Native Computing and Optimization on the Intel Xeon Phi Coprocessor

John D. McCalpin
mccalpin@tacc.utexas.edu

Outline

- Intro (very brief)
- Compiling & Running Native Apps
- Controlling Execution
- Tuning
  - Vectorization
  - Parallelization
  - Performance Expectations: Memory Bandwidth
  - Performance Expectations: Cache Latency and Bandwidth

Compiling a native application

- Cross-compile on the host (login or development nodes)
  - No compilers installed on coprocessors
  - Compilers not available on compute nodes (except development queue)
- Xeon Phi is fully supported by the Intel C/C++ and Fortran compilers (v13+):

```
icc   -O3 -openmp -mmic mysource.c   -o myapp.mic
ifort -O3 -openmp -mmic mysource.f90 -o myapp.mic
```

- The *-mmic* flag causes the compiler to generate a native coprocessor executable
- It is convenient to use a *.mic* extension to differentiate coprocessor executables
- Many/most optimization flags are the same for coprocessor

Running a native application

- Options to run on `mic0` from a compute node:
  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`

Environment Variables on the MIC

- If you run directly on the card use the regular names:
  - OMP_NUM_THREADS
  - KMP_AFFINITY
  - I_MPI_PIN_PROCESSOR_LIST
- If you use the launcher, use the MIC_ prefix to define them on the host:
  - MIC_OMP_NUM_THREADS
  - MIC_KMP_AFFINITY
  - MIC_I_MPI_PIN_PROCESSOR_LIST
- You can also define a different prefix:
  - export MIC_ENV_PREFIX=MYMIC
  - MYMIC_OMP_NUM_THREADS

Control for Performance and Repeatability

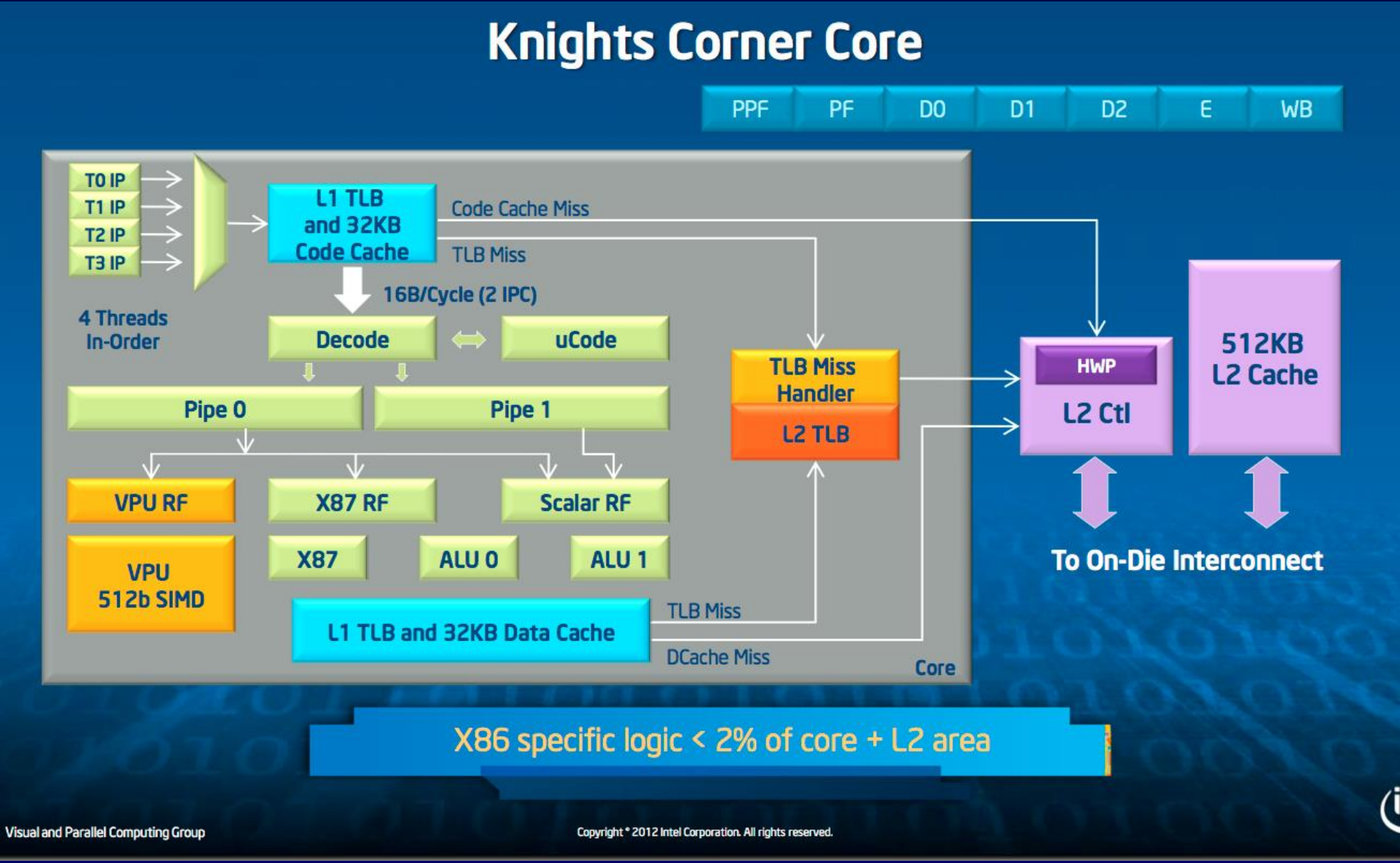
- Always bind processes to cores
  - For MPI tasks (more in next presentation)
    - `I_MPI_PIN, I_MPI_PIN_MODE, I_MPI_PIN_PROCESSOR_LIST`
  - For OpenMP
    - Use environment variables described on the following slides
    - `KMP_AFFINITY` and `KMP_PLACE_THREADS`
- The Xeon Phi is a single chip with interleaved memory controllers, so there is no need for `numactl`
- If the runtime library affinity options can't be used
  - The Linux `taskset` command is available
  - The Linux `sched_getaffinity` & `sched_setaffinity` interfaces are supported
- SEE BACKUP SLIDES FOR MORE DETAILS!!!

Before we can optimize…

XEON PHI KEY HARDWARE FEATURES

Xeon Phi Key Features

- *Single-thread scalar performance: ~1 GHz Pentium*
- Vectorization
  - 512-bit vector width = 8 doubles or 16 floats
  - Understanding aliasing is critical
  - Understanding alignment is important for data in L1, less important for data in L2 or memory
- Parallelization
  - You will need to use most or all of the cores
  - Different codes will do best with different numbers of threads/core
- Xeon Phi has *both* high FLOPS rate and high memory bandwidth
  - Not just an accelerator for computationally dense kernels



George Chrysos, Intel, Hot Chips 24 (2012):

http://www.slideshare.net/IntelXeon/under-the-armor-of-knights-corner-intel-mic-architecture-at-hotchips-2012

TACC

Multi-threading and Instruction Issue

- Xeon Phi always has multi-threading enabled
  - Four thread contexts ("*logical processors*") per physical core
    - Registers are private to each thread context
    - L1D, L1I, and (private, unified) L2 caches are shared by threads
- Thread instruction issue limitation:
  - A **core** can issue 1 or 2 instructions per cycle
    - only 1 can be a vector instruction
    - "vector" prefetch instructions are not vector instructions
  - L1D Cache can deliver 64 Bytes (1 vector register) every cycle
  - But a **thread** can only issue instructions *every other* cycle
  - So: at least two threads needed to fully utilize the vector unit
    - Using 3-4 threads does not increase maximum issue rate, but often helps tolerate latency
- In-order cores stall on L1 cache misses!

OPTIMIZATION PART 1: VECTORIZATION

How Do I Tune for Vectorization?

- 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

- Watch especially for "aliasing" in compiler reports – the compiler cannot vectorize or parallelize if aliasing is present!
  - C aliasing rules can be subtle – learn to use the "`restrict`" keyword

Vectorization Compiler reports

- Option `-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
  - Option `-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

`vec-report` Example

- 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 can break 64-Byte alignment – alignment depends on thread count

Vector Alignment Issues

- Vector loads are full 64 Byte L1 cache lines
  - Arithmetic instructions with memory operands can only operate on aligned data
  - Data that is not 64-Byte aligned has to be loaded *twice!*
    - One load to get the "lower" part of the vector register
    - One load to get the "upper" part of the vector register
  - This can be a problem if most of the data is in cache
    - If there are lots of stalls on memory, then the extra time required for the extra loads is "hidden" and does not degrade performance much
- Some code modification is usually required to enable good alignment
  - Easiest changes are with allocation:

```
a = (double *) memalign(boundary, size)
```

  - More intrusive changes with declarations:

```
__declspec(align(64)) double psum[8];
```

OPTIMIZATION PART 2: PARALLELIZATION

Overheads of Parallelization

- Parallel Synchronization Overheads are often 6x-10x higher on Xeon Phi than on host processors
  - Many more cores, lower frequency, support for higher memory bandwidth, etc.
  - E.g., 240 thread `OMP PARALLEL FOR` ~20 microseconds

- These based on hardware attributes, so they apply to OpenMP, MPI (using shared memory within Xeon Phi), pthreads, etc.
  - There is still room for improvement in the software stacks
  - But overheads will never be as low as we get on the host processors

- So more work is required in each parallel region, and additional serial regions can become bottlenecks
  - Example: 2D code spends 2% of time in serial “boundary” loops on Xeon E5, but >50% on Xeon Phi. Parallelizing these loops reduced this to <10% of total time.

OpenMP Tuning Recommendations:

- Get to know the Intel OpenMP environment variables
  - Use "`export KMP_SETTINGS=1`" ("`MIC_KMP_SETTINGS=1`")
  - Add "`verbose`" to `KMP_AFFINITY/MIC_KMP_AFFINITY`
- Make parallel regions as big as possible
  - OpenMP pragma should be on outermost loop available
  - Include multiple loops if possible, rather than separate OMP PARALLEL FOR on each loop nest
- Static scheduling seems to work best
  - Dynamic scheduling requires more synchronizations?
- Experiment!
  - Be sure to run each case multiple times to get a feel for the intrinsic variability in run time

OPTIMIZATION: MISCELLANEOUS NOTES

Additional Compiler Reports

- Option `-opt-report-phase hpo` provides good info on OpenMP parallelization
- Option `-opt-report-phase hlo` provides info on software prefetching
- Option `-opt-report 1` gives a medium level of detail from all compiler phases, split up by routine
- Option `-opt-report-file=filename` saves the lengthy optimization report output to a file

Profiling Native Applications

- Profiling is supported by Intel's `Vtune` product
  - Available *"Real Soon Now!"* on Stampede
  - Vtune is a complex profiling software that requires its own training session
- Manual timings:
  - The Xeon Phi core cycle counter can be read inline with very low overhead (<10 cycles)

PERFORMANCE EXPECTATIONS: MEMORY BANDWIDTH

Memory Bandwidth Limitations

- Peak Memory Bandwidth of Xeon Phi SE10P is 352 GB/s
  - Best sustained value reported is ~178 GB/s – why?
- Little's Law (from queueing theory) can be rewritten as:
  - Latency * Bandwidth = Concurrency
  - This tells how much data must be "in flight" to "fill the pipeline", given a latency (or queue occupancy) and a bandwidth.
- Xeon Phi SE10P Required Concurrency:
  - Observed average memory latency is ~277 ns (idle system)
  - 277 ns * 352 GB/s = 97,504 Bytes = 1524 cache lines
    - This is ~25 concurrent cache misses per core
    - But each core can only support 8 outstanding L1 Data Cache misses
    - Any additional concurrency must come from the L2 Hardware Prefetchers
    - The actual latency increases under load, further increasing the concurrency requirement
  - So Memory Bandwidth on the Xeon Phi is very strongly tied to the effectiveness of Software and Hardware Prefetching

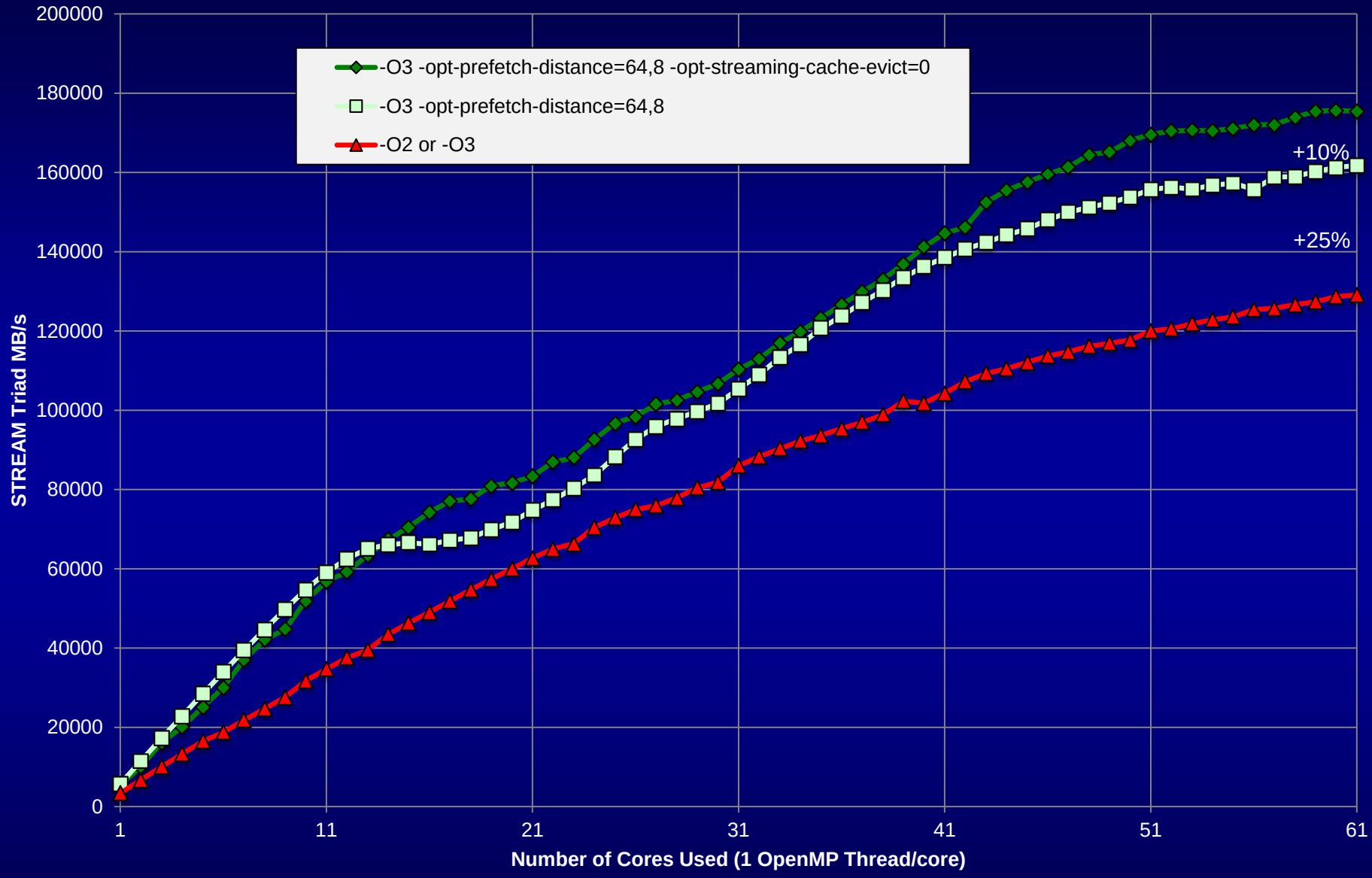
Tuning Memory Bandwidth on the MIC

- STREAM Benchmark performance varies considerably with compilation options
  - "-O3" flags, small pages, malloc: 63 GB/s to 98 GB/s (aliasing!)
  - "-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 Bandwidth might 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

- STREAM Triad gets best performance with 1 thread/core
  - Performance drops with more threads: loses ~10% with 4 threads/core
  - Rule of Thumb:
    - Total Memory Access Streams (streams/thread * #threads) should be =< 256 (number of DRAM banks)

STREAM Triad Bandwidth Scaling on Xeon Phi SE10P



TACC

PERFORMANCE EXPECTATIONS: CACHE LATENCY & BANDWIDTH

Cache Hierarchy Overview

- L1I and L1D are 32kB, 8-way associative, 64-Byte cache lines
  - 1 cycle latency for scalar loads, 3 cycles for vector loads
  - Bandwidth is 64 Bytes/cycle
    - Remember than unaligned data must be loaded twice: `vloadunpacklo, vloadunpackhi`
  - *Shared* by threads on core
- L2 (unified, private, inclusive) is 512kB, 8-way associative, 64-Byte lines
  - Latency ~25 cycles (idle), increases under load
  - Peak Bandwidth is 1 cache line every other cycle
    - Maximum *sustainable* BW is 8 cache lines every ~24 cycles (for reads): ~2/3 of peak
  - *Shared* by threads on core
- On an L2 cache miss: Check directories to see if data is in another L2
  - Clean or Dirty data will be transferred from remote L2 to requestor's L2 and L1D
    - This eliminates load from DRAM on shared data accesses
  - Cache-to-Cache transfers are about 275ns (average), independent of relative core numbers
    - (Latency varies by physical address in a complicated and undocumented way)

Cache Performance Notes

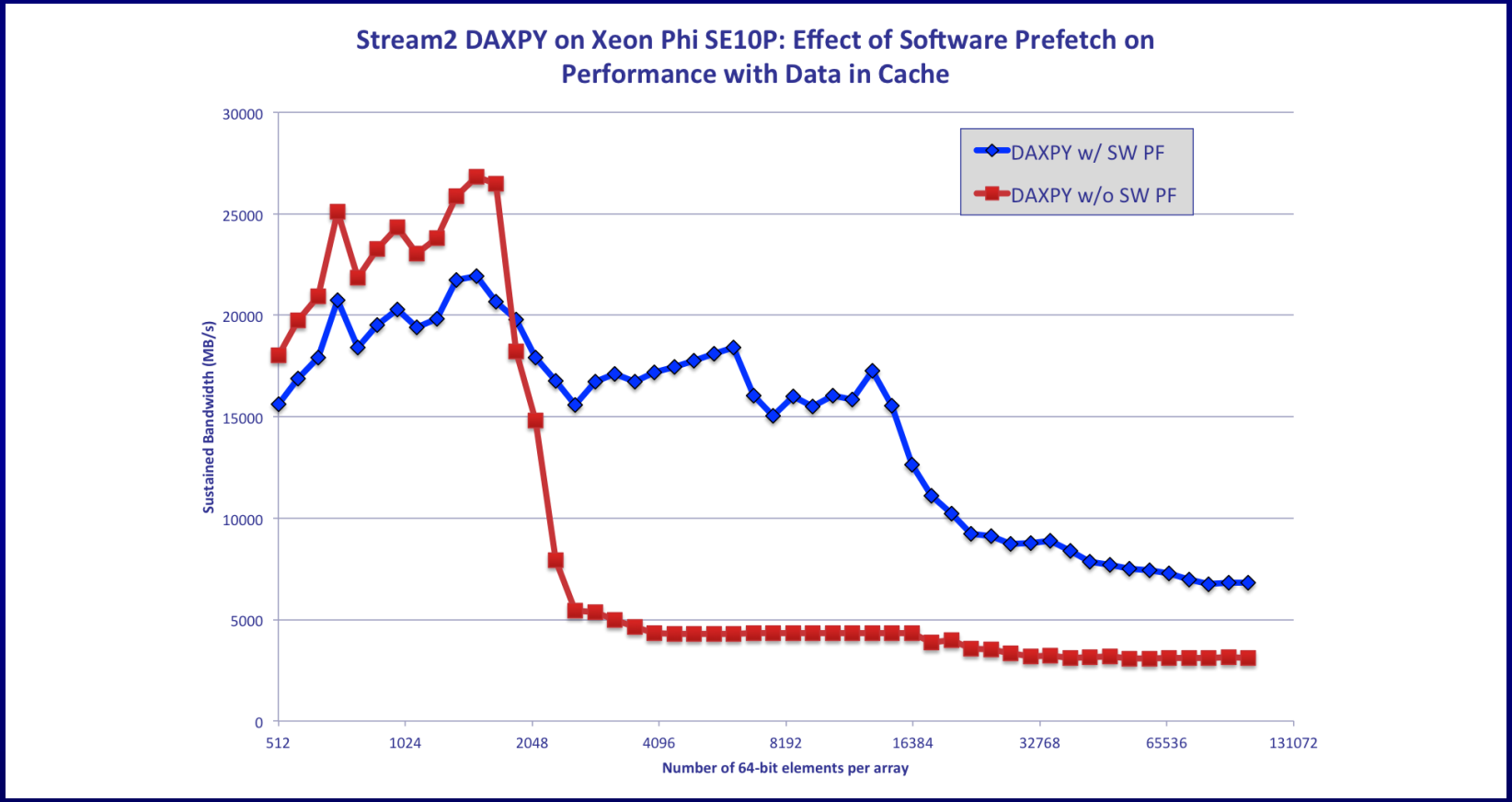
- In-order design *stalls* thread on L1 Data Cache Misses
  - (Except for Store Misses)
  - Other threads on the same core can continue
  - Idle Memory Latency is ~275-280 ns: ~300 processor clock cycles
- Required Concurrency:
  - 277 ns * 352 GB/s = 97,504 Bytes = 1524 cache lines
    - This is ~25 concurrent cache misses per core
    - Theoretically supported by the HW, but not attainable in practice
    - So *sustained bandwidth is limited by concurrency*, not by wires
- Hardware Prefetch
  - No L1 Hardware prefetchers
    - Requires SW prefetch (VPREFETCH0) to tolerate ~24 cycle L2 latency
  - Simplified L2 prefetcher
    - Identifies strides up to 2 cache lines (less aggressive than on Xeon E5)
    - Prefetches up to ~24 cache lines per core
    - Monitors up to 16 streams (on different 4kB pages)
      - These are *shared* by the hardware threads on a core
- Software prefetch tuning is usually required to obtain good bandwidth

Software Prefetch for Data in Cache

- 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

Effect of SW Prefetch with Data on Cache



Intel 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 Intel reference Material

- 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)
  - http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture

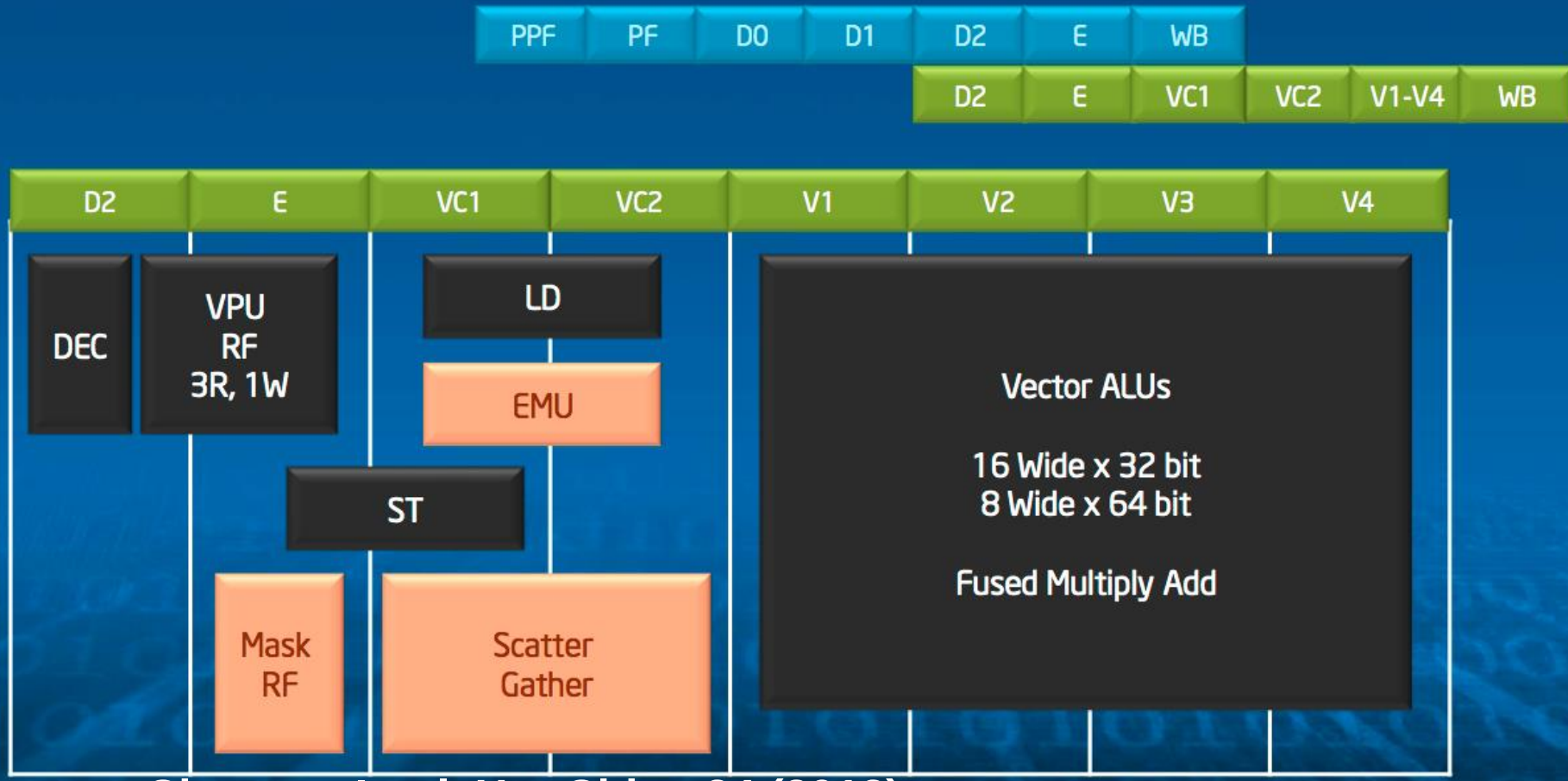
Questions?

For more information:
www.tacc.utexas.edu

TACC

BACKUP Material

Vector Processing Unit



George Chrysos, Intel, Hot Chips 24 (2012):

http://www.slideshare.net/IntelXeon/under-the-armor-of-knights-corner-intel-mic-architecture-at-hotchips-2012

arallel Computing Group

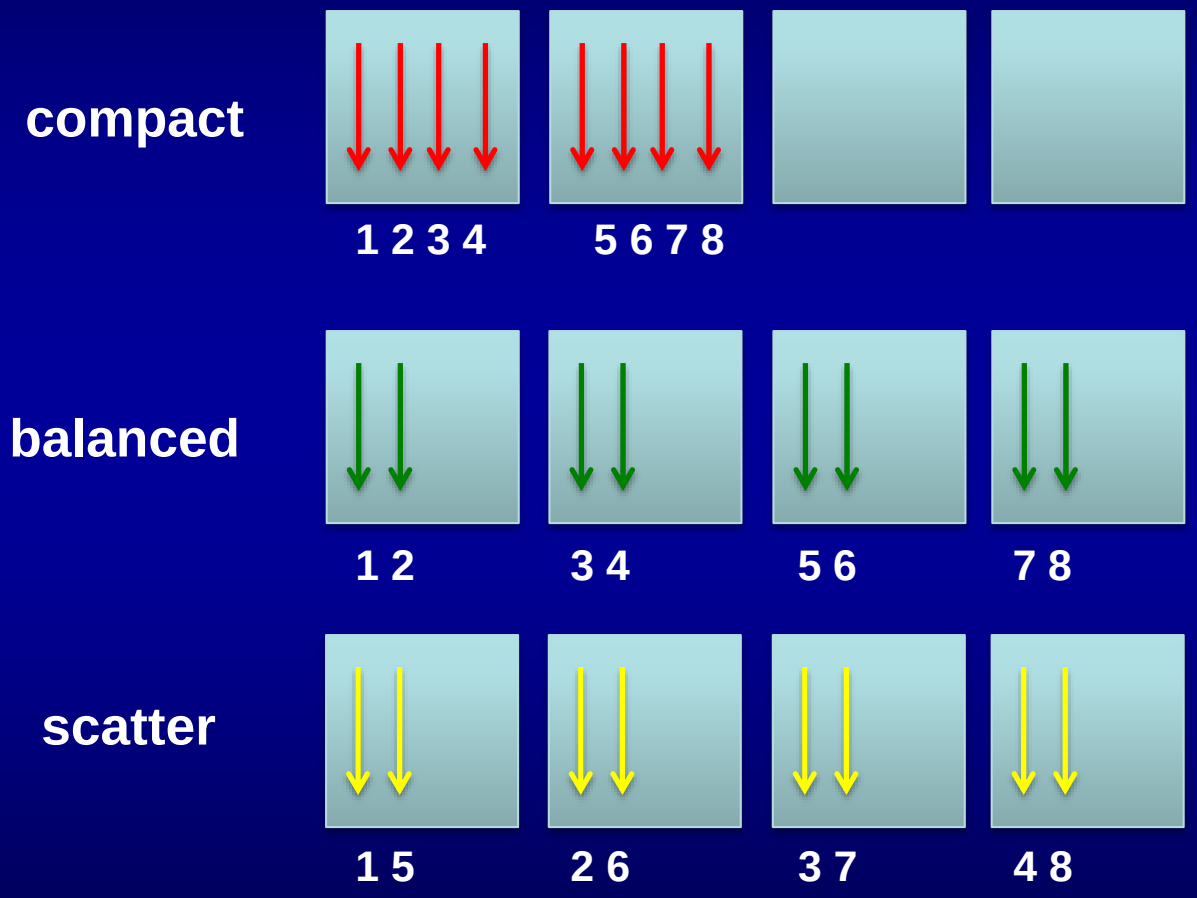
Copyright © 2012 Intel Corporation. All rights reserved.

TACC

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
  - `KMP_PLACE_THREADS` limits the range of cores & thread contexts
    - E.g., `KMP_PLACE_THREADS=60c,2c` with `KMP_AFFINITY=compact` and `OMP_NUM_THREADS=120` places 2 threads on each of the first 60 cores

KMP_AFFINITY distribution options



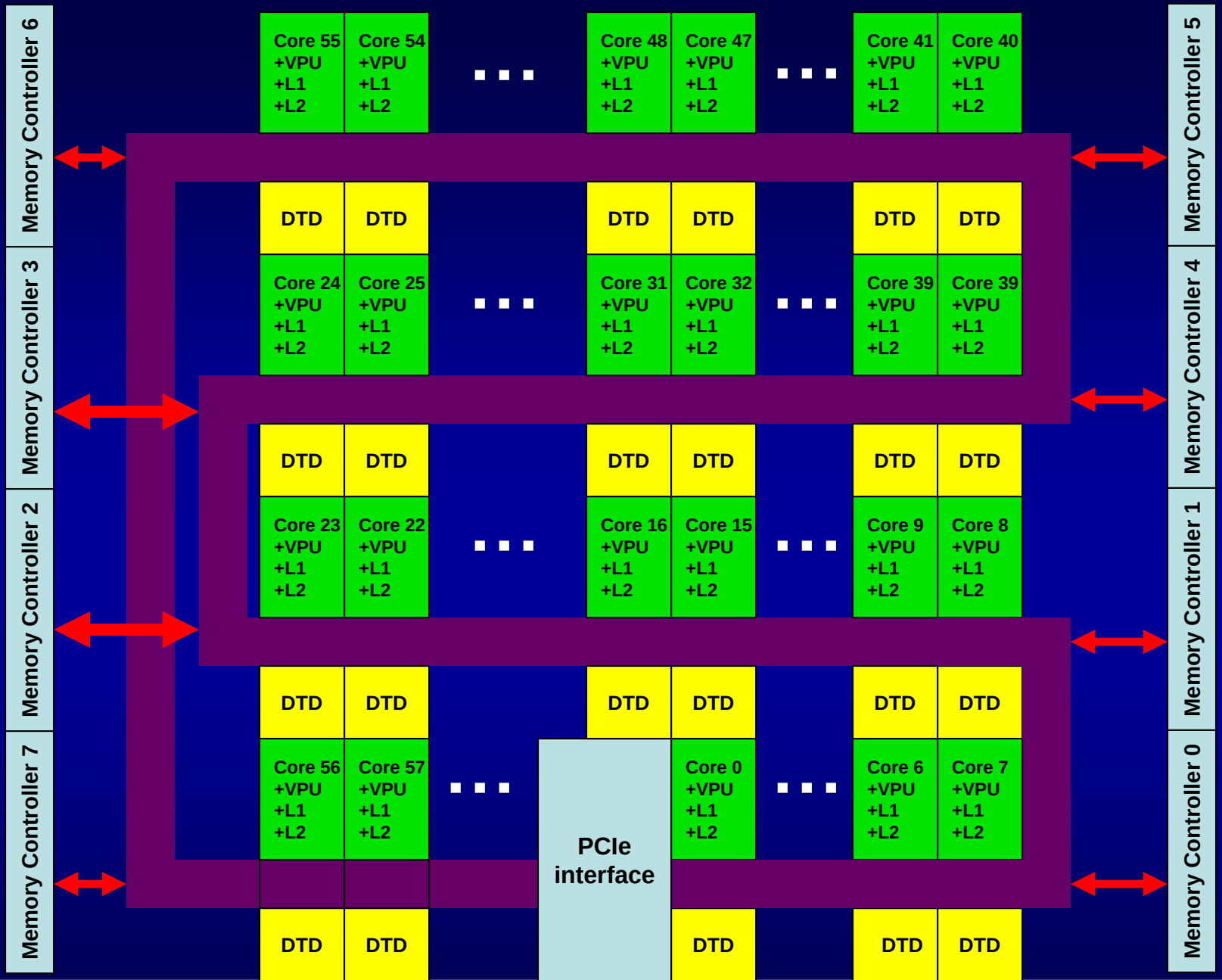
← New on Xeon Phi: like “scatter”, but adjacent threads are placed on adjacent logical processors

KMP_AFFINITY quirks

- System software tries to run on core #60 (i.e., the 61st core)
  - With KMP_AFFINITY=compact, no OpenMP threads will be placed on core #60 until you request more than 240 threads
  - With KMP_AFFINITY=scatter or balanced, the system will start putting threads on core #60 as soon as you request 61 threads.
- Intel added a new feature KMP_PLACE_THREADS to overcome this problem
  - KMP_PLACE_THREADS defines the set of physical and logical processors that an OpenMP program can use
  - KMP_AFFINITY defines the distribution of the OpenMP threads across that set of allowed processors
- KMP_PLACE_THREADS is not required, but it is easier than generating an explicit PROCLIST for KMP_AFFINITY

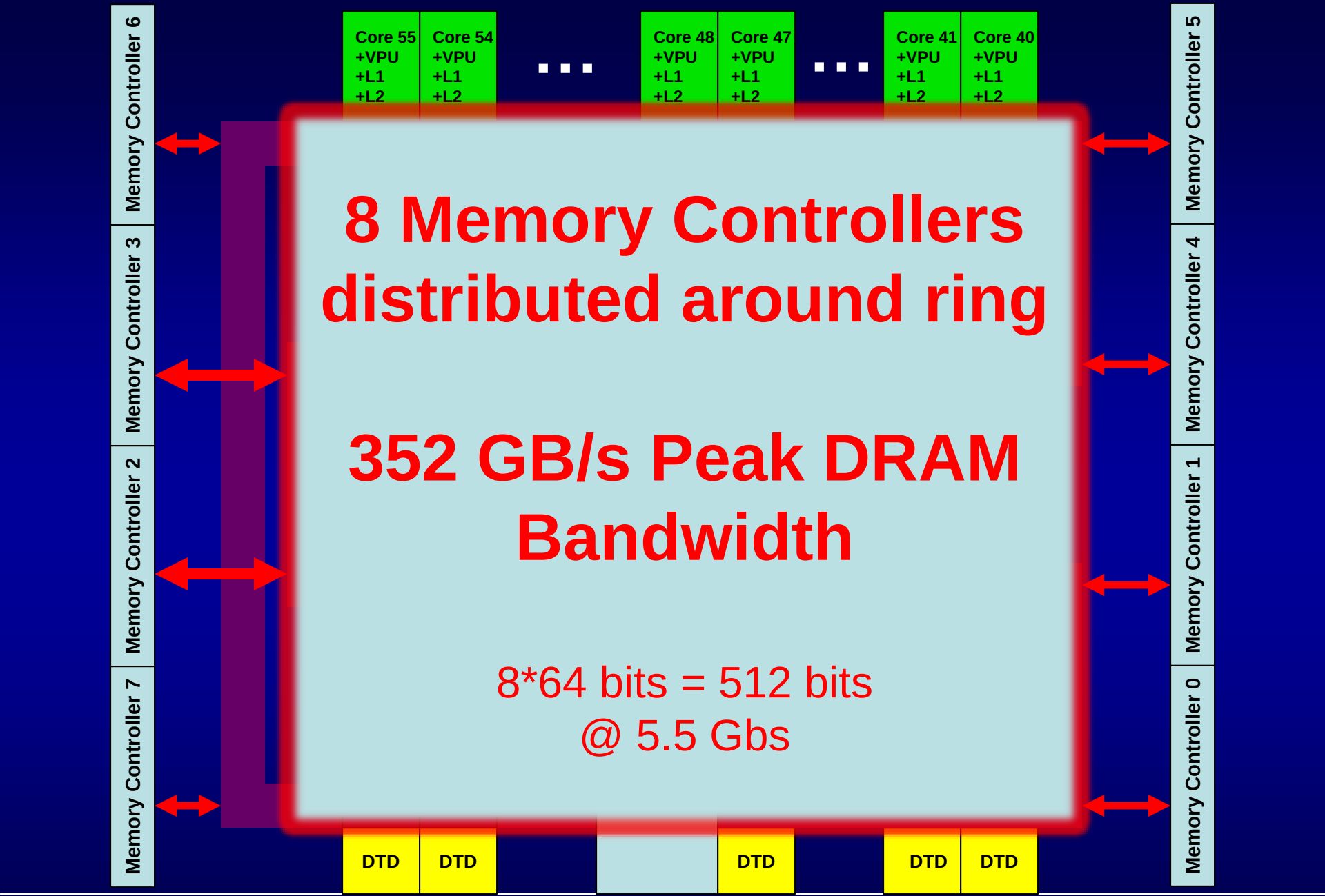
OpenMP thread binding

- First: Define available logical processors
  - `KMP_PLACE_THREADS=iC,jT,kO`
    - i defines the number of physical cores desired
    - j defines the number of threads per physical core
    - k defines the starting physical core number (optional: default=0)
  - E.g., `export KMP_PLACE_THREADS=60C,3T`
    - Requests 60 cores with 3 threads per core, starting with core 0
    - If you define OMP_NUM_THREADS, it should match this value of 180 threads
- Second: Define mapping of threads to processors
  - `KMP_AFFINITY={compact,scatter,balanced}`
  - Compact and Balanced will be the same in this case
    - Adjacent threads on adjacent logical processors
  - Scatter will be round-robin
  - E.g., `export KMP_AFFINITY=compact,verbose`
    - `"verbose"` is your friend! It explains all the numbering and binding



Xeon Phi chip layout

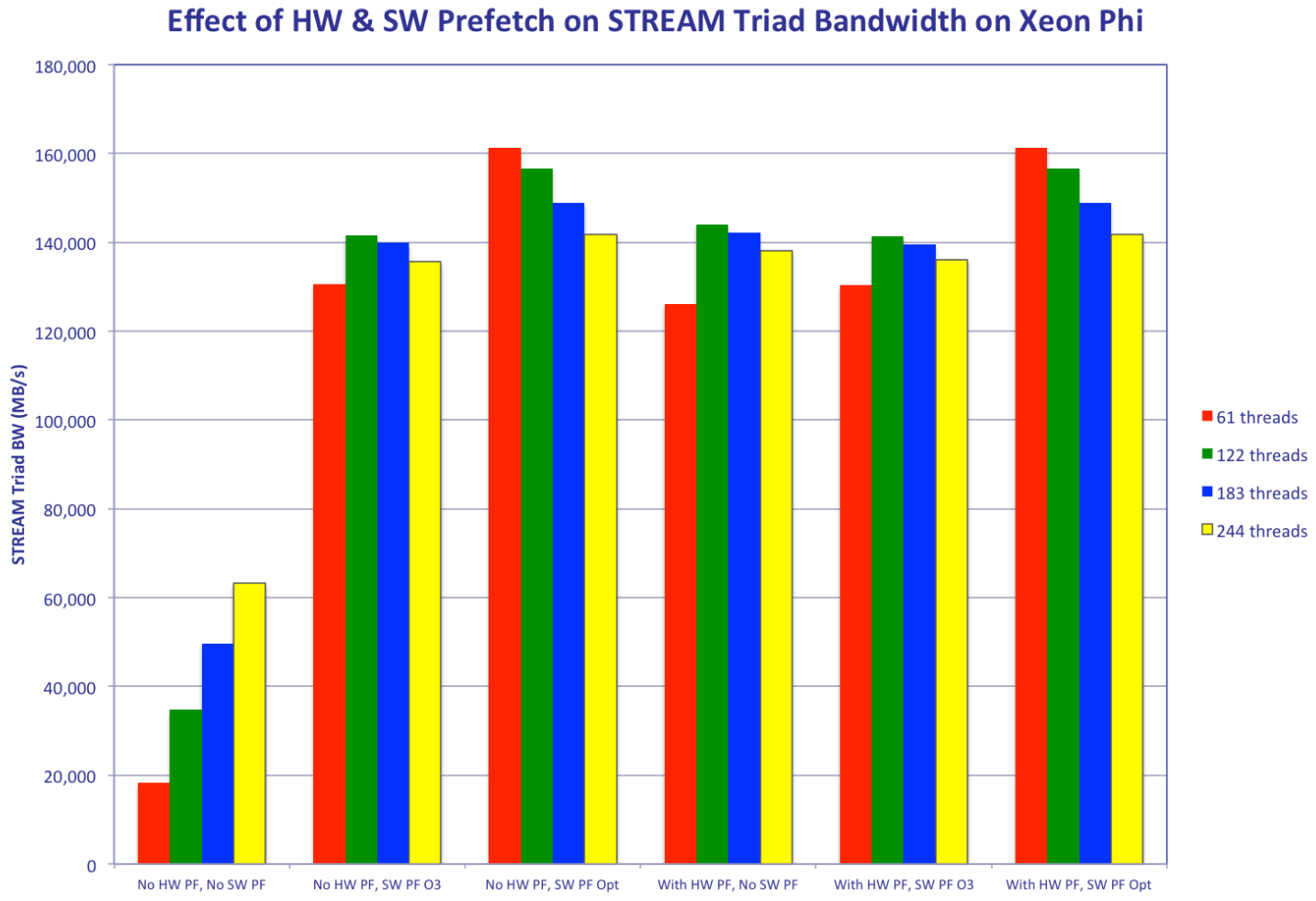
Core and memory controller numbering is John McCalpin's interpretation, not based on Intel disclosures



Xeon Phi chip layout

Core and memory controller numbering is John McCalpin's interpretation, not based on Intel disclosures

Prefetch and Memory Bandwidth



Cache Hierarchy (2)

- Compared to Xeon E5 (“Sandy Bridge”)
  - Xeon Phi L1 is the same size & associativity, but shared by up to 4 threads
  - Xeon Phi L2 is twice as large, same associativity, but with higher latency and shared by up to 4 threads
  - Xeon Phi has No L3

- Note about how the L2 caches work together
  - L2 caches can hold different data or the same data
    - No restrictions – same as on any other microprocessor with private L2
  - But: Modified or Clean data is retrieved from other L2’s instead of Memory
    - Most processors only retrieve modified data from other L2 or L3 caches
    - This reduces memory traffic without needing a shared L3 cache

Additional Cache Bandwidth Notes

- Some Complexities
  - L1-contained data
    - Peak BW is 1 cache line per cycle, so 2 threads/core are needed (each issuing one load every other cycle)
  - L2-contained data
    - 1 thread can get maximum sustainable L2 cache bandwidth (using SW prefetches), but only if the loads from the L1 cache are aligned
    - E.g., 8 loads from L2 in 24 cycles
      - Requires 8 Vector Load instructions and 8 SW Prefetch instructions (vprefetch0)
      - These can dual-issue, so only require 8 of the 12 issue cycles available to a single thread
    - But if L1 loads are not aligned
      - Requires 16 Vector Load instructions (8 vloadunpacklo + 8 vloadunpackhi) and 8 SW prefetch instructions
      - Requires 16 issue slots in 24 cycles, which means that at least 2 threads need to be running
- Performance of Stores is less well understood

Arithmetic Differences

- Xeon Phi has some small differences in arithmetic relative to other modern Intel processors
  - Fused Multiply-Add without intermediate rounding
    - More accurate than separate Multiply and Add from SSE and AVX, but different
  - No hardware divide or square root instructions
    - Both are emulated by iterative functions, typically with lower accuracy (4 ULP)
  - Flush denorms to zero is default at all optimization levels except –O0
  - Exceptions are very different – no traps so code must check flags
  - Requesting higher accuracy (e.g., -fp-model strict) *disables vectorization* in favor of x87 instructions – very slow.

- Learn about how to control arithmetic on Xeon Phi
  - http://software.intel.com/sites/default/files/article/326703/floating-point-differences-sept11_0.pdf

Cache Coherence

- Remote L2 caches are not *directly* snooped on cache misses
  - Content is tracked by 64 "Tag Directories" distributed around ring
- Procedure:
  1. Miss in private L1/L2 caches
  2. Send load or store request to Tag Directory owning that address
  3. Tag Directory requests data from either another cache (if available) or from the Memory Controller owning that address
- This eliminates load from DRAM on shared data accesses
- But adding the tag lookup makes Cache-to-Cache transfers slower
  - About 275ns, independent of relative core numbers – see next slide →

Xeon Phi Tag Directories

- Each cache line address is mapped to one of 64 “Tag Directories” that holds info about which core(s) have copies of a line.
- Cache to cache transfers average about one full loop around the ring, even if the cores are adjacent (since the tags are distributed).