TEXAS ADVANCED COMPUTING CENTER

WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

Introduction to Intel Xeon Scalable and Knights Landing Hardware

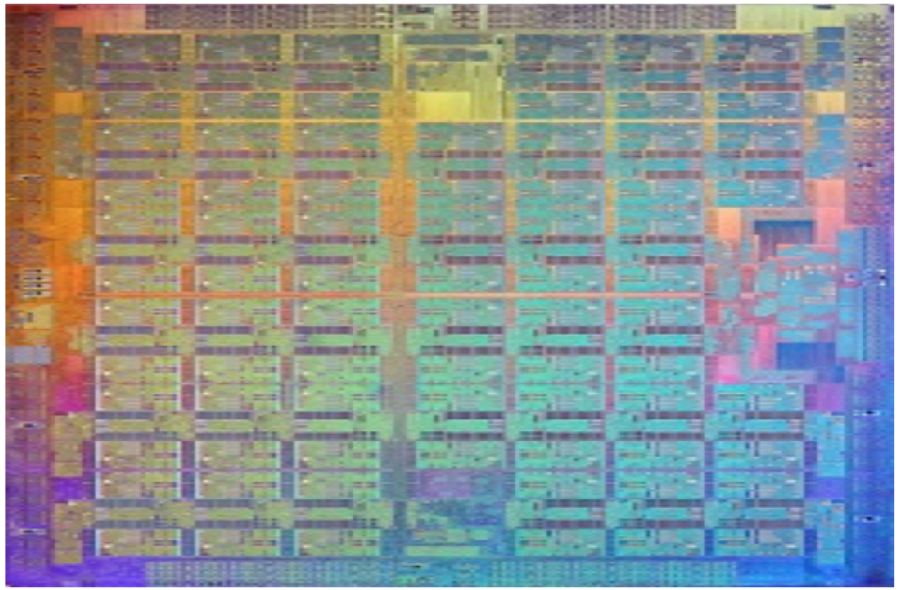
PRESENTED BY:

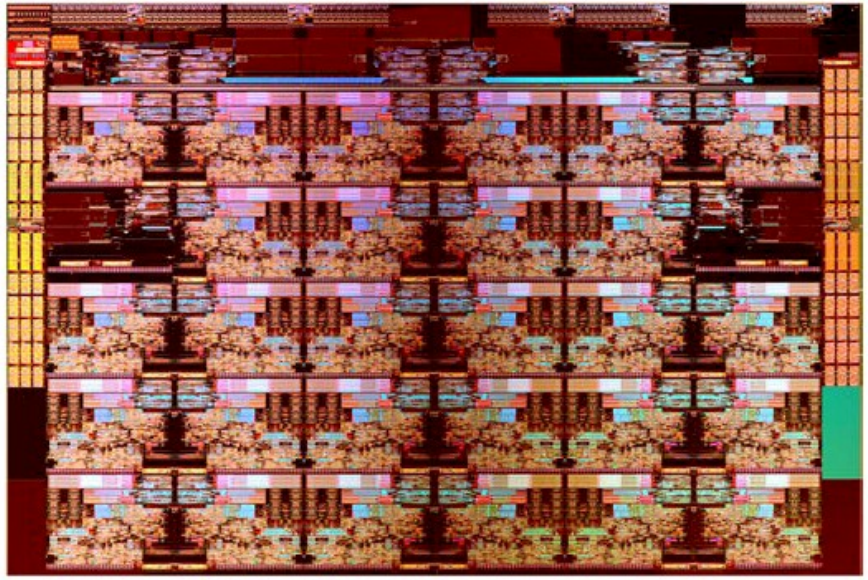
John Cazes

What is a Multicore or Manycore Processor?

Multicore & Manycore Definition

- Processor with multiple execution units (cores)
- Cores are capable of executing different instructions simultaneously
- Multicore vs. Manycore: no formal distinction, 10s vs. 100s

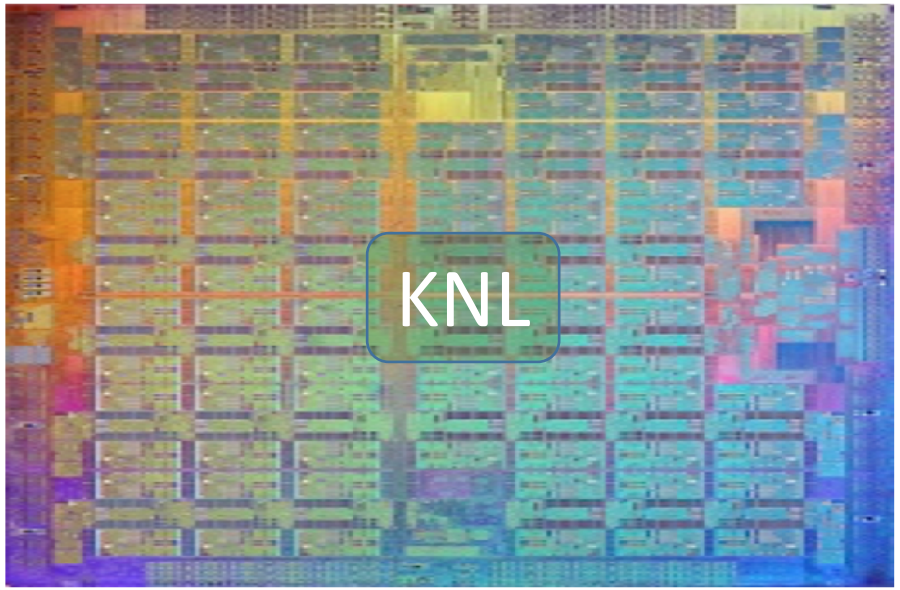




What is a Multicore or Manycore Processor?

Multicore & Manycore Definition

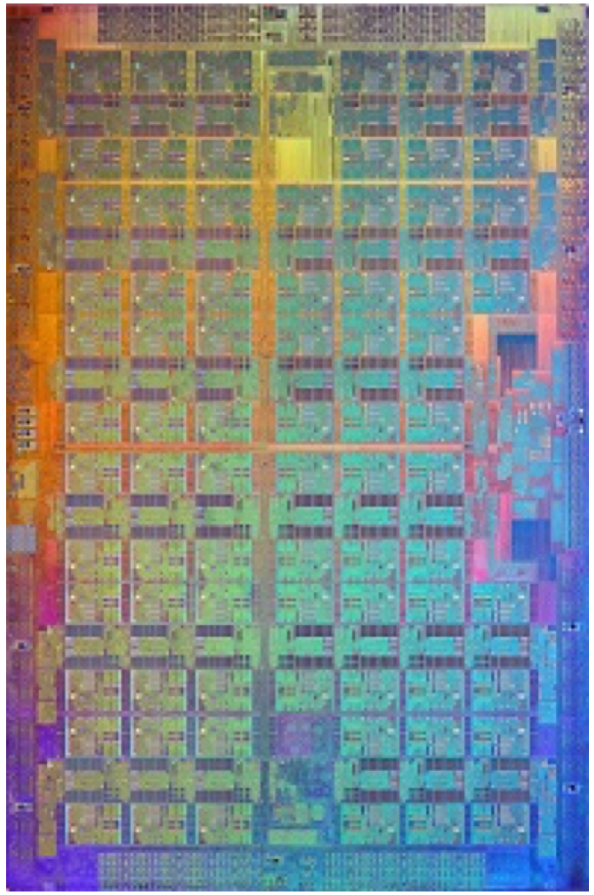
- Processor with multiple execution units (cores)
- Cores are capable of executing different instructions simultaneously
- Multicore vs. Manycore: no formal distinction, 10s vs. 100s



Intel Knights Landing

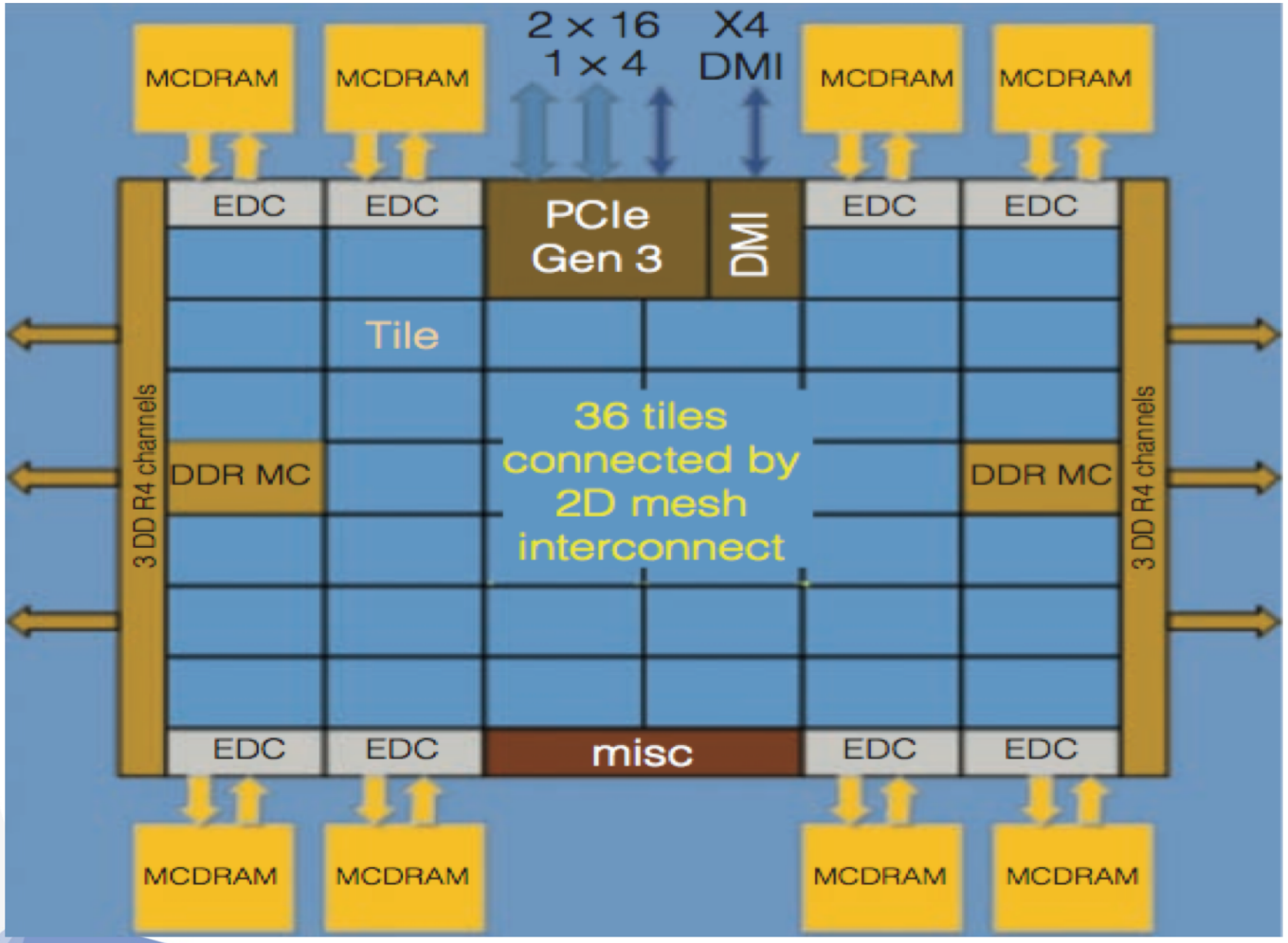
- Released mid 2016
- Leverages x86 architecture: binary compatible with Intel's mainline ISA
- Simpler x86 cores: reasonable serial performance with power efficiency
- Supports standard programming models
  - Fortran, C/C++
  - MPI, OpenMP, pthreads
- Designed for floating point performance
- Provides high memory bandwidth & capacity
  - DDR memory provides capacity
  - MCDRAM provides bandwidth
- Runs an operating system: "self-hosted"

KNL Architecture



- Up to 72 cores (based on Silvermont Atom)
  - Local L1 cache
  - 4 hardware-threads per core
  - Possible 288 threads
- 36 tiles, 2 cores per tile, share L2 cache
- 2D mesh interconnect
- 16 GB MCDRAM (Multi-channel) on-package
- Up to 384 GB DDR4 (96 GB on Stampede2)
- Single socket - self-hosted
- 3+/6+ TFLOP DP/SP peak performance
- 450+ GB/s STREAM performance
- Supports Intel Omni-Path Frabric

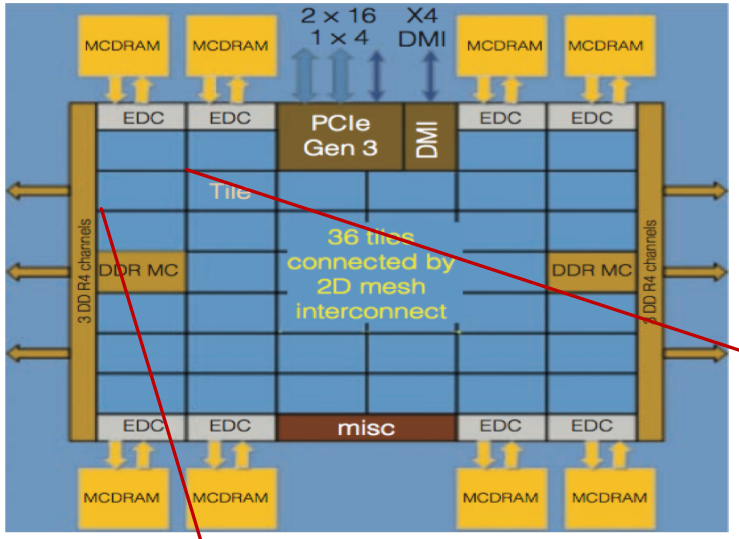
KNL Diagram

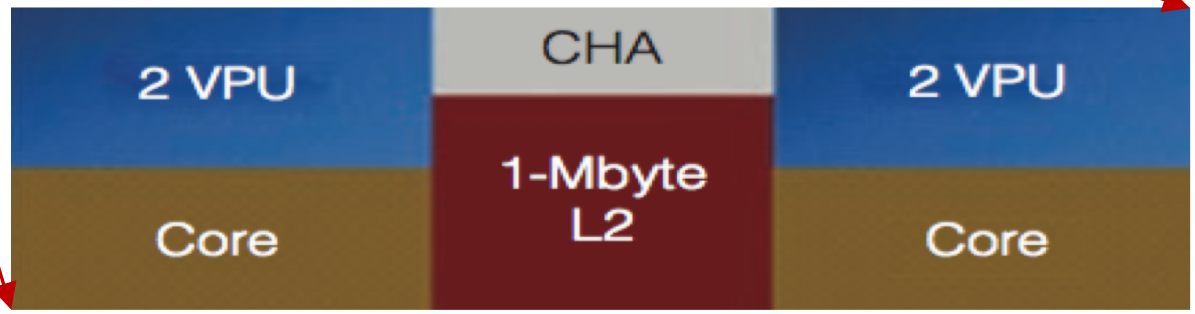


- Up to 36 tiles
  - 2 cores/tile
- 2D mesh interconnect
- 2 DDR memory controllers
- 6 channels DDR4
  - Up to 90 GB/s
- 16 GB MCDRAM
  - 8 embedded DRAM controllers
  - Up to 450 GB/s

(*KNIGHTS LANDING: SECOND- GENERATION INTEL XEON PHI PRODUCT* ,
A. Sodani, et.al.,IEEE Micro March/April 2016)

KNL Tile





- 2 cores/tile
- Shared mesh connection
- 1 MB shared L2 cache
- 2 512-bit VPUs per core
- Based on Intel Atom Silvermont architecture

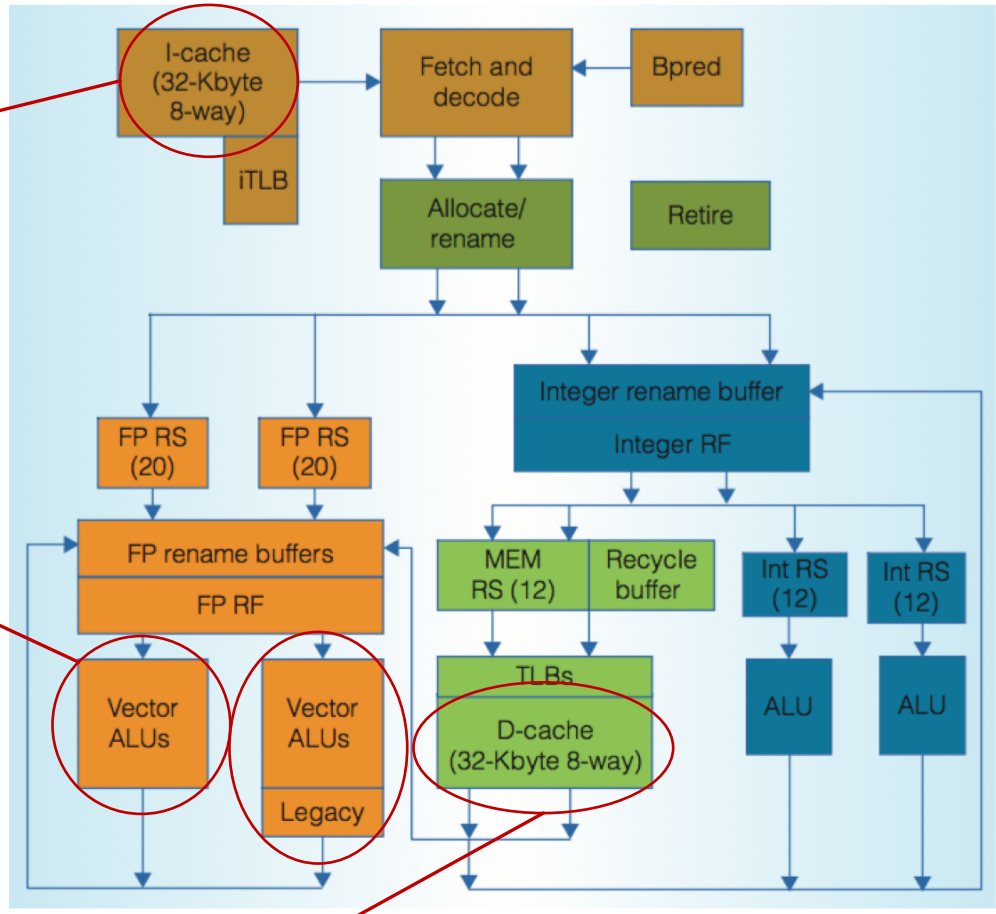
(*KNIGHTS LANDING: SECOND- GENERATION INTEL XEON PHI PRODUCT*, A. Sodani, et.al.,IEEE Micro March/April 2016)

KNL Core

8-way 32KB instruction cache

2 VPUs, only one has support for legacy floating point ops

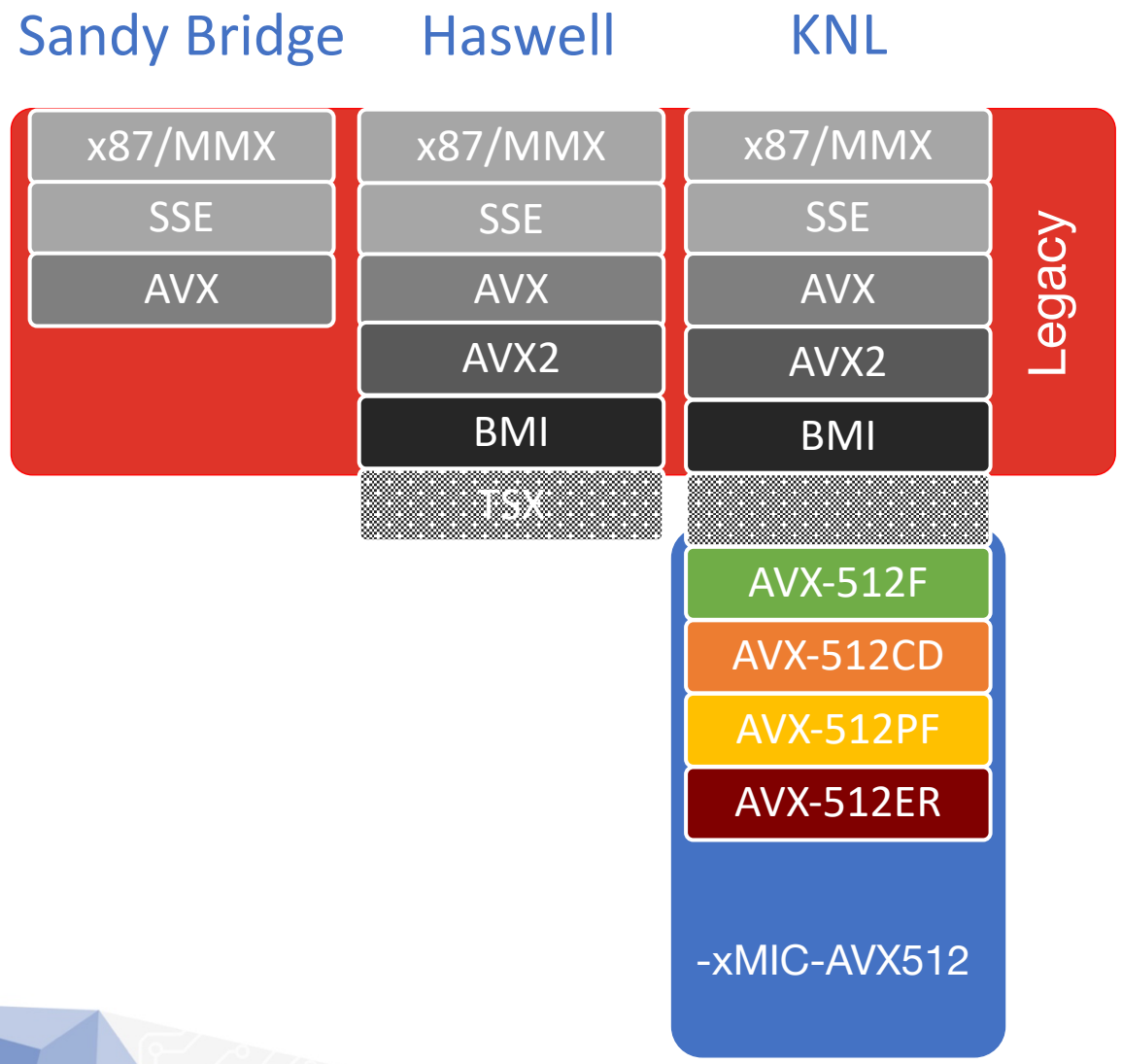
- Compile with **-xMIC-AVX512** to use both VPUs
  (Only supported by Intel compilers)



8-way 32KB data cache

(*KNIGHTS LANDING: SECOND- GENERATION INTEL XEON PHI PRODUCT*, A. Sodani, et.al.,IEEE Micro March/April 2016)

KNL ISA



- KNL supports all legacy instructions
- Introduces AVX-512 Extensions:
  - Foundations (common between Xeon and Xeon Phi)
  - Conflict Detection
  - Prefetch
  - Exponential and Reciprocal

AVX-512 ISA: http://goo.gl/TGQIKE

Tiles, Quadrants, and Directories

Memory hierarchy (configurable MCDRAM)

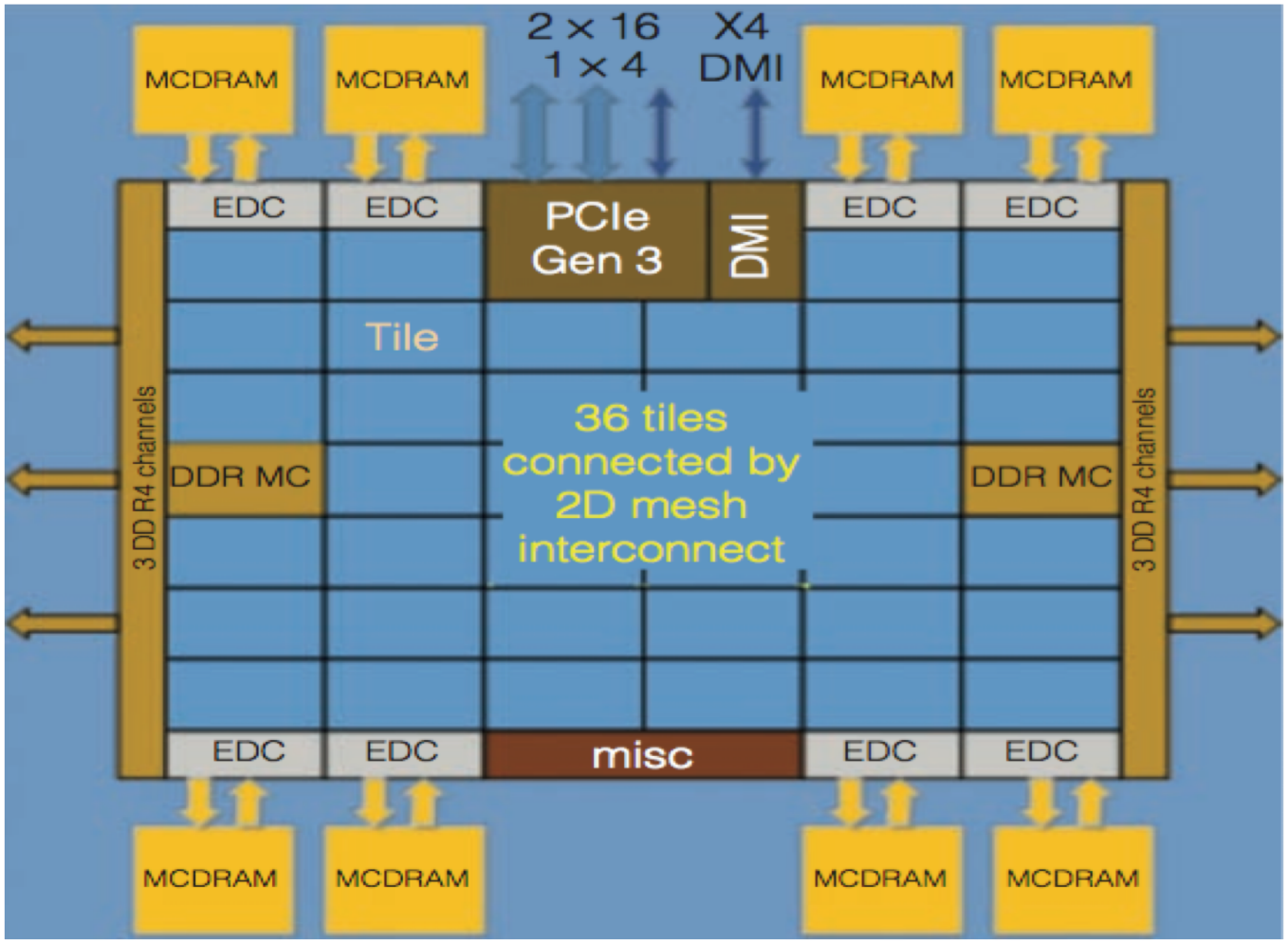
- DDR4 (96 GB) / MCDRAM (16 GB)
- Tile L2 (1 MB) / Core L1(32 KB)

A **tile** is a set of two cores sharing a 1MB L2 cache and connectivity to the mesh.

A **quadrant** is a virtual concept and not a hardware property. It is a way to divide the tiles at a logical level.

A **tag directory** tracks cache line locations in all L2s. It provides the block of data or (if not available in L2) a memory address to the memory controller.

KNL Diagram



(*KNIGHTS LANDING: SECOND- GENERATION INTEL XEON PHI PRODUCT* ,
A. Sodani, et.al.,IEEE Micro March/April 2016)

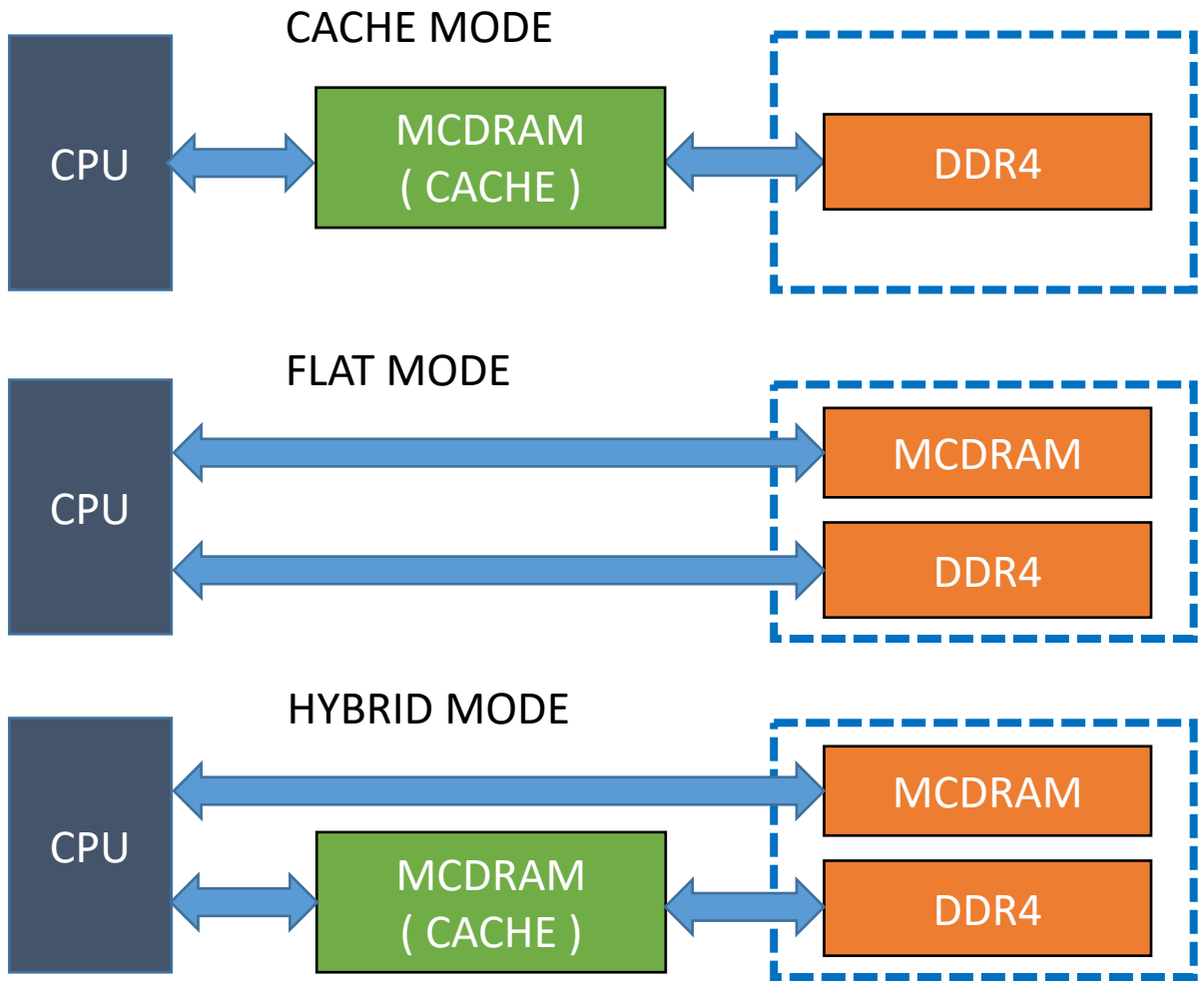
Cluster Modes

Cluster Modes determine L2 coherency traffic flow

- Five supported modes
  - All-to-all (A2A)
  - Quadrant (Quad)/Hemisphere
  - Sub-NUMA Cluster (SNC4/SNC2)
- Cluster modes modify the distance that coherency traffic flows go through in the mesh
- Remember these **quadrants** we mention are logical entities, not hardware features
- Performance differences between modes appears to be minimal
- Quadrant is easiest to use and is slightly better than all-to-all
  - Default cluster mode on Stampede2

Memory Architecture

- Two main memory types
  - DDR4
  - MCDRAM
- Three memory modes
  - Cache
  - Flat
  - Hybrid
- Hybrid mode
  - Three choices
  - 25% / 50% / 75%
  - 4GB / 8 GB / 12GB



Memory Modes

Memory footprint	Flat	Cache
< 16GB	Optimal	Couple of percent lower
>16 GB: many arrays	May work well. Requires code modifications (user identifies which arrays to store in MCDRAM)	May work well for codes that exhibit high memory bandwidth
>16 GB: few large arrays	Does not work, if no array fits in the MCDRAM without hand coded buffering	May work well

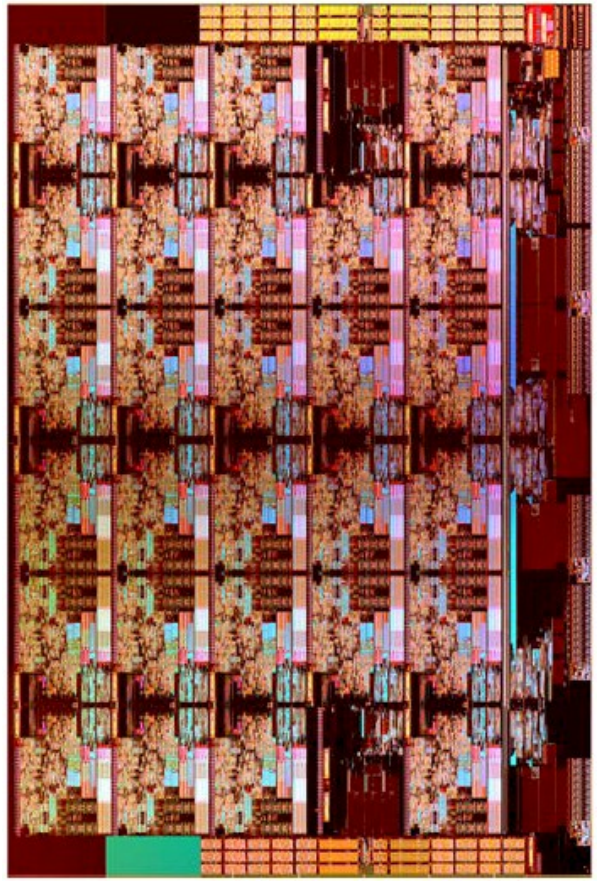
Cache mode is a good option

May sacrifice a few percent for some applications

Intel Xeon Processor Scalable Family – Skylake Server (SKX)

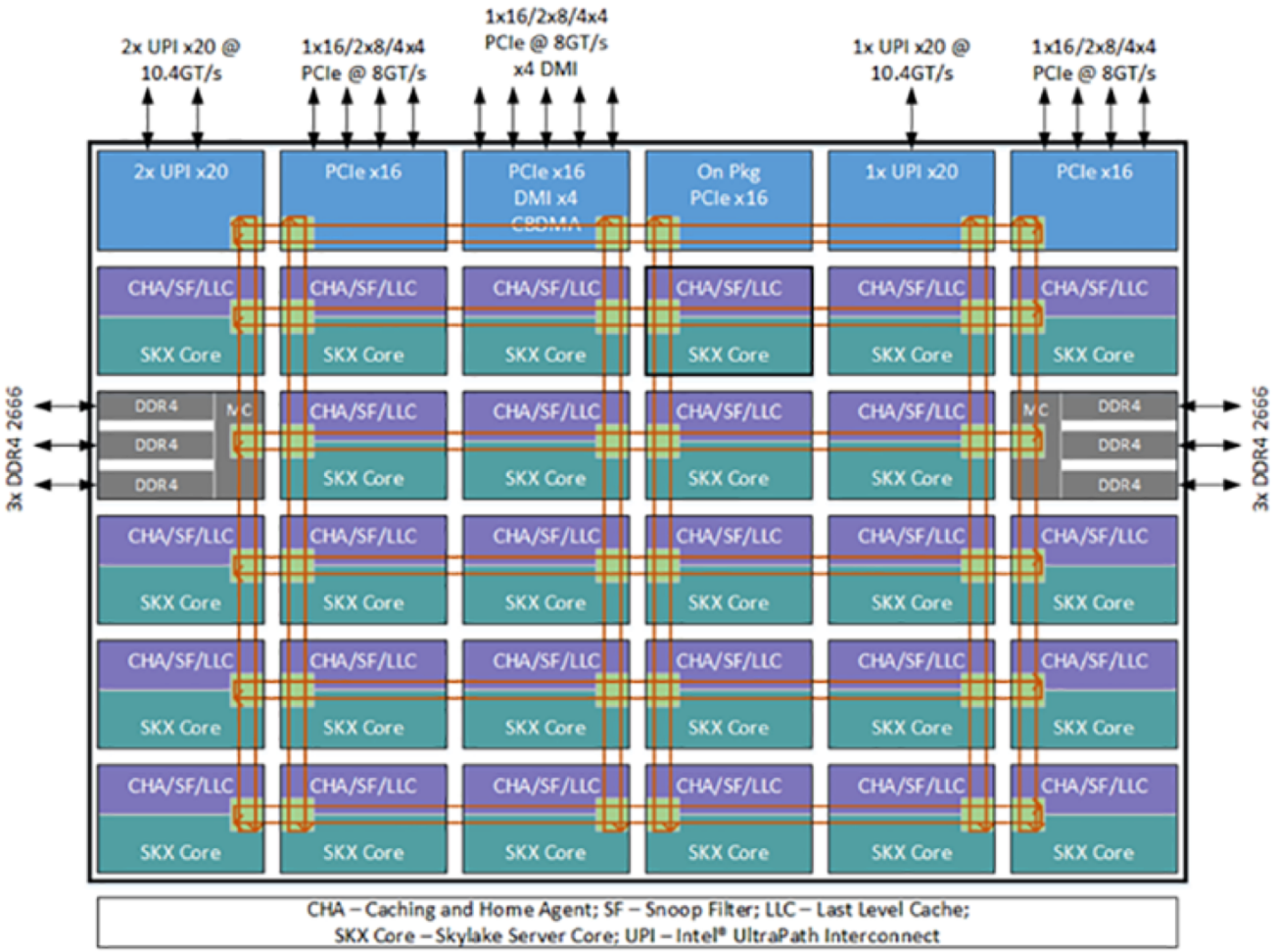
- Released mid 2017
- Major architecture upgrade from Haswell/Broadwell family
- Increased floating point performance
  - 2 512-bit FMA VPUs
- Increased memory bandwidth performance
  - 6 DDR4 2666 GHz channels
- Increased flexibility for legacy codes
  - AVX2, AVX, and SSE instructions can run at higher frequencies than AVX512 instructions
- Scalable from 2 – 8 sockets
- PCI Express 3.0 – 48 lanes

SKX Architecture



- Up to 28 cores
  - 2 hyper threads per core
  - 24 cores per socket on Stampede2
- 32 KiB L1D cache per core
- 1 MiB L2 cache per core
- Up to 38.5 MiB shared L3 cache
  - 1.375 MiB per core
- 2D mesh interconnect
- Up to 768 GiB DDR4 per socket
  - 96 GiB/socket (192 GiB/node) on Stampede2
- Up to 8 sockets per node
  - Dual socket nodes on Stampede2
- Up to 2240 Gflops per socket
- Up to 128 GiB/s DDR bandwidth per socket

SKX Diagram

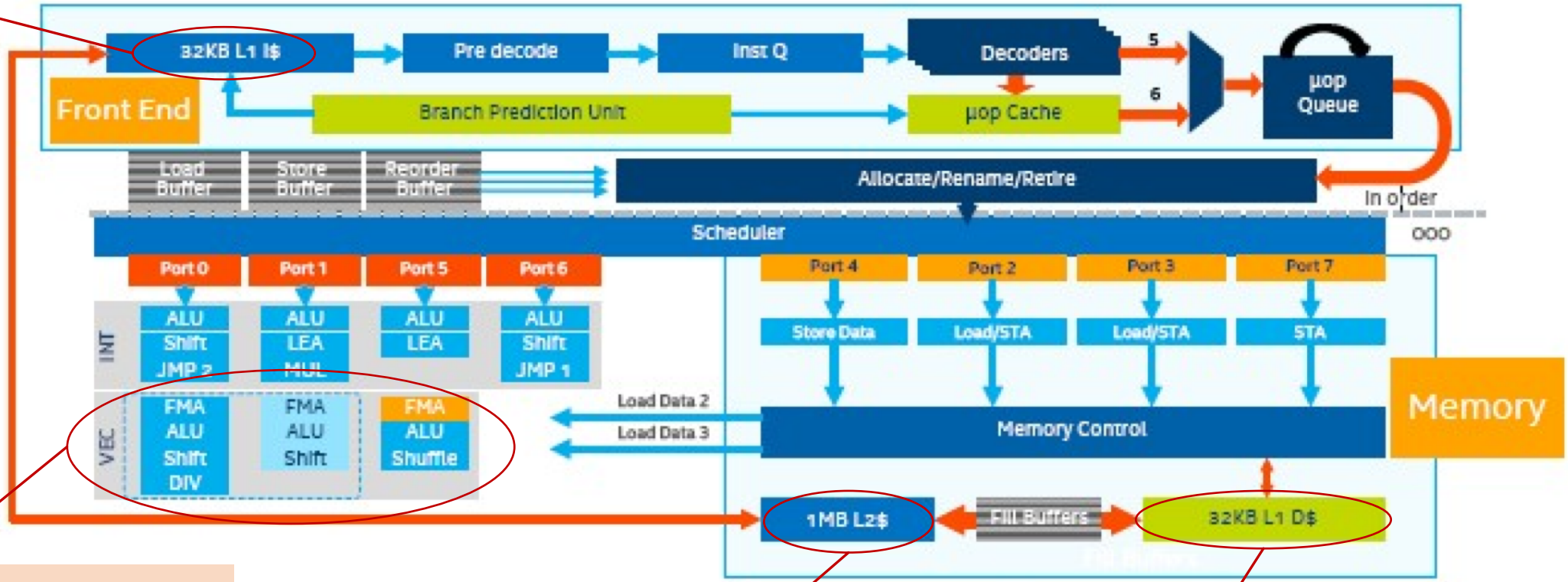


- Up to 28 cores
  - Similar to a 1 core tile
- 2D mesh interconnect
- 6 channels DDR4
  - Up to 128 GiB/s
- 3 16x PCI channels

https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview

SKX Core

8-way 32KiB instruction cache



2 VPUs

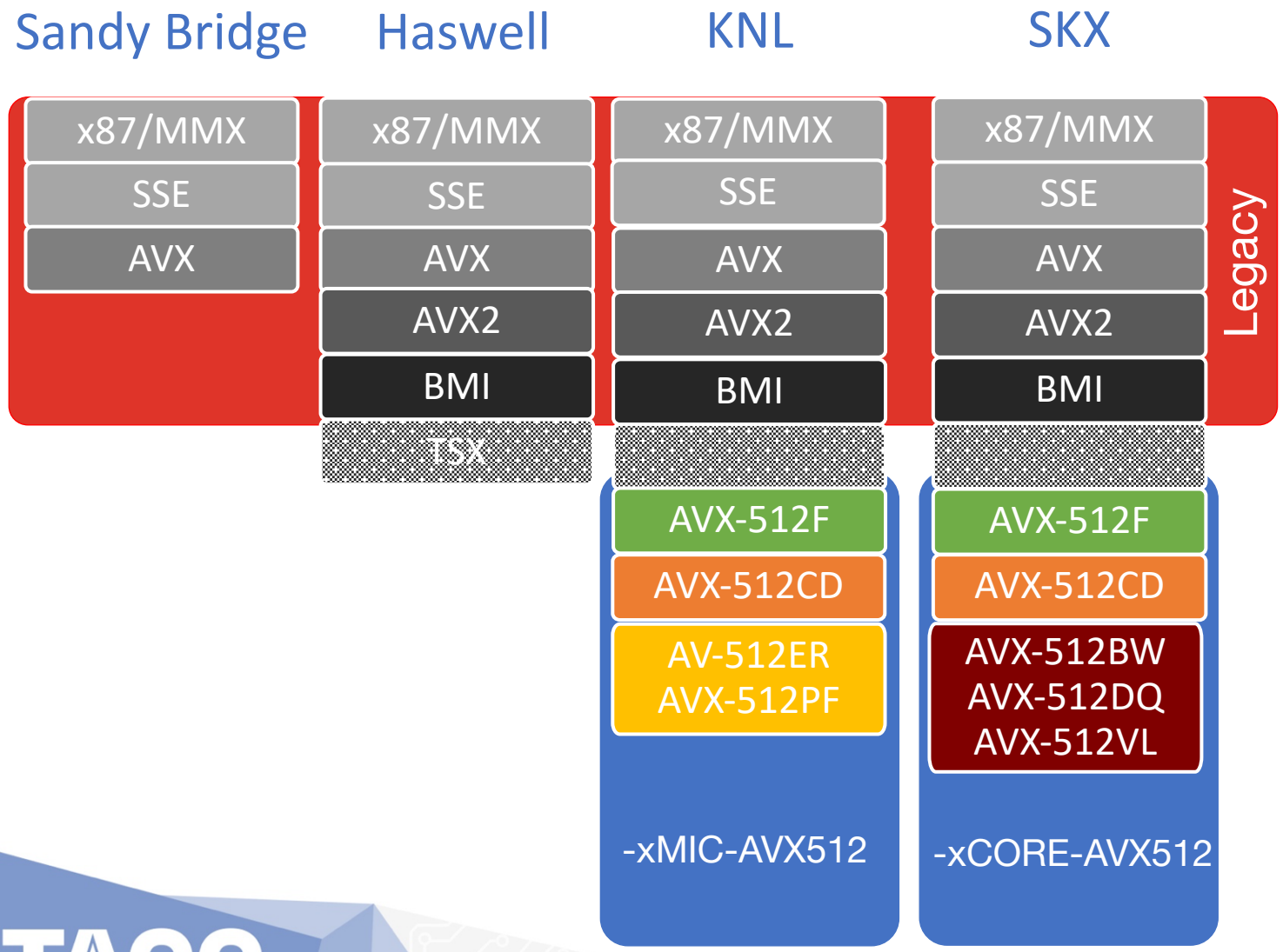
- Compile with **-xCORE-AVX512** to use 512-bit vector lengths
- Both VPUs support legacy code

1 MiB L2 cache

8-way 32KiB data cache

(*KNIGHTS LANDING: SECOND- GENERATION INTEL XEON PHI PRODUCT*, A. Sodani, et.al.,IEEE Micro March/April 2016)

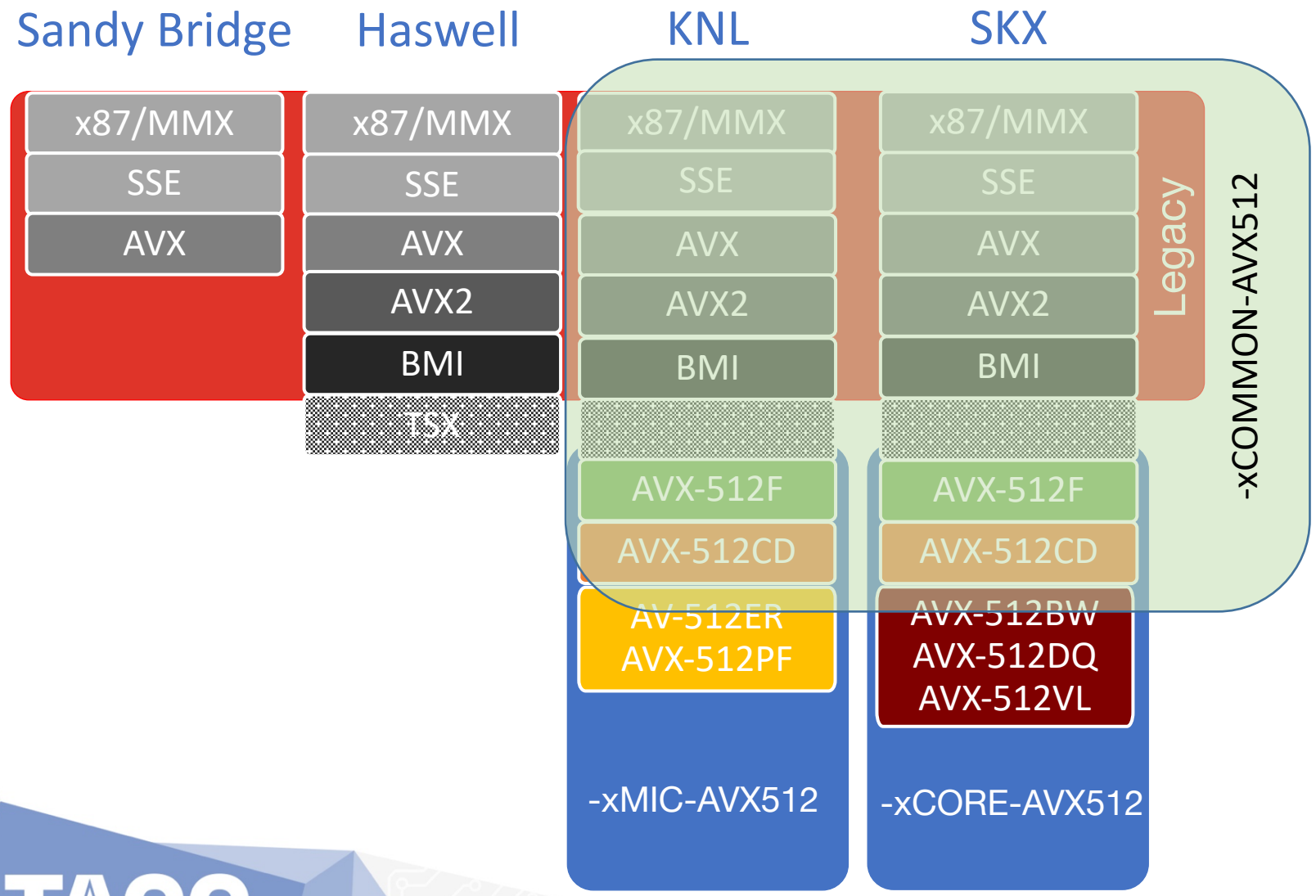
KNL and SKX Instruction Set Architecture



- KNL and SKX support all legacy instructions
- Introduces AVX-512 Extensions:
  - Foundations (common between Xeon and Xeon Phi)
  - Conflict Detection
  - Prefetch, Exponential and Reciprocal
  - 8- & 16- bit integer, Double word & Quad word, Vector Length extensions(use 128- & 256- bit registers)

AVX512 ISA: http://goo.gl/TGQIKE

KNL and SKX Instruction Set Architecture



- KNL and SKX support all legacy instructions
- Common-AVX512 supports both SKX and KNL:
  - Foundations (common between Xeon and Xeon Phi)
  - Conflict Detection
- But doesn't support extensions specific to KNL or SKX

AVX512 ISA: http://goo.gl/TGQIKE

How to compile?

KNL Only:

- `-O3 –xMIC-AVX512`

SKX Only:

- `-O3 –xCORE-AVX512`

KNL and SKX ”skinny” binary:

- `-O3 –xCOMMON-AVX512`
- Leaves out architecture optimizations

KNL and SKX “fat” binary:

- `-O3 –xCORE-AVX2 –axCORE-AVX512,MIC-AVX512`
- Creating a “fat” binary requires a base instruction set shared by both architectures

SKX: COMMON-AVX512 vs CORE-AVX512

For some codes COMMON-AVX512 is faster than CORE-AVX512

- CORE-AVX512 is tuned for “low” 512-bit register (ZMM) use
  - Works well with codes that can’t support 512-bit vector lengths
- COMMON-AVX512 is tuned for “high” 512-bit register (ZMM) use
  - Probably to maintain support for KNL also – only one VPU supports “legacy” instructions

If you believe your code supports 512-bit vectors, try using this option:

- `-O3 –xCORE-AVX512 -qopt-zmm-usage=high`
- Supported by the 18.0 and 17.0.5 Intel compilers

Your mileage may vary.

SKX: Frequency scaling

Skylake *turbo* frequencies are controlled by two parameters:

- Total # of active cores per chip
  - 1-24 on Stampede2 (Intel Xeon Platinum 8160)
- Instruction set
  - SSE, AVX2, AVX512
- Frequency can range from 1.4 GHz to 3.7 GHz

Possible for a code compiled with SSE or AVX2 to be faster than AVX512

SKX: Frequency scaling

Skylake *turbo* frequency range for different instruction sets on Intel Xeon Platinum 8160

	Base Freq. (GHz)	Turbo Frequencies (GHz)		
		1 core	2 – 23 cores	24 cores
SSE	2.1	3.7	3.7 – 2.8	2.8
AVX2	1.8	3.6	3.6 – 2.5	2.5
AVX512	1.4	3.5	3.5 – 2.0	2.0

- AVX512 instructions could drop your turbo frequency by 29% over SSE
  - By 20% over AVX2
- Assume the processor is running in turbo mode constantly
- If you suspect your code doesn't vectorize well, try AVX2 and SSE

How to utilize all the cores?

48 cores, up to 96 ‘Hyper threads’

OR

68 cores, up to 272 ‘Hyper threads’

Some rules of thumb

- At most one process per core
- Use threads for the ‘Hyper threads’
- Divide tasks along cores
  - KNL
    1, 2, 4, 17, 34 , 68 MPI tasks: use all cores
    8, 16 MPI tasks: use 64 cores (very small impact from 4 idle cores)
  - SKX
    1, 2, 4, 8, 12, 24, 48 MPI tasks: use all cores
- Hybrid codes (MPI + OpenMP) have an advantage

Hybrid Computing on KNL

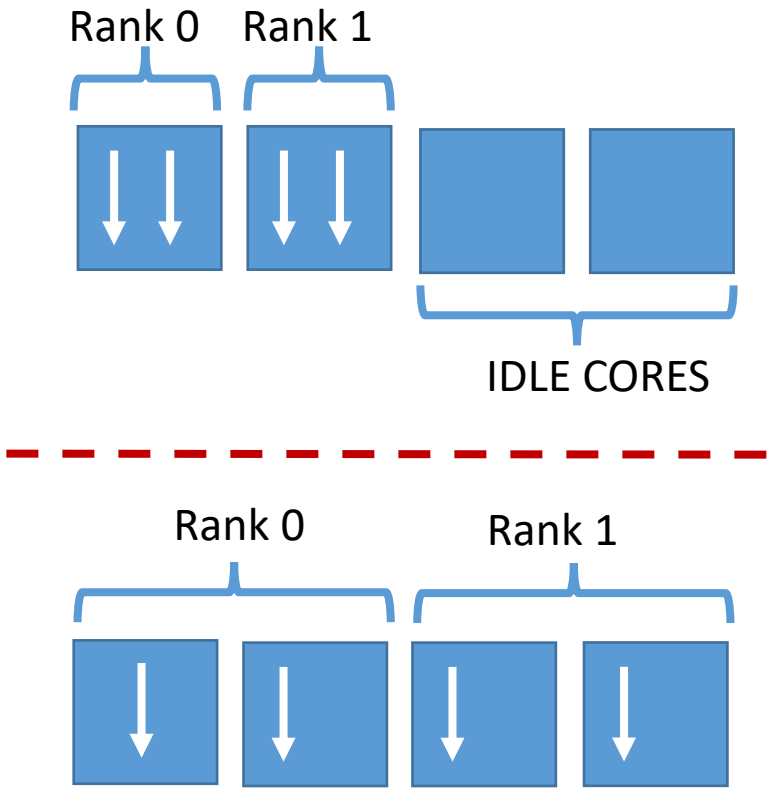
Hybrid Computing in a Slide

- MPI processes act as containers for threads
- Each MPI process is assigned a range of processors
- Threads spawned by an MPI process can only run within the assigned range.
- Two main issues:
  - Process binding (Where does my process run?)
  - Process affinity (Where does my process access memory?)

Why should I Care About Process Affinity?

- Process binding and affinity are critical in hybrid codes because of hardware complexity
- Often hybrid codes use less MPI tasks than cores
- If MPI binding is "compact" some hardware is not utilized!
- Multiple threads could be running on the same logical processor, overloading the hardware and serializing execution

Example: 2 MPI tasks, OMP_NUM_THREADS=2
Assume: 4 core CPU, 2 logical processors / core



Welcome to some Context Switching

- On the other hand, without any binding…
- Threads can switch from a processor to another…
- Thrashing the cache
- And reducing performance due to the first touch policy
  - First touch: the first time you "touch" an array (allocation) determines the location in memory it resides
  - That location will be LOCAL if your thread does not float around
- So yes, you need to set up binding

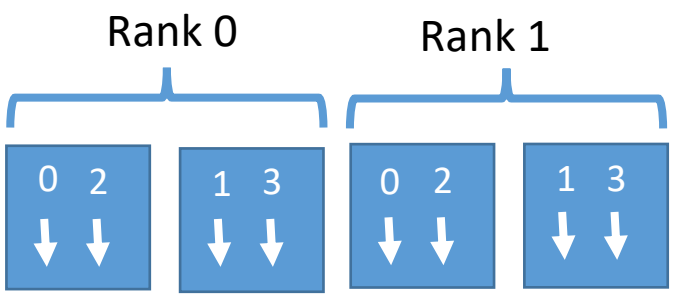
Defaults with Intel MPI

- Sensible
- MPI tasks will be given a range of processors equal to the `OMP_NUM_THREADS` value
- Threads will then be assigned in a "scatter" sequence within the allowed processor range for each MPI task

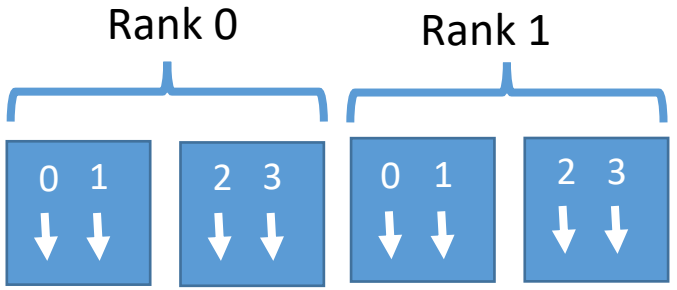
Going back to our Small Example…

Assume 2 MPI tasks and OMP_NUM_THREADS=4

Default



Likely Optimal



Setting Up Thread Affinity

- Multiple ways of doing this from the environment
  - Intel KMP_* environmental variables
  - OMP environmental variables
- I will describe the OMP way because it is more portable
- OMP has two ways of setting process affinity:
  - Define a binding policy (simpler, adequate for most cases)
  - Define specific binding places (more involved, finer granularity)

Binding Policies

- OpenMP 4.5 and above
- Distribution Policy (set in `OMP_PROC_BIND` variable)

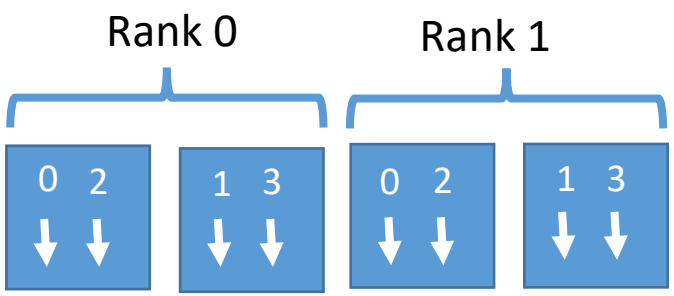
CLOSE - packs threads close together
SPREAD - spreads threads out
MASTER - for nested parallel loops (uses master's affinity)

```
export OMP_PROC_BIND=close
export OMP_PROC_BIND=spread
```

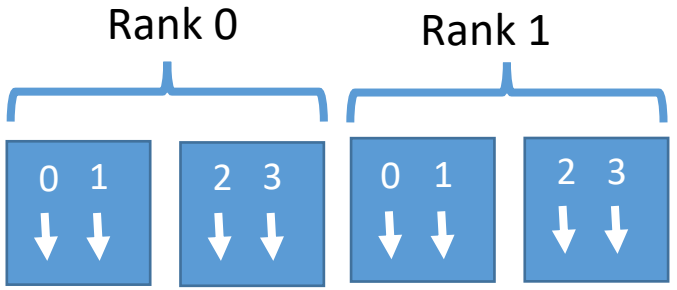
Small Example

Assume 2 MPI tasks and OMP_NUM_THREADS=4

Intel Default

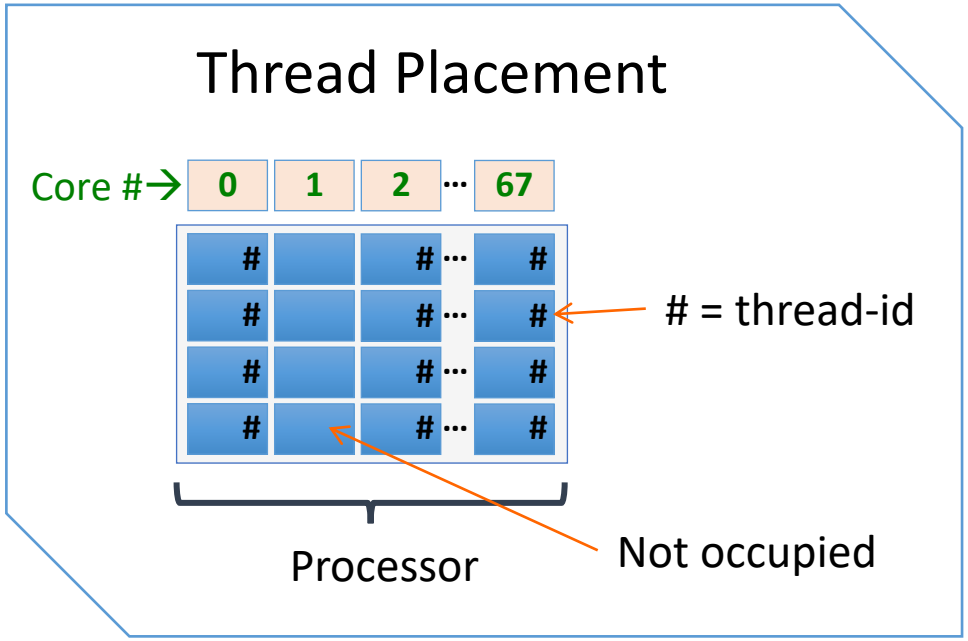


OMP_PROC_BIND=spread



How does that look on a 68 core KNL?

We had to come up with a different way to visualize this…



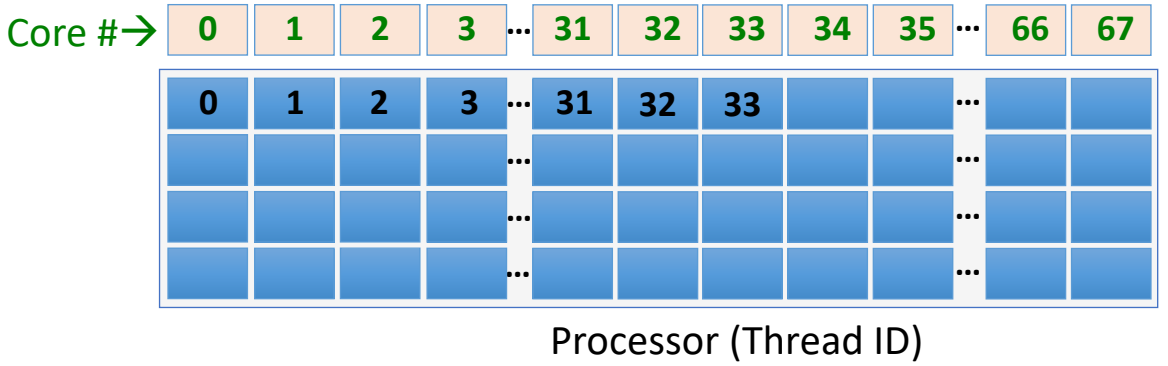
Each Core has 4 *hardware threads*
These are often called *hyperthreads*
or *logical processors*

Your software threads will be assigned to different logical processors depending on how you set up your affinity.

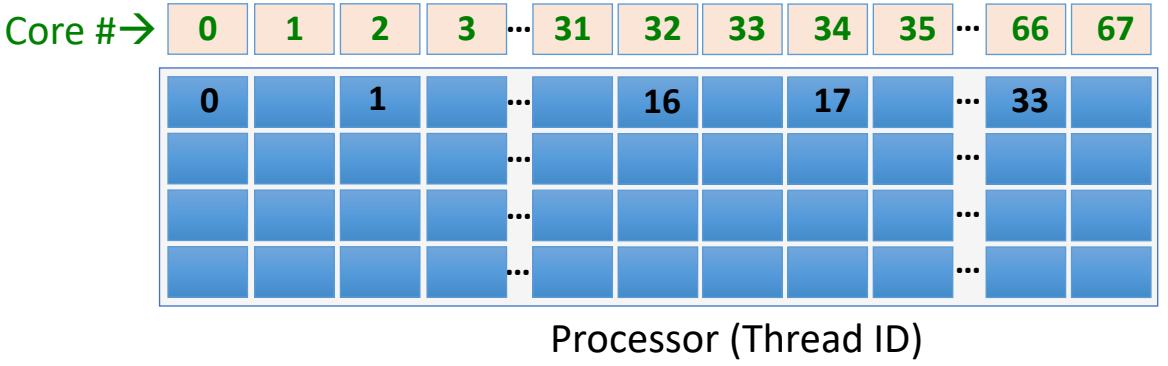
Let's start with a single MPI rank placement

Using Less Than 1 Thread / Core

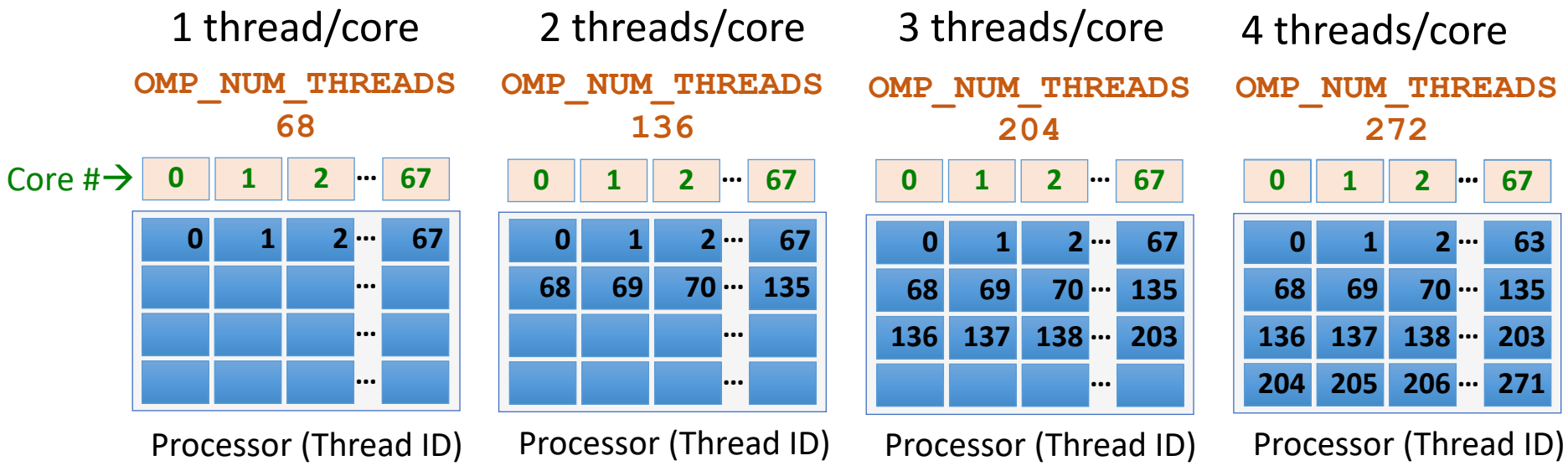
```
OMP_PROC_BIND=close
OMP_NUM_THREADS=34
```




```
OMP_PROC_BIND=spread
OMP_NUM_THREADS=34
```



Defaults for More Than 1 Thread / Core



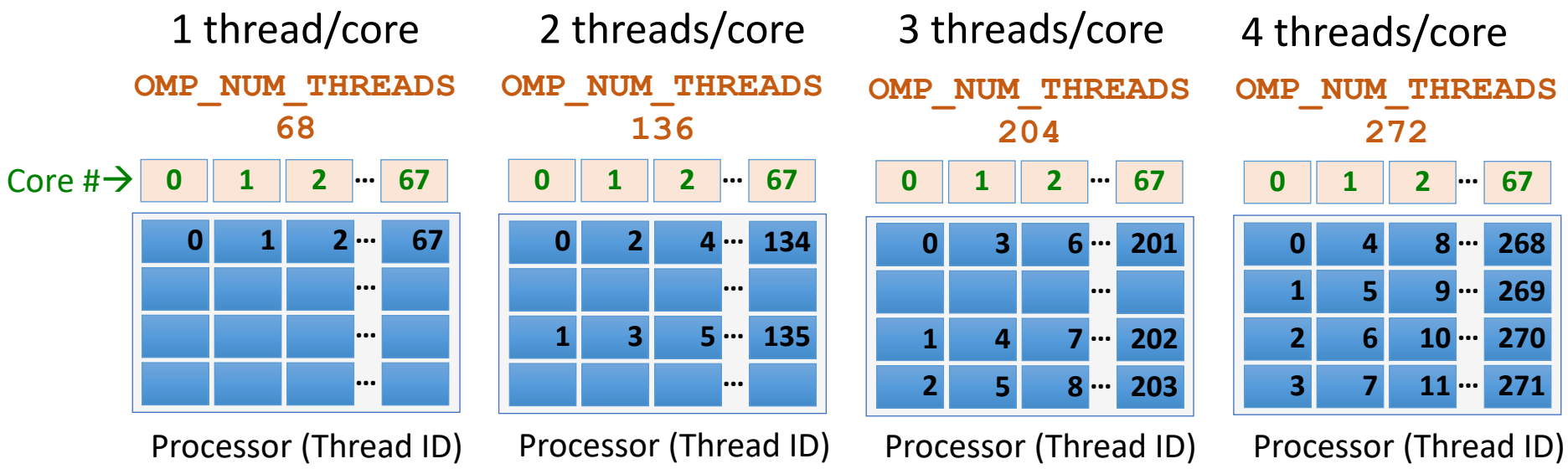
This is OK for some applications, but:

1.) Putting sequential threads on the same core may be more cache friendly,

2.) Allowing core threads to "float" in the core may be more balance friendly.

Spread For More Than 1 Thread / Core

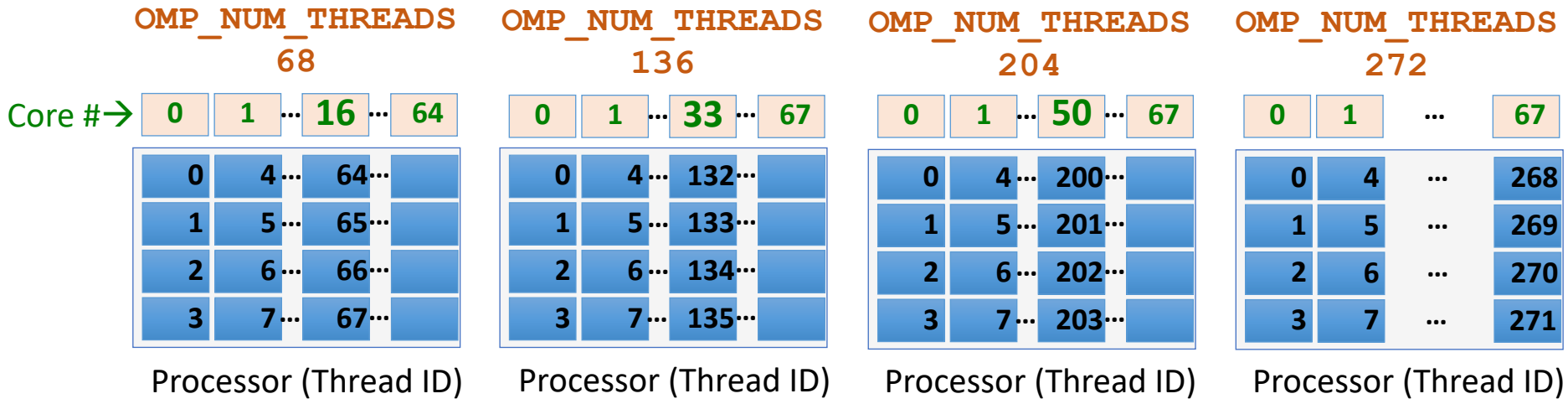
```
export OMP_PROC_BIND=spread
```



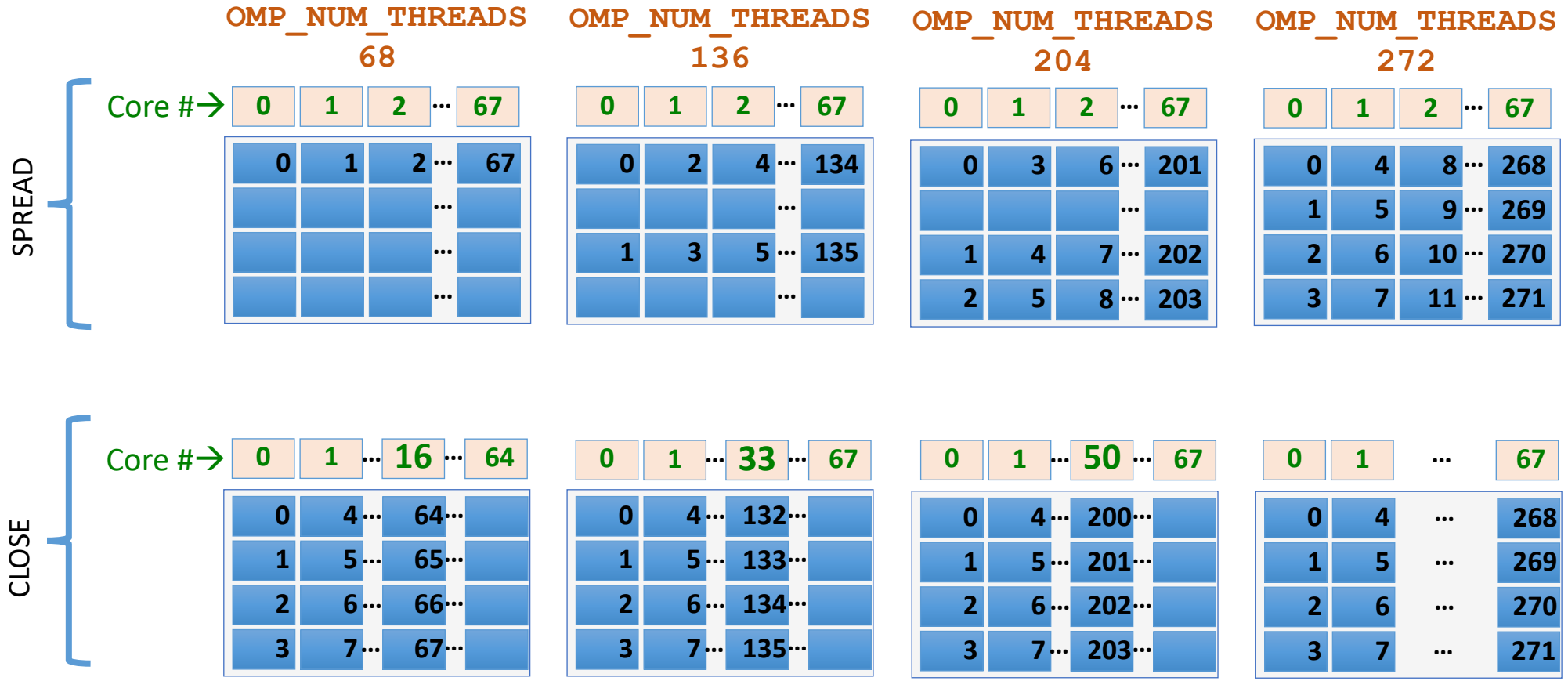
Sequential threads are on the same core.
Try this affinity setting when increasing the threads per core.

Close For More Than 1 Thread / Core

```
export OMP_PROC_BIND=close
```



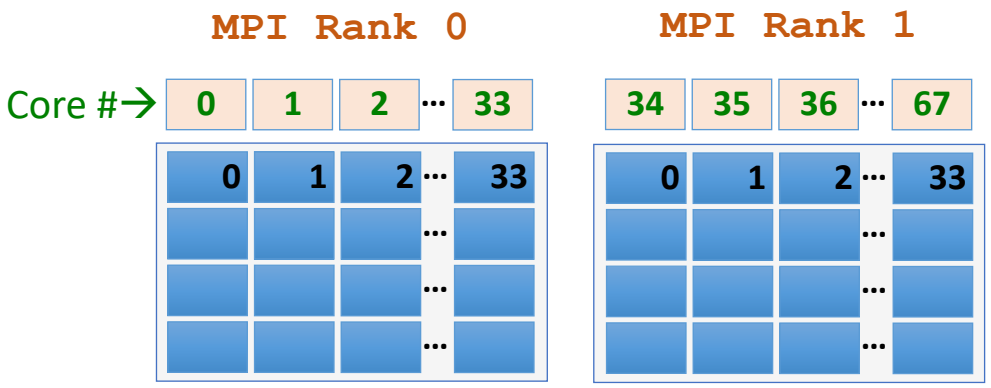
Spread vs Close



Using 2 MPI Tasks

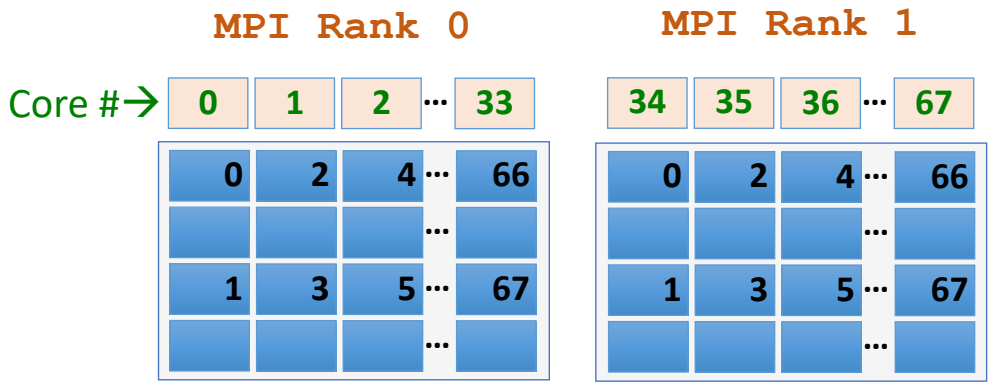
```
export OMP_PROC_BIND=spread
export OMP_NUM_THREADS=34
```

Uses all cores, 1 thread / core
Total number of threads = 2 x 34 = 68




```
export OMP_PROC_BIND=spread
export OMP_NUM_THREADS=68
```

Uses all cores, 2 threads / core
Total number of threads = 2 x 68 = 136



Setting Up Memory Policy

- In most cases you can get away with the already mentioned command `numactl --membind=1`
- If using SCN2 or SNC4, however, you need more
- Each MPI task must be launched using the correct memory policy, and it is not sufficient to wrap your MPI launcher in a numactl command.

Use tacc_affinity

`tacc_affinity`

- Assumes Intel MPI will not spread task bindings across numanodes
- Determines which core is being used
- Determines HBW memory numanodes available
- Determines which numanode process is bound to
- Sets memory affinity to the corresponding HBW numanode

Caveats of tacc_affinity

- Only performs memory affinity operations
- Assumes binding is handled by the MPI implementation
- Use “task_affinity” if you are running multiple MPI executions within one batch job:

  ibrun –o 0 –N 32 ./myapp1.exe &

  ibrun –o 32 –N 32 ./myapp2.exe &

  wait&
- task_affinity implements process binding as well as memory affinity
  - Designed for multiple MPI executions within a batch job and SNC2 and SNC4 KNL configurations

Checking MPI Process Binding

- Use the following to get Intel MPI to report process bindings:

```
export I_MPI_DEBUG=4
```

- As part of your standard output you will see something like:

```
[0] MPI startup(): Rank    Pid      Node name       Pin cpu
[0] MPI startup(): 0       105228   test1          {0,1,2,3,4,5, … 31, … }
[0] MPI startup(): 1       105229   test1          {32,33,34,35, … 63, … }
```