THE PARALLEL UNIVERSE

Issue 10
June 2012

DEVELOPER ROCK STARS:
Shannon Cepeda | **Wendy Doerner**

Help Future-Proof Performance of Your Application with VECTORIZATION in Six Steps

by Shannon Cepeda and Wendy Doerner

Intel Software

Intel® Cilk™ Plus

Your performance booster.

Intel® Cilk™ Plus is now available in open source and for GCC 4.7.

Three Intel Cilk Plus keywords provide a simple, yet surprisingly powerful model for parallel programming.

› Runtime and template libraries offer a well-tuned environment for building parallel applications

› Ideal for C++ and Fortran developers in Linux*, Windows*, open source, or GCC 4.7

Look for Intel® Cilk™ Plus in Intel® Parallel Studio XE

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

CONTENTS

THE PERFORMANCE OPPORTUNITY:

How to Achieve it—from Clusters to Devices

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 is truly a developer-to-developer issue as our software engineers take a close look at the techniques and tips that can increase application performance. These do not have to be complex to achieve good outcomes—many of the basic parallelism processes can be performed efficiently and successfully with the help of Intel® tools.

Help Future-Proof Performance of Your Application with Vectorization in Six Steps looks at exploiting a form of parallelism known as vector parallelism, by performing certain operations on multiple pieces of data at once. Scale-forward methods allow you to vectorize your code without changes for future architectures and CPUs, as well.

Pockets of Parallel Computation, Fitting in Your Pocket: The Intel® Integrated Performance Primitives Landscape for the Intel® Atom™ Processor applies parallelism to small form-factor devices with some very interesting ramifications for mobile application developers.

And, *Tools that Boost .NET Apps Reliability and Performance* assesses the value of two tools in Intel Parallel Studio XE, the Intel® Inspector XE correctness tool, and the Intel® VTune™ Amplifier XE performance analysis tool, for developers of .NET code, native code, and "mixed" (.NET and native) applications during the development cycle.

I hope you will enjoy this issue, apply some of the code, assess the impact on your own development process and software, broaden your personal performance toolkit—and, most of all, achieve a performance boost.

James Reinders

Sign up for future issues | Share with a friend

Help Future-Proof Performance of Your Application with VECTORI in Six Steps

by Wendy Doerner and Shannon Cepeda,
Software Technical Consulting Engineers, Intel

DEVELOPER ROCK STARS:

Shannon Cepeda | **Wendy Doerner**

ZATION

Learn how to get a significant performance boost from processing data in parallel inside a single CPU core. Then, combine with threading and/or cluster parallelism for even more performance.

Sign up for future issues | Share with a friend

The Parallel Universe has featured many articles about parallelism aimed at exploiting multiple cores through threading and cluster programming. This article takes a look at exploiting an equally important form of parallelism known as vector parallelism. Vectorization is parallelism *within a single CPU core* and is a key form of hardware support for data parallelism.

With vectorization, certain operations can be performed on multiple pieces of data at once. It is accomplished by using special instructions called (SIMD) Single Instruction, Multiple Data operations. SIMD instructions, and the hardware that goes along with them, has been present in Intel® processors for over a decade. Some examples of our SIMD instructions sets are Intel® Streaming SIMD Extensions (Intel® SSE), first introduced in 1999 and expanded several times, and Intel® Advanced Vector Extensions (Intel® AVX), introduced last year.

How Does Vectorization Work?

In the typical scalar (non-vectorized) case, each variable you use will each be stored in its own CPU register. If you perform an operation on two variables, such as addition, the two register quantities are added and the result stored back into a register. The vectorized version of this example would first fill a register with multiple variables to be added, which is called "packing" the register. For example, on processors supporting Intel AVX, up to eight single, precision, floating-point data elements can be packed into one register. Then, using one SIMD instruction, these data elements can be combined with another packed register full of elements, generating multiple results at once. Being able to do these operations in parallel rather than separately can result in significant performance gains for suitable code.

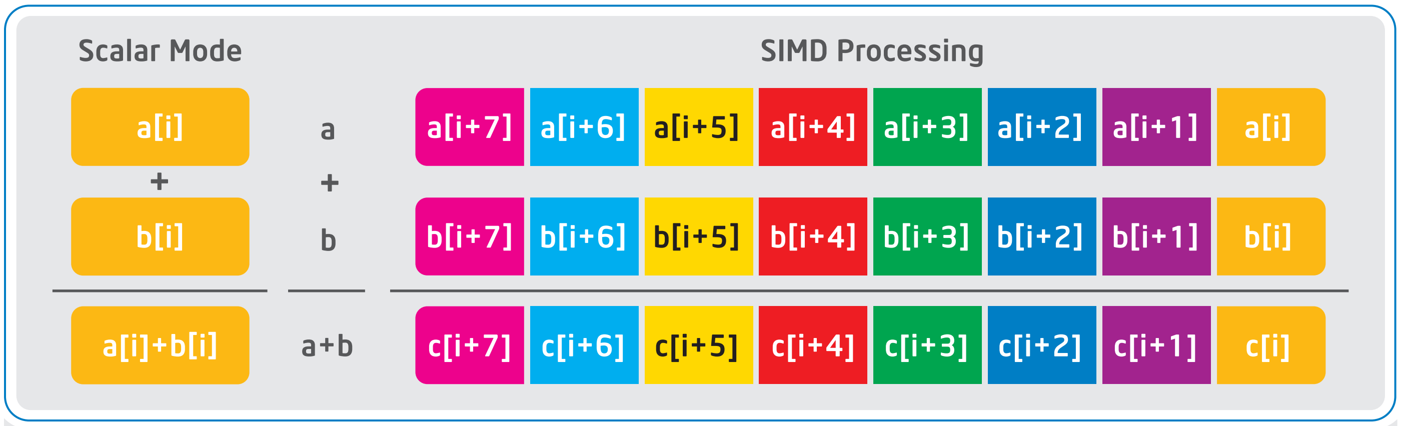

Scalar Mode
SIMD Processing
a[i]
+
b[i]
a[i]+b[i]
a
+
b
a+b
a[i+7] a[i+6] a[i+5] a[i+4] a[i+3] a[i+2] a[i+1] a[i]
b[i+7] b[i+6] b[i+5] b[i+4] b[i+3] b[i+2] b[i+1] b[i]
c[i+7] c[i+6] c[i+5] c[i+4] c[i+3] c[i+2] c[i+1] c[i]

Figure 1: A SIMD addition applied to array elements

"Vectorization allows you to process data in parallel within a single CPU core. Alone, it can provide a significant performance boost and it can also be combined with threading and/or cluster parallelism… Using one of our recommended methods to vectorize your code gives you two advantages: increased performance now, and forward scaling for the future."

As **Figure 1** shows, vectorization is typically applied to array or vector data elements that are processed in loops. Besides addition, there are SIMD instructions for many mathematical functions, logical operations, and even string operations. The SIMD instruction sets are designed for applications that process large data sets. Applications in the scientific, engineering, financial, media, and graphical domain areas may be candidates for vectorization, as well as any others fitting the description above.

How Do I Vectorize My Code?

Developers can access the SIMD instructions in their applications in a variety of ways. Traditionally, vectorization has been accomplished by manipulating SIMD instructions and registers directly, using assembly code or intrinsics provided by the compiler. This method required developers to become experts in SIMD architecture and to hand-tune the code for various CPUs. In addition to requiring significant effort to develop and maintain the code, vectorizing using assembly or intrinsics also has the disadvantage that code is not portable across compilers. Because code must be written for a targeted set of SIMD hardware, the vectorization achieved with this method would *not* scale forward, meaning it must be re-implemented for new CPUs.

Fortunately, today you can choose from several other vectorization methods that require less effort and *do* scale forward. The techniques below rely on a vectorizing compiler, and we recommend our Intel® Compiler, available for C++ or Fortran. The Intel Compiler includes an advanced vectorizer, as well as several options, reports, and extensions to support vectorization. Our "six steps to vectorization" process utilizes several of these features.

Methods that Scale Forward

When you vectorize your code with one of the methods below, it will be vectorizable without changes for future architectures and CPUs as well. When compiling on those future architectures, the Intel Compiler (or a compiler fully supporting the method) will make the appropriate choice about how to vectorize.

> Using the Intel Compiler auto-vectorizer: When enabled, the compiler auto-vectorizer will look for opportunities to vectorize loops with no changes required to your source code. This method may be all that is needed for vectorization-friendly code. However, the compiler will not vectorize loops if it can't prove that it will be safe to perform the operations in parallel. For this reason, you may see even more vectorization by following up this method with one of the techniques below.

> Using a high-level construct provided by Intel® Cilk™ Plus: Cilk Plus is a set of language extensions for C and C++ (and in one case, Fortran) that support parallelism and vectorization. Currently Cilk Plus is fully supported by the Intel Compiler, and partially implemented in GCC. It is an open standard (for more details, see cilk.com). Cilk Plus provides a variety of constructs that can be applied to your code to give the compiler information that it needs to vectorize. Many of these constructs are simple to add to your code—involving only a change in notation or the addition of a pragma.

> Using a high-level Fortran construct: Fortran includes several vectorization-friendly features, such as array notations and the DO CONCURRENT loop. The Cilk Plus mandatory vectorization directive (!DIR$ SIMD) is also available for Fortran. Like Cilk Plus, these constructs are used to give information to the compiler so that it knows when it can vectorize.

BLOG highlights

Serial Equivalence of Cilk Plus programs

ROBERT GEVA, Intel

There is a trend in the C++ community to grow capabilities thru more libraries and as much as possible, avoid adding language keywords. Consistent with these trends are the Intel® Threading Building Blocks and the Microsoft Parallel Patterns Library.* The question arises, then, why implement Intel® Cilk™ Plus as language extensions rather than a library?

One of the answers is that the language is implemented by compilers, and compilers can provide certain guarantees. One such guarantee is serial equivalence. Every Cilk Plus program that uses the three taking keywords for parallelism has a well-defined serial elision. The serial elision is defined by replacing each cilk_spawn and each cilk_sync with white spaces, and each cilk_for with the for keyword. Obviously, the serial elision of a Cilk Plus program is a valid C/C++ program.

A program has a determinacy race if two logically parallel strands both access the same memory location and at least one of them modifies the memory location. If a Cilk Plus parallel program has no determinacy race, then it will produce the same results as its serial elision. What are the compiler's contributions to the serial equivalence guarantees? Consider the following code illustration ...

SEE THE REST OF ROBERT'S BLOG:

Visit **Go-Parallel.com**

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

The Six-Step Process for Vectorizing Your Application

The methods above, and the Intel Compiler are part of a six-step process we have designed for vectorizing an application. You can try this method on your entire application, or selectively on parts of it. To help you follow the process we have documented the steps online in our *Vectorization Toolkit*. The toolkit also includes links to additional resources for each step. Check out the toolkit online at: http://software.intel.com/en-us/articles/vectorization-toolkit/.

Step 1. Measure Baseline Release Build Performance

You need to have a baseline for performance so you know if changes to introduce vectorization are effective. In addition, you should have a baseline to set your ultimate performance goals, so that you know when you have achieved them.

A release build should be used instead of a debug build. A release build will contain all the optimizations in your final application, and may alter the hotspots or even the code that is executed. For instance, a release build may optimize away a loop in a hotspot that otherwise would be a candidate for vectorization.

A release build is the default in the Intel Compiler. You have to specifically turn off optimizations by doing a DEBUG build on Windows* (or using the -Zi switch) or using the -Od switch on Linux* or Mac OS* X. If using the Intel Compiler, ensure you are using optimization levels 2 or 3 (-O2 or –O3) to enable the auto-vectorizer.

Step 2. Determine Hotspots Using Intel® VTune™ Amplifier XE

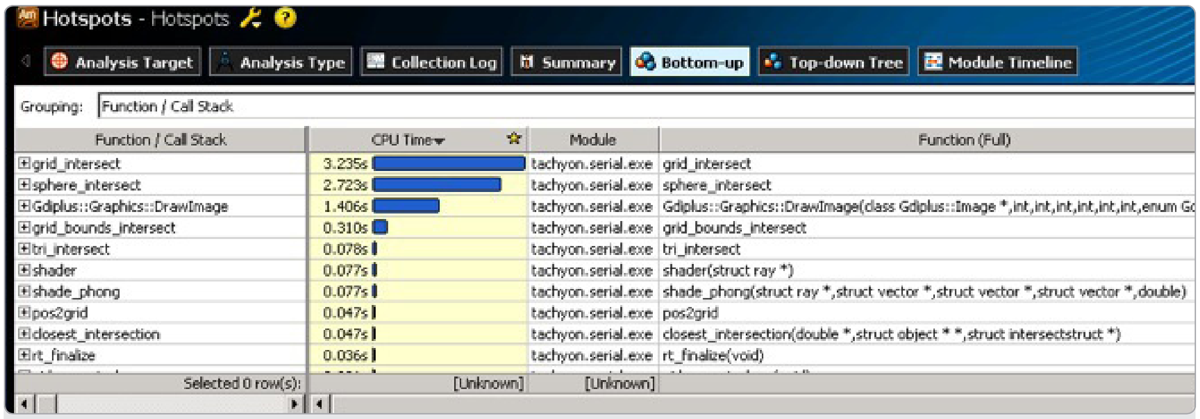
You can use Intel® VTune™ Amplifier XE, our performance profiler, to find the most time-consuming functions in your application. The "Hotspots" analysis type is recommended; although "Lightweight Hotspots" would work as well (it will profile the whole system, as opposed to just your application).

Identifying which areas of your code are taking the most time will allow you to focus your optimization efforts in the areas where performance improvements will have the most effect. Generally you want to focus only on the top few hotspots, or functions taking at least 10% of your application's total time. Make note of the hotspots you want to focus on for the next step.

Step 3. Determine Loop Candidates Using Intel Compiler Vec-Report

The vectorization report (or vec-report) of the Intel Compiler can tell you whether or not each loop in your code was vectorized. Ensure that you are using Compiler optimization level 2 or 3 (-O2 or –O3) to enable the auto-vectorizer. Run the vec-report and look at the output for the hotspots you determined in **Step 2**. If there are loops in your hotspots that did not vectorize, check whether they have math, data processing, or string calculations on data in parallel (for instance in an array). If they do, they might benefit from vectorization. Move to **Step 4** if any candidates are found.

To run the vec-report, use the "-vec-report2" or "/Qvec-report2" option.



Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Module	Function (Full)
⊞grid_intersect	3.235s	tachyon.serial.exe	grid_intersect
⊞sphere_intersect	2.723s	tachyon.serial.exe	sphere_intersect
⊞Gdiplus::Graphics::DrawImage	1.406s	tachyon.serial.exe	Gdiplus::Graphics::DrawImage(class Gdiplus::Image *,int,int,int,int,int,int,enum Gd
⊞grid_bounds_intersect	0.310s	tachyon.serial.exe	grid_bounds_intersect
⊞tri_intersect	0.078s	tachyon.serial.exe	tri_intersect
⊞shader	0.077s	tachyon.serial.exe	shader(struct ray *)
⊞shade_phong	0.077s	tachyon.serial.exe	shade_phong(struct ray *,struct vector *,struct vector *,struct vector *,double)
⊞pos2grid	0.047s	tachyon.serial.exe	pos2grid
⊞closest_intersection	0.047s	tachyon.serial.exe	closest_intersection(double *,struct object * *,struct intersectstruct *)
⊞rt_finalize	0.036s	tachyon.serial.exe	rt_finalize(void)
Selected 0 row(s):	[Unknown]	[Unknown]	

Figure 2: Intel® VTune™ Amplifier XE Hotspots view

```
.\main.cpp(30): warning : LOOP WAS VECTORIZED.
.\scalar_dep.cpp(80): warning : LOOP WAS VECTORIZED.
.\main.cpp(47): warning : loop was not vectorized: not inner loop.
.): warning : loop was not vectorized: nonstandard loop is not a vectorization candidate.
.): warning : loop was not vectorized: nonstandard loop is not a vectorization candidate.
.): warning : loop was not vectorized: existence of vector dependence.
.): warning : loop was not vectorized: not inner loop.
.): warning : loop was not vectorized: existence of vector dependence.
.): warning : loop was not vectorized: not inner loop.
.): warning : loop was not vectorized: existence of vector dependence.
.): warning : loop was not vectorized: not inner loop.
```

Figure 3: Output from the Intel® Compiler vec-report

Note that the Intel Compiler can be run on just a portion of the code and will be compatible with the native compilers (gcc on Linux and Mac OS X and Microsoft Visual Studio* on Windows).

Step 4.
Get Advice Using the Intel Compiler GAP Report and Toolkit Resources

Run the Intel Compiler Guided Auto-parallelization (or GAP) report to see suggestions from the compiler on how to vectorize your loop candidates from **Step 3**. Examine the advice and refer to additional toolkit resources as needed.

Run the GAP report using the "guide" or "/Qguide" options for the Intel Compiler.

Note: You can run the Intel Compiler on just parts of your application if needed.

```
for (i=0; i<n; i++) {

  if (A[i] > 0) { b=A[i]; A[i] = 1 / A[i]; }
  if (A[i] > 1) { A[i] += b;}
}
```

Figure 4: Example of a nonvectorizing loop

```
scalar_dep.cpp(80): warning #30515: (VECT)
Assign a value to the variable(s) "b" at
the beginning of the body of the loop in
line 80. This will allow the loop to be
vectorized. [VERIFY] Make sure that, in the
original program, the variable(s) "b"
read in any iteration of the loop has been
defined earlier in the same iteration.
```

Figure 5: GAP advice for the loop in Figure 4

Step 5.
Implement GAP Advice and Other Suggestions
(Such as Using Elemental Functions and/or Array Notations)

Now that you know the GAP report suggestions for your loop, it's time to implement them if possible.

The report may suggest making a code change. Make sure the change would be "safe" to do. In other words, make sure the change does not affect the semantics or safety of your loop. One way to ensure that the loop has no dependencies that may be affected is to see if executing the loop in backwards order would change the results. Another is to think about the calculations in your loop being in a scrambled order. If the results would be changed, your loop has dependencies and vectorization would not be "safe." You may still be able to vectorize by eliminating dependencies in this case.

Modify your source to give additional information to the Compiler or optimize your loop for better vectorization.

At this point you may introduce some of the high-level constructs provided by Cilk Plus or Fortran. You can find links to full details of the available constructs in the online Vectorization Toolkit.

Step 6:
Repeat!

Iterate through the process as needed until performance is achieved or until there are no good candidates left in your hotspots.

```
for (i=0; i<n; i++) {
      b = A[i];
   if (A[i] > 0) {A[i] = 1 / A[i];}
   if (A[i] > 1) {A[i] += b;}
}
```

Figure 6: The loop from Figure 4, modified to vectorize

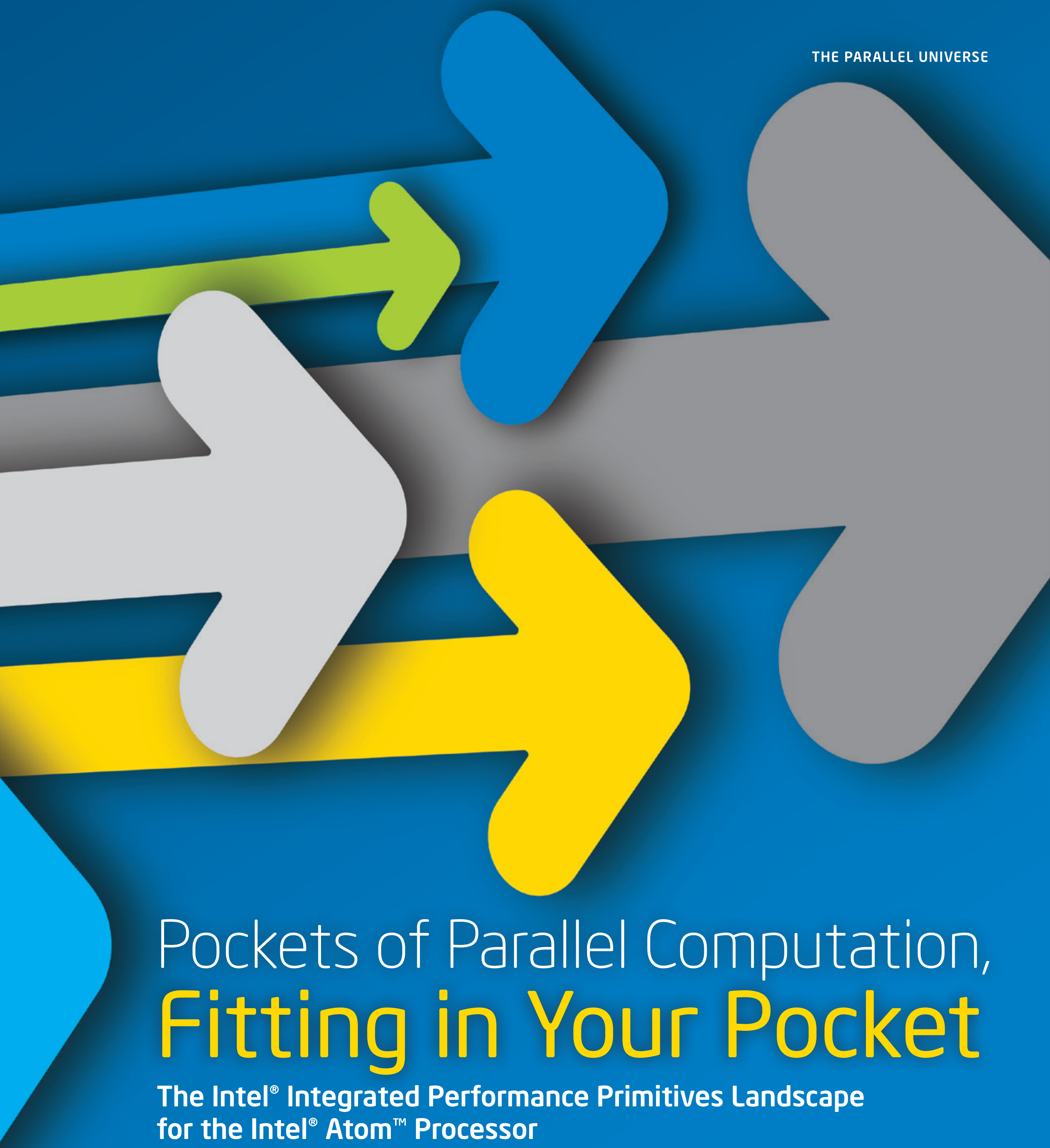
Conclusion

Vectorization allows you to process data in parallel within a single CPU core. Alone, it can provide a significant performance boost and it can also be combined with threading and/or cluster parallelism. Using vectorization is important for performance on current Intel CPUs, such as those in the Intel® Xeon® and Intel® Core™ processor families. In the future, it will be an even more critical component of performance for those processors, as well as for utilizing the Intel® Many Integrated Core architecture.

Using one of our recommended methods to vectorize your code gives you two advantages: increased performance now, and forward scaling for the future. Please visit our Vectorization Toolkit to see the latest advice, processes, and resources to help with your vectorization effort. See the webinar: Future-Proof Your Application's Performance › ☐

Sign up for future issues | Share with a friend

intel
Intel® Integrated Performance Primitives
intel
Intel®
Integrated
Performance
Primitives



Pockets of Parallel Computation, Fitting in Your Pocket

The Intel® Integrated Performance Primitives Landscape for the Intel® Atom™ Processor

by Robert Mueller, Noah Clemons, and Paul Fischer
Software Technical Consulting Engineers, Intel

Discover how performance libraries, such as the Intel® Integrated Performance Primitives (Intel® IPP), impact applications for small form-factor devices, streamlining and unifying the computational execution flow for data-intensive tasks.

Employing performance library functions in applications running on small form-factor devices—otherwise known as *intelligent systems*—based on the Intel® Atom™ processor can be a great way to streamline and unify the computational execution flow for data-intensive tasks. This minimizes the risk of data stream timing issues and so called *heisenbugs*. Heisenbugs are hard to reproduce, and often seemingly random, runtime issues caused by the finely orchestrated timing of different System on Chip (SoC) components and digital signal processor (DSP) devices getting out of tune.

Performance libraries such as the Intel® Integrated Performance Primitives (Intel® IPP) contain highly optimized algorithms and code for common functions including signal processing, image processing, video and audio encoding and decoding, cryptography, data compression, speech coding, and computer vision. Advanced instruction sets help the developer take advantage of new processor features that are specifically tailored for certain applications. One calls the Intel IPP, as if it is a black box pocket of computation for a low-power or embedded device: 'in' flows the data and 'out' receives the result. In this fashion, using Intel IPP can take the place of many processing units created for specific computational tasks. Intel IPP excels in a wide variety of domains (**Figure 1**) where the Intel Atom processor for intelligent systems is utilized:

Without the benefit of highly optimized performance libraries, developers would need to carefully hand-optimize computationally intensive functions to obtain adequate performance. This optimization process is complicated, time consuming, and must be updated with each new processor generation. Intelligent systems often have a long lifetime in the field and there is a high maintenance effort to hand-optimize functions.

As seen in **Figure 1**, signal processing and advanced vector math are the two function domains that are most in demand across the different types of intelligent systems. Frequently, a digital signal processor (DSP) is employed to assist the general purpose processor with these types of computational tasks. A DSP may come with its own well-defined application interface and library function set. However, it is usually poorly suited for general purpose tasks. DSPs are designed to quickly execute basic mathematical operations (add, subtract, multiply, and divide). The DSP repertoire includes a set of very fast multiply and accumulate (MAC) instructions to address matrix math evaluations that appear frequently in convolution, dot product, and other multi-operand math operations. The MAC instructions that comprise much of the code in a DSP application are the equivalent of Intel® Supplemental Single Instruction Multiple Data Streaming Extension 3 (Intel® SSSE3) instructions. Like the MAC instructions on a DSP, these Intel SSSE3 instructions perform mathematical operations very efficiently on vectors and arrays of data. Unlike a DSP, the Single Instruction Multiple Data (SIMD) instructions on an Intel Atom Processor are easier to integrate into applications using complex vector and array mathematical algorithms, since all computations execute on the same processor and are part of a unified logical execution stream.

For example, an algorithm that changes image brightness by adding (or subtracting) a constant value to each pixel of that image must read the RGB values from memory, add (or subtract) the offset, and write the new pixel values back to memory. When using a DSP coprocessor, that image data must be packaged for the DSP (placed in a memory area that is accessible by the DSP), signaled to execute the transformation algorithm, and finally returned to the general purpose

	Aerospace and Defense	Network Equipment	Consumer	Medical	Industrial	Automotive
Data Integrity	■	■		■		
Realistic Rendering			■	■		
String Processing	■	■				
Matrix/Vector Math	■	■	■	■	■	■
Speech Coding	■	■	■			■
Audio Coding	■		■			■
Video Compression	■		■	■		■
Image Compression	■		■	■		
Computer Vision	■				■	
Image Color Conversion			■	■		
Image Processing			■	■	■	■
Signal Processing	■	■	■	■	■	■
Cryotography	■	■		■		
Data Compression	■	■		■		

Figure 1: Intel® Integrated Performance Primitives function domains for the embedded space

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

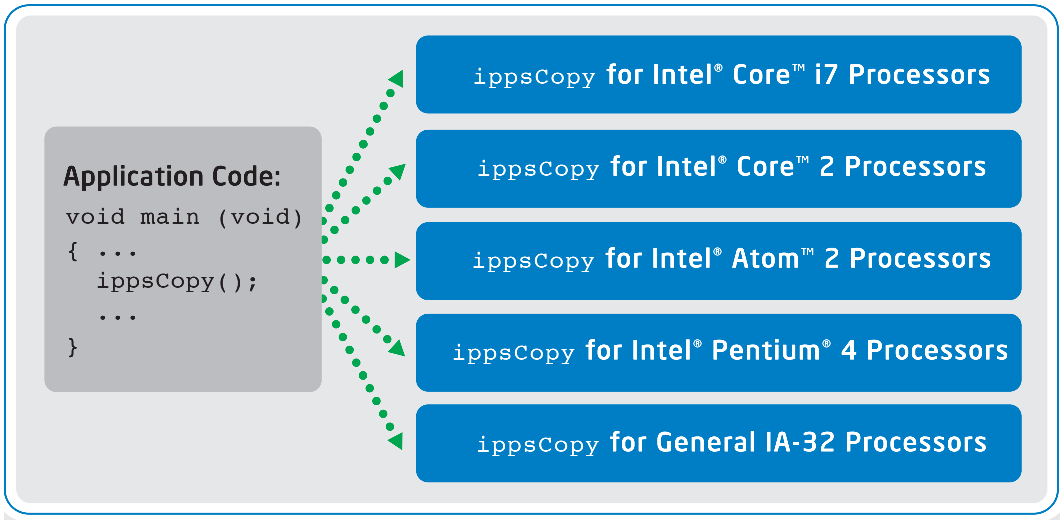

Application Code:
void main (void)
{ ...
ippsCopy();
...
}
ippsCopy for Intel® Core™ i7 Processors
ippsCopy for Intel® Core™ 2 Processors
ippsCopy for Intel® Atom™ 2 Processors
ippsCopy for Intel® Pentium® 4 Processors
ippsCopy for General IA-32 Processors

Figure 2: Library dispatch for processor targets

processor. Using a general purpose processor with SIMD instructions simplifies this process of packaging, signaling, and returning the data set. The Intel IPP library primitives are optimized to match each SIMD instruction set architecture so that multiple versions of each primitive exist in the library.

Intel IPP can be reused over a wide range of Intel® architecture-based processors and, due to automatic dispatching, the developer's code base will always pick the execution flow optimized for the architecture in question without having to change the underlying function call (**Figure 2**). This is especially helpful if an embedded system employs both an Intel® Core™ processor for data analysis and aggregation and a series of Intel® Atom™ processor based SoCs for data preprocessing and collection. In this scenario, the same code base may be used in part on both the Intel Atom processor SoCs in the field and the Intel Core processor in the central data aggregation point.

With specialized SoC components for data streaming and I/O handling, combined with a limited user interface, one may think that there are not a lot of opportunities to take advantage of optimizations and/or parallelism with the Intel Atom Processor line, but that is not the case. There is room for:

- Heterogeneous asynchronous multi-processing (AMP) based on different architectures, and
- Synchronous multi-processing (SMP), taking advantage of the Intel® Hyper-Threading Technology and dual-core design used with the latest generation of Intel Atom processors designed for low-power intelligent systems.

Both concepts often coexist in the same SoC. Code with failsafe real-time requirements is protected within its own wrapper (managed by a modified round-robin real-time scheduler), while the rest of the operating system (OS) and application layers are managed using standard SMP multiprocessing concepts. Intel Atom processors contain two Intel Hyper-Threading Technology-based cores, and may contain an additional two physical cores resulting in a quad-core system. In addition, Intel Atom processors support the Intel SSSE3 instruction set. A wide variety of Intel IPP functions found at http://software.intel.com/en-us/articles/new-atom-support/ are tuned to take advantage of Intel Atom processor architecture specifics as well as Intel SSSE3 (**Figure 3**).

Throughput intensive applications can benefit from the ease of use of Intel SSSE3 vector instructions, and parallel execution of multiple data streams through extra-wide vector registers for SIMD processing. As just mentioned, modern Intel Atom processor designs provide up to four virtual processor cores. This fact makes threading an interesting proposition. While there is no universal threading solution that is best for all applications, Intel IPP has been designed to be thread-safe:

- Primitives within the library can be called simultaneously from multiple threads within your application
- The threading model you choose may have varying granularity
- Intel IPP functions can take advantage of the available processor cores directly via OpenMP*.
- Intel IPP functions can be combined with OS-level threading using native threads, Intel® Cilk™ Plus, or any other member of Intel's family of parallel models.

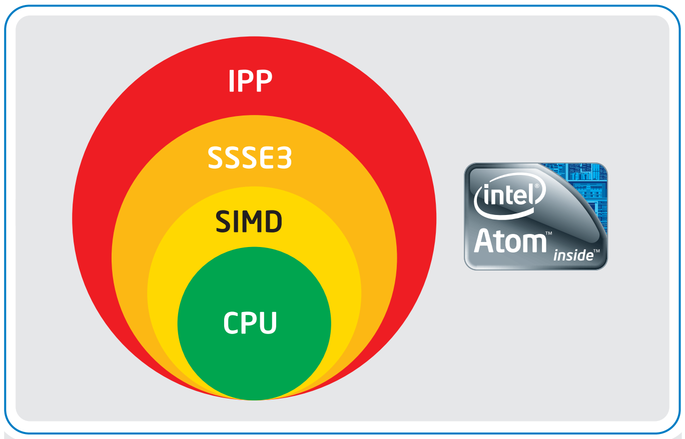


Figure 3: Intel® Integrated Performance Primitives is tuned to take advantage of the Intel® Atom™ Processor and the Intel® Supplemental Single Instruction Multiple Data Streaming Extension 3 instruction set

Sign up for future issues | Share with a friend

The quickest way to multithread an application that uses the Intel IPP extensively is to take advantage of the OpenMP threading built into the library. No significant code rework is required. However, only about 15 to 20 percent of Intel IPP functions are threaded. In most scenarios. It is therefore preferable to look to higher-level threading to achieve optimum results. Since the library primitives are thread safe, the threads can be implemented directly in the application, and the performance primitives can be called directly from within the application threads. This approach provides additional threading control and meets the exact threading needs of the application (**Figure 4**).

When applying threading at the application level, it is generally recommended to disable the library's built-in threading. Doing so eliminates competition for hardware thread resources between the two threading models, and thus avoids oversubscription of software threads for the available hardware threads.

Besides performance and maintainability, footprint and power consumption are also important considerations when developing a software stack for low-power intelligent systems. On a per instruction basis, SIMD instructions can consume slightly more energy than non-SIMD instructions. However, because they execute more efficiently—allowing an application to complete in less time—the net result can be less total energy consumed to complete the job. Thus, the use of SIMD instructions allows improved performance of critical workloads, while draining the battery less.

Intelligent systems frequently must work within confined memory and storage limits. Thus, the need to control the application footprint on a storage device (total binary size) or in memory during execution can be critical. If an application cannot be executed within the available resources, the reliability of the entire system can be in jeopardy.

Intel IPP provides flexibility in linkage models to strike the right balance between portability and footprint management (**Table 1**).

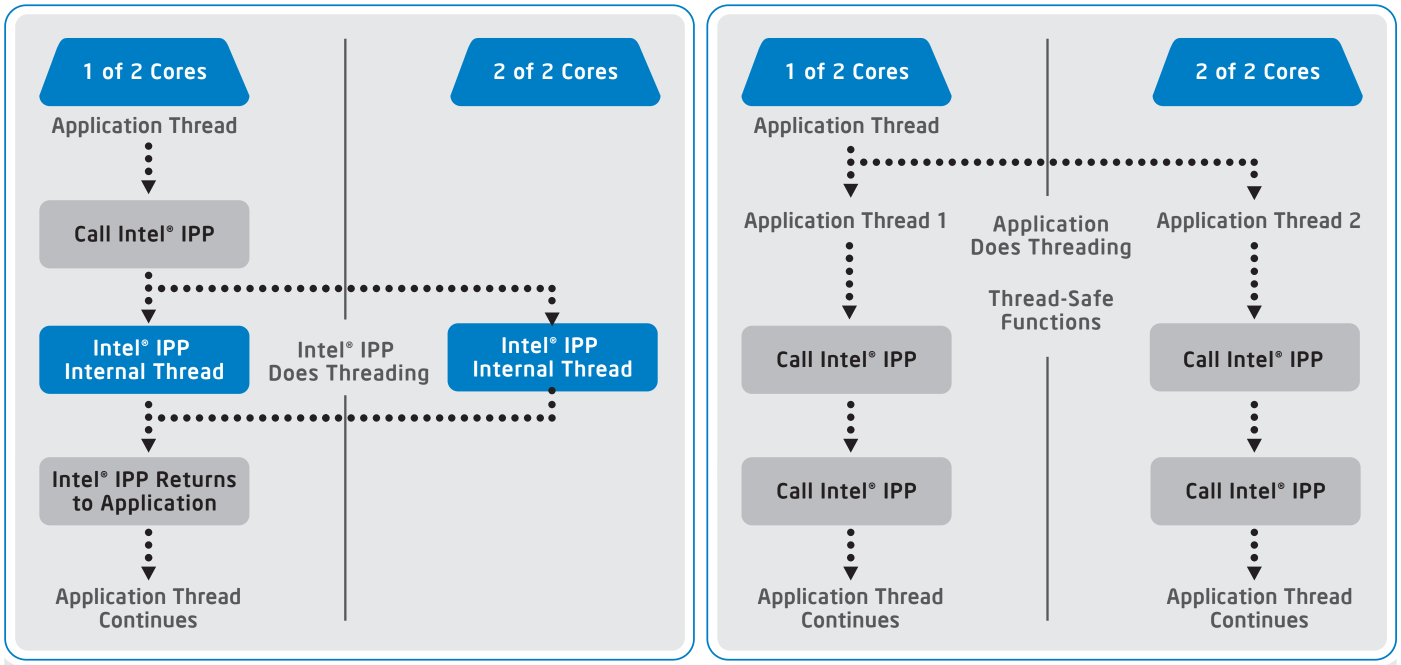

1 of 2 Cores
2 of 2 Cores
Application Thread
Call Intel® IPP
Intel® IPP Internal Thread
Intel® IPP Does Threading
Intel® IPP Internal Thread
Intel® IPP Returns to Application
Application Thread Continues
1 of 2 Cores
2 of 2 Cores
Application Thread
Application Thread 1
Application Does Threading
Application Thread 2
Thread-Safe Functions
Call Intel® IPP
Call Intel® IPP
Call Intel® IPP
Call Intel® IPP
Application Thread Continues
Application Thread Continues

Figure 4: Function-level threading and application-level threading using Intel® Integrated Performance Primitives

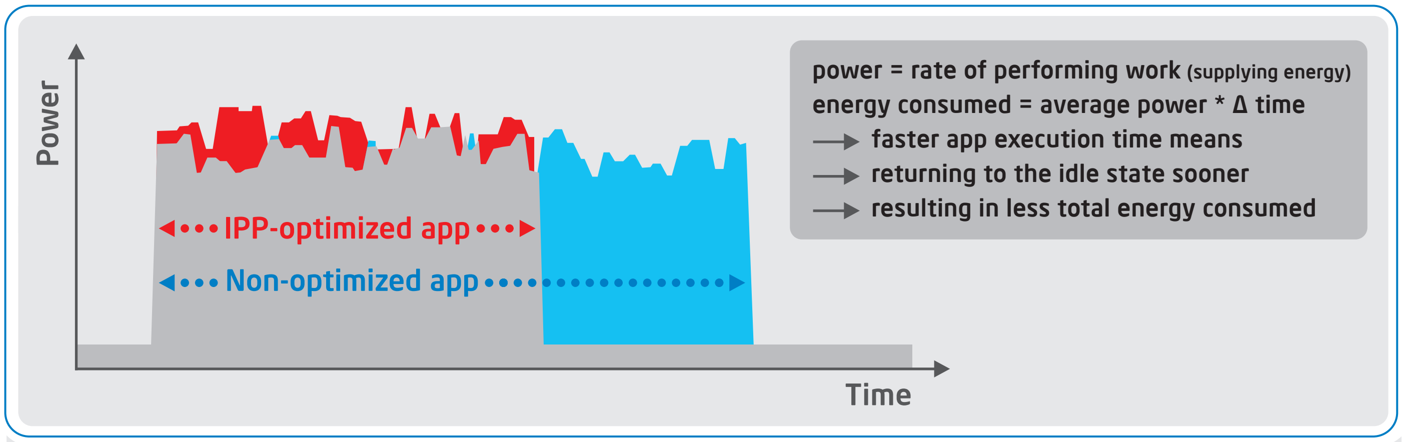


Figure 5: Impact of optimized library usage on power-consumption

The standard dynamic and dispatched static models are the simplest options to use in building applications with Intel IPP. The standard dynamic library includes the full set of processor optimizations and provides the benefit of runtime code sharing between multiple Intel IPP-based applications. Detection of the runtime processor and dispatching to the appropriate optimization layer is automatic.

	Standard Dynamic	Custom Dynamic	Dispatched Static	Non-Dispatched Static
Optimizations	All SIMD sets	All SIMD sets	All SIMD sets	Single SIMD set
Distribution	Executable(s) and standard Intel IPP DLLs	Executable(s) and custom DLLs	Executable(s) only	Executable(s) only
Library Updates	Redistribute as-is	Rebuild and redistribute	Recompile application and redistribute	Rebuild custom library, recompile application, and redistribute
Executable Only Size	Small	Small	Large	Medium
Total Binary Size	Large	Medium	Medium	Small
Kernel Mode	No	No	Yes	Yes

Table 1: Intel® Integrated Performance Primitives linkage model comparison

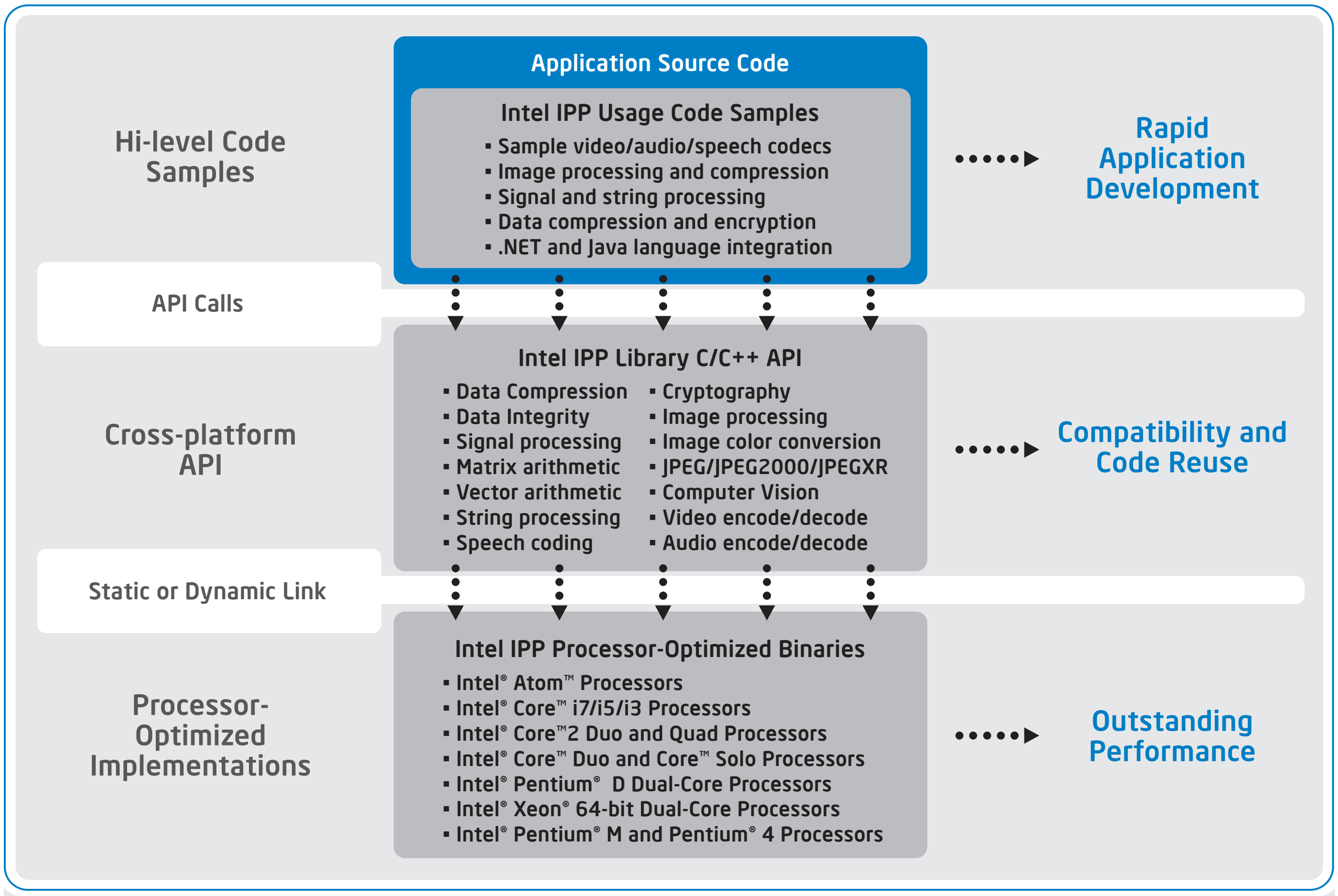

Hi-level Code Samples
Application Source Code
Intel IPP Usage Code Samples
▪ Sample video/audio/speech codecs
▪ Image processing and compression
▪ Signal and string processing
▪ Data compression and encryption
▪ .NET and Java language integration
Rapid Application Development
API Calls
Cross-platform API
Intel IPP Library C/C++ API
▪ Data Compression
▪ Data Integrity
▪ Signal processing
▪ Matrix arithmetic
▪ Vector arithmetic
▪ String processing
▪ Speech coding
▪ Cryptography
▪ Image processing
▪ Image color conversion
▪ JPEG/JPEG2000/JPEGXR
▪ Computer Vision
▪ Video encode/decode
▪ Audio encode/decode
Compatibility and Code Reuse
Static or Dynamic Link
Processor-Optimized Implementations
Intel IPP Processor-Optimized Binaries
▪ Intel® Atom™ Processors
▪ Intel® Core™ i7/i5/i3 Processors
▪ Intel® Core™2 Duo and Quad Processors
▪ Intel® Core™ Duo and Core™ Solo Processors
▪ Intel® Pentium® D Dual-Core Processors
▪ Intel® Xeon® 64-bit Dual-Core Processors
▪ Intel® Pentium® M and Pentium® 4 Processors
Outstanding Performance

Figure 6: Intel® Integrated Performance Primitives overview. Many domains are covered along with sample source code for rapid development.

If the number of Intel IPP functions used in your application is small, and the standard shared library objects are too large, using a custom dynamic library may be an alternative.

To optimize for minimal total binary footprint, linking against a non-dispatched static version of the library may be the approach to take. This approach yields an executable containing only the optimization layer required for your target processor. This model achieves the smallest footprint at the expense of restricting your optimization to one specific processor type and one SIMD instruction set. This linkage model is useful when a self-contained application running on only one processor type is the intended goal. It is also the recommended linkage model for use in kernel mode (ring 0) or device driver applications.

The development ecosystem for embedded intelligent systems is not focused exclusively on the dominant operating systems of the desktop and server world. Frequently a SoC may use a custom embedded Linux OS based on something like Yocto Project* (http://www.yoctoproject.org) or Wind River* Linux (http://www.windriver.com) for the application and user interface layer, while other parts of the chipset may be running a fail-safe, real-time operating system (FTOS or RTOS).

Currently, Intel IPP is delivered for and validated against five different operating systems: Microsoft Windows*, Linux, Mac OS* X, QNX Neutrino*, and Wind River VxWorks*. QNX* and VxWorks* are limited to single-threaded static variants of Intel IPP. Application of Intel IPP to a "non-supported" operating system requires that the OS is compatible with the Application Binary Interface (ABI) defined by one of the aforementioned five operating systems, and that memory allocation routines can be accessed through a standard C library or mapped via *glue code* using a special i_malloc interface.

The atomic nature (no locks or semaphores) of the Intel IPP function implementation means that it is safe to use in the deterministic environment of a RTOS. An example of the value of applying the Intel IPP library to an RTOS would be the TenAsys INtime* RTOS for Windows. (http://www.tenasys.com). The INtime RTOS is an OS designed to run alongside Windows, handling real-time requirements on a Windows-based platform. The ABI used by INtime RTOS is compatible with the Windows ABI and employs Windows compatible function calling conventions. Using the Intel IPP in conjunction with such an RTOS expands its appeal by providing the performance of SIMD-based data throughput intensive processing, with determinism usually only characteristic of DSPs.

Intel IPP addresses the needs of the native application developer found in the personal computing world, as well as the intelligent system developer who must satisfy real-time system requirements with the interaction between the application layer and the software stack underneath. By taking Intel IPP into the world of middleware, drivers, and OS interaction, it can also be used for embedded devices with real-time requirements and dominant execution models. The limited dependency on OS libraries and its support for flexible linkage models makes it simple to add to embedded cross-build environments, whether they are RTOS-specific or follow one of the popular GNU*-based cross-build setups like Poky-Linux* or MADDE*.

Developing for intelligent systems and small form factor devices frequently means that native development is not a feasible option. Intel IPP can be easily integrated with a cross-build environment and used with cross-build toolchains that accommodate the flow requirements of many of these real-time systems. Use of Intel IPP allows embedded intelligent systems to take advantage of Intel SSSE3 vector instructions and extra-wide vector registers on the Intel Atom processor. Developers can also meet determinism requirements, without increasing the risks associated with cross-architecture data handshakes of complex SoC architectures.

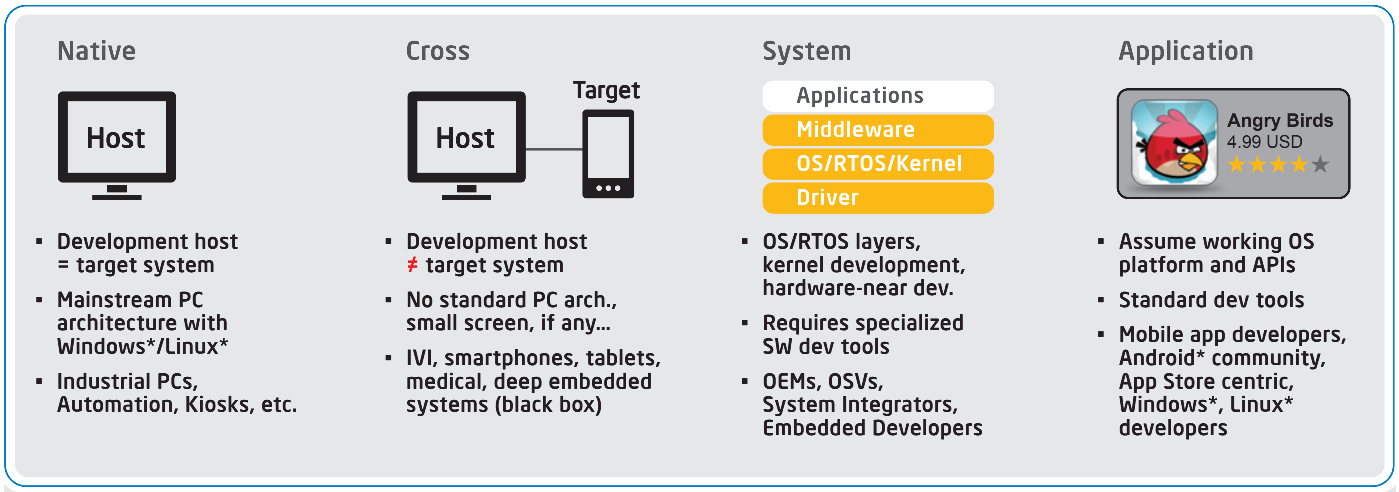

Native
Host
Development host = target system
Mainstream PC architecture with Windows*/Linux*
Industrial PCs, Automation, Kiosks, etc.
Cross
Target
Host
Development host ≠ target system
No standard PC arch., small screen, if any...
IVI, smartphones, tablets, medical, deep embedded systems (black box)
System
Applications
Middleware
OS/RTOS/Kernel
Driver
OS/RTOS layers, kernel development, hardware-near dev.
Requires specialized SW dev tools
OEMs, OSVs, System Integrators, Embedded Developers
Application
Angry Birds
4.99 USD
Assume working OS platform and APIs
Standard dev tools
Mobile app developers, Android* community, App Store centric, Windows*, Linux* developers

Figure 7: Intel® Integrated Performance Primitives can be used over a full range of development setups and software stack targets

Developing for embedded small form factor devices also means that applications with deterministic execution flow requirements have to interface more directly with the system software layer and the OS (or RTOS) scheduler. Software development utilities and libraries for this space need to be able to work with the various layers of the software stack, whether it is the end-user application or the driver that assists with a particular data stream or I/O interface. Intel IPP has minimal OS dependencies and a well-defined ABI to work with the various modes addressed in (Figure 7). One can apply highly optimized functions for embedded signal and multimedia processing across the platform software stack, while taking advantage of the underlying application processor architecture and its strengths—all without redesigning and returning the critical functions with successive hardware platform upgrades.

Getting Started with Intel® Integrated Performance Primitives on Intel® Atom™ Processors

- Purchase Intel IPP, or download a trial copy (http://software.intel.com/en-us/articles/intel-ipp/)
- Check out the free code samples that come with Intel IPP (http://software.intel.com/en-us/articles/intel-integrated-performance-primitives-code-samples/), and see if any match the needs of one of your algorithms (implemented or planned). The Intel IPP code samples are available for the Windows, Linux, and Mac OS* operating systems. Instructions on how to build each sample can be found in sample-specific ReadMe files located in each sample's main directory. Some samples include more detailed documentation in a doc directory, usually in the form of a PDF file.
- If no example code sample matches your needs, take a look at the reference manual (http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/ippxe/ipp_manual_lnx/index.htm), which is organized by function domains, including signal processing, and then by classes of functions within that domain. See **Figure 1** for the 12k Intel IPP functions in fourteen different domains.
- Next, choose a linking option from the many described in **Table 1** that works for you, and you are ready to begin. The article located at http://software.intel.com/en-us/articles/introduction-to-linking-with-intel-ipp-70-library/ provides a more in depth discussion of these options. ☐

BLOG highlights

Some Performance Advantages of Using a Task-Based Parallelism Model

SHANNON CEPEDA, Intel

As part of my focus on software performance, I also support and consult on implementing scalable parallelism in applications. There are many reasons to implement parallelism, as well as many methods for doing it—but this blog is not about either of those things. This blog is about the performance advantages of one particular way of implementing parallelism— and luckily, that way is supported by several models available.

I am talking about task-based parallelism, which means that you design your algorithms in terms of "tasks"(work to be done) instead of the specifics of threads and CPU cores. There are several parallelism models currently available that use tasks: Intel® Threading Building Blocks (TBB), Intel® Cilk Plus, Microsoft* Parallel Patterns Library, and OpenMP 3.0, to name a few. With a task-based library (like the ones mentioned) you can use pre-coded functions, pragmas, keywords, or templates to add parallelism to your application ...

SEE THE REST OF SHANNON'S BLOG:

Visit **Go-Parallel.com**

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

bottleneck
data race
invalid memory access
memory error
threading error
memory leak

TOOLS THAT BOOST .NET APPS RELIABILITY AND PERFORMANCE

Now .NET developers can boost application performance and increase the code quality and reliability needed for high-performance computing and enterprise applications

by Levent Akyil and Asaf Yaffe, *Software and Services Group, Intel Corporation*

Developer-friendly tools that find errors early in the development cycle can have a great payoff. Errors that make it to a released product may damage the product's reputation and are generally very costly to fix. The earlier we can detect and correct hard-to-find errors such as threading (data race and deadlocks) and memory errors (memory leaks) in any application, the easier it's likely to fix them. The confidence and the performance tools aid developers by attributing problems to source code lines, along with call stack and timeline visualization of events, giving developers a clear picture about the issues in their software. Intel® Parallel Studio XE enables .NET developers to identify critical performance problems such as the most time-consuming functions or lines of code, scalability issues, and time spent waiting on synchronization constructs and IO activity. While doing this, Parallel Studio XE reveals potential micro-architectural bottlenecks caused by issues such as branch mispredicts, cache misses, and memory bandwidth problems.

In this article, we highlight examples of how two tools in Intel Parallel Studio XE, the Intel® Inspector XE correctness tool, , and the Intel® VTune™ Amplifier XE performance analysis tool, are valuable to developers of .NET code, native code, and "mixed" (.NET and native) applications during the development cycle. After explaining the current .NET support in Inspector XE and VTune™ Amplifier XE, we demonstrate the key features in action on C# applications.

Current .NET support in Intel® VTune™ Amplifier XE and Intel® Inspector XE

Inspector XE and VTune Amplifier XE products support the analysis of pure .NET applications, as well as "mixed" applications that contain both managed and unmanaged code.

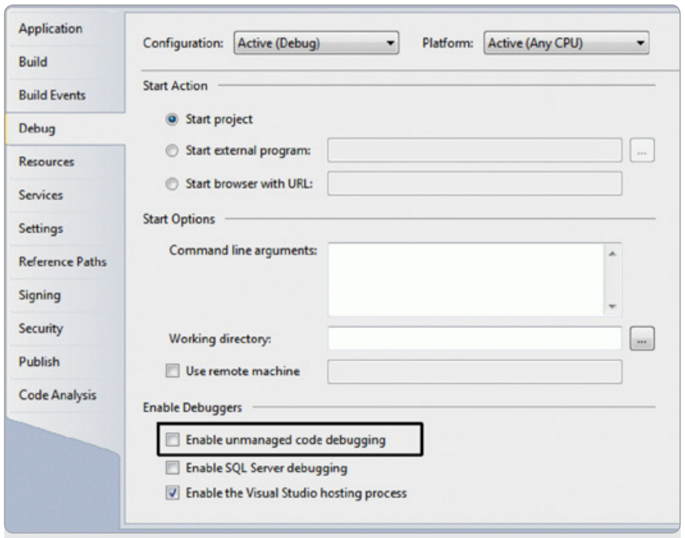
The Inspector XE thread analyzer can detect potential deadlocks and data races in .NET programs, in a similar way as it does for native code. Inspector XE monitors object allocations and accesses to shared memory on the garbage-collected heap and the static data areas, and flags unsynchronized accesses (at least one of which is a write operation) of multiple threads to the same object/class data member as a potential data race. Inspector is also aware of all the .NET 2.0 through .NET 3.5 locking APIs, and can detect deadlocks and lock hierarchy violations.

VTune Amplifier XE assists developers in fine-tuning serial and parallel applications for optimal performance on modern processors and makes it simple for .NET developers to quickly find performance bottlenecks in their pure .NET or mixed applications. VTune Amplifier XE's hotspot analysis highlights the functions and source locations where the application spends most of its execution time. Concurrency and Locks & Waits analyses visualize the work distribution between threads as well as thread synchronization points, and helps users identify work distribution problems and excessive threads synchronization which prevent parallel execution. VTune Amplifier XE can also help developers identify microarchitectural performance issues by using the CPU's Performance Monitoring Unit (PMU) to sample processor events and identify the architectural bottlenecks on a given Intel® processor.

Configuring .NET analysis in VTune Amplifier XE and Inspector XE

Users can select whether to analyze the managed parts (""managed" mode), native parts ("native" mode) or both ("mixed" mode). These types are supported as follows:

- **Native** mode collects data on native code only and does not attribute data to managed code.
- **Managed** mode collects data on managed code only and does not attribute data to native code.
- **Mixed** mode collects and attributes data to both native and managed code as appropriate. Use this option when analyzing managed executables that make calls to native code.
- **Auto** mode automatically detects the type of the target executable. It switches to mixed mode when a managed application is detected and to native mode when a native application is detected.



Configuration: Active (Debug)   Platform: Active (Any CPU)

Start Action

(●) Start project
( ) Start external program:
( ) Start browser with URL:

Start Options

Command line arguments:

Working directory:

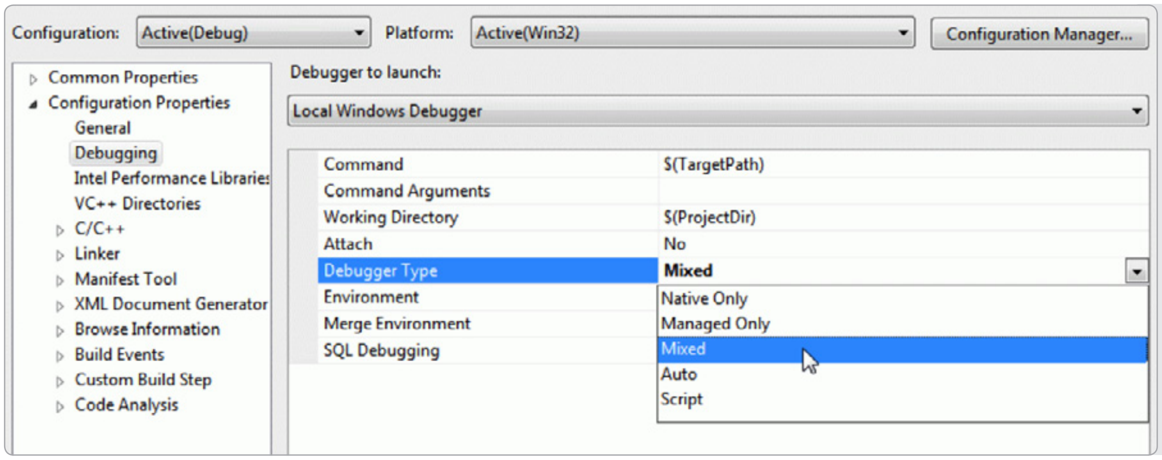
[ ] Use remote machine

Enable Debuggers

[ ] Enable unmanaged code debugging
[ ] Enable SQL Server debugging
[✓] Enable the Visual Studio hosting process

Figure 1: Microsoft Visual Studio* Debug Properties for a .NET project

The way to configure the analysis mode depends on the way one uses the tools. When using the tools from the command line, specify the mode using the "*-mrte-mode*" switch. When using the tools within Microsoft Visual Studio*, the analysis mode is automatically selected based on the active project type: for native projects (C/C++ applications) the default analysis mode in both VTune Amplifier XE and Inspector XE is set to "native.". For .NET projects (C# applications), the default analysis mode in VTune Amplifier XE is set to "managed", while the default analysis mode in Inspector XE is set to "mixed". Users can use the Visual Studio Debug Properties page to select a different mode. To enable "mixed" analysis mode for a .NET project, enable the "unmanaged code debugging" feature (**Figure 1**). Similarly, to enable "mixed" analysis mode for a native project, set the Debugger Type property to "mixed" (**Figure 2**). When using the standalone graphical interface of VTune Amplifier XE or Inspector XE, users can configure the analysis mode from the Project Properties dialog (**Figures 3** and **4**).

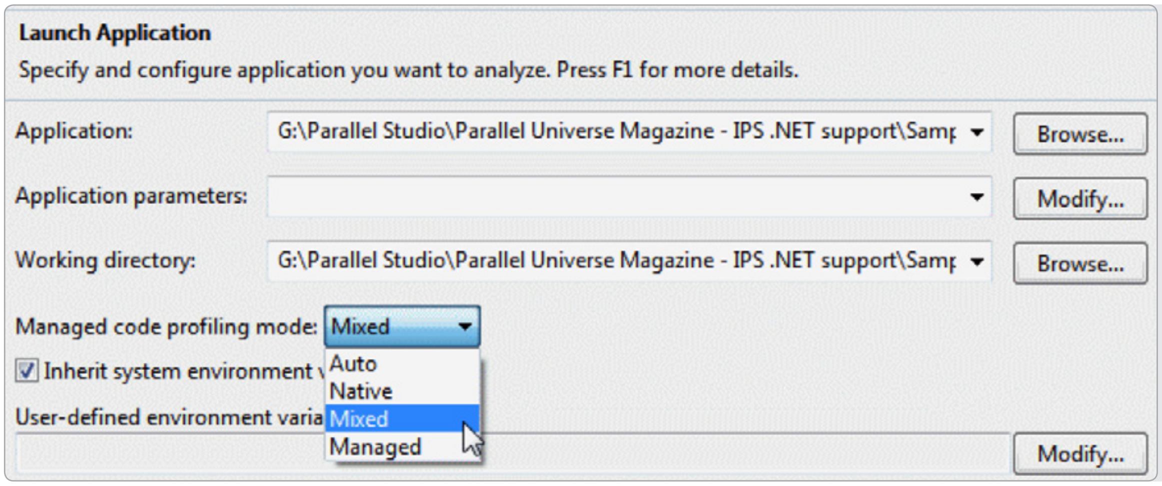

Configuration: Active(Debug)
Platform: Active(Win32)
Configuration Manager...

Common Properties
Configuration Properties
General
Debugging
Intel Performance Libraries
VC++ Directories
C/C++
Linker
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Code Analysis

Debugger to launch: Local Windows Debugger

Command	$(TargetPath)
Command Arguments	
Working Directory	$(ProjectDir)
Attach	No
Debugger Type	Mixed
Environment	
Merge Environment	
SQL Debugging	

Figure 2: Microsoft Visual Studio* Debug Properties for a Native project


Launch Application
Specify and configure application you want to analyze. Press F1 for more details.

Application: G:\Parallel Studio\Parallel Universe Magazine - IPS .NET support\Samp
Application parameters:
Working directory: G:\Parallel Studio\Parallel Universe Magazine - IPS .NET support\Samp
Managed code profiling mode: Mixed
Inherit system environment v
User-defined environment varia

Figure 3: Intel® VTune™ Amplifier XE project properties

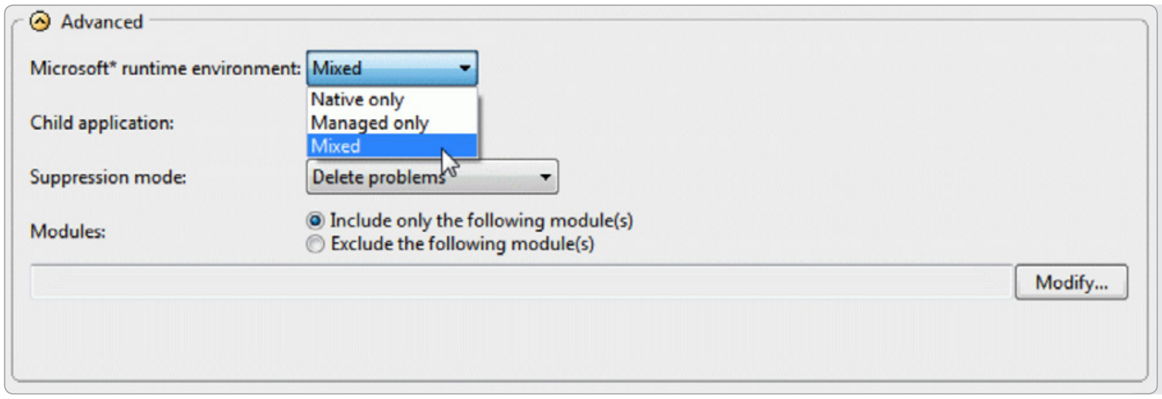

Advanced
Microsoft* runtime environment: Mixed
Child application:
Suppression mode: Delete problems
Modules: Include only the following module(s) / Exclude the following module(s)

Figure 4: Intel® Inspector XE project properties

Sign up for future issues | Share with a friend

```
    namespace VTuneAmplifierXE.Examples
    {
        public class POTENTIAL_MT
        {
            private static Thread[] threads = new Thread[Benchmarks.gThreadCount];
            private static workerThread[] workerThreads = new workerThread[Benchmarks.gThreadCount];
            private static object[] threadparams = new object[Benchmarks.gThreadCount];

            //Start and done signals
            static AutoResetEvent[] goSignals = new AutoResetEvent[Benchmarks.gThreadCount];
            static AutoResetEvent[] doneEvents = new AutoResetEvent[Benchmarks.gThreadCount];

            public static double potential = 0.0;
            public double potentialTotal = 0.0;

            public class workerThread
            {
                private volatile bool stopNow = false;
                private ThreadParameters threadParameters;

                public void RequestStop()
                {
                    stopNow = true;
                }

                private void computePot_mt(int tid)
                {
                    int start, end;
                    double distx, disty, distz, dist;

                    start = threadParameters.chunkBegin;
                    end = threadParameters.chunkEnd;

                    potential = 0.0;

                    for (int i = start; i < end; i++)
                    {
                    for (int j = 0; j < i - 1; j++)
                    {
                         distx = Math.Pow((r[0][j] - r[0][i]), 2);
                         disty = Math.Pow((r[1][j] - r[1][i]), 2);
                         distz = Math.Pow((r[2][j] - r[2][i]), 2);
                         dist = Math.Sqrt(distx + disty + distz);
                         potential += 1.0 / dist;
                    }
                    }
     }
            …

     public void doWork(object parameter)
     {
     threadParameters = (ThreadParameters)parameter;
     Console.WriteLine("Thread: {0} ready to start <{1} - {2}>",
            threadParameters.tid,
            threadParameters.chunkBegin,
            threadParameters.chunkEnd);

     while (!stopNow)
     {
      threadParameters.goSignal.WaitOne();
      computePot_mt(threadParameters.tid);
      threadParameters.eventDone.Set();
     }
     Console.WriteLine("worker thread: terminating gracefully.");
     }
     } // end workerThread
        …
    }// end POTENTIAL_MT
```

Figure 5

Sample Code

To demonstrate how Inspector XE and VTune Amplifier XE support .NET applications, we use a C# program computing the potential energy of a system of particles based on the distance in three dimensions. This is a threaded application which uses the .NET thread pool to create as many threads as the number of cores available. The goal of this article is not to introduce C# threads or how to thread efficiently with .NET framework, but rather to demonstrate how the tools can help to identify threading issues and significantly aid in developing high-performing, scalable parallel applications.

The code below shows the part of the application executed by each worker thread. The `computePot` method is where the action happens. Each thread uses the stored boundaries indexed by the thread's assigned identification number (tid). This helps to fix the start and end range of particles to be used. After each thread initializes its iteration space (start and end values), it starts computing the potential energy of the particles.

Intel® Inspector XE in Action

Let's start by running Inspector XE on our sample code. From the Visual Studio Tools menu, select "Intel Inspector XE 2011," and then "New Analysis" (**Figure 6**). In the Configure Analysis Type page that opens, select "Locate Deadlocks and Data Races" (**Figure 7**), and click the "Start" button to start threading correctness analysis.

Running Inspector XE on our sample code reveals that we have a data race (**Figure 8**).

The Problems pane lists individual problems. Code Locations shows source code locations that are relevant for the selected problems. The Filters pane allows you to filter the Problems view by severity, problem type, module, and source files.

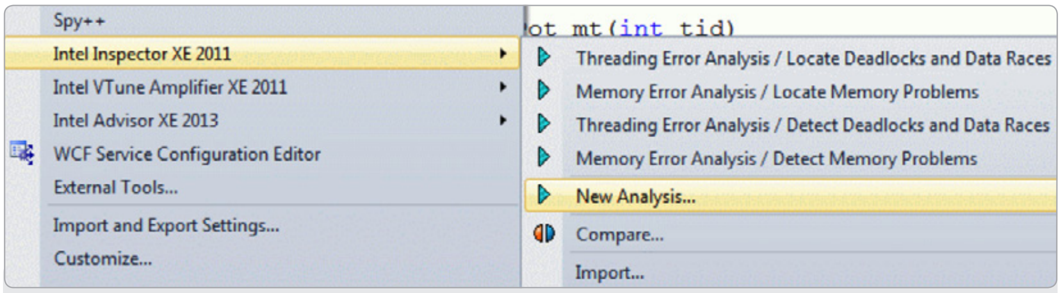

Spy++
Intel Inspector XE 2011
Intel VTune Amplifier XE 2011
Intel Advisor XE 2013
WCF Service Configuration Editor
External Tools...
Import and Export Settings...
Customize...
ot mt(int tid)
Threading Error Analysis / Locate Deadlocks and Data Races
Memory Error Analysis / Locate Memory Problems
Threading Error Analysis / Detect Deadlocks and Data Races
Memory Error Analysis / Detect Memory Problems
New Analysis...
Compare...
Import...

Figure 6: Starting a new Inspector XE analysis from Visual Studio

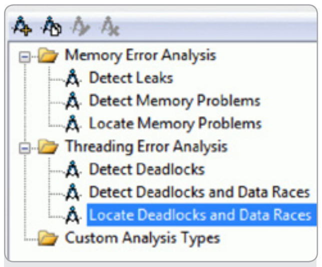
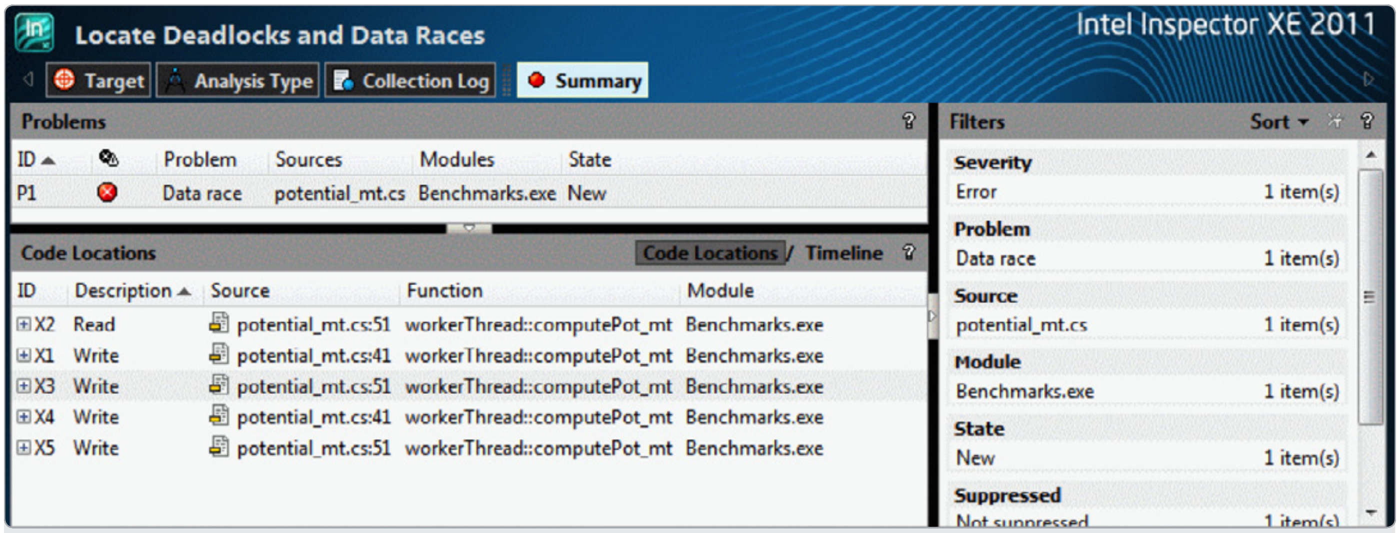


Figure 7: Selecting an Inspector XE analysis type



Problems

ID		Problem	Sources	Modules	State
P1		Data race	potential_mt.cs	Benchmarks.exe	New

Code Locations

ID	Description	Source	Function	Module
X2	Read	potential_mt.cs:51	workerThread::computePot_mt	Benchmarks.exe
X1	Write	potential_mt.cs:41	workerThread::computePot_mt	Benchmarks.exe
X3	Write	potential_mt.cs:51	workerThread::computePot_mt	Benchmarks.exe
X4	Write	potential_mt.cs:41	workerThread::computePot_mt	Benchmarks.exe
X5	Write	potential_mt.cs:51	workerThread::computePot_mt	Benchmarks.exe

Filters

Severity	
Error	1 item(s)
Problem	
Data race	1 item(s)
Source	
potential_mt.cs	1 item(s)
Module	
Benchmarks.exe	1 item(s)
State	
New	1 item(s)
Suppressed	
Not suppressed	1 item(s)

Figure 8: A potential data race Identified by Inspector XE 2011

Double-clicking on the problem takes us to the Sources page (**Figure 9**) where we can further investigate the problem.

This page shows two representative threads of execution that perform an unsynchronized access to a shared memory location, including a detailed call stack for each thread. Using this view we can quickly determine that we have an unsynchronized access to the "potential" static class member—a classic data race. We can double-click any source line to jump directly into the source code and fix the issue.

One trivial solution to this data race is to make sure all access to the "potential" class member is properly synchronized with a lock. However, this solution will introduce a serial region (a critical section) into our parallel code and will likely affect performance. A better solution is for each thread to store a private copy of the potential in a thread local variable, and then accumulate the results to compute the final value. This solution reduces dependencies and synchronization between threads and is likely to speed up the parallel code.

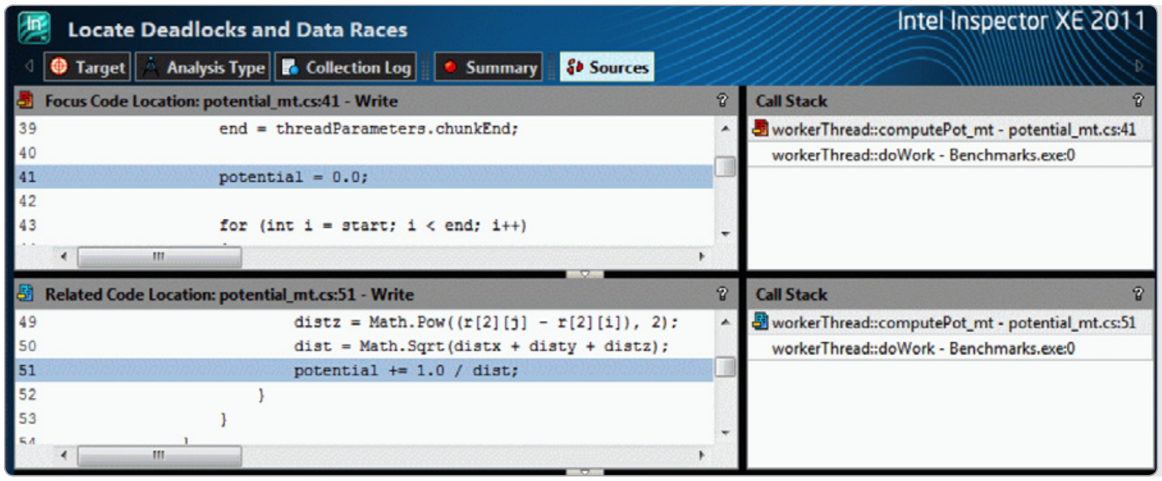
A Few Words about Memory Checker

The Inspector XE memory analyzer is also aware of .NET code and can be used for finding memory leaks and errors in mixed applications that contain managed and native code. While the memory analysis is conducted only for the native portions of the applications (as most of the detectable errors are irrelevant for .NET code), the analysis results show complete stack traces, including the .NET call chain that led to the memory error in native code.

Combining the threading analysis and memory analysis capabilities make Inspector XE a powerful and invaluable tool for analyzing the correctness of complex applications that combine .NET and native code in a single program.

Intel® VTune™ Amplifier XE in Action

Now that we solved our correctness issues, let's start analyzing the performance of our application. **Figure 10** shows how to start the oncurrency analysis within Visual Studio*. If our application is analyzed on a quad-core Intel® 2nd generation core architecture family processor running at 2.5GHz, we get the results summary, as shown in **Figure 11**.



Focus Code Location: potential_mt.cs:41 - Write

```
39                  end = threadParameters.chunkEnd;
40
41                  potential = 0.0;
42
43                  for (int i = start; i < end; i++)
```

Call Stack

workerThread::computePot_mt - potential_mt.cs:41
workerThread::doWork - Benchmarks.exe:0

Related Code Location: potential_mt.cs:51 - Write

```
49                          distz = Math.Pow((r[2][j] - r[2][i]), 2);
50                          dist = Math.Sqrt(distx + disty + distz);
51                          potential += 1.0 / dist;
52                      }
53                  }
```

Call Stack

workerThread::computePot_mt - potential_mt.cs:51
workerThread::doWork - Benchmarks.exe:0

Figure 9: Data race problem details

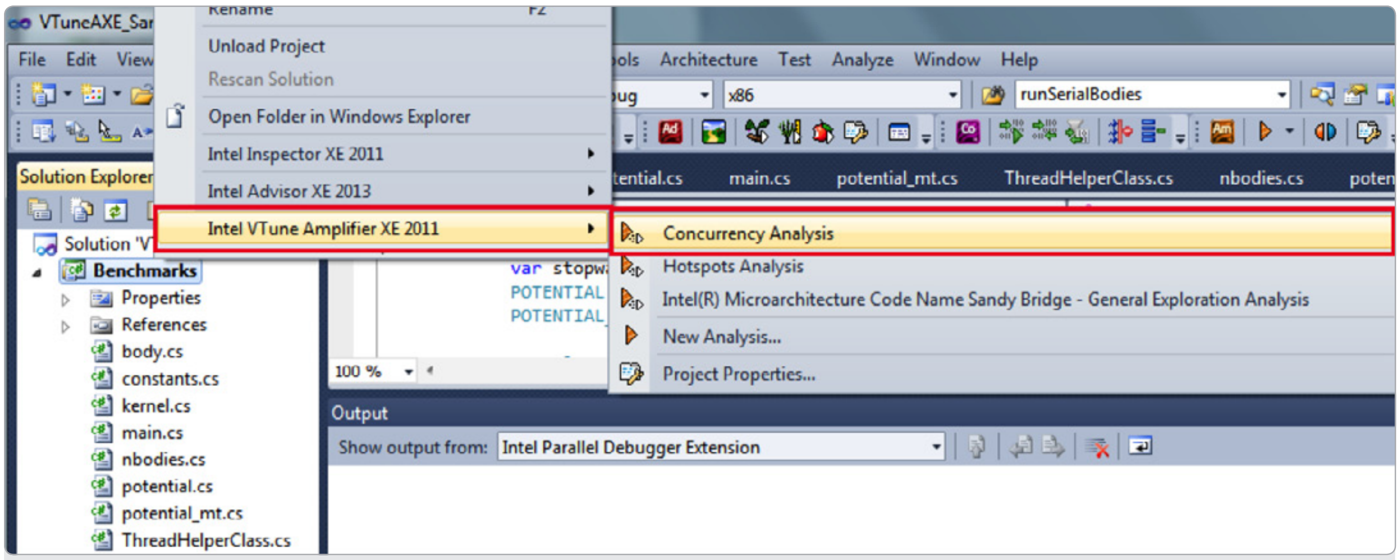
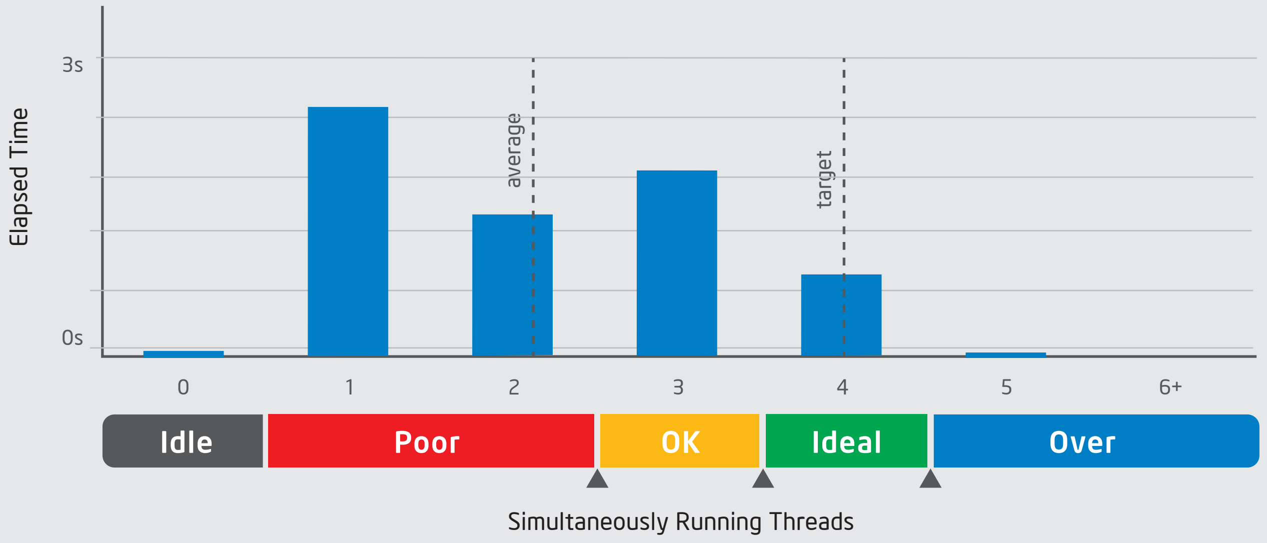


Figure 10: Launching Intel® VTune™ Amplifier XE concurrency analysis within Visual Studio

Thread Concurrency Histogram

This histogram represents a breakdown of the Elapsed Time. It visualizes the percentage of the wall time the specific number of threads were running simultaneously. Threads are considered running if they are either actually running on a CPU or are in the runnable state in the OS scheduler. Essentially, Thread Concurrency is a measurement of the number of threads that were not waiting. Thread Concurrency may be higher than CPU usage if threads are in the runnable state and not consuming CPU time.



CPU Usage Histogram

This histogram represents a breakdown of the Elapsed Time. It visualizes what percentage of the wall time the specific number of CPUs were running simultaneously. CPU Usage may be higher than the thread concurrency if a thread is executing code on a CPU while it is logically...

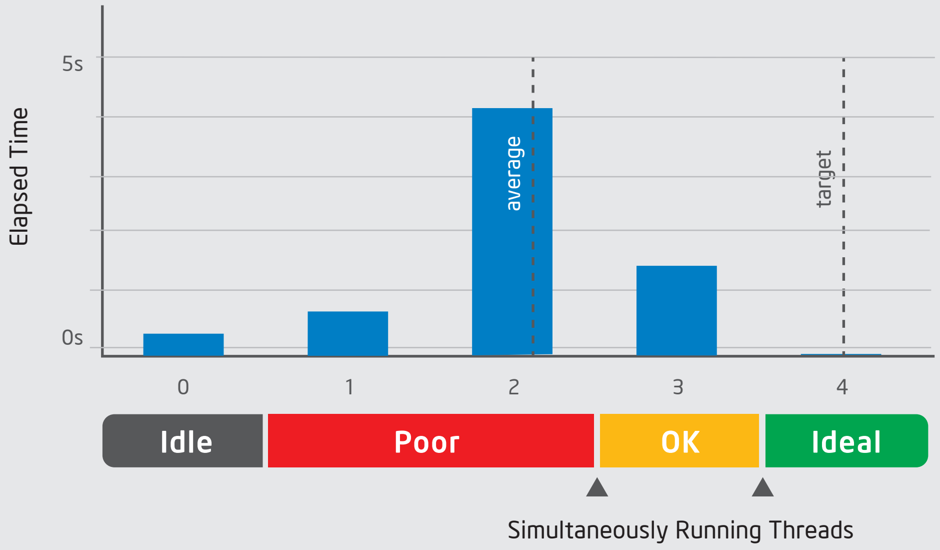
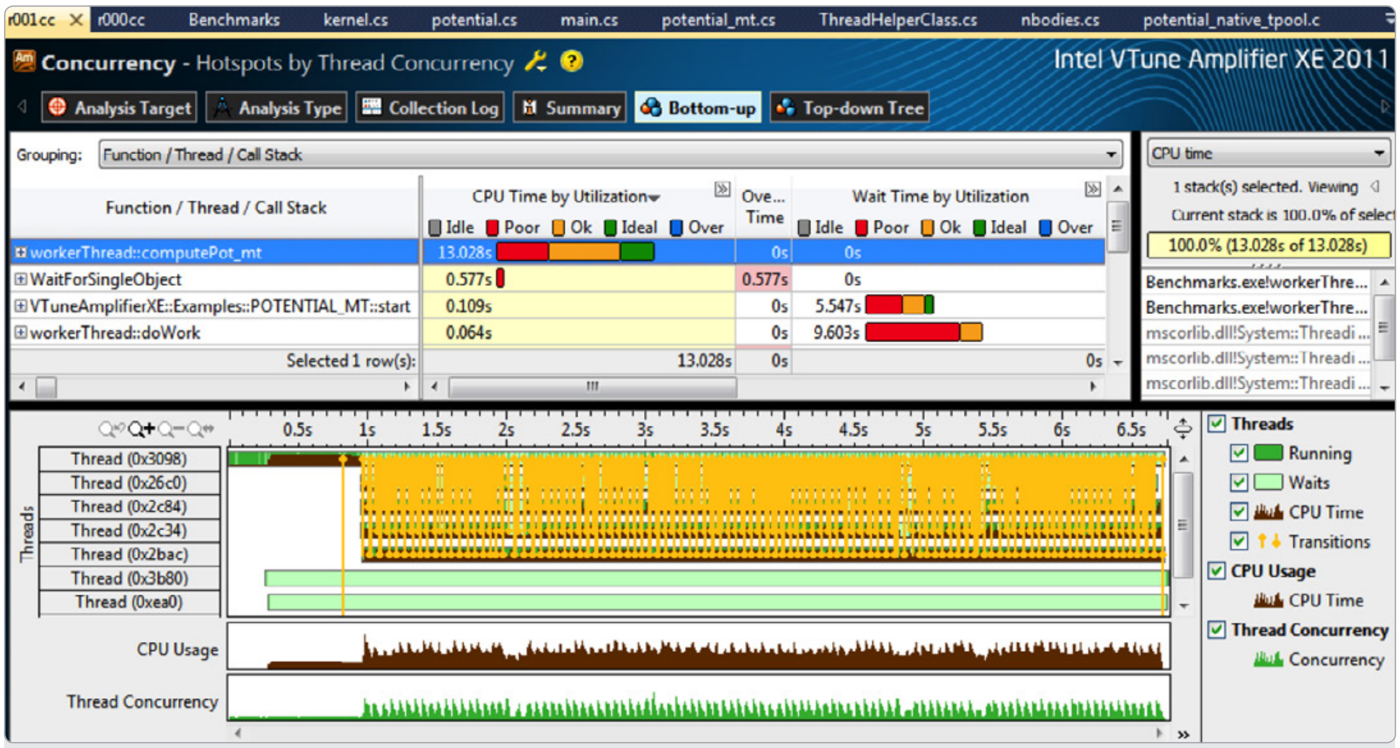


Figure 11: Thread concurrency and CPU usage histograms for the analysis



Grouping: Function / Thread / Call Stack

Function / Thread / Call Stack	CPU Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Overhead Time	Wait Time by Utilization (Idle, Poor, Ok, Ideal, Over)
workerThread::computePot_mt	13.028s	0s	0s
WaitForSingleObject	0.577s	0.577s	0s
VTuneAmplifierXE::Examples::POTENTIAL_MT::start	0.109s	0s	5.547s
workerThread::doWork	0.064s	0s	9.603s
Selected 1 row(s):	13.028s	0s	0s

CPU time

1 stack(s) selected. Viewing
Current stack is 100.0% of select
100.0% (13.028s of 13.028s)

Benchmarks.exe!workerThre...
Benchmarks.exe!workerThre...
mscorlib.dll!System::Threadi...
mscorlib.dll!System::Threadi...
mscorlib.dll!System::Threadi...

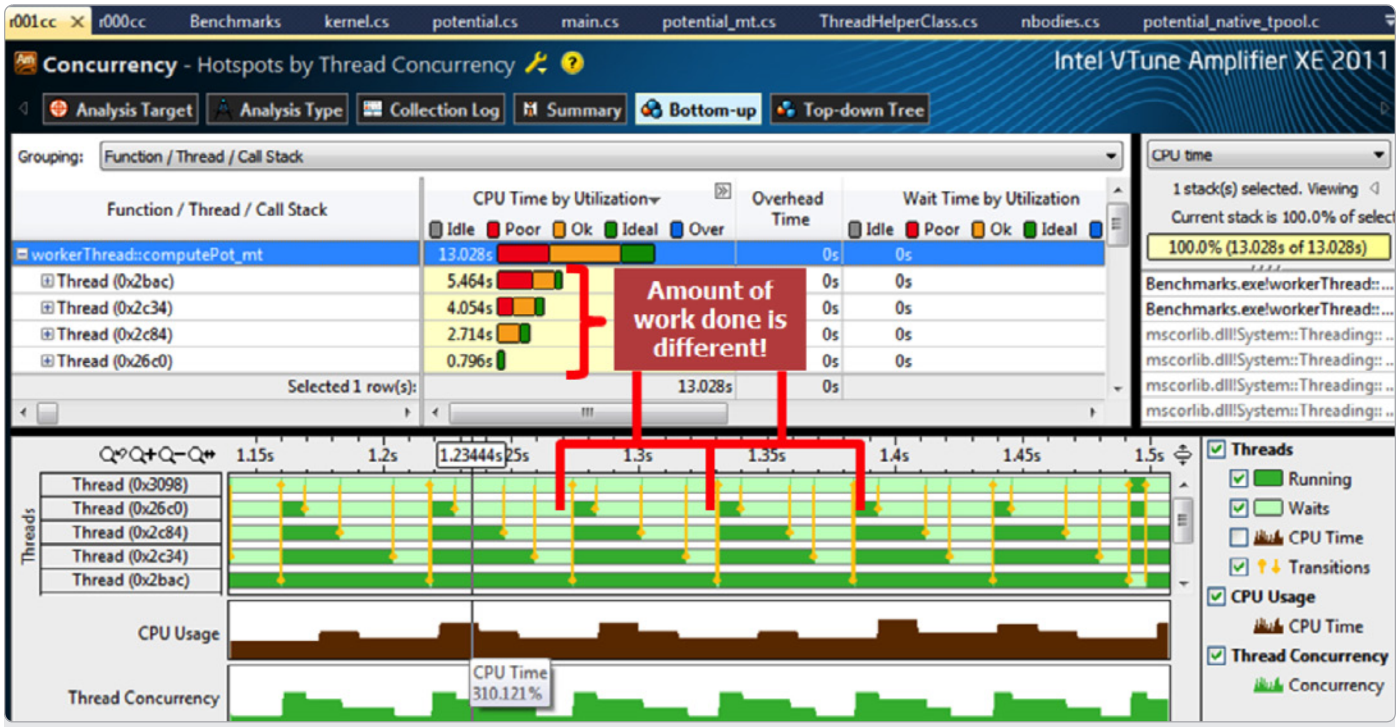
Figure 12: Hotspots by Thread Concurrency view

Figure 11 shows that our threaded application is not fully utilizing all available cores. The bottom-up view (**Figure 12**) gives a closer look at the results. Our `workerThread::computePot_mt` method is consuming most of the CPU time, and has significant amount of time identified as poor (red) and okay (orange) CPU utilization.

This indicates that this particular method is a hotspot (i.e., consuming most of the CPU Time) and threaded, but not fully utilizing the available cores. Therefore, it makes sense to zoom in to the timeline and look at each thread executing this particular method. **Figure 13** makes it clear that four threads, which are executing the `workerThread::computePot_mt` method, consume different amount of CPU time, causing a load imbalance and sub-optimal utilization of the cores. Such load imbalance issues will prevent applications from scaling as desired on more cores and needs to be fixed.

Even though each thread executes the outer `for` loop the same number of times, the inner loop is executed more by the thread operating on the last chunk, and least by the thread operating on the first chunk. Distributing the iteration's cyclical offset by the thread count will fix the load imbalance and the threads will utilize the available cores better. The concurrency analysis and the results not only enable us to identify load imbalance issues, but also help us speed up the application. The change below allows all the threads to stay busy and keep running (**Figure 16**).

"Load imbalance issues will prevent applications from scaling as desired on more cores and need to be fixed."



Function / Thread / Call Stack	CPU Time by Utilization	Overhead Time	Wait Time by Utilization
workerThread::computePot_mt	13.028s	0s	0s
Thread (0x2bac)	5.464s	0s	0s
Thread (0x2c34)	4.054s	0s	0s
Thread (0x2c84)	2.714s	0s	0s
Thread (0x26c0)	0.796s	0s	0s
Selected 1 row(s):	13.028s	0s	

1 stack(s) selected. Viewing
Current stack is 100.0% of select
100.0% (13.028s of 13.028s)
Benchmarks.exe!workerThread::...
Benchmarks.exe!workerThread::...
mscorlib.dll!System::Threading::...
mscorlib.dll!System::Threading::...
mscorlib.dll!System::Threading::...
mscorlib.dll!System::Threading::...

Threads: Thread (0x3098), Thread (0x26c0), Thread (0x2c84), Thread (0x2c34), Thread (0x2bac)

Figure 13: Showing the thread load imbalance on the timeline and CPU time of each thread executing

	Thread 1 (range)	Thread 2 (range)	Thread 3 (range)	Thread 4 (range)
Original version	0 -249	250 - 499	500 - 749	750 - 999
After changes	0,4,8,...,992, 996	1,5,9,...,993,997	2,6,10,...,994,998	3,7,11,...,995,999

Figure 14

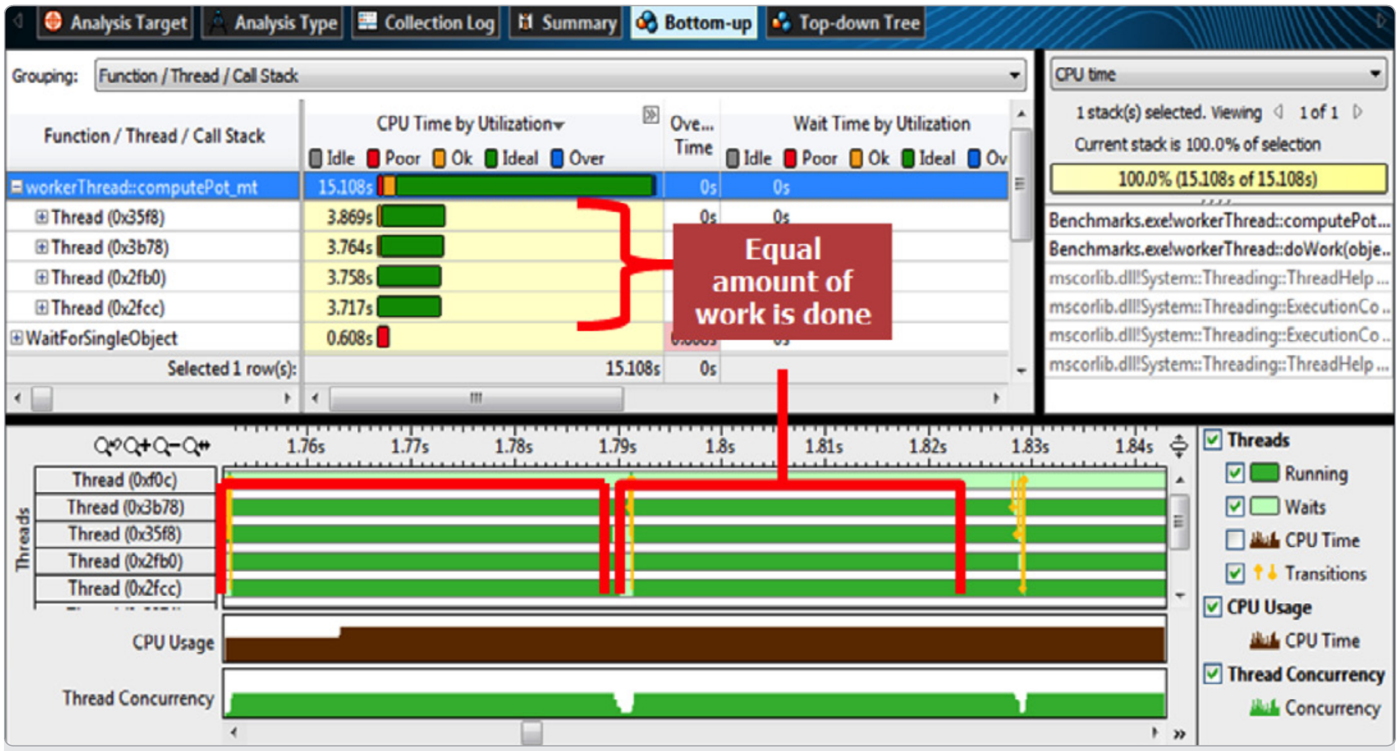
Original Version

```
chunkBegin = tid * (nParts / threadCount);
chunkEnd = (tid + 1) * (nParts / threadCount);
…

start = threadParameters.chunkBegin;
end = threadParameters.chunkEnd;

for (int i = start; i < end; i++)
{
 for (int j = 0; j < i - 1; j++)
 {
 distx = Math.Pow((r[0][j] - r[0][i]), 2);
 disty = Math.Pow((r[1][j] - r[1][i]), 2);
 distz = Math.Pow((r[2][j] - r[2][i]), 2);
 dist = Math.Sqrt(distx + disty + distz);
 potential += 1.0 / dist;
 }
}
```

Load-balanced Version

```
for (int i = tid; i < constants.POT_NPARTS; i +=
threadParameters.threadCount)
{
 for (int j = 0; j < i - 1; j++)
 {
 distx = Math.Pow((r[0][j] - r[0][i]), 2);
 disty = Math.Pow((r[1][j] - r[1][i]), 2);
 distz = Math.Pow((r[2][j] - r[2][i]), 2);
 dist = Math.Sqrt(distx + disty + distz);
 potential += 1.0 / dist;
 }
}
```

Figure 15



Function / Thread / Call Stack	CPU Time by Utilization	Ove... Time	Wait Time by Utilization
workerThread::computePot_mt	15.108s	0s	0s
Thread (0x35f8)	3.869s	0s	0s
Thread (0x3b78)	3.764s		
Thread (0x2fb0)	3.758s		
Thread (0x2fcc)	3.717s		
WaitForSingleObject	0.608s		
Selected 1 row(s):	15.108s	0s	

1 stack(s) selected. Viewing 1 of 1

Current stack is 100.0% of selection

100.0% (15.108s of 15.108s)

```
Benchmarks.exe!workerThread::computePot...
Benchmarks.exe!workerThread::doWork(obje...
mscorlib.dll!System::Threading::ThreadHelp...
mscorlib.dll!System::Threading::ExecutionCo...
mscorlib.dll!System::Threading::ExecutionCo...
mscorlib.dll!System::Threading::ThreadHelp...
```

Figure 16: workerThread::computePot_mt method is executed by four threads, which do an equal amount of work

Are we done? Not yet. Let's give a try to the architectural analysis of VTune™ Amplifier XE to check if the tool can identify more opportunities for performance improvements. To demonstrate the architectural analysis feature in VTune Amplifier XE, let's use General Exploration analysis pre-configured for 2nd Generation Core™ architecture.

The 2nd generation Core microarchitecture is capable of reaching Cycles Per Instructions as low as 0.25 in ideal situations. The greater value of CPI for a given workload indicates that there are more opportunities for code tuning to improve performance. **Figure 18** shows the results of the General Exploration analysis. In this case, the invocation of the `Math.Pow()` function consumes significant amount of clockticks. Replacing `Math.Pow()` with a simple multiplication gives us much better performance and reduces the CPI ratio to 1.5.

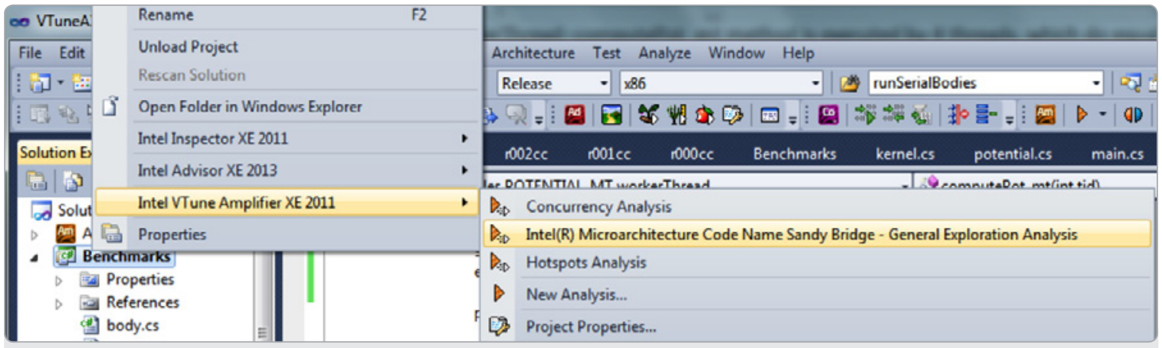
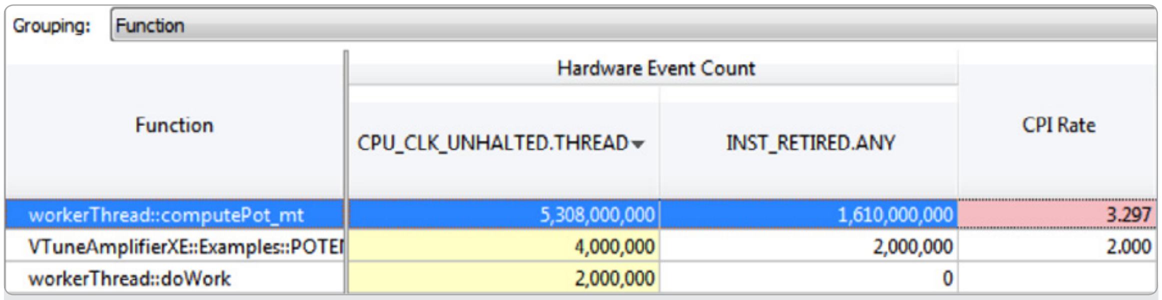


Figure 17: Running Intel® VTune™ Amplifier XE General



Grouping: Function

Function	Hardware Event Count: CPU_CLK_UNHALTED.THREAD	Hardware Event Count: INST_RETIRED.ANY	CPI Rate
workerThread::computePot_mt	5,308,000,000	1,610,000,000	3.297
VTuneAmplifierXE::Examples::POTEI	4,000,000	2,000,000	2.000
workerThread::doWork	2,000,000	0	

Figure 18: General Exploration results highlighting problematic functions

```
for (int i = tid; i < constants.POT_NPARTS; i +=
threadParameters.threadCount)
{
 for (int j = 0; j < i - 1; j++)
 {
 distx = Math.Pow((r[0][j] - r[0][i]), 2);
 disty = Math.Pow((r[1][j] - r[1][i]), 2);
 distz = Math.Pow((r[2][j] - r[2][i]), 2);
 dist = Math.Sqrt(distx + disty + distz);
 potential += 1.0 / dist;
 }
}
```

```
for (int i = tid; i < constants.POT_NPARTS; i +=
threadParameters.threadCount)
{
 for (int j = 0; j < i - 1; j++)
 {
 distx = (r[0][j] - r[0][i]) * (r[0][j] - r[0][i]);
 disty = (r[1][j] - r[1][i]) * (r[1][j] - r[1][i]);
 distz = (r[2][j] - r[2][i]) * (r[2][j] - r[2][i]);
 dist = Math.Sqrt(distx + disty + distz);
 potential += 1.0 / dist;
 }
}
```

Figure 19

Original Version (sec)	Load Imbalance Fixed (sec)	Math.Pow() replaced with * (sec)
5.4	4.65	0.39

Figure 20

"Intel® Parallel Studio XE enables .NET developers to identify critical performance problems such as the most time-consuming functions or lines of code, scalability issues, and time spent waiting on synchronization constructs and IO activity. While doing this, Parallel Studio XE reveals potential micro-architectural bottlenecks caused by issues such as branch mispredicts, cache misses, and memory bandwidth problems."

Performance Tuning Metric

Cycles Per Instruction retired, or CPI, is a fundamental performance metric indicating approximately how much time each executed instruction took, in units of cycles. Modern superscalar processors issue up to four instructions per cycle, suggesting a theoretical best CPI of .25. Various effects (long-latency memory, floating-point, or SIMD operations; non-retired instructions due to branch mispredictions; instruction starvation in the front-end) tend to pull the observed CPI up. Nonetheless, CPI is an excellent metric for judging an overall potential for application performance tuning.

For advanced and deeper microarchitectural analysis, the tool is equipped with predefined analysis types, which use Performance Monitoring Unit (PMU) to sample processor events to identify microarchitectural 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™ microarchitecture, Intel® Core™ microarchitecture (aka Nehalem and Westmere) and Intel® 2nd Generation Core™ microarchitecture (aka Sandy Bridge). When these advanced predefined analysis types are used, the tool gives hints and suggestions by highlighting the problematic functions.

Supported .NET Versions

VTune Amplifier XE and Inspector XE support the basic synchronization mechanisms available in .NET versions 2.0 to 3.5. The tools do not support the new synchronization APIs introduced in .NET 4.0 and the new Task Parallel Library.

Summary

Inspector XE and VTune Amplifier XE provide valuable technologies to .NET developers. These tools combine error checking and performance profiling tools under Intel Parallel Studio XE. They help boost application performance and increase the code quality and reliability needed by high-performance computing and enterprise applications. At the same time, the suite eases the procurement of all the necessary tools for high performance, and simplifies the transition from multicore to manycore processors for the future. View the source code › ☐

RESOURCES AND SITES OF INTEREST

Go Parallel

The mission of Go Parallel is to assist developers in their efforts toward "Translating Multicore Power into Application Performance." Robust and full of helpful information, the site is a valuable clearinghouse of multicore-related blogs, news, videos, feature stories, and other useful resources.

"What If" Experimental Software

What if you could experiment with Intel's advanced research and technology implementations that are still under development? And then what if your feedback helped influence a future product? It's possible here. Test drive emerging tools, collaborate with peers, and share your thoughts via the What If blogs and support forums.

Intel® Software Network

Check out a range of resources on a wide variety of software topics for a multitude of developer communities ranging from manageability to parallel programming to virtualization and visual computing. This content-rich collection includes Intel® Software Network TV, popular blogs, videos, tools, and downloads.

Step Inside the Latest Software

See these products in use, with video overviews that provide an inside look into the latest Intel® software. You can see software features firsthand, such as memory check, thread check, hotspot analysis, locks and waits analysis, and more.

Intel® Inspector XE

Intel® VTune™ Amplifier XE

Intel® Software Evaluation Center

The Intel® Software Evaluation Center makes 30-day evaluation versions of Intel® Software Development Products available for free download. For high-performance computing products, you can get free support during the evaluation period by creating an Intel® Premier Support account after requesting the evaluation license, or via Intel® Software Network Forums. For evaluating Intel® Parallel Studio, you can access free support through Intel® Software Network Forums ONLY.

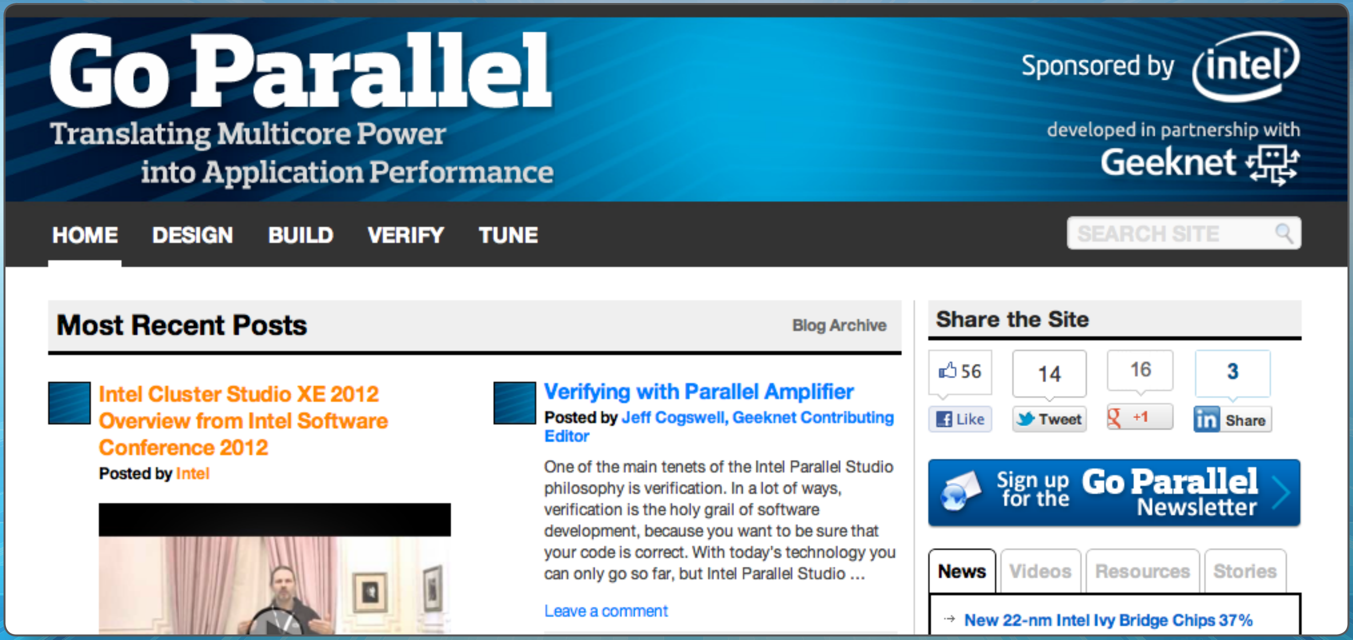
Sign up for future issues | Share with a friend

Go Parallel Has *New Attitude*

Announcing a refresh to your online portal for all things parallel.

Get the latest industry insights, news, blogs, videos, how-tos, and technical documentation on parallelism. Contributors include game-changing developers Jeff Cogswell and John Jainschigg, and leading subject matter experts like Intel's James Reinders, Heinz Bast, and Levent Akyil, as well as Geeknet's Stephen Wellman and Chris Yeich.

Translating multicore performance into application power.



Go Parallel

Translating Multicore Power into Application Performance

Sponsored by intel

developed in partnership with Geeknet

HOME DESIGN BUILD VERIFY TUNE SEARCH SITE

Most Recent Posts Blog Archive

Intel Cluster Studio XE 2012 Overview from Intel Software Conference 2012
Posted by Intel

Verifying with Parallel Amplifier
Posted by Jeff Cogswell, Geeknet Contributing Editor

One of the main tenets of the Intel Parallel Studio philosophy is verification. In a lot of ways, verification is the holy grail of software development, because you want to be sure that your code is correct. With today's technology you can only go so far, but Intel Parallel Studio ...

Leave a comment

Share the Site

56 Like | 14 Tweet | 16 +1 | 3 Share

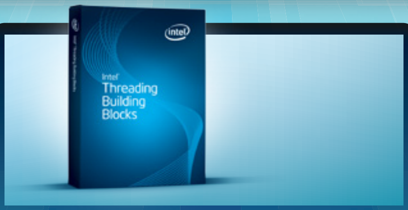
Sign up for the Go Parallel Newsletter

News Videos Resources Stories

New 22-nm Intel Ivy Bridge Chips 37%

Video

Building Parallel Applications with Cilk Plus and Threading Building Blocks, Chris Yeich, Intel



Tutorials

Using Intel® TBB in network applications: Network Router emulator

Blogs

Wellington and Austin: Programming Lots of Cores
James Reinders, Intel

Intel® Cilk™ Plus: A Simple Path to Parallelism
Resolve Resource Leaks in Your Applications with Intel® Parallel Studio XE

Evaluation Guides

Intel® Cilk™ Plus

Tech Documents

The Parallel Universe Magazine

Optimizing VLife* Molecular Design Suite Using Intel® Parallel Studio XE
WHITE PAPER
Introduction
Background

White Papers

Optimizing VLife* Molecular Design Suite Using Intel® Parallel Studio XE

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

intel® Software

intel® Software

TAKE PERFORMANCE TO THE EXTREME.

XE

INTEL® PARALLEL STUDIO XE

From one-person startups to enterprises with thousands of developers working on a single application, Intel® Parallel Studio XE 2011 extends industry-leading development tools for unprecedented application performance and reliability.

Advanced compilers and libraries
Intel® Composer XE

Advanced memory, threading, and security analyzer
Intel® Inspector XE

Advanced performance profiler
Intel® VTune™ Amplifier XE

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

Get a free 30-day trial of Intel Parallel Studio XE today at **www.intel.com/software/products**.