An Overview of Programming for Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors

Introduction

Intel Xeon Phi coprocessors are designed to extend the reach of applications that have demonstrated the ability to fully utilize the scaling capabilities of Intel Xeon processor-based systems and fully exploit available processor vector capabilities or memory bandwidth. For such applications, the Intel Xeon Phi coprocessors offer additional power-efficient scaling, vector support, and local memory bandwidth, while maintaining the programmability and support associated with Intel Xeon processors.

Most applications in the world have not been structured to exploit parallelism. This leaves a wealth of capabilities untapped on nearly every computer system. Such applications can be extended in performance by a highly parallel device only when the application expresses a need for parallelism through parallel programming.

Advice for successful parallel programming can be summarized as "." Since most applications have not yet been structured to take advantage of the full magnitude of parallelism available in any processor, understanding how to restructure to expose more parallelism is critically important to enable the best performance on processors and coprocessors. This restructuring itself will generally yield benefits on most general-purpose computing systems, a bonus due to the emphasis on common programming languages, models, and tools that span these processors and coprocessors. I refer to this bonus as the dual-transforming-tuning advantage.

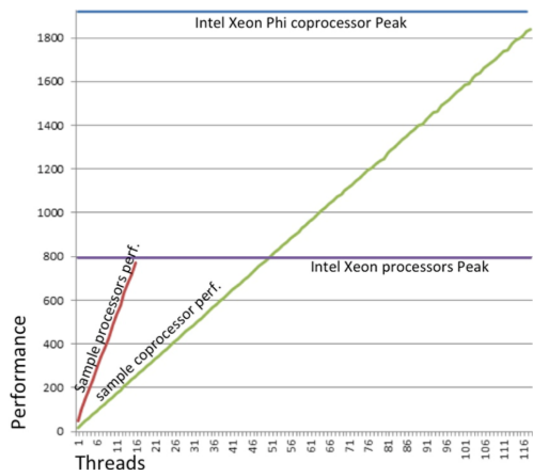

Figure 1: For understanding the motivation, potential and needs of the Intel Xeon Phi Coprocessor, this is the picture that speaks a thousand words. The numerical values are illustrative and cannot represent every application.

It has been said that a single picture can speak a thousand words; for understanding Intel Xeon Phi coprocessors (or any highly parallel device) Figure 1 is that picture. We should not dwell on the exact numbers as they are based on some models that may be as typical as

Copyright © 2012, Intel Corporation. All rights reserved.

applications can be. The picture speaks to this principle: Intel Xeon Phi coprocessors offer the ability to build a system that can potentially offer exceptional performance while still being buildable and power efficient. Intel Xeon processors deliver performance much more readily for a broad range of applications but do reach a practical limit on peak performance as indicated by the end of the line in Figure 1. The key is "ready to use parallelism." Note from the picture that more parallelism is needed to make Intel Xeon Phi coprocessor reach the same performance level, and that requires programming adapted to deliver that higher level of parallelism required. In exchange for the programming investment, we may reach otherwise unobtainable performance. The transforming-and-tuning double advantage of Intel products is that use of the same parallelism model, programming languages and familiar tools to greatly enhance preservation of programming investments. I will revisit this picture later.

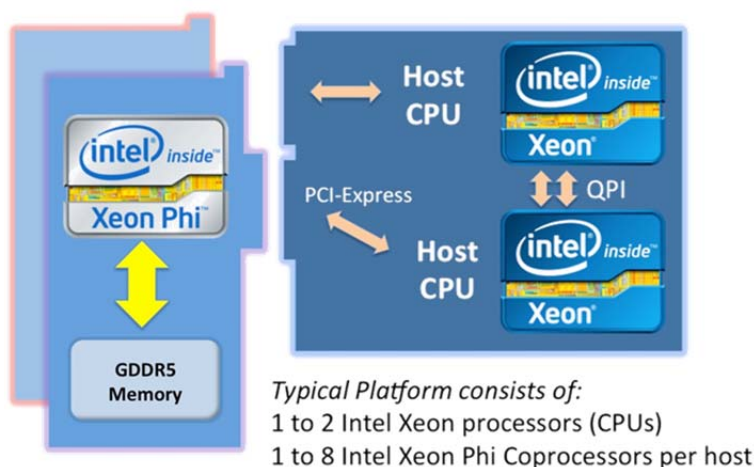

Figure 2: Intel Xeon processors and Intel Xeon Phi Coprocessors in a platform together

A system

A typical platform is shown in Figure 2. Multiple such platforms may be interconnected to form a cluster or supercomputer. A platform cannot consist of only coprocessors. Processors are cache coherent and share access to main memory with other processors. Coprocessors are cache coherent SMP-on-a-chip[1] devices that connect to other devices via the PCIe bus, and are not hardware cache coherent with other processors or coprocessors in the system.

The Intel Xeon Phi coprocessor runs Linux. It really is an x86 SMP-on-a-chip running Linux. Every card has its own IP address. I logged onto one of our pre-production systems in a terminal window. I first got my shell on the host (an Intel Xeon processor), and then I did "ssh mic0" which logged me into the first coprocessor card in the system. Once I had this window, I listed `/proc/cpuinfo`. The result is 6100 lines long, so I'm showing the first 5 and last 26 lines in Figure 3.

[1] SMP: Symmetric Multi-Processor, a multiprocessor system with shared memory and running a single operating system.

Copyright © 2012, Intel Corporation. All rights reserved.

```
root@dpdknf01:/KNC — ssh — 100×35
% cat /proc/cpuinfo | head -5
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 11
model           : 1
model name      : 0b/01
%
% cat /proc/cpuinfo | tail -26

processor       : 243
vendor_id       : GenuineIntel
cpu family      : 11
model           : 1
model name      : 0b/01
stepping        : 1
cpu MHz         : 1090.908
cache size      : 512 KB
physical id     : 0
siblings        : 244
core id         : 60
cpu cores       : 61
apicid          : 243
initial apicid  : 243
fpu             : yes
fpu_exception   : yes
cpuid level     : 4
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr mca pat fxsr ht syscall lm lahf_lm
bogomips        : 2192.10
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:

%
```

Figure 3: Screenshot of an ssh session on a pre-production Intel Xeon Phi Coprocessor with the beginning and end of the 6100 lines of "cat /proc/cpuinfo"

In some ways, for me, this really makes the Intel Xeon Phi coprocessor feel very familiar. From this window, I can "ssh" to the world. I can run "emacs" (you can run "vi" if that is your thing). I can run "awk" scripts or "perl." I can start up an MPI program to run across the cores of this card, or to connect with any other computer in the world.

If you are wondering how many cores are in an Intel Xeon Phi coprocessor, the answer is "it depends." It turns out there are, and will be, a variety of configurations available from Intel all with more than 50 cores. For years, we have been able to buy processors in a variety of clock speeds. More recently, an additional variation in offerings is based on the number of cores. The results in Figure 3 are from a 61-core pre-production Intel Xeon Phi coprocessor that is a precursor to the production parts known as an Intel Xeon Phi coprocessor SE10x. It reports a processor number 243 because the threads are enumerated 0..243 meaning there are 244 threads (61 cores times 4 threads per core).

The First Intel Xeon Phi Coprocessor: code name Knights Corner

While programming does not require deep knowledge of the implementation of the device, it is definitely useful to know some attributes of the coprocessor. From a programming standpoint, treating it as an x86-based SMP-on-a-chip with over 50 cores, with multiple threads per core and 512-bit SIMD instructions, is the key. It is not critical to completely absorb everything else in this part of the paper, including the micro-architectural diagrams in Figure 4 and 5 that I chose to include for those who enjoy such things as I do.

Copyright © 2012, Intel Corporation. All rights reserved.

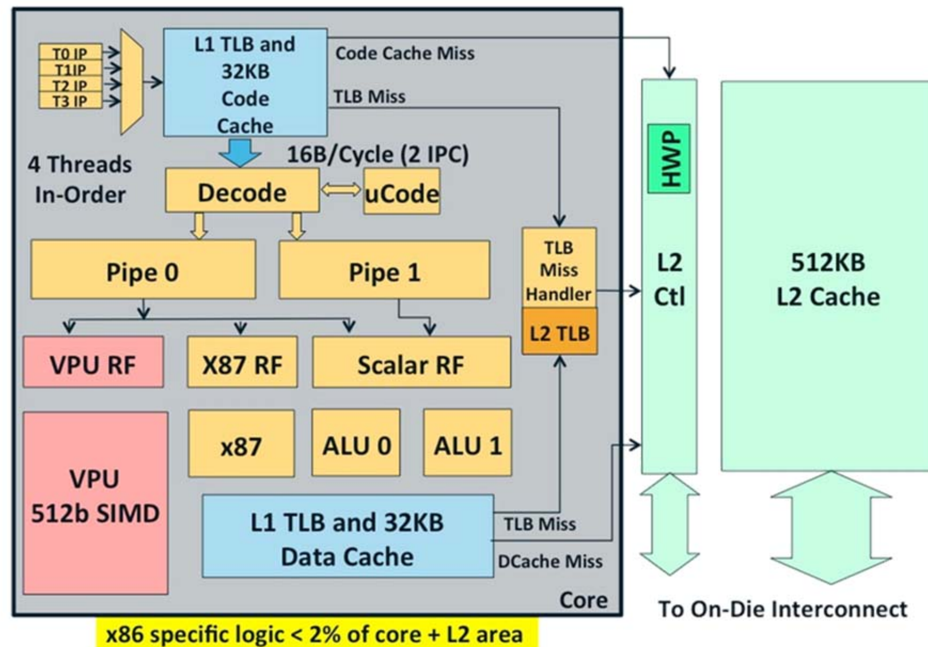

Figure 4: Knights Corner Core

The cores are in-order dual issue x86 processor cores which trace some history to the original Pentium design, but with the addition of 64-bit support, four hardware threads per core, power management, ring interconnect support, 512 bit SIMD capabilities and other enhancements these are hardly the Pentium cores of 20 years ago. The x86-specific logic (excluding L2 caches) makes up less than 2% of the die area for an Intel Xeon Phi coprocessor.

Here are key facts about the first Intel Xeon Phi coprocessor product:

- A coprocessor (requires at least one processor in the system), in production in 2012.
- Runs Linux* (source code available http://intel.com/software/mic).
- Manufactured using Intel's 22nm process technology with 3-D Trigate transistors.
- Supported by standard tools including Intel Cluster Studio XE 2013. A list of additional tools available can be found online (http://intel.com/software/mic).
- Many cores:
    - More than 50 cores (it will vary within a generation of products, and between generations; it is good advice to avoid hard-coding applications to a particular number).
    - In-order cores support 64-bit x86 instructions with uniquely wide SIMD capabilities.
    - Four hardware threads on each core (resulting in more than 200 hardware threads available on a single device) are primarily used to hide latencies implicit in an in-order microarchitecture. In practice, use of at least two threads per core is nearly always beneficial. As such, it is much more important that applications

Copyright © 2012, Intel Corporation. All rights reserved.

use these multiple hardware threads on Intel Xeon Phi coprocessors than they use hyper-threads on Intel Xeon processors.
  - Cores interconnected by a high-speed bidirectional ring.
  - Cores clocked at 1 GHz or more.
  - Cache coherent across the entire coprocessor.
  - Each core has a 512-KB L2 cache locally with high-speed access to all other L2 caches (making the collective L2 cache size over 25 MB).
  - Caches deliver highly-efficient power utilization while offering high bandwidth memory.
- Special instructions in addition to 64 bit x86:
  - Uniquely wide SIMD capability via 512 bit wide vectors instead of the narrower MMX, SSE or AVX capabilities.
  - High performance support for reciprocal, square root, power and exponent operations.
  - Scatter/gather and streaming store capabilities to achieve higher effective memory bandwidth.
- Special features:
  - On package memory controller supports up to 8GB GDDR5 (varies based on part).
  - PCIe connect logic is on-chip.
  - Power management capabilities.
  - Performance monitoring capabilities for tools like Intel® VTune™ Amplifier XE 2013.

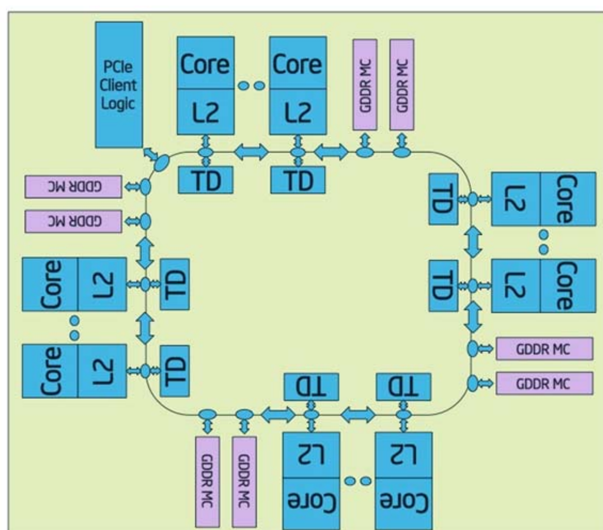

Figure 5: Knights Corner Microarchitecture

When to use an Intel Xeon Phi coprocessor

Applications can use both Intel Xeon processors and Intel Xeon Phi coprocessors to contribute to application performance. Applications should utilize a coprocessor for processing when it

Copyright © 2012, Intel Corporation. All rights reserved.

can contribute to the performance of a node. Generally speaking that will be during the portions of an application that can exploit high degrees of parallelism. For some workloads, the coprocessor(s) may contribute significantly more performance than the processor(s) while on others it may be less. System designs that include Intel Xeon Phi coprocessor(s) extend the range of node performance beyond what is possible with processors only. Because of the sophisticated power management in both Intel Xeon processors and Intel Xeon Phi coprocessors, the power efficiency of a node can be maintained across a broad range of applications by consuming power only when needed to contribute to node performance.

The Importance of Maximizing Performance on Intel Xeon processors first

The single most important lesson from working with Intel Xeon Phi coprocessors is this: the best way to prepare for Intel Xeon Phi coprocessors is to fully exploit the performance that an application can get on Intel Xeon processors first. Trying to use an Intel Xeon Phi coprocessor, without having maximized the use of parallelism on Intel Xeon processor, will almost certainly be a disappointment. Figure 6 illustrates a key point: higher performance comes from pairing parallel software with parallel hardware because it takes parallel applications to access the potential of parallel hardware. Intel Xeon Phi coprocessors offer a corollary to this: higher performance comes from pairing *highly* parallel software with *highly* parallel hardware. The best place to start is to make sure your application is maximizing the capabilities of an Intel Xeon processor.

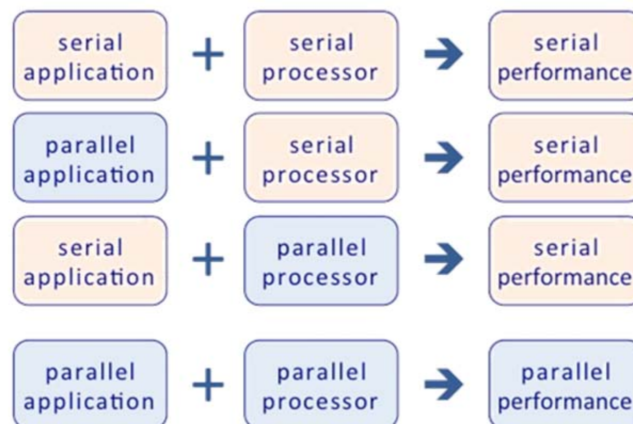

Figure 6: High performance comes from combining parallel software with parallel hardware.

Why scaling past one hundred threads is so important

In getting an application ready for utilizing an Intel Xeon Phi coprocessor, nothing is more important than scaling. An application must scale well past one hundred threads to qualify as highly parallel. Efficient use of vectors and/or memory bandwidth is also essential. Applications that have not been created or modified to utilize high degrees of parallelism (task, threads, vectors, and so on) will be more limited in the benefit they derive from hardware that is designed to offer high degrees of parallelism.

Copyright © 2012, Intel Corporation. All rights reserved.

Figure 7 and 8 show examples of how application types can behave on Intel Xeon processors versus Intel Xeon Phi coprocessors in two key cases: computationally bound and memory bound applications. Note that a logarithmic scale is employed in the graph therefore the performance bars at the bottom represent substantial gains over bars above; results will vary by application. Measuring the current usage of vectors, threads and aggregate bandwidth by an application can help understand where an application stands in being ready for highly parallel hardware. Notice that “more parallel” enhances both the processor and coprocessor performance. A push for “more parallel” applications will benefit Intel Xeon processors and Intel Xeon Phi coprocessors because both are general-purpose programmable devices.

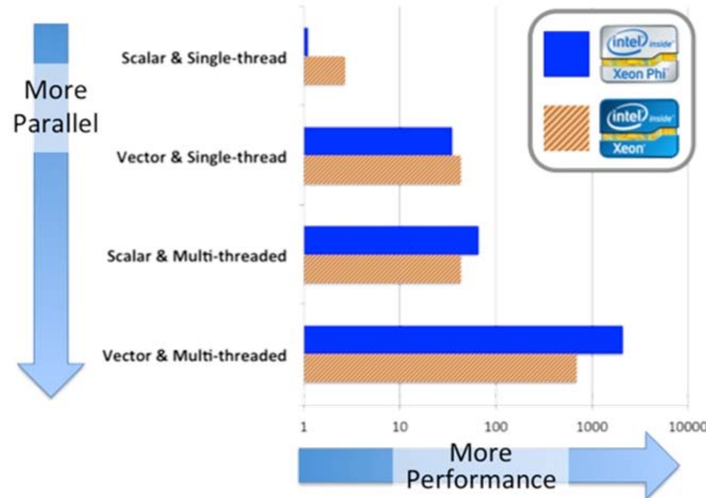

Figure 7: Combining Threads and Vectors works best for Processors and Coprocessors, Coprocessors extend the reach of Processors

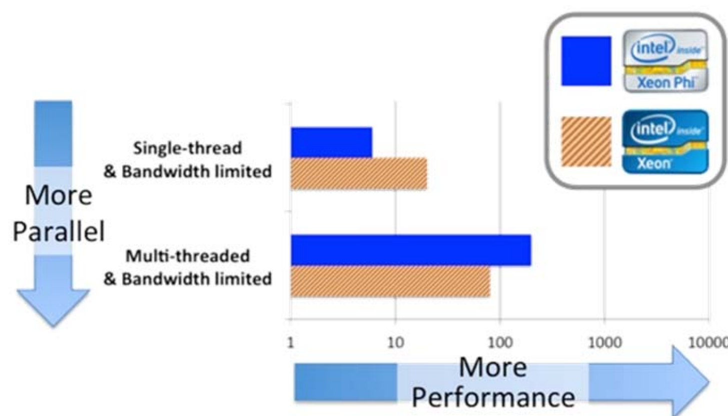

Figure 8: High Memory Needs are Helped with Threads on Processors and Coprocessors, Coprocessors extend the reach of Processors

Copyright © 2012, Intel Corporation. All rights reserved.

Coming back to the very first figure of this paper, Figure 9, 10 and 11 offer a view to illustrate the same need for constructing to use lots of threads and vectors. The first figure illustrates model data to make a point: Intel Xeon Phi coprocessors can reach heights in performance beyond that of an Intel Xeon processor, but it requires more parallelism to do so. The other two figures are simply close-ups of parts of the first figure to make a couple of points. Figure 10 illustrates the universal need for more parallelism to reach the same performance level on a device optimized for high degrees of parallelism (in this case, an Intel Xeon Phi coprocessor). Figure 11 illustrates that limiting "highly parallel" to the levels of parallelism that peak an Intel Xeon processor are insufficient to be interesting on an Intel Xeon Phi coprocessor. These close up looks drive home the point of the first figure: to go faster, you need more parallelism, while adding the less obvious "to go the same speed, you need more parallelism."

How to get more than hundred threads?

Fortunately, use of OpenMP*, Fortran *do concurrent*, Intel® Threading Building Blocks (TBB) and Intel® Cilk™ Plus offer capabilities that support creating this many threads in a program. Most commonly an outermost loop is transformed by a directive, keyword or template to become a parallel loop that *offers* the ability for many threads. I say *offers* because a loop with a million iterations may *offer* the opportunity for a million threads but we will show restraint and we will create far fewer at run time to efficiently match the hardware. Listings 1 and 2 give the simplest examples of what creating threads from a loop in OpenMP looks like through the addition of a single directive or pragma. Generally some additional restructuring or alignment work is needed for optimal performance, but that work generally looks quite natural after it has been done and the directive remains the notable code change you will see on inspection of a program.

Copyright © 2012, Intel Corporation. All rights reserved.

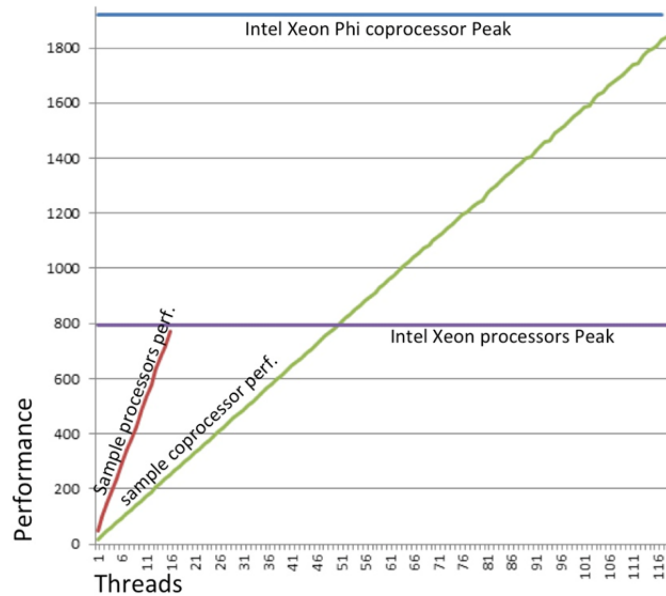

Figure 9: The potential is higher, but so is the parallelism needed to get there.

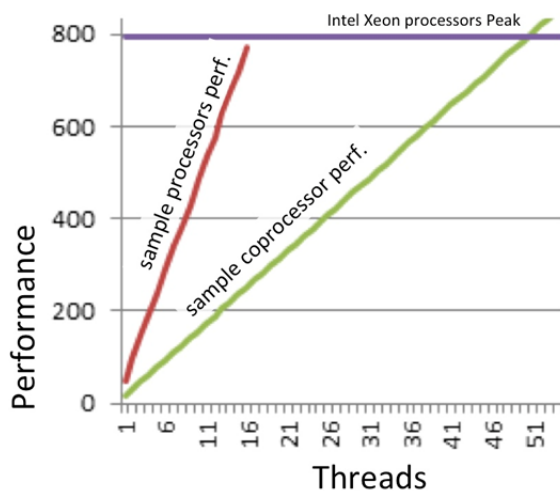

Figure 10: If you only need the peak of an Intel Xeon processor, then an Intel Xeon processor can do it with fewer threads.

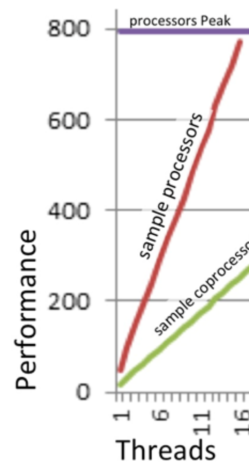

Figure 11: If you limit threading to only what an Intel Xeon processor needs, then you are not highly parallel.

Copyright © 2012, Intel Corporation. All rights reserved.

```
!$OMP PARALLEL do PRIVATE(j,k)
   do i=1, M
      ! each thread will work its own part of the problem
      do j=1, N
          do k=1, X
              ! calculations
          end do
      end do
  end do
```

Listing 1: Fortran do loop transformed to create many threads using an OpenMP directive

```
#pragma omp parallel for private(j,k)
   for (i=0; i<M; i++) {
      // each thread will work its own part of the problem
      for (j=0; j<N; j++) {
         for (k=0; k<X; k++)  {
            // calculation
         }
      }
   }
```

Listing 2: C for loop transformed to create many threads using an OpenMP pragma

Maximizing parallel program performance

When choosing whether to run an application on Intel Xeon processors or Intel Xeon Phi coprocessors, we can start with two fundamental considerations to achieve high performance:

1. Scaling: Is the scaling of an application ready to utilize the highly parallel capabilities of an Intel Xeon Phi coprocessor? The strongest evidence of this is generally demonstrated scaling on Intel Xeon processors.
2. Vectorization and Memory usage: Is the application either:
   a. Making strong use of vector units?
   b. Able to utilize more local memory bandwidth than available with Intel Xeon processors?

If these two fundamentals (both #1 and #2) are true for an application, then the highly parallel and power-efficient Intel Xeon Phi coprocessor is most likely to be worth evaluating.

Ways to measure readiness for highly parallel execution

To know if your application is maximized on an Intel Xeon processor based system, you should examine how your application scales, uses vectors and uses memory. Assuming you have a working application, you can get some impression of where you are with regards to scaling and vectorization by doing a few simple tests.

To check scaling, create a simple graph of performance as you run with various numbers of threads (from one up to the number of cores, with attention to thread affinity) on an Intel Xeon processor based system. This can be done with settings for OpenMP*, Intel® Threading Building Blocks (Intel TBB) or Intel® Cilk™ Plus (for example, OMP_NUM_THREADS for OpenMP). If the performance graph indicates any significant trailing off of performance, you have tuning work you can do to improve your application scaling before trying an Intel Xeon Phi coprocessor.

Copyright © 2012, Intel Corporation. All rights reserved.

To check vectorization, compile your application with and without vectorization. For instance, if you are using Intel compilers auto-vectorization: disable vectorization via compiler switches: -no-vec –no-simd, use at least –O2 -xhost for vectorization. Compare the performance you see. If the performance difference is insufficient you should examine opportunities to increase vectorization. Look again at the dramatic benefits vectorization may offer as illustrated in Figure 7. If you are using libraries, like the Intel Math Kernel Library (MKL), you should consider that math routines would remain vectorized no matter how you compile the application itself. Therefore, time spent in the math routines may be considered as vector time. Unless your application is bandwidth limited, the most effective use of Intel Xeon Phi coprocessors will be when most cycles executing are in vector instructions[2]. While some may tell you that "most cycles" needs to be over 90%, I have found this number to vary widely based on the application.

Intel VTune Amplifier XE 2013 can help measure computations on Intel Xeon processors and Intel Xeon Phi coprocessors to assist in your evaluations.

Aside from vectorization, being limited by memory bandwidth on Intel Xeon processors can indicate an opportunity to improve performance with an Intel Xeon Phi coprocessor. For this to be most efficient, an application needs to exhibit good locality of reference and utilize caches well in its core computations.

The Intel VTune Amplifier XE product can be utilized to measure various aspects of a program, and among the most critical is *L1 Compute Density*. This is greatly expanded upon in a paper titled "Using Hardware Events for Tuning on Intel® Xeon Phi™ Coprocessor (code name: Knights Corner)."

When using MPI, it is desirable to see a communication versus computation ratio that is not excessively high in terms of communication. The ratio of computation to communication will be a key factor in deciding between using offload versus native model for programming for an application. Programs are also most effective using a strategy of overlapping communication and I/O with computation. Intel® Trace Analyzer and Collector, part of Intel Cluster Studio XE 2013, is very useful for profiling MPI communications to help visualize bottlenecks and understand the effectiveness of overlapping with computation.

What about GPUs?

While GPUs cannot offer the programmability of an Intel Xeon Phi coprocessor, they do share a subset of what can be accelerated by scaling combined with vectorization or bandwidth. In other words, applications that show positive results with GPUs should always benefit from Intel Xeon Phi coprocessors because the same fundamentals of vectorization or bandwidth must be present. The opposite is not true. The flexibility of an Intel Xeon Phi coprocessor includes support for applications that cannot run on GPUs. This is one reason that a system built including Intel Xeon Phi coprocessors will have broader applicability than a system using GPUs. Additionally, tuning for GPU is generally too different from a processor to have the dual-transforming-tuning benefit we see in programming for Intel Xeon Phi coprocessors. This can

[2] In other words, a Vector Processing Unit (VPU) instructions being used on vector (not scalar) data.

Copyright © 2012, Intel Corporation. All rights reserved.

lead to substantial rise in investments to be portable across many machines now and into the future.

Beyond the Ease of Porting to Increased Performance

Because an Intel Xeon Phi coprocessor is an x86 SMP-on-a-chip, it is true that a port to an Intel Xeon Phi coprocessor is often trivial. However, the high degree of parallelism of Intel Xeon Phi coprocessors is best suited to applications that are structured to use the parallelism. Almost all applications will benefit from some tuning beyond the initial base performance to achieve maximum performance. This can range from minor work to major restructuring to expose and exploit parallelism through multiple tasks and use of vectors. The experiences of users of Intel Xeon Phi coprocessors and the "forgiving nature" of this approach are generally promising but point out one challenge: the temptation to stop tuning before the best performance is reached. This can be a good thing if the return on investment of further tuning is insufficient and the results are good enough. It can be a bad thing if expectations were that working code would always be high performance. There ain't no such thing as a free lunch! The hidden bonus is the "transforming-and-tuning" double advantage of programming investments for Intel Xeon Phi coprocessors that generally applies directly to any general-purpose processor as well. This greatly enhances the preservation of any investment to tune working code by applying to other processors and offering more *forward scaling* to future systems.

Transformation for Performance

There are a number of possible user-level optimizations that have been found effective for ultimate performance. These advanced techniques are not essential. They are possible ways to extract additional performance for your application. The "forgiving nature" of the Intel Xeon Phi coprocessor makes transformations optional but should be kept in mind when looking for the highest performance. It is unlikely that peak performance will be achieved without considering some of these optimizations:

- Memory access and loop transformations (for example: cache blocking, loop unrolling, prefetching, tiling, loop interchange, alignment, affinity)
- Vectorization works best on unit-stride vectors (the data being consumed is contiguous in memory). Data structure transformations can increase the amount of data accessed with unit-strides (such as AoS[3] to SoA[4] transformations or recoding to use packed arrays instead of indirect accesses).
- Use of full (not partial) vectors is best, and data transformations to accomplish this should be considered.
- Vectorization is best with properly aligned data.
- Large page considerations (we recommend the widely used Linux libhugetlbfs library)
- Algorithm selection (change) to favor those that are parallelization and vectorization friendly.

[3] Array of Structures (AoS)
[4] Structure of Arrays (SoA)

Copyright © 2012, Intel Corporation. All rights reserved.

Hyper-threading versus Multi-threading

The Xeon Phi coprocessor utilizes multithreading on each core as a key to masking the latencies inherent in an in-order microarchitecture. This should not be confused with hyper-threading on Xeon processors that exists primarily to more fully feed a dynamic execution engine. In HPC workloads, very often hyper-threading may be ignored or even turned off without degrading effects on performance. This is not true of Xeon Phi coprocessor hardware threads, where multithreading of programs should not be ignored and hardware threads cannot be turned off.

The Intel Xeon Phi coprocessor offers four hardware threads per core with sufficient memory capabilities and floating-point capabilities to make it generally impossible for a single thread per core to approach either limit. Highly tuned kernels of code may reach saturation with two threads, but generally applications need a minimum of three or four active threads per core to access all that the coprocessor can offer. For this reason, the number of threads per core utilized should be a tunable parameter in an application and be set based on experience in running the application. This characteristic of programming for Intel products will continue into the future, even though the "hyper-threading versus hardware threading" and the number of hardware threads may vary. Programs should parameterize the number of cores and the number of threads per core in order to easily run well on a variety of current and future processors and coprocessors.

Coprocessor major usage model: MPI versus Offload

Given that we know how to program the Intel Xeon processors in the host system, the question that arises is how to involve the Intel Xeon Phi coprocessors in an application. There are two major approaches: (1) a processor centric "offload" model where the program is viewed as running on processors and offloading select work to coprocessors, (2) a "native" model where the program runs natively on processors and coprocessors which may communicate with each other by various methods. An MPI program can be structured using either model. An MPI program with ranks only on processors may employ offload to access the performance of the coprocessors. An MPI program may run in a native mode with ranks on both processors and coprocessors. There is really no machine "mode" in either case, only a programming style that can be intermingled in a single application if desired. Offload is generally used for finer grained parallelism and as such generally involves localized changes to a program. MPI is more often done in a coarse grained manner often requiring more scattered changes in a program in order to add MPI calls. Intel MPI is tuned for both processors and coprocessors, so can exploit hardware features like remote direct memory access (RDMA).

Being separate and on a PCIe bus creates two additional considerations. One is the need to fit problems or subproblems into the more limited memory on the coprocessor card, and the other is the overhead of data transfers that favor minimization of communication to and from the card. It is worth noting also, that the number of MPI ranks used on an Intel Xeon Phi coprocessor should be substantially fewer than the number of cores in no small part because of limited memory on the coprocessor. Consistent with parallel programs in general, the advantages of overlapping communication (MPI messages or offload data movement) with computation are important to consider as well as techniques to load balance work across all

Copyright © 2012, Intel Corporation. All rights reserved.

the cores available. Of course, involving Intel Xeon processor cores and Intel Xeon Phi coprocessor cores adds the dimension of "big cores" and "little cores" to the balancing work even though they share x86 instructions and programming models. While MPI programs often already tackle the overlap of communication and computation, the placement of ranks on coprocessor cores still have to deal with the highly parallel programming needs and limited memory. This is why an offload model can be attractive, even within an MPI program where ranks are on the processors.

The offload model for Intel Xeon Phi coprocessors is quite rich. The syntax and semantics of the Intel Language Extensions for Offload includes capabilities not present in some other offload models including OpenACC. This provides for greater interoperability with OpenMP, ability to manage multiple coprocessors (cards), and the ability to offload complex program components that an Intel Xeon Phi coprocessor can process but that a GPU could not (hence OpenACC does not allow). We expect that a future version of OpenMP will include offload directives that provide support for these needs, and Intel plans to support such a standard for Intel Xeon Phi coprocessors as part of our commitment to providing OpenMP capabilities. Intel Language Extensions for Offload also provide for an implicit sharing model that is beyond what OpenMP will support. It rests on a shared memory model supported by Intel Xeon Phi coprocessors that allow a shared memory-programming model (Intel calls "MYO") between Intel Xeon processors and Intel Xeon Phi coprocessors. This bears some similarity to PGAS (partitioned global address space) programming models and is not an extension provided by OpenMP. It is not a PGAS implementation however! The Intel "MYO" capability offers a global address space within the node allowing sharing of virtual addresses, for select data, between processors and coprocessor on the same node. It is offered in C and C++, but not Fortran since future support of Coarray will be a standard solution to the same basic problem. Offloading is available as Fortran offloading via pragmas, C/C++ offloading with pragmas and optionally shared (MYO) data. Use of MPI can distribute applications across the system as well.

Compiler and programming models

No popular programming language was designed for parallelism. In many ways, Fortran has done the best job adding new features, such as DO CONCURRENT, to address parallel programming needs as well as benefiting from OpenMP. C users have OpenMP as well as Intel Cilk™ Plus. C++ users have embraced Intel Threading Building Blocks and more recently have Intel Cilk™ Plus to utilize as well. C++ users can use OpenMP and OpenCL as well.

Intel Xeon Phi coprocessors offer the full capability to use the same tools, programming languages and programming models as an Intel Xeon processor. However, as a coprocessor designed for high degrees of parallelism – some models are more interesting than others.

In a way, it is quite simple: an application needs to deal with having lots of tasks and deal with vector data efficiently (also known as vectorization).

There are some recommendations we can make based on what has been working well for developers. For Fortran programmers, use OpenMP, DO CONCURRENT and MPI. For C++ programmers, use Intel TBB, Intel Cilk Plus and OpenMP. For C programmers, use OpenMP and Intel Cilk Plus. Intel TBB is a C++ template library that offers excellent support for task

Copyright © 2012, Intel Corporation. All rights reserved.

oriented load balancing. While Intel TBB does not offer explicit support for vectorization, it does not interfere with any choice of solution for vectorization. Intel TBB is open source and available on a wide variety of platforms supporting most operating systems and processors. Intel Cilk Plus is a bit more complex in that it offers support for both tasking and vectorization. Fortunately, Intel Cilk Plus fully interoperates with Intel TBB. Intel Cilk Plus offers a simpler set of tasking capabilities than Intel TBB but by using keywords in the language so as to have full compiler support for optimizing.

Intel Cilk Plus also offers elemental functions, array syntax and "#pragma SIMD" to help with vectorization. Best use of array syntax is done along with blocking for caches, which unfortunately means naïve use of constructs such as A[:] = B[:] + C[:]; for large arrays may yield poor performance. Best use of array syntax ensures the vector length of single statements is short (some small multiple of the native vector length, perhaps only 1X). Finally, and perhaps most important to programmers today, Intel Cilk Plus offers mandatory vectorization pragmas for the compiler called "#pragma SIMD." The intent of "#pragma SIMD" is to do for vectorization what OpenMP has done for parallelization. Intel Cilk Plus requires compiler support. It is currently available from Intel for Windows, Linux and Apple OS X. It is also available in a branch of gcc.

If you are happy with OpenMP and MPI, you have a great start to using Intel Xeon Phi coprocessors. Additional options may be interesting to you over time, but OpenMP and MPI are enough to get great results when used with an effective vectorization method. Auto-vectorization may be enough for you especially if you code in Fortran with the possible additional considerations for efficient vectorization such as alignment and unit-stride accesses. The "#pragma SIMD" capability of Intel Cilk Plus (available in Fortran too) is worth a look. In time, you may find it has become part of OpenMP.

Dealing with tasks means specification of work to be done, and load balancing among them. MPI has been used for decades with both full flexibility and full responsibility given to the programmer. More recently, shared memory programmers have Intel TBB and Intel Cilk Plus to assist them with built-in tasking models. Intel TBB has widespread usage in the C++ community, and Intel Cilk Plus extends Intel TBB to offer C programmers a solution as well as help with vectorization in C and C++ programs.

Cache optimizations

The most effective use of caches comes by paying attention to maximizing the locality of references, blocking to fit in L2 cache, and ensuring that prefetching is utilized (by hardware, by compiler, by library or by explicit program controls).

Organizing data locality to fit 512K or less L2 cache usage per core generally gives best usage of the L2 cache. All four hardware threads per core share their "per core" local L2 cache but have high-speed access to the caches associated with other cores. Any data used by a particular core will occupy space in that local L2 cache (it can be in multiple L2 caches around the chip). While Intel Xeon processors have a penalty for "cross-socket" sharing, which occurs after about 16 threads (assuming 8 cores, two hyper-threads each), the Intel Xeon Phi coprocessors have a lower penalty across more than 200 threads. There is a benefit to having locality first organized

Copyright © 2012, Intel Corporation. All rights reserved.

around the threads being used on a core (up to 4) first, and then around all the threads across the coprocessor. While the defaults and automatic behavior can be quite good, ensuring that code designed for locality performs best will likely include programmatic specification of affinity such as the use of KMP_AFFINITY when using OpenMP and I_MPI_PIN_DOMAIN with MPI. Note that while there is a strong benefit in sharing for threads on the same core, but beyond that you should not expect to see performance variations based on how close a core is to another core on coprocessor. While this may seem surprising, the hardware design is so good in this respect we have yet to see any appreciable performance benefit based on adjacency of cores within the coprocessor so I would not advise spending time trying to optimize placement beyond locality to a core and then load balancing across the cores (for instance using KMP_AFFINITY=scatter to round-robin work allocation).

The coprocessor has hardware prefetching into L2 that is initiated by the first cache miss within a page. The Intel compilers issue software prefetches aggressively for memory-references inside loops by default (-O2 and above, report on compiler decisions available with -opt-report3 -opt-report-phase hlo). Typically, the compiler issues two prefetches per memory reference: one from memory into L2 and a second one for prefetching from L2 into L1. The prefetch distance is computed by the compiler based on the amount of work inside the loop. Explicit prefetching can be added either with prefetch pragmas (#pragma prefetch in C/C++ or CDEC$ prefetch in Fortran) or prefetch intrinsics (_mm_prefetch in C/C++, or mm_prefetch in Fortran); you may want to explicitly turn off compiler prefetching (-opt-prefetch=0 to turn off all compiler prefetches *or* -opt-prefetch-distance=0,2 to turn off compiler prefetches into L2) when you are adding prefetch intrinsics manually. Software prefetches do not get counted as cache misses by the performance counters. This means that "cache misses" can truly be studied with performance tools with a goal of driving them to essentially zero (aim for low single digit percentages of memory accesses counted as misses) inside loops when you can successfully fetch all data streams with prefetches. Prefetching needs to be from memory to L2 and separately from L2 to L1. Utilizing prefetches from memory to L1 directly should not be expected to yield great performance because the latency of such will generally lead to more outstanding L1 prefetches than are available in the hardware. Such limits mean that organizing data streams is best when the number of streams of data per thread is less than eight and prefetching is actively used on each stream of data. As a rule of thumb, the number of active prefetches per core should be managed to about 30 or 40 and be divided across the active data streams.

Beyond the caches, certain memory transforms can be employed for additional tuning for TLBs including the ability to use large or small pages, organizing data streams, and to organizing data to avoid hot spots in the TLBs.

Keeping the "Ninja Gap" under control

On the premise that parallel programming can require Ninja (expert) programmers, the gaps in knowledge and experience needed between expert programmers and the rest of us have been referred to as the "Ninja Gap." Optimization for performance is never easy on any machine, but it is possible to control the level of complexity to manageable levels to avoid a high "Ninja Gap." To understand more about how this "Ninja Gap" can be quantified, you might read the June 2012 ISCA paper "Can Traditional Programming Bridge the Ninja Performance Gap for Parallel

Copyright © 2012, Intel Corporation. All rights reserved.

Computing Applications?" The paper shares measurements of the challenges and shows how Intel Xeon Phi coprocessors offer the advantage of controlling the "Ninja Gap" to levels akin to general-purpose processors. The approach taken is able to rely on the same industry standard methods as general-purpose processors and the paper helps show how that benefit can be measured and shown to be similar to general-purpose processors.

Summary: transforming-and-tuning double advantage

Programming should not be called easy, and neither should *parallel programming*. We can however, work to keep the fundamentals the same: maximizing parallel computations and minimizing data movement. Parallel computations are enabled through scaling (more cores and threads) and vector processing (more data processed at once). Minimal data movement is an algorithmic endeavor, but can be eased through the higher bandwidth between memory and cores that is available with the Intel Many Integrated Core (MIC) Architecture that is used by Intel Xeon Phi coprocessors. This leads to *parallel programming* using the same programming languages and models across Intel Xeon processors and Intel Xeon Phi coprocessors, which are generally also shared across all general-purpose processors in the industry. Languages such Fortran, C and C++ are fully supported. Popular programming methods such as OpenMP*, MPI and Intel TBB are fully supported. Newer models with widespread support such as Coarray Fortran, Intel® Cilk™ Plus and OpenCL* can apply as well.

Tuning on Intel Xeon Phi coprocessors, for scaling, vector usage and memory usage, all stand to also benefit the application when run on Intel Xeon processors. This protection of investment by maintaining a value across Intel products is critical for helping preserve past and future investments. Applications that initially fail to get maximum performance on Intel Xeon Phi coprocessors, generally trace problems back to scaling, vector usage or memory usage. When these issues are addressed, these improvements to the application usually have a related positive effect when run on Intel Xeon processors. Some people call this the double advantage of "transforming-and-tuning" and have found it to be among the most compelling features of the Intel Xeon Phi coprocessors.

Additional reading

Additional material regarding programming for Intel Xeon Phi coprocessors can be found at http://intel.com/software/mic.

A new book titled "Intel® Xeon Phi™ Coprocessor High Performance Programming, Volume 1: Essentials" by Jim Jeffers and James Reinders, © 2013, published by Intel Press, is expected to be available in early 2013.

Copyright © 2012, Intel Corporation. All rights reserved.

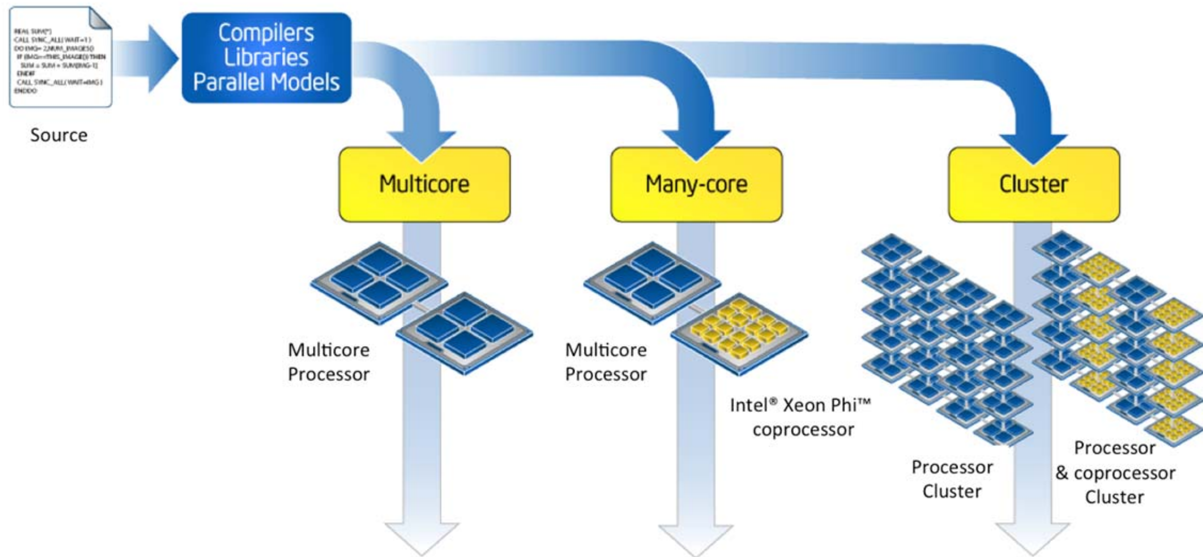

Figure 12: The double advantage of transforming-and-tuning means that optimizations are shared across the Intel products; Capabilities of Intel Xeon Processors are extended by Intel Xeon Phi Coprocessors.

Copyright © 2012, Intel Corporation. All rights reserved.

About the Author

James Reinders, Director, Software Evangelist, Intel Corporation

James Reinders is a senior engineer who joined Intel Corporation in 1989 and has contributed to projects including systolic arrays systems WARP and iWarp, the world's first TeraFLOP/s supercomputer (ASCI Red), the world's first TeraFLOP/s microprocessor (Intel Xeon Phi coprocessor, code name Knights Corner), as well as compilers and architecture work for multiple Intel processors and parallel systems.

James has been a driver behind the development of Intel as a major provider of software development products, and serves as their chief software evangelist. James is author of books VTune™ Performance Analyzer Essentials (Intel Press), Threading Building Blocks (O'Reilly Media), and Structured Parallel Programming (Morgan Kauffman, 2012). James is currently co-authoring a book about programming for the Intel Xeon Phi coprocessor due out in early 2013 from Intel Press.

James is currently involved in multiple efforts at Intel to bring parallel programming models to the industry including for the Intel MIC architecture. James received his B.S.E. in Electrical and Computing Engineering and M.S.E. in Computer Engineering from the University of Michigan.

Acknowledgements

James wishes to acknowledge those who helped with their encouragement, input and feedback on this paper. If this list fails to mention anyone who helped, James regrets the omission and asks that you please let him know. Those who helped include: Mani Anandan, Andrew Brownsword, George Chrysos, Kim Colosimo, James Cownie, Joe Curley, Pradeep Dubey, Gustavo Esponosa, Robert Geva, Milind Girkar, Ron Green, Michael Greenfield, John Hengeveld, Scott Huck, David E. Hudak, Jim Jeffers, Mike Julier, Rakesh Krishnaiyer. Belinda Liverio, David Mackay, Bill Magro, Larry Meadows, Chris Newburn, Brian Nickerson, Beth Reinders, Frances Roth, Bill Savage, Elizabeth Schneider, Sanjiv Shah, Lance Shuler, Brock Taylor, Philippe Thierry, Xinmin Tian, Frank Titze, Andrey Vladimirov, Matt Walsh, and Joe Wolf.

Copyright © 2012, Intel Corporation. All rights reserved.

Notice and Disclaimers

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY 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.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document 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.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Intel, Xeon, Xeon Phi, Cilk, VTune and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Copyright © 2012, Intel Corporation. All rights reserved.

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

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*

Copyright © 2012, Intel Corporation. All rights reserved.