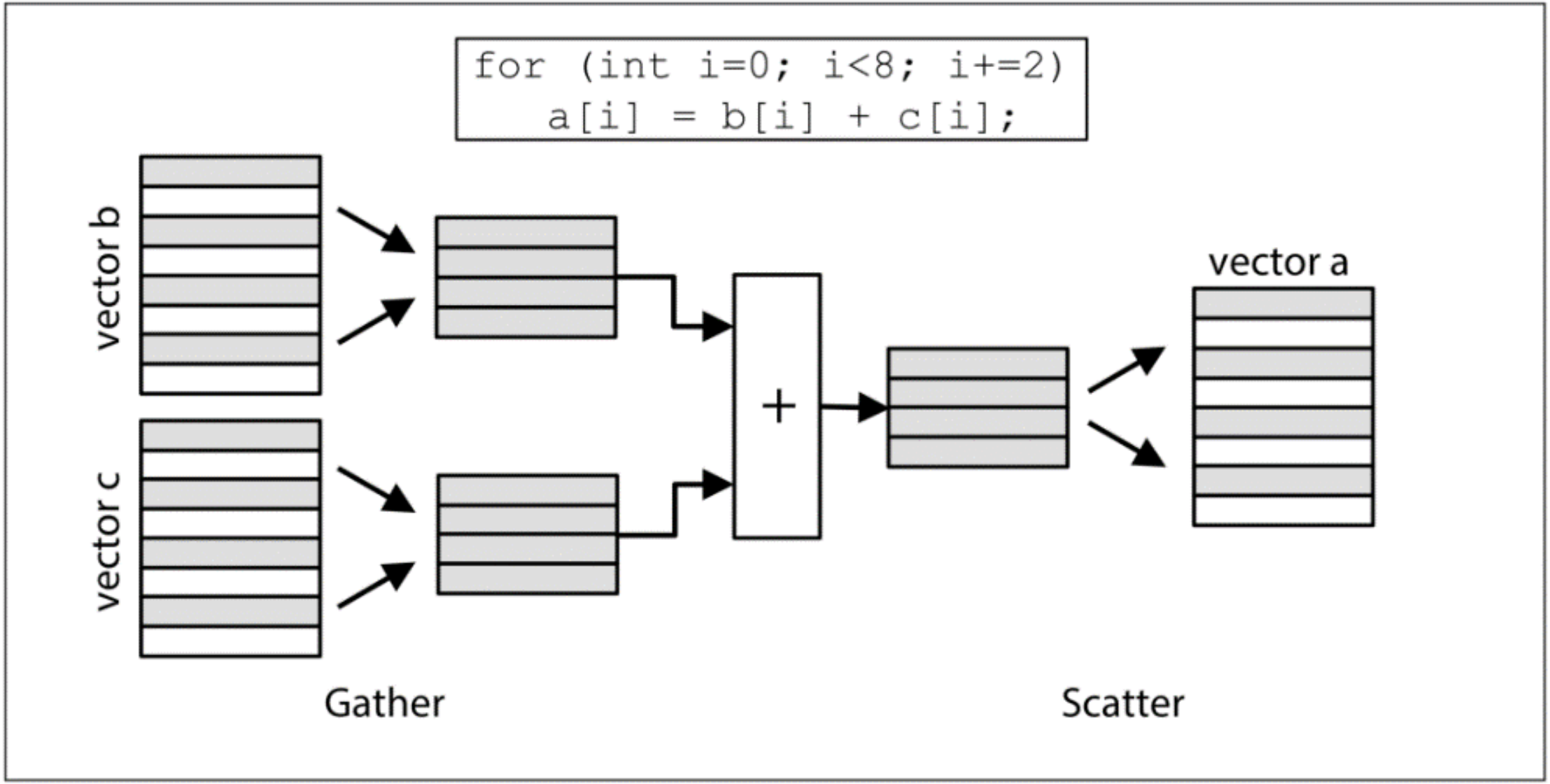
Conflict Detection: Improves Vectorization

Prefetch: Gather and Scatter Prefetch

Exponential and Reciprocal Instructions

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

Gather and scatter scalar data elements in and out of a vector



AVX-512 PF, ER and CD Instructions

- Intel AVX-512 Prefetch Instructions (PFI)
- Intel AVX-512 Exponential and Reciprocal Instructions (ERI)
- Intel AVX-512 Conflict Detection Instructions (CDI)

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 vector 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

AVX-512 CD: Instructions for enhance vectorization

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

```
index = vload &B[i]              // Load 16 B[i]
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]]
```

Code is wrong if any values within B[i] are duplicated

```
index = vload &B[i]                              // Load 16 B[i]
pending_elem = 0xFFFF;                           // all still remaining
do {
   curr_elem = get_conflict_free_subset(index, pending_elem)
   old_val = vgather {curr_elem} A, index  // Grab A[B[i]]
   new_val = vadd old_val, +1.0                  // Compute new values
   vscatter A {curr_elem}, index, new_val  // Update A[B[i]]
   pending_elem = pending_elem ^ curr_elem // remove done idx
} while (pending_elem)
```

AVX-512 Conflict Detection

- VPCONFLICT{D,Q} zmm1{k1}, zmm2/mem
- VPBROADCASTM{W2D,B2Q} zmm1, k2
- VPTESTNM{D,Q} k2{k1}, zmm2, zmm3/mem
- VPLZCNT{D,Q} zmm1 {k1}, zmm2/mem

Building Executables for KNL Compute Nodes

- The default environment on Cori currently has the craype-haswell module loaded as default. As a result, executables built with the Cray compiler wrappers (ftn, cc, CC) are targeted to run on the Haswell compute nodes.
- KNL nodes can run legacy binaries (e.g., built for Haswell nodes) without recompilation.
- But Haswell executables can't fully utilize KNL hardware.
- To build for KNL compute nodes, use either of the following 2 options:
  - Under the default craype-haswell environment, explicitly add the compiler flag -xMIC-AVX512 (for the default Intel compilers)
  - Do a module swap before compiling:
    module swap craype-haswell craype-mic-knl

http://www.nersc.gov/users/computational-systems/cori/programming/compiling-codes-on-cori/

Building for both KNL and Haswell

Using the default Intel compilers, we can use the compiler flag -axMIC-AVX512,CORE-AVX2 to build a fat binary for both KNL and Haswell. For example:

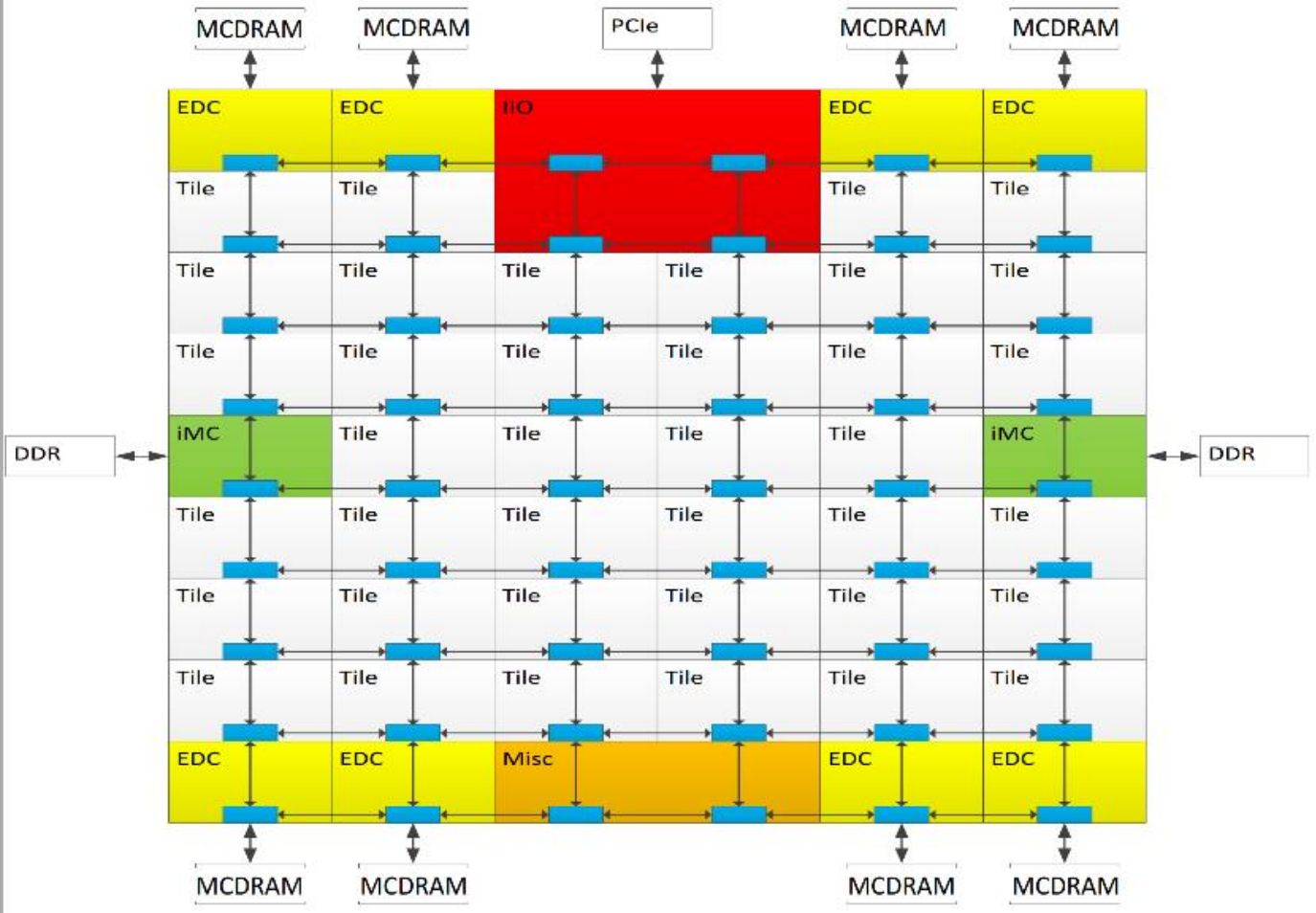
```
CC -axMIC-AVX512,CORE-AVX2 <more-options> mycode.c++
```

Recommendations: for best performance on KNL, use both

```
module swap craype-haswell craype-mic-knl
CC -axMIC-AVX512,CORE-AVX2 -O3 -c myfile.c++
```

The craype-mic-knl module ensures the compiler wrappers will link knl-optimized libraries

KNL Mesh Interconnect



Mesh of Rings

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

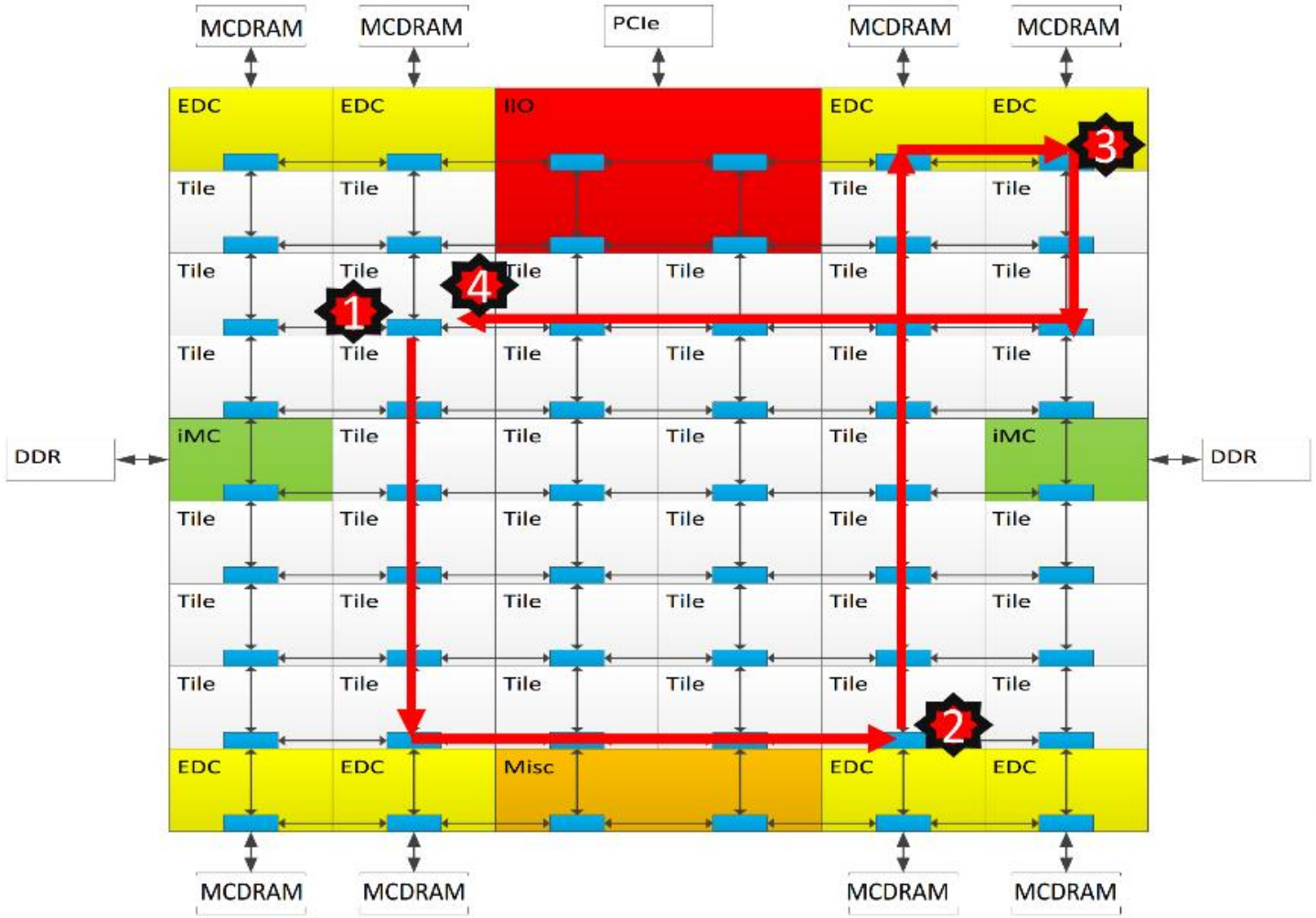
Cache Coherent Interconnect

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

Three Cluster Modes

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

Cluster Mode: All-to-All



Address uniformly hashed across all distributed directories

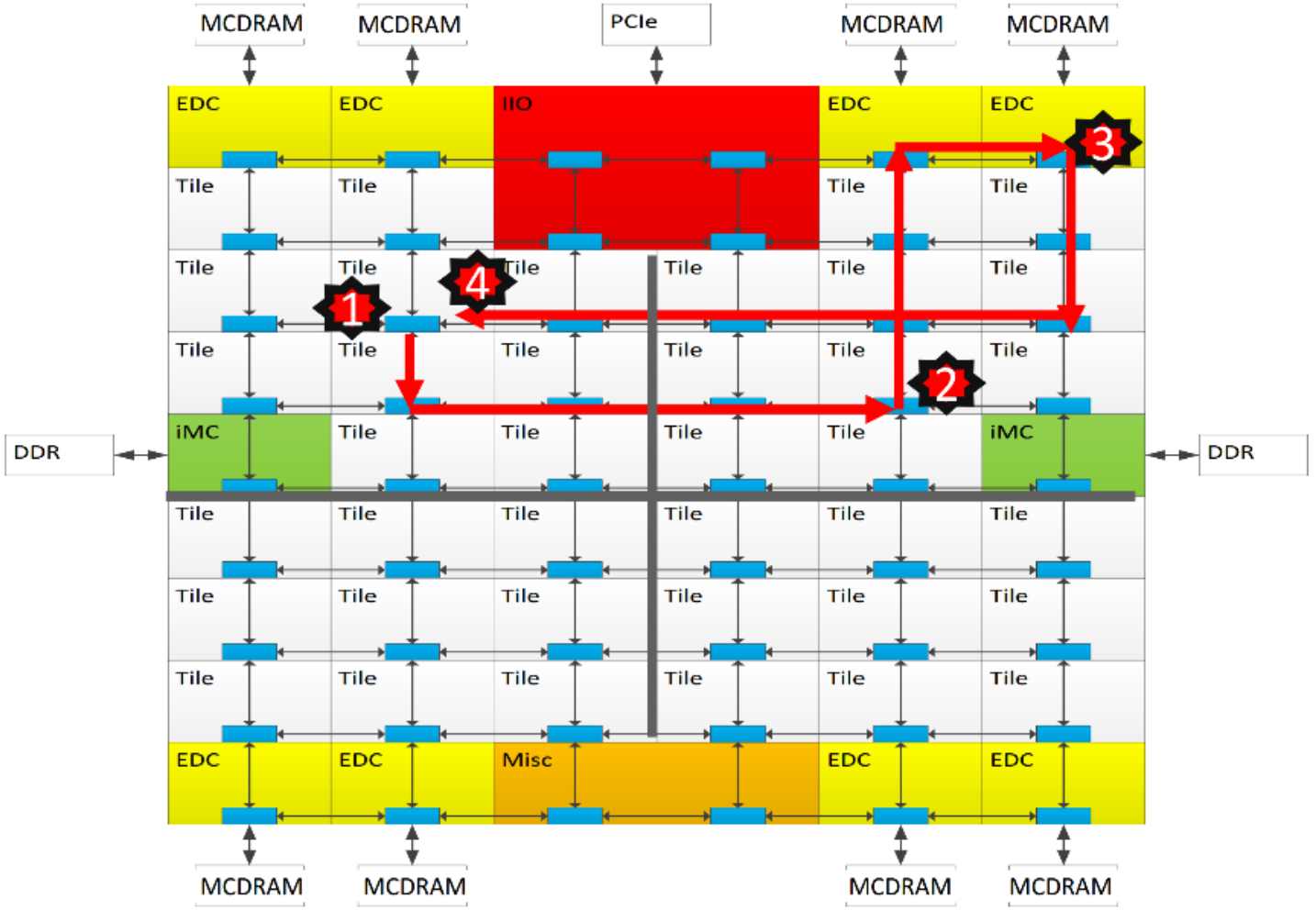
No affinity between Tile, Directory and Memory

Most general mode. Lower performance than other modes.

Typical Read L2 miss

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

Cluster Mode: Quadrant



Chip divided into four virtual Quadrants

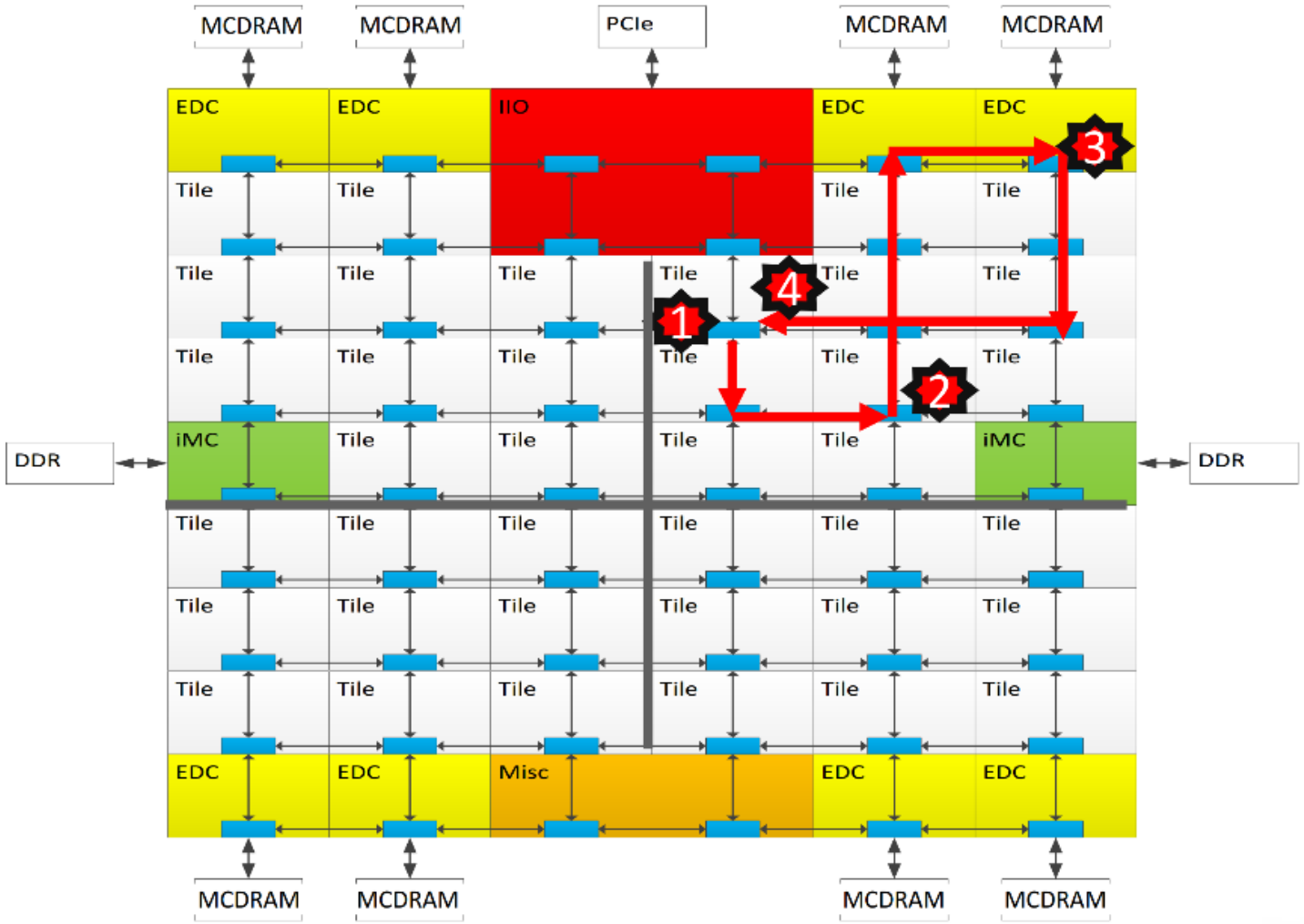
Address hashed to a Directory in the same quadrant as the Memory

Affinity between the Directory and Memory

Lower latency and higher BW than all-to-all. SW Transparent.

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

Cluster Mode: Sub-NUMA Clustering (SNC)



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

Looks analogous to 4-Socket Xeon

Affinity between Tile, Directory and Memory

Local communication. Lowest latency of all modes.

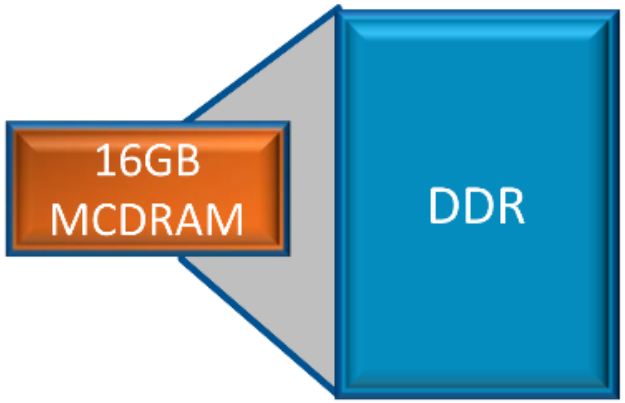
SW needs to NUMA optimize to get benefit.

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

KNL Memory Modes

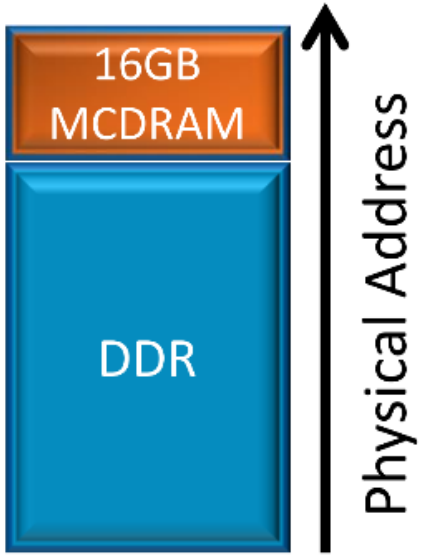
Three Modes. Selected at boot

Cache Mode



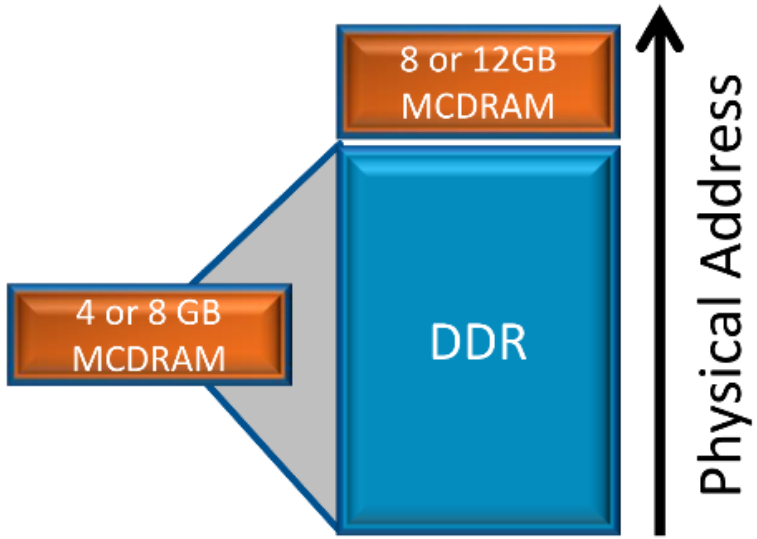
- SW-Transparent, Mem-side cache
- Direct mapped. 64B lines.
- Tags part of line
- Covers whole DDR range

Flat Mode



- MCDRAM as regular memory
- SW-Managed
- Same address space

Hybrid Mode



- Part cache, Part memory
- 25% or 50% cache
- Benefits of both

Cori KNL Configurations

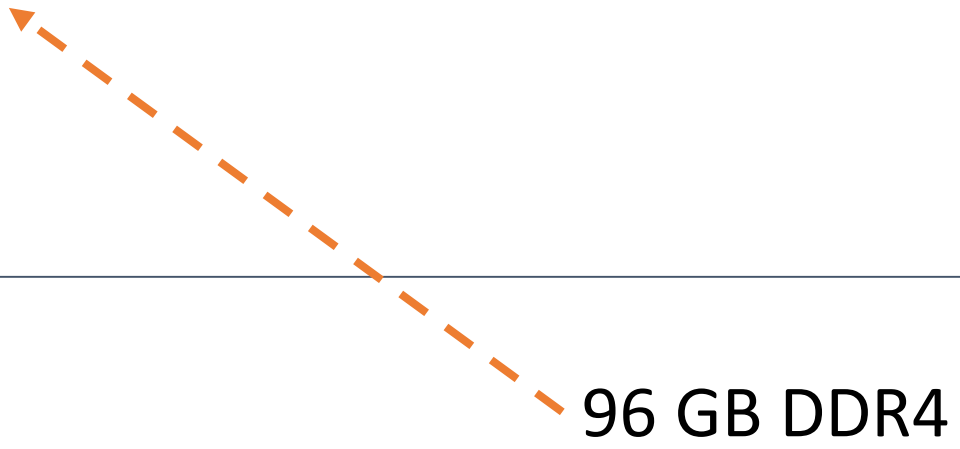
Cori only supports a sane subset of all possible KNL configurations. These are choices at boot time and each combination has pros/cons.

- Cluster Modes
  - Quad
  - SNC2
  - SNC4
- Memory Modes
  - Cache
  - Flat

http://www.nersc.gov/users/computational-systems/cori/configuration/knl-processor-modes/

Quad + Cache

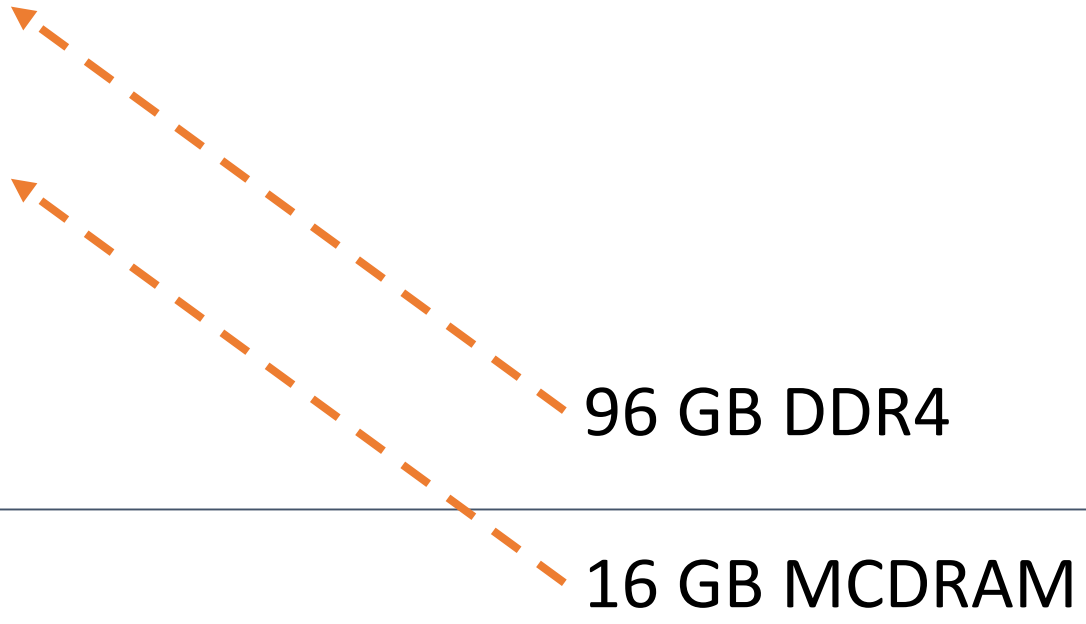
```
cori08:> salloc -N 1 -p debug -t 00:10:00 -L SCRATCH -C knl,quad,cache

nid06527:~> numactl --hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
... 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271
node 0 size: 96762 MB
node 0 free: 92936 MB
node distances:
node   0
  0:  10
```



```
nid06527:~> lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                272
On-line CPU(s) list:   0-271
Thread(s) per core:    4
Core(s) per socket:    68
Socket(s):             1
NUMA node(s):          1
Model name:            Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz
Stepping:              1
CPU MHz:               1401.000
CPU max MHz:           1401.0000
CPU min MHz:           1000.0000
BogoMIPS:              2799.81
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
NUMA node0 CPU(s):     0-271
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu
pni pclmulqdq dtes64 monitor ds_cpl est tm2 ssse3 fma cx16 xtpr pdcm sse4_1 sse4_2 x2apic
movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch ida arat
epb xsaveopt pln pts dtherm fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms avx512f rdseed adx
avx512pf avx512er avx512cd
```

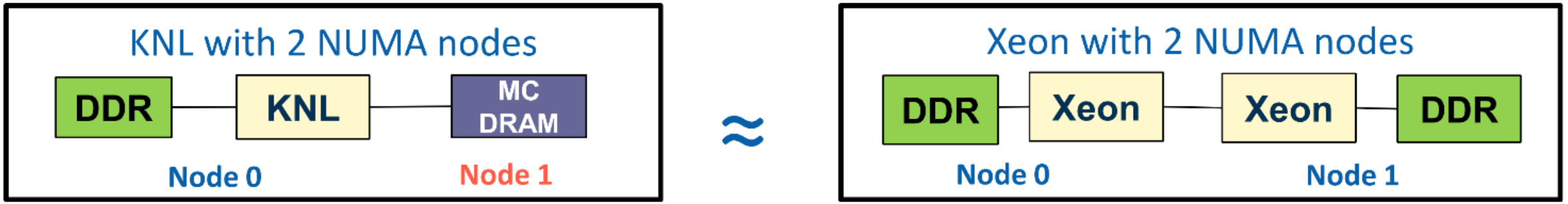
Quad + Flat

```
cori08:> salloc -N 1 -p debug -t 00:10:00 -L SCRATCH -C knl,quad,flat

nid12395:> numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
... 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271
node 0 size: 96764 MB
node 0 free: 93363 MB
node 1 cpus:
node 1 size: 16157 MB
node 1 free: 16092 MB
node distances:
node   0   1
  0:  10  31
  1:  31  10
```



Flat MCDRAM: SW Architecture

MCDRAM exposed as a separate NUMA node



Memory allocated in DDR by default → Keeps non-critical data out of MCDRAM.

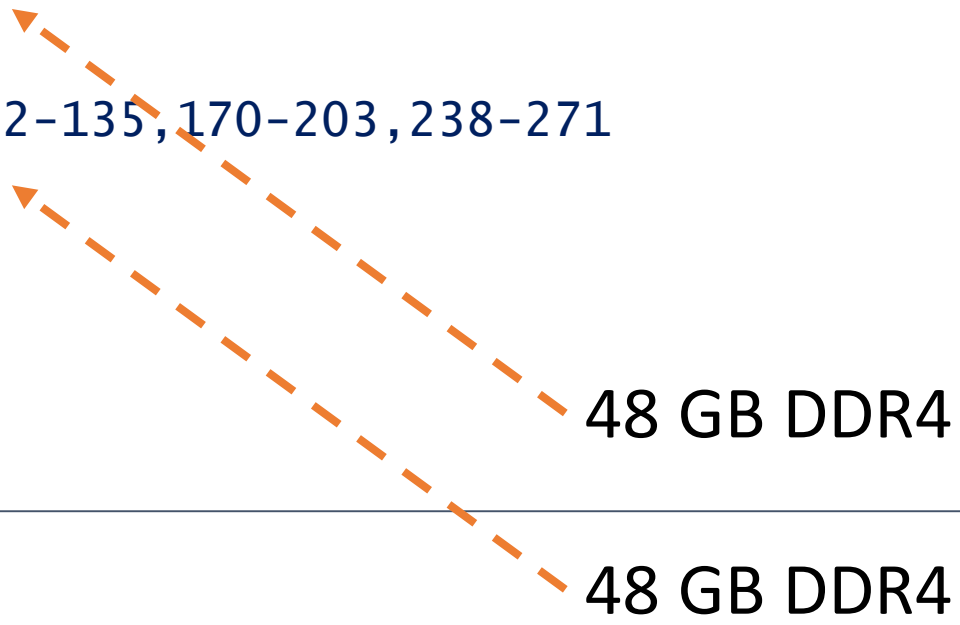
Apps explicitly allocate critical data in MCDRAM. Using two methods:

- **"Fast Malloc"** functions in High BW library (https://github.com/memkind/memkind)
  - Built on top to existing *libnuma* API
- **"FASTMEM"** Compiler Annotation for Intel Fortran

Flat MCDRAM with existing NUMA support in Legacy OS

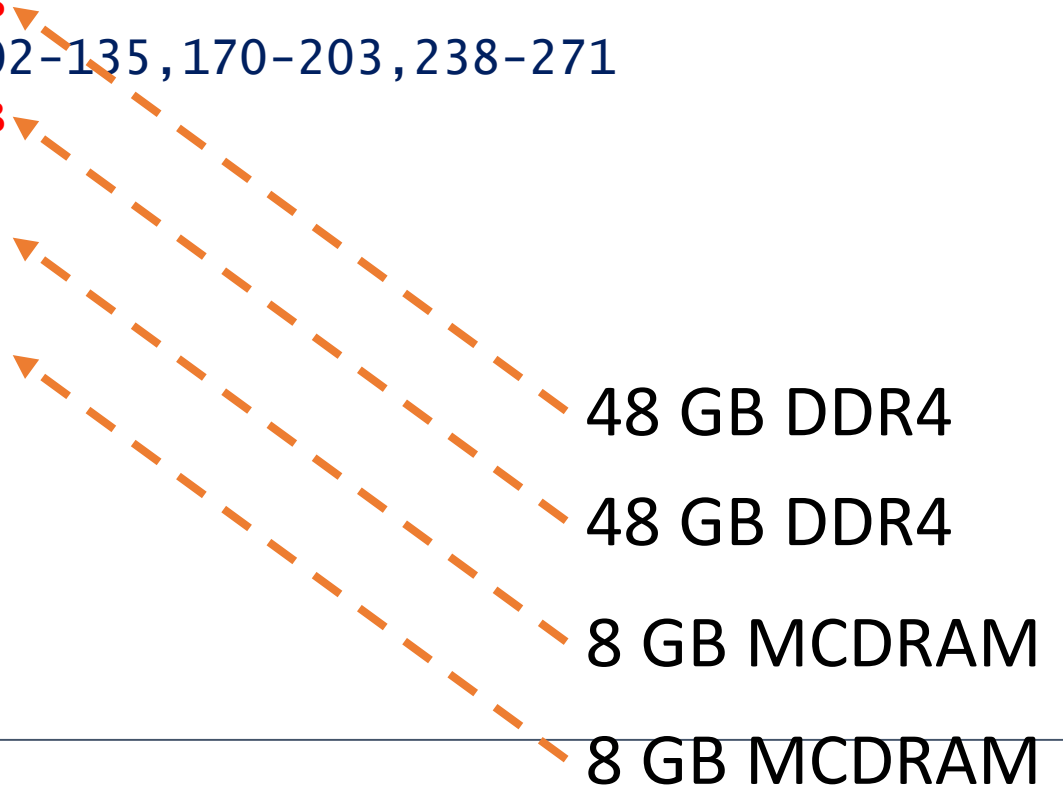
SNC2 + Cache

```
cori08:> salloc -N 1 -p debug -t 00:10:00 -L SCRATCH -C knl,snc2,cache

nid11099:~> numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0-33,68-101,136-169,204-237
node 0 size: 48298 MB
node 0 free: 46338 MB
node 1 cpus: 34-67,102-135,170-203,238-271
node 1 size: 48463 MB
node 1 free: 45834 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```



SNC2 + Flat

```
cori08:> salloc -N 1 -p debug -t 00:10:00 -L SCRATCH -C knl,snc2,flat

nid09582:~> numactl -H
available: 4 nodes (0-3)
node 0 cpus: 0-33,68-101,136-169,204-237
node 0 size: 48298 MB
node 1 cpus: 34-67,102-135,170-203,238-271
node 1 size: 48466 MB
node 2 cpus:
node 2 size: 8079 MB
node 3 cpus:
node 3 size: 8077 MB
node distances:
node   0   1   2   3
  0:  10  21  31  41
  1:  21  10  41  31
  2:  31  41  10  41
  3:  41  31  41  10
```



SNC4 + Cache

```
cori08:> salloc -N 1 -p debug -t 00:10:00 -L SCRATCH -C knl,snc4,cache

nid12253:~> numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0-17,68-85,136-153,204-221
node 0 size: 24065 MB
node 1 cpus: 18-35,86-103,154-171,222-239
node 1 size: 24232 MB
node 2 cpus: 36-51,104-119,172-187,240-255
node 2 size: 24233 MB
node 3 cpus: 52-67,120-135,188-203,256-271
node 3 size: 24230 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
```

SNC4 + Flat

```
cori08:> salloc -N 1 -p debug -t 00:10:00 -L SCRATCH -C knl,snc4,flat

nid04893:~> numactl -H
available: 8 nodes (0-7)
node 0 cpus: 0-17,68-85,136-153,204-221
node 0 size: 24065 MB
node 1 cpus: 18-35,86-103,154-171,222-239
node 1 size: 24232 MB
node 2 cpus: 36-51,104-119,172-187,240-255
node 2 size: 24233 MB
node 3 cpus: 52-67,120-135,188-203,256-271
node 3 size: 24233 MB
node 4 cpus:
node 4 size: 4039 MB
node 5 cpus:
node 5 size: 4039 MB
node 6 cpus:
node 6 size: 4039 MB
node 7 cpus:
node 7 size: 4037 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  31  41
  3:  21  21  21  10  41  41  41  31
  4:  31  41  41  41  10  41  41  41
  5:  41  31  41  41  41  10  41  41
  6:  41  41  31  41  41  41  10  41
  7:  41  41  41  31  41  41  41  10
```

Application Porting and Performance

Many applications will need code modifications in order to run efficiently on Cori's Intel Xeon Phi "Knights Landing" manycore processors.

Applications need to have

- good thread scalability to take advantage of the 68-core Xeon Phi processors,
- a data structure layout that can effectively use the 16 GB of onboard MCDRAM fast memory,
- and loop structures that exploit the 512-bit vector units.

http://www.nersc.gov/users/computational-systems/cori/application-porting-and-performance/

MPI + OpenMP

- There are 68 cores per KNL processor, with each core having the ability to support 4 hardware threads, resulting in up to 272 threads per single-socket node.
- Good thread scaling is important for an application to get good performance on KNL nodes.
- A hybrid MPI+OpenMP programming model is recommend
  - MPI targeting inter-node communication
  - OpenMP targeting on-node parallelism
  - Can reduce memory usage, by having fewer MPI tasks per node
  - Can send few MPI messages, with larger message sizes
- The best combination of MPI tasks and threads per task should be determined by experimentation.

http://www.nersc.gov/users/computational-systems/cori/application-porting-and-performance/improving-openmp-scaling/

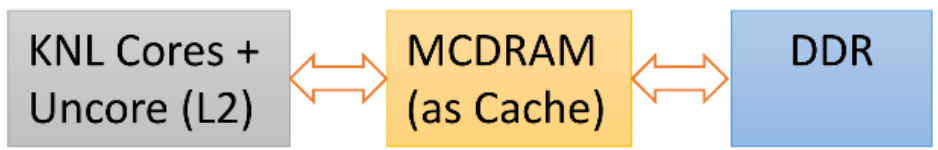
Example Batch Script: pure MPI in quad cache Mode

```
#!/bin/bash -l
#SBATCH -N 2
#SBATCH -p debug
#SBATCH -C knl,quad,cache
#SBATCH -t 30:00
#SBATCH -L SCRATCH

# only needed for hybrid MPI/OpenMP codes built with "-qopenmp" flag
export OMP_NUM_THREADS=1

# Add the following "sbcast" line here for jobs larger than 1500 MPI tasks:
# sbcast ./mycode.exe /tmp/mycode.exe

# 68 MPI ranks per node for a total of 136 MPI tasks
srun -n 136 ./mycode.exe
```

http://www.nersc.gov/users/computational-systems/cori/running-jobs/example-batch-scripts-for-knl/

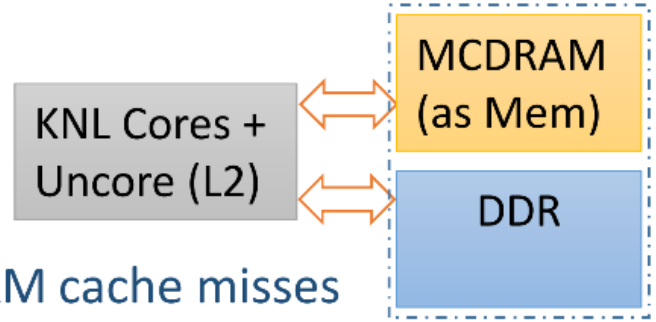
MCDRAM as Cache

- Upside
  - No software modifications required
  - Bandwidth benefit (over DDR)
- Downside
  - Higher latency for DDR access
    - i.e., for cache misses
  - Misses limited by DDR BW
  - All memory is transferred as:
    - DDR -> MCDRAM -> L2
  - Less addressable memory



MCDRAM as Flat Mode

- Upside
  - Maximum BW
  - Lower latency
    - i.e., no MCDRAM cache misses
  - Maximum addressable memory
  - Isolation of MCDRAM for high-performance application use only
- Downside
  - 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



https://software.intel.com/sites/default/files/managed/5f/5e/MCDRAM_Tutorial.pdf

Accessing MCDRAM in Flat Mode

- Option A: Using numactl
  - Works best if the whole app can fit in MCDRAM
- Option B: Using libraries
  - Memkind Library
    - Using library calls or Compiler Directives (Fortran*)
    - Needs source modification
  - AutoHBW (interposer library based on memkind)
    - No source modification needed (based on size of allocations)
    - No fine control over *individual* allocations
- Option C: Direct OS system calls
  - mmap(1), mbind(1)
  - Not the preferred method
    - Page-only granularity, OS serialization, no pool management

https://software.intel.com/sites/default/files/managed/5f/5e/MCDRAM_Tutorial.pdf

Option A: Using numactl to Access 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* <*your_command*>
    - Where *mcdram_id* is the ID of MCDRAM "node"
- 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* <*your_command*>
      - Allocations that don't fit into MCDRAM spills over to DDR
    - numactl --interleave=*nodes* <*your_command*>
      - Allocations are interleaved across all *nodes*

https://software.intel.com/sites/default/files/managed/5f/5e/MCDRAM_Tutorial.pdf

Example Batch Script: MPI/OpenMP in quad flat Mode

```
#!/bin/bash -l
#SBATCH -N 2
#SBATCH -p regular
#SBATCH -C knl,quad,flat
#SBATCH -S 4                  # use core specialization
#SBATCH -t 3:00:00
#SBATCH -L SCRATCH,project

export OMP_NUM_THREADS=2      # 2 OpenMP threads per MPI task
export OMP_PROC_BIND=true     #"spread" is also good for Intel and CCE compilers
export OMP_PLACES=threads

# Add the following "sbcast" line here for jobs larger than 1500 MPI tasks:
# sbcast ./mycode.exe /tmp/mycode.exe

# 32 MPI ranks per node for a total of 64 MPI tasks
# -p 1: prefer binding memory to NUMA domain 1
srun -n 64 -c 8 --cpu_bind=cores numactl -p 1 ./mycode.exe
```

http://www.nersc.gov/users/computational-systems/cori/running-jobs/example-batch-scripts-for-knl/

Recall: Quad + Flat

```
cori08:> salloc -N 1 -p debug -t 00:10:00 -L SCRATCH -C knl,quad,flat

nid12395:> numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
... 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271
node 0 size: 96764 MB
node 0 free: 93363 MB
node 1 cpus:
node 1 size: 16157 MB <- - - - - - - - - MCDRAM
node 1 free: 16092 MB
node distances:
node   0   1
  0:  10  31
  1:  31  10
```

Option B.1: Using Memkind Library to Access MCDRAM

Allocate 1000 floats from DDR

```
float   *fv;

fv = (float *)malloc(sizeof(float) * 1000);
```

Allocate 1000 floats from MCDRAM

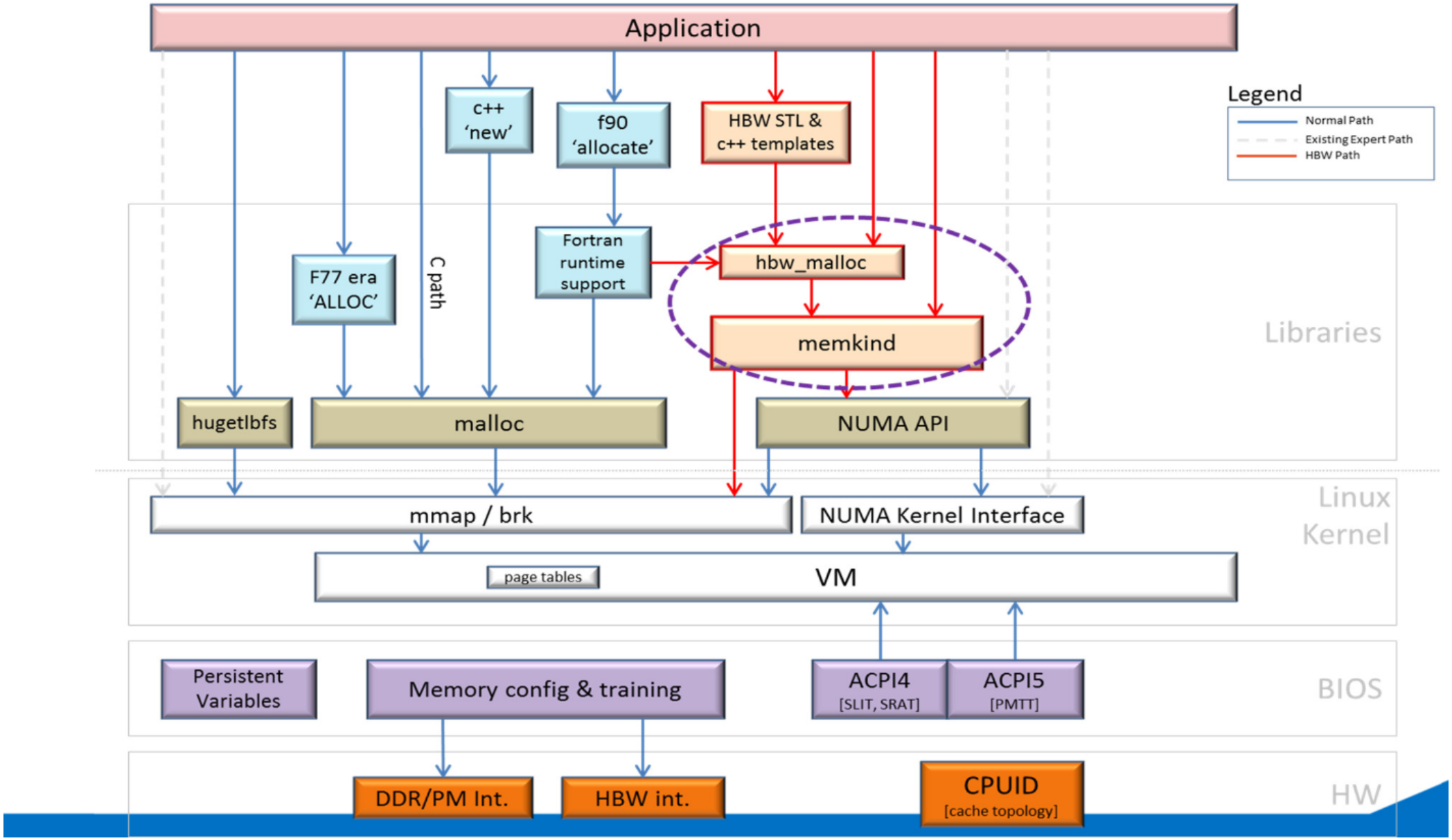
```
#include <hbwmalloc.h>

float   *fv;

fv = (float *)hbw_malloc(sizeof(float) * 1000);
```

Allocate arrays from MCDRAM and DDR in Intel® Fortran Compiler

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

!DEC$ ATTRIBUTES FASTMEM :: A

       NSIZE=1024
c
c      allocate array 'A' from MCDRAM
c
       ALLOCATE (A(1:NSIZE))
c
c      Allocate arrays that will come from DDR
c
       ALLOCATE  (B(NSIZE), C(NSIZE))
```

https://software.intel.com/sites/default/files/managed/5f/5e/MCDRAM_Tutorial.pdf

Memkind Architecture



https://software.intel.com/sites/default/files/managed/5f/5e/MCDRAM_Tutorial.pdf

Building with libmemkind

- Memkind: http://memkind.github.io/memkind
- C/C++:
  On Cori, load the memkind module:
  module load memkind

  Compilers wrappers (cc & CC) will add
  -lmemkind -ljemalloc -lnuma
- Fortran: currently only Intel compiler support FASTMEM directive, e.g.
  !DIR$ ATTRIBUTES FASTMEM :: a, b, c
  or
  !DEC$ ATTRIBUTES FASTMEM :: a, b, c

Syntax Rules for Compiler Directives: https://software.intel.com/en-us/node/678870

Memkind Policies and Memory Types

- How do we make sure we get memory only from MCDRAM?
  - This depends on POLICY
    - See man page and hbw_set_policy() / hbw_get_policy()
  - BIND : Will cause app to die when it runs out of MCDRAM
  - PREFERRED : Will allocate from DDR if MCDRAM not sufficient (default)
- Allocating 2 MB and 1 GB pages
  - Use hbw_posix_memalign_psize()

https://software.intel.com/sites/default/files/managed/5f/5e/MCDRAM_Tutorial.pdf

Option B.2: AutoHBW

- 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
- Demo
  - see /examples/autohbw_test.sh
  - Run gpp with AutoHBW
- Environment variables (see autohbw_README)
  - `AUTO_HBW_SIZE=x[:y]`
  - `AUTO_HBW_LOG=level`
  - `AUTO_HBW_MEM_TYPE=memory_type` *# useful for interleaving*
- Finding source locations of arrays
  - `export AUTO_HBW_LOG=2`
  - `./app_name > log.txt`
  - `autohbw_get_src_lines.pl log.txt app_name`

https://software.intel.com/sites/default/files/managed/5f/5e/MCDRAM_Tutorial.pdf

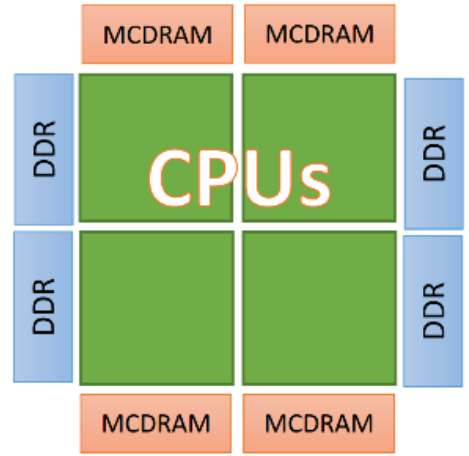
AutoHBM: Automatic memkind

- Uses *array* size to determine whether an array should be allocated to HBM (High Bandwidth Memory, a.k.a., MCDRAM)
- No code changes is necessary
  module load autohbw
  Compilers wrappers (cc, CC & ftn) will add -lautohbw
- Runtime environment variables

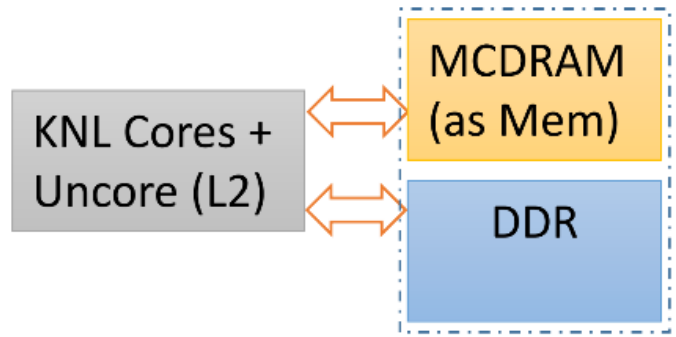
```
# any allocation > 4KB will be placed in HBM
export AUTO_HBW_SIZE=4K
# allocations between 4KB and 8KB will be placed in HBM
export AUTO_HBW_SIZE=4K:8K
```

Advanced Topic: MCDRAM in SNC4 Mode

- SNC4: Sub-NUMA Clustering
  - KNL die is divided into 4 clusters (similar to a 4-Socket Haswell)
    - SNC4 configured at boot time
    - Use numactl --hardware to find out nodes and distances
      - There are **4 DDR (+CPU) nodes + 4 MCDRAM (no CPU) nodes**, in flat mode



- Running 4-MPI ranks is the easiest way to utilize SNC4
  - Each rank allocates from closest DDR node
    - If a rank allocates MCDRAM, it goes to closest MCDRAM node



Compare with 2 NUMA nodes

- If you run only 1 MPI rank and use numactl to allocate on MCDRAM
  - Specify all MCDRAM nodes
    - E.g., numactl –m 4,5,6,7

https://software.intel.com/sites/default/files/managed/5f/5e/MCDRAM_Tutorial.pdf

Recall: SNC4 + Flat

```
cori08:> salloc -N 1 -p debug -t 00:10:00 -L SCRATCH -C knl,snc4,flat

nid04893:~> numactl -H
available: 8 nodes (0-7)
node 0 cpus: 0-17,68-85,136-153,204-221
node 0 size: 24065 MB
node 1 cpus: 18-35,86-103,154-171,222-239
node 1 size: 24232 MB
node 2 cpus: 36-51,104-119,172-187,240-255
node 2 size: 24233 MB
node 3 cpus: 52-67,120-135,188-203,256-271
node 3 size: 24233 MB
node 4 cpus:
node 4 size: 4039 MB
node 5 cpus:
node 5 size: 4039 MB
node 6 cpus:
node 6 size: 4039 MB
node 7 cpus:
node 7 size: 4037 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  31  41
  3:  21  21  21  10  41  41  41  31
  4:  31  41  41  41  10  41  41  41
  5:  41  31  41  41  41  10  41  41
  6:  41  41  31  41  41  41  10  41
  7:  41  41  41  31  41  41  41  10
```

Example Batch Script: MPI/OpenMP in snc4 flat Mode

```
#!/bin/bash -l
#SBATCH -N 2
#SBATCH -p regular
#SBATCH -C knl,snc4,flat
#SBATCH -S 4                # use core specialization
#SBATCH -t 3:00:00
#SBATCH -L SCRATCH,project

export OMP_NUM_THREADS=16 # 16 OpenMP threads per MPI task, using 64 cores in total
export OMP_PROC_BIND=true #"spread" is also good for Intel and CCE compilers
export OMP_PLACES=threads

# Add the following "sbcast" line here for jobs larger than 1500 MPI tasks:
# sbcast ./mycode.exe /tmp/mycode.exe

# 4 MPI ranks per node for a total of 8 MPI tasks
# executable linked with libmemkind
srun -n 8 -c 68 --cpu_bind=cores ./mycode.exe
```

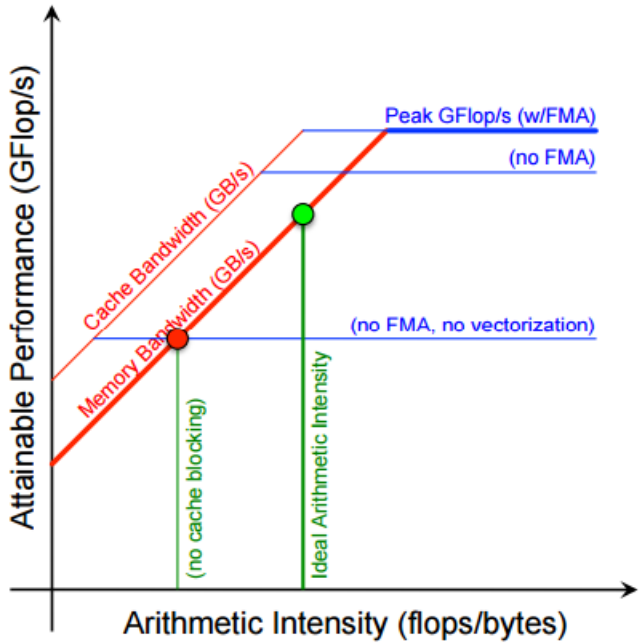
http://www.nersc.gov/users/computational-systems/cori/running-jobs/example-batch-scripts-for-knl/

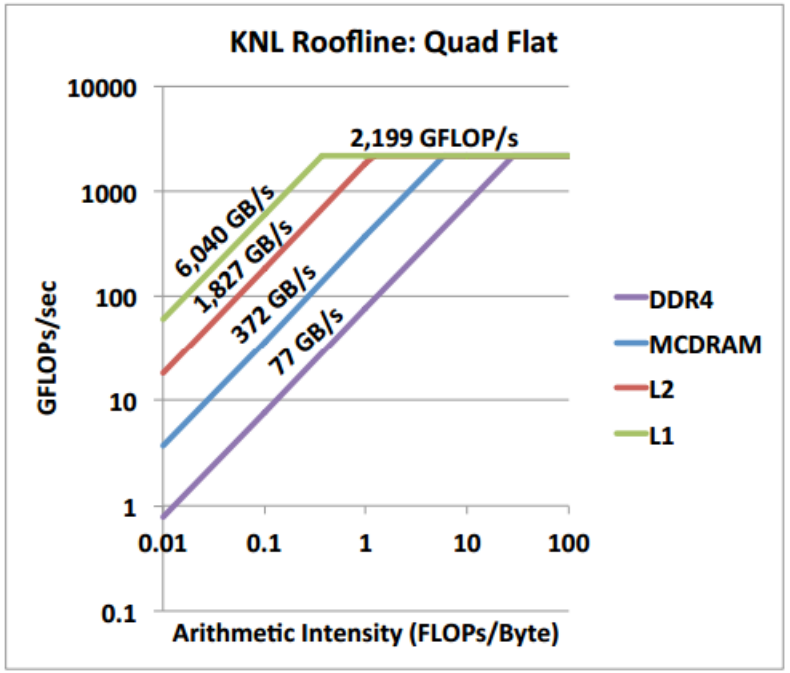
Vectorization

- To achieve high performance on Cori's KNL nodes, **vectorization** is a key component
- Each core on "Knights Landing" Xeon Phi processor has two 512-bit VPUs (Vector Processing Units), supporting AVX-512 extensions

  AVX-512 : 8 double-precision words, or 16 single-precision words, per instruction

  16 DP / 32 FP FLOP per cycle per VPU





https://crd.lbl.gov/assets/Uploads/ixpug16-roofline.pdf

Vectorization

To vectorize a loop, the compiler first unrolls the loop by the vector length, and then packs multiple scalar instructions into a single vector instruction.

```
DO I = 1, N
  Z(I) = X(I) + Y(I)
ENDDO
```

Code Transformation

```
DO I = 1, N, 8
  Z(I) = X(I) + Y(I)
  Z(I+1) = X(I+1) + Y(I+1)
  Z(I+2) = X(I+2) + Y(I+2)
  Z(I+3) = X(I+3) + Y(I+3)
  Z(I+4) = X(I+4) + Y(I+4)
  Z(I+5) = X(I+5) + Y(I+5)
  Z(I+6) = X(I+6) + Y(I+6)
  Z(I+7) = X(I+7) + Y(I+7)
ENDDO
```

```
DO I = 1, N, 8
  VLOAD X(I), X(I+1), ..., X(I+7)
  VLOAD Y(I), Y(I+1), ..., Y(I+7)
  VADD Z(I, ..., I+7) X+Y(I, ..., I+7)
  VSTORE Z(I), Z(I+1), ..., Z(I+7)
ENDDO
```

Vector Instructions

How to Vectorize Your Code?

- Auto-Vectorization analysis by the compiler
- Auto-Vectorization analysis by the compiler enhanced with directive – code annotations that suggest what can be vectorized
- Explicit vectorization using SIMD intrinsics (not portable)
- Explicit vectorization using OpenMP 4.5 SIMD pragmas
- Use vendor-supplied optimized libraries

https://www.nersc.gov/assets/Training-Materials/NERSC-VectorTrainingOct2014.pdf

Compiler Auto-Vectorization of Loops

- Compilers can auto-vectorize loops that are considered **safe** for vectorization.
- In case of the Intel compiler, this happens when you compile at default optimization level (-O2) or higher.
- On the other hand, if you want to disable vectorization for any loop in a source file for any reason, you can do that by specifying the '-no-vec' compile flag.

http://www.nersc.gov/users/computational-systems/cori/application-porting-and-performance/vectorization/

Vectorization Inhibitors

Not all loops can be vectorized

- The loop trip count must be known at entry to the loop at runtime.
- In general, branching in the loop inhibits vectorization.
- Only the *innermost* loop is eligible for vectorization.
- A function call or I/O inside a loop prohibits vectorization. Intrinsic math functions and inlined function are exceptions.
- There should be no data dependency in the loop. Dependency examples:
  - Read-after-write (flow dependency) – can't be vectorized
  - Write-after-read (anti-dependency) – can be vectorized
  - Write-after-write (output dependency) – can't be vectorized
- Non-contiguous memory access hampers vectorization efficiency.

http://www.nersc.gov/users/computational-systems/cori/application-porting-and-performance/vectorization/

Vector Analysis Tools

Intel provides two very useful vector analysis tools.

- The compiler option '-vec-report=<n>' will generate a diagnostic report about vectorization.

```
% ftn -o v.out -vec-report=3 yax.f
yax.f(12): (col. 10) remark: LOOP WAS VECTORIZED
yax.f(13): (col. 22) remark: loop was not vectorized: not inner loop
```

- The compiler option '-guide' causes Guided Auto-Parallelization (GAP) feature turned on. It generates parallelization diagnostics, suggesting ways to improve auto-vectorization, auto-parallelization, and data transformation.

```
% ftn -real-size 64 -vec-report=3 -parallel -guide -o v.out alg.F
alg.F(20): remark #30525: (PAR) Insert a "!dir$ loop count min(512)"
statement right before the loop at line 20 to parallelize the loop.
[VERIFY] Make sure that the loop has a minimum of 512 iterations.
```

OpenMP 4.5 SIMD Construct

OpenMP 4.0 standard now has the SIMD construct to specify the execution of a loop in vectorization mode (i.e., SIMD operations).

Fortran	`!$OMP SIMD [clause ...]`
	`  SAFELEN (length)`
	`  ALIGNED (list[:alignment])`
	`  REDUCTION (operator: list)`
	`  COLLAPSE (n)`
	`  do-loops`
	`!$OMP END SIMD`

C/C++	`#pragma omp simd [clause ...] newline`
	`  safelen (length)`
	`  aligned (list[:alignment])`
	`  reduction (operator: list)`
	`  collapse (n)`
	`  for-loops`

- OpenMP 4.5 Complete Specifications
  http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
- OpenMP 4.5 Examples
  http://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf

Example of the OMP SIMD construct

```
void simd_loop(double *a, double *b, double *c, int n)
{
  int i;

  #pragma omp simd
  for (i=0; i<n; i++)
    a[i] = b[i] + c[i];
}
```

Example of Using OMP FOR and OMP SIMD Constructs

```
#define N 1000000
float x[N][N], y[N][N];
#pragma omp parallel
{
  #pragma omp for
  for (int i=0; i<N; i++) {
    #pragma omp simd safelen(18)
    for (int j=18; j<N-18; j++) {
      x[i][j] = x[i][j-18] + sinf(y[i][j]);
      y[i][j] = y[i][j+18] + cosf(x[i][j]);
    }
  }
}
```

cannot be vectorized by the compiler without additional knowledge about vectorization and safety information for SIMD vector length selection.

Explicit Vector Programming with OpenMP 4.0 SIMD Extensions
http://primeurmagazine.com/repository/PrimeurMagazine-AE-PR-12-14-32.pdf

Example that uses the composite FOR SIMD construct

```
double compute_pi(int n)
{
  const double dH = 1.0 / (double) n;
  double dX, dSum = 0.0;

  #pragma omp parallel for simd private(dX) \
                          reduction(+:dSum) schedule(simd:static)
  for (int i=0; i<n; i++) {
    dX = dH * ((double) i + 0.5);
    dSum += (4.0 / (1.0 + dX * dX));
  } // End parallel for simd region

  return dH * dSum;
}
```

SIMD Construct for Functions

OpenMP 4.0 introduces the declare simd construct, which can be applied to a function (C, C++ and Fortran) or a subroutine (Fortran) to enable the creation of one or more versions that can process multiple instances of each argument using SIMD instructions from a single invocation from a SIMD loop.

Fortran	`!$OMP DECLARE SIMD (proc-name) [clause ...]`
C/C++	`#pragma omp declare simd [clause ...] newline` `  function definition or declaration`

- OpenMP 4.5 Complete Specifications
  http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
- OpenMP 4.5 Examples
  http://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf

Example of using DECLARE SIMD directives

```
#pragma omp declare simd
extern float sfoo(float);
void invokesfoo(float *restrict a, float *restrict x, int n)
{
  for (int i=0; i<n; i++) a[i] = sfoo(x[i]);
}

#pramga omp declare simd
float sfoo(float x) {
 … … … // function body
}
```

cannot be auto-vectorized by the compiler when the loop contains a call to a user-defined function

Explicit Vector Programming with OpenMP 4.0 SIMD Extensions
http://primeurmagazine.com/repository/PrimeurMagazine-AE-PR-12-14-32.pdf

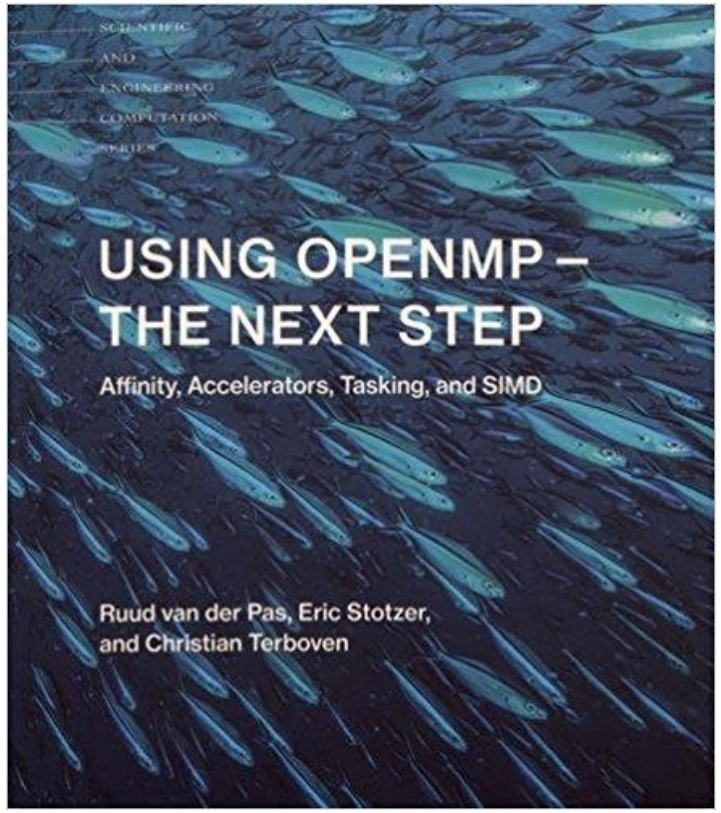
OpenMP 4 Book and Examples

- Book:

  Using OpenMP — The Next Step: Affinity, Accelerators, Tasking, and SIMD,
  by Ruud van der Pas, Eric Stotzer, & Christian Terboven,
  MIT Press, 2017
  https://www.ieeeexplore.ws/xpl/bkabstractplus.jsp?bkn=8169743

- Examples:

  https://github.com/shawfdong/ams250/tree/master/examples/openmp/simd



Further Readings

- **Intel Xeon Phi Processor High Performance Programming,** Knight Landing Edition, *by* Jim Jeffers, James Reinders & Avinash Sodani, Morgan Kaufmann, 2016 http://www.sciencedirect.com/science/book/9780128091944
- Cori Application Porting and Performance http://www.nersc.gov/users/computational-systems/cori/application-porting-and-performance/
- Cori KNL: Programming and Optimization http://www.nersc.gov/users/training/events/cori-knl-programming-and-optimization-214-2162017/
- NERSC Cori KNL Training http://www.nersc.gov/users/training/events/cori-knl-training/
- MCDRAM on Knights Landing: Analysis Methods & Tools https://software.intel.com/sites/default/files/managed/5f/5e/MCDRAM_Tutorial.pdf
- Tutorial on Intel Xeon Phi Processor Optimization https://software.intel.com/en-us/articles/tutorial-on-intel-xeon-phi-processor-optimization
- A Guide to Vectorization with Intel C++ Compilers https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf