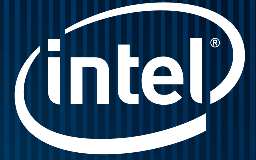
# THE PARALLEL UNIVERSE



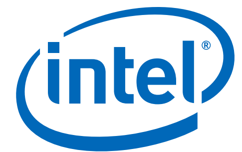

Issue 7
June 2011

## Intel® Cilk™ Plus:
### A C/C++ Language Extension for Parallel Programming
**By Robert Geva**

### Easing the Performance Analysis of Serial and Parallel Applications
**by Levent Akyil**

### Three Flavors of 'for' Loops with Intel® Parallel Building Blocks
**by Noah Clemons**

DEVELOPER ROCK STAR:
**Robert Geva**



# Read. Watch. Learn.

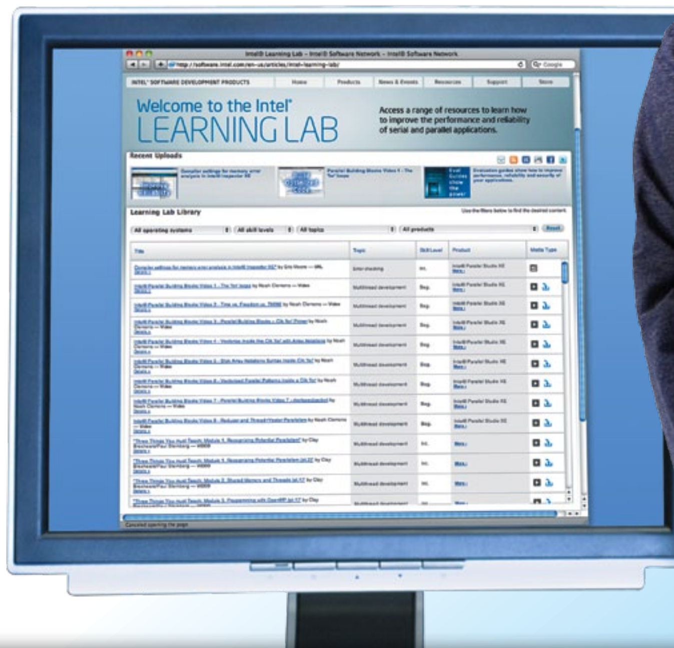
## Intel® Learning Lab puts high performance at your fingertips

Access a broad array of resources addressing key C/C++ and Fortran development issues and a host of Intel® Software Development Products. The robust repository offers everything from white papers to webcasts to videos for a range of experience levels.

Improving the performance and reliability of serial and parallel applications requires information. You'll find that information on Intel® Learning Lab.

VISIT INTEL LEARNING LAB TODAY

**Steve Lionel**
*Senior Member, Technical Staff, and Learning Lab contributor*



## Intel Learning Lab Resource Spotlight

In this three-part series, Dr. Clay Breshears of the Intel® Academic Community explores why parallelism is important, how to recognize where it can be introduced, and basic methods for addressing parallelism in code.

WATCH NOW

**To see a complete list of all available resources, visit Intel Learning Lab at the link above.**

Rock your code.
**ROCK YOUR WORLD.**

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.
© 2011, Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries. *Other names and brands may be claimed as the property of others.

# CONTENTS

**Letter from the Editor**
**Specifics:** FLOP Count and Parallel Programming, BY JAMES REINDERS .......... 4

Reinders, lead evangelist and director of Intel® Software Development Products, shares answers to three of the most common questions he receives, addressing FLOPS and Intel® VTune™ Amplifier XE, Intel® Cilk Plus, and Intel® Parallel Building Blocks (Intel® PBB) "parallel for."

**Intel® Cilk™ Plus:** A C/C++ Language Extension for Parallel Programming, BY ROBERT GEVA .......... 6

Intel offers new products for parallel programming based on the new programming model called Intel® Parallel Building Blocks (Intel® PBB).

**Three Flavors of 'for' Loops with Intel® Parallel Building Blocks (Intel® PBB),** BY NOAH CLEMONS .......... 12

Each of the models in Intel Parallel Building Blocks offers a different kind of 'for' loop. Learn why it is important to understand the build environment, type of parallelism it represents, and level of parallel abstraction before choosing a model.

**Easing the Performance Analysis of Serial and Parallel Applications,** BY LEVENT AKYIL .......... 16

Intel® VTune™ Amplifier XE is a powerful performance analysis tool that helps software developers identify issues in their applications. Explore how its improved and intuitive user interface performs powerful performance analyses with just a few mouse clicks.

**Case Study: Massachusetts General Hospital*,** BY BEVIN BRETT .......... 26

Follow the experiences of and lessons learned by developers at Massachusetts General Hospital and Intel as they identify, prioritize, and make changes to the C++ code to improve the serial algorithms and introduce parallelism to benefit virtual colonoscopies.

© 2011, Intel Corporation. All rights reserved. Intel, the Intel Logo, Intel Cilk, Intel Core, Intel Xeon, MMX, and VTune are trademarks of Intel Corporation in the U.S. and other countries. *Other names and brands may be claimed as the property of others.

**Sign up for future issues | Share with a friend**

***The Parallel Universe*** is a free quarterly magazine. **Click here** to sign up for future issue alerts and to share the magazine with friends.

# SPECIFICS: FLOP COUNT AND PARALLEL PROGRAMMING

## LETTER FROM THE EDITOR

**James Reinders** Chief Software Evangelist at Intel Corporation. His articles and books on parallelism include *Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism*, which has been translated into Japanese, Chinese, and Korean. Reinders is also widely interviewed on the subject of parallelism.

**This issue** gives us three nice pieces that answer common questions I hear all the time:

1. How do I count FLOPS (FLoating-point OPerationS) with Intel® VTune™ Amplifier XE?
2. What is Intel® Cilk™ Plus?
3. Intel® Parallel Building Blocks offers "parallel for" in different flavors—can you tell me more?

## 1. FLOPS

You might be happy to get a single number after you run the program and have Intel VTune Amplifier XE report something like "Your program used 12,506,718,902 floating-point operations." In my experience, that turns out to be pretty unsatisfying because you'll want to know more. Questions like, "How on earth did I use that many FLOPS?" or, "Where did the FLOPS get used?" come up quickly. Fortunately, Intel VTune Amplifier XE is prepared to answer them down to the precise thread and statement line in your program or aggregate them at thread and/or function levels. Also, Intel VTune Amplifier XE estimates the FLOPS by statistical sampling so that it is relatively non-intrusive while it gathers such detailed information about behavior on the system in a single run.

Performance-minded programmers often want to see how long-latency FLOPS are impacting their application. Operations including divide and square root are particularly rewarding to eliminate or reduce because of how much time they take to execute. Intel VTune Amplifier XE can help pinpoint where they are, and how often they occur.

There are many types of FLOPS and some come from using vector instructions such as streaming SIMD extensions (SSE) or advanced vector extensions (AVX). Operations done in vector instructions can improve performance because they may do multiple FLOPS per instruction. With the detailed information you can find using Intel VTune Amplifier XE you can decide how much performance you may be gaining or losing through the use or non-use of vector instructions.

If you are performance tuning a floating-point-intensive program, I think this issue's article on the subject will help inspire you to utilize Intel VTune Amplifier XE.



For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

“Operations done in vector instructions can improve performance because they may do multiple FLOPS per instruction.”

## 2. Intel Cilk Plus

Cilk originated in the mid-1990s at M.I.T. and has evolved to Intel Cilk Plus today and is supported on Windows*, Linux*, and soon on Mac OS* X, too. Intel Cilk Plus represents a very simple but powerful way to specify parallelism in C and C++. The simplicity and power come, in no small part, from compiler technology. The compiler allows very simple keywords to slip into existing programs and modify a serial application into a parallel application.

Cilk started with only two keywords—Cilk spawn and Cilk sync. To purists, this was enough and the simplicity seemed unbeatable. Having just the ability to send a function running in the background (a separate thread), and the flexibility to wait for all the spawned functions to finish with a sync, was a great start.

Over time, three more things have been added to make Intel Cilk Plus. The first was Cilk ‘for.’ Transforming a loop into a parallel loop by simply changing ‘for’ to “cilk_for” is so easy to learn and use, it has proven irresistible. Additionally, the alternative of spawning each iteration imposes too much overhead by requiring every iteration to be a task. Giving the compiler the information that the iterations of a loop may be done in parallel is enough for the compiler to produce code that “does the right thing” at run time. If the program is run on a single core, the need for iterations being broken into multiple tasks is not the same as it would be running on a quad-core system.

The second thing added to Cilk was hyper-objects, which are not covered in the article. They would make an excellent future topic for the magazine. Hyper-objects are already addressed in the compiler documentation from Intel if you cannot wait. Hyper-objects help break up a shared variable into private copies to allow parallelism without bottlenecking on a shared variable; they also allow specification of reduction operations. Sound complicated? Well, the beauty of hyper-objects is that they make it look simple and they are very easy to learn. We’ll definitely have to include an article on them in the future, but you’ll want to start with the knowledge in this issue.

The third thing brought to Intel Cilk Plus was the ability to specify data-parallel operations explicitly with new vector and array notations for C and C++. Take as an example the writing of a loop such as the following:

```
for (i=0; i<10000; i++) {
  a[i] = b[i] + c[i];
}
```

It is messy when you consider that what you really want to say is this:

```
a[:] = b[:] + c[:];
```

Now you can do just that. Of course, there is more detail in the article about Intel Cilk Plus later in this issue, including how to handle more complex slices of arrays and elemental functions.

The functionality of Intel Cilk Plus will appear in more compilers and standards in the future because the time has come.

## 3. Looping in Intel Parallel Building Blocks (Intel PBB)

Looping is fundamental, and how loops are made parallel in Intel PBB is explored in this issue. I think you’ll find it educational, but also reassuring because the variations are minor and intuitive.

Thank you for having a look as we explore these topics in more detail to both educate and whet your appetite for more. I hope you’ll find we did both.

Enjoy!

**James Reinders**
June 2011

Sign up for future issues | Share with a friend

# Intel® Cilk™ Plus:

## A C/C++ Language Extension for Parallel Programming

**by Robert Geva**
*Principal Engineer, Intel Software and Solutions Group*

Intel offers new products for parallel programming based on Intel® Parallel Building Blocks. Explore the benefits of one of its key tools, Intel® Cilk™ Plus.

### Editor's Note:

Robert introduces the Cilk keywords with the names _Cilk_spawn, _Cilk_sync, and _Cilk_for. Most developers, including the next article's author Noah and myself, prefer to spell them cilk_spawn, cilk_sync, and cilk_for by having #include <cilk.h> in our program. The versions with leading underscores are the real keywords, because the compiler does not create these new keywords in the normal space where it can theoretically conflict with a valid name. Instead, it leaves the nicer names to the cilk.h include file. Robert decided to expose the real names. Now you know and the choice is yours.



For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

Sign up for future issues | Share with a friend

**We are now well into the era** in which multicore processors are commonplace, not just in servers, but also in mobile devices, laptops, and PCs. Intel® processors are, increasingly, multicore in nature, offering use-benefits such as improved application performance, usability, power-reduction, and more. To help developers write code for these systems, Intel continues to invest in standards such as OpenMP* and MPI*. At the same time, Intel is offering new products for parallel programming, based on the new programming model called Intel® Parallel Building Blocks (Intel® PBB). Intel PBB currently consists of the following three components:

1. Intel® Cilk™ Plus: This is a language extension that provides both task and data parallelism constructs. It is currently implemented and supported in Intel® C++ Compiler. This article provides an introduction to Intel Cilk Plus.
2. Intel® Threading Building Blocks (Intel® TBB): Intel TBB provides both low-level tasking primitives and high-level parallel algorithms. Intel TBB is implemented as a compiler-independent template library for C++. It is available as a supported product and as open source.
3. Intel® Array Building Blocks (Intel® ArBB): Intel ArBB is a C++ library that provides a generalized vector parallel programming solution that frees application developers from dependencies on particular low-level parallelism mechanisms or hardware architectures. It is currently available in beta-test form.

## Programmer focus

The Intel PBB programming model takes into consideration the programmer perspective and the multiple levels of parallel resources provided in the processor hardware.

The processor architecture provides multiple parallel resources, including multiple cores, vectors, hyper-threading, and caches. Different programmers have different levels of interest in writing code to take advantage of these resources. Intel PBB, and in particular Intel Cilk Plus, support two approaches. First, programmers may choose to explicitly describe the parallel work at the core level, using tasks, and within each task write explicit vector-level parallelism. Second, programmers can let their compiler and system software figure out how processing should be done. This second approach assumes that the work to be done in parallel is clearly defined and that the data on which the work will be done is clearly identified.

In addition to utilization of HW resources, a central part of value added by Intel PBB in general, and by Intel Cilk Plus in particular, has to do with "composability." Many applications are composed of independently written components, which can be homegrown or come from third parties (such as libraries). Typically, the components (and their designers) communicate with each other through higher-level interfaces rather than at a lower, implementation-oriented level. To allow that level of collaboration in a parallel program, the task scheduler has to allow composability (i.e., when modules are combined to form the application, they continue to work and perform well).

How does Intel Cilk Plus support composability? The Intel Cilk Plus language, as well as the other components of Intel PBB, utilize user-mode work-stealing task schedulers. Work stealing is a task-scheduling technique that is characterized by a mechanism in which units of work can migrate from the worker that generates them to another worker that will execute them—if the second worker has no work of its own to process. A worker that runs out of work steals work from a victim that has a queue of work items. In contrast, there is no mechanism by which a worker that encounters opportunities for parallel work can delegate that work to other workers. The precise details of the scheduling mechanism are out of the scope of this article, which focuses on the language aspects. However, the work-stealing technique is at the heart of the composability property, which allows for modular implementation of parallel programs.

## Task parallelism with Intel Cilk Plus

The Intel Cilk Plus language provides two constructs for task level parallelism. The first includes two keywords: _Cilk_spawn and _Cilk_sync. **Figure 2** and **Figure 3** compare an implementation of the recursive implementation of the well-known Fibonacci code in serial C against a parallel implementation, which was derived from the serial code by adding the new keywords:

## Intel® Parallel Building Blocks

Comprehensive tools to deliver outstanding app performance

Intel® Parallel Building Blocks

| | Intel® Cilk™ Plus | Intel® Threading Building Blocks | Intel® Array Building Blocks |
|---|---|---|---|
| What it is | Language extensions to simplify parallelism | Widely used C++ template library for parallelism | Sophisticated C++ library for data parallelism |
| Features | • 3 simple keywords<br>• Hyper-objects<br>• Array notations<br>• Sequential semantics<br>• Vectorization support | • Parallel Algorithms<br>• Data Structures<br>• Scalable Memory Allocator<br>• Task Scheduler<br>• Synchronization Primitives | • Uses both SIMD and multiple cores for data parallelism<br>• Safety guarantees to avoid data races and deadlocks<br>• Vectorization support |
| Reasons to Use | • Simplest way to parallelize your code<br>• Serial semantics + low overhead = powerful solution<br>• Supports C & C++; Windows* and Linux* | • Rich feature set for general purpose parallelism<br>• Available as open source or commercial<br>• Supports C++; Windows, Linux, Mac OS*, other OSs | • Sophisticated data parallel support includes vectorization, dense, sparse and irregular matrix support<br>• JIT & VM technology = flexible and powerful<br>• Supports C++; Windows & Linux |

MIX AND MATCH TO OPTIMIZE YOUR APPLICATION'S PERFORMANCE

**Figure 1:** Summary of Intel® Parallel Building Blocks



For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

### Serial Code

```
int fib (int n)
{
  if (n <= 2)
      return n;
  else {
      int x,y;
      x = fib(n-1);
      y = fib(n-2);
      return x+y;
  }
}
```

**Figure 2:** Serial implementation of Fibonacci code

### Serial Code made parallel with Intel® Cilk™ Plus keywords

```
int fib (int n)
{
  if (n <= 2)
      return n;
  else {
      int x,y;
      x = _Cilk_spawn fib(n-1);
      y = fib(n-2);
      _Cilk_sync;
      return x+y;
  }
}
```

**Figure 3:** The same code made parallel using Intel® Cilk™ Plus keywords. Note that no code was changed. Only the Intel Cilk Plus keywords were inserted.

Notice that the addition of the keywords is non-intrusive. Changing the serial program to a parallel program only requires insertion of the keywords as appropriate. It does not require changes to the underlying program. The insertion also does not impact the readability and maintainability of the program. The original serial code is easy to see and the program logic is as evident as in the original. This observation highlights one of the main design principles of Intel Cilk Plus: It is designed for parallelizing existing serial C/C++ code. Thus, a benefit of Intel Cilk Plus is the ability it gives developers to easily add parallelism to code by making minimal changes, while providing strong guarantees for serial equivalence.

## The _Cilk_spawn keyword

Let us now take a closer look at the _Cilk_spawn keyword, which applies to a function call, such as fib(n-1) in **Figure 3**. It means that the spawned function can execute concurrently with the remainder of the enclosing function (fib(n) in this example). The enclosing function is termed a "spawning function" or the parent, and the function being spawned is the child. The _Cilk_sync keyword means that the parent has to wait until all child tasks return control to the parent.

Note that this mechanism reuses the concept of a function, which is a well-understood concept in C/C++ programming, both as a parallel region (on the parent side) and as the unit that can be spawned (on the child side). Both of these serve the purpose of bringing parallelism to existing programs with minimal effort. The reuse as a parallel region alleviates the need to introduce a new syntactic construct that would serve as a parallel region. The reuse of a function as the unit defining what can be spawned may be viewed as a limitation. In parallelizing an existing program, the programmer may want to indicate that a smaller lexical scope, such as a statement or a loop, can execute concurrently with another statement.

However, by using functions as a spawn unit, the data model remains well understood to the programmer and the language extensions do not have to introduce new rules, which have the potential to introduce programming errors. In the current model, it is clear to the programmer which variables are in the scope of the spawning functions, which variables belong to the child function, and how arguments are being passed. Indeed, argument passing is very much conformant to the way it is done in the underlying language. The values of the arguments are being evaluated by the parent task before detaching the child and enabling concurrency. In the fib(n-1) example in **Figure 3**, the value of n-1 is computed by the parent before the child is spawned. A less trivial example can be observed in **Figure 4** where the computation of the arguments is more involved.

The arguments to the function f() in **Figure 4** are themselves function calls. If the operation of _Cilk_spawn applied to the whole expression, then g() and h() might have been allowed to execute concurrently with respect to each other, as well as with respect to the spawning function. In a case of a data race between the code within the function g() and h(), there is no place to insert a synchronization directive.

Arguably, the most important attribute in the behavior of _Cilk_spawn, in support of equivalence to serial execution, relates to the asymmetry between the two parts of the parent at a _Cilk_spawn point. A _Cilk_spawn does not enforce parallel execution, but instead provides an opportunity for parallel execution. The worker whose execution reached the _Cilk_spawn keyword will continue processing one of the two branches identified by the keyword, while the remaining branch is detached and put in a queue for later execution. As described earlier, the work item in the queue may or may not be picked up for execution by another worker executing on another core.

```
val = _Cilk_spawn f(g(x),h(y));
```

**Figure 4:** A less trivial example of _Cilk_spawn.

It might seem intuitive to think that the spawned function is the one that is queued for later execution, and that the worker executing the parent will continue executing. But, in fact, it is the other way around. The worker that hit the _Cilk_spawn keyword will detach and be queued up to continue executing its work.

Why does this asymmetry matter? The reason is that the order of evaluation corresponds to the order of evaluation of a function call in a serial C/C++ program. The spawning function will continue to execute the child, as is the case in a regular function call, and if the continuation is not stolen by another worker. Then, it will return, pop the continuation as a work item, and continue executing it. The important thing to note is that this order of evaluation is the same as that of a function call in a serial program.

Sign up for future issues | Share with a friend

It is also important to note that Intel Cilk Plus keywords can be elided. For example, the programmer can use the C/C++ preprocessor and #define them away using the syntax in **Figure 5**.

```
#define _Cilk_spawn
#define _Cilk_sync
```

**Figure 5:** Intel® Cilk™ Plus keywords can be elided to enable correct serial execution of apps built with compilers that do not support Intel Cilk Plus keywords. This helps keep your source code portable.

Compiling/building an application that uses the Intel Cilk Plus keywords, which elides them as shown, will produce a valid serial application. As long as the parallel version of the program does not introduce data races, the serialization of the program will be semantically equivalent to the parallel execution of the program and will produce the same results.

## The _Cilk_sync keyword

The _Cilk_sync keyword in the example of the fib() function in **Figure 3** is necessary to guarantee that the values in the variables x and y are updated by the asynchronous execution of fib(n-1) and fib(n-2) before they are used. In addition to the explicit form of the keyword, there is an implicit _Cilk_sync at the end of every spawning function. The implicit keyword is, in fact, inserted by the compiler.

The effect of the implicit _Cilk_sync is that the spawning function executes as long as any of the child tasks it spawned are executing. One benefit of this property is that it allows the programmer to continue viewing functions as units of work, and to see when a function returns all its work and when it is done. A more practical benefit is that if the spawning function is passing an address of any of its stack variables to a child, and the child task may write onto that location, then it is guaranteed that the stack of the parent is in memory during the execution of the child.

## The _Cilk_for keyword

The second tasking construct provides the keyword _Cilk_for. It enables programmers to parallelize C/C++ for loops. Consider the statement in **Figure 6**.

```
_Cilk_for (int i = 0; i < N; ++i)  {
  body;
}
```

**Figure 6:** The _Cilk_for keyword

When this keyword is used, the compiler enforces a few restrictions on the 'for' construct. These include:

- The index variable (*i* in this example in **Figure 6**) appearing in all three expressions of the loop is initialized in the first instance.
- It is compared to a value that does not change within the loop.
- It is incremented by a value that does not change inside the loop.
- It cannot be changed within the body of the loop.

The effect of these restrictions is that when the loop is about to execute, its trip count is known to the runtime scheduler.

The execution of the _Cilk_for loop uses a divide-and-conquer approach. The worker whose execution reached the _Cilk_for construct computes the trip count. It takes the upper half of the iterations and puts it in its own work queue for later evaluation. It then continues to divide and conquer the lower trip count, until the trip count becomes sufficiently small (determined heuristically at run time). It then executes the iterations sequentially. Upon completion, the worker pops the next work item, which was posted last, from its own work queue. This will be the second set of iteration of the lower half. If another worker has an empty work queue, it might steal work from the core that created the list of work items corresponding to the loop iterations. This is how concurrent work is created.

If this happens, it will steal an item from the top of the queue that corresponds to a maximal number of iterations. This order of stealing work is the least likely to interfere with the cache locality of the worker from which the item was taken. It also provides the most amount of work to the thief using the least number of steals. Using a minimal number of steals provides work for all workers with minimal overhead. As long as work does not get stolen, the order of evaluation of the loop iterations executed by a single worker is exactly the same as the order in serial execution.

## Vector-level parallelism in Intel Cilk Plus

The Intel Cilk Plus language extension provides several ways to take advantage of the hardware-based parallelism available in Intel® multicore processors.

A serial 'for' loop can be changed into a parallel loop by changing the keyword 'for' to "_Cilk_for." This change tells the compiler that there is no ordering among the iterations of the loop. As described above, the compiler arranges batches of iterations to execute in parallel. In addition, it will attempt to vectorize the code within batches of consecutive iterations.

## Array notations

A new syntax provided with Intel Cilk Plus also allows simple operations on arrays. The C/C++ language standards do not provide ways to express operations on arrays. Instead, the programmer has to write a loop and express the operation in terms of elements of the arrays, creating serial order, which is sometimes unintended. The downside of writing loops with serial ordering is that in order to convert the serial loop to a vector loop, the compiler has to prove that the vector processing would be equivalent to the scalar processing implied by the loop and mandated by the language. In the majority of the cases, these proofs are bound to fail.

For example, when the program uses pointers to reference the array (a reasonable programming practice meant to allow for operations on any array, rather than write code to operate on a specific array), the compiler is unlikely to prove that two pointers point to areas of memory that are non-intersecting.

Writing a[:] = b[:] + c[:]; indicates to the compiler that the elements of the arrays "b" and "c" need to be added and that there is no order required among the addition operations. These semantics allow the compiler to always generate vector code, instead of generating scalar code and attempting to prove that vector code would be equivalent.


For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

## Intel Cilk Plus array section operator

**The Intel Cilk Plus language extensions define an array section operator whose syntax is array_base[start:length:stride] where the following is true:**

- array_base is any array expression allowed in C/C++, including arrays, pointers, and C99 variable-length arrays.
- "Start" is the first array element to which the section applies.
- Length provides the number of array elements.
- Stride is an increment between elements. The use of stride is optional. If it is not provided, the default value 1 is used.

The section operator can be applied to multidimensional arrays and can be intermixed with array subscripts, such as in a[0:5][3][0:8:2]. In this example, "a" has to be a three-dimensional array or a pointer to an array. The rank of the expression is the number of dimensions in which the section operator is used, rather than a subscript. In the example provided, the rank is 2.

A few intrinsic functions are provided for commonly used operations such as summation of an elemental of an array. A dot-product operation might be expressed as:

```
x = __sec_reduce_add(a[:] * b[:]);
```

**Figure 7:** A dot-product operation expression using _sec_reduce_add

The array notations might remind some programmers of arrays in Fortran 90. A significant difference is that the Intel Cilk Plus array notations expect that the arrays on the right-hand side of an assignment statement do not partially overlap the array being assigned to on the left-hand side. A violation results in undefined behavior. Therefore, unlike in Fortran 90, the compiler does not generate temporary arrays to hold the intermediate values of the right-hand side of an array expression. This change results in higher performance.

Writing an array expression can be a natural way to write the program when the programmer expresses the algorithm and the design at the level of operations on an array. For an existing serial program, a change to array notations might be intrusive. A less intrusive change is also available in Intel Cilk Plus in the form of #pragma simd. The pragma can be added to a loop to indicate that while the loop is written in scalar syntax, the implied serial ordering is not intended. Instead, the programmer's intent is for the compiler to generate vector code to implement the loop. As a language construct (rather than a performance hint), the pragma allows the compiler to generate vector code without having to prove that the vector code would be equivalent to scalar code, and scalar code is not the programmer's intent.

A third programming construct allows the programmer to write a scalar function in standard C/C++ and declare it as an "elemental function." When using an elemental function, the programmer's typical intent is to deploy it on many elements of arrays without prescribing an order of operations among the array elements. The simple example in **Figure 8** shows the use of elemental functions to perform element-wise addition of arrays.

The use of __declspec(vector) indicates to the compiler that the function "v_add" is intended to be used as an elemental function. The Intel® C++ Compiler will generate two versions of code for such a function. In addition to "standard" code, a vector version will be generated, which receives a vector of arguments for "x1" and a vector of arguments for "x2." It performs the operations of the function using the hardware vector registers across all the input arguments. This returns a vector of results instead of a single result. The function can be called in a scalar context. In such a circumstance, the compiler will translate the function call into a call to the standard, scalar function.

```
__declspec(vector)
float v_add(float x1, float x2)
{
  return x1+x2;
}
caller:
for (int j = 0; j < N; ++j) {
  res[j] = v_add(a[j],b[j]);
}
```

**Figure 8:** Example of the use of elemental functions to perform element-wise addition of arrays.

When it is called in a data parallel context, such as the example of the loop shown in **Figure 8**, the compiler will call the vector version. If the target of the compilation is a CPU supporting the XMM vector register, for example with the SSE2 instruction set extension, then the compiler determines that four consecutive values of the input arrays can fit within a register. It will generate a version of the function that operates on four consecutive array elements, and the function will be called N/4 time (or 1, 2, or 3 additional time in scalar version, if N is not divisible by four). The benefit, of course, is enhanced performance.

Instead of calling the function using a 'for' loop, the programmer may choose a "_Cilk_for" as the construct to call the function. In such a case, not only will the compiler call the vector version, it will also facilitate loop iteration execution by using multiple cores. With this use of the language construct, the programmer can get the combined benefit of both core- and vector-level parallelism.

## Summary

**Intel Cilk Plus is a language extension to C/C++. The main benefits include:**

1. It enables the application developer to use all parallel resources available in the HW, including cores, vectors, and caches.
2. It provides multiple levels of abstraction, enabling the developer to choose how parallel work can be done—by cores separately from work done by the vector—or they can choose not to program explicitly to the hardware and instead to indicate the intent for parallelism, allowing the compiler to map the operations to the hardware.
3. It supports composability. By using tasking, with a work-stealing scheduler, the extensions allow integration of an application from independently developed components (possible third-party libraries, or modules developed independently by sub-groups) without the need to coordinate the parallelism architecture between the component and without risk of hardware resource oversubscriptions resulting from the integration.
4. It simplifies adding parallelism to existing serial programs. Multiple attributes of the extensions support this, including the non-intrusive syntax, the ability to easily revert back to the serial program, the guarantee of serial semantics equivalence, and the low overhead of spawning a task. ☐

Sign up for future issues | Share with a friend

# THREE FLAV

## with Intel® Parallel Buildin

by Noah Clemons

## Each of the models in Intel Parallel Building Blocks offers a different kind of ‘for’ loop. Learn what you need to know before choosing a model.

A common performance challenge is when most of the processing time is being spent inside one or more ‘for’ loops. A programmer’s first reaction is usually to get the ‘for’ parallelized across some or all of the available cores. **Each of the models included in Intel® Parallel Building Blocks (Intel® PBB) offers a different flavor of the ‘for’ loop, bringing similarities and differences:**

- Intel® Cilk™ Plus : cilk_for
- Intel® Threading Building Blocks (Intel® TBB): parallel_for
- Intel® Array Building Blocks (Intel® ArBB): _for in conjunction with arbb::map()

Before choosing one, it is important to understand the build environment that each ‘for’ works in, what type of parallelism the particular ‘for’ represents, and the level of parallel abstraction offered by the particular ‘for.’ From there, the programmer can make a decision about which ‘for’ to use without reading the reference manual for each.

The ‘for’ loops within Intel PBB consist of the following:

- Data parallel and general-purpose parallelism solutions
- Language extensions and template library solutions
- Varying levels of API control from beginner to expert

| | Data Parallel or Task Parallel? | Language Extension or Template Library? | API Control |
|---|---|---|---|
| cilk_for | Task parallel — use array notations inside the loop to add data parallelism | Language extension | Beginner |
| parallel_for | Task parallel | Template library | Intermediate, but has expert options |
| _for + arbb::map() | Data parallel | Template library | Intermediate |

Table 1

```
cilk_for (int x = 0; x < 1000000; ++x) { … }
```

Figure 1

```
int x = 0;
cilk_for(int i = 0; i < N; i++)
     x++;
```

Figure 2



For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

# ORS OF 'FOR' LOOPS

g Blocks (Intel® PBB)

## cilk_for

The Cilk component of Intel Cilk Plus has a keyword style syntax that will spawn threads to execute the 'for,' as shown in **Figure 1**.

Using this loop has a few caveats:

- Any or all iterations may execute in parallel with one another.
- All iterations complete before the program continues.

There are also a few constraints:

- Programmers are limited to a single control variable.
- Programmers have to keep in mind that jumping to the start of any iteration happens at random.
- Iterations should be independent of one another.

In comparison with parallel_for and the Intel ArBB _for, this one has the least amount of control. The programmer is relying on Intel® Parallel Composer to make threading decisions. **Figure 2** simply increments a variable, but does so in parallel.

Using cilk_for indicates to the runtime that it can split the work of the 'for' into chunks that can be run on different CPU cores at the same time. There is no implicit vectorization just by using the keyword. However, it is recommended to use the array notations component in Intel Cilk Plus for vectorization. Since the cilk_for is such a simple keyword, there is not a great deal of flexibility for manipulating the parameters of the loop and how it is threaded. But as a consequence, cilk_for is very easy to learn and start coding with, making it the simplest way to parallelize an existing 'for' loop among the Intel® tools.

Note the potential data race problem on the increment of x, as one thread might read x but not increment and write it back to memory before another thread can load the value of x and increment it. One can use the Intel Cilk Plus reducer to resolve that data race, as in **Figure 3**.

Simply declaring a variable with the provided Intel Cilk Plus template class type allows one to safely increment the shared variable while still maintaining scalable performance. The set_value and get_value reducer member function allows access to the data in a reducer outside of the parallel region.

Note that in the simple example above, the compiler would have simply optimized away the 'for' loop if the cilk_for had not been used. Programmers frequently find situations where the cilk_for may be more useful in conjunction with vectorization. The following example is a blocked, vectorized, and threaded implementation of a vector dot product while only adding three more lines of code than the original implementation. Note in **Figure 4** that any possibility of a race condition on sum is prevented by use of the reducer.

```
#include <cilk\reducer_opadd.h>
cilk::reducer_opadd<int> x;
x.set_value(0);
cilk_for(int i = 0; i < N; i++)
     x++;
// Use x.get_value() to access x afterward
```

**Figure 3**

## parallel_for

Intel TBB parallel_for is well-suited for a build environment that allows for a template library with a high degree of freedom to implement task parallel algorithms. It is very convenient for those that already have a background in parallel programming. Most concepts learned in a parallel programming course or through on-the-job training are templatized for easy use within the API.

The parallel_for offers the greatest ability to manipulate behind-the-scenes details. While the focus is more on tasking rather than managing threads, experts still have the freedom to dig deep. Intel TBB can do an auto-grain size determination or it can be expressed explicitly as an option for fine-tuning. This control is best addressed by the example in **Figure 5**.

Sign up for future issues | Share with a friend

```
float sprod(float *a, float *b, int size) {
  float sum = 0.0;
  for (int i=0; I < size; i++)
    sum += a[i] * b[i];
  return sum;
}
```

```
  float sprod(float* a, float* b, int size) {
  int s = 4096;
  cilk::reducer_opadd<float> sum(0);
  cilk_for (int i=0; i<size; i+=s) {
    int m = std::min(s,size-i);
    sum += __sec_reduce_add(
      a[i:m] * b[i:m] );
  }
  return sum.get_value();
}
```

Figure 4

Notice that one uses templatized components to "build" a parallel 'for' loop with Intel TBB.

- ChangeArray class defines a for-loop body for parallel_for.
- The Intel TBB template blocked_range represents the 1D iteration space.
- As usual with C++ function objects, the main work is done inside operator().
- Finally, one can then invoke ChangeArrayParallel, which will call a template function parallel_for<Range, Body>: with arguments Range -> blocked_range Body -> ChangeArray.

In many scenarios, programmers can use some of these components to do for-style computations but not necessarily have to invoke a 'for,' as illustrated in the vector dot product example utilizing blocked_range in **Figure 6**.

```
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"
using namespace tbb;
class ChangeArray{
    int* array;
public:
    ChangeArray (int* a): array(a) {}
    void operator()( const blocked_
range<int>& r ) const{
        for (int i=r.begin(); i!=r.end();
i++ ){
            foo (array[i]);
        }
    }
};

void ChangeArrayParallel (int* a, int n )
{
    parallel_for (blocked_range<int>(0, n),
ChangeArray(a));
}
int main (){
    int A[N];
    // initialize array here…
    ChangeArrayParallel (A, N);
    return 0;
}
```

Figure 5

## Intel ArBB _for

Intel ArBB is the data parallel analogue of Intel TBB. It uses Intel TBB for threading and generates vectorized code through its own run time code generation process. Programmers think differently about 'for' loop computations than about the other Intel PBB models. Intel ArBB is an entire library and language to specify an arbitrary computation, and the key word there is *arbitrary*. It is much more high-level than Intel TBB. It is an entire programming platform when compared to Cilk's three keywords, and is well-suited for algorithm scientists that do not want to deal with specifics of either managing tasks or vectorization.

The Intel ArBB _for is similar to the C/C++ loop, but is not meant for parallel execution alone—it is a regular serial loop. Other parallel operations intrinsic to the API should be used, or the _for loop should be used in conjunction with arbb::map(). Using _for with arbb::map allows for both vectorization and threading without the programmer having to manage tasks.

To illustrate how many parallel operations are abstracted away, rendering a 'for' loop unnecessary, take a look at an Intel ArBB implementation of matrix/vector multiplication that works but is not actually parallelized, in **Figure 7**.

Except for the use of Intel ArBB types, operators, and keywords, this code looks very similar to the serial C implementation. However, just like the C version, it runs sequentially. This is not the right way to compose an efficient parallel program using Intel ArBB.

See **Figure 8** for a much better implementation that shows the simplicity of Intel ArBB syntax and expresses the intrinsic parallel nature of the algorithm.

Notice the use of container operators instead of scalar operators, as well as the use of collective operators. Not only is this code simpler, it also allows the Intel ArBB runtime to parallelize the computation through vectorization and/or multithreading.

So when should the _for loops be used?

The _for loop should be used in the following situations:

- Inside Intel ArBB functions
- To express serially dependent iterative computation. This is the case where a computation must be done incrementally, with the current step depending on the result of the previous step. A good example would be heat dissipation using an iterative stencil in **Figure 9**.

In this code, computing each stencil-based update step is parallelized through the use of the arbb::map() function. But the updating must be done multiple times, repetitively, in a sequence in order to compute the solution over time.

In conclusion, each of the 'for' loops represented in Intel PBB addresses different programmer needs, and we encourage you to find which one works best for your workload. □



For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

```
float sprod(float *a, float *b,
          int size) {
  float sum = 0.0;
  for (int i=0; i < size; i++)
    sum += a[i] * b[i];
  return sum;
}
```

```
  float sprod(const float a[], const float b[],
size_t n ) {
  return tbb::parallel_reduce(
    tbb::blocked_range<size_t>(0,n),
    0.0f,

    [=](
      tbb::blocked_range<size_t>& r,
      float in
    ) {
      return std::inner_product(
        a+r.begin(), a+r.end(),
        b+r.begin(), in );
      },
      std::plus<float>()
  );
}
```

Figure 6

```
void matvec_product(const dense<f32,
2>& matrix, const dense<f32>& vector,
dense<f32>& result)
{
    usize rows = matrix.num_rows();
    usize cols = matrix.num_cols();
    _for (usize i = 0, i < rows, ++i) {  //
SERIAL LOOP
        f32 sum(0.0);
        _for (usize j = 0, j < cols, ++j) {
// SERIAL LOOP
            sum += matrix(j, i) * vector[j];

        } _end_for
        result = replace(result, i, sum);
    } _end_for
}
```

Figure 7

```
void matvec_product(const dense<f32,
2>& matrix, const dense<f32>& vector,
dense<f32>& result)
{
    result = add_reduce(matrix * repeat_
row(vector, matrix.num_rows()));
}
```

Figure 8

```
void apply_stencil(dense<f64, 2>& grid, i32
iterations) {
    _for(i32 i = 0, i < iterations, ++i) {
        map(stencil)(grid);
    } _end_for
}
void stencil(f64& cell) {
    arbb::array<usize, 2> coord;
    position(coord);
    usize x = coord[0], usize y = coord[1];
    _if(x != 0 && y != 0 && x != WIDTH-1 &&
y != HEIGHT-1) {
        cell = 0.25 * (neighbor(cell, -1,
0) + neighbor(cell, 1, 0) +
 neighbor(cell, 0, -1) + neighbor(cell, 0,
1));
    } _end_if
}
```

Figure 9

"Using _for with arbb::map allows for both vectorization and threading without the programmer having to manage tasks."

Sign up for future issues | Share with a friend

# Easing the Performance Analysis of Serial and Parallel Applications

by Levent Akyil

Explore how the improved and intuitive user interface in Intel® VTune™ Amplifier XE performs powerful performance analyses with just a few mouse clicks.



For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

**Intel® VTune™ Amplifier XE** is a powerful performance analysis tool that helps software developers identify algorithmic and microarchitectural performance issues in their applications. Intel VTune Amplifier XE, with its improved and intuitive user interface, performs powerful performance analyses with a few mouse clicks. It brings new, innovative, easy-to-run analysis types for both algorithmic and micro-architectural performance analyses; some of the pre-defined analysis types for algorithmic tuning are lightweight hotspot analysis, hotspot analysis, concurrency analysis, and locks and waits analysis.

- Hotspot analysis helps the developer understand the application flow (i.e., call stack information) and identify the sections of code that took a long time to execute (hotspots) by leveraging a low overhead statistical sampling (a.k.a. user-mode stack sampling) technology.
- Concurrency and locks and wait analyses, similar to hotspot analysis, identify hotspot functions and call stacks to those functions, but additionally measure how an application utilizes the available processors on a given system by leveraging thread-profiling technology.
  - Concurrency analysis identifies where processor utilization is poor, and how and when threads are running, synchronizing, and waiting.
  - Locks and waits analysis helps identify the cause of the ineffective processor utilization. The most common problem for poor utilization is caused by threads waiting too long on synchronization objects (locks).
- Lightweight hotspot analysis is similar to hotspot analysis but instead uses hardware-based event-based sampling (EBS) technology to locate the hotspots in a given application. **Figure 1**

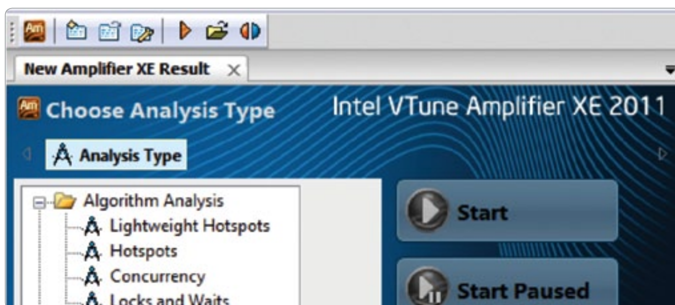



**Figure 1:** The predefined algorithm analyses: Lightweight Hotspot, Hotspot, Concurrency, and Locks and Waits.

For more advanced and deeper microarchitectural analysis, the tool is equipped with pre-defined analysis types, which use a performance monitoring unit (PMU) to sample processor events to identify micro-architectural issues such as cache misses, stall cycles, branch mispredictions, and many more. The advanced analysis types are defined for processor architectures such as Intel® Core™2 microarchiteture, Intel® Core™ microarchitecture (a.k.a. Nehalem, Westmere), and 2nd generation Intel® Core™ microarchitecture (a.k.a. SandyBridge). When these advanced pre-defined analysis types are used, the tool gives hints and suggestions by highlighting the problematic functions.

In this article, you will learn how some of the advanced features of Intel VTune Amplifier XE and EBS technology can help developers identify computational issues and estimate key performance metrics such as floating-point operations per second (FLOPS). FLOPS (also known as flops or flop/s) is a measurement that is heavily used in high-performance computing and is a common way of measuring the performance and computational capabilities of a given microprocessor. Other analysis types such as hotspots, concurrency, and locks and waits leverage user-mode stack sampling and API instrumentation technologies.

## Identifying computational issues

Performance tuning usually focuses on reducing the time it takes to complete a well-defined workload. Performance events can be used to measure the elapsed time; therefore, reducing the elapsed time of completing a workload is equivalent to reducing measured processor cycles (clockticks). The Lightweight Hotspot, which is a pre-defined analysis type of Intel VTune Amplifier XE, uses processor cycles and instructions retired[1] to analyze the application. The count of cycles, also known as clockticks, forms the fundamental basis for measuring how long a program takes to execute. The total cycle measurement is the start-to-finish view of the total number of cycles to complete the application of interest. In typical performance-tuning situations, the metric total cycles can be measured by the clockticks.

One of the goals when performing microarchitectural analysis and optimization is to identify cycles where no micro-operations are dispatched for execution. Micro-operations, also known as a micro-ops or μops, are simple microprocessor instructions used to implement more complex instructions.

Cycles where no μops were dispatched will be referred to as stall cycles and can be counted with the hardware PMU events as demonstrated in **Table 1**. These very stalls can turn the execution unit of a processor into a major bottleneck. The execution unit by definition is always the bottleneck because it defines the throughput and an application will perform as fast as its bottleneck. Therefore, it is extremely critical to identify the causes for the stall cycles and remove them if possible. In sum, the execution unit should not sit idle and wait for any reason.

| Architecture | Event | Description |
|---|---|---|
| **Intel® Core™2 processor family (Intel® Core™2 Duo/Quad)** | DIV | Counts the number of divide operations executed. This includes integer divides, floating point divides, and square-root operations executed. |
| | CYCLES_DIV_BUSY | Counts the number of cycles the divider is busy executing divide or square-root operations. The divide can be integer, X87 or Streaming SIMD Extensions (SSE). The square-root operation can be either X87 or SSE. |
| **Intel® Core™ architecture (Intel® Core™ i7, i5, i3; Nehalem)** | ARITH.DIV | Counts the number of divide or square-root operations. The divide can be integer, X87 SSE. The square-root operation can be either X87 or SSE. |
| | ARITH.CYCLES_DIV_BUSY | Counts the number of cycles during which the divider is busy executing divide or square-root operations. The divide can be integer, X87, or SSE. The square-root operation can be either X87 or SSE. |
| **2nd generation Intel® Core™ architecture (a.k.a. SandyBridge)** | ARITH.FP_DIV | Counts the number of the divide operations executed. |
| | ARITH.FPU_DIV_ACTIVE | Counts the number of cycles during which the divider is busy executing divide or square-root operations. |

**Table 1:** PMU events used to count the specific architectural events.

There are many contributing factors to the stall cycles and sub-optimal usage of the execution unit. Examples include memory accesses (e.g., cache misses), branch mis-predictions (pipeline flushes as a result), computational issues (e.g., long latency operations such as division, fp control word change, etc.), and μops not retiring due to the out of order (OOO) engine.

Long latency instructions such as division and square-root can introduce stalls during the execution. Intel VTune Amplifier XE can help pinpoint where such operations are taking place and if these operations are contributing to stall cycles during the execution.

Example: Let's consider the N-Body problem for this exercise. The N-Body problem predicts the motion of a group of celestial objects that interact with each other gravitationally. The sample application proceeds over time steps and in each step computes the net force on every body and updates its position, acceleration, and velocity accordingly. This serial implementation requires O(N2) operations in each iteration as shown in **Figure 2**.

**Figure 3** reveals the analysis of the sample code on an Intel® Core™ i7 (x980)-based system (3.33GHz, 6 core + Hyper-Threading enabled) with Intel VTune Amplifier XE.

One way to optimize the code is to replace the division with reciprocal multiplication as shown in **Figure 4**.

```
...
// Run the simulation over a fixed range of time steps
for (double s = 0.; s < STEPLIMIT; s += TIMESTEP)
{
  // Compute the accelerations of the bodies
  for (i = 0; i < n - 1; ++i)
  {
      for (j = i + 1; j < n; ++j)
      {
          // compute the distance between them
          double dx = body[i].pos[0]-body[j].pos[0];
          double dy = body[i].pos[1]-body[j].pos[1];
          double dz = body[i].pos[2]-body[j].pos[2];

          double distsq = dx*dx + dy*dy + dz*dz;
          if (distsq < MINDIST) distsq = MINDIST;
          double dist = sqrt(distsq);

          // compute the unit vector from j to i
          double ud[3];
          ud[0] = dx / dist;
          ud[1] = dy / dist;
          ud[2] = dz / dist;

          // F = G*mi*mj/distsq, but F = ma, so ai = G*mj/distsq
          double Gdivd = GFORCE/distsq;
          double ai = Gdivd*body[j].mass;
          double aj = Gdivd*body[i].mass;

          // apply acceleration components using unit vectors
          for (int k = 0; k < 3; ++k)
          {
              body[j].acc[k] += aj*ud[k];
              body[i].acc[k] -= ai*ud[k];
          }
      }
  }
  // apply acceleration and advance bodies
  for (i = 0; i < n; ++i)
  {
      for (j = 0; j < 3; ++j)
      {
          body[i].vel[j] += body[i].acc[j] * TIMESTEP;
          body[i].pos[j] += body[i].vel[j] * TIMESTEP;
          body[i].acc[j] = 0.;
      }
  }
}

...
```

**Figure 2**

The optimized code consumes 4,668 million fewer clockticks and reduces the dispatch stalls from 7,448 million cycles to 3,020 million cycles. **Figure 5**

Additionally, the application can be parallelized not only to leverage the available cores, but also to reduce the impact of the DIVs. Parallelization will allow utilization of all the ports on the cores performing the divisions.

## Estimating FLOPS

In this next section, you will find out how hardware-based EBS technology can help developers estimate the FLOPS performed by their applications. FLOPS will refer to 32-bit and 64-bit floating-point operations and the operations will be either addition or multiplication (computational).

As **Figures 6, 7,** and **8** demonstrate, FLOPS can be performed on legacy x87 registers or on SSE registers, depending on how the compiler generates the code. If the floating-point instructions are executed on SSE registers, then they can be either scalar or packed operations.

**Table 2** and **Table 3** give the PMU event names which can be used to statistically estimate the computational floating point operations executed by the hardware. Please keep in mind that not all the executed instructions are retired due to the speculative nature of the architecture. Therefore, it is possible to experience over-counting of these events.

Intel VTune Amplifier XE can use any of the events individually or all of them at the same time to estimate the FLOPS executed by the hardware. In order to measure the elapsed time, the CPU_CLK_UNHALTED (a.k.a. clockticks) event can be used. If the processor frequency is constant during the measuring period, you can use the clockticks event to calculate the elapsed wall clock time.



For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

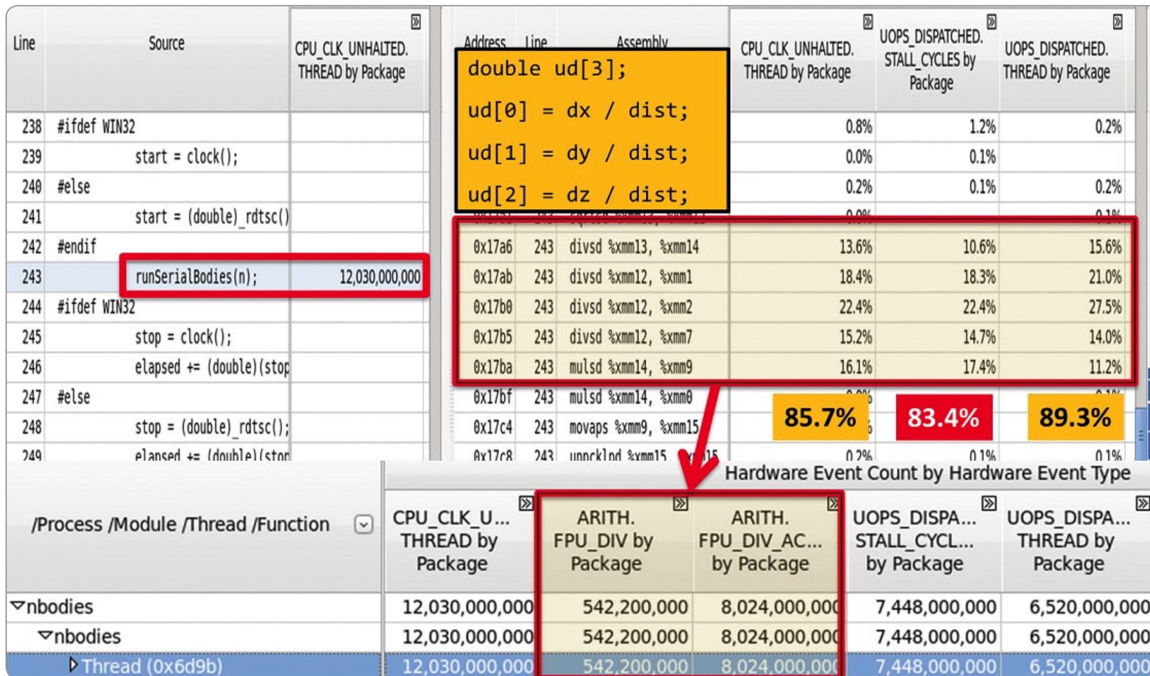
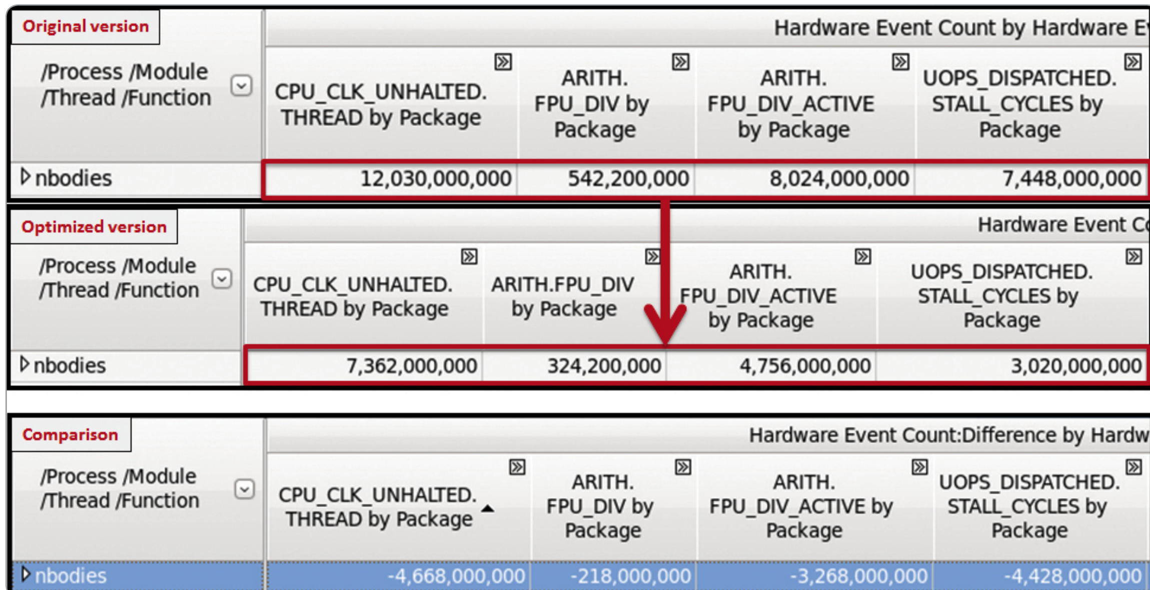



**Figure 3:** Analysis of the sample code — 85% of the clockticks, 89.3% of the total µops dispatch, and 83.4% of the dispatch, stalls are happening in this code segment.

```
// compute the unit vector from j to i
double ud[3];
ud[0] = dx / dist;
ud[1] = dy / dist;
ud[2] = dz / dist;
```

```
// compute the unit vector from j to i
double ud[3];
double dd=1.0 / dist;
ud[0] = dx * dd;
ud[1] = dy * dd;
ud[2] = dz * dd;
```

**Figure 4**



Original version — Hardware Event Count by Hardware E

| /Process /Module /Thread /Function | CPU_CLK_UNHALTED. THREAD by Package | ARITH. FPU_DIV by Package | ARITH. FPU_DIV_ACTIVE by Package | UOPS_DISPATCHED. STALL_CYCLES by Package |
|---|---|---|---|---|
| nbodies | 12,030,000,000 | 542,200,000 | 8,024,000,000 | 7,448,000,000 |

Optimized version — Hardware Event C

| /Process /Module /Thread /Function | CPU_CLK_UNHALTED. THREAD by Package | ARITH.FPU_DIV by Package | ARITH. FPU_DIV_ACTIVE by Package | UOPS_DISPATCHED. STALL_CYCLES by Package |
|---|---|---|---|---|
| nbodies | 7,362,000,000 | 324,200,000 | 4,756,000,000 | 3,020,000,000 |

Comparison — Hardware Event Count:Difference by Hardw

| /Process /Module /Thread /Function | CPU_CLK_UNHALTED. THREAD by Package | ARITH. FPU_DIV by Package | ARITH. FPU_DIV_ACTIVE by Package | UOPS_DISPATCHED. STALL_CYCLES by Package |
|---|---|---|---|---|
| nbodies | -4,668,000,000 | -218,000,000 | -3,268,000,000 | -4,428,000,000 |

**Figure 5:** Comparisons of the original and optimized version

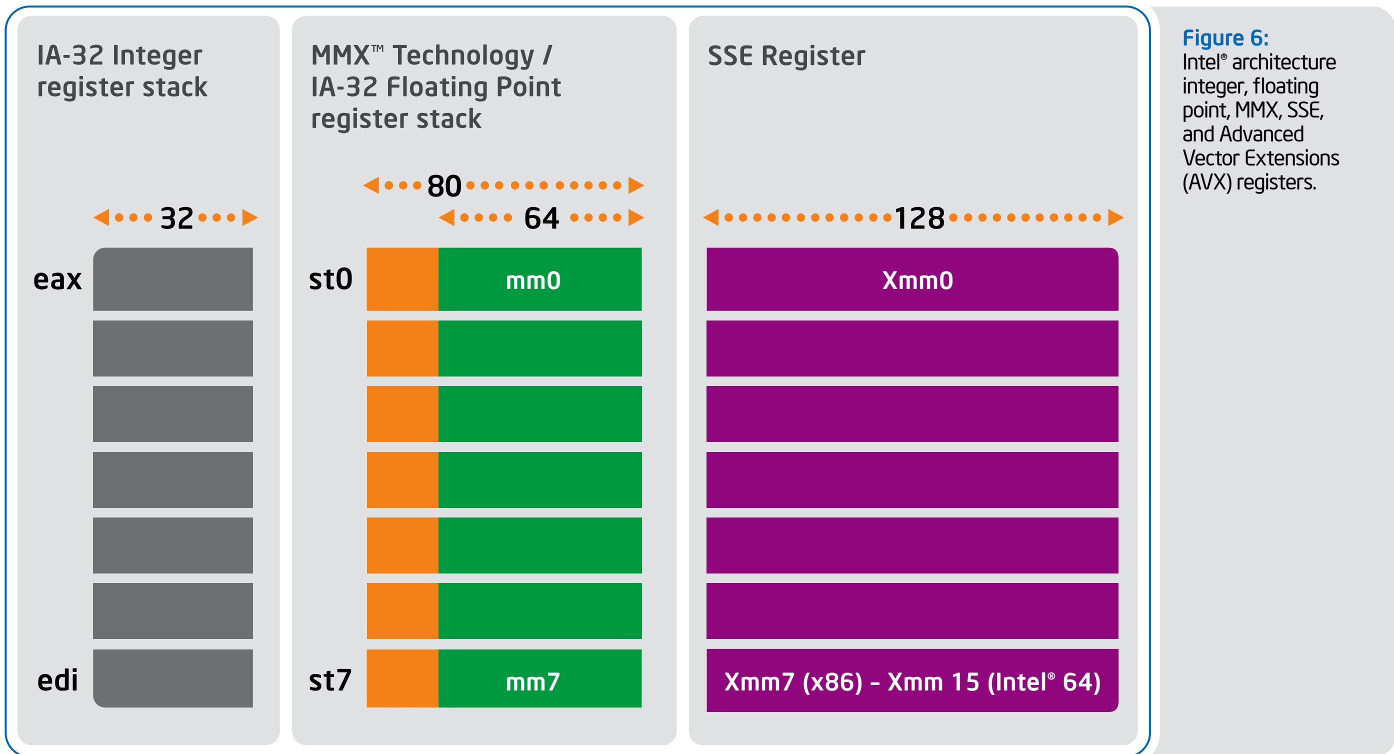



**Figure 6:**
Intel® architecture integer, floating point, MMX, SSE, and Advanced Vector Extensions (AVX) registers.

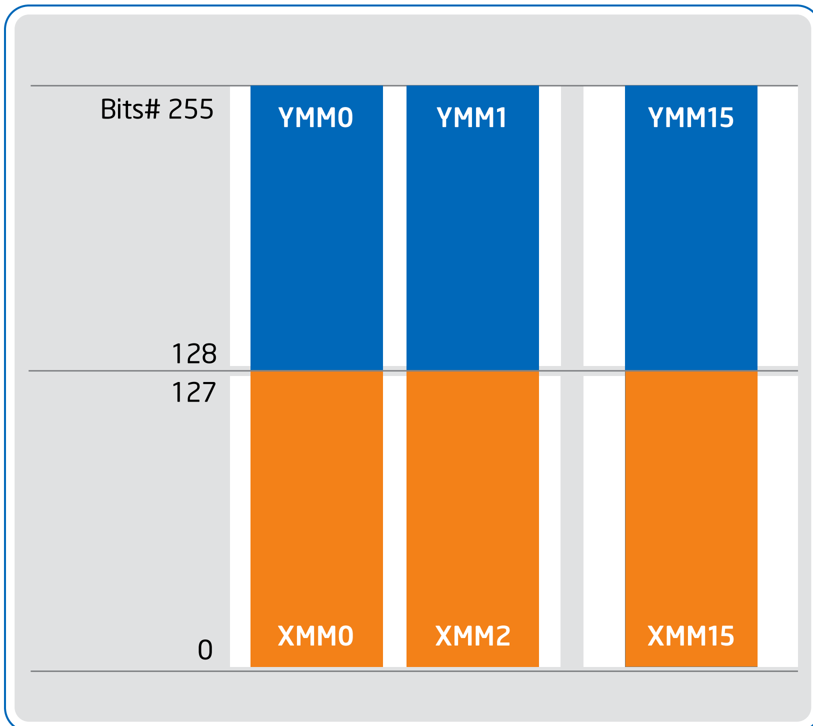



**Figure 7:**
Intel AVX introduced support for 256-bit wide SIMD registers (YMM0-YMM7 in operating modes that are 32-bit or less, YMM0-YMM15 in 64-bit mode). The lower 128 bits of the YMM registers are aliased to the respective 128-bit XMM registers.

"Performance tuning usually focuses on reducing the time it takes to complete a well-defined workload."



For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

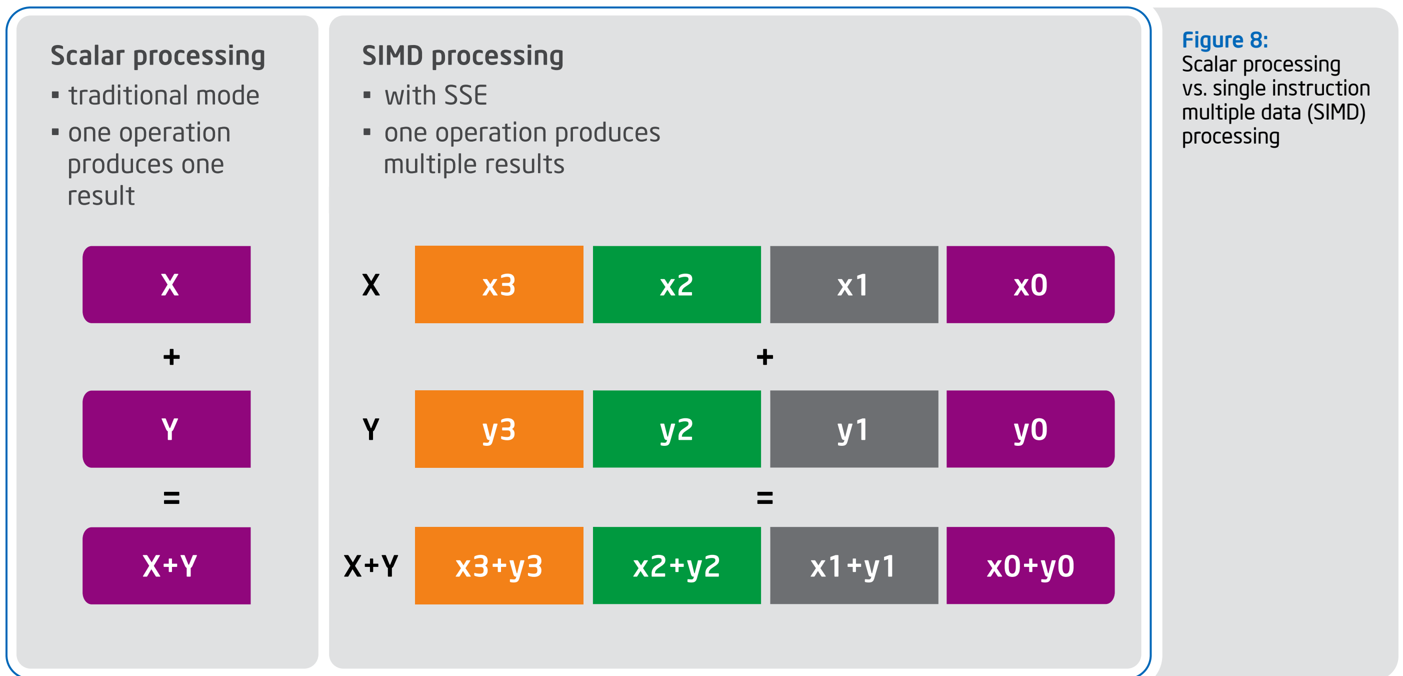



**Figure 8:** Scalar processing vs. single instruction multiple data (SIMD) processing

| Processor Generation | Processor Event Names | | |
|---|---|---|---|
| | Floating-point operations using legacy x87 | Floating-point operations using SIMD | |
| **Intel® Core™2 processor family** (Intel® Core™2 Duo/Quad) | X87_OPS_RETIRED.ANY | Packed 64-bit | SIMD_COMP_INST_RETIRED.PACKED_DOUBLE |
| | | Packed 32-bit | SIMD_COMP_INST_RETIRED.PACKED_SINGLE |
| | | Scalar 64-bit | SIMD_COMP_INST_RETIRED.SCALAR_DOUBLE |
| | | Scalar 32-bit | SIMD_COMP_INST_RETIRED.SCALAR_SINGLE |
| **Intel® Core™ architecture** (Intel® Core™ i7, i5, i3 — a.k.a. Nehalem) | FP_COMP_OPS_EXE.x87 | Packed 64-bit | FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION |
| | | Packed 32-bit | FP_COMP_OPS_EXE.SSE_SINGLE_PRECISION |
| | | Scalar 64-bit | FP_COMP_OPS_EXE.SSE_FP_SCALAR |
| | | Scalar 32-bit | FP_COMP_OPS_EXE.SSE_FP_SCALAR |
| **2nd generation Intel® Core™ architecture** (a.k.a. SandyBridge) | FP_COMP_OPS_EXE.X87 | Packed 64-bit | FP_COMP_OPS_EXE.SSE_PACKED_DOUBLE |
| | | Packed 32-bit | FP_COMP_OPS_EXE.SSE_PACKED_SINGLE |
| | | Scalar 64-bit | FP_COMP_OPS_EXE.SSE_SCALAR_DOUBLE |
| | | Scalar 32-bit | FP_COMP_OPS_EXE.SSE_SCALAR_SINGLE |

**Table 2:** PMU events are used to count the computational floating-point operations.

Alternatively, CPU_CLK_UNHALTED.REF, which counts the number of reference cycles and is not affected by thread frequency changes, can be used. The difference between the reference clocktick event and clocktick event is that even if a thread enters the halt state (by running the HLT instruction), the reference clocktick event continues to count as if the thread is continuously running at the maximum frequency.

The FLOPS formula can be given as follows:

FLOPS = ((number of FP ops / clock) * number of total computational FP ops) / Elapsed Time

Elapsed time = CPU_CLK_UNHALTED / Processor-Frequency / Number-of-Cores. The cores with a non-zero CPU_CLK_UNHALTED event count need to be considered for this formula.

To demonstrate how EBS technology can be used to estimate the FLOPS, a simple multithreaded matrix multiplication will be used. This sample application leverages the thread pool concept and each thread in the thread pool executes the code in **Figure 9**.

The application also reports the FLOPS measured by dividing the total floating-point operations (2 / iteration * NUM * NUM * NUM) with the elapsed time. The elapsed time only includes the matrix multiplication part and does not include the initialization and thread creation overhead.

In order to collect samples for the relevant code section, __itt_pause() (pauses the collection) and __itt_resume() (resumes the collection) APIs are used. Please refer to Intel VTune Amplifier XE documentation for information on how to leverage the user APIs.

Intel VTune Amplifier XE on the Intel® Xeon® processor can be configured as shown in **Figure 10** on an Intel Core i7 (x980)-based system (six core + hyper-threading).

## Using x87 registers

The sample application is compiled in released mode (optimization level set to 0x) on a Windows* system using Microsoft Visual Studio*.

**Figure 11** and **Figure 12** demonstrate what the application reports when analyzed with Intel VTune Amplifier XE. The results offer insight about how the compiler generated the code. In this run, we can clearly see that we only collected samples on FLOPS using x87 register stacks.

Next, plug the numbers into the formula found in **Figure 13**.

| Processor Generation | Processor Event Names | |
|---|---|---|
| | Floating-point operations using AVX | |
| **2nd generation Intel® Core™ architecture** (a.k.a SandyBridge) | Packed 64-bit | SIMD_FP_256. PACKED_DOUBLE |
| | Packed 32-bit | SIMD_FP_256. PACKED_SINGLE |
| | **Note:** Packed AVX-256 can be counted as one, and will count for SIMD FP 128. | |

**Table 3:** PMU events are used to count the computational floating-point operations using AVX.

```
double a[NUM][NUM];
double b[NUM][NUM];
double c[NUM][NUM];
...
slice = (unsigned int) tid;
from  = (slice * NUM) / NUM_THREADS;
to    = ((slice + 1) * NUM) / NUM_THREADS;
for(i = from; i < to; i++)
{
      for(j = 0; j< NUM; j++)
      {
          for(k = 0; k < NUM; k++)
              // 2 fp ops / iteration: 1 add, 1 multiply
              c[i][j] += a[i][k] * b[k][j];
      }
}
...
```

**Figure 9**



For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

**FLOPS** [Edit]

Identify your most time-consuming source code. Unlike Hotspots, Lightweight Hotspots has lower overhead because it does not collect stack information. It can also be used to sample all processes on a system. This analysis type uses hardware event-based sampling collection.

**Hardware Event-based Sampling Settings** [Edit]

Events configured for CPU: Intel(R) Xeon(R) Processor

NOTE: For analysis purposes, Amplifier XE may adjust the Sample After values in the table below by a multiplier. The multiplier depends on the value of the Duration time estimate option specified in the Project Properties dialog.

| Event Name | Sample After | LBR Filter | Event Description |
|---|---|---|---|
| CPU_CLK_UNHALTED.REF_P | 2000000 | None | Reference base clock (133 Mhz) cycles when thread is not halted (programmable counter) |
| CPU_CLK_UNHALTED.THREAD | 2000000 | | Cycles when thread is not halted (fixed counter) |
| FP_COMP_OPS_EXE.SSE_FP_PACKED | 2000000 | None | SSE FP packed Uops |
| FP_COMP_OPS_EXE.SSE_FP_SCALAR | 2000000 | None | SSE FP scalar Uops |
| FP_COMP_OPS_EXE.X87 | 2000000 | None | Computational floating-point operations executed |
| INST_RETIRED.ANY | 2000000 | | Instructions retired (fixed counter) |

**Figure 10:** Shows a custom analysis type, which is created to measure relevant PMU events.

```
D:\Examples\flops\msvc\Release>flops.exe
MFLOPS: 1130.180 mflops
Thread #:12 Elapsed time = 15.201000 seconds

D:\Examples\flops\msvc\Release>_
```

**Figure 11**

| /Function /Thread /H/W Context | Hardware Event Count by Hardware Event Type: CPU_CLK_UNHALT... THREAD by Package | INST_RETIRED. ANY by Package | FP_COMP_OPS_EXE. X87 by Package | FP_COMP_OPS_EXE. SSE_FP_PACKED by Package | FP_COMP_OPS_EXE. SSE_FP_SCALAR by Package |
|---|---|---|---|---|---|
| matrixMultiply | 607,652,000,000 | 31,220,000,000 | 18,470,000,000 | 0 | 0 |
| Thread (0x1b64) | 51,438,000,000 | 2,594,000,000 | 1,544,000,000 | 0 | 0 |
| Thread (0x2318) | 51,346,000,000 | 2,620,000,000 | 1,548,000,000 | 0 | 0 |
| Thread (0x2030) | 51,102,000,000 | 2,550,000,000 | 1,536,000,000 | 0 | 0 |
| Thread (0x12e0) | 50,944,000,000 | 2,612,000,000 | 1,548,000,000 | 0 | 0 |
| Thread (0x221c) | 50,930,000,000 | 2,588,000,000 | 1,528,000,000 | 0 | 0 |
| Thread (0x22e8) | 50,724,000,000 | 2,596,000,000 | 1,534,000,000 | 0 | 0 |
| Thread (0x1604) | 50,600,000,000 | 2,614,000,000 | 1,544,000,000 | 0 | 0 |
| Thread (0x1258) | 50,594,000,000 | 2,642,000,000 | 1,542,000,000 | 0 | 0 |
| Thread (0x1920) | 50,556,000,000 | 2,588,000,000 | 1,522,000,000 | 0 | 0 |
| Thread (0x102c) | 50,392,000,000 | 2,606,000,000 | 1,542,000,000 | 0 | 0 |
| Thread (0x1774) | 49,880,000,000 | 2,606,000,000 | 1,544,000,000 | 0 | 0 |
| Thread (0x1d00) | 49,146,000,000 | 2,604,000,000 | 1,538,000,000 | 0 | 0 |
| Selected 1 row(s): | 607,652,000,000 | 31,220,000,000 | 18,470,000,000 | 0 | 0 |

**Figure 12:** Shows the results of Intel® VTune™ Amplifier XE analysis of the application leveraging x87 register stack. The matrixMultiply() function is executed almost equally by all of the threads.

**MFLOPS formula** = FP_COMP_OPS_EXE.FP / $1\times10^{6}$ / Elapsed Time
**Elapsed time** = CPU_CLK_UNHALTED.THREAD / Processor-Frequency / Number-of-Cores

Elapsed Time = 607,652,000,000.00 / $3.33 \times 10^{9}$ / 12 = 15.206 secs
MFLOP = 18,470,000,000.00 / $1\times10^{6}$/ 15.206 secs = 1214.652 Mflops

**Figure 13**

Sign up for future issues | Share with a friend

```
D:\Examples\flops\msvc\Release>flops.exe
MFLOPS: 10437.345 mflops
Thread #:12 Elapsed time = 1.646000 seconds

D:\Examples\flops\msvc\Release>_
```

Figure 14

| /Function /Thread /H/W Context | Hardware Event Count by Hardware Event Type: CPU_CLK_UNHALTED. THREAD by Package | INST_RETIRED. ANY by Package | FP_COMP_OPS_EXE. X87 by Package | FP_COMP_OPS_EXE. SSE_FP_PACKED by Package | FP_COMP_OPS_EXE. SSE_FP_SCALAR by Package |
|---|---|---|---|---|---|
| ⊟ threadPool | 66,178,000,000 | 20,460,000,000 | 0 | 9,152,000,000 | 0 |
| ⊞ Thread (0x1f90) | 5,810,000,000 | 1,714,000,000 | 0 | 774,000,000 | 0 |
| ⊞ Thread (0x1b0c) | 5,774,000,000 | 1,702,000,000 | 0 | 768,000,000 | 0 |
| ⊞ Thread (0x14f8) | 5,724,000,000 | 1,698,000,000 | 0 | 762,000,000 | 0 |
| ⊞ Thread (0x1e64) | 5,708,000,000 | 1,706,000,000 | 0 | 774,000,000 | 0 |
| ⊞ Thread (0x1a3c) | 5,652,000,000 | 1,704,000,000 | 0 | 768,000,000 | 0 |
| ⊞ Thread (0x2348) | 5,600,000,000 | 1,720,000,000 | 0 | 760,000,000 | 0 |
| ⊞ Thread (0x1048) | 5,516,000,000 | 1,700,000,000 | 0 | 756,000,000 | 0 |
| ⊞ Thread (0x1430) | 5,502,000,000 | 1,702,000,000 | 0 | 762,000,000 | 0 |
| ⊞ Thread (0x188c) | 5,482,000,000 | 1,714,000,000 | 0 | 776,000,000 | 0 |
| ⊞ Thread (0x8f0) | 5,442,000,000 | 1,710,000,000 | 0 | 746,000,000 | 0 |
| ⊞ Thread (0xc54) | 5,000,000,000 | 1,700,000,000 | 0 | 752,000,000 | 0 |
| ⊞ Thread (0x21b0) | 4,968,000,000 | 1,690,000,000 | 0 | 754,000,000 | 0 |

**Figure 15:** Shows the results of Intel® VTune™ Amplifier XE analysis of the application leveraging SSE registers.

| Line | Source | Hardware Event Count by Hardware Event Type: CPU_CLK_UNHALTED. THREAD by Package | INST_RETIRED. ANY by Package | FP_COMP_OPS_EXE. X87 by Package | FP_COMP_OPS_EXE. SSE_FP_PACKED by Package | FP_COMP_OPS_EXE. SSE_FP_SCALAR by Package |
|---|---|---|---|---|---|---|
| 91 | DWORD WINAPI threadPool(LPVOID pArg) { | | | | | |
| 92 | | | | | | |
| 93 | int tid = *(int *)pArg; | | | | | |
| 94 | | | | | | |
| 95 | while (!done) { | | | | | |
| 96 | WaitForSingleObject(bSignal[tid], INFINITE); | | | | | |
| 97 | matrixMultiply((void *)tid); | 66,178,000,000 | 20,460,000,000 | | 9,152,000,000 | |
| 98 | SetEvent(eSignal[tid]); | | | | | |
| 99 | } | | | | | |
| 100 | return 0; | | | | | |
| 101 | } | | | | | |

**Figure 16:** Shows the events that are associated with the function call.

```
MFLOPS formula = 2 * FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION / Elapsed Time
Elapsed time = CPU_CLK_UNHALTED.THREAD / Processor-Frequency / Number-of-Cores
Elapsed time = (66,178,000,000 / 3.33 x10⁹ / 12 ) = 1.656 secs
MFLOPS = 2 * FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION / 1 x 10⁶ / 1.656 secs = 11053.140 mflops
```

Figure 17



For more information regarding performance and optimization choices in Intel® software products, visit http://software.intel.com/en-us/articles/optimization-notice.

## Using SSE registers

Now, let us look at the same application when SSE registers are used. If we compile the application using Intel® Compiler version 12.0, we see the results with Intel VTune Amplifier XE in **Figure 14.**

One thing you will notice right away in the new result displayed in **Figure 15** is the difference in the function names where the samples are happening. In the earlier example (see **Figure 12**), we were getting the samples in matrixMultiply function, but now we see the samples in threadPool function. This is due to the inlining, and drilling down into the threadPool function call makes this clear.

It is also easy to see how clockticks consumed by each thread are reduced from ~50 billion cycles (in the version using x87 register stacks) down to 5 billion cycles in this current version (see **Figure 15**). Please note that this improvement is not only due to the usage of vectorization or SSE registers. Other optimizations performed by the Intel Compiler are also contributing to the improvement.

In **Figure 17**, you see the FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION event is multiplied by two because two packed, double precision floating operations can be performed on 128-bit XMM registers. For single precision floating-point operations, the total count for packed single precision floating operations needs to be multiplied by four.

In this article, I explained, with examples, how the powerful EBS technology in Intel VTune Amplifier XE can be used to identify computational issues introduced by long latency instructions, and how it can help estimate FLOPS. Even though this article leveraged some of the advanced features of Intel VTune Amplifier XE, please keep in mind that Intel VTune Amplifier XE introduced intuitive and easy-to-use, pre-defined analysis types, which simplify the performance analysis in both serial and parallel applications. Parallel applications tend to have their own unique sets of problems due the complexities introduced by parallelism. Intel VTune Amplifier XE is designed with these challenges in mind so that it assists developers in identifying bottlenecks in serial applications and in parallel applications by providing essential profiling tools and techniques.

For more information, please check out the product site. □

1. Instructions retired: Recent generations of Intel® 64 and IA-32 processors feature microarchitectures using an out-of-order execution engine. They are also accompanied by an in-order front end and retirement logic that enforces program order. Instructions executed to completion are referred to as instructions retired.

"Parallelization will allow utilization of all the ports on the cores performing the divisions."

# BLOG highlights

## Blogs and Videos About PBB

**NOAH CLEMONS,** Technical Consulting Engineer, Parallel Programming Products

There are many articles and trainings on the Intel® Software Network (ISN) about each one of the Parallel Building Blocks, with each of them varying from beginner to expert levels. But what if you want to learn something new in 15 minutes or less a day about them?

I would like to relate what I have learned from customers and my own studies of each of the Intel® Parallel Building Blocks (Intel® PBB) in a daily article and complementing video. My strategy for the ISN blogs/videos is to highlight a particular aspect of Intel PBB, starting with Intel® Cilk Plus, that can be read and watched in 15 minutes or less. Each article will have some commentary as well about how the model in question compares with the other two models.

These articles will start from the very beginners' level and ramp up, starting with the evolution of the 'for' loop—something we learned about in the first week of CS101. By the end of these blogs and videos, you should have excellent knowledge of each of the PBB models and how they can enhance your workload for the latest multi- (and many-) core processors.

SEE NOAH'S 15-MINUTE ENTRIES:

Visit **Go-Parallel.com**

Browse other blogs exploring a range of related subjects at Go Parallel: Translating Multicore Power into Application Performance.

## Case Study:

# Massachusetts General Hospital*

by Bevin Brett

**Follow the experiences of and lessons learned by developers at Massachusetts General Hospital and Intel as they seek to improve virtual colonoscopies.**

**A team of researchers** (Dr. Hiro Yoshida PhD., Dr. Yin Wu PhD., and others) at Massachusetts General Hospital (MGH) 3D Imaging Research had hardware and prototype software for doing a virtual colonoscopy. However, the execution and display times were much too slow for production use, taking more than 45 minutes combined.

Steve Aylward (Microsoft* general manager for commercial health and life sciences) described it this way in an article for eWeek.com:

> *[They] sought the advice of Microsoft and Intel to see how the virtual colonoscopies could be speeded up while avoiding the invasive procedures, chalky laxatives, sedation and higher costs of traditional colonoscopies.*

MGH and Intel developers used the latest Intel® Parallel Studio products to identify, prioritize, and make changes to the C++ code to improve the serial algorithms and introduce parallelism. The team started with the goal of reducing the execution and display times to under five minutes to demonstrate the application at industry conferences. The changes were made over a period of a few months, meeting the tight deadline, and achieving the initial performance goals. The revised code was used in demonstrations at SC 2010* and RSNA 2010*.

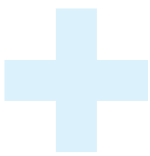
This article describes the experiences of and the lessons learned by these developers as they met the challenge of rapidly improving the performance of a C++ program.

## Caveat

Real programming, especially by multicompany teams, is complex. Time constraints, learning curves, and other work intervene to prevent activities from happening in the ideal order. Note that the following discussion does not attempt to capture all the activities or the precise order in which the activities occurred.



For more information regarding performance and optimization choices in Intel® software products, visit http://software.intel.com/en-us/articles/optimization-notice.





## The application

While virtual colonoscopies are already being administered to people, the process requires that the patient avoid food and receive an enema—both unpleasant experiences. The improved prototype virtual colonoscopy process will have the patient eating low-texture food and taking a pill that contains an X-ray opaque dye. A computed tomography (CT) scan takes cross-sectional X-ray pictures of the patient's lower abdomen, and then a computer program processes the pictures to erase the dyed food. A doctor can interactively view the processed data, seeing the virtual wall of the colon.

A virtual colonoscopy has three major steps:

1. Gather the initial data from the CT scan into a cube data structure representing the density at locations within a cubic volume containing the patient's abdomen.
2. Process the cube with the "electronic cleansing" code to produce a similar, enhanced cube. This step erases the dyed food in the colon and does some image enhancement.
3. Use a visualizer to let the medical expert do a "fly through" of the colon in the final cube to search for interesting features, such as polyps.

The second step is the main computational step. It is the focus of this article, although work was also done on the other steps. In the second step, data flows through a series of stages. Each stage receives one or more cubes produced by earlier stages, and produces new cubes for consumption by later stages.

The input to the first stage is density data giving the density of evenly spread samples within the patient's abdomen encoded as DICOM data structures. It is about 500 cross-sectional, 512 x 512, gray-scale X-ray images of the patient's abdomen, with each voxel (a pixel in a 3-D volume) having values between -2000 (air density) and 2000 (something that almost completely blocks X-rays). Bone and X-ray contrast agents are typically somewhere in the middle. Muscle, water, and fat are less than bone but more than air. The density recorded in each voxel depends on the relative amounts of the various substances contributing to it. The raw data is about 0.5 K x 0.5 K x 0.5 K x 2 bytes = 0.25 Gb in this form.

The large amounts of data have a significant impact on the CPU time used in the stages, and the data structures and algorithms must take into consideration the amount of RAM on the system and the time it takes to transfer such large amounts of data to and from the disk.

Many of the intermediate cubes of data require 4-byte floats for their voxels. Thus, they would take closer to 1 Gb, except that often only about 1/8th of the space requires a non-zero value. The non-zeros are clumped together, so they can also be compressed to about 0.1 .. 0.25 Gb just by omitting areas with only zeros.

Passing through the stages there may be as many as 20 or more of these cubes, and each stage may require some intermediate cubes as well. As a result, data quantity gets into the 2 Gb to 5 Gb range, and any parallelism that causes the intermediate cubes to co-exist can raise this into the 10 Gb to 20 Gb range.

The data flowing through the pipe can be written to, and read from, the disk so that stages in the pipe can be executed standalone. The final data is written to the disk.

One of the first decisions was to use a 64-bit address space and keep the cubes in memory as much as possible. Because there were relatively few pointers in the data structures, the extra length of the pointers did not significantly increase the data size.

## The data structure for the cube

The cube is implemented as a C++ template class, where the template argument provides the type of the voxel. Various member functions provided operations to read and write at any (X, Y, Z) index within the cube, and to iterate over the cube.

The cube is implemented as an array of pointers to smaller cubes, and the smaller cubes themselves have pointers to arrays of the voxels. These smaller cubes are also implemented as a C++ template class. To save space, sub-cubes can be omitted and share their arrays of voxels.

## Using ZLIB

The decision to make a 64-bit application introduced the need for a 64-bit ZLIB library—and preferably one that was already optimized for the hardware. This was found in the Intel® Integrated Performance Primitives (Intel® IPP) samples, where there is an optimized version of exactly the support that was needed.

## Coding change methodology

The change to 64 bits, the introduction of ZLIB, and various small improvements to the cube template classes introduced enough problems that it took a few days to get the new code to compile and start to run again.

It took several weeks to get data all the way through the pipeline and debug all the stages, making it difficult to determine if an impact had been made. The preliminary feeling was that the earlier stages were running faster.

Despite being faster, each stage was taking more than five minutes when running a release build on production data sets. When a debug build was used, times were much worse—for some parts of the code, the times were 100x the release build times. For this reason, many of the files were compiled optimized, and only files that needed to be debugged were built that way.

The work of debugging the later stages continued in parallel as the performance of the earlier stages was improved.

The team adopted the following methodology for the performance improvement work:

1. Profile the application, and look at the top-down call tree and loop view shown by Intel® Parallel Advisor, a tool that helps programmers introduce parallelism.
2. Find the next hot outermost loop that had not already been improved but that offered the promise of either improved serial time or the possibility to introduce parallelism. Do the serial improvement first, choosing approaches that can be parallelized. If adding parallelism, add the site/task annotations and verify that the site is suitable.
3. Create a unit test for that loop or its immediate surroundings.
4. Measure the unit test.
5. Modify the loop.
   a. If introducing parallelism, use Intel Parallel Advisor to look for possible problems.
   b. Make the serial changes or introduce parallelism.
6. If parallelism has been introduced, use Intel® Parallel Inspector to look for problems. Intel Parallel Inspector includes support for finding data races.
7. Verify that the unit test still passes.
8. Measure the unit test, and decide whether the desired speedup has been achieved. If it has not, repeat the process.

Sign up for future issues | Share with a friend

## The measurements

The goal for the electronic cleansing (second) step was to get the time below five minutes.

At first sight, the measurements were disappointing. Time was fairly evenly spread over the five stages of the pipeline, and within the stages there were a total of about 20 loops that consumed the time, most of them taking a lot longer than their share of the five-minute goal. Most would have to be improved to significantly reduce the elapsed time.

## The unit tests

Each loop was applying a complex algorithm to a large amount of input data. The input data was coming from earlier in the pipeline and could not easily be faked; it was not at all obvious how to create a suitable small input set that would have the same characteristics. This meant that the unit tests would have to be conducted on real data.

For each algorithm to be replaced, a unit test was created. The code was modified to write the algorithm's inputs to a file. The unit test was created to:

1. Read the inputs from the disk.
2. Apply the existing algorithm to the inputs to get the current "new" output.
3. Apply a copy of the existing algorithm to the inputs to get the old output.
4. Compare the two outputs, and verify that they are the same.

Of course, they should be the same. This comparison was really testing the unit test, and could identify cases where all the inputs were not captured.

The existing algorithm was then changed to the new algorithm, and the unit test rerun to verify the same output was produced.

## Using Intel Integrated Performance Primitives

The Partners HealthCare researchers had already identified the Intel Integrated Performance Primitives (Intel IPP) convolution support as being a good match for several of the algorithms used in the steps.

A convolution is a mathematical operation between two arrays. Basically, one of the arrays is zero-extended at each end by as many zeros as there are elements in the second array. The second array is then slid along this extended array. At each position in the slide, the dot-product is computed (corresponding elements are multiplied, and the products summed) and the sum is placed at that position in the output.

However, the algorithms were not doing one-dimensional convolutions. Instead, they were doing 3-D convolutions by first going along the Y*Z rows parallel to the X axis to produce an intermediate result. They then went along the X*Z rows parallel to the Y axis of this intermediate cube to produce another intermediate result. Finally, they went along the X*Y rows parallel to the Z axis to produce the final result.

The result was that each of the three loops for the three dimensions spent most of its time indexing into the rows to produce short vectors of Float32s for the Intel IPP convolution code to work on. It then indexed into the output cube to write the values back to.

Four significant changes were made to this algorithm.

1. Keep the intermediate results as Float32s.
2. Reorder the indices, so that in terms of the original cube's indices the cell that started as (X, Y, Z) in the original was written into (Y, Z, X) in the first intermediate result, then into (Z, X, Y) in the second intermediate, and finally back into (X, Y, Z) in the final result.
3. Assemble longer vectors of data to feed into the convolution function.

   Consider two original rows that consisted of the following:

   (0,0,0,0, 0,**1**,0,0, 0,0,**2**,0, 0,0,0,0)

   (0,0,0,0, **1**,0,**1**,0, 0,**2**,**2**,0, 0,0,0,0)

   Both are to be convolved with the length=3 vector (0.25, 0.5, 0.25) to do some blurring.

   One can concatenate the two rows, with enough 0s padding in between, to form a long vector. The resulting vector can be fed into the convolution code, yielding high-performance execution. The portion of the result corresponding to the padding is then discarded.

   Furthermore, one can shrink long bursts of zeros, and use zeros for these eliminated portions in the result.

   The final version turns the above two vectors into input for the convolution:

   (0,0,**1**,0, 0,**2**,0, 0,**1**,0,**1**,0, 0,**2**,**2**,0, 0)

   The convolution results are applied in the following:

   (0,0.25,**0.5**,0.25,0.5,**1.0**,0.5,0.25,**0.5**,0.5,**0.5**,0.25,0.5,**1.5**,**1.5**, 0.5,0.0)

   The zeros and end-of-rows are reinserted to create the final answer:

   (0,0,0,0, 0.25,**0.5**,0.25,0.0, ...) and

   (...)
4. If the entire cube was done at once, the intermediate results would not fit in the primary or secondary caches, and main memory would be accessed for most of the traffic. Instead, the large cubes are decomposed into smaller cubes, and they are treated as separate problems. Experimentation was used to size the smaller cubes. When they are too small, the need to go beyond the ends of the smaller cubes results in duplicated effort, so there is a hardware-specific optimal size.

## Benefiting from multicore parallelism

The change to using Intel IPP and other serial enhancements significantly improved the time spent in the electronic cleansing step, but a lot more speedup was needed to get under the five-minute target.

The Intel Parallel Advisor survey feature pointed to many loops that could be parallelized, and those were attacked one at a time. Two of the stages performed multiple independent convolutions, and other stages had algorithms that iterated over large subsets of the voxels within the cubes.

## Parallelizing the convolutions

One stage did three convolutions and another did nine. In both cases, all the convolutions done by the stage were on the same data.

There were three obvious choices for introducing parallelism:

1. Do the individual convolutions in parallel.
2. Do the 3-D convolution of the parts of a cube in parallel.
3. Do the 1-D convolution of the vectors in parallel.

The 1-D convolutions were already being done by Intel IPP, and so were assumed to be efficiently exploiting the hardware.

Clearly, with nine independent convolutions, doing each on a separate core involved the least amount of work and had the potential for a nine-fold speedup. The team used the Intel Parallel Advisor correctness capability to verify that they were independent, tried running them in parallel, and measured the achieved speedup.

The achieved speedup from doing all nine convolutions in parallel on a 16-core system was approximately 3x, even when the cube was


For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

decomposed into the optimal-size sub-cubes. There was so much data to read and write, and so little reuse of the data, the main memory bandwidth was the gating factor and the caches did not hide this.

The team did more experiments correlating the number of convolutions being run in parallel with the achieved speedup, and discovered that, for this algorithm and hardware, three-way parallelism, with carefully chosen decomposition cube size, was about the best that could be done. The nine convolutions were load balanced into three tasks doing three convolutions each. Similarly, the three convolutions of the other stage were balanced into three tasks doing one convolution each.

## For-all-points and wave-front algorithms

There were several algorithms that processed many voxels within the cube, either based on their location or on their presence in a set. All of these could be done by processing various portions of the cube in parallel.

In cases where it first looked like locks were going to be needed, it was often possible to break the iteration space into contiguous slices, and then process these slices in a manner that avoided the need to lock.

This was done by processing these slices within an outer three-iteration loop, as shown in **Figure 1**.

The middle loop could then be parallelized, knowing that the x-1, x, and x+1 elements of the inp and out were not shared across iterations of this loop and hence did not need to be locked.

## Results

The combined changes reduced the program execution times below the five-minute target, even before all the possible candidates had been parallelized. As a result, the team was able to demo the application as planned.

```
for (int interleave = 0; interleave <
3; interleave++)
      for ( int slice = interleave;
slice < xMax/sliceSize;
slice += 3)

for ( int x = (slice+0)*sliceSize;
x < min((slice+1)*sliceSize, xMax);
x++)

operateOn(
inp[x-1],inp[x],inp[x+1],
out[x-1],out[x],out[x+1]);
```

Figure 1

## Summary

Intel Parallel Advisor made it possible to identify the places to focus efforts, to avoid wasted effort on sites that would not pay off, and to identify correctness issues before they caused non-deterministic crashes and wrong answers.

In conjunction with Intel Parallel Inspector and Intel Parallel Amplifier, Intel Parallel Advisor enabled major improvements to be made in complex code on a tight schedule.

Intel IPP provided two optimized sets of functions (convolutions and ZLIB) that met the needs of the application for reliable optimized code that the team was not capable of producing itself, saving a lot of time. □

# BLOG highlights

## Performance Analysis of Intel® Threading Building Blocks

**DAVID MACKAY,** Lead Technical Consultant

Intel® Threading Building Blocks (Intel® TBB) is a popular abstraction for expressing parallelism in C++ software. Intel TBB leads to good decomposition for threading. But do you know how to check how well it is tuned, so you use Intel Threading Building Blocks most effectively?

Douglas Armstrong, Intel® VTune™ Amplifier XE architect, joins me to share tips on using VTune Amplifier XE for tuning TBB software. Intel VTune Amplifier XE has built-in support for helping find and tune the granularity of domain decomposition in Intel TBB. Douglas feels this is an under-appreciated feature and has captured some screen shots to share with us. Douglas created a sample Intel TBB application and analyzed it with the concurrency option of Intel VTune Amplifier XE.

SEE THE REST OF DAVID'S BLOG:

Visit **Go-Parallel.com**

Browse other blogs exploring a range of related subjects at Go Parallel: Translating Multicore Power into Application Performance.

## 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.

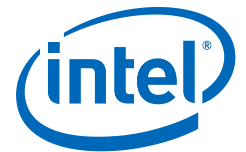
Notice revision #20110307



**Sign up for future issues | Share with a friend**

***The Parallel Universe*** is a free quarterly magazine. **Click here** to sign up for future issue alerts and to share the magazine with friends.

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.



# A quick and easy code boost

## Try the free Intel® evaluation guides

Software development is a battle. Every day C/C++ and Fortran developers square off against stubborn hotspots, nettle-some memory errors, elusive resources leaks, and a host of other obstacles.

**Now, you have another weapon in the fight for stellar application performance and reliability: the Intel® quick evaluation guides.**

Download the step-by-step guide that addresses your development challenge, and see how quickly you can improve your code.

CHECK OUT THE FREE GUIDES TODAY

**Ron Green**
*Manager, HPC and Fortran Compiler Support*

## Find the guide that fits your need

Intel evaluation guides offer free, hands-on tips and techniques for resolving a range of development issues. Here are just a few examples:

- **BOOST PERFORMANCE:** Learn how in many cases recompiling a single file can give you a major performance boost.
- **ADD PARALLELISM:** Easily apply a parallel_for to a conforming for loop for a significant performance increase—even without further tuning.
- **ELIMINATE MEMORY ERRORS:** Find memory and threading errors before they happen at any point in your development cycle.

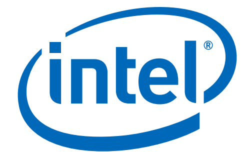
**Intel hopes you find the guides useful in your pursuit of faster, more reliable code.**

Rock your code.
**ROCK YOUR WORLD.**

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

© 2011, Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries. *Other names and brands may be claimed as the property of others.

**INTEL® PARALLEL STUDIO XE**



# Kick your code into high gear

The performance solution for Windows* and Linux* C/C++ and Fortran developers

## Intel® Parallel Studio XE

**The ultimate all-in-one performance suite includes the following industry-leading tools:**

**Performance compiler** Immediately increase performance after recompiling—without changing code.

**Memory checker** Locate memory leaks, buffer overflow, and memory allocation mismatches to boost quality and reliability.

**Static security analyzer** Find 250+ security vulnerabilities and defects to improve source-code quality and security.

**Concurrency analysis** Uncover race conditions and deadlocks to enhance quality and reliability.

## What you can do to produce immediate results

**Go parallel!:** A feature of the Intel® C++ Compiler, guided auto-parallelization (GAP) offers selective advice and, when correctly applied, results in auto-vectorization or auto-parallelization for serially coded applications.

**Stop guessing:** Hotspot analysis finds the functions using the most time. Click [+] for the call stacks. Double-click to see the source.

**Go multimedia:** Intel® Integrated Performance Primitives (Intel® IPP) functions are thread safe. That means you can thread your application and call Intel IPP functions without synchronization problems if the same functions are called by different threads.

| /Function /Call Stack | CPU Time |
|---|---|
| ⊞ initialize_2D_buffer | 11.768s |
| ⊟ grid_intersect | 5.916s |
| ⊞ ↖ intersect_objects | 5.431s |
| ⊞ ↖ grid_intersect ← intersect_objects | 0.485s |
| ⊞ sphere_intersect | 5.044s |

## Boost Application Performance with Recompile on Intel® Xeon® Processor

**Estimated SPECfp*_base2006 (C/C++) floating point benchmark**

| Compiler | Result |
|---|---|
| Intel® C++ Compiler 12.0 for Windows | 1.69X |
| Intel® C++ Compiler 11.1 for Windows | 1.32X |
| Next Best Compilers (Best of Microsoft Visual Studio* 2010 and PGI* C++ Compiler 10.6) | Baseline |

**Estimated SPECint*_base2006 integer benchmark**

| Compiler | Result |
|---|---|
| Intel® C++ Compiler 12.0 for Windows | 1.19X |
| Intel® C++ Compiler 11.1 for Windows | 1.1X |
| Next Best Compilers (Best of Microsoft Visual Studio* 2010 and PGI* C++ Compiler 10.6) | Baseline |

## Drive performance and scalability

Locks and waits analysis identifies a common cause of a slow parallel program: waiting too long on a synchronization object (lock).

Rock your code. **Rock your world.**

Download free 30-day trials of Intel® Software Development Products at **www.intel.com/software/products/eval**.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel® products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, refer to **www.intel.com/performance/resources/benchmark_limitations.htm**. Intel® 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. Configuration info at: **http://software.intel.com/en-us/articles/benchmark-info/#ICC**.

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

© 2011, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Xeon, and VTune are trademarks of Intel Corporation in the U.S. and other countries. *Other names and brands may be claimed as the property of others.