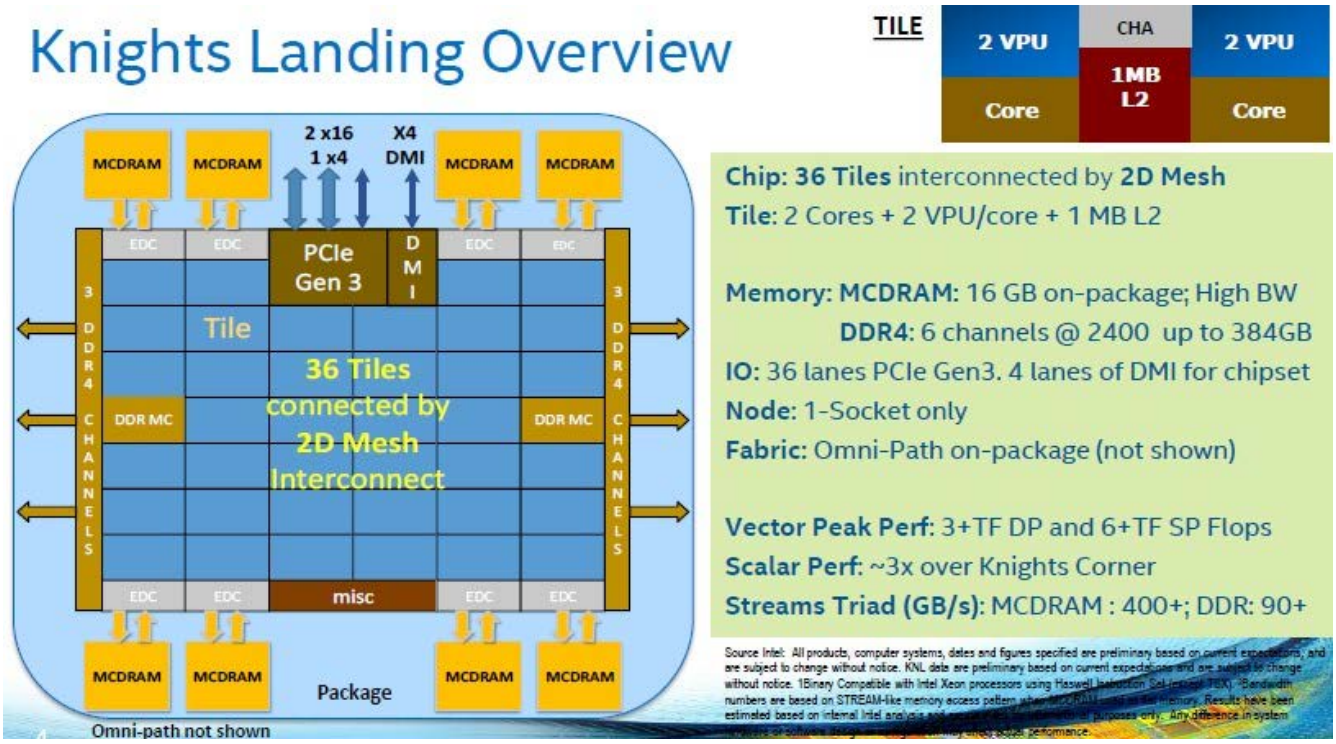
Xeon Phi Knights Landing (KNL)

Technical details

The Knights Landing chip is etched in 14 nanometer manufacturing processes with over 8 billion transistors, it is the largest chip that Intel has ever made.



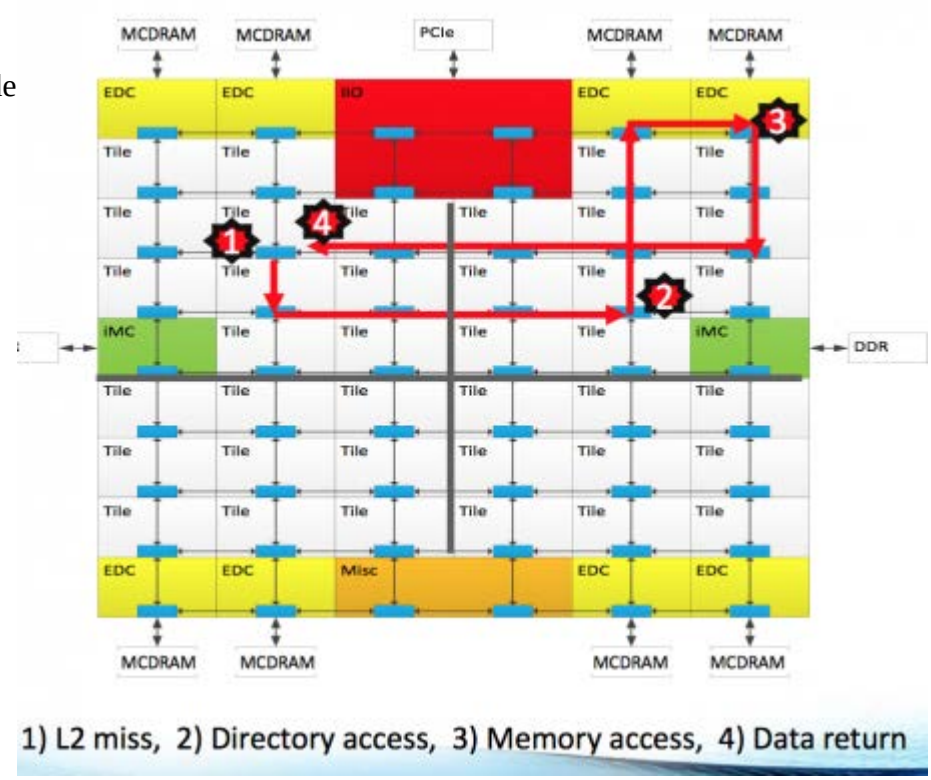
The cores on the Knights Landing chip are based on a heavily modified “Silvermont” Atom core that has four threads. These are tiled in pairs, with each core having two AVX512 512-bit vector processing units and 1 MB of L2 memory shared across the tile. The tiles are linked to each other using a 2D mesh interconnect, which also hooks into the two DDR4 memory controllers that feed into so-called far memory, which scales up to 384 GB capacity and which delivers around 90 GB/sec of bandwidth on the STREAM Triad memory benchmark test. That 2D mesh also hooks the cores into eight chunks of high bandwidth stacked MCDRAM memory, which scales up to 16 GB of capacity; this is known as the “near memory” in the Knights Landing chip and offers more than 400 GB/sec of bandwidth to keep those cores well fed. Intel has a number of different modes of memory addressing in the Knights Landing processor, including using the combined memory as a single address space or using the MCDRAM as a L3 cache for the DRAM memory.

NUMA on KNL

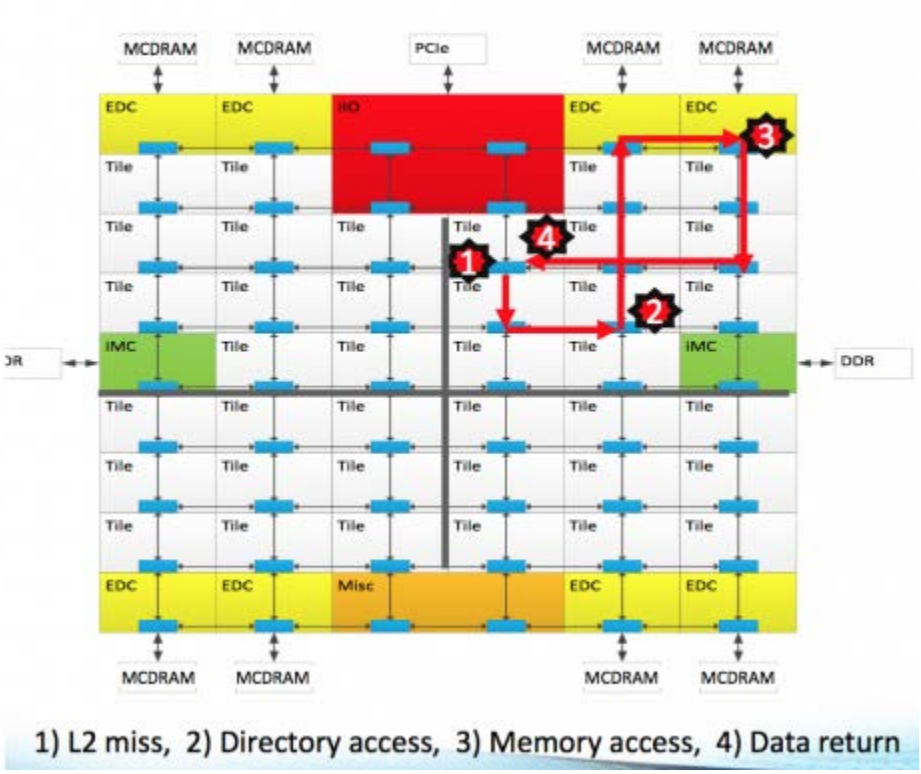
NUMA stands for Non-Uniform Memory Access. It represents the situation where certain cores on a node can be considered "closer" to some part of the memory space than others or if some memory on the node has different latencies or bandwidths to the cores. Of these situations occur on the KNL nodes - there are two different types of memory available with different specs, and different channels of memory are closer to different cores in the 2d mesh.

NUMA Mode Options on KNL

In quadrant mode the chip is divided into four virtual quadrants, but is exposed to the OS as a single NUMA domain. The diagram below shows illustrates the affinity of tag directory to the memory. In many cases, this mode is the easiest to use and will provide good performance.



In sub-NUMA mode each quadrant (or half) of the chip is exposed as a separate NUMA domain to the OS. In SNC4 (SNC2) mode the chip is analogous to a 4 (2) socket Xeon. This mode has potential for high performance, but software must be optimized for NUMA architectures to benefit. On Cori SNC4 can be complicated to fully utilize as there are non-homogenous number of cores per quadrant on a 68 core CPU (Because the 34 tiles cannot be evenly distributed among 4 quadrants).



MCDRAM Memory Options on KNL

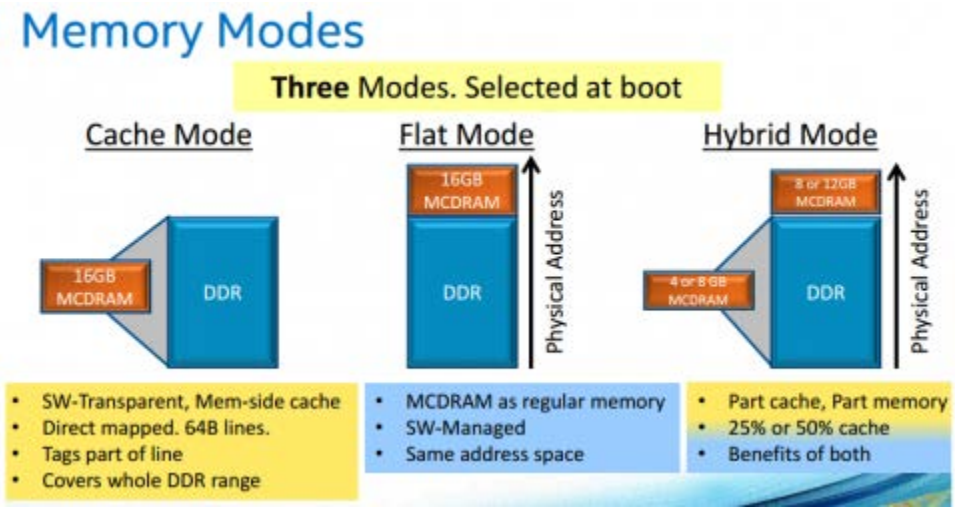
There is no shared L3 cache on the KNL processor. However, the 16 GB of MCDRAM (spread over 8 channels) can be configured either as a direct-mapped cache or as addressable memory. When configured as a cache, recently accessed data is automatically stored in cache, similarly to an L3 cache on a Xeon processor. However, there are some notable differences:

- The cache (16GB) is significantly larger than a typical L3 cache on a Xeon processor (usually in the tens of MB).

- The cache is direct-mapped. Meaning it is non-associative - each cache-line worth of data in DRAM has one location it can be cached in MCDRAM. This can lead to possible conflicts for apps with greater than 16GB working sets.

- Data is not prefetched into the MCDRAM cache

The MCDRAM may be configured either as a cache, addressable memory or as a mix of the two. This is shown in the figure below:



When the MCDRAM is configured at least in part as addressable memory, it is presented to the operating system and the user as an additional NUMA domain (quadrant mode) or multiple additional NUMA domains (SNC2 or SNC4 mode) as described above.

Code modifications for the Intel Xeon Phi

While the Phi should be able to run many applications unmodified, it is expected that most applications will require code changes to achieve good performance.

Optimization Areas

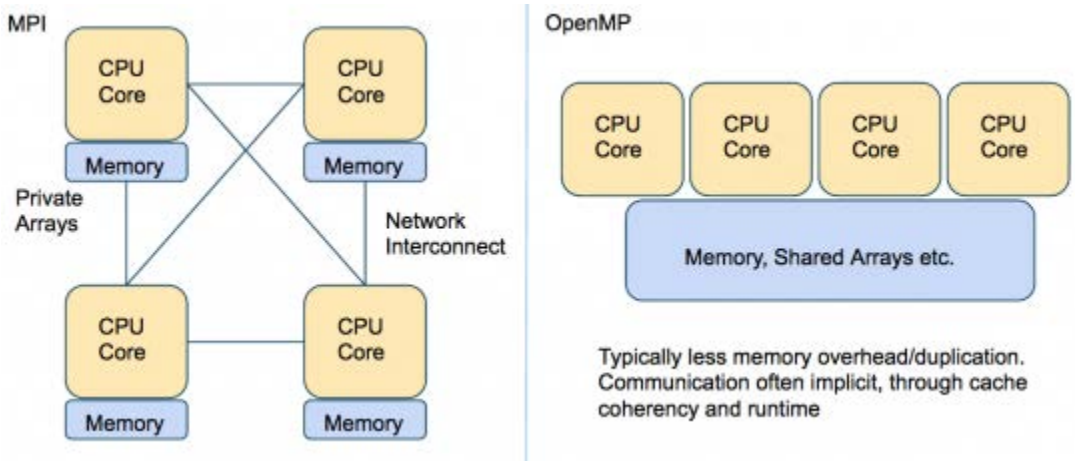
1. **Identifying and adding more node-level parallelism** and exposing it in your application. An MPI+X programming approach is encouraged where MPI represents a layer of internode communication and X represents a conscious intra-node parallelization layer where X could again stand for MPI or for OpenMP, pthreads, PGAS etc...
2. **Evaluating and improving your Vector Processing Unit (VPU) utilization and efficiency**. The KNL processors have an 8 double-precision wide vector unit. Meaning, if your code produces scalar, rather than vector instructions, you miss on a potential 8x speedup.
3. **Evaluating and optimizing for your memory bandwidth and latency requirements**. Many codes run are performance limited not by the CPU clock speed or vector width but by waiting for memory accesses. As described in detail below, a lot can be done to optimize such code including memory locality improvements and use of the on-package high-bandwidth memory (HBM).

Optimization Concepts

On-Node Parallelism

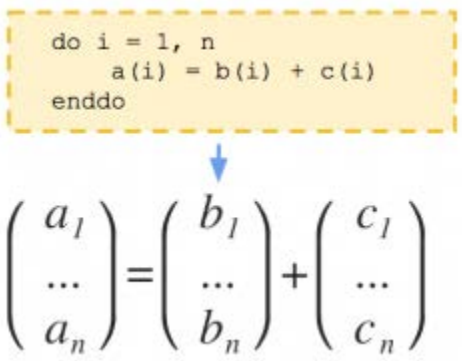
It will be necessary to specifically think both about inter-node parallelism as well as on-node parallelism. The baseline programming model for KNL is MPI+X where X represents some conscious level of on-node parallelism, which could also be expressed as MPI or a shared memory programming model like OpenMP, pthreads etc. For a lot of code, running without changes on 72 (or up to 288) MPI tasks per node could be troublesome. Examples are codes that are MPI latency sensitive like 3D FFTs and codes that duplicate data structures on MPI ranks (often MPI codes don't perfectly distribute every data structure across ranks) which could more quickly exhaust the HBM if running in pure MPI mode.

The difference between the typical pure MPI application targeting inter-node communication and an OpenMP parallel code targeting on-node parallelism is shown in the following figure:



Vectorization

Vectorization is actually another form of on-node parallelism. Modern CPUs have Vector Processing Units (VPUs) that allow the processor to do the same instruction on multiple data (SIMD) per cycle. Consider the figure below. The loop can be written in "vector" notation:



The following figure shows examples of code the compiler won't generally choose to vectorize

Loop dependency:

```
do i = 1, n
    a(i) = a(i-1) + b(i)
enddo
```

Task forking:

```
do i = 1, n
    if (a(i) < x) cycle
    if (a(i) > x) …
enddo
```

In the top case, compilers are unable to give you vector code because the i^{th} iteration of the loop depends on the $i-1^{th}$ iteration. In other words, it is not possible to compute both these iterations at once. The bottom case shows an example where the execution of the loop forks based on an if statement. Depending on the exact situation, the compiler may or may not vectorize the loop. Compilers typically use heuristics to decide if they can execute both paths and achieve a speedup over the sequential code.

More on dependency analysis: https://en.wikipedia.org/wiki/Dependence_analysis

Memory Bandwidth

Processors have a finite amount of Memory Bandwidth - that is: there is maximum number of bytes per second they can read from memory. This limits the performance of many applications.

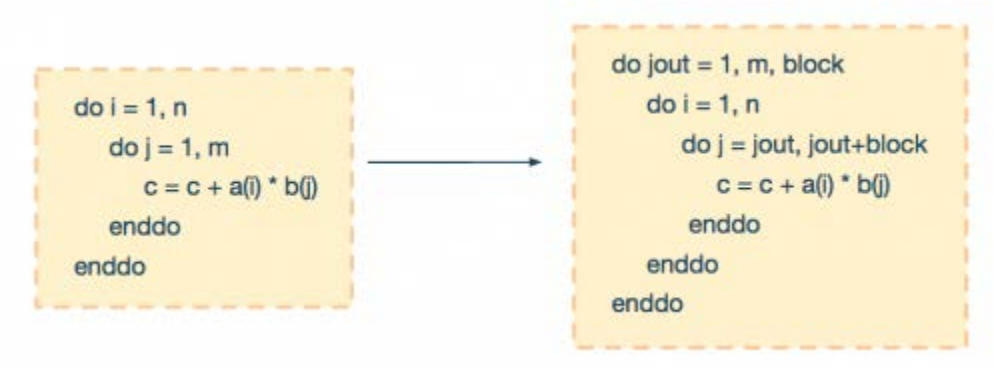
```
do i = 1, n
   do j = 1, m
      c = c + a(i) * b(j)
   enddo
enddo
```

Assuming the arrays a(:) and b(:) are very large - too large to fit in any level of cache on the processor - then, every time this block of code is executed, n*m + n data elements must be streamed into the processor from memory. The n*m factor comes from streaming in the complete array b(:) from memory n times, and the extra factor of n comes from the need to stream array a(:) in from memory once.

Let’s assume for a minute that both a(:) and b(:) are arrays of double precision numbers (i.e. 8 bytes per element). Since there are 2 *n*m flops (one for add and one for multiply) the above loop has an approximate flop to byte ratio of 1/4. The flop to byte ratio as defined above is what we call the "Operational Intensity" of the code.

The code above is "memory bandwidth bound" - i.e. its performance is limited not by the CPU clock speed, or ability to vectorize, but by the rate at which it can bring data from memory into the processor.

Consider now the following transformation from our code block above that represents a functionally equivalent change:



In the code on the right, for each i, we now access chunks of array b(jout:jout+block) at a time. Assuming these chunks fit into the last level of cache on the processor, the amount of data we now need to stream into the CPU from memory during execution is n*m/block + m. The first term comes from the fact that we stream in all of array a(:) m/block times. Thus, the amount of data accessed from memory has gone down significantly and our operational intensity is now 2*block/8 - a big improvement!

All information in this document based on:

http://www.nersc.gov/