Programming for Multi-core and Many-core Intel® Architecture
(Including language extensions for offloading to an Intel® MIC co-processor)

Abstract
This paper provides the foundation for understanding how multi-core and many-core processors are best programmed using a unified programming approach that is abstracted, intuitive and effective. The programming models in use today, used for multi-core processors every day, are available for many-core processors as well. This approach is simple and natural because it fits applications of today easily and yields strong results. When combined with the common base of Intel Architecture (IA) utilized by Intel's many-core and multi-core processors, the result is performance for highly parallel computing with substantially less difficulty than with other less intuitive approaches.

Programs that utilize multi-core and many-core processors have a wide variety of options to meet varying needs. These options fully utilize existing widely adopted solutions, such as C, C++, Fortran, OpenMP®, MPI and Intel® Threading Building Blocks (Intel® TBB), and are rapidly driving the development of additional emerging standards such as Co-array Fortran, OpenCL® and Intel® Parallel Building Blocks (Intel® PBB).
Introduction

Single core processors are a shrinking minority of all the processors in the world. Multi-core processors, offering parallel computing, have displaced single core processors permanently. The future of computing is parallel computing, and the future of programming is parallel programming.

The methods to utilize multi-core processors have evolved in recent years, offering more and better choices for programmers than ever. Nothing exemplifies this more than the rapid rise in popularity of Intel Threading Building Blocks (Intel TBB) or the industry interest and support behind OpenCL.

At the same time that multi-core processors and programming methods are becoming common, Intel is introducing many-core processors that will participate in this evolution without sacrificing the benefits of IA. Additional capabilities that are new with many-core processors are addressed in a natural and intuitive manner. Intel many-core processors allow use of the same tools, programming languages, programming models, the same execution models, memory models and behaviors as in Intel's multi-core processors.

This paper explains the programming methods available for multi-core processors and many-core processors with a focus on widely adopted solutions plus emerging standards.

Parallel Programming Today

Since the goal of using IA in both multi-core and many-core processors is intuitive and common programming methods, it is important to first review where parallel programming for multi-core stands today and understand where it is headed. Because of their common IA foundations, this will also precisely define the basis for parallel programming for many-core processors.

Libraries (e.g. MKL and IPP)

Libraries provide an important abstract parallel programming method that needs to be considered before jumping into programming. Library implementations for algorithms including BLAS, video or audio encoders and decoders, Fast Fourier Transforms (FFT), solvers and sorters, are important to consider. Libraries such as the Intel Integrated Performance Primitives (IPP) and Intel® Math Kernel Library (Intel® MKL) already offer advanced implementations of many algorithms that are highly tuned to utilize SSE, Intel® AVX, multi-core and many-core processors. A program can get these benefits by adding a single call to a routine in one of the libraries. Intel MKL offers the normal interfaces in Fortran, and supports the new industry standard for C interfaces to Linear Algebra PACKage (LAPACK) that Intel helped create. Standards combined with Intel’s pursuit of high performance, make libraries an easy choice to utilize as the first preference in parallel programming.

When libraries do not solve specific programming needs, developers turn to programming languages that have been in use for many years.

None of the most popular programming languages were designed for parallel programming. This has brought about many proposals for new programming languages as well as extensions for the pre-
existing languages. In the end, these experiences have led to the emergence of a number of widely deployed solutions for parallel programming using C, C++ and Fortran.

The most widely used abstractions for parallel programming are OpenMP (primarily C and Fortran), Intel Threading Building Blocks (primarily C++) and MPI (C, C++ and Fortran).

Additionally, the native threading methods of the operating system are directly available for programmers. These interfaces, including POSIX threads (pthreads) and Windows* threads, offer a low level interface for full control but without the benefits of high level programming abstractions. These interfaces are essentially assembly language programming for parallel computing. Programmers have all but completely moved to higher levels of abstraction and abandoned assembly language programming. Similarly, avoiding direct use of threading models has been a strong trend that has accelerated with the introduction of multi-core processors. This shift to program in “tasks” and not “threads” is a fundamental and critical change in programming habits that is well supported by the abstract programming models.

Most deployed parallel programming today is either done with one of the three most popular abstractions for parallelism (OpenMP, MPI or Intel TBB), or done using the raw threading interfaces of the operating system.

These standards continue to evolve and new methods are proposed. Today, the principle technical drivers of these evolutions are highly data parallel hardware and advancing compiler technology. Both of these driving forces are motivated by a strong desire to program at higher levels of abstraction so as to increase programmer productivity leading to faster time-to-money and reduced development and maintenance costs.

Intel® Parallel Building Blocks (Intel® PBB)

Learn more at http://Intel.com/go/parallel

For reasons explained in this paper, the most composable parallel programming methods are Intel Threading Building Blocks (Intel TBB), Intel Cilk™ Plus and Intel Array Building Blocks (Intel ArBB). They are collectively referred to as Intel Parallel Building Blocks (Intel PBB) because they offer consistent advantages for effective abstract programming that yield performance and preserve programming investments. Intel PBB provides recombinant components that can be selected and assembled for effective parallel programs. Even though part of Intel PBB can be described and studied individually, they are best thought of as a collection of capabilities that are easily utilized both individually and together. This is incredibly important since it offers compositability for mixing modules including libraries. Components of Intel PBB have self-composability, which is not the case for threading and for OpenMP. Uses of OpenMP and OpenCL have limitations on compositability, but are still preferable to the use of raw threads. One of the wonderful things about IA multi-core and many-core processors is the strong support for all these methods that is available, offering the solution that best fits your current and future programming needs.

OpenMP

Learn more at http://openmp.org/wp/
In 1996, the OpenMP standard was proposed as a way for compilers to assist in the utilization of parallel hardware. Now, after more than a decade, every major compiler for C, C++ and Fortran supports OpenMP. OpenMP is especially well suited for the needs of Fortran programs as well as scientific programs written in C. Intel is a member of the OpenMP work group and a leading vendor of implementations of OpenMP and supporting tools. OpenMP is applicable to both multi-core and many-core programming.

OpenMP dot product example, in Fortran:

```
!$omp do
do j = 1, n
   adotb = adotb + a(j) * b(j)
end do
!$omp end do
```

OpenMP summation (reduction) example, in C:

```
#pragma omp parallel for reduction(+: s)
for (int i = 0; i < n; i++)
   s += x[i];
```

Intel TBB

Intel introduced Intel® Threading Building Blocks (Intel® TBB) in 2006 and the open source project for INTEL TBB was started in 2007. By 2009, it had grown in popularity to exceed that of OpenMP in terms of number of developers using it (per Evans Data Corp: [http://www.evansdata.com/research/market_alerts.php](http://www.evansdata.com/research/market_alerts.php), and subsequent research reports). Intel TBB is especially well suited for the needs of C++ programmers.

Parallel function invocation example, in C++, using Intel TBB:

```
parallel_for (0, n,

   [=](int i) {
      Foo(a[i]);
   });
```

The emergence of Intel TBB, which does not directly require nor leverage compiler technology, emphasized the value of programming to tasks and led the way for wide acceptance of using task-stealing systems. Compiler technology continues to evolve to help address parallel programming and led to the creation of the Intel Cilk™ Plus (Cilk Plus) project. Increased use of compiler technology is better able to unlock the full potential of MIC. Intel remains a leading participant and contributor in the Intel TBB open source project as well as a leading supplier of Intel TBB support and supporting tools. Intel TBB is applicable to multi-core and many-core programming.
Programming for Multi-core and Many-core Intel® Architecture
(INCLUDING LANGUAGE EXTENSIONS FOR OFFLOADING TO AN INTEL MIC CO-PROCESSOR)

MPI

For programmers utilizing a cluster, in which processors are connected by the ability to pass messages but not always the ability to share memory, the Message Passing Interface (MPI) is the most common programming method. In a cluster, communication continues to use MPI, as they do today, regardless of whether a node has many-core processors or not.

Today's MPI based programs move easily to MIC based systems because MIC supports ranks that can talk to other MIC ranks and multi-core (e.g., Intel Xeon® processor) ranks. A MIC processor, like a multi-core processor, may create as many ranks as the programmer desires. Such ranks communicate with other ranks regardless of whether they are on multi-core or many-core processors.

Because MIC processors are general-purpose processors, MPI jobs run on MIC processors. This is very powerful because no algorithmic recoding or refactoring is required to get working results from an existing MPI program. The general capabilities of MIC combined with the power of MPI support on MIC produce immediate results in a manner that is intuitive for MPI programmers.

The widely used Intel MPI library offers both high performance and support for virtually all interconnects. The Intel MPI library supports multi-core and many-core processor based systems creating ranks on multi-core and many-core processors in a fashion that is familiar and consistent with MPI programming today.

MPI on MIC also composes with other thread models (e.g., OpenMP, Intel TBB, Intel® Cilk Plus) as has become common on multi-core processors based systems.

Intel is a leading vendor of MPI implementations and tools. MPI is applicable to multi-core and many-core programming.

Parallel Programming Emerging Standards

For data parallel hardware, the emergence of extensions to C, OpenCL, and Intel ArBB offer important options for developers and address programmer productivity in different ways.

Co-Array Fortran

The latest Fortran standard introduced the “co-array” extensions as an additional method to use Fortran as a robust and efficient parallel programming language.

The Intel Fortran Compiler supports parallel programming using co-arrays as defined in the Fortran 2008 standard for both shared memory and distributed memory systems. Implementations of the new co-array feature in Fortran for MIC do not yet exist. Co-array remains an emerging standard to follow.
Programming for Multi-core and Many-core Intel® Architecture
(Including language extensions for offloading to an Intel MIC co-processor)

Sum in Fortran, using co-array feature:

```
REAL SUM[*]
CALL SYNC_ALL( WAIT=1 )
DO IMG= 2,NUM_IMAGES()
   IF (IMG==THIS_IMAGE()) THEN
      SUM = SUM + SUM[IMG-1]
   ENDIF
   CALL SYNC_ALL( WAIT=IMG )
ENDDO
```

Intel is a leading vendor of Fortran compilers and tools. Fortran is applicable to multi-core and many-core programming.

Intel® Cilk™ Plus
Learn more at http://cilk.com
Intel introduced Intel Cilk Plus in 2010. Built on research from M.I.T. and productization experiences by industry leader Cilk Arts, Intel implemented support for task stealing in compilers for Linux and Windows. Intel has published full specifications for Intel Cilk Plus to help enable other implementations as well as optional usage of Intel runtime or construction of interchangeable runtimes via API compliance. There is already an alternate runtime developed by M.I.T.; Intel is actively working with other compilers to offer support in the future for more compilers. Intel is proud to be the leading supporter in industry of Intel Cilk Plus with products and tools.

Intel Cilk Plus provides three new keywords, special support for reduction operations, and data parallel extensions. The keyword cilk_spawn can be applied to a function call, as in $x = \text{cilk\_spawn} \ fib(n-1)$, to indicate that the function $fib$ can execute concurrently with the subsequent code. The keyword cilk_sync indicates that execution has to wait until all spawns from the function have returned. The use of the function as a unit of spawn makes the code readable, relies on the baseline C/C++ language to define scoping rules of variables, and allows Intel Cilk Plus programs to be composable.

Parallel spawn in a recursive fibonacci computation, in C, using Intel Cilk Plus:

```
int fib (int n) {
   if (n < 2) return 1;
   else {
      int x, y;
      x = cilk_spawn fib(n-1);
      y = fib(n-2);
      cilk_sync;
      return x + y;
   }
}
```
Cilk offers exceptionally intuitive and effective compiler support for C and C++ programmers. Cilk is very easy to learn and poised to be widely adopted. A regular “for” loop can be transformed into a parallel loop by simply changing the keyword “for” into “cilk_for.” This indicates to the compiler that there is no ordering among the iterations of the loop.

```
cilk_for (int i=0; i<n; ++i){
    Foo(a[i]);
}
```

Cilk programmers still utilize Intel TBB for certain algorithms or features where new compiler keywords or optimizations are not needed, such as the thread aware memory allocator or a sort routine. This is valid due to the fact that Intel Cilk Plus, Intel TBB and Intel ArBB do interoperate and is consistent with the family approach of Intel Parallel Building Blocks (PBB), in which INTEL TBB, Cilk Plus and Intel ArBB are parts of a larger emerging standard way to program. Cilk Plus is applicable to multi-core and many-core programming.

**C/C++ data parallel extensions**

Learn more at [http://cilk.com](http://cilk.com)

Debate about how to extend C (and C++) to directly offer data parallel extensions is on-going. Implementations, experiences and adoption are important steps toward standardization. Intel has implemented extensions for fundamental data parallelism as part of Intel Cilk Plus for Linux, Windows and Mac OS X systems. Intel is actively working with other compilers to offer support in the future. An intuitive syntactic extension, similar to the array operations of Fortran 90, is provided as a key element of Intel Cilk Plus and allows simple operations on arrays. The C/C++ languages do not provide a way to express operations on arrays. A programmer has to write a loop and express the operation in terms of elements of the arrays, creating unnecessary explicit serial ordering. A better opportunity exists to write `a[:] = b[:] + c[:];` to indicate the per element additions but without specifying unnecessary serial ordering. These simplified semantics free up a compiler to always generate vector code instead of generating non-optimal scalar code.

An additional method to avoid unintended serialization, allows a programmer to write a scalar function in standard C/C++ and declare it as an "elemental function." This will trigger the compiler to generate a short vector version of that function, which instead of operating on a single set of arguments to the function, will operate on short vectors of arguments by utilizing the vector registers and vector ISA. In common cases, where the control flow within the function does not depend on data values, the execution of the short vector function can yield a vector of results in roughly the same time it would take the regular function to produce a single result.
Elemental functions, in C, using Cilk Plus:

```c
__declspec (vector) void saxpy(float a, float x, float &y)
{
    y += a * x;
}
```

Intel is supporting these syntactic extensions for C and C++ with products and tools as well as discussions with other compiler vendors for wider support. C, C++ and data parallel extensions are applicable to multi-core and many-core programming.

**OpenCL**

Learn more at [http://Intel.com/go/opencl](http://Intel.com/go/opencl)

OpenCL was first proposed by Apple and then moved to an industry standards body of which Intel is a participant and supporter. OpenCL offers a "close to the hardware" interface, offering some important abstraction and substantial control coupled with wide industry interest and commitment. OpenCL may require the most refactoring of any of the solutions covered in this whitepaper; Specifically, refactoring based on advanced knowledge of the underlying hardware. Results from refactoring work may be significant for multi-core and many-core performance, and the resulting performance may or may not be possible without such refactoring. A goal of OpenCL is to make an investment in refactoring productive when it is undertaken. Solutions other than OpenCL offer methods to avoid the need for refactoring based on advanced knowledge of the underlying hardware.

Simple per element multiplication using OpenCL:

```c
kernel void
    dotprod(  global const float *a,
              global const float *b,
              global float *c)
{
    int myid = get_global_id(0);
    c[myid] = a[myid] * b[myid];
}
```

Intel is a leading participant in the OpenCL standard efforts, and a vendor of solutions and related tools with early implementations available today. OpenCL is applicable to multi-core and many-core programming.

**Intel® ArBB**

Learn more at [http://Intel.com/go/arbb](http://Intel.com/go/arbb)

Intel® Array Building Blocks (Intel ArBB) was born of a merger of Intel research with the renowned RapidMind team and their industry-leading product. The infusion of resources by Intel into this important area has led to substantial improvements. Together, with C/C++ data parallel extensions and OpenCL, these offer new options for exploiting the emerging field of highly data parallel hardware.
Programming for Multi-core and Many-core Intel® Architecture
(Including language extensions for offloading to an Intel MIC co-processor)

Intel ArBB takes a higher-level approach in order to offer a more intuitive connection with mathematical concepts including sparse and irregular matrices and provide safety (completely avoids data race and deadlock programming errors by default).

Simple per element multiplication using Intel ArBB:

```c
void dotprod(
    const arbb::dense<arbb::f32>& a,
    const arbb::dense<arbb::f32>& b,
    arbb::dense<arbb::f32>& c)
{
    c = a * b;
}
```

Intel is a leading proponent of advanced solutions for data parallel abstractions with solutions and tools. Intel ArBB is applicable to multi-core and many-core programming.

**Composability Using Multiple Models**

Composability is an important concept. With multiple programming options to fit differing needs, it is essential that these methods not be mutually exclusive. The abstract programming methods discussed above can be mixed in a single application. By offering newer programming models that support composable programming, programmers are freed from subtle and unproductive limitations on the mixing and matching of programming methods in a single application.

The most composable methods are Intel TBB, Intel Cilk Plus (including the C/C++ data parallel extensions) and Intel ArBB. In part due to the special attention on composability with these methods, Intel has collectively referred to them as Intel Parallel Building Blocks (PBB). Use of OpenMP and OpenCL has limitations on composability, but are still preferable to the use of raw threads.

PBB provides recombinant components that can be selected and assembled for effective parallel programs. This is incredibly important since it offers composability for mixing modules including libraries. Components of PBB have self-composability, which is not the case for threading and for OpenMP.

**Harnessing Many-core**

Combining the power of both multi-core and many-core, and utilizing them together, offers enormous possibilities.

Intel MIC processors are designed to offer attached processing to offload highly parallel work from multi-core processors. Such platforms, containing both multi-core processors and MIC processors, can be referred to as *heterogeneous* platforms. Such a heterogeneous platform offers the best of both worlds, multi-core processors that are very flexible to handle general-purpose serial and parallel workloads as well as more specialized many-core processing capabilities for highly parallel workloads. A heterogeneous platform can be programmed as such and utilize a programming model to manage copying of data and transfer of control.

Intel Confidential
Programming for Multi-core and Many-core Intel® Architecture
(Including language extensions for offloading to an Intel MIC co-processor)

Applications are still built with a single source base. The versatility of IA multi-core and many-core processors allows for programming that is both intuitive and effective.

**Explicit vs. Implicit use of Many-core**

Many-core processors may be used implicitly through the use of libraries, like Intel MKL, by provisioning code to detect and utilize many-core processors when present. Explicit controls for Intel libraries are available to the developer, but the simple approach of relying on a library to decide if and when to use the attached multi-core processors can be quite effective. Similarly, programs written using Intel Array Building Blocks may be employed to optionally use many-core processors when present without requiring source code changes at all.

Additional programming opportunities are possible by explicit directions from the programmer in the source code. Writing an application to explicitly utilize many-core is done by writing a heterogeneous application. This program would consist of writing a parallel application, and splitting the work between the multi-core and many-core processors.

Even with explicit control, Intel has designed the extensions to be flexible enough to work if no many-core processors are present and to also be ready for a converged future. These two benefits are incredibly important. First, a single source program can provide direction to offload to a MIC co-processor. However at runtime, if the co-processor is not present on the system being utilized, the use of IA on both the multi-core and many-core processors means that the code available for offloading to MIC can be executed seamlessly on either type of processor.

**Offloading**

The reality of today’s hardware is that a heterogeneous platform contains multiple distinct memory spaces, one (or more in a cluster) for the multi-core processors and one for each many-core processor. The connection between multi-core and many-core processors can be a bottleneck that needs some consideration.

There are two approaches to utilizing such a heterogeneous platform. One approach treats the memory spaces as wholly distinct, and uses offload directives to move control and data to and from the multi-core processors. Another approach simplifies data concerns by utilizing a software illusion of shared memory called MYO to allow sharing between multi-core and many-core processors that reside in a single system or a single node on a cluster. MYO is an acronym for “Mine Yours Ours” and refers to the software abstraction that shares memory within a system for determining current access and ownership privileges.

The first approach exposes completely that the multi-core processors and many-core processors do not share memory. Compiler support of directives for this execution model are able to free the programmer from specifying the low level details of the system, while exposing the fundamental property that the target is heterogeneous and leaving the programmer to devote their time to solving harder problems.

Intel Confidential
Simple offload, in Fortran:

```fortran
!dir$ offload target(MIC1)
$omp parallel do
do i=1,10
  A(i) = B(i) * C(i)
enddo
$omp end parallel
```

The compiler provides a pragma for offload (#pragma offload) that a programmer can use to indicate that the subsequent language construct may execute on the Intel MIC. The pragma also offers clauses that allow the programmer to specify data items that would need to be copied between multi-core and many-core (Intel MIC) memory before the offloaded code executes. The clauses also allow the developer to specify data that should be copied back to multi-core processor memory afterwards. The offload pragma is available for C, C++ and Fortran.

Simple offload, in C, with data transfer:

```c
float *a, *b; float *c;

#pragma offload target(MIC1) \ in(a, b : length(s)) \ out(c : length(s) alloc_if(0))
for (i=0; i<s; i++) {
c[i] = a[i] + b[i];
}
```

An alternate approach is a run time user mode library called MYO. MYO allows synchronization of data between the multi-core processors and an Intel MIC, and with compiler support enabling allocation of data at the same virtual addresses. The implication is that data pointers can be shared between the multi-core and many-core memory spaces. Copying of pointer based data structures such as trees, linked lists, etc. is supported fully without the need for data marshaling. To use the MYO capability, the programmer will mark data items that are meant to be visible from both sides with the _Shared keyword, and use the _Offload keyword to invoke work on the Intel MIC. The statement `x = _Offload func(y);` means that the function `func()` is executed on the Intel MIC, and the return value is assigned to the variable `x`. The function may read and modify shared data as part of its execution, and the modified values will be visible for code executing on all processors.

The offload approach is very explicit and fits some programs quite well. The MYO approach is more implicit in nature and has several advantages. MYO allows copying of classes without marshaling and copying of C++ classes, which is not supported using offload pragmas. Importantly, MYO does not copy all shared variables upon synchronization. Instead, it only copies the values that have changed between two synchronization points.
These offload programming methods, while designed to allow control to direct work to many-core processors, are applicable to multi-core and many-core programming so as to allow code to be highly portable and long lasting even as systems evolve. Source code will not need to differ for systems with and without many-core processors.

**Additional Offload Capabilities**

Both the keyword and pragma mechanisms perform the copying of the data triggered by the invocation of work on the Intel MIC. Future directive options allow initiation of data copying ahead of invoking computation in order to be able to schedule other work while data is being copied.

Since systems may be configured with multi-core processors and more than one Intel MIC, additional language support will allow the programmer to choose between forcing offloading or allowing a runtime determination to be made. This option offers the potential for more dynamic and optimal decisions depending on the environment.

**Standards**

By utilizing IA on multi-core and many-core, programming tools and models are best able to serve both. With insights and use of the right models, a single source base can be constructed that is well equipped to utilize multi-core processor systems, heterogeneous systems and future converged systems in an intuitive and effective manner. This can be accomplished with a single source base in familiar and current programming languages.

Tried and true solutions, including C, C++, Fortran, OpenMP, MPI and Intel TBB apply to these IA multi-core and many-core systems.

Emerging efforts including PBB, Co-Array Fortran, offload extensions and OpenCL are strongly supported by Intel, and are poised for broad adoption and support in the future.

The path to standardization starts with strong products and published specifications, progresses to users (customers) and support by additional vendors. Viable standards will follow. With Co-Array Fortran, OpenCL, PBB (specifically Intel ArBB and Intel Cilk Plus) and offload extensions, the product support and specifications exist and customer usage is well under way. It is reasonable to expect that wider support and standards refined based on user experiences will follow.

**Summary**

By utilizing Intel Architecture and industry standard programming tools, Intel’s multi-core and many-core processors offer parallel programming methods that can be applied across both. These methods can employ a single source code base using familiar tools, programming languages and previous source code investments. Current and emerging solutions allow applications to grow into a single code section that best utilizes multi-core and many-core processors together. The methods available to utilize multi-core and many-core parallelism offer performance while preserving investments and offering intuitive programming methods.

Standards play an important role in programming methods. Intel has invested heavily to support and implement standard programming models and methods. In addition, Intel has been a leader in the evolution of standards to solve new challenges.
When programming for Intel MIC, applications can get the power of Intel MIC in a maintainable and performant application that is highly portable and scales to future architectures while fully supporting multi-core systems with the same code.

Clusters of multi-core and many-core processors, organized in nodes, will be able to take advantage of this very rich set of tools and programming models available for IA in an intuitive, maintainable and effective manner.

The ability to utilize existing developer tools, standards, performance and offer flexibility puts Intel multi-core and many-core solutions in a class of their own.
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, and the world’s first TeraFLOP supercomputer (ASCI Red), as well as compilers and architecture work for multiple Intel processors and parallel systems. James has been a leader in the development of Intel as a major provider of software development products, and serves as their chief software evangelist. His most recent book is “Intel Threading Building Blocks” from O’Reilly Media which has been translated to Japanese, Chinese and Korean. James has published numerous articles and has contributed to several books. James received his B.S.E. in Electrical and Computing Engineering and M.S.E. in Computer Engineering from the University of Michigan.
Notices

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.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

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, the Intel logo, Cilk, and Xeon are trademarks of Intel Corporation in the U.S. and other countries.

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

Copyright© 2011 Intel Corporation. All rights reserved.
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 #20110307