Native Computing & Optimization on Xeon Phi

John D. McCalpin, Ph.D.

Texas Advanced Computing Center

Intel has **lots** of good reference material

- Main Software Developers web page:
  - http://software.intel.com/en-us/mic-developer
- A list of links to very good training material at:
  - http://software.intel.com/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture
- Many answers can also be found in the Intel forums:
  - http://software.intel.com/en-us/forums/intel-many-integrated-core
- Specific information about building and running "native" applications:
  - http://software.intel.com/en-us/articles/building-a-native-application-for-intel-xeon-phi-coprocessors
- Debugging:
  - http://software.intel.com/en-us/articles/debugging-intel-xeon-phi-coprocessor-targeted-applications-on-the-command-line

More important reference documents

- Search for these at www.intel.com by document number
  - This is more likely to get the most recent version than searching for the document number via Google.
- Primary Reference:
  - "Intel Xeon Phi Coprocessor System Software Developers Guide" (document 488596 or 328207)
- Advanced Topics:
  - "Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual" (document 327364)
  - "Intel Xeon Phi Coprocessor (codename: Knights Corner) Performance Monitoring Units" (document 327357)

Native Computing & Optimization on Xeon Phi

- Background: How do you use a Xeon Phi?
- What is a “native” application?
- Why would I want to run a native application
- How do I run a native application?
- How do I tune a native application?

How do you get the Xeon Phi to do "stuff"?

- Possible Models
  - **Offload** – starts on host, then sends work to coprocessor
    - Manual: controlled by directives in your source code
    - Automatic: implemented in a library that your code calls
  - **MPI**
    - Symmetric mode: MPI tasks run on both host(s) and coprocessor(s) → topic of next session
    - Coprocessor-only MPI: subset of symmetric mode
  - **Reverse Offload**
    - Start execution on coprocessor, then offload some work to host – not much current support
  - **Native** – runs on coprocessor, with or without MPI

What is a Native Application on Xeon Phi?

- It is possible to login and run applications directly on Xeon Phi
- But, Xeon Phi is *not* binary-compatible with the host
  - Instruction set is *similar* to Pentium, with *most* (not all) of the 64 bit scalar extensions added
  - Instruction set includes new 512-bit vector extensions, but ***not*** MMX, SSE (1/2/3/4), or AVX.
- Applications compiled to run on the Xeon Phi coprocessor are called “native applications”

Why Run a Native Application?

- MPI applications that run in “symmetric mode” require a binary to run on the host(s) and a “native” binary to run on the coprocessors
  - This is the topic of the next talk
- Obviously Coprocessor-Only MPI requires a native binary to run on the coprocessors
- But Native Applications are also an easy way to get acquainted with the properties of the Xeon Phi
  - E.g., run performance studies directly on the Xeon Phi
    - No need to measure and compensate for overhead of offload or MPI

How do I Create a Native Application?

- Native Applications must be cross-compiled
  - No compilers installed on coprocessors
- Xeon Phi is fully supported by the Intel C/C++ and Fortran compilers (version 13 or newer)
  - Adding "-mmic" to the command line causes the compiler to generate a native Xeon Phi executable
  - It is convenient to append ".mic" to the executable name
- Cross-compilation can be done on either the login nodes or the compute nodes
  - Compute node access is for convenience – we have a limited number of compiler licenses, so please don't launch compilations on thousands of cores!

How do I Run a Native Application?

- Options to run on mic0 from (for example) c422-703
  1. Traditional ssh remote command execution
     - `c422-703% ssh mic0 ls`
     - Clumsy if environment variables or directory changes needed
  2. Interactively login to mic:
     - `c422-703% ssh mic0`
     - Then use as a normal server
  3. Explicit launcher:
     - `c422-703% micrun ./a.out.mic`
  4. Implicit launcher:
     - `c422-703% ./a.out.mic`
- The launcher options have nice features →

Native Application Launcher

- The "micrun" launcher has three nice features:
  - It propagates the current working directory to the coprocessor
  - It propagates the shell environment (with translation) to the coprocessor
    - Environment variables that need to be different on host and coprocessor need to be defined using the "MIC_" prefix on the host. E.g.,
      - `c422-703% export MIC_OMP_NUMTHREADS=183`
      - `c422-703% export MIC_KMP_AFFINITY="verbose,balanced"`
  - It propagates the command return code back to the host shell
- These features work whether the launcher is used explicitly or implicitly

Native application quirks

- The Xeon Phi runs a somewhat different version of Linux, based on “BusyBox”
  - Some tools are missing, e.g., “w”, “numactl”
  - Some tools have reduced functionality, e.g., “ps”
- Relatively few libraries have been ported to the coprocessor environment
- These issues make the implicit or explicit launcher approach even more convenient

Best Practices for Running Native Apps

- Always bind processes to cores
  - For OpenMP, the Intel compiler runtime makes this easy:
    - `export KMP_AFFINITY=[compact,scatter,balanced]`
    - "compact" and "scatter" are supported on both host and coprocessor
    - "balanced" is a new distribution supported only on the coprocessor
    - Binding to specific thread contexts ("granularity=fine") may help in codes with heavy L1 cache re-use.
    - Adding "verbose" will cause the application to dump the full affinity information when it is run, e.g.:
      - `export KMP_AFFINITY="verbose,scatter"`
- Xeon Phi is a single chip, so no need for "numactl"
- Task binding for MPI is described in the next presentation

KMP_AFFINITY distribution options

- "compact"
  - E.g., 80 threads mapped 4 threads/core for cores 0..19
  - Allows scaling studies by fully loaded core count
    - I.e., cores 20..60 not used in this case
- "scatter"
  - E.g., 80 threads:
    - first 61 threads mapped one per core for cores 0..60
    - Last 19 threads mapped one (more) per core for cores 0..18
  - Allows "1 thread per core" studies for 1 to 61 threads
- "balanced"
  - Variant of "scatter"
  - E.g., 80 threads:
    - First 38 threads mapped two per core for cores 0..18
    - Remaining 42 threads mapped one per core for cores 19..60
  - Better if adjacent OpenMP threads are sharing data (since hardware contexts share the same L1 & L2 caches)
- "explicit"
  - Allows exact specification of mapping using "proclist" modifier
    - `export KMP_AFFINITY='explicit,proclist=[0,1,2,3,4]'`
  - BUT – watch out for unexpected Logical to Physical Processor Mapping and unexpected OpenMP thread to Logical Processor Mapping → next slide →

Logical to Physical Processor Mapping

- Hardware:
  - Physical Cores are 0..60
  - Logical Cores are 0..243
- Mapping is not what you are used to!
  - Logical Core 0 maps to Physical core 60, thread context 0
  - Logical Core 1 maps to Physical core 0, thread context 0
  - Logical Core 2 maps to Physical core 0, thread context 1
  - Logical Core 3 maps to Physical core 0, thread context 2
  - Logical Core 4 maps to Physical core 0, thread context 3
  - Logical Core 5 maps to Physical core 1, thread context 0
  - […]
  - Logical Core 240 maps to Physical core 59, thread context 3
  - Logical Core 241 maps to Physical core 60, thread context 1
  - Logical Core 242 maps to Physical core 60, thread context 2
  - Logical Core 243 maps to Physical core 60, thread context 3
- OpenMP threads start binding to logical core 1, not logical core 0
  - For “compact” mapping 240 OpenMP threads are mapped to the first 60 cores
    - No contention for the core containing logical core 0 – the core that the O/S uses most
  - But for “scatter” and “balanced” mappings, contention for logical core 0 begins at 61 threads
    - Not much performance impact unless O/S is very busy?
    - Best to avoid core 60 for offload jobs & MPI jobs with compute/communication overlap

Preliminary Performance Tuning Notes (1)

- Xeon Phi always has HyperThreading enabled
  - Four thread contexts per physical core
  - Registers are replicated
  - L1D, L1I, and (private, unified) L2 caches are shared
- Vector Unit instruction issue limitation:
  - The vector unit can issue one instruction per cycle
  - L1D Cache can deliver 64 Bytes (1 vector register) every cycle
  - But a thread can only issue a vector instruction *every other* cycle
  - Using 3-4 threads does not increase maximum issue rate, but often helps tolerate latency

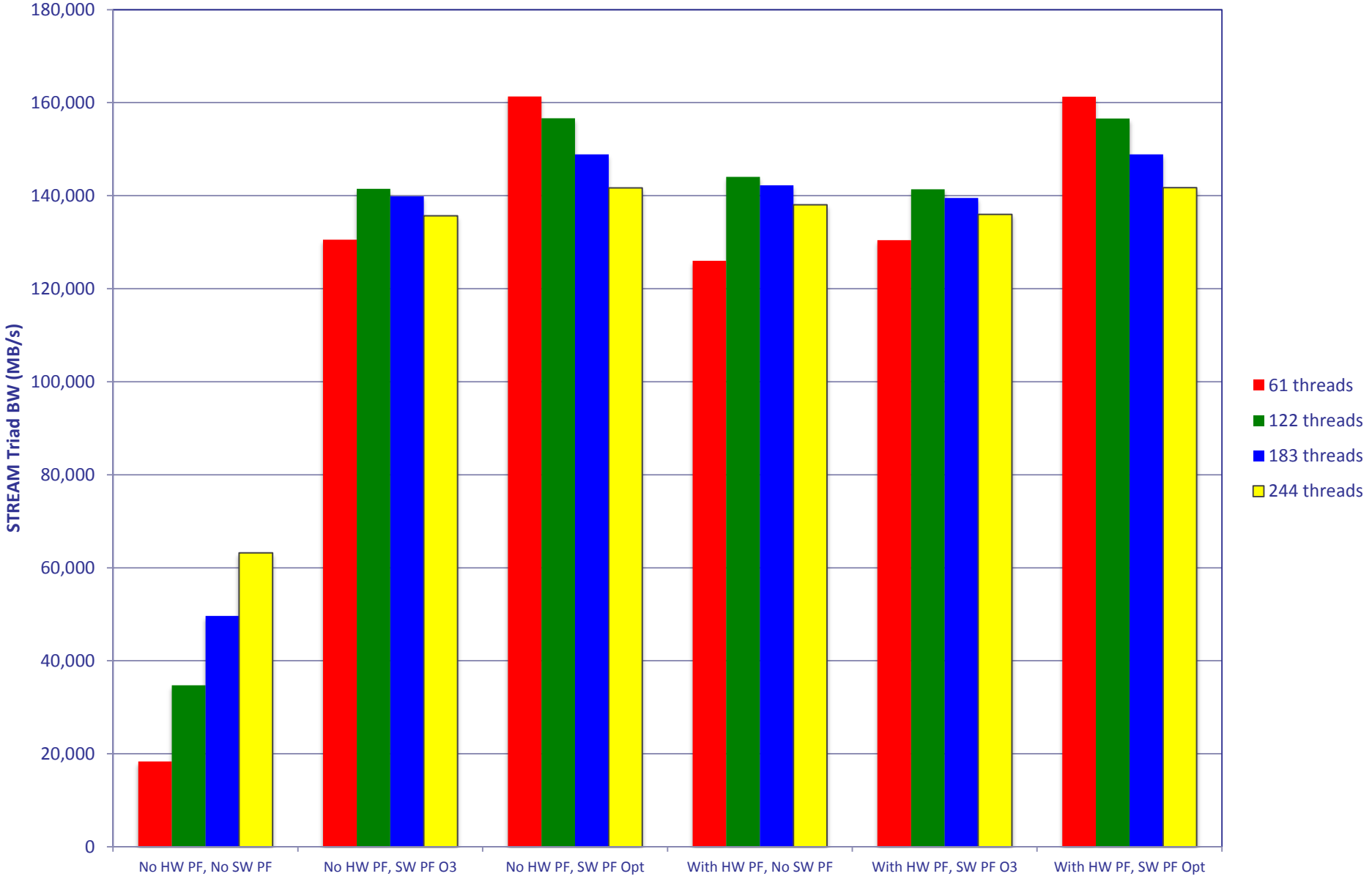
Preliminary Performance Tuning Notes (2)

- Cache Hierarchy:
  - L1I and L1D are 32kB, 8-way associative, 64-Byte cache lines
    - Same sizes & associativity as Xeon E5 ("Sandy Bridge"), but *shared* when using multiple threads/core
    - 1 cycle latency for scalar loads, 3 cycles for vector loads
  - L2 (unified, private) is 512kB, 8-way associative, 64-Byte lines
    - Latency ~25 cycles (idle), increases under load
    - Bandwidth is 1 cache line every other cycle
  - All 60 of the other L2 caches are snooped on an L2 cache miss
    - Clean or Dirty data will be transferred to requestor's L1D
    - This eliminates load from DRAM on shared data accesses
    - Cache-to-Cache transfers are about 275ns, independent of relative core numbers

Preliminary Performance Tuning Notes (3)

- Idle Memory Latency is ~275-280 ns
- Required Concurrency:
  - 277 ns * 352 GB/s = 97,504 Bytes = 1524 cache lines
    - This is ~25 concurrent cache misses per core
    - Theoretically supported by the HW, but not attainable in practice
    - The actual number increases under load as the latency increases
- Hardware Prefetch
  - No L1 prefetchers
  - Simplified L2 prefetcher
    - Only identifies strides up to 2 cache lines
    - Prefetches up to 4 cache-line-pairs per stream
    - Monitors up to 16 streams (on different 4kB pages)
      - These are *shared* by the hardware threads on a core
- Software prefetch is often required to obtain good bandwidth

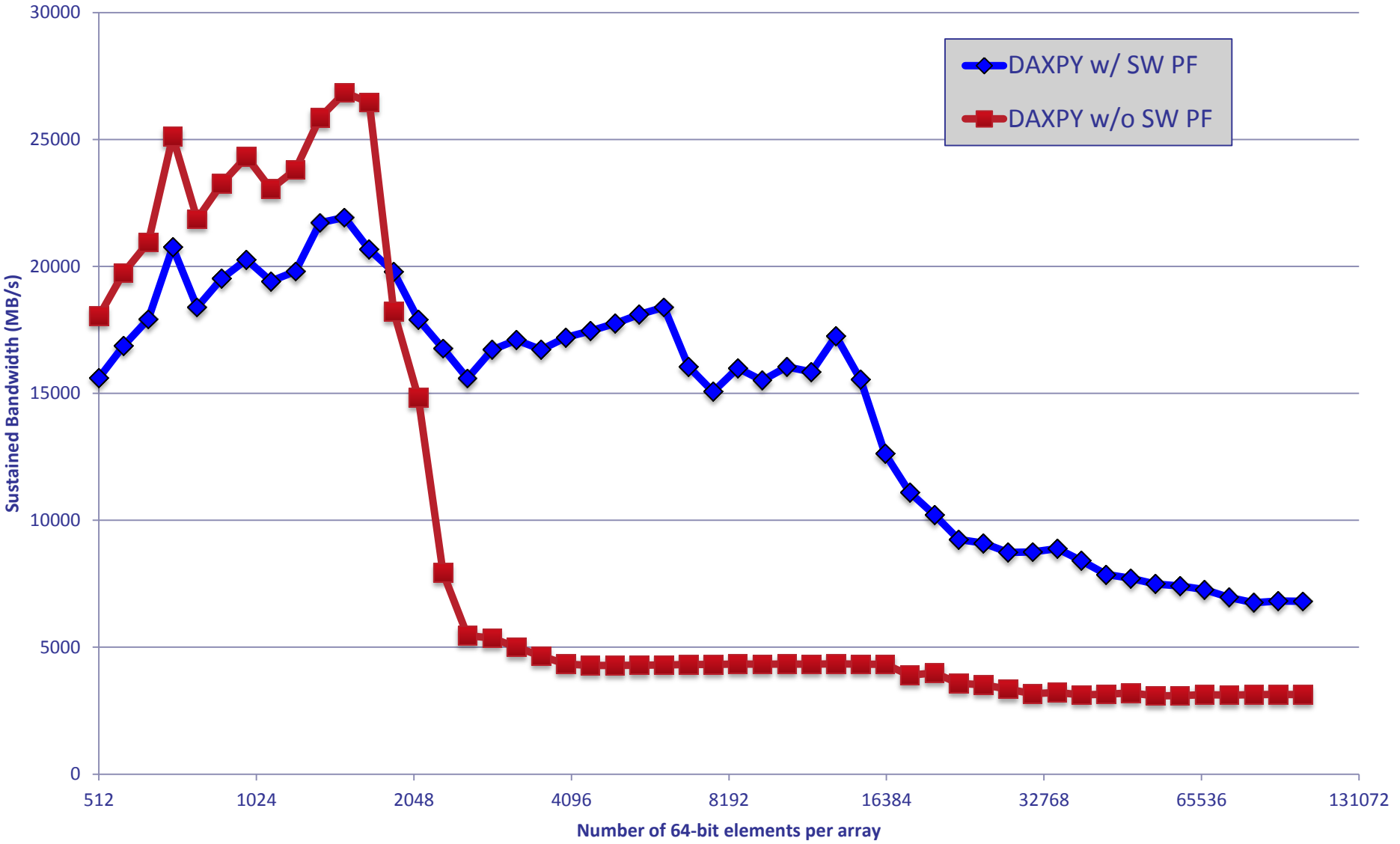
Effect of HW & SW Prefetch on STREAM Triad Bandwidth on Xeon Phi



Software Prefetch vs Data Location

- Xeon Phi can only issue one vector instruction every other cycle from a single thread context, so:
  - If data is already in the L1 Cache, Vector Prefetch instructions use up valuable instruction issue bandwidth
  - But, if data is in the L2 cache or memory, Vector Prefetch instructions provide significant increases in sustained performance.
- The next slide shows the effect of including vector prefetch instructions (default with "-O3") vs excluding them (with "-no-opt-prefetch")
  - Data is L1 contained for array sizes of 2k elements or less
  - Data is L2-contained for array sizes of ~32k elements or less

Stream2 DAXPY on Xeon Phi SE10P: Effect of Software Prefetch on Performance with Data in Cache



Tuning Memory Bandwidth on Xeon Phi

- STREAM Benchmark performance varies considerably with compilation options
  - "-O3" flags, small pages, malloc: 63 GB/s to 98 GB/s
  - "-O3" flags, small pages, -fno-alias: 125 GB/s to 140 GB/s
  - "tuned" flags, small pages: 142 GB/s to 162 GB/s
  - "tuned" flags, large pages: up to 175 GB/s
- Best Performance can be obtained with 1, 2, 3, or 4 threads per core
  - Aggressive SW prefetch or >4 memory access streams per thread gives best results with 1 thread per core
  - Less aggressive SW prefetch or 1-4 memory access streams per thread give better results with more threads
- Details:
  - "-O3" compiler flags:

```
-O3 -openmp -mcmodel=medium -fno-alias
```

  - "tuned" compiler flags use "-O3" flags plus:

```
-mP2OPT_hlo_use_const_pref_dist=64 \
-mP2OPT_hlo_use_const_second_pref_dist=32 \
-mGLOB_default_function_attrs="knc_stream_store_controls=2"
```

How Do I Tune Native Applications?

- Vectorization and Parallelization are critical!
  - Single-thread scalar performance: ~1 GHz Pentium
- Vector width is 512 bits
  - 8 double precision values / 16 single precision values
  - You don't want to lose factors of 8-16 in performance
- Compiler reports provide important information about effectiveness of compiler at vectorization
  - Start with a simple code – the compiler reports can be very long & hard to follow
  - There are lots of options & reports! Details at:
  - http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm

Compiler Reports for Vectorization

- "-vec-report3" gives diagnostic information about every loop, including
  - Loops successfully vectorized (also at –vec-report1)
  - Loops not vectorized & reasons (also at –vec-report2)
  - Specific dependency info for failures to vectorize
  - "-vec-report6" provides additional info:
    - Array alignment for each loop
    - Unrolling depth for each loop
- Quirks
  - Functions typically have most/all of the vectorization messages repeated with the line number of the call site – ignore these and look at the messages with the line number of the actual loop
  - Reported reasons for not vectorizing are not very helpful – look at specific dependency info & remember about C aliasing rules

Examples of multiple messages from vec-report

- Code: STREAM Copy kernel

```
#pragma omp parallel for
        for (j=0; j<STREAM_ARRAY_SIZE; j++)
            c[j] = a[j];
```

- Vec-report messages

```
stream_5-10.c(354): (col. 6) remark: vectorization support: reference c has aligned access.
stream_5-10.c(354): (col. 6) remark: vectorization support: reference a has aligned access.
stream_5-10.c(354): (col. 6) remark: vectorization support: streaming store was generated for c.
stream_5-10.c(353): (col. 2) remark: LOOP WAS VECTORIZED.
stream_5-10.c(354): (col. 6) remark: vectorization support: reference c has unaligned access.
stream_5-10.c(354): (col. 6) remark: vectorization support: reference a has unaligned access.
stream_5-10.c(354): (col. 6) remark: vectorization support: unaligned access used inside loop body.
stream_5-10.c(353): (col. 2) remark: loop was not vectorized: vectorization possible but seems inefficient.
```

- Many other combinations of messages are possible
  - Remember that OpenMP will split loops in ways that break 64-Byte alignment – alignment depends on thread count

Additional Compiler Reports

- "-opt-report-phase hpo" provides good info on OpenMP parallelization
- "-opt-report-phase hlo" provides info on software prefetching
  - This is important on Xeon Phi, but is typically not used on other Xeon processors
- "-opt-report 1" gives a medium level of detail from all compiler phases, split up by routine

Tuning Limitations

- Profiling (as in “gprof”) is ***not*** supported for the “-mmic” target by the Intel compilers
- Profiling ***is*** supported by Intel’s “Vtune” product
  - Vtune is not yet installed on the Stampede nodes
  - Vtune is a relatively complex system that needs its own training session