The Intel® Xeon Phi Coprocessor

Dr.-Ing. Michael Klemm
Software and Services Group
Intel Corporation
(michael.klemm@intel.com)
Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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.

Copyright 2014©, Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, Core, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

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 SSSE3 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
Intel Technologies for HPC

**Processors**
*Intel® Xeon® Processor*

**Coprocessor**
*Intel® Many Integrated Core*

**Network & Fabric**

**I/O & Storage**

**Software & Services**}

Copyright® 2014, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.*
Transforming the Economics of HPC

Executing to Moore’s Law

Predictable Silicon Track Record – well and alive at Intel. Enabling new devices with higher performance and functionality while controlling power, cost, and size.

Future options subject to change without notice.
Driving Innovation and Integration
Enabled by Leading Edge Process Technologies

Integrated Today

Coming in the Future
The Magic of Integration
Moore's Law at Work & Architecture Innovations

1970s
150 MFLOPS
CRAY-1

2013
1000000 MFLOPS
Intel® Xeon Phi™

6666x
#1  TOP500 June 2014

33  PFLOPS HPL

54  PFLOPS Peak

32000  Intel® Xeon® E5v2 Processors

48000  Intel® Xeon Phi™ Coprocessors
Intel® Many Integrated Core Architecture (Intel® MIC) & Intel® Xeon Phi™ Coprocessor
Intel Architecture Multicore and Manycore

<table>
<thead>
<tr>
<th></th>
<th>Intel Xeon processor 64-bit</th>
<th>Intel Xeon processor 5100 series</th>
<th>Intel Xeon processor 5500 series</th>
<th>Intel Xeon processor 5600 series</th>
<th>Intel Xeon E5 Product Family</th>
<th>Intel Xeon Ivy Bridge</th>
<th>Intel Xeon Haswell</th>
<th>Intel Xeon Phi™ Coprocessor code name Knights Cornerr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core(s)</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>12</td>
<td>18</td>
<td>61</td>
</tr>
<tr>
<td>Threads</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>24</td>
<td>36</td>
<td>244</td>
</tr>
</tbody>
</table>

Images do not reflect actual die sizes. Actual production die may differ from images.

Intel® Xeon Phi™ Coprocessor extends established CPU architecture and programming concepts to highly parallel applications.
Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit

Core unit based on Intel® Pentium® processor family

- Two pipelines (U and V)
  - Dual-issue on scalar instructions
- Scalar pipeline 1 clock latency
- 64-bit data path

4 hardware threads per core

- Thread context: GPRs, ST0-7, etc.
- “Smart” round-robin scheduling
  - Prefetch buffers 2 inst-bundles / context
  - Next ready context selected in order
Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit

Instruction decoder is fully pipelined but is designed as a 2-cycle unit

- Enables significant increase to maximum core frequency, but...
  - Core cannot issue instructions from same context in adjacent cycles
  - Means minimum two threads per core to use all available compute cycles
Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread vector unit

Vector unit width 512 bits!

- 32 512-bit vector registers per context
  - Each holds 16 floats or 8 doubles
  - ALUs support int32/float32 operations, float64 arithmetic, int64 logic ops
  - Ternary ops including Fused-Multiply-Add
  - Broadcast/swizzle support, float16 up-convert
  - 8 vector mask registers for per lane conditional operations
  - Most ops: 4-cycle latency 1-cycle throughput
    - Matches 4-cycle round robin of integer unit
  - Mostly IEEE 754 2008 compliant
    - Not supported: MMX™ technology, Streaming SIMD Extensions (SSE), Intel® Advanced Vector Extensions (Intel® AVX)
Individual cores are tied together via fully coherent caches into a bidirectional ring on the Intel® Xeon Phi™ coprocessor.

**L1** 32K I/D-cache per core
- 1 cycle access latency
- 3 cycle addr-gen interlock
- 8-way associativity
- 64-byte cache line
- ~38 concurrent access/core

**L2** 512K cache per core
- 11 cycle raw latency
- 8-way associativity
- 64-byte cache line
- Streaming HW prefetcher
- ~38 concurrent access/core

**Bidirectional ring**
- 180 GB/sec
- Distributed Tag Directory (DTD) reduces ring snoop traffic
- Gen2x16 PCI Express* 64-256 byte packets peer-to-peer R/W

**GDDR5 Memory**
- 16 32-bit channels
- Up to 5.5 GT/sec
- 8 GB – 275 ns access latency

**PCle**

---

Copyright© 2014, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.
## Cache Hierarchy

<table>
<thead>
<tr>
<th>Parameter</th>
<th>L1</th>
<th>L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coherence</td>
<td>MESI</td>
<td>MESI</td>
</tr>
<tr>
<td>Size</td>
<td>32KB + 32 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>Associativity</td>
<td>8-way</td>
<td>8-way</td>
</tr>
<tr>
<td>Line Size</td>
<td>64 Bytes</td>
<td>64 Bytes</td>
</tr>
<tr>
<td>Banks</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Access Time</td>
<td>2 cycle</td>
<td>11 cycle</td>
</tr>
<tr>
<td>Policy</td>
<td>Pseudo LRU</td>
<td>Pseudo LRU</td>
</tr>
<tr>
<td>Duty Cycle</td>
<td>1 per clock</td>
<td>1 per clock</td>
</tr>
<tr>
<td>Ports</td>
<td>Read or Write</td>
<td>Read or Write</td>
</tr>
</tbody>
</table>

There is no L3 cache!
Intel® Xeon Phi™ Product Family
based on Intel® Many Integrated Core (MIC) Architecture

2013:
Intel® Xeon Phi™ Coprocessor x100 Product Family
“Knights Corner”
22 nm process
Up to 61 Cores
Up to 16 GB Memory

2015:
Intel® Xeon Phi™ Coprocessor x200 Product Family
“Knights Landing”
14 nm
Processor & Coprocessor
Up to 72 cores
On Package, High-Bandwidth Memory

Future Knights:
Upcoming Gen of the Intel® MIC Architecture
In planning
Continued roadmap commitment

*Per Intel’s announced products or planning process for future products

Copyright © 2014, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.
Next Intel® Xeon Phi™ Processor
Codename: Knights Landing

- Designed using Intel’s cutting-edge 14nm process
- Not bound by “offloading” bottlenecks
- Standalone CPU or PCIe Coprocessor
- Leadership compute & memory bandwidth
- Integrated On-Package Memory

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.
Next Intel® Xeon Phi™ Processor
Codename: Knights Landing

Compute: Energy-efficient IA cores²
- Microarchitecture enhanced for HPC³
- 3x Single Thread Performance vs Knights Corner⁴
- Intel Xeon Processor Binary Compatible⁵

On-Package Memory:
- up to 16GB at launch
- 5X Bandwidth vs DDR4⁷
- 1/3x the Space⁶
- 5X Power Efficiency⁶

Jointly Developed with Micron Technology

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Copyright © 2014, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.
Programming for Intel Architectures
Highly Parallel Applications

Efficient vectorization, threading, and parallel execution drives higher performance for suitable scalable applications.
Parallel Programming for Intel® Architecture

<table>
<thead>
<tr>
<th>NODES</th>
<th>Use Intel® MPI, Co-Array Fortran</th>
</tr>
</thead>
</table>
| CORES | Use threads directly (pthreads) or via OpenMP®, C++11  
       | Use tasking, Intel® TBB / Cilk™ Plus |
| VECTORS | Intrinsics, auto-vectorization, vector-libraries  
        | Language extensions for vector programming (SIMD) |
| BLOCKING | Use caches to hide memory latency  
        | Organize memory access for data reuse |
| DATA LAYOUT | Structure of arrays facilitates vector loads / stores, unit stride  
            | Align data for vector accesses |

Parallel programming to utilize the hardware resources, in an abstracted and portable way
Heterogeneous Programming

- Directive-based offloading & Virtual Shared Memory model
- Messaging libraries and internal infrastructure
- Parallel Compute

Intel® MKL
OpenMP®
OpenCL
Intel® TBB
Intel® Cilk™ Plus
C/C++
Fortran

Intel Parallel Building Blocks

Copyright © 2014, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.
Native Programming for Intel Xeon Phi

Parallelization
- Intel® Math Kernel Library
- OpenMP*
- Intel® Threading Building Blocks
- Intel® Cilk™ Plus
- POSIX threads*

Vectorization
- Intel® Math Kernel Library
- Auto vectorization
- Semi-auto vectorization: #pragma (vector, ivdep, simd)
- Intel® Cilk™ Plus Array Notations
- C/C++ Vector Classes (F32vec16, F64vec8)
- Intrinsics

Ease of use
Fine control

Copyright© 2014, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.
Flexible Execution Models for Heterogeneous Platforms

SOURCE CODE

Compilers, Libraries, Runtime Systems

MAIN()

SOURCE CODE

RESULTS

SERIAL AND MODERATELY PARALLEL CODE

HIGHLY PARALLEL CODE

Multicore Only

Multicore Hosted with Manycore Offload

Symmetric

Manycore Only (Native)

Compilation, Libraries, Runtime Systems

MAIN()

RESULTS

MAIN()

RESULTS

MAIN()

RESULTS

MAIN()

RESULTS

Copyright© 2014, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.
Case Study: NWChem CCSD(T)
Finding Offload Candidates

Requirements for offload candidates
- Compute-intensive code regions (kernels)
- Highly parallel
- Compute scaling stronger than data transfer, e.g., compute $O(n^3)$ vs. data size $O(n^2)$

Finding offload candidates
- Create a benchmark to trigger application code of interest
- Find hotspots in the application
- Determine input and output data
- Determine data sizes transferred
Offload Analysis Methodology

1. Create Benchmark
2. Hotspot Analysis
3. Call-Tree Analysis
4. Optimizing Kernel
5. Optimizing Data Flow
6. Add Offload
7. Loop Analysis
Example: NWChem Hotspots

NWChem hotspot profile

Communication

Total of 38% of time spent in sd_t_dX_Y
Example: NWChem Hotspots

Call-tree analysis shows relationship of hotspots

This function is the common anchor for all hotspots.
Example: Loop Analysis

All kernels expose the same structure

7 perfectly nested loops

Trip count per loop is equal to “tile size” (20-30)

Naïve per-kernel solution is obvious

```
subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
                      h7d,triplesx,t2sub,v2sub)

implicit none
integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
integer h3,h2,h1,p6,p5,p4,h7
double precision triplesx(h3d,h2d,h1d,p6d,p5d,p4d)
double precision t2sub(h7d,p4d,p5d,h1d)
double precision v2sub(h3d,h2d,p6d,h7d)

do p4=1,p4d
  do p5=1,p5d
    do p6=1,p6d
      do h1=1,h1d
        do h2=1,h2d
          do h3=1,h3d
            do h7=1,h7d
              triplesx(h3,h2,h1,p6,p5,p4)=
              triplesx(h3,h2,h1,p6,p5,p4)
              - t2sub(h7,p4,p5,h1)*v2sub(h3,h2,p6,h7)
            enddo
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
end subroutine
```
Issues w/ Naïve Offload Solution

Offloading individual kernels requires GBs of data transfers (1-2 GB per offload)

Outer loop does not expose enough parallelism (20-30 threads)

Vectorization potential too low (about 80%)

```fortran
subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
1 h7d,triplesx,t2sub,v2sub)
  implicit none
  integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
  integer h3,h2,h1,p6,p5,p4,h7
  double precision triplesx(h3d,h2d,h1d,p6d,p5d,p4d)
  double precision t2sub(h7d,p4d,p5d,h1d)
  double precision v2sub(h3d,h2d,p6d,h7d)
  cdir$ offload target(mic) in(t2sub:length(h7d*p4d*p5d*h1d))
1 in(v2sub:length(h3d*h2d*p6d*h7d))
2 inout(triplesx:length(h3d*h2d*h1d*p6d*p5d*p4d))
  !$omp parallel do
do p4=1,p4d
do p5=1,p5d
do p6=1,p6d
do h1=1,h1d
do h2=1,h2d
do h3=1,h3d
do h7=1,h7d
  triplesx(h3,h2,h1,p6,p5,p4)=triplesx(h3,h2,h1,p6,p5,p4)
  - t2sub(h7,p4,p5,h1)*v2sub(h3,h2,p6,h7)
  !$omp end parallel do
end
```
Optimization of Data Transfers

Use call-tree analysis to find common anchor for hotspots

- Hoist data transfers up as high as possible
- Make offload regions as large as possible

```c
offload_transfer target(mic) nocopy(triplesx:length(triplesx) ALLOC)
offload_transfer target(mic) nocopy(t2sub:length(t2sub) ALLOC)
offload_transfer target(mic) nocopy(v2sub:length(v2sub) ALLOC)
offload target(mic) nocopy(triplesx:length(0) REUSE)

call zero_triplesx(triplesx)
do ...
if (...) 
  offload target(mic) in(triplesx:length(0),REUSE)
  in(t2sub:length(t2sub),REUSE)
in(v2sub:length(v2sub),REUSE)
  in(h3d,h2d,h1d,p6d,p5d,p4d,h7d)
  call sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,h7,triplesx,t2sub,v2sub)
endif

c sd_t_d1_2 until sd_t_d1_9
enddo

c Similar structure for sd_t_d2_1 until sd_t_d2_9

offload_transfer target(mic) out(triplesx:length(triplesx) REUSE)
```
Kernel Optimizations

subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
  h7d,triplesx,t2sub,v2sub)
  implicit none
  integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
  integer h3,h2,h1,p6,p5,p4,h7
  double precision triplesx(h3d,h2d,h1d,p6d,p5d,p4d)
  double precision t2sub(h7d,p4d,p5d,h1d)
  double precision v2sub(h3d,h2d,p6d,h7d)
!
!$omp parallel do collapse(3)
  do p4=1,p4d
    do p5=1,p5d
      do p6=1,p6d
        do h1=1,h1d
          do h2=1,h2d
            do h3=1,h3d
              do h7=1,h7d
                triplesx(h3,h2,h1,p6,p5,p4)=triplesx(h3,h2,h1,p6,p5,p4)
                1 - t2sub(h7,p4,p5,h1)*v2sub(h3,h2,p6,h7)
              enddo
            enddo
          enddo
        enddo
      enddo
    enddo
  enddo
end
!$omp end parallel do

Use collapse clause to increase parallelism by 1-2 orders of magnitude.

Outer loop does not expose enough parallelism (20-30 threads)
Kernel Optimizations, Part 2

- Loop ordering not optimal for SIMD execution
  - Too low trip count for inner loop
- Index analysis shows that loops can be reordered
  - Swap h7 and h3
  - Swap h7 and h2 again

```fortran
subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
  h7d,triplesx,t2sub,v2sub)
  implicit none
  integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
  integer h3,h2,h1,p6,p5,p4,h7
  double precision triplesx(h3d,h2d,h1d,p6d,p5d,p4d)
  double precision t2sub(h7d,p4d,p5d,h1d)
  double precision v2sub(h3d,h2d,p6d,h7d)
  !$omp parallel do collapse(3)
  do p4=1,p4d
    do p5=1,p5d
      do p6=1,p6d
        do h1=1,h1d
          do h2=1,h2d
            do h3=1,h3d
              do h7=1,h7d
                triplesx(h3,h2,h1,p6,p5,p4)=
                triplesx(h3,h2,h1,p6,p5,p4)
                1
                - t2sub(h7,p4,p5,h1)*v2sub(h3,h2,p6,h7)
                enddo
              enddo
            enddo
          enddo
        enddo
      enddo
    enddo
  enddo
  !$omp end parallel do
end
```
Kernel Optimizations, Part 2

- Loop ordering not optimal for SIMD execution
  - Too low trip count for inner loop
- Index analysis shows that loops can be reordered
  - Swap h7 and h3
  - Swap h7 and h2 again
- Loops h2 and h3 can be collapsed

```fortran
subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
  1 h7d,triplesx,t2sub,v2sub)
  implicit none
  integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
  integer h3,h2,h1,p6,p5,p4,h7
  double precision triplesx(h3d,h2d,h1d,p6d,p5d,p4d)
  double precision t2sub(h7d,p4d,p5d,h1d)
  double precision v2sub(h3d,h2d,p6d,h7d)

!$omp parallel do collapse(3)
  do p4=1,p4d
    do p5=1,p5d
      do p6=1,p6d
        do h1=1,h1d
          do h7=1,h7d
            do h2=1,h2d
              do h3=1,h3d
                triplesx(h3,h2,h1,p6,p5,p4)=
                  triplesx(h3,h2,h1,p6,p5,p4)
                  1 - t2sub(h7,p4,p5,h1)*v2sub(h3,h2,p6,h7)
              enddo
            enddo
          enddo
        enddo
      enddo
    enddo
  enddo
end
```

multi-threading
Kernel Optimizations, Part 2

- Loop ordering not optimal for SIMD execution
  - Too low trip count for inner loop
- Index analysis shows that loops can be reordered
  - Swap h7 and h3
  - Swap h7 and h2 again
- Loops h2 and h3 can be collapsed

```fortran
subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
  h7d,triplesx,t2sub,v2sub)
  implicit none
  integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
  integer h3,h2,h1,p6,p5,p4,h7
  double precision triplesx(h3d,h2d,h1d,p6d,p5d,p4d)
  double precision t2sub(h7d,p4d,p5d,h1d)
  double precision v2sub(h3d,h2d,p6d,h7d)
  !$omp parallel do collapse(3)
  do p4=1,p4d
    do p5=1,p5d
      do p6=1,p6d
        do h1=1,h1d
          do h7=1,h7d
            do h2=1,h2d
              do h3=1,h3d
                triplesx(h3,h2,h1,p6,p5,p4)=triplesx(h3,h2,h1,p6,p5,p4)
                - t2sub(h7,p4,p5,h1)*v2sub(h3,h2,p6,h7)
              enddo
            enddo
          enddo
        enddo
      enddo
    enddo
  enddo
  !$omp end parallel do
  end
```
• Loop ordering not optimal for SIMD execution
  • Too low trip count for inner loop
• Index analysis shows that loops can be reordered
  • Swap h7 and h3
  • Swap h7 and h2 again
• Loops h2 and h3 can be collapsed

```fortran
subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
  h7d,triplesx,t2sub,v2sub)
  implicit none
  integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
  integer h3,h2,h1,p6,p5,p4,h7
  double precision triplesx(h3d*h2d,h1d,p6d,p5d,p4d)
  double precision t2sub(h7d,p4d,p5d,h1d)
  double precision v2sub(h3d*h2d,p6d,h7d)
!
  !$omp parallel do collapse(3)
  do p4=1,p4d
    do p5=1,p5d
      do p6=1,p6d
        do h1=1,h1d
          do h7=1,h7d
            do h2h3=1,h2d*h3d
              triplesx(h2h3,h1,p6,p5,p4)=triplesx(h2h3,h1,p6,p5,p4)
                - t2sub(h7,p4,p5,h1)*v2sub(h2h3,p6,h7)
            enddo
          enddo
        enddo
      enddo
    enddo
  enddo
!
  !$omp end parallel do
end
```

about 50% speed-up
Kernel Optimizations, Multi-versioning

subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
1    h7d,triplesx,t2sub,v2sub)
implicit none
integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
integer h3,h2,h1,p6,p5,p4,h7
integer rmndr
double precision triplesx(h3d*h2d,h1d,p6d,p5d,p4d)
double precision t2sub(h7d,p4d,p5d,h1d)
double precision v2sub(h3d*h2d,p6d,h7d)
rmndr = mod(h3d,8) + mod(h2d,8) + mod(h1d,8) +
1    mod(p6d,8) + mod(p5d,8) + mod(p4d,8) +
2    mod(h7d,8)
if (rmndr.eq.0) then
!$omp parallel do collapse(3)
do p4=1,p4d
do p5=1,p5d
do p6=1,p6d
do h1=1,h1d
do h7=1,h7d
!dec$ vector aligned
do h2h3=1,h2d*h3d
    triplesx(h2h3,h1,p6,p5,p4)=triplesx(h2h3,h1,p6,p5,p4)
1    - t2sub(h7,h4,p5,h1)*v2sub(h2h3,p6,h7)
enddo
endif
!$omp end parallel do
!$omp parallel do collapse(3)
do p4=1,p4d
do p5=1,p5d
do p6=1,p6d
do h1=1,h1d
do h7=1,h7d
!dec$ vector aligned
do h2h3=1,h2d*h3d
    triplesx(h2h3,h1,p6,p5,p4)=triplesx(h2h3,h1,p6,p5,p4)
1    - t2sub(h7,h4,p5,h1)*v2sub(h2h3,p6,h7)
enddo
endif
!$omp end parallel do

about 15% speed-up
Device Partitioning

Host executes several MPI ranks
- Utilize coprocessor from several host processes concurrently
- Utilize host CPUs for increased performance

Partition coprocessors through OpenMP* runtime
- Less threading overhead, better overall system utilization
- Rank 0: OFFLOAD_DEVICES=0 KMP_PLACE_THREADS=30c,4t,0o
- Rank 1: OFFLOAD_DEVICES=0 KMP_PLACE_THREADS=30c,4t,30o
- Rank 4: OFFLOAD_DEVICES=1 KMP_PLACE_THREADS=30c,4t,0o
- Rank 5: OFFLOAD_DEVICES=1 KMP_PLACE_THREADS=30c,4t,30o
Performance Results

Performance tests 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. System configuration: Atipa Visione v442 server with two Intel Xeon E5-2670 8-core processors at 2.6 GHz (128 GB DDR3 with 1333 MHz, Scientific Linux release 6.5) and Intel C600 IOH, two Intel Xeon Phi coprocessors 5110P (GDDR5 with 3.6 GT/sec, driver v3.1.2-1, flash image/micro OS 2.1.02.0390, Intel Composer XE 14.0.1.106). Benchmark perturbative triples correction to the CCSD(T) correlation energy of the 1,3,4,5-tetrasilylimidazol-2-ylidene molecule (formula Si$_4$C$_3$N$_2$H$_{12}$) in its triplet state.