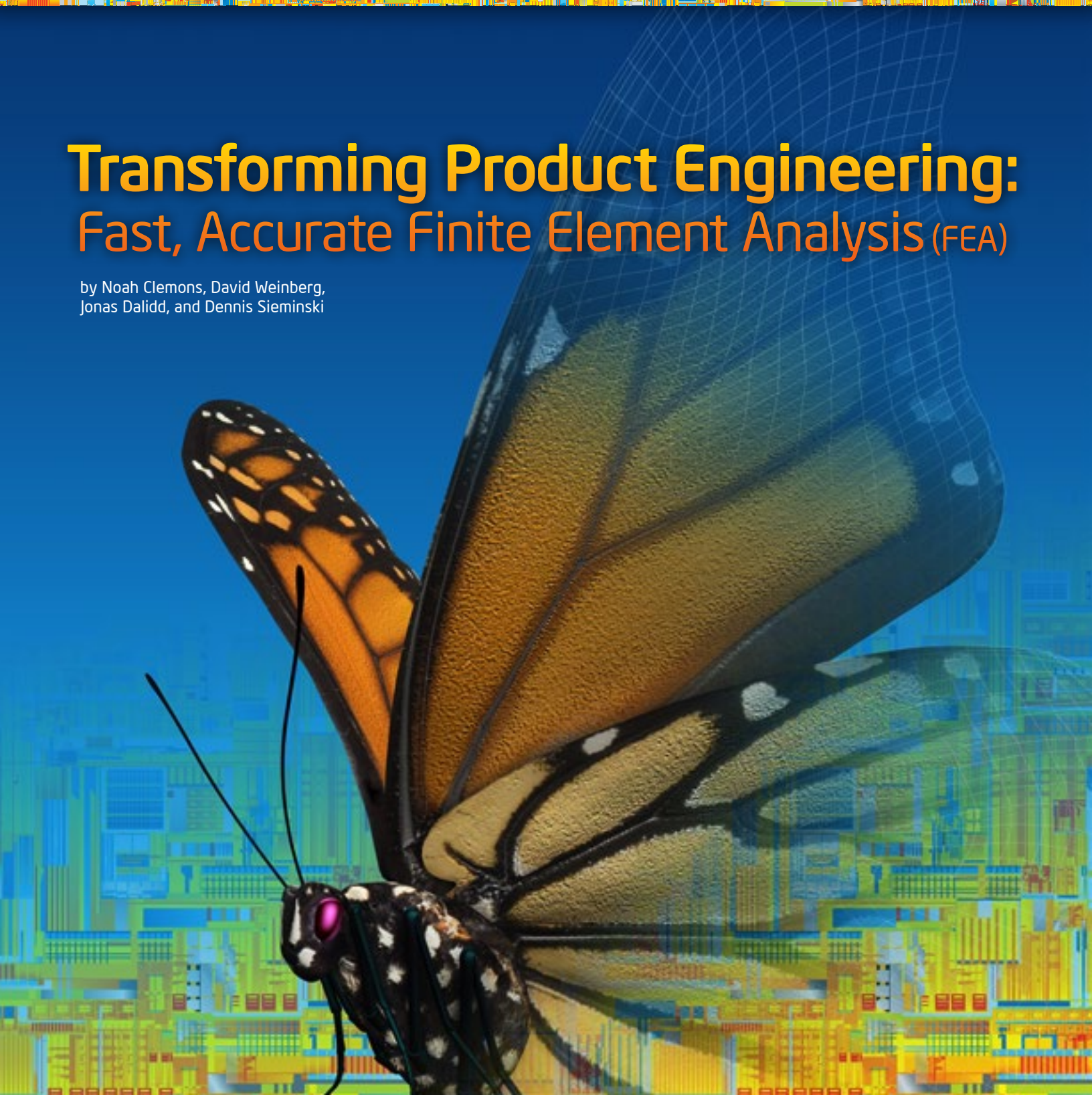
The Parallel Universe

intel Software

Issue 13
FEBRUARY 2013

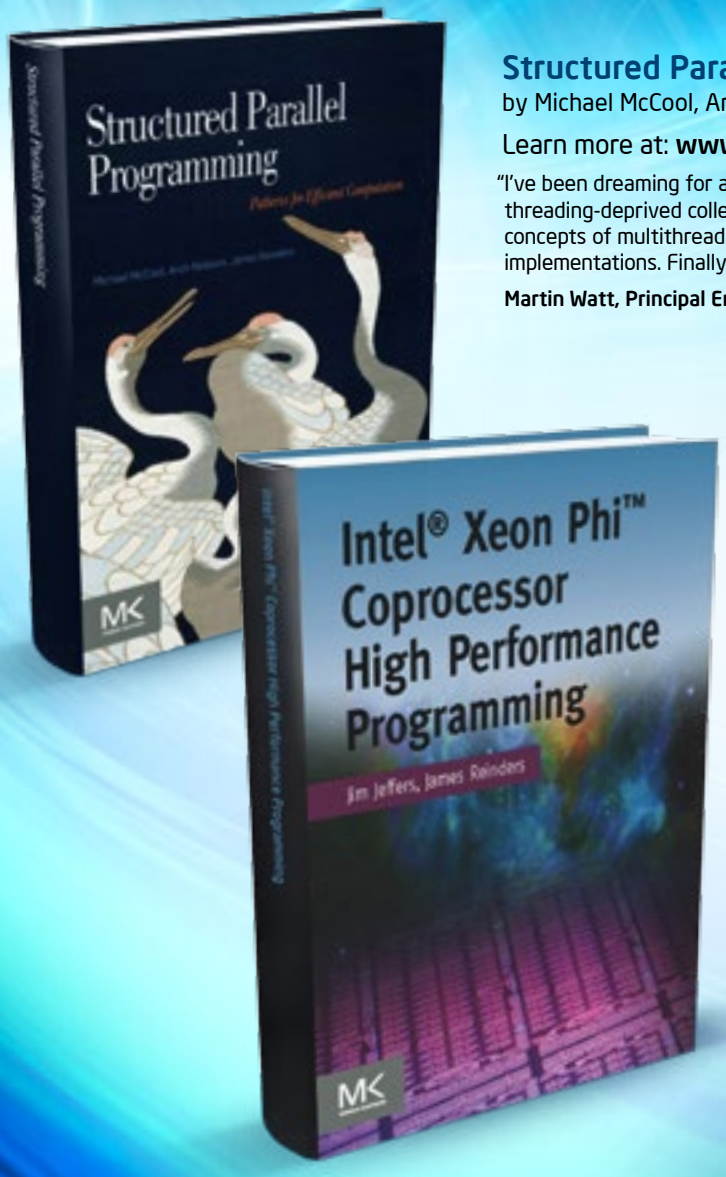
Transforming Product Engineering:
Fast, Accurate Finite Element Analysis (FEA)

by Noah Clemons, David Weinberg,
Jonas Dalidd, and Dennis Sieminski



NEW RESOURCES MAY CHANGE THE WAY YOU BUILD APPLICATIONS.

Explore the latest on parallel programming from threading to HPC.



Structured Parallel Programming

by Michael McCool, Arch Robison, and James Reinders

Learn more at: **www.parallelbook.com**

"I've been dreaming for a while of a modern, accessible book that I could recommend to my threading-deprived colleagues and assorted enquirers to get them up to speed with the core concepts of multithreading, as well as something that covers all the major current interesting implementations. Finally I have that book."

Martin Watt, Principal Engineer, DreamWorks Animation

Intel® Xeon Phi™ Coprocessor High Performance Programming

by Jim Jeffers and James Reinders

Learn more at: **www.lotsofcores.com**

"This book belongs on the bookshelf of every HPC professional. It takes us back to the universal fundamentals of high performance computing, including how to think and reason about the performance of algorithms mapped to modern architectures, and it puts into your hands powerful tools that will be useful for years to come."

Robert J. Harrison, Institute for Advanced Computational Science, Stony Brook University

MEET THE AUTHORS

James Reinders, director and chief evangelist for Intel® Software, or Jim Jeffers, software product application engineer for Intel® Many Integrated Core (Intel® MIC) will discuss their new book and deliver the keynote address at the upcoming Intel® Software Conference 2013 in four U.S. cities. **See full agenda and register** ⊙

intel® Software

©2013, Intel Corporation. All rights reserved. Intel, the Intel logo, and Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.

CONTENTS

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.

© 2013, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Core, Intel Inside, Cilk Plus, Pentium, VTune, VPro, Xeon and Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.

Forging Ah

LETTER FROM THE EDITOR

James Reinders, Director of Parallel Programming Evangelism at Intel Corporation. James is a coauthor of two new books from Morgan Kaufmann, *Intel® Xeon Phi™ Coprocessor High Performance Programming* (2013), and *Structured Parallel Programming* (2012). His other books include *Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism* (O'Reilly Media, 2007, available in English, Japanese, Chinese, and Korean), and *VTune™ Performance Analyzer Essentials* (Intel Press, 2005).

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

ead with Software

Our first issue, this year, explores some of the techniques, tools, and coding models enabling an explosion of innovation driven by software. The reality that software is changing the world is very evident in our feature article.

This feature article, *Fast, Accurate Finite Element Analysis (FEA),* explores finite element analysis (FEA) and its role in extremely complex 3D simulations that are revolutionizing product engineering—from NASA and Formula 1 race cars to medical implants and environmental analysis.

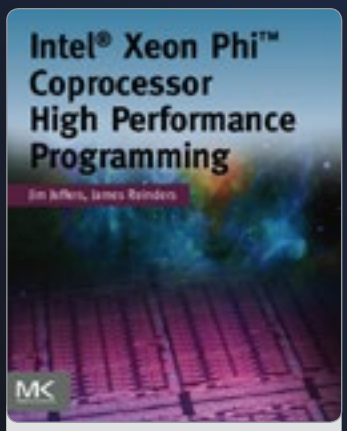
Parallel Power: Optimize Software for Intel® Xeon Phi™ Coprocessors provides a straightforward code optimization methodology to maximize performance on Intel® Xeon® processors *and* take advantage of the newest Intel® Xeon Phi™ coprocessors. A related article, *Unleash Intel® Xeon Phi™ Coprocessor Performance,* offers tips on optimization using the Intel® Math Kernel Library (Intel® MKL)—a key ingredient in unleashing the performance of Intel® architectures in real-world applications. I found both of these articles inspirational as I was working with my coauthor to put the finishing touches on our book about programming for the exciting new Intel Xeon Phi coprocessor (learn more about the book at: **http://lotsofcores.com**). I recommend these articles and our book for anyone interested in programming the Intel Xeon Phi coprocessor.

Tune OpenMP Applications* provides step-by-step guidance for increasing insight and analyzing and tuning performance for parallel HPC applications. *Expand Your Debugging Options* looks at debugging across Intel Xeon Phi coprocessor use cases, considering both offload and native models.

Parallel Execution Using HTML5 covers the API options for parallel HTML5 applications, allowing developers to benefit from parallelization performance increases.

As we see the theoretical possibility of what software can make possible become a reality, it's a great time to explore the techniques and tools that can bring our own applications to the next level. I look forward to seeing what you make possible in the year ahead.

James Reinders
February 2013



New book examines programming for the Intel® Xeon Phi™ coprocessor and Intel® Many Integrated Core (Intel® MIC) architecture

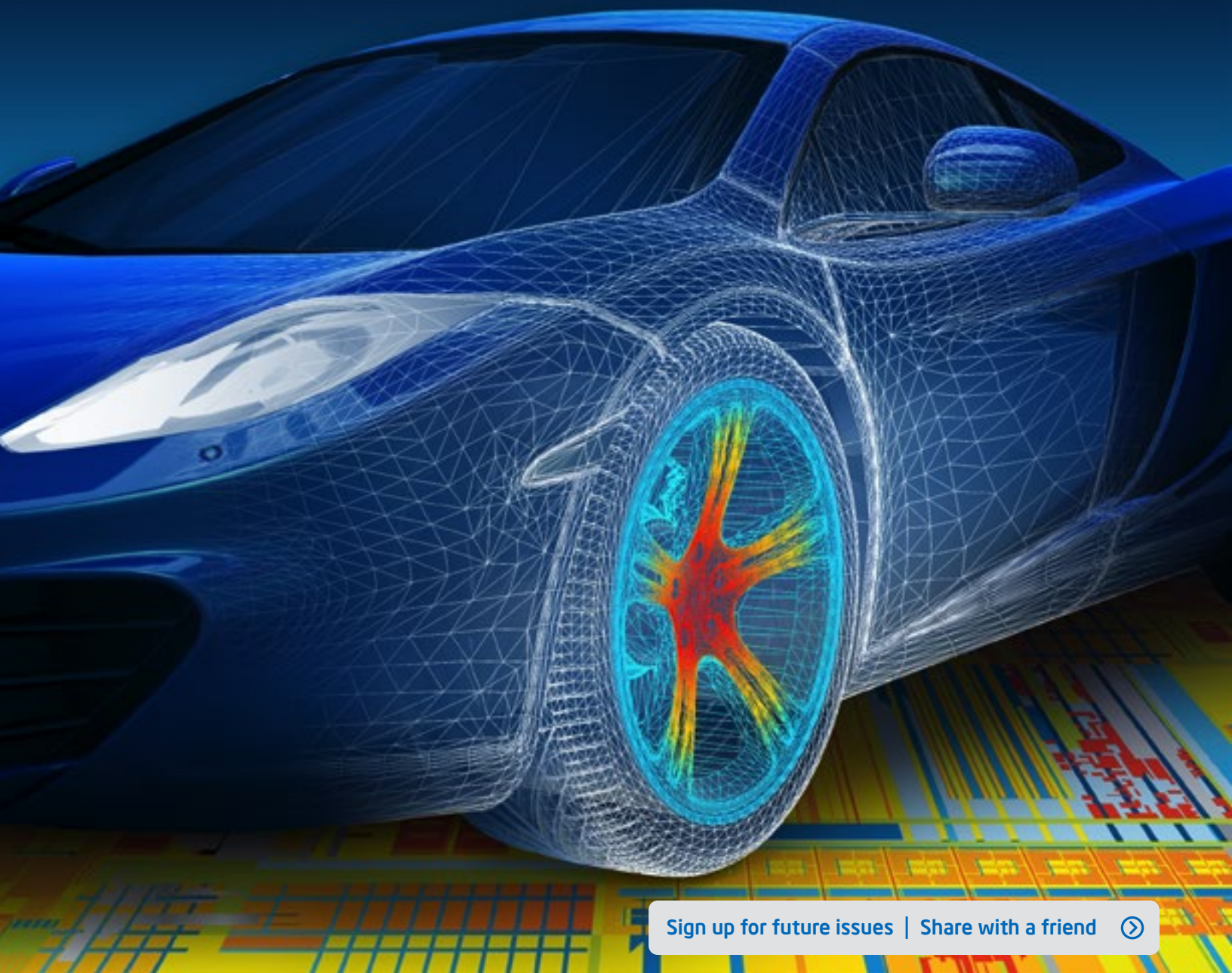
TRANSFORMING PRODUCT ENGINEERING:

Fast, Accurate Finite Element

by Noah Clemons, *Technical Consulting Engineer, Embedded Compute, Debuggers, and Performance Libraries Group, Intel,* David Weinberg, *NEi,* Jonas Dalidd, *NEi,* and Dennis Sieminski, *NEi*

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

Analysis (FEA)



Sign up for future issues | Share with a friend

Engineered products of all kinds are improved through finite element analysis (FEA). FEA allows engineers to examine structures with incredible thoroughness and flexibility. This paper introduces FEA, and shows how the performance benefits of the Intel® Math Kernel Library (Intel® MKL) improve analysis accuracy, while supporting analyst productivity.

Today's media features a continual stream of news on impressive engineering projects. The stories cover an incredibly wide spectrum of industries—from progress on spacecraft capable of sending tourists on real-life space travel experiences to medical implants that restore vital and sensitive faculties like eyesight and hearing. Even the sports world features new equipment designs that change performance expectations, including Tour de France bicycles, America's Cup yachts, or Formula 1 race cars. What is driving such widespread sophisticated technical capability? These examples from NEi Software's customer case studies (www.NEiSoftware.com) illustrate the role a numerical method known as finite element analysis (FEA) has come to play in elevating contemporary engineering practice. FEA software is a tool that gives engineers the power to virtually test 3D models of their product designs. Engineers can simulate the application of loads, forces, impacts, vibrations, heat, and temperature conditions. They see how the virtual product will respond not only with numerical data and graphs, but also through the clever use of 3D graphics and animation. Huge data files generated on technical entities such as stress, strain, vibration modes, and temperature gradients can be turned into animations and colorful visual presentations that aid in developing a more comprehensive technical understanding of the performance of complex physical systems. The insight gained through working with these digital prototypes provides numerous benefits. A big advantage comes from the ability to have problem areas highlighted, so they can be addressed while still in the digital design phase where change is much easier, design alternatives can be tried, and various operating conditions can be thoroughly explored until results are satisfactory. All the changes can be made before the first part is made or prototype has been built. The result is a productive and insightful design process that brings a cascade of benefits and savings across subsequent steps in the product development cycle—fewer prototypes, less physical testing, and a reduction in the number of design iterations before release to manufacturing. The upshot is lower costs, faster time to market, and better-quality products. It is clear that with these benefits the introduction of FEA has had a profound and widespread effect on the practice of product engineering.

A quick look at how FEA reached its present stage is instructive; its path provides context for some of the current issues facing the next stage in the development of the technology. The origin of commercial FEA software is typically traced to the public release in the early 1970s of code developed under the auspices of the National Aeronautics and Space Administration (NASA). The program was called NASTRAN, an acronym for NASA Structural Analysis. The cost of computer resources and the specialized personnel needed to use the software initially confined NASTRAN to expensive, high-profile projects—typically aerospace programs funded by the federal government. Over the next two decades, improvements in computing hardware, FEA software capabilities, and costs allowed proliferation of the software into the top tier of aerospace, automotive, maritime, and nuclear applications. The next phase for FEA software usage occurred with the dramatic transition from mainframe computers. As part of that pioneering effort, NEi Software developed NEi Nastran* to run on PCs (www.NEiNastran.com). With FEA capabilities within reach of average-sized engineering departments, usage expanded in traditional aerospace, automotive, and maritime markets. More important, an even wider range of industry segments, such as civil, medical, consumer, and recreation were able to embrace the technology.

Improving computational speed without sacrificing accuracy continues to be of paramount importance. For the technology to be viable and enjoy widespread usage, it cannot take weeks, days, or even too many hours of computer time to render the result of each FEA test case. There needs to be quick turnaround and availability of results. What's different today is that engineers are building on past successes and tackling more complex simulations. The models are larger and more detailed. The physical phenomena they seek to replicate are nonlinear, multidiscipline, and interrelated. Plus, new materials such as composites and shape memory alloys are more complex in their behavioral properties. Using this background, we can look at the mathematical and programming structures found in FEA software to see where and how improvements in speed and accuracy may be accomplished.

Solving a problem in mechanics or physics means predicting the mechanical or physical system's behavior due to the action of given loads. The analytical solution of such a problem is only possible on geometrically simple domains (e.g., rectangles, circles). In order to tackle more complex domains, numerical discretization methods such as FEA are required. As the name implies, the premise of FEA is to subdivide the complex domain into a finite number of subdomains, or *finite elements*, and solve the physical problem on each of them. A representation of a continuous domain with a set of finite elements is called a *finite element model*. All finite elements in the model form a *finite element mesh*. Each element in the finite element model has a fixed number of *nodes* that define the element's boundaries to which loads and boundary conditions can be applied. The finer the mesh, the closer approximation of the geometry of the structure, the load application,

Figure 1: Cessna 510 Citation Mustang* designed and analyzed using NEi Nastran*. Colors superimposed on the empennage indicate levels of stress in the structure. The red end of the spectrum is used for higher values versus blue for lower values.

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

and the stress and strain gradients. However, there is a tradeoff: the finer the mesh, the more computational power needed to solve the complex problem. The trend in FEA is for larger, more detailed, and complicated models that require greater computing power and for advanced tools, such as Intel MKL, to be built into FEA applications. FEA is comprised of three phases:

1. **Preprocessing:** The analyst develops a finite element mesh based on the geometry of the structure, and applies material properties, boundary conditions, and loads to it.
2. **Solution:** The FEA solver assembles the stiffness matrix and applied load vector and solves for displacements, strains, and stresses. The solution for displacements requires the solving of a large number of simultaneous equations and can be the most numerically intensive operation in any FEA application.
3. **Post-processing:** The analyst obtains results usually in the form of deformed shapes, contour plots, and other graphic visualizations which help to check the validity of the solution.

The pre- and post-processing phases typically rely on an FEA modeler such as Siemens Femap* or NEi Nastran in-CAD*. Femap is a stand-alone FEA modeler and post-processor that works externally with computer-aided design (CAD) applications. CAD is the tool typically used to define the geometry of the structure. NEi Nastran in-CAD is an internal CAD application that works directly with CAD programs to build the FEA model, run the solution, and post-process within the more familiar CAD application.

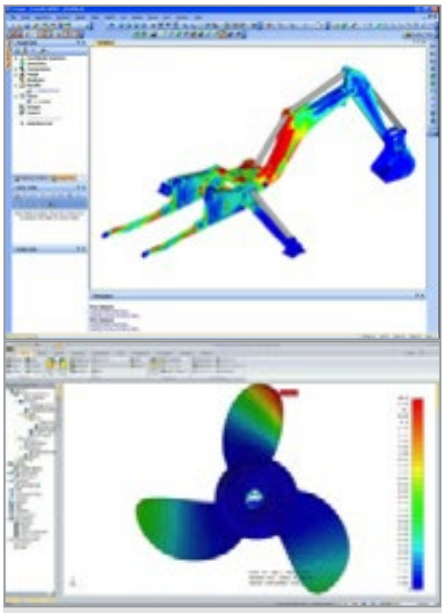


Figure 2: The Femap pre- and post-processor (above top) and NEi Nastran* in-CAD* (above) user interfaces. Note: colors are used to represent levels of stress in these example analyses. The red end of the spectrum is used for higher values versus blue for lower values.

The solution phase relies on an FEA solver such as NEi Nastran. It is the most numerically intensive of all three phases and will require the most memory and CPU processing time. The analyst is typically looking for the fastest and most accurate solution. These two criteria are often diametrically opposed to each other—as faster means a coarser FEA mesh, which is less accurate. The challenge is to have both. Within an FEA solver are specialized subcomponents optimized to provide the best performance possible. For NEi Nastran and many other FEA solvers, the critical subcomponents are the linear equation solver and eigenvalue/eigenvector solver. For best performance, no individual subcomponent should dominate the solution time. Therefore, it is critical that subcomponents take advantage of memory and CPU architecture. NEi Nastran uses Intel MKL, specifically its PARDISO solver and DGEMM matrix multiplication routine, to reduce analysis time and avoid solution bottlenecks. The PARDISO solver can reduce overall analysis time by a factor of 100, or even 1,000 in some cases, because of the parallel processing scalability and its tuning to specific Intel® CPU architecture. The importance of the PARDISO solver is magnified in the nonlinear case, where it can take thousands of decompositions and backsolves to complete. The PARDISO solver is the default solver for nonlinear analysis in NEi Nastran, and is typically the fastest solver.

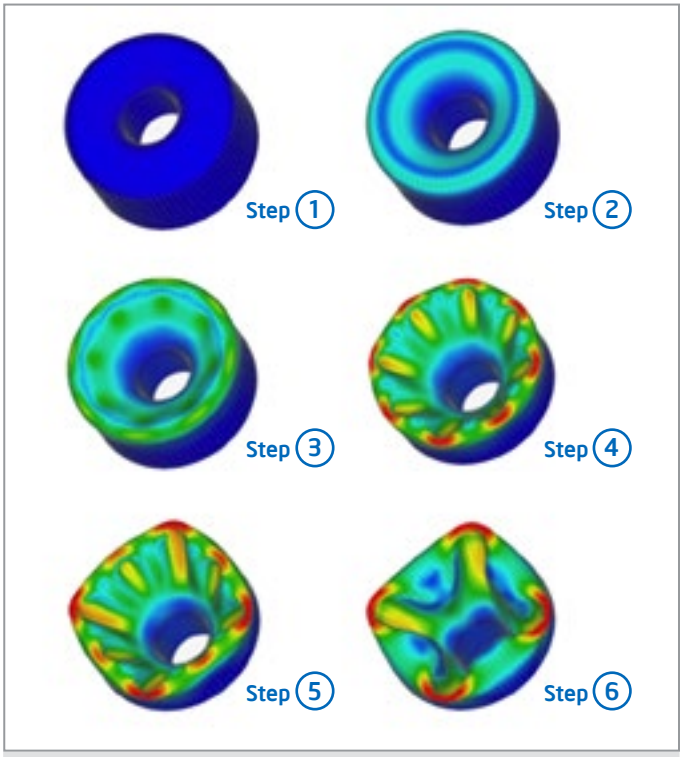


Figure 3: A nonlinear transient analysis of a rubber membrane. The analysis involves large displacements, rotations, and post-buckling behavior. The numbered sequence shows changes to part shape caused by the loading. Stress levels are represented by colors. To view this as an animation, and for other examples, go to www.nenastran.com/fea/animations.

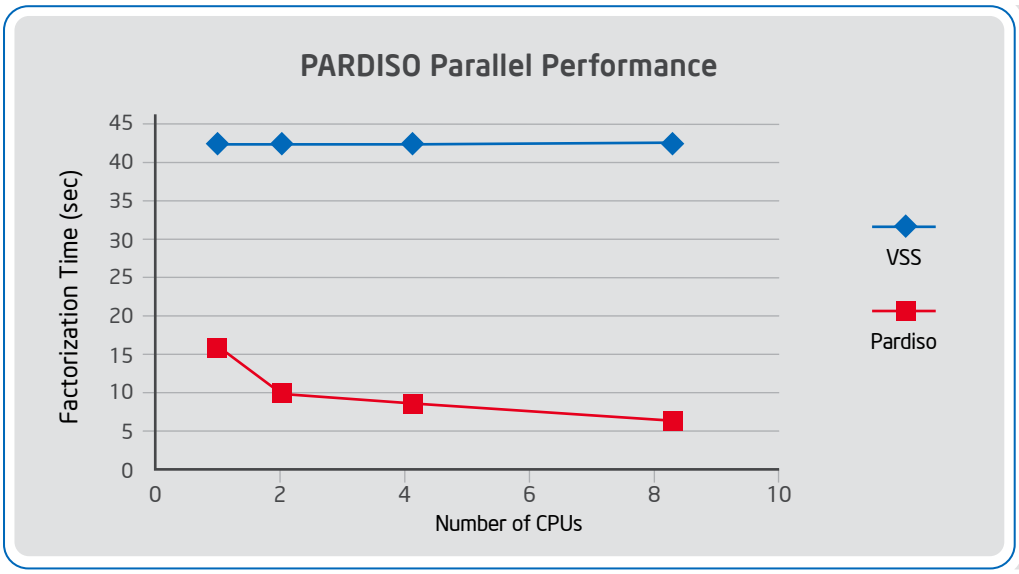


Figure 4: Factorization time for the rubber membrane analysis showing parallel scalability of the PARDISO solver. Typical nonlinear transient analyses perform hundreds of factorizations per simulation. (Hardware: 2 x Intel® Xeon® processor E5-2670 2.6GHz, 60.5GB RAM, SAS; Software: Windows Server* 2008 R2 64-bit, Intel® Math Kernel Library 11.0.1.)

```
C
C WRITE OUT STATUS
C
        STR806 = 'REORDERING'
        CALL WRIT1STA( STR806)
C
        PHASE = 11
C
C REORDERING AND SYMBOLIC FACTORIZATION
C
        CALL PARDISO( PPOINTER, MAXFACTMATRIX, MATRIX, PMTYPE, PHASE,
     1                NFDOF, A, LN, TLA, INTNUM, NRHS, INTPARAMETER,
     2                MSGLVL, REALNUM, REALNUM, ERRORSTATUS)
        IF ( ERRORSTATUS .NE. 0 ) THEN
         IF ( (ERRORSTATUS .EQ. PSS_OOC_INSUFFICIENT_MEMORY) .OR.
     1         (ERRORSTATUS .EQ. PSS_INSUFFICIENT_MEMORY) ) THEN
          IF ( L .EQ. 1 ) THEN
           AMULTIBLOCK = .TRUE.
C
C FREE SPARSE SOLVER MEMORY (IF ALLOCATED).
C
            CALL FREEPSS
C
C FORCE OUT OF CORE MODE.
C
            INTPARAMETER(60) = 2
C
C RESET PSS CONFIGURATION FILE.
C
            CALL SETPSSCF( 'RESET')
C
           CYCLE
          ELSE
           GOTO 307
          ENDIF
         ELSE
          GOTO 207
         ENDIF
        ENDIF
C
        SPARSEMEMALLOC = .TRUE.
C
C WRITE OUT STATUS.
C
        WRITE (STR806, 1000) 0
        CALL WRIT1STA( STR806)
C
        PHASE = 22
C
C NUMERICAL FACTORIZATION.
C
        CALL PARDISO( PPOINTER, MAXFACTMATRIX, MATRIX, PMTYPE, PHASE,
     1                NFDOF, A, LN, TLA, INTNUM, NRHS, INTPARAMETER
     2                MSGLVL, REALNUM, REALNUM, ERRORSTATUS)
C
```

Figure 5: Call to PARDISO solver for factorization. Within Intel® MKL there are several ways to call the PARDISO solver. Here we are first calling it for reordering and symbolic factorization to determine and minimize memory requirements (PHASE=11). Next we call it for the action factorization that will use the most memory and CPU demand (PHASE=22). Provisions are made within NEi Nastran* to revert to an out-of-core mode if needed, which will use less memory and more I/O but will result in slower performance.

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

PARDISO implementation is fairly straightforward and uses the sparse matrix format (storing only non-zero terms) essential for solving today's large FEA problems. Most FEA solutions deal with large sparse matrixes. A typical call to the PARDISO solver is shown in **Figure 5** where provisions are made for handling matrixes too large to fit into available physical memory.

FEA analysis also involves dense matrixes and their multiplication. The Intel MKL DGEMM routine provides a very fast, scalable routine for the multiplication of large, dense matrixes. Like PARDISO, it is optimized for Intel CPU architecture. Eigenvalue analysis is often performed to determine natural frequencies and mode shapes of structures, as well as for dynamic response analysis.

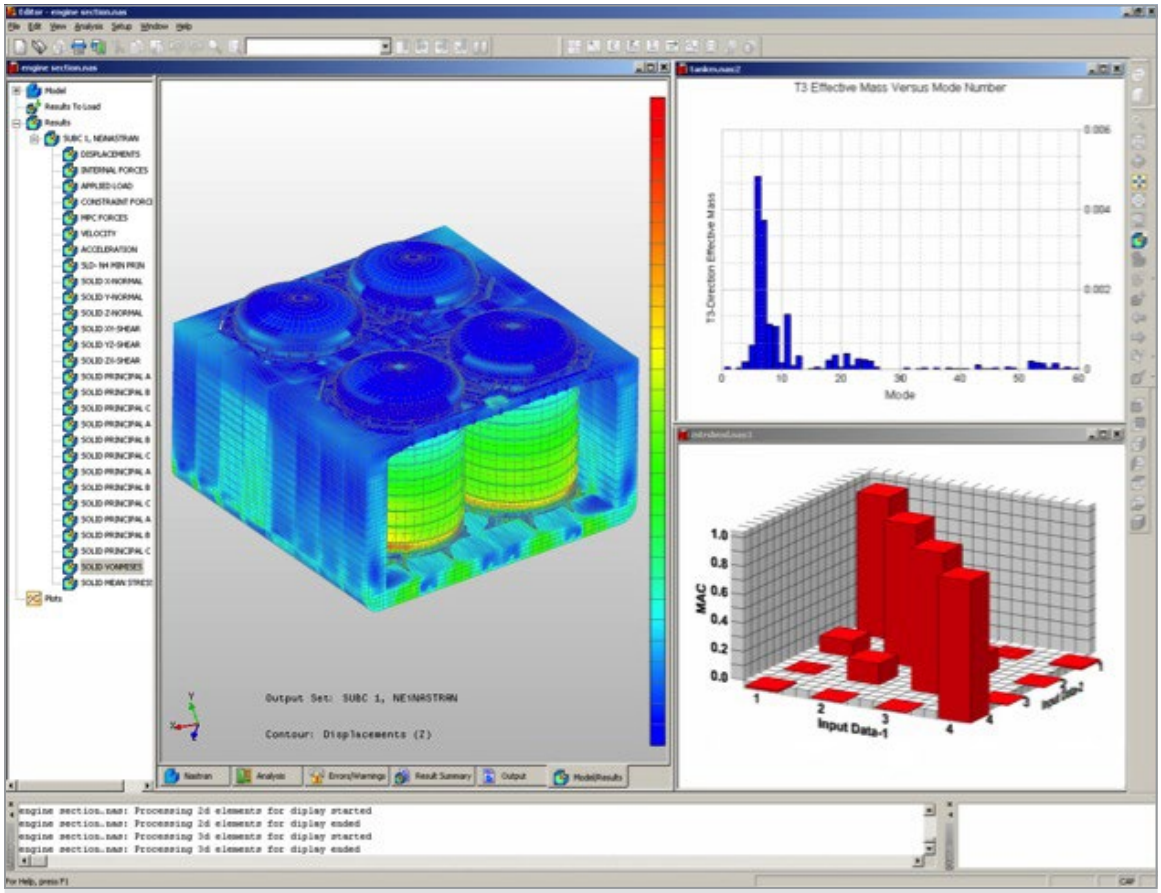


Figure 6: The NEi Nastran Editor* showing the stress contour on a section of a ship. Dynamic analyses allow multiple ways to interrogate results including 2-d and 3-d plots.

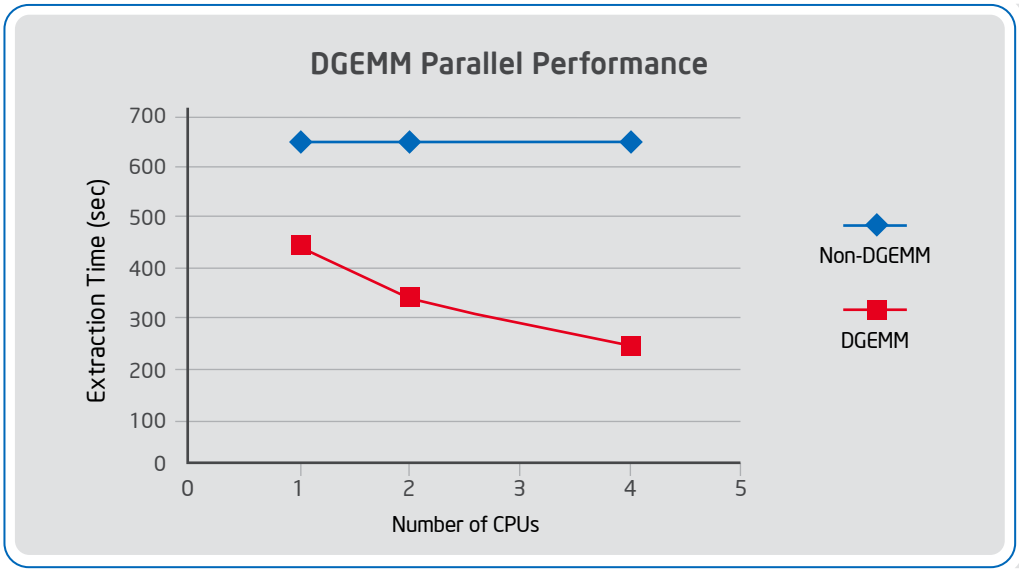


Figure 7: A comparison of the modal extraction time for the ship section. Ten modes were extracted during the analysis. (Hardware: Intel® Core™ i7 860 2.8GHz, 16GB RAM, Samsung 830* Series SSD; Software: Windows* 7 SP1 64-bit, Intel® Math Kernel Library 11.0.1.1)

```
C
            STR801 = 'EXTRACTING EIGENVALUES FOR SUBCASE '
     1               //SUBCSTR(1:NCHARSUBC)//'  ITERATION '
     2               //ITERSTR(1:NCHARITER)
            STR802 = 'VECTOR:       1  PERCENT COMPLETE:    0'
            CALL WRIT2STA( STR801, STR802)
C
            IC = 0
C
C CALCULATE THE PROJECTIONS OF A AND B.
C
            CALL ASOLNEGS( UU, A, ADIAG, LA, NA, EIGVEC, NSOL, INCREMENT,
     1                     IC, INTERVAL, NC)
C
            CALL DGEMM( OPXT, OPXN, NC, NC, NN, ONE, EIGVEC, NDOF, UU, NDOF,
     1                  ZERO, AR NC)
C
            DO J=1, NC
             DO K=1, NN
              EIGVEC(K, J) = UU(K, J)
             ENDDO
            ENDDO
C
            CALL AMULTEGS( UU, A, ADIAG, B, BDAIG, EIGVEC, LA, NA,
     1                     SHIFTFLAG, SHIFT, INCREMENT, IC, INTERVAL,
     2                     NC, 'B')
C
            CALL DGEMM( OPXT, OPXN, NC, NC, NN, ONE, EIGVEC, NDOF, UU, NDOF,
     1                  ZERO, BR NC)
C
            IF ( .NOT.(CONVERGED) ) THEN
             DO J=1, NC
              DO K=1, NN
               EIGVEC(K, J) = UU(K, J)
              ENDDO
             ENDDO
            ENDIF
```

Figure 8: DGEMM call. DGEMM is optimized for large/dense matrix multiplications. In the above subspace eigensolver routine, DGEMM is used to expand eigenvectors from modal to physical space. Matrix sizes range from a square matrix of 10 to 1,000 rows and columns being multiplied by a rectangular matrix of 10 to 1,000 by 1,000 to 1,000,000. OPXT and OPXN are variables that define if the matrix or matrix transpose should be multiplied. NC and NN are the matrix dimensions where NN (the model size in degrees of freedom) is typically much larger than NC (the number of eigenvalues to be determined).

```
C
            PHASE = 33
C
C SPARSE FORWARD AND BACKWARD SUBSTITUTION.
C
            CALL PARDISO( PPOINTER, MAXFACTMATRIX, MATRIX, PMTYPE, PHASE,
     1                    NFDOF, A, LN, LA, INTNUM, NRHS, INTPARAMETER,
     2                    MSGLVL, S, X, ERRORSTATUS)
            DEALLOCATE (LN)
C
            IF ( ERRORSTATUS .NE. 0 ) THEN
             IF ( ERRORSTATUS .EQ. PSS_INSUFFICIENT_MEMORY ) THEN
              GOTO 307
             ELSE
              IOSTATUS = ERRORSTATUS
              GOTO 207
             ENDIF
            ENDIF
C
C UNPARTITION X.
C
            CALL VSPR2GLB( X, V, LP)
```

Figure 9: Call to PARDISO for forward/back substitution and solution. Once the matrix is factorized, as shown in **Figure 5**, it can be repeatedly solved for various load cases or right-hand sides. It is more convenient and optimal to do this in FEA, especially in nonlinear analysis where many of these forward/back substitution calls are needed.

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

DGEMM implementation is also straightforward. A typical DGEMM call is shown in **Figure 8**. Here the multiplication can be performed on very large matrixes. In this example, two rectangular matrixes of size NC x NDOF are multiplied to form a square matrix of size NC. NC ranges typically from 10 to 1,000 and NDOF from 1,000 to 1,000,000. In **Figure 9** we show that the call to ASOLNEGS also uses the PARDISO solver, and its fast backsolve performance further takes advantage of Intel MKL.

In summary, FEA technology gives engineers the ability to examine structures with an incredible degree of thoroughness and flexibility. They can consider structures with complicated geometry, explore designs using new materials with nonlinear properties, and determine the effects that a variety of service conditions will have, including environmental forces like wind and earthquakes. Fast and accurate solutions are essential in this endeavor. Intel MKL PARDISO and DGEMM routines have been very effective in reducing analysis time, while allowing for larger model sizes and complexity. This increases analysis accuracy, while maintaining analyst productivity. It is hard to think of an engineered product category that has not been touched by FEA technology. The benefit has been new levels of product performance with exceptional quality and reliability. □

Read the blog:
Transforming Product Engineering: Fast, Accurate Finite Element Analysis using NEi Nastran

"The result is a productive and insightful design process that brings a cascade of benefits and savings across subsequent steps in the product development cycle—fewer prototypes, less physical testing, and a reduction in the number of design iterations before release to manufacturing. The upshot is lower costs, faster time to market, and better quality products."

BLOG highlights

But Will It Scale?

JACKSON M, (Intel)

Has this ever happened to you: You work tirelessly to add threads to your serial code, all your correctness tests are passing, and your application is zooming along almost twice as fast as the serial version on your 2-core machine. Now your friend sees your results and would love to run your program on his machine which is fully loaded with four cores that are all equipped with Intel® Hyper-Threading Technology (that's eight "logical" processors). You're expecting your newly parallelized application to be blazing fast on his machine, maybe even four times faster than it was on yours! But to your dismay ... it runs the same speed as it did on the 2-core machine. What's going on? One possibility is that you have a scaling problem.

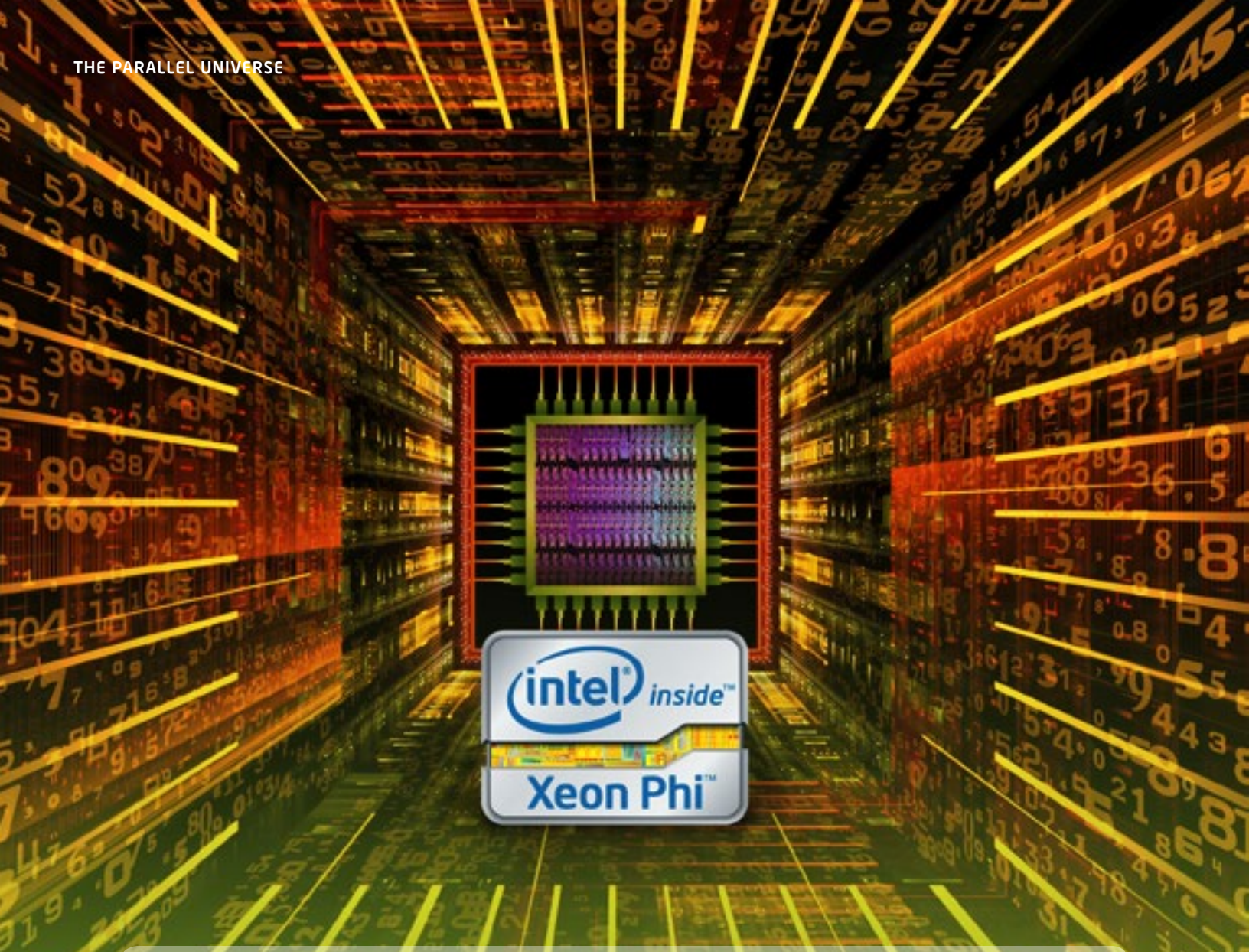
A scaling problem arises when parallelized software isn't designed to take advantage of more cores when they are available in the hardware. For example, task-level parallelism, where a predetermined number of jobs are assigned to the threads, will never scale to core counts beyond the total number of jobs created. There just isn't enough division of labor to take advantage of more hardware.

Creating parallel software that scales is essential to developing applications that will remain relevant and competitive as hardware evolves without a major redesign effort. Intel® Advisor XE can give you confidence that your newly parallel solution will scale to higher core counts BEFORE you invest the time into threading your code.

SEE THE REST OF JACKSON'S BLOG:

Visit **Go-Parallel.com**

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



PARALLEL POWER:

Optimize Software for Intel® Xeon Phi™ Coprocessors

This article outlines a software optimization methodology appropriate for applications currently optimized for Intel® Xeon® processors and targeting Intel® Xeon Phi™ coprocessors.

by Amanda Sharp, *Technical Consulting Engineer, HPC Compiler Support, Intel*

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

Overview

The Intel® Xeon Phi™ coprocessor is a highly parallel processor, based on Intel® Xeon™ processor architecture extended with up to 61 cores and a 512-bit wide vector engine. Both processors share common programming languages, coding techniques, and software tools. Optimizations already implemented in software running on Intel Xeon processors also benefit applications running on Intel Xeon Phi coprocessors. By following the steps below, your application may yield even more performance gains.

Code Optimization Recipe

1. Determine Suitability: Verify that the application meets the requirements for good performance on a highly parallel architecture.
2. Choose an Execution Model: Send computations from the host system to the coprocessor or run the application directly on the coprocessor.
3. Tune the Code: Use parallelization and vectorization techniques to tune the code for Intel Xeon Phi coprocessors.

1. Determine Suitability

When targeting a highly parallel architecture, you will need to determine if your application or a sufficient portion of your application is likely to run optimally on the target. Applications that can make effective use of the considerable hardware resources available on Intel Xeon Phi coprocessors generally have the following characteristics:

- **Highly parallel algorithms scalable to a minimum of 100 threads:** Each processor core supports four hardware threads. At runtime, a typical application may have more than 200 active threads. Your parallel implementation should show close to linear scaling up to the maximum number of CPU cores.
- **A significant amount of efficiently vectorized code:** Each processor core contains a vector processing unit (VPU), the main source of computational power. Your vectorized application should deliver significant speedup when compared with the non-vectorized implementation.

Suitable algorithms typically spend at least 90 percent of execution time in parallel and vectorized code segments after optimization. If your application is already highly parallelized and highly vectorized for Intel Xeon processors and it meets these criteria, you are ready to select an execution model and tune your code.

2. Choose an Execution Model

An Intel Xeon Phi coprocessor can be programmed as a coprocessor(s) or as an autonomous processor. The appropriate model depends on application and context.

Heterogeneous Execution

Heterogeneous (offload) execution occurs when the host system executes scalar portions of the application and delegates parallel segments to the coprocessor. The host and coprocessor do not share memory, thus all data exchange happens over the PCI-E bus. The Intel® C/C++ and Intel® Fortran Compilers support language extensions for offload, based on C++ pragmas and Fortran directives. This includes extensive support for data allocation and marshaling, including overlapping data transfer with computation. The Intel C++ Compiler also supports a second model that creates and manages a virtual shared memory system for C++ applications.

The Intel® Compilers automatically detect offload language extensions in your code and create a binary that runs on both the host and the coprocessor. See The Heterogeneous Programming Model for more information about offload programming syntax. Effective Use of Compiler Features for Offloading is a comprehensive article on effectively programming Intel Xeon Phi coprocessors that includes tuning tips.

Native Execution

Native execution occurs when an application runs entirely on the coprocessor. For best performance, a native application should have very few serial segments, limited I/O usage, and be smaller than the physical memory on the coprocessor. Intel Xeon Phi coprocessors run a Linux*-based operating system. Use the Intel Compiler option `–mmic` to compile and generate a binary for the coprocessor. Then, connect to the coprocessor via a secure shell, copy the required binaries to the coprocessor, and run the application. For more information, refer to Building a Native Application for Intel® Xeon Phi™ Coprocessors.

3. Tune the Code

There are many techniques for tuning code for Intel Xeon Phi coprocessors. The amount of performance gain you experience depends on your code and the number of techniques you use in your application.

Efficient Parallelization

Intel Xeon Phi coprocessors support four thread contexts per core. Given the large number of cores, implementing efficient parallelization to take advantage of all parallel resources is the key to maximizing performance. This usually requires the use of at least two threads per core. Here we will discuss key recommendations for OpenMP* -based code.

REDUCE SYNCHRONIZATION COSTS

Remove or reduce any use of barrier synchronization, locks, and critical sections in your code as far as possible, consistent with correctness. Use reduction operations where possible.

LOOP SCHEDULING

By default, the runtime library implements static loop scheduling, which may not be ideal for some applications. Workloads that run fine with eight or sixteen threads can exhibit significant load imbalance with static loop scheduling over a large number of cores. Try specifying different loop scheduling algorithms and chunk sizes to improve performance. In some cases, using the OMP collapse clause can help.

THREAD AFFINITY CONTROL

Thread affinity can have a dramatic effect on the execution speed of a program. Determine the optimal number of application threads. Try different numbers of threads from N-1 threads to 4 x (N-1) threads, where N is the number of physical cores on the coprocessor. Use a maximum of N-1 threads to avoid scheduling worker threads on the coprocessor core that runs the OS and offload services.

Keep in mind that the default values of OpenMP parameters may vary between host and coprocessor, and between offloaded and native execution. You can use Best Known Methods for Using OpenMP* as a guide. Intel® Composer XE supports a variety of parallelization methods for Intel Xeon Phi coprocessors. See the article Efficient Parallelization, OpenMP* Thread Affinity Control for details.

Vectorization

Intel Xeon Phi coprocessors provide a wide vector unit for processing highly data-parallel workloads. The techniques below will help you to take advantage of the specialized VPU, which is essential for optimal performance.

DATA ALIGNMENT

Data alignment is very important for Intel Xeon Phi coprocessors. Using proper data alignment will streamline the process of loading and storing data. There are two steps:

1. **Align your data on 64-byte boundaries.** For C and C++, use `__attribute__((aligned(64)))` for static arrays and `_mm_malloc()` and `_mm_free()` for managing dynamic data. Compile Fortran applications with `-align array64byte`.
2. **Alert the compiler that data are aligned so it can generate vectorized code.** One method is to insert pragmas/directives before a loop. For C and C++, use `#pragma vector aligned` or for Fortran, `!dir$ vector aligned`.

MEMORY ACCESS PATTERNS

Non-unit-stride memory accesses can lead to inefficient vectorization and can have considerable impact on performance. This occurs when consecutive iterations of your inner loop access memory from non-adjacent locations. This pattern may also cause cache misses if the data elements come from different cache lines. Adopt vector friendly data structures and algorithms that maximize use of unit-stride vectorization in all hotspots. In some cases, use of vector array notation can help. In other situations, you may have to change the data layout from "array of structures" to "structure of arrays." For more information on this topic, refer to Memory Layout Transformations.

ENFORCEMENT

The SIMD pragma/directive (#pragma simd or!dir$ simd with appropriate clauses added) is a powerful feature that tells the compiler to vectorize a loop. By default, the compiler attempts to vectorize innermost loops in nested loop structures. However, if an outer loop contains more work, a combination of elemental functions, strip mining, and pragma/directive SIMD can force vectorization at the outer level. You can use the Intel Compiler Vectorization Reports as a guide.

ADVANCED OPTIMIZATIONS

Advanced optimizations, such as data prefetching and use of streaming stores, can improve performance of your application. Using the appropriate floating-point options, while considering performance vs. accuracy tradeoffs, can have a big impact on performance. For more information on these topics, refer to Advanced Optimizations for Intel® MIC Architecture.

Summary

Applications targeting highly parallel architectures must be able to leverage extensive hardware resources. Once your application is optimized for Intel Xeon processors, you can maximize performance on Intel Xeon Phi coprocessors by following a straightforward code optimization methodology. First, ensure that the application can benefit from execution on a highly parallel architecture. Then, select the best execution model for your application and use the recommended optimization techniques to tune the code. To learn advanced techniques, including code examples, please review the article Compiler Methodology for Intel® MIC Architecture. For more information, visit the Intel® Developer Zone for Intel® Xeon Phi™ Coprocessors. ☐

"Applications targeting highly parallel architectures must be able to leverage extensive hardware resources. Once your application is optimized for Intel® Xeon® processors, you can maximize performance on Intel® Xeon Phi™ coprocessors by following a straightforward code optimization methodology."

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

Unleash Intel® Xeon Phi™ Coprocessor Performance

by Todd Rosenquist, *Technical Consulting Engineer, Intel® Math Kernel Library, Intel Corporation,*
and Shane Story, *Manager of Intel® MKL Technical Strategy, Intel Corporation*

The Intel® Math Kernel Library (Intel® MKL) makes it easy for users to realize the performance benefits provided with each new processor including the new Intel® Xeon Phi™ coprocessor. Those who have not used Intel MKL before should look for opportunities to use its optimized kernels rather than tuning their own. Those who use Intel MKL can take advantage of support for natively optimized functions for the Intel Xeon Phi coprocessor as well as functions that automatically detect and make the best use of all the processors and coprocessors on the system—a model we call Automatic Offload (AO).

Intel MKL AO mode is simple to use. You start by linking with Intel MKL as you do now, and either call `mkl_mic_enable()` in your program or set `MKL_MIC_ENABLE=1` in your environment. Subsequent calls to Intel MKL linear algebra functions with large computation-to-data ratios will internally assess the total hardware resources available, and automatically distribute the computational work across them. This means that all the required data transfer is transparently done for you. In addition, functions that used to run only on the multicore processor now benefit automatically from the added computational power of the Intel Xeon Phi coprocessor. An added bonus is that the Intel MKL AO mode can be used with any compiler.

The natively tuned functions in Intel MKL can be used in multiple ways. You can code your program to offload these functions to the coprocessor using the pragmas supported by the Intel® compilers—called compiler assisted offload (CAO). Or, you can compile the whole program and run it by logging in to the coprocessor as described in the adjacent article.

With the introduction of Intel Xeon Phi, Intel MKL remains a key ingredient in unleashing the performance of Intel® architectures in real-world applications. Intel MKL provides top performance for the high-performance LINPACK benchmark, used to characterize the performance of the world's fastest Top500 computers.

Learn more about programming for the Intel Xeon Phi coprocessor at: http://software.intel.com/en-us/articles/intel-mkl-on-the-intel-xeon-phi-coprocessors. □

Tune OpenM APPLICATIONS

by Alexei Alexandrov, *Senior Software Developer, Intel*

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

P*

ALEXEI ALEXANDROV
Senior Software Developer

Sign up for future issues | Share with a friend

High performance computing (HPC) has a long history and today is critical to business, research, and science. Clusters consisting of thousands of machines help enable many advances of modern science with both theoretical and practical implications, working 24/7 to enrich the lives of every person on earth.

HPC parallelism is exploited in three levels: process, thread, and SIMD vectorization (including auto-vectorization, for example in the Intel® compiler). For process-level parallelism, the Message Passing Interface (MPI) is the de facto standard in the HPC industry. For thread-level parallelism, the OpenMP* programming model is prevalent in the HPC community, and other models—such as Intel® Threading Building Blocks and Intel® Cilk™ Plus—are gaining traction. While Intel® VTune™ Amplifier XE supports all of these paradigms, this article will focus on useful techniques for profiling HPC programs that use OpenMP.

The examples shown are collected from an OpenMP application running on an Intel® Xeon Phi™ coprocessor, but the techniques used are valid for tuning all OpenMP programs. The examples use Intel VTune Amplifier's command line to collect data, as this is the most common way to gather data on systems with Intel Xeon Phi processors. But you don't need to worry about learning all the switches. The graphic user interface (GUI) is commonly used to set up the analysis and generate a command line that you can cut and paste.

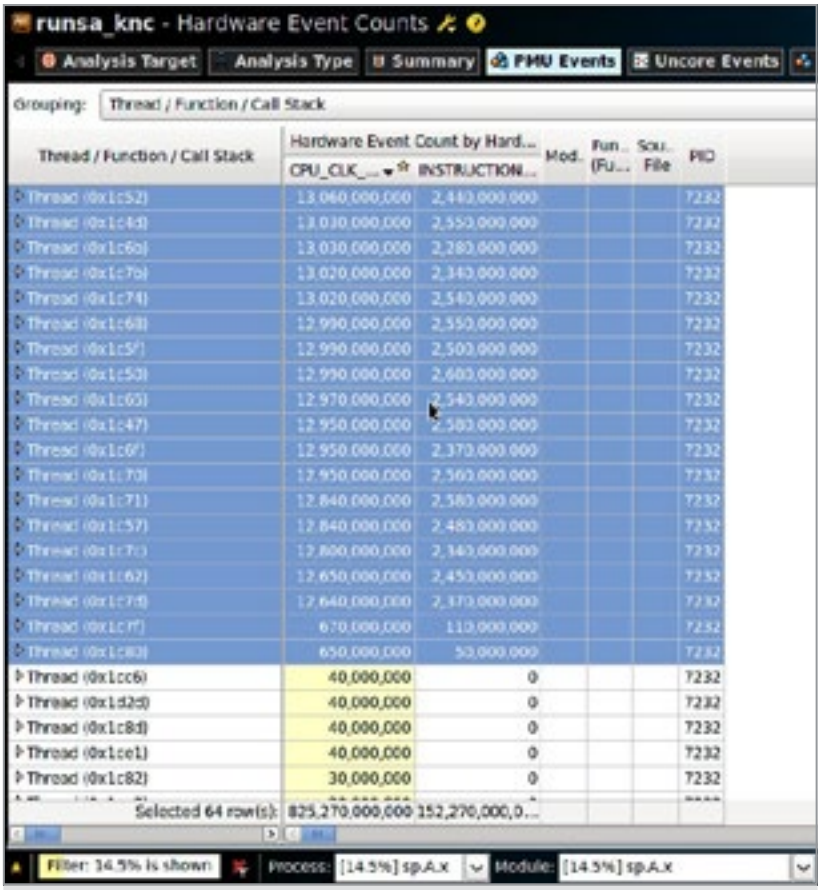


Figure 1: Using grouping and filtering to analyze the balance of an OpenMP* program with Intel® VTune™ Amplifier XE.

Profiling OpenMP programs with Intel VTune Amplifier XE

We'll begin by walking through a simple scenario of collecting and analyzing performance data for an OpenMP program. This will include three steps:

1. Collect the data using a command-line interface.
2. Analyze the data using an interactive GUI, with examples of capabilities such as filtering and loop analysis.
3. Generate the hotspot profile using an Intel VTune Amplifier command line interface. This is often useful for automating performance regression testing or feeding the data into another program.

In **step 1**, to profile a program that uses OpenMP on an Intel® Xeon® processor-based computer, you can use any of the supported analysis types, just launching the data collection as usual:

```
$ amplxe-cl -collect hotspots --
~/sp.A.x
```

or using hardware event-based sampling analysis types to profile a program executed on an Intel Xeon Phi coprocessor:

```
$ amplxe-cl -c knc-lightweight-
hotspots -search-dir all:p=/lib/
firmware/mic -- ssh mic0 ~/sp.A.x
```

The search directory is specified here to make sure the Intel Xeon Phi runtime binaries can be found on the host during result post-processing.

In **step 2**, having collected the result and opened it in the Intel VTune Amplifier XE GUI, you can begin your analysis. For an OpenMP program, it is a good idea to start looking into data by grouping the data by threads in the Bottom-up view. To do this, select a thread-based grouping in the Grouping combo-box above the grid (**Figure 1**). Selecting several rows in the grid allows you to easily see the number of selected threads and the summary statistics for them—this is useful for understanding how a given team of threads behaved. By right-clicking the selected items, you can also filter in or filter out the selected data. Combined usage of grouping and filtering allows you to dice and slice the data as needed. The filter feature is particularly useful for filtering out the time spent in the OpenMP runtime library to see the pure contribution of the user code

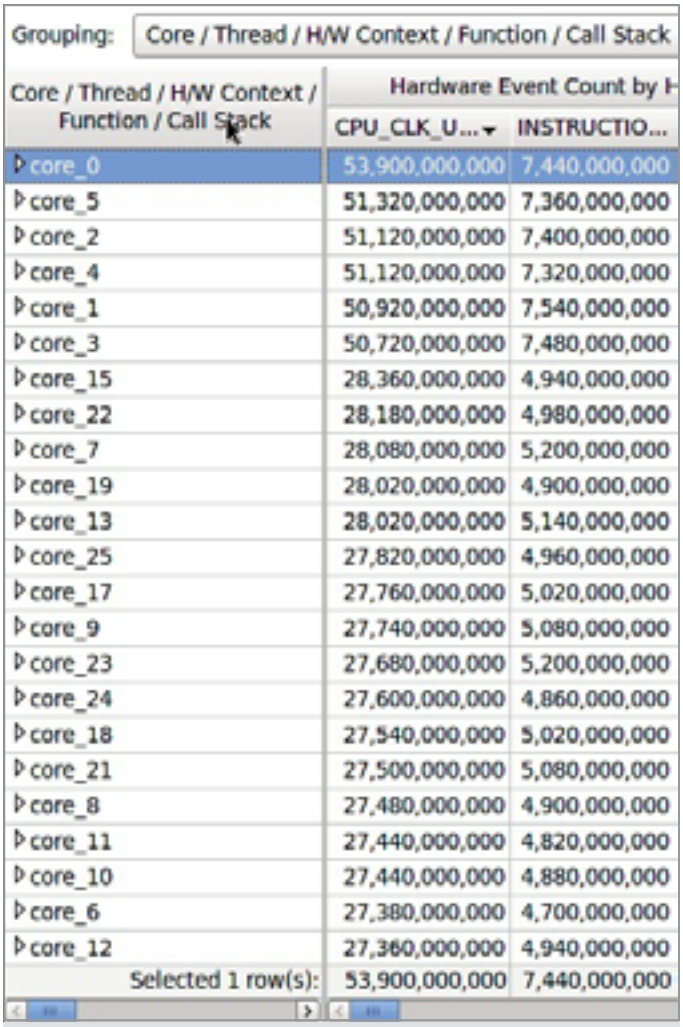
For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

on overall performance. For example, **Figure 1** shows that the user module only contributed 14.5 percent of CPU cycles; the rest of the cycles were spent in other modules, mostly spinning because of non-optimal CPU affinity and the number of OpenMP threads running.

In the case above, using the thread grouping revealed that only 64 software threads were effectively executing the user code—although there were 244 hardware threads available since this is an Intel Xeon Phi coprocessor with 61 cores. From the program source code, it became clear that the available parallelism is limited by one dimension of the input problem size, so that the program needs to be changed to adopt the higher available parallelism. Or, at the very least, the affinity and number of OpenMP threads to use should be set to match the workload properties. Setting `KMP_AFFINITY` to "balanced" and `OMP_NUM_THREADS` to 64 indeed provided better execution time and better balance between the threads. Since the Intel® OpenMP Library actively uses spinning instead of waiting, the CPU time spent outside of the user module (and usually in the OpenMP module) is a common indication of imbalance or excessive serial execution. You can also choose grouping by Core to understand the program balance in terms of hardware cores. This is often useful to understand the performance of Simultaneous Multithreading (SMT) parallelism (**Figure 2**).

Having identified and fixed high-level balance and overhead issues, you can narrow down to function-level analysis (**Figure 3**). When using the Intel compiler together with the Intel OpenMP implementation, the OpenMP region bodies are conveniently aggregated into pseudo functions with names like `compute_rhs_$omp$parallel@17`, so that it's easier to distinguish the time spent inside and outside of the region body. For instance, in this case the name reads as "the OpenMP Parallel Region at line 17 in function compute_rhs."

Knowing a hot function, you can dive into its C, C++, or Fortran source and assembly to identify which source lines were taking most of the time and which assembly code was generated by the compiler for them. Understanding the latter often provides guidance for how you should direct the compiler to vectorize the inner loops. Starting with version 3.0, Intel VTune Amplifier XE also helps you understand the structure of the program in terms of loops (useful for loopy HPC codes). To enable that mode, switch the "Loop Mode" to "Loops and functions" in the GUI filter bar (or use "-loop-mode=loop-and-function" in the amplxe-cl command line interface). The top-down view will show the looping structure of the program (**Figure 4**). In this example, we can easily see that the OpenMP region function has a loop at line 295, which nests to loop at line 296, and then to loop at line 297—with the latter loop being peeled by the compiler as part of the vectorization process. This is identified by observing two loop instances belonging to the same source line. Note that the first of the peeled instances takes the larger fraction of the time, since the second instance is a remainder loop with a small iteration count.



Grouping: Core / Thread / H/W Context / Function / Call Stack

Core / Thread / H/W Context / Function / Call Stack	Hardware Event Count by H	
	CPU_CLK_U...	INSTRUCTIO...
core_0	53,900,000,000	7,440,000,000
core_5	51,320,000,000	7,360,000,000
core_2	51,120,000,000	7,400,000,000
core_4	51,120,000,000	7,320,000,000
core_1	50,920,000,000	7,540,000,000
core_3	50,720,000,000	7,480,000,000
core_15	28,360,000,000	4,940,000,000
core_22	28,180,000,000	4,980,000,000
core_7	28,080,000,000	5,200,000,000
core_19	28,020,000,000	4,900,000,000
core_13	28,020,000,000	5,140,000,000
core_25	27,820,000,000	4,960,000,000
core_17	27,760,000,000	5,020,000,000
core_9	27,740,000,000	5,080,000,000
core_23	27,680,000,000	5,200,000,000
core_24	27,600,000,000	4,860,000,000
core_18	27,540,000,000	5,020,000,000
core_21	27,500,000,000	5,080,000,000
core_8	27,480,000,000	4,900,000,000
core_11	27,440,000,000	4,820,000,000
core_10	27,440,000,000	4,880,000,000
core_6	27,380,000,000	4,700,000,000
core_12	27,360,000,000	4,940,000,000
Selected 1 row(s):	53,900,000,000	7,440,000,000

Figure 2: Using grouping by Core to understand the hardware core balance.

"Applications targeting highly parallel architectures must be able to leverage extensive hardware resources. Once your application is optimized for Intel® Xeon® processors, you can maximize performance on Intel® Xeon Phi™ coprocessors by following a straightforward code optimization methodology."

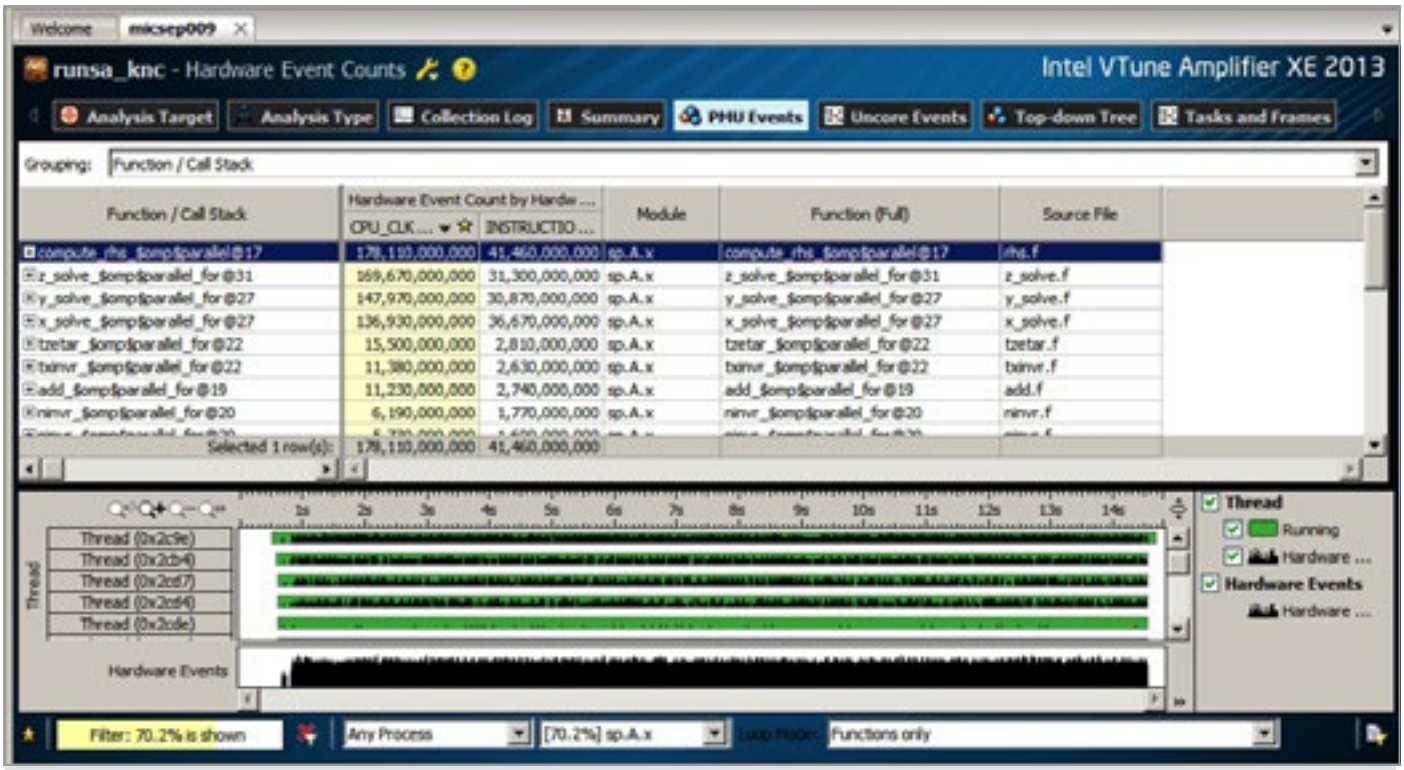


Figure 3: Function-level hotspot view.

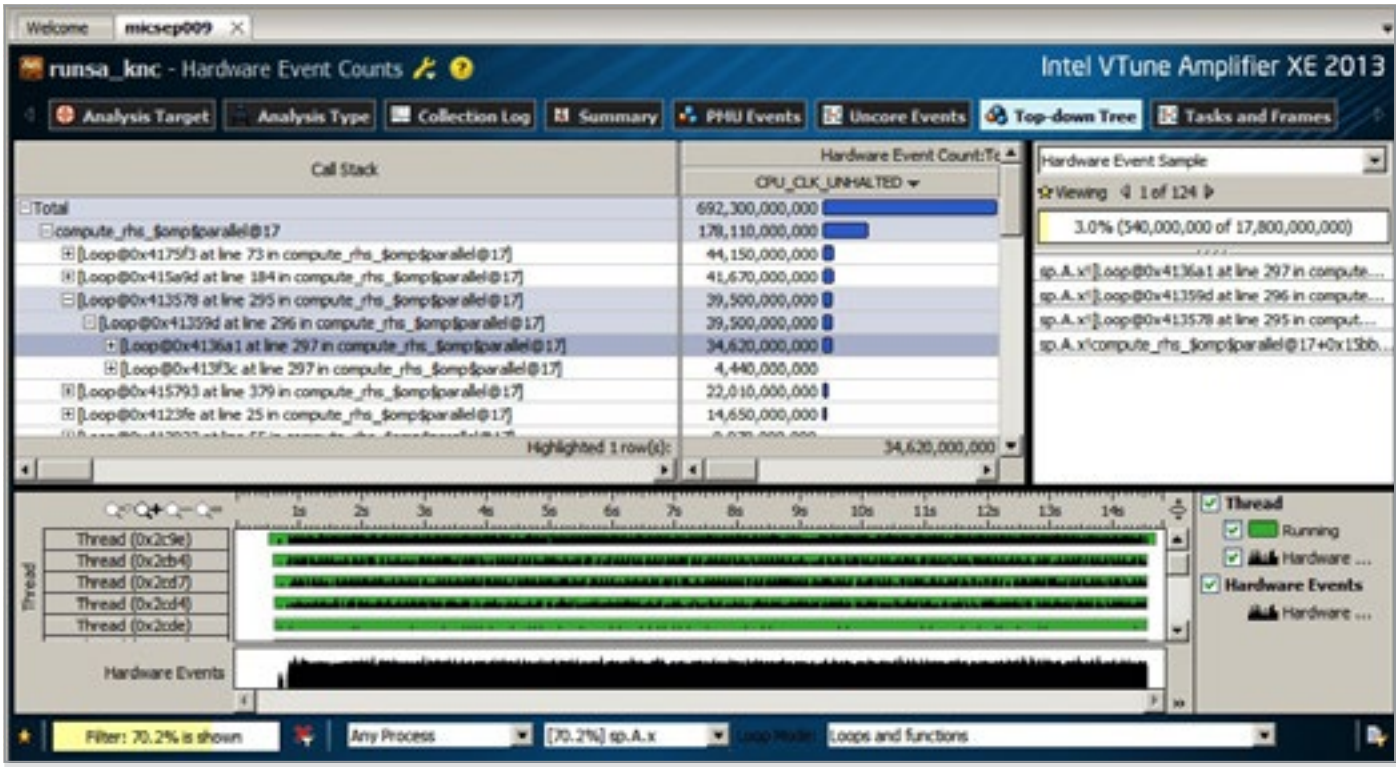


Figure 4: Using loop analysis to understand the looping structure of the program.

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

The timeline view can be used to narrow down the scope of the data to a specific time region. For example, to skip the program startup "cold" phase or to magnify the data into the execution of specific region. Using the ITT API, supported by Intel VTune Amplifier, to mark up the program execution may be useful for the latter. When viewing results that ran many threads, use the "Tiny" mode of timeline bands to easily fit more data onto your display:

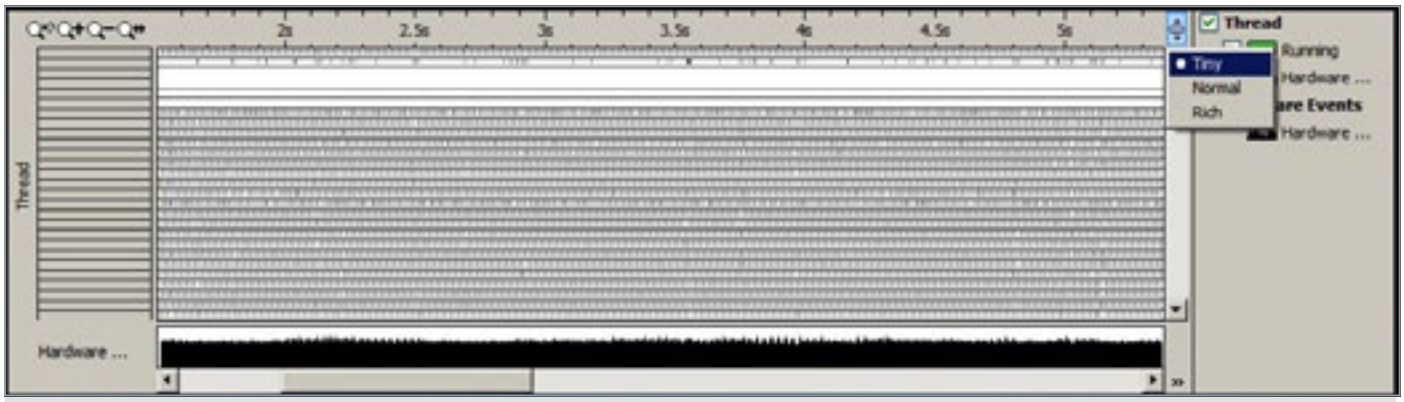


Figure 5: Using tiny mode for timeline bands to view many-thread data over time.

Finally, in step 3 of our simplified optimization flow, it may be useful to automate the hotspot extraction using the command line interface (e.g., for automatic performance regression testing). Most of the capabilities mentioned above can be used from the command line. For example, here you can see how to output the ten top hotspots grouped by thread and function, filtered by a specific module and time region, and with loop analysis on:

```
$ amplxe-cl.exe -R hotspots -loop-mode=loop-and-function -filter module=sp.A.x -limit=10 -group-by
thread,function -r <result directory path>

Thread           Function                                                          Module  CPU Time:Self
---------------  ----------------------------------------------------------------  ------  -------------
Thread (0x2cd3)  [Loop@0x4136a1 at line 297 in compute_rhs_$omp$parallel@17]  sp.A.x          0.394
Thread (0x2cd5)  [Loop@0x4136a1 at line 297 in compute_rhs_$omp$parallel@17]  sp.A.x          0.385
Thread (0x2cb5)  [Loop@0x4136a1 at line 297 in compute_rhs_$omp$parallel@17]  sp.A.x          0.376
Thread (0x2cd7)  [Loop@0x4136a1 at line 297 in compute_rhs_$omp$parallel@17]  sp.A.x          0.376
Thread (0x2cc1)  [Loop@0x4136a1 at line 297 in compute_rhs_$omp$parallel@17]  sp.A.x          0.358
Thread (0x2ccb)  [Loop@0x4200d8 at line 294 in x_solve_$omp$parallel_for@27]  sp.A.x          0.358
Thread (0x2ca2)  [Loop@0x4136a1 at line 297 in compute_rhs_$omp$parallel@17]  sp.A.x          0.349
Thread (0x2ccf)  [Loop@0x4200d8 at line 294 in x_solve_$omp$parallel_for@27]  sp.A.x          0.339
Thread (0x2cd2)  [Loop@0x4200d8 at line 294 in x_solve_$omp$parallel_for@27]  sp.A.x          0.339
Thread (0x2cab)  [Loop@0x4136a1 at line 297 in compute_rhs_$omp$parallel@17]  sp.A.x          0.330
```

Figure 6

Conclusion

The Intel VTune Amplifier XE techniques described here are useful for diving into performance analysis of an HPC program, but there is still more to learn. The product documentation and online resources (see the Intel Knowledge Base) can provide further information on the features of the product, including MPI program analysis, loop and inline function analysis, ITT API usage, performance analysis automation using command-line reporting, and many more. The techniques were illustrated using an Intel Xeon Phi coprocessor, but apply equally well to an Intel® Xeon® system. One nice benefit is that tuning to improve the parallelism in your application usually yields performance benefits when running on both Intel Xeon processors and Intel Xeon Phi coprocessors: a double win! □

Expand Your Debugging Options

by Nicolas Blanc, *Software Engineer, Intel*
and Georg Zitzlsberger, *Technical Consulting Engineer, Intel*

Overview

Execution of typical applications for the Intel® Xeon Phi™ coprocessor is distributed among the host and one or more coprocessors. In general, there are two basic domains for Intel Xeon Phi coprocessor-based applications:

- Heterogeneous applications that execute seamlessly across the host and selected coprocessors using either the explicit or implicit offload model
- Applications solely executed on each of the available coprocessors, using the so-called native model

Developing and debugging applications in both native and offload models requires dedicated support. Intel® Parallel Studio XE 2013 for Linux* provides a range of solutions, along with ease of use and full awareness of the Intel Xeon Phi architecture. Vendors, such as Allinea or Rogue Wave, enrich the Intel Xeon Phi environment with further debugging options. Thus, developers for Intel Xeon Phi can select among many debugging options, depending on their field of application. Next, we'll look at a selection of Intel debugging solutions, grouped by the two typical domains.

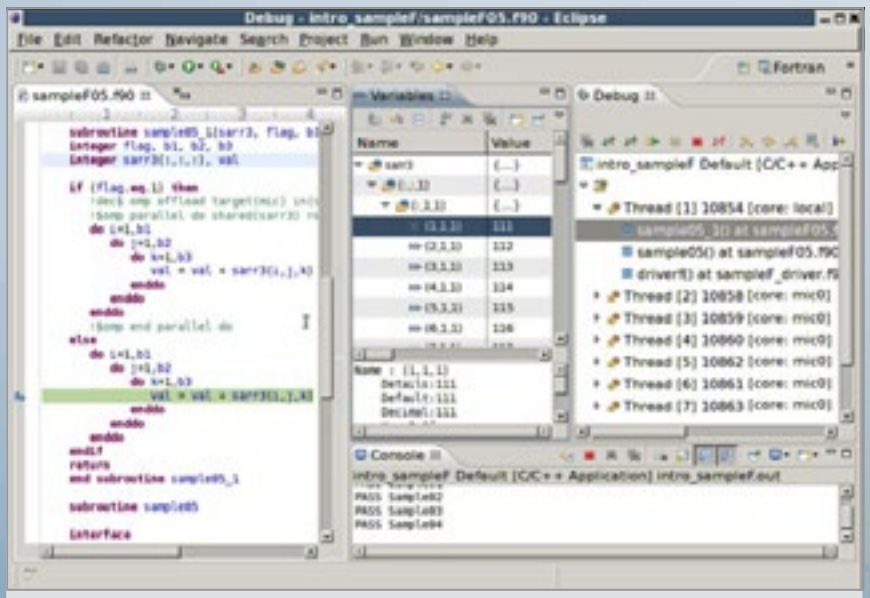


Figure 1

Offload Model

There are two different offload models: explicit and implicit. Both provide a simple, yet flexible programming environment to develop applications running on the Intel Xeon Phi coprocessors *and* host systems. An offload application comes as a single executable file containing both host and target code. Detection of available coprocessors, including data transfers, is automatically handled at runtime. Therefore, work packages are scheduled transparently among any coprocessors and the host. In case no coprocessor is available, the execution remains entirely native on the host system.

The programming flexibility makes debugging with standard tools more complex. Therefore, Intel Parallel Studio XE 2013 for Linux provides an Eclipse* debugger plugin with full awareness of both offload models. This allows instant debugging without further configuration. Using Eclipse as an integrated development environment (IDE) provides an easy-to-use, well-known graphical interface for this debugger. The integration also offers scalability up to hundreds of threads on multiple coprocessors for C, C++, and Fortran—a necessity for Intel Xeon Phi because of its manycore architecture.

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

Native Model

For the native model—where applications run exclusively on the coprocessor—different debug solutions are provided by the Intel Parallel Studio XE 2013 suite for Linux. Here, development takes place on one system while the created applications are running on a remote coprocessor. Such a coprocessor may not necessarily be installed on the same development system, but can be installed on another host, reachable via network.

Debug support for this use case requires remote capabilities, which are also available via the aforementioned Eclipse plugin. A second solution is the Intel® Debugger (IDB) for command-line debugging. A typical IDB debug session starts with the following simple steps:

```
$ idbc_mic –tco –rconnect=tcpip:<coprocessor>
(idb) idb file-remote <path_to_executable_on_coprocessor>
(idb) file <path_to_executable_on_dev_system>
...
```

The debugger is started via `idbc_mic` by providing the name or IP address of the coprocessor. When the debugger is started, the path of the executable to be run on the coprocessor is specified first, followed by the path of the same executable on the development system that launched the debugger. Afterwards, IDB can be used as usual.

Alternatively, IDB can also attach to an application already running on the coprocessor via its process ID `<pid>`.

```
$ idbc_mic –tco –rconnect=tcpip:<coprocessor>
(idb) attach <pid> <path_to_executable_on_dev_system>
...
```

The advantages of using IDB are its speed, compatibility with C, C++, and Fortran, and GNU GDB* syntax for a flat learning curve. With its focus on the command line, it can easily be used for automated script-based testing as well. More information on how to use IDB can be found in the Intel® Debugger User's and Reference Guide.

A third solution for the native mode is using GNU GDB for Intel Xeon Phi. Like IDB, it provides remote debugging capabilities; in addition, it can be hosted directly on the coprocessor. Developers preferring GDB on the host can continue using it for debugging on the coprocessor. Intel added support to GDB for the Intel® Many Integrated Core (Intel® MIC) architecture of Intel Xeon Phi. This version is not part of Intel Parallel Studio XE 2013 for Linux, but can be downloaded from our Intel® Many Integrated Core Architecture Forum.

Summary

The wide variety of debugging tools supports the different, versatile use cases of the Intel Xeon Phi coprocessor. Developers can choose between comfortable GUI based or fast, low overhead command line debuggers. There are a number of different vendors offering such solutions, including Intel. □

Learn More

Intel® Parallel Studio XE 2013

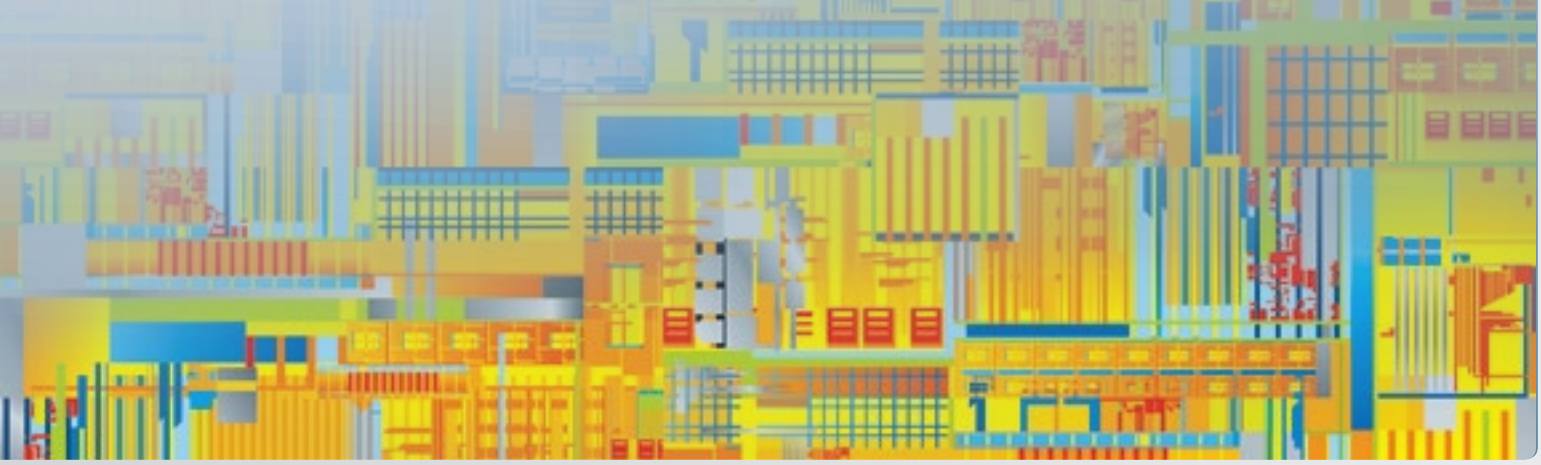
Intel® Xeon Phi™ coprocessor and Intel® Many Integrated Core Architecture (Intel® MIC)

Intel® Debugger User's and Reference Guide

Intel® Many Integrated Core Architecture resources (including GNU GDB sources)

Allinea releases tools for Intel® Xeon Phi™ coprocessor developers

Rogue Wave announces support for the Intel® Xeon® Phi™ coprocessor in key products



PARALLEL EXECUTION USING HTML

by Max Domeika,
HTML5 Product Manager, Intel Corporation

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

HTML5 Moves Forward

Evolving humbly from a text markup language,

HTML5 is embracing parallel processing to better meet developers' demands for a high performance application platform. The HTML5 language specification, currently under development by the World Wide Web consortium (W3C) with completion targeted for the end of 2014, includes improvements in source code readability, multimedia, graphics, device access, offline storage, connectivity, and performance over the previous HTML4 specification.

APIs Support Multicore

Like most developers, HTML5 developers value application performance and have developed a number of APIs to expose the performance potential offered by multicore processors. These APIs offer programmability of both homogenous and heterogeneous multicore processors and include Web Workers*, Parallel JavaScript*, WebGL*, and WebCL*. Widespread adoption in HTML5 applications is just beginning. However, the expectation is for continued growth as example applications spur demand and a greater number of browser vendors implement support. Let's look at some examples of these technologies.

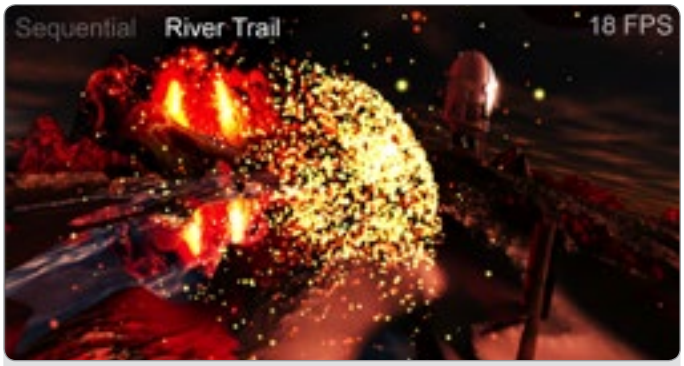


Figure 1: A glimpse of what is possible with Parallel JavaScript*

The **Web Workers** API specifies a message-passing-based programming model that can be likened to traditional multiprocessing. Instantiated Web Workers do *not* share application data space with each other or with the primary Web object, referred to as the Document Object Model (DOM). Web Workers are supported by most major browsers and the API specification is in the final phases of approval.

Figure 2 shows a sample program employing Web Workers and comprising two files, main.js and compute.js. The code in main.js creates a new Web Worker, which executes the code in compute.js. Communication is managed through the onMessage property and postMessage method.

WebGL is a JavaScript API based on the OpenGL ES 2.0* standard from the Khronos Group. It allows calling of the 3D graphics primitives, which act within the HTML5 canvas object. The HTML5 canvas is a pixel-based drawing surface, enabling both 2D and 3D graphics. WebGL leverages the parallel processing capabilities available in graphics accelerators. For more details, refer to the WebGL website.

Another standard defined by the Khronos Group is WebCL, which provides JavaScript bindings to OpenCL, a native standard for heterogeneous multicore processing. The programming model for WebCL is characterized as an accelerator model, where you create a compute kernel containing the desired processing and dispatch the kernel and the data to a series of accelerators.

Parallel JavaScript, also known by the codename River Trail, extends JavaScript with data-parallel constructs. The technology leverages OpenCL, while providing a higher level of abstraction than the accelerator model exposed by WebCL. Parallel JavaScript extends JavaScript by adding the ParallelArray data structure and parallel operations, such as map, combine, reduce, scan, scatter, and filter. **Figure 3** shows two JavaScript code snippets to compute the sum of a set of values: one that executes serially and another that employs Parallel JavaScript. For more details, refer to the Intel Labs' River Trail Wiki.

```
//main.js
var worker = new Worker('compute.js');
worker.onmessage = function(event) { alert (event.data); };
worker.postMessage('information');

//compute.js
self.onmessage = function(event) {  // Processing task
self.postMessage("received: " + event.data); };
```

Figure 2: Web Workers* example

```
//Serial
var val = new Array(1,2,3,4,5,6);
var sum = 0;
for (var i=0; i<6; i++){
  sum += val[i];
}

//Parallel
var val = new ParallelArray([1,2,3,4,5,6]);
var sum = val.reduce(function plus(x,y) { return x+y; });
```

Figure 3: Parallel JavaScript* example

Choosing an API

The following can help guide which API should be assessed first for deployment in your application.

My application can benefit from:	API for consideration:
Parallelism provided via dedicated access to the Graphics Processing Unit	WebGL
More general-purpose task parallelism that is not vector or graphical in nature	Web Workers
Performing vector operations in parallel, but with greater programmability than afforded by WebGL	
> I prefer an accelerator programming model where compute tasks are bundled and dispatched to the processing unit	WebCL
> I prefer a higher abstraction for parallel computation	Parallel JavaScript

Table 1

Conclusion

HTML5 is embracing multicore processing, as shown by these four API standards. Expect continuing evolution and refinement as these specifications are implemented by developers and engineers. ☐

"Like most developers, HTML5 developers value application performance and have developed a number of APIs to expose the performance potential offered by multicore processors. These APIs offer programmability of both homogenous and heterogeneous multicore processors and include Web Workers*, Parallel JavaScript*, WebGL*, and WebCL*."

BLOG highlights

Rogue Wave tools support Intel® Xeon Phi™ coprocessors

JAMES REINDERS, (Intel)
Director of Parallel Programming Evangelism

Rogue Wave Software recently **announced expansion of their support of Intel Xeon Phi coprocessors** which will now include their SourcePro* C++, IMSL* Numerical Libraries, TotalView* debugger, and the ThreadSpotter* cache memory optimizer products. You can check out their **press release** for details Scott Lasica, VP Products and Alliances at Rogue Wave Software, helped me understand this value by sharing with me, "When we started porting TotalView to run on an Xeon Phi coprocessor, we progressed in a week to what took us more than a year with an accelerator." The common x86 programming model, Linux*-based environment, and suite of Intel® tools allowed them to support Intel Xeon Phi coprocessors in a few weeks instead of in many months without the Intel Xeon Phi coprocessor advantages.

I'm pleased to have Rogue Wave offering solutions for our customers for Intel Xeon Phi coprocessor development. Many of our customers have expressed to me personally how happy they are to have Rogue Wave tools supporting Intel Xeon Phi coprocessors.

As with other tools, a key benefit is that these are NOT new tools or wildly different add-ons to their tools.

SEE THE REST OF JAMES' BLOG:

Visit **Go-Parallel.com**

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

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

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.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

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**.


intel
Intel® Software Adrenaline
Mobile game-changers
think differently.
INTEL SOFTWARE
ADRENALINE
The future of software in your world.
SUBSCRIBE TODAY: Free magazine and more
softwareadrenaline.intel.com
© 2013 Intel Corporation. All rights reserved.

Intel® System Studio

intel® Software

Accelerate time to market.

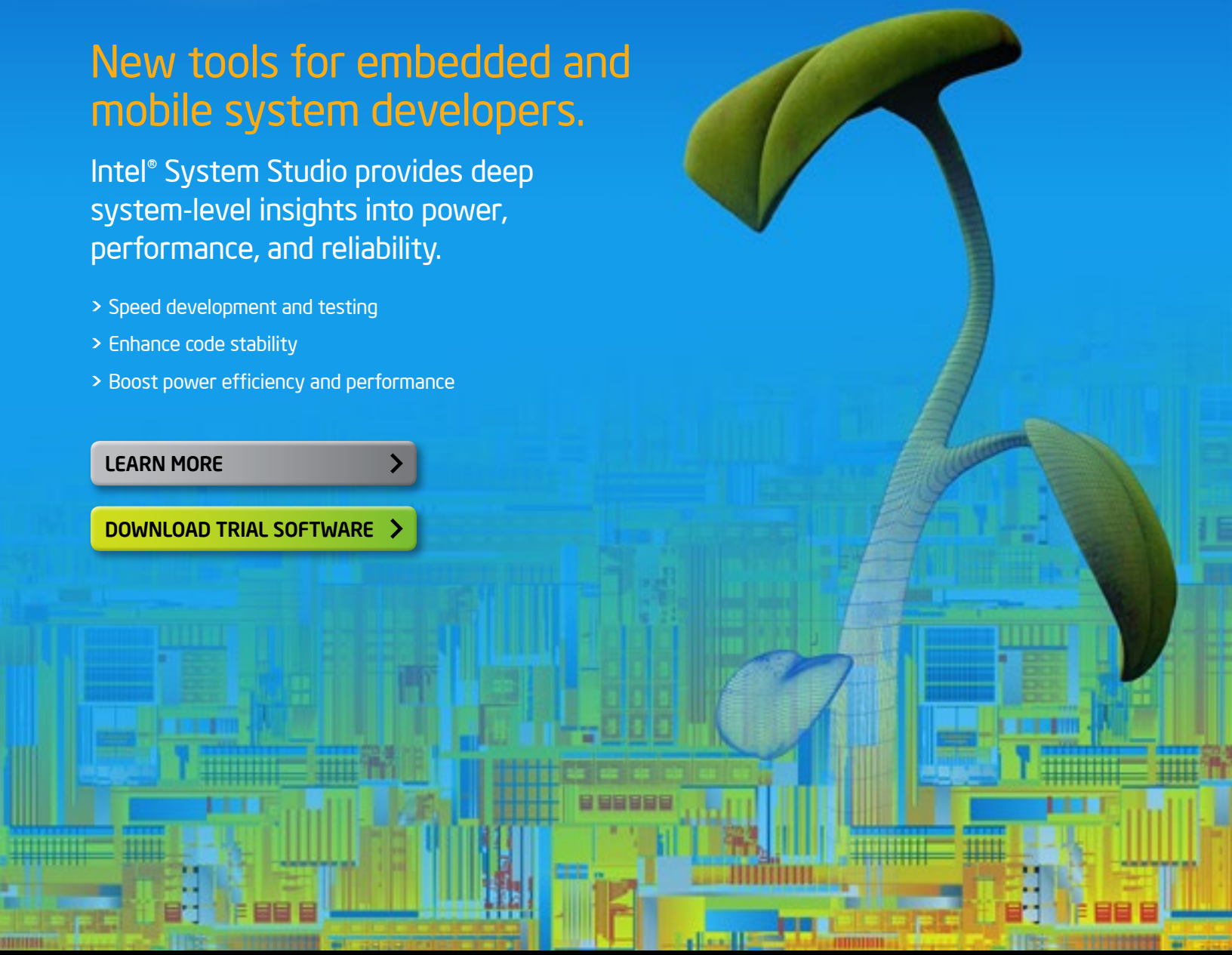
New tools for embedded and mobile system developers.

Intel® System Studio provides deep system-level insights into power, performance, and reliability.

- Speed development and testing
- Enhance code stability
- Boost power efficiency and performance

LEARN MORE >

DOWNLOAD TRIAL SOFTWARE >



©2013, Intel Corporation. All rights reserved. Intel, the Intel logo, and Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.