Intel® Xeon Phi™ Coprocessor Optimization and Tuning
Legal Disclaimer

- INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL® PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

- Intel may make changes to specifications and product descriptions at any time, without notice.

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

- Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

- Sandy Bridge and other code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user

- 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 http://www.intel.com/performance

- Intel, Core, Xeon, VTune, Cilk, Intel and Intel Sponsors of Tomorrow. and Intel Sponsors of Tomorrow. logo, and the Intel logo are trademarks of Intel Corporation in the United States and other countries.

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

- Copyright ©2013 Intel Corporation.

- Hyper-Threading Technology: Requires an Intel® HT Technology enabled system, check with your PC manufacturer. Performance will vary depending on the specific hardware and software used. Not available on all Intel® Core™ processors. For more information including details on which processors support HT Technology, visit http://www.intel.com/info/hyperthreading

- Intel® 64 architecture: Requires a system with a 64-bit enabled processor, chipset, BIOS and software. Performance will vary depending on the specific hardware and software you use. Consult your PC manufacturer for more information. For more information, visit http://www.intel.com/info/em64t

- Intel® Turbo Boost Technology: Requires a system with Intel® Turbo Boost Technology capability. Consult your PC manufacturer. Performance varies depending on hardware, software and system configuration. For more information, visit http://www.intel.com/technology/turboboost

Intel® Many Integrated Core Architecture
Software & Services Group, Developer Relations Division
Copyright© 2013, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.
Objective of Module

• Provide optimization techniques useful for the Intel® Xeon Phi™ coprocessor and highlight specific coprocessor recommendations.

Intel Xeon Phi coprocessor specific recommendations are highlighted.
Optimization and Tuning Topics

• The Intel® Compiler’s Vectorizer
• Intel® Xeon Phi™ Coprocessor General Performance Guidelines
• Threading Optimization
• Vectorization Optimization
• Memory Optimization
• Other Optimizations
Module Outline

• The Intel® Compiler’s Vectorizer Reports
  • Intel® Xeon Phi™ Coprocessor General Performance Guidelines
  • Threading Optimization
  • Vectorization Optimization
  • Memory Optimization
  • Other Optimizations
Example: The Compiler’s Vectorizer

• Restructuring to facilitate compiler vectorization
• Getting compiler vectorization reports
• Using compiler guidance suggestions
• Understanding the compiler report
Using Vector Reports to Learn What Code Vectorized - Vectorization Hints and Notification

- **Compiler vectorization report**: `-vec-report[n]`
  - Tells you what code was vectorized
    - If you rely on compiler auto-vectorization, always turn on the vectorization report

```
$ icc example.c -mmic -O3 -vec-report2 ...
```

- **Vectorization guidance**: `-guide-vec[=n]`
  - Can suggest code transformations that may cause more code to vectorize

- **Additional guidance**: `-guide`
  - Gives advice on a variety of optimizations that can improve code optimization

- **Compiler optimization report**: `-opt-report [n]`
  - Informs you how the compiler transforms your code while attempting to generate an optimal version
  - Good for answering questions like “why did the compiler say that?”
Vectorization Reports - Error Example

```c
#include <stdio.h>

void mmul(double *a, int lda, double *b, int ldb, double *c, int ldc, int n) {
    /* &a(i,j) = a + lda * j + i */
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                c[j * ldc + i] += a[k * lda + i] * b[j * ldb + k];
}

$ icc -mmic -vec-report3 serialmmul.cc
serialmmul.cc(7): (col. 10) remark: loop was not vectorized: existence of vector dependence.
serialmmul.cc(8): (col. 14) remark: vector dependence: assumed FLOW dependence between c line 8 and b line 8.
serialmmul.cc(8): (col. 14) remark: vector dependence: assumed ANTI dependence between b line 8 and c line 8.
serialmmul.cc(6): (col. 7) remark: loop was not vectorized: not inner loop.
```

Problem: all iterations of inner loop (k) modify the same element of c, and there is no guarantee that a, b, and c do not point to the same memory – classic data race
Vectorization Reports - Loop Reordering by Hand

```c
#include <stdio.h>
void mmul(double *a, int lda, double *b, int ldb, double *c, int ldc, int n) {
    /* &a(i,j) = a + lda * j + i */
    for (int j = 0; j < n; ++j)
        for (int k = 0; k < n; ++k)
            for (int i = 0; i < n; ++i)  // moved to inside loop from outside
            c[j * ldc + i] += a[k * lda + i] * b[j * ldb + k];
}
```

Each iteration of inner loop (i) modifies a different element of c, and they are far apart. This implementation looked promising enough for the compiler to go ahead and perform run-time data race checks between c[] and a[]/b[] ("multiversioned" – one version for when c[] overlaps with a[]&b[], one when it does not).
Vectorization Reports - Getting Advice From -guide-vec

$ icc -mmic -guide-vec=4 serialmmul.cc
GAP REPORT LOG OPENED ON Wed Mar 30 13:58:35 2011

remark #30761: Add -parallel option if you want the compiler to generate recommendations for improving auto-parallelization.

serialmmul.cc(7): remark #30536: (LOOP) Add -fargument-noalias option for better type-based disambiguation analysis by the compiler, if appropriate (the option will apply for the entire compilation). This will improve optimizations such as vectorization for the loop at line 7. [VERIFY] Make sure that the semantics of this option is obeyed for the entire compilation. [ALTERNATIVE] Another way to get the same effect is to add the "restrict" keyword to each pointer-typed formal parameter of the routine "mmul". This allows optimizations such as vectorization to be applied to the loop at line 7. [VERIFY] Make sure that semantics of the "restrict" pointer qualifier is satisfied: in the routine, all data accessed through the pointer must not be accessed through any other pointer.

Number of advice-messages emitted for this compilation session: 1.

END OF GAP REPORT LOG

$
Vectorization Reports - Success by Using Advice From -guide-vec

```
#include <stdio.h>
void mmul(double * restrict a, int lda, double * restrict b, int ldb,
          double * restrict c, int ldc, int n)
{
    /* &a(i,j) = a + lda * j + i */
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                c[j * ldc + i] += a[k * lda + i] * b[j * ldb + k];
}
$ icc -mmic -restrict -vec-report3 restructmmul.cc
restructmmul.cc(5): (col. 4) remark: PERMUTED LOOP WAS VECTORIZED.
restructmmul.cc(7): (col. 10) remark: loop was not vectorized: not inner loop.
restructmmul.cc(6): (col. 7) remark: loop was not vectorized: not inner loop.
```

Compiler, realizing that a, b, and c point to different memory, decides it can safely reorder the loops in order to vectorize. Because of restrict, the compiler no longer emits a multiversioned loop. This helps lower code size and eliminates the overhead of run-time data race check.
Vectorization Reports – Using `-opt-report` to Understanding What the Compiler Did

```bash
$ icc -mmic -restrict -opt-report restructmmul.cc

HLO REPORT LOG OPENED ON WWW MMM DD HH:MM:SS NNNN

<restrict.cpp;-1:-1;hlo;_Z4mmulPfiS_iS_ii;0>
High Level Optimizer Report (_Z4mmulPfiS_iS_ii)
Total # of lines prefetched in _Z4mmulPfiS_iS_ii for loop at line
5=8
Total # of lines prefetched in _Z4mmulPfiS_iS_ii for loop at line
5=8

<restrict.cpp;5:5;hlo_linear_trans;_Z4mmulPfiS_iS_ii;0>
LOOP INTERCHANGE in loops at line: 5 6 7
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )
```

Here we see the compiler explain how it reordered the loops.
Module Outline

- The Intel® Compiler’s Vectorizer
- **Intel® Xeon Phi™ Coprocessor General Performance Guidelines**
  - Threading Optimization
  - Vectorization Optimization
  - Memory Optimization
  - Other Optimizations
General Performance Guidelines

• Three things you must do to attain maximum performance on the Intel® Xeon Phi™ coprocessor
  1. Optimize for memory access
     o Minimize gathers and scatters by converting arrays of structures (AOS) to structures of arrays (SOA)
     o “Block” algorithms to maximize time data spends in cache
  2. Exploit thread and task parallelism
  3. Optimize for SIMD
     o Choose SIMD-friendly algorithms
       ▪ Remove back-to-back dependencies between loop iterations
       ▪ Minimize scatter/gather
       ▪ Minimize branch misprediction
       ▪ etc.
     o Vectorize inner loops with #pragma simd or the like
     o Vectorize outer loops using Intel® Cilk™ Plus array notation

• Balance performance and accuracy
  – Ex: use fast sqrt, rsqrt, sin, exp, etc. when possible
Other Optimization Suggestions

• See the Intel® Xeon Phi™ coprocessor optimization and programming guides under the “Programming” tab of

  http://software.intel.com/mic-developer
Module Outline

- The Intel® Compiler’s Vectorizer
- Intel® Xeon Phi™ Coprocessor General Performance Guidelines

**Threading Optimization**
- Vectorization Optimization
- Memory Optimization
- Other Optimizations
Threading

• Degree of parallelism
  – Assure fraction of the app that’s parallel is very high
  – Assure degree of thread parallelism is adequate
  – Check for serialization, e.g., locks

• OpenMP* overheads
  – Look at VTune™ Amplifier time line for load balance
  – Look at VTune Amplifier hot spots for overhead time in libiomp
Parallelization

• Use an architecturally independent threading model
  – OpenMP®, Intel® Threaded Building Blocks, Intel® Cilk™ Plus

• Avoid “forced” sequential code as much as possible
  – “Single-threaded” code
  – Atomic operations (e.g., #pragma omp atomic)
  – Locking operations (e.g., #pragma omp critical)
  – Barriers (e.g., #pragma omp barrier)

• Fuse parallel loops where possible
  – Manually by merging loop bodies
  – Conceptually by using “nowait” for OpenMP worksharing constructs
Affinity & Load Balancing – Controlling Threads per Core Using OpenMP* (1 of 3)

- It can be important to tell OpenMP* how you want threads distributed in the system using KMP_AFFINITY (default = none)
  - Especially when OMP_NUM_THREADS is less than the number of available threads
  - KMP_AFFINITY="compact" assigns the OpenMP thread <n>+1 to a free thread context as close as possible to the thread context where the <n> OpenMP thread was placed
    - Good when data locality matters
    - Can result in a load imbalance if not used carefully
  - KMP_AFFINITY="scatter" distributes the threads as evenly as possible across the entire system
    - Good when you want to make maximum use of system resources
Affinity & Load Balancing – Controlling Threads per Core Using OpenMP* (2 of 3)

• It can be important to tell OpenMP* how you want threads distributed in the system using KMP_AFFINITY (default = none)
  – Additional tuning is possible using granularity=
  – You can also explicitly assign threads to specific thread contexts (proclist)
  – But be careful – setting affinity or thread count incorrectly can negatively affect performance

• Examples:
  export MIC_KMP_AFFINITY=scatter
  export MIC_KMP_AFFINITY=explicit,proclist=[7,17,19],verbose
Affinity & Load Balancing – Controlling Threads per Core Using OpenMP* (3 of 3)

<table>
<thead>
<tr>
<th>(u)ser threads</th>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical 0</td>
<td>u0</td>
<td>u3</td>
<td>u5</td>
</tr>
<tr>
<td>Logical 1</td>
<td>u1</td>
<td>u4</td>
<td>u6</td>
</tr>
<tr>
<td>Logical 2</td>
<td>u2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: 7 user threads over a 3 core coprocessor

- **KMP_AFFINITY=“balanced”**
  - With “scatter”, maintains as much locality as possible. Distributes threads but keeps adjacent thread numbers next to each other
    - Provides locality between adjacent threads that can be exploited
  - Essentially “scatter” when # threads <= # available cores
  - Differs when # of threads > # available cores

- For detailed information on controlling affinity in OpenMP*, see “Thread Affinity Interface (Linux* and Windows*)” in the Intel® Compiler XE 12.1 User and Reference Guides
Thread Optimization

• Correct affinity settings are key on the Intel® Xeon Phi™ coprocessor
  – 32 registers of 512 bits make up 2 KB of register file to swap in and out on context switches
  – You want to keep your threads on the same (logical) core!

• Tweak number of threads and thread affinity
  – Find the best value of OMP_NUM_THREADS and KMP_AFFINITY
  – Try 2-3 threads per core
  – Try KMP_AFFINITY=balanced,granularity=fine
Thread Optimization

• Use one MPI rank per KNC core, OpenMP* within core
  – OpenMP synchronization is faster within a core than across cores

• OMP_NUM_THREADS
  – Balance MPI and OMP thread parallelism for target

• #pragma omp for collapse (n)
  – Increase thread-parallelism for utilization, load balance

• KMP_AFFINITY
  – Use balanced to avoid OS collision & avoid migration
Affinity & Load Balancing – How Many Threads per Core?

- While KNC allows 4 threads per core, good reasons to use fewer are:
  - Minimize cache/TLB thrashing when too many threads compete for L1 or L2
  - Lower competition for the core’s single vector unit
  - Minimize requests to main memory by lowering total thread count

- Good reasons to have all 4 threads in use are:
  - To take advantage of data locality between threads and data fits in cache
  - To keep per-core thread working set small enough

- What is best depends...
  - 1 thread/core for workloads with high DRAM bandwidth requirements
  - 4 threads/core for many of the GEMM workloads
  - 3 threads/core is the sweet spot for most assembly code we wrote
  - >=2 threads/core gives you >90% of instruction issue bandwidth

- Best case, all threads on a core should collaborate
- OpenMP* will avoid core 0 (OS & services) for performance reasons
- This topic is vastly more complicated than we can discuss in one slide
Module Outline

• The Intel® Compiler’s Vectorizer
• Intel® MIC Architecture General Performance Guidelines
• Threading Optimization

• Vectorization Optimization
• Memory Optimization
• Other Optimizations
Transformations – Enable Parallelism

• Loop interchange
  – Better memory locality raises instruction- and pipeline-
    parallelism

• Enable vectorization
  – Simplify reductions, especially when they appear inside a conditional
    o Turn accumulator into a temp that's declared outside of the loop
    o Accumulate into that temp in the loop
    o Add that temp to the real accumulator outside the loop
  – Avoid constructors in a loop, by extending scope of stack variables to outside a loop, and converting return values to structs
Other Optimization Suggestions

• Make sure vectorized code does array accesses to contiguous memory elements
  – Improves data locality and minimizes masking
Vectorization Optimization

• Unleash the vectorizer by providing context information
  - :#pragma ivdep, #pragma vector always
  - Intel® Cilk™ Plus vectorization pragmas (#pragma simd)
  - Avoid aliasing and let the compiler know it
    o “restrict” keyword
    o -ansi-alias
    o -fno-alias
    o Use Fortran 😊
Sample: **simd Pragmas**

```c
float sprod(float *a, float *b, int n)
{
    float sum = 0.0f;
    for (int k=0; k<n; k++)
        sum += a[k] * b[k];
    return sum;
}

void sprod(float *a, float *b, int n)
{
    float sum = 0.0f;
    #pragma simd vectorlength(16) reduction(+:sum)
    for (int k=0; k<n; k++)
        sum += a[k] * b[k];
    return sum;
}
```
 Improving the Odds That Code Will Vectorize – 1 of 4

• Remove conditionals and function calls from inner loops
• Increase the odds of vectorization if you need function calls in inner loops by:
  – Using #pragma simd, SIMD enabled functions marked with __attribute__((vector)), or Intel® Cilk™ Plus Array Notation
• If you compile with -g, be sure to specify the optimization level (-Ox)
  – When doing debugging or doing performance analysis on optimized code, use “-g”
  – Otherwise, the default optimization level switches from -O2 to -O0d and no vectorization occurs
  – Example: icc -O2 -g -vec-report3 -offload-build foo.c

Note: Some outer loops will vectorize in 12.1 and beyond
Improving the Odds That Code Will Vectorize—2 of 4

• For best performance, use signed integers as loop counters
  – Not unsigned integers or floats
• For best performance, branches that do conditional assignments should use a loop-local variable
  – Then store the value of the local to the desired variable outside the branch
• Avoid unintended type conversions where possible
  – e.g., use: `float x; x = sin(x + 1.0) + 1.0`
• Vectorize inner loops and collapse outer loops to allow more work for parallelism
• When porting code from Fortran, make sure to account for the switch from column-major to row-major array layout (swap inner and outer loop indices when traversing array)
• File a bug if you see vectorization differences between Intel® Xeon Phi™ coprocessor and Intel® Xeon® processor
Vectorization and Compute-bound Limits

- Deep dive on reasons why hot loops don’t vectorize profitably
  - Look at messages, patterns, idioms

- Compare against compute-bound limit

- Check assembly code
  - Compare path length and generated code with expectations
  - Assure vectorization by checking for vector instructions

- Check degree of vectorization with PMU data
  - % vector instructions: $\frac{\text{VPU\_INSTRUCTIONS\_EXECUTED}}{\text{INSTRUCTIONS\_EXECUTED}}$
  - Avg. % elements used per vector: $\frac{\text{VPU\_ELEMENTS\_ACTIVE}}{\text{INSTRUCTIONS\_EXECUTED}}$
Loop Optimizations

• If possible use a signed 32-bit integer

• int64 operations in the vector unit are (at best) half the bandwidth of int32 operations in vector unit!

• Use –vec-report1 through –vec-report3 to figure out what does not vectorize and why
  – But only in hot code
Improving the Odds That Code Will Vectorize—3 of 4

• Loop trip counts
  – Improves quality of compiler optimizations such as prefetching, vectorization
  – Can use `#pragma loop_count(n), or #pragma loop_count min(n), max(n), avg(n)`
  – Loop profiling not available on the Intel® Xeon Phi™ coprocessor

• Help the compiler with loop trip counts
  – If you see “low trip count” in `-vec-report`, help the compiler by using `#pragma loop_count`
  – Make sure loops operating on doubles have at least 8 iterations, operations on floats have at least 16 iterations

• For best performance, try marking any functions called by the loop you are trying to vectorize with combinations of
  – `#pragma inline, forceinline, or noinline`
  – `__attribute__((vector))`
Improving the Odds That Code Will Vectorize – 4 of 4

• Avoid use/construction of int64 vectors if possible
  – int64 types on Intel® MIC Architecture: size_t, pointer, long
  – Performance: int64 does not mix well with int32
  – It is much harder to vectorize
    o Avoid arithmetic, loop counters, array indexing, scatter/gather, etc. with int64 types
    o “unsupported data type” or “dereference/subscript too complex” result
  – Avoid converting between int64 and float
    o Another cause of “unsupported data type” -vec-report output

• Limited 8-bit and 16-bit data support
  – Look for “unsupported data type” in -vec-report
  – Try manually converting to/from int to prevent the compiler from attempting automatic integer type conversions
Avoiding set conflicts

• Background
  – Level 1 cache holds 8 instances of 64-B cache lines that are a multiple of 32 KB/8=4 KB away
  – Level 2 cache holds 8 instances of 64-B cache lines that are a multiple of 512 KB/8=64 KB away
  – Because LRU is imperfect, it may act like it has <8 sets

• Recipe for identifying a bottleneck
  – High cache miss rate, even though working set < cache capacity
  – References or (array dimensions*element size) are a multiple of 4 K (L1) or 64 K (L2) apart

• Remedy
  – Pad array dimensions
  – Move the base of a subset of the references
Module Outline

• The Intel® Compiler’s Vectorizer
• Intel® Xeon Phi™ Coprocessor General Performance Guidelines
• Threading Optimization
• Vectorization Optimization
• **Memory Optimization**
• Other Optimizations
Ensuring Memory Alignment

• Memory alignment is an essential part of high-speed vector processing

• Unaligned memory will incur performance penalties

• Align the data AND tell the compiler
  – In most cases, static compiler does not have the alignment information of references inside loops, so does extra work to cover misalignment
  – Align the data using alignment attributes, using \_mm\_malloc, using Fortran option -align array64byte, etc.
  – Tell compiler about alignment using a clause before the vector-loop
    o assume\_aligned clause, vector aligned pragma, etc.
Memory Optimizations - Alignment

- Allocating aligned memory
  - Static memory
    - Allocated by compiler/linker
    - Add `__attribute__((aligned(n)))` in front of variable declaration
    - Applies to global/local static variables as well as stack/auto variables
  - Dynamic memory
    - Allocated by language runtime
    - Use `__mm_aligned_malloc(size, alignment_bytes)`
    - Pair it with `__mm_aligned_free()`

- Using Intel® Threading Building Blocks
  - Dynamic memory allocation API that supports the Intel® MIC Architecture
  - Use `scalable_aligned_malloc()`/`scalable_aligned_free()`
  - Include `<tbb/scalable_allocator.h>` and link with `-ltbbmalloc`
Memory Optimizations - Alignment

• Mechanisms

```cpp
__declspec(align(64)) float array[SIZE];
#pragma vector aligned
__assume_aligned(p1, 64);
__assume(n1%16==0);
void *__offload_myoSharedAlignedMalloc(size_t size, size_t alignment);
#pragma offload target(mic) align(64)
```
Memory Optimizations - Huge Pages and Pre-faulting

• Intel® Architecture-based processors support multiple page sizes; commonly 4 K and 2 MB

• *Some* applications will benefit from using huge pages
  – Applications with sequential access patterns will improve due to larger TLB “reach”

• TLB miss vs. Cache miss
  – TLB miss means walking the 4 level page table hierarchy
    ▪ Each page walk could result in additional cache misses
  – TLB is a scarce resource, so “manage” it well

• On Intel® Xeon Phi™ coprocessor
  – 64 entries for 4 K, 8 entries for 2 MB
  – Additionally, 64 entries for second level DTLB
    ▪ Page cache for 4 K, L2 TLB for 2 MB pages

• Linux* supports huge pages – CONFIG_HUGETLBFS
  – 2.6.38 also has support for Transparent Huge Pages (THP)

• Pre-faulting via MAP_POPULATE flag to mmap()
Memory Optimizations - Huge Pages and Pre-faulting

• Page sizes
  – Use libhugetlbfs to force use of 2 M pages for non-offloaded data

  – Use mmap to selectively control size

  – Use the environment variable MIC_USE_2MB_BUFFERS to force runtime to allocate offloaded data into 2 MB pages
Memory Optimizations - Eviction Control

- Streaming data trashes cache, doesn’t need residency
  - Mark with #pragma vector nontemporal
  - clevict can be used to evict cache lines sooner and at a higher rate than HW can
  - Intel® Xeon® processor: MOVNTQ

- `-mGLOB_default_function_attrs="clevict_level=N"`
  where N = 0, 1, 2 or 3 (default is 3 on Intel® Xeon Phi™ coprocessor)
  0 - do not generate clevict
  1 - generate clevict0, from L1
  2 - generate clevict1, from L2
  3 - generate L1 and L2 clevict
Eviction Control

- Streaming data trashes cache, doesn’t need residency
  - KNC: clevict0, clevict1
Memory Optimizations

- Check assembly for gathers/scatters, change data structures or code to avoid them
- Compare against bandwidth limit
- Check L2 and L1 cache miss ratios. Loop interchange, tile and change data structures as necessary to increase locality.
- Check & tune prefetching, particularly for gathers and scatters
- Determine what the best page size is: 4 K or 2 M. Use libhugetlbfs or mmap if appropriate. Check TLB miss rates against expected access patterns.
Memory Optimizations

• Pick data pitches to minimize number of cache lines accessed
  – In both L1 and L2
  – Also pay attention to how many memory addresses alias to the same cache line – working on multiples will thrash the cache

• Each output cache line should be written to by only one core.

• Latency to main memory can rise >3X when GDDR is heavily burdened
  – Thereby lowering per-core maximum bandwidth
Memory Optimizations - Gather / Scatter

• Avoid scatter/gather where possible
  – Array of Structures to Structure of Arrays (AoS → SoA)
  – Special-case code to cover unit stride if it’s a common occurrence
  – But gather/scatter a good thing if used carefully
Other Optimization Suggestions – Performance is *All* About Memory

- Prefetching
  - The Intel® Xeon Phi™ coprocessor contains a L2 cache streaming prefetcher that can selectively prefetch code, read, and RFO (read-for-ownership) cache lies into the L2. Additionally, the compiler generates prefetches automatically.

  - If you have to do it yourself, prefetch to L2 about 1000 cycles in advance of demand usage. Then prefetch to L1 about 50 cycles in advance.

  - (Do NOT prefetch directly to L1!)
Prefetching Control

• By default for –O2 and above
• Tuning distance
  – -mP2OPT_hlo_use_const_pref_dist={n=8,16,32,64 vectorized iterations} (vprefetch1 to L2)
  – -mP2OPT_hlo_use_const_second_pref_dist={n=1,2,4,8 vectorized iterations} (vprefetch0 from L2 to L1)
  – #pragma prefetch var:hint{0=T0,1=NT1}:distance
• Overprefetching?
  – Use -mP2OPT_hlo_pref_issue_second_level_prefetch=F to skip vprefetch0
  – Use -mP2OPT_hlo_pref_issue_first_level_prefetch=F to skip vprefetch1
  – #pragma noprefetch just before a loop or as 1st line of function
Module Outline

• The Intel® Compiler’s Vectorizer
• General Performance Guidelines
• Threading Optimization
• Vectorization Optimization
• Memory Optimization

• Other Optimizations
Inlining Control - Pragmas

• Statement-specific inline pragmas:
  - #pragma inline [recursive] – hint, subject to heuristics
  - #pragma forceinline [recursive] – dictate, whenever possible
  - #pragma noinline – dictate
  - When placed before a C/C++ statement, applies to all calls and statements nested within that statement
  - There are corresponding directives for Fortran
Inlining Controls – Compiler Switches

-\([\text{no-}] \text{inline-factor}=n,\)
  - Specifies % multiplier that should be applied to the following inlining options that define upper limits, i.e., \(n=200\) means multiply upper limits by 2
    - -\([\text{no-}] \text{inline-min-size}=n\)
    - -\([\text{no-}] \text{inline-max-size}=n\)
    - -\text{inline-max-per-routine}=n
    - -\text{inline-max-per-compile}=n

-\text{inline-forceinline}\n  - Specifies that an inlined routine should be inlined whenever the compiler can do so
Intel® Xeon Phi™ coprocessor specific Transformations

- Related to Intel Xeon Phi coprocessor ISA support

- Floating point
  - Use single vs. double precision where possible
  - Use various precision controls where applicable: -imf-*, -[no-]prov-*
  - Rewrite “/const” as “*1/const”

- Signed vs. unsigned 32b ints

- Convert to using 32b vs. 64b ints wherever possible
  - More elements per SIMD vector
  - Enable vectorization for scatter/gather
  - Enable vectorization for type conversion
Optimization and Tuning Summary

• The Intel® Compiler’s Vectorizer
• General Performance Guidelines
• Threading Optimization
• Vectorization Optimization
• Memory Optimization
• Other Optimizations
Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options.” Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools 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 Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® 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.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101
Backup
## Interesting Cache Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Demand Reads</td>
<td>(\text{DATA_READ} - \text{L1_DATA_PF1} - \text{L1_DATA_PF2})</td>
</tr>
<tr>
<td>L1 Demand Misses</td>
<td>(\text{DATA_READ_MISS} - \text{L1_DATA_PF1_MISS} - \text{L2_DATA_PF2_MISS} - \text{L2_DATA_PF1_MISS})</td>
</tr>
<tr>
<td>L1 Miss Ratio</td>
<td>(\frac{\text{L1 Demand Misses}}{\text{L1 Demand Reads}})</td>
</tr>
<tr>
<td>L2 Prefetch Hits</td>
<td>(\text{L2_DATA_PF2} - \text{L2_DATA_PF2_MISS})</td>
</tr>
<tr>
<td>L2 Demand Reads</td>
<td>(\text{L2_READ_HIT_E} + \text{L2_READ_HIT_M} + \text{L2_READ_HIT_S}) - \text{L2 Prefetch Hits})</td>
</tr>
</tbody>
</table>
### Interesting Cache Metrics cont.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 Demand Misses</td>
<td>( L2_READ_MISS - L2_DATA_PF2_MISS )</td>
</tr>
<tr>
<td>L2 Miss Ratio</td>
<td>( \frac{L2 \text{ Demand Misses}}{L2 \text{ Demand Reads}} )</td>
</tr>
<tr>
<td>L1 Cache Hit Ratio</td>
<td>( \frac{(DATA_READ_OR_WRITE - DATA_READ_MISS_OR_WRITE_MISS)}{DATA_READ_OR_WRITE} )</td>
</tr>
<tr>
<td>L2 Cache Hit Ratio</td>
<td>( \frac{(DATA_READ_OR_WRITE - L2_READ_MISS)}{DATA_READ_OR_WRITE} )</td>
</tr>
</tbody>
</table>
## Interesting Prefetcher Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 PF Miss Ratio</td>
<td>$\frac{L1_DATA_PF1_MISS}{L1_DATA_PF1}$</td>
</tr>
<tr>
<td>L2 PF Miss Ratio</td>
<td>$\frac{L2_DATA_PF2_MISS}{L2_DATA_PF2}$</td>
</tr>
<tr>
<td>L1 PF Inflight Ratio</td>
<td>$\frac{L1_DATA_HIT_INFLIGHT_PF1}{L1_DATA_PF1}$</td>
</tr>
<tr>
<td>L2 PF Inflight Ratio</td>
<td>$\frac{L2_DATA_HIT_INFLIGHT_PF2}{L2_DATA_PF2}$</td>
</tr>
<tr>
<td>L1 PF Drop Ratio</td>
<td>$\frac{L1_DATA_PF1_DROP}{L1_DATA_PF1}$</td>
</tr>
<tr>
<td>L2 PF Drop Ratio</td>
<td>$\frac{L2_DATA_PF1_DROP}{L2_DATA_PF2}$</td>
</tr>
</tbody>
</table>