Introduction to the Intel® Xeon Phi™ Processor
(previously code-named “Knights Landing”)

John Pennycook, Intel Corporation
May 2018, ALCF Performance Workshop
Agenda

- What is Knights Landing?
  - Architecture
  - MCDRAM & Cluster Modes
- Making the Most of Knights Landing
  - Vectorization with AVX-512
- Summary
What is Knights Landing?
Knights Landing Architecture

Full Xeon ISA compatibility through AVX-512
Over 3 TF DP peak¹

Up to 72 cores
2D mesh architecture

Up to 16GB high-bandwidth on-package memory (MCDRAM)
Exposed as NUMA node
~500 GB/s sustained BW²

2x 512b VPU per core
(Vector Processing Units)

Up to 72 cores

6 channels DDR4
Up to 384GB

Based on Intel® Atom Silvermont processor with many HPC enhancements
Deep out-of-order buffers
Gather/scatter in hardware
Improved branch prediction
4 threads/core
High cache bandwidth
& more

¹ Based on calculated theoretical peak double precision performance capability for a single Intel® Xeon Phi™ Processor 7250: 2 instructions/cycle * 2 floating-point operations (FMA) * 8-wide SIMD * 68 cores * 1.4 GHz = 3.046 TFLOP/s
² Based on STREAM benchmark results for Intel® Xeon Phi™ Processor 7250, 68 cores, 2 MB page alignment and MCDRAM flat mode; see Slide 31.

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. For more complete information visit www.intel.com/benchmarks.

Benchmark results were obtained prior to implementation of recent software patches and firmware updates intended to address exploits referred to as “Spectre” and “Meltdown”. Implementation of these updates may make these results inapplicable to your device or system.
Memory Modes

Cache

- Direct-mapped cache.
- Misses are expensive (higher latency) because they access MCDRAM and DDR.
- No source changes required.

Flat

- Physical address space.
- Exposed as NUMA node(s).
  - `numactl -H, lscpu` to display configuration.
- Accessed through libraries/numactl.

Hybrid mode combines the above in 50% / 50% and 25% / 75% configurations.
Memory Modes: Which to Choose?

Easiest ways to use MCDRAM

<table>
<thead>
<tr>
<th>DDR Only</th>
<th>MCDRAM as Cache</th>
<th>MCDRAM Only</th>
<th>Flat DDR + MCDRAM</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Effort</td>
<td>No software changes required.</td>
<td>Change allocations for bandwidth-critical data.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Latency and bandwidth impact
- Limited memory capacity
- Most flexible configuration + opportunity for new algorithms
Cluster Modes: All-to-All (All2All)

Address Hash:
Uniform across all directories.

Affinity:
No affinity between tile, directory and memory.

Performance:
Lower than other modes.

Software Changes:
None.

Usage Scenario:
Fallback mode when DDR is unevenly populated.

1. L2 miss; 2. Directory access; 3. Memory access; 4. Data return
Cluster Modes: Quadrant (Quad)

Address Hash:
By quadrant.

Affinity:
Directory and memory in same quadrant.

Performance:
Lower latency and higher bandwidth than All2All.

Software Changes:
None.

Usage Scenario:
Default.

1. L2 miss; 2. Directory access; 3. Memory access; 4. Data return
Cluster Modes: Sub-NUMA Clustering (SNC)-4

Address Hash:
By quadrant.

Affinity:
Tile, directory and memory in same quadrant.

Performance:
Lowest latency of all modes.

Software Changes:
Application/environment must be NUMA-aware.

Usage Scenario:
NUMA-optimized codes or multiple MPI ranks.

1. L2 miss; 2. Directory access; 3. Memory access; 4. Data return
## Cluster Modes: Which to Choose?

<table>
<thead>
<tr>
<th></th>
<th>All2All</th>
<th>Quadrant</th>
<th>SNC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Software Effort</strong></td>
<td>No software changes required.</td>
<td>Changes for NUMA.</td>
<td></td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>Worst</td>
<td>Good</td>
<td>Good++</td>
</tr>
</tbody>
</table>

- **Easiest to use.**
- **Small performance difference.**

**No software changes required.**

**Changes for NUMA.**
Making the Most of Knights Landing
A New ISA: AVX-512

- Knights Landing is the first micro-architecture to support AVX-512.
  - AVX-512F, AVX-512 CDI, AVX-512 ERI, AVX-512 PFI

- Differences from AVX2:
  - 32 x 512 bit SIMD registers (zmm0 – zmm31)
  - Dedicated mask registers (k0 – k7)
  - New instructions: gather/scatter (F), expand/compress (F), conflict detection (CDI), exponential and reciprocal (ERI), prefetch (PFI)

- Differences from IMCI (Knights Corner ISA):
  - Backwards compatible with SSE/AVX.
  - New instructions: conflict detection (CDI)
Why Masks Matter

- Even code that looks simple to humans is not simple to the compiler.

```c
#pragma omp simd
for (int i = 0; i < N; ++i)
{
    if (input[i] != 0)
    {
        output[i] = x / input[i];
    }
}
```

Compiler: This memory reference may be invalid when input[i] == 0.

Human: The branch is intended to prevent division by zero.

- With AVX-512, running all instructions in the branch under a mask will prevent both arithmetic exceptions and page faults.
Compiling for AVX-512

```c
int main(int argc, char* argv[]) {
    // example comment
    first_function();
    second_function();
    return 0;
}
```

**Compiler Libraries**
- `-xCORE-AVX512`
- `-xCOMMON-AVX512` (or `-xSSE, -xAVX...
- `-xMIC-AVX512`

**Parallel Models**
- Intel® Xeon® Processors
- Intel® Xeon® Processors
- Intel® Xeon Phi™ Processor
- Intel® Xeon Phi™ Processor

1 Processors previously codenamed “Skylake” supporting AVX-512 instructions.
2 Processors previously codenamed “Knights Landing”.
3 Binaries with TSX instructions are unsupported on processors previously codenamed “Knights Landing”.

© 2018 Intel Corporation
AVX-512 Exponential & Reciprocal (ERI)

- Three new instructions:
  - **VEXP2**: approximate $2^x$
  - **VRCP28**: approximate $1/x$
  - **VRSQRT28**: approximate $1/\sqrt{x}$

- Compiler can be encouraged to use approximations via code transforms:

  ```
  float x = y / z;
  vs
  float x = y * (1.0f / z);
  ```

- Floating-point precision can be further tuned via compiler flags:

  - `fimf-absolute-error`
  - `fimf-accuracy-bits`
  - `fimf-arch-consistency`
  - `fimf-max-error`
  - `fimf-precision`
  - `fimf-domain-exclusion`
  - `[no-]prec-div`
  - `[no-]prec-sqrt`

**Example:**
- `fimf-precision=low`
- `fimf-domain-exclusion=15`

https://software.intel.com/en-us/node/522977
Enable with \texttt{--qopt-report=[1-5]; direct with \texttt{--qopt-report-phase=[all, cg, ipo, vec, ...]}

LOOP BEGIN at example.cpp(5,5)

remark #15388: vectorization support: reference \texttt{a[i]} has aligned access [ example.cpp(8,9) ]
remark #15389: vectorization support: reference \texttt{b[i]} has unaligned access [ example.cpp(7,17) ]
remark #15381: vectorization support: unaligned access used inside loop body
remark #15412: vectorization support: streaming store was generated for \texttt{a[i]} [ example.cpp(8,9) ]
remark #15415: vectorization support: irregularly indexed load was generated for the variable \\
\texttt{<c[*((b+i*4)]>}, part of index is read from memory [ example.cpp(8,22) ]
remark #15305: vectorization support: vector length 16
remark #15309: vectorization support: normalized vectorization overhead 0.464
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
remark #15442: entire loop may be executed in remainder
remark #15449: unmasked aligned unit stride stores: 1
remark #15450: unmasked unaligned unit stride loads: 1
remark #15462: unmasked indexed (or gather) loads: 1
remark #15467: unmasked aligned streaming stores: 1
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 12
remark #15477: vector cost: 1.750
remark #15478: estimated potential speedup: 5.990
remark #15487: type converts: 2
remark #15488: --- end vector cost summary ---

LOOP END

SIMD Patterns: Common Pitfalls

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Error checking within a vectorizable loop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue</td>
<td>Number of loop iterations unknown at compile time.</td>
</tr>
<tr>
<td>Original</td>
<td>if (error condition) { exit }</td>
</tr>
<tr>
<td>Transform</td>
<td>if (error condition) { error = true; } ... if (error) exit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Memory access guarded by conditional.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue</td>
<td>Compiler cannot assume memory is safe to access.</td>
</tr>
<tr>
<td>Original</td>
<td>if (condition) a[i] += result;</td>
</tr>
<tr>
<td>Transform</td>
<td>a[i] += (condition) ? result : 0;</td>
</tr>
</tbody>
</table>
## SIMD Patterns: Common Pitfalls

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Loading neighbour values, with a branch for boundary conditions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue(s)</td>
<td>Compiler may generate a gather; array[position-1] may be unsafe.</td>
</tr>
<tr>
<td>Original</td>
<td>double left = (column &gt; 0) ? array[position -1] : boundary</td>
</tr>
<tr>
<td>Transform</td>
<td>Pad array with appropriate halo data: avoids branch and ensures load is safe.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Loop contains OpenMP atomics, intrinsics, inline assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue</td>
<td>Compiler cannot vectorize these things (atomics allowed in OpenMP 5.0)</td>
</tr>
<tr>
<td>Original</td>
<td>for (...) {</td>
</tr>
<tr>
<td></td>
<td>// vectorizable</td>
</tr>
<tr>
<td></td>
<td>// non-vectorizable</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Transform</td>
<td>Separate vectorizable and non-vectorizable code into two loops, or use vector_variant.</td>
</tr>
</tbody>
</table>
Performance Considerations – Lane Divergence

#pragma omp simd reduction(+:....)
For (p=0; p<N; p++)
{
    // Blue work
    if(...)
    {
        // Green work
    }
    else
    {
        // Red work
    }
    while(...) 
    {
        // Gold work
        // Purple work
    }
    y = foo (x);
    Pink work
}

- Masking enables vectorization of complex control flows, but divergence impacts SIMD efficiency.
- Use Intel® Vector Advisor XE or SDE to identify how many of your lanes are masked out.
AVX-512F: Compress

A compress “packs” data from non-contiguous locations in a register into contiguous memory.

```
n = 0;
#pragma vector always assert
for (i = 0; i < VLEN; ++i)
{
    if (mask[i])
    {
        packed[n++] = data[i];
    }
}
```

```
n = 0;
#pragma omp simd
for (i = 0; i < VLEN; ++i)
{
    if (mask[i])
    {
        #pragma omp ordered simd monotonic(n)
        packed[n++] = data[i];
    }
}
```
AVX-512F: Expand

Data:

Unpacked Data:

Mask:

An expand “unpacks” contiguous data from memory into non-contiguous locations in a register.

```
n = 0;
#pragma vector always assert
for (i = 0; i < VLEN; ++i)
{
    if (mask[i])
    {
        unpacked[i] = data[n++];
    }
}
```

```
n = 0;
#pragma omp simd
for (i = 0; i < VLEN; ++i)
{
    if (mask[i])
    {
        #pragma omp ordered simd monotonic(n)
        unpacked[i] = data[n++];
    }
}
```
Example: Compress & Expand for “SIMD Queue”

- Use compress & expand to recover SIMD inefficiency from divergence.
- If a function is expensive but not called often, consider building a list to defer function execution until later.
- Speed-up depends on function cost relative to compress/expand.
- More complex queue/enqueue behavior can be useful (e.g. for Monte Carlo particle transport).

```c
int* queue;
for (int i = 0; i < N; ++i)
{
    int qlen = 0;
    #pragma omp simd
    for (int k = 0; k < num_neighbors[i]; ++k)
    {
        int j = neighbors[i][k];
        float rsq = distance(x[i], x[j]);
        if (rsq < cutoff)
        {
            #pragma omp ordered simd monotonic(qlen)
            queue[qlen++] = j;
        }
    }
    #pragma omp simd
    for (int k = 0; k < qlen; ++k)
    {
        int j = queue[k];
        f[i] += force(x[i], x[j]);
    }
}
```
Performance Considerations – Data Layout

- “Gather” and “Scatter” operations:
  - Uniform base pointer
  - 16 x 32-bit or 8 x 64-bit offsets

- Hardware is still optimized for contiguous loads/stores:
  - Gather/scatter instructions have limited throughput
  - Non-contiguous memory accesses are not cache friendly

- Can arise from indirection or object-oriented programming (e.g. structs, classes)

```c
#pragma omp simd
for (i = 0; i < N; ++i)
{
    int j = index[i];
    foo(data[j]);
}

#pragma omp simd
for (i = 0; i < N; ++i)
{
    foo(data[i].x, data[i].y, data[i].z);
}
```
## Performance Considerations – Data Layout

<table>
<thead>
<tr>
<th>Array-of-Structs (AoS)</th>
<th>Struct-of-Arrays (SoA)</th>
<th>Hybrid (AoSoA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x y z x y z x y z</td>
<td>x x x x x x x</td>
<td>x x y y z z z</td>
</tr>
<tr>
<td>x y z x y z x y z</td>
<td>y y y y y y y</td>
<td>x x y y z z z</td>
</tr>
<tr>
<td>x y z x y z x y z</td>
<td>z z z z z z z</td>
<td>x x y y z z z</td>
</tr>
</tbody>
</table>

- **Pros:**
  - Good locality of \{x, y, z\}.
  - 1 memory stream.

- **Cons:**
  - Potential for gather/scatter.

- **Pros:**
  - Contiguous load/store.

- **Cons:**
  - Poor locality of \{x, y, z\}.
  - 3 memory streams.

- **Pros:**
  - Contiguous load/store.
  - 1 memory stream.

- **Cons:**
  - Not a “normal” layout.
AVX-512 Prefetch (PFI)

- Direct prefetches for linear accesses:
  `prefetcht0, prefetcht1, prefetcht2`
  `prefetchnta`

- Indirect prefetches for gather/scatter:
  `vgatherpf0*, vgatherpf1*`
  `vscatterpf0*, vscatterpf1*`

- User-directed prefetches:
  `#pragma prefetch var:level:distance _mm_prefetch(address, level);`

- Compiler-generated prefetches:
  `-qopt-prefetch=5`

```c
#pragma prefetch A:1:3
#pragma omp simd
for (int i = 0; i < N; ++i)
{
    C[i] = A[B[i]];
}
```

Remark: #25033: Number of indirect prefetches=1, dist=2
Remark: #25035: Number of pointer data prefetches=2, dist=8
Remark: #25150: Using directive-based hint=1, distance=3 for indirect memory
Remark: #25540: Using gather/scatter prefetch for indirect memory reference, dist=3
AVX-512 Conflict Detection (CDI)

- Indirect accesses can introduce write-conflicts.

- With AVX-512 CDI:
  1. Test indices for conflicts at run-time
  2. Loop over conflict-free subsets

- Test and loop add overhead; should not be used if you know there are no conflicts!

```c
#pragma vector always assert
for (i = 0; i < N; ++i) {
    int j = index[i];
    histogram[j]++;
}

#pragma omp simd
for (i = 0; i < N; ++i) {
    int j = index[i];
    #pragma omp ordered simd overlap(j)
    histogram[j]++;
}
```
Summary

- Knights Landing is a high-throughput successor to Knights Corner:
  - Socketable, bootable processor with access to large amounts of RAM
  - Greatly improved single-thread performance
  - Very high bandwidth, flexible MCDRAM

- Code modernization is **required** to fully exploit the features of the chip:
  - Use all the cores through parallelization
  - Use all the SIMD lanes through vectorization
  - Use high-bandwidth memory
Recommended Reading (1)

- “High Performance Parallelism Pearls” by James Reinders and Jim Jeffers

- Written as “cookbooks” for modern parallel programming:
  - Real world code examples.
  - Successful techniques for vectorization, load balancing, data structure and memory tuning.
  - Multiple application domains.

- www.lotsofcores.com for code downloads.
Recommended Reading (2)

- “The Knights Landing Book” by Jim Jeffers, James Reinders and Avinash Sodani (KNL Architect)

- Covers everything you need to know about Knights Landing:
  - Section I: Knights Landing Microarchitecture
  - Section II: Programming for Knights Landing
  - Section III: “Pearls” (Application Studies)

- www.lotsofcores.com for code downloads.
Intel eXtreme Performance Users Group (IXPUG)

- Independent group of >400 users

- Conferences and workshops throughout the year:
  - IXPUG Spring & Fall Conferences
  - Workshops & BoFs at SC, ISC, others

Legal Disclaimers

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. § For more information go to www.intel.com/benchmarks.

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 www.intel.com. §

§ Configurations:

Intel, the Intel logo, Look Inside, Xeon, Xeon Phi, are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

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

© Intel Corporation.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Utilizing SIMD – Intel® Intrinsics Guide

The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, and more - without the need to write assembly code.


Filter by ISA.

Filter by functionality.

Expand any intrinsic for a detailed description.
AVX-512 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.
**vector_variant**

- Specifies a vector variant of a scalar function.
  - Compiler will call vector variant if function is called in an auto-vectorized loop.
  - Can be used in conjunction with `#pragma omp declare simd`.

```c
__declspec(noinline) float MyAdd(float* a, int b) { return *a + b; }
__declspec(vector_variant(implements(MyAdd(float* a, int b)), linear(a), vectorlength(16), nomask, processor(future_cpu_23))) __m512 MyAddVec(float* v_a, __m512i v_b)
{
    __m512 cvt_b = _mm512_cvtepi32_ps(v_b);
    return _mm512_add_ps(*((__m512*)v_a), cvt_b);
}

#pragma omp simd
for (int i = 0; i < N; ++i)
{
    x[i] = MyAdd(&y[i], i);
}
```

- **vector_variant functions cannot be inlined (yet).**

https://software.intel.com/en-us/node/523350