Finding Non-trivial Opportunities for Parallelism in Existing Serial Code using OpenMP

By Erik Niemeyer (Intel Corporation) and Ken Strandberg (Catlow Communications*)

What do you do when parallelism opportunities seem to elude you? How do you find those opportunities and what methods are best to parallelize them in your existing code? This paper looks at identifying non-trivial parallel opportunities in existing serial code and how you can turn them into performance-optimized parallel code.
Table of Contents

Introduction........................................................................................................................................3
Basic Parallelism Concepts ................................................................................................................3
  Objectives of Parallel Programming...............................................................................................3
  Parallelism vs. Concurrency ............................................................................................................3
Functional vs. Data Decomposition...................................................................................................4
Parallelism and Correctness ...............................................................................................................4
Parallelism Performance: Determining speedup and Amdahl’s Law.................................................4
  What it means................................................................................................................................5
Parallelization Methods.....................................................................................................................5
  MPI ..................................................................................................................................................5
  Explicit Threading ..........................................................................................................................6
Compiler-directed ..............................................................................................................................6
Parallel Libraries ..............................................................................................................................6
Your Next Steps – Finding Opportunities.........................................................................................7
Finding Parallelism Candidates........................................................................................................7
  Profiling Applications to Find Bottlenecks....................................................................................7
  The Method....................................................................................................................................7
Are Candidates Parallelizable? ..........................................................................................................9
  Loops (for, do, while, etc.)..............................................................................................................9
Do you have a sequential algorithm (i.e. Huffman encoding)? .......................................................11
Is it a function? ................................................................................................................................12
  Other conditions............................................................................................................................12
Conclusion ........................................................................................................................................12
References ..........................................................................................................................................12
Get involved in the Community..........................................................................................................12
Resources for Further Study ............................................................................................................13
About the Authors.............................................................................................................................13
**Introduction**

This article discusses the fundamentals of finding non-trivial opportunities for parallelism (threading) in existing serial code. Threading can deliver significant benefits in application performance by simultaneously executing large tasks, or large sections of major tasks, using multiple threads. Threading does, however, present specific challenges when attempting to retrofit existing serial code for multi-threaded execution.

Building well-structured parallel code is a broad topic with much writing devoted to it. This article is intended only as a primer for *adding parallelism to existing serial code*, using methods proven by Intel engineers.

The article is divided into three sections.

- A review of basic parallelism concepts
- A review of methods to parallelize existing serial code
- How to identify candidates in serial code for parallelization.

If you are familiar with parallelism concepts and parallel development, you can safely skip the first two sections and focus on section three.

**Basic Parallelism Concepts**

**Objectives of Parallel Programming**

Parallelized code runs simultaneously, using more compute resources (multiple cores, multiple execution engines in a single core, etc.) to accomplish more tasks in the same amount of time or to complete a fixed-size task in a shorter amount of time. Code running in parallel can be multiple instances of the same code working on different data, or different sets of code working on completely separate tasks. Depending on which type of parallelism is executing, the benefits are either improved turnaround times for fixed-size problems or increased capacity to handle larger problems in the same amount of time (throughput).

**Parallelism vs. Concurrency**

Given today’s processor microarchitectures, it’s preferable, when possible, to thread your code to execute concurrently and in parallel. While related, parallelism and concurrency are not synonymous. The impact of the two on coding decisions the developer makes is important to understand.

*Concurrency* happens when two or more threads are *in progress* at the same time. They are not, however, necessarily *executing* at the same time. They might both be in the pipeline, but not simultaneously using resources and thus not presenting the same risks and benefits parallel execution does.

*Parallelism* happens when multiple concurrent threads simultaneously *execute*. This condition presents a whole set of benefits and risks not present with concurrent execution.

When threads simultaneously execute, you have opportunities for increased performance and better compute resource utilization. But, multiple threads increase the complexity of an application and raise the difficulty of debugging; finding data races, deadlocks, and other parallel challenges takes time (although programming tools, like Intel® VTune™ Amplifier XE, help identify problems).
Using threading and parallelism efficiently may offer significant benefits in application performance and the ability to scale to the size of the compute problem.

**Functional vs. Data Decomposition**

The objective of parallelizing code is frequently to complete single tasks in the smallest amount of time. You can achieve this by breaking the work into smaller pieces and applying resources to complete each in parallel. But, how do you break up the work? Are there a lot of different independent functions (tasks) that can be done simultaneously? Is there a lot of data that can be partitioned into sizeable workloads? Are both present?

These questions help to identify the type of inherent parallelism present in the code, which has a major impact on how the code must be parallelized.

**Functional decomposition** divides the work into functional differences; it is best used when the amount of work scales well with the number of independent tasks. Functional decomposition is also referred to as task-based parallelization. For example, a single person building a house results in the longest time to complete the building. It is much more efficient and faster to apply skilled professionals, each doing their own tasks, possibly in parallel. For example, concrete workers build foundations, carpenters frame the building, electricians wire the structure while plumbers install pipe, wallboard professionals cover the framing, finish carpenters install molding while tilers install countertops, etc. Separate activities done in parallel, then, minimize build time. An automobile assembly line is another example of functional decomposition. Different workers do different things and pass their part to the next: body and engine tasks are done in parallel and meet at some point to complete the build.

**Data decomposition** divides the work into independent chunks of data on which work can be done. This is best used when the amount of work scales with the amount of data. For example, grading school exams is much faster when multiple answer sheets are divided amongst multiple graders; ten graders can significantly cut the grading time of 1000 exams compared to a single grader (potentially in 1/10th the time). The search for parts of the Skylab space station after falling to Earth is another example of the data decomposition concept; using different searchers doing the same thing, the search field is broken into chunks. In a typical application, searching for a specific data pattern, such as eye color, in a massive amount of data, such as a picture of many people on the street, can be done more quickly by partitioning the data and applying the same search on each partition.

**Parallelism and Correctness**

While thoroughly parallelized code can deliver significant performance improvements, if it delivers wrong results it is not true to the algorithm and original intent of the code. A developer must choose algorithms that parallelize well or accept the incorrect result (which is acceptable in some cases). Some challenging ones include parallelized random number generation or sequential algorithms. Also, combining rounded, parallelized floating point calculations must be carefully watched in order to ensure correct results. Articles discussing these topics can be found on the [Intel® Software Network](https://software.intel.com/en-us/articles).

**Parallelism Performance: Determining speedup and Amdahl’s Law**

How much speedup can be achieved by parallelizing an application? For situations where reducing turn-around time is desired, Amdahl’s law is a great tool to help determine the speedup possible. It tells us that for a given compute problem, the amount you can reduce the execution
time of said problem is limited by the size of the serial portion of the code you will parallelize and the available compute resources.

Where:

\[ P = \text{portion of code that is parallelized (e.g. .3 or .7)} \]
\[ N = \text{number of processors available for the parallel portion} \]
\[ (1 - P) = \text{represents the serial portion of the code} \]

It’s easy to see that as \( N \) becomes very large, \( P/N \) approaches zero, and the result is dependent on the size of the serial portion.

**What it means**

To maximize speedup, you have to minimize the serial execution time of your code and maximize your parallel opportunities. This means not only parallelizing the obvious parts of your algorithms, but potentially redesigning your serial algorithms to be better expressed in a parallel way.

Keep in mind that while parallelized code often performs well, it is not always the best solution. There are situations where well-optimized serial code can perform better than a parallelized version of the same algorithm.

Now that we have reviewed basic parallelism concepts, let’s look at different methods for implementing parallelism, and then discuss how to find potential opportunities for parallelism.

**Parallelization Methods**

Several methods are available for parallelizing an application. The decision of which to use depends on several variables, including the following:

- The compute platform on which you’re running the code. Are you using a fully distributed compute platform, such as a large cluster of independent machines; a 32-, 64-, or 128-core symmetric multi-processor system, or a four-core workstation?
- How coarsely or finely grained your parallelization needs to be, which is often a reflection of your compute platform. This is analogous to the amount of work each thread does.
- The problem you are trying to solve. Are you parallelizing an embedded rocket control system or a drug modeling application running on a super computer?
- Performance objective. Are you primarily trying to increase throughput or decrease turnaround time?

The most common parallel coding methods are described next.

**MPI**

The Message Passing Interface (MPI) is a distributed, multi-threaded model used in high-performance computing (HPC) and widely distributed processing systems. The MPI API passes
information around the compute system using explicit messages. Every node runs standalone
and creates and responds to MPI messages.

MPI is used in coarsely grained parallelism. Examples of compute platforms suitable for MPI
include clusters of standalone servers interconnected by a high-performance communication
network, such as LoBoS* (Lots of Boxes on Shelves) at the National Institutes of Health.

In a distributed computing platform that uses MPI, communication amongst nodes impacts overall
system performance, because the network must pass the messages and data between nodes.
Memory architectures and partitioning also play a significant part in performance, since data
needed by one node might be stored on another node. Thus, data models and optimization for
communications need to be considered in overall program design to gain the best system
performance.

Explicit Threading

Explicit threading uses POSIX threads (Pthreads), Windows* threads, or other threading families
to explicitly create threads within the code. Threading is completely under the control of the
programmer, so code is threaded exactly to the developer’s needs. Explicit threading creates
complex code, but gives the developer greater control over how and when threads are created,
used, and destroyed. Explicit threading is used in both finely grained parallelism and coarsely
grained parallelism. Explicit threading can be combined with other forms, although it increases
the challenges for the developer and complexity of the overall application.

Compiler-directed

Compiler-directed, or automatic parallelization, uses an API, like OpenMP* or Intel* Threading
Building Blocks, to indicate within the code where threading opportunities lie. A compatible
compiler then automatically creates the necessary threads at compile time. Threading is less
complex for the developer, but the programmer has less control over how the threads are
implemented. It should be noted here that compiler-directed parallelism is extremely flexible and
easily used to retrofit existing serial code. As an engineer, it is the best way I have found to do
rapid-prototyping of parallel code. As such, this paper will focus specifically on finding the
opportunities most applicable to this method of implementing parallelism.

Parallel Libraries

Intel offers various libraries for parallel optimization on Intel* processors. These libraries do the
heavy lifting parallelizing algorithms for specific application domains:

Intel® Math Kernel Library  (Intel® MKL) is a library of highly optimized, extensively threaded math
routines for science, engineering, and financial applications that require maximum performance.
Core math functions include BLAS, LAPACK, ScaLAPACK. (ScaLAPACK is not supported under
Mac OS* X), Sparse Solvers, Fast Fourier Transforms, Vector Math, and more.

Offering performance optimizations for current and next-generation Intel* processors, it includes
improved integration with Microsoft Visual Studio*, Eclipse*, and XCode*. Intel® MKL allows for
full integration of the Intel Compatibility OpenMP* run-time library for greater Windows*/Linux*
cross-platform compatibility.

Intel® Integrated Performance Primitives  (Intel® IPP) is an extensive library of multicore-ready,
highly optimized software functions for multimedia, data processing, and communications
applications. Intel® IPP offers thousands of optimized functions covering frequently used fundamental algorithms.

**Intel® Threading Building Blocks** (Intel® TBB) is a C++ template library that abstracts threads to tasks to create reliable, portable, and scalable parallel applications. Just as the C++ Standard Template Library (STL) extends the core language, Intel® TBB offers C++ users a higher level abstraction for parallelism. To implement Intel® TBB, developers use familiar C++ templates and coding style, leaving low-level threading details to the library. It is also portable between architectures and operating systems. With Intel® TBB, developers get the benefits of faster programming, scalable performance, and easier-to-maintain code.

**Your Next Steps – Finding Opportunities**

After reviewing the basics of parallelism and knowledge of some of the methods for implementing parallelism, we come to the most complex issue: *How do you find parallel opportunities?* In the next section, we will review how to find good candidates for parallelization and overcome common challenges to parallelizing these candidates.

**Finding Parallelism Candidates**

**Profiling Applications to Find Bottlenecks**

There may be many places to parallelize your code, but will the invested efforts return worthwhile performance improvements? This is often difficult to determine until you profile the code to see where the processor spends its time. Profiling with a performance analyzer, such as the **Intel® Performance Bottleneck Analyzer** (Intel® PBA) will help you find where the performance bottlenecks occur. From there you can determine where to work on optimizations and if opportunities exist for parallelization or algorithm optimization.

**The Method**

There are many papers and articles describing various methods for finding parallelism candidates in existing serial code. In practice, as an engineer looking for these opportunities, I have found the simplest approach is the best. Look for the *hot code*. Hot code is defined as sections of an application where significantly more time is spent than in other sections. This is commonly, but not always, found in the body of loops.

This method involves using some type of sampling toolset, like the Intel® VTune™ Amplifier, Intel® Performance Tuning Utility, or **Intel® Performance Bottleneck Analyzer**, to find the hot code.

To collect this data, you must have a representative workload that exercises the application in a realistic way; for more information on creating a good workload, see this article: [http://software.intel.com/en-us/articles/the-three-stages-of-preparation-for-optimizing-parallel-software/](http://software.intel.com/en-us/articles/the-three-stages-of-preparation-for-optimizing-parallel-software/).

The aforementioned toolsets have detailed tutorials to get you started with basic analysis once you have a good workload (see the Reference section below for links to these tools).

Once you have a good workload and a toolset to collect the data, finding candidates for parallelism is very straightforward, just look for the hottest code. A report of code execution on an example workload from the Intel® PBA is shown in Figure 1.
Once you have located your hot code, you need to categorize the hot code by the time (or cycles) spent in it and the functions associated with it. Again, the best candidates are often in the hottest code. A common practice is to start with the top five hot sections of code. For ease of development, it is often easier to map these hot sections of code to their associated parent functions. See Figure 2.

With some candidates identified, the next step is to determine whether they can be parallelized.
Are Candidates Parallelizable?

There are many ways to express parallelism in existing serial code. But, by nature, iterative operations lend themselves to parallelization more so than other types. There are also myriad ways to implement multithreading, as highlighted in this article. However, in the interests of brevity, the examples in this section will focus only on the use of OpenMP compiler-directed pragmas. As previously mentioned, OpenMP is lightweight, portable, and easily integrated into existing serial code in an opportunistic way.

Loops (for, do, while, etc.)

Loops are without question the best candidates for adding parallelism to existing serial code. This is chiefly because loops are predisposed to allow easy application of ‘worksharing’ constructs. Worksharing simply means splitting up a large existing problem into multiple parallel threads of execution and distributing the work amongst the threads. In the case of loops, this is a common example of data decomposition (mentioned above).

A simple for loop

Let’s consider whether the following loop can be parallelized.

```c
1  for(i = 0; i < N; i++)
2       c[i] = a[i] + b[i];
```

There are several questions that must first be answered about the loop to determine its suitability for parallelization. Most important is this:

Is the current iteration independent of the previous iteration(s)?

Meaning, in the current iteration, there are no dependencies on any value from a previous iteration. In the above example, the sum of c[i] is never referenced outside the iteration it is assigned in (within the loop body). Therefore, there are no loop dependencies that would pose challenges to parallelizing this code.

Using OpenMP, the loop could be parallelized with the following change:

```c
1  #pragma omp parallel for
2  for(i = 0; i < N; i++)
3       c[i] = a[i] + b[i];
```

The omp parallel for directive instructs the compiler to distribute loop iterations within a team of threads that encounters this worksharing construct.

Assuming N=12 and there are 4 threads in a team where this parallel region resides, the above pragma addition and a compatible compiler would generate the following behavior.
1. The number of loop iterations, N, is divided by the number of threads, 4, to arrive at a unit of work for each thread, which is 3.
2. A different set of iterations is assigned to each thread. So, one thread is assigned to work on iterations 1-3, the next thread works on iterations 4-6, etc.
3. At the end of the parallel region, the threads rejoin and a single thread exits the parallel region.

Our simple change would cut the execution time significantly. The parallelized execution time for this loop would be about 1/4 of the serial time – an improvement of about 4x. Even if you are unfamiliar with OpenMP, it should be obvious that pragma-based parallelization is extremely easy to integrate into existing serial code with a significant potential benefit.

A more complex loop

Now, let’s look at a more complicated loop, one which poses some challenges to parallelization, and see how to overcome the challenges. This loop is an example of numerical integration, where the goal is to compute the value of Pi. We’ll do this using typical code for a basic integration.

This code computes the integral from 0 to 1 of the function. Some of you may remember from calculus what this integral evaluates to. For those that are a little rusty in the math department, this integral on the range 0 – 1 yields an approximation of Pi.

To approximate the area under the curve – which approximates the integral – we will have many small areas (many = num_steps) that we add up. Each area will be “step” wide, which will be the reciprocal of num_steps. Then we add the small area we just calculated to a running total of all the areas we have computed up to this point.

Here is the code:

```
1 static long num_steps=100000;
2 double step, pi;
3 void main(){
4  int I;
5   double x, sum = 0.0;
6   step = 1.0/(double) num_steps;
7   for (i=0; I <num_steps; i++){
8      x = (i +0.5) * step;
9      sum = sum + 4.0/(1.0 + x * x);
10  }
11  pi = step * sum;
12 }
```

First, we need to identify any challenges to parallelization. In this example, there are several, and, fortunately, there are solutions to each one:
We need to maintain private (non-shared) copies of \( x \), because \( x \) is used in each iteration to calculate the partial sum (line 9 above).

Each thread will need to accumulate its own \( \text{sum} \) before passing it off to the final \( \pi \) calculation (line 11 above). We will use what is called a reduction clause in OpenMP, which gives each thread its own private copy of \( \text{sum} \).

At the end of the loop, the code needs to perform some operation on each thread's \( \text{sum} \) value. In this case, we are computing partial sums, so the “+” operator is used with the reduction clause in 2 to sum the partial sum from each thread.

Here is the code with the necessary omp pragma:

```c
1 static long num_steps=100000;
2 double step, pi;
3 void main(){
4   int I;
5   double x, sum = 0.0;
6   step = 1.0/(double) num_steps;
7   #pragma omp parallel for private(x) reduction(+:sum)
8   for (i=0; I <num_steps; i++){
9     x = (i +0.5) * step;
10    sum = sum + 4.0/(1.0 + x * x);
11  }
12  pi = step * sum;
13 }
```

At this point, if you're confused by the OpenMP jargon, don't worry about it. The important concept here is that this loop has some serious loop-dependent challenges, but we can still parallelize it by managing those dependencies. This is made very easy for us with OpenMP; it is also possible to do this with traditional, explicit threading implementations.

**Do you have a sequential algorithm (i.e. Huffman encoding)?**

Sequential algorithms are one of the single biggest challenges to parallelizing existing serial code. By definition, a sequential algorithm has to be processed in a certain order to obtain the correct result. A classic example of such an algorithm is Huffman encoding. This is commonly used in codecs and creates a dependency for the current frame on previous frames with the purpose of reducing the bitrate of encoded content. Unfortunately, this poses some serious challenges for parallelizing the code. While there are methods for overcoming these limitations for specific algorithms, they greatly exceed the scope of this article. For simplicity, when attempting to thread a sequential algorithm, try to determine if the algorithm can be changed to a more parallel-friendly one or move on to other candidates.
Is it a function?

Often, hot code will comprise an entire function. When this function is called iteratively (within a loop body), or independent of other hot functions, it may be possible to parallelize it using functional decomposition. This involves spawning a new thread to execute the function, allowing the parent thread to continue what it was doing (assuming the task the function performs is not gating to continued loop execution). The classic example of this scenario is the automatic spell checker feature of a word processor. The word processor runs a loop waiting for user input. When it receives input, a separate thread within the loop begins scanning the word you are typing, while the parent thread continues to accept your input.

Other conditions

There are many other operations that lend themselves to parallelization. While a detailed discussion of them is beyond the scope of this article, there is one that is worth mentioning. Often, in serial code, sections exist where tasks are performed sequentially, but not out of any innate requirement to do so.

Conclusion

Quite often many opportunities to parallelize serial code do exist, and threading the code can offer significant performance benefits. Recognizing and taking advantage of these opportunities require you to understand parallelism concepts, the different programming methods that can be used to implement parallelism, and how to identify these opportunities.

By using available profiling and threading tools on the market, such as Intel® Performance Bottleneck Analyzer, you can quickly identify threading opportunities in serial code. Then, after identifying potential challenges to threading the serial portion, threading methods, such as OpenMP®, can help you quickly parallelize your code. While there are challenges to parallelizing serial code, there are also effective techniques and threading methods to overcome those challenges. The result is a well-tuned, high-performance parallel application.

References

The tools referenced in this article can be found here:

- Intel® VTune™ Amplifier
- Intel® Performance Tuning Utility
- Intel® Performance Bottleneck Analyzer
- Intel® Threading Building Blocks

Get involved in the Community

The best way to learn is to share with others on the same path. Please tell us about your efforts in creating parallel applications, including what has worked and what hasn’t. Connect with industry peers as well as Intel experts to help resolve any outstanding issues you may encounter (or to help someone else out): Community Forum: Threading on Intel® Parallel Architectures
Resources for Further Study

This article is only an overview of the concepts, processes, and methods needed to identify parallelism candidates. Here are a few other resources for studying this topic at a deeper level:

Intel® Parallel Programming and Multi-Core Developer Community provides cutting-edge documentation, tools, and community resources like blogs and forums for the software industry.

Intel® Software Development Products streamline the process of coding and optimizing software, with special attention given to effective threading throughout the software lifecycle.

Technical Books for Multi-Core Software Developers is a list maintained by the Intel® Software Network that identifies some of the best resources available.

Parallel Programming Glossary of Technical Terms is an Intel® Software Network public wiki that extends your vocabulary using the community’s collective intelligence.

About the Authors

Erik Niemeyer is a Software Engineer in the Software & Solutions Group at Intel Corporation. Erik has been working on performance optimization of applications running on Intel microprocessors for over a decade. Erik specializes in rapid parallel development and micro-architectural tuning. When Erik is not working he can probably be found on top of a mountain somewhere. Erik can be reached at erik.a.niemeyer@intel.com.

Ken Strandberg is principal of Catlow Communications*, a technical marketing communications firm (www.catlowcommunications.com). Mr. Strandberg writes a wide range of technical marketing and non-marketing content, video and interactive scripts, and educational programming for emerging technology companies, Fortune 100 enterprises, and multi-national corporations. He writes across a broad range of hardware and software industries. Mr. Strandberg can be reached at ken@catlowcommunications.com.
**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, 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 #20101101