The Parallel Universe

Turbocharging

Open Source Python, R, and Julia-based HPC Applications

intel® Software

Issue 17 2014

CONTENTS

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).

Performance Master Class

With ever-increasing demands for performance, developers are tapping new languages and methodologies that bring HPC-level capabilities to their applications. In this issue, we explore the types of "insider" developer techniques and skillful problem solving that can enhance and extend the reach of applications.

Our feature article, *Turbocharging Open Source Python, R, and Julia-based HPC Applications*, provides guidance for performance optimizations—including exploiting vectorization and OpenMP*—on Linux* using Intel® software tools.

In *Multithreading and Task Analysis with Intel® VTune™ Amplifier XE 2013*, we look at expediting pipelining during serial and parallel stages of execution through task analysis and threading and task management.

Speed Threading Performance—Enabling Intel® TSX Using Intel® C++ Compiler explores Intel® Transactional Synchronization Extensions (Intel TSX), a transactional memory solution implemented in a hardware layer. It examines ways to both evaluate and increase efficiency through concurrent execution of critical application sections.

Tachyon Ray Tracer: Porting on the Intel® Xeon Phi™ Coprocessor walks through the steps needed to port Tachyon—an open source ray tracer and part of the Spec MPI* suite—to the Intel® Xeon® processor or the Intel® Xeon Phi™ coprocessor.

Finally, find out how to get more from your Android applications in *Building Native Android Apps With Intel® C++ Compiler*.

If you feel compelled to move forward at lightning speed, you are not alone. Whether performance optimization is a mission-critical requirement, a competitive necessity, or the ground of innovation, we hope the tools and tips presented here will speed up your knowledge base, along with your software.

James Reinders
March 2014

Turbocharging Open Source Python, R, and Julia-based HPC Applications

By Vipin Kumar E K, *Technical Consulting Engineer, Intel Software and Services*

Using Intel® Compilers and Intel® Math Kernel Library to Boost Performance

Interest is growing in the HPC community in using open source languages such as Python, R, and the new Julia. This article covers building and installing NumPy/SciPy, R, and Julia languages with Intel® compilers and Intel® Math Kernel Library (Intel® MKL), rather than using the GNU compilers and default math libraries shipped with them. Building these languages with Intel compilers on the Linux* platform enables them to exploit vectorizations, OpenMP*, and other compiler features, and significantly boost application performance with highly optimized Intel MKL on Intel® platforms.

Many of the performance optimization features available with Intel compilers and Intel MKL are not part of the default installation for these languages. One such feature is vectorization. This can take advantage of the latest SIMD vector units such as Intel® Advanced Vector Extensions (Intel® AVX1/AVX2), the 512-bit-wide SIMD available on Intel® Xeon Phi™ coprocessors, and the upcoming Intel® AVX-512 registers and instructions. With Intel® software tools, we can also exploit Intel® architectural features using interprocedural optimizations, better cache and register usage, and parallelization for maximum usage of the available cores in the CPU.

These languages also allow us the advantage of Intel Xeon Phi many-core architecture platforms by enabling the automatic offload feature in Intel MKL, to further boost performance, if such a coprocessor is present in the system. Some of the BLAS and LAPACK functions in Intel MKL including GEMM, SYMM, TRSM, TRMM, LU, QR, and Cholesky decompositions will automatically divide computation across the host CPU and the Intel Xeon Phi coprocessor. This can be enabled by setting the MKL_MIC_ENABLE=1 environment variable and it works better when problem sizes are larger.

Before proceeding to build these languages, download the latest Intel® Parallel Studio XE 2013 or Intel® Cluster Studio XE 2013 from **http://software.intel.com**.

Set up the environment variables, such as PATH, LD_LIBRARY_PATH, etc. for Intel® C/C++ and Fortran Compilers and Intel MKL, and include files by sourcing the compilervars.sh from the Intel® tools installation folder. The default installation location for the latest Intel compiler and Intel MKL is /opt/intel/composer_xe_2013_sp1 on a Linux platform.

To set up the environment for the 64-bit Intel® architecture (IA) run the command as:

```
$source /opt/intel/composer_xe_2013_sp1/compilervars.sh intel64
```

Or, to set up the environment for the 32-bit IA e run the command as:

```
$source /opt/intel/composer_xe_2013_sp1/compilervars.sh ia32
```

Installing Python with Intel MKL and Intel Compilers

Python is a modern, open source, interpreted object-oriented language that is easily portable and enables faster program development. Since it is easy to write and has support of other libraries like NumPy and SciPy, it's very popular and is now one of the most widely used languages after C/C++ and Fortran in the scientific computing community. The beauty of Python is that it can be easily integrated with software written in other languages and thus serves as a control language to drive existing programs. Python can act as a "glue" language to combine different systems together.

NumPy/SciPy Installation with Intel MKL and Compiler Optimizations

These are fundamental libraries that provide objects and routines for scientific or high performance computing. You can accelerate the BLAS and LAPACK performance of these libraries using Intel MKL integration. From our tests on an Intel® Core™ i5 machine with four threads we could see around 200% performance improvement in matrix multiplication with Intel MKL over the open source ATLAS library with four threads.

NumPy automatically maps operations on vectors and matrices to the BLAS and LAPACK functions wherever possible. Since Intel MKL supports these de facto interfaces, NumPy can benefit from Intel MKL optimizations through simple modifications to the NumPy scripts.

NumPy (**http://numpy.scipy.org**) is the fundamental package required for scientific computing with Python and consists of:

- Powerful N-dimensional array object
- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transforms, and random number capabilities.

Besides its obvious scientific uses, NumPy can also be used as an efficient multidimensional container of generic data.

SciPy (**http://www.scipy.org**) includes modules for statistics, optimization, integration, linear algebra, Fourier transforms, signal and image processing, ODE solvers, and more. The SciPy library depends on NumPy, which provides convenient and fast N-dimensional array manipulation. The SciPy library is built to work with NumPy arrays, and provides many user-friendly and efficient numerical routines, such as routines for numerical integration and optimization for Python users.

You can download the NumPy and SciPy source code from **http://www.scipy.org/Download**.

When the latest versions of NumPy and SciPy tar balls are downloaded, extract them to create their source directories. Then follow these steps:

1. Change directory to `numpy-x.x.x`, the numpy root folder and create a site.cfg from the existing one.
2. Now, edit site.cfg as follows:
   a. Add the following lines to `site.cfg` in your top level NumPy directory to use Intel MKL, if you are building on a 64-bit Intel platform, assuming the default path for the Intel MKL installation from the Intel® Parallel Studio XE 2013 or Intel® Composer XE 2013 versions:

```
[mkl]
library_dirs = /opt/intel/composer_xe_2013_sp1/mkl/lib/intel64
include_dirs = /opt/intel/mkl/include
mkl_libs = mkl_rt
lapack_libs =
```

   b. If you are building NumPy for 32-bit, add as follows:

```
[mkl]
library_dirs = /opt/intel/composer_xe_2013_sp1/mkl/lib/ia32
include_dirs = /opt/intel/mkl/include
mkl_libs = mkl_rt
lapack_libs =
```

3. Modify cc_exe in numpy/distutils/intelccompiler.py to be something like:

```
self.cc_exe = 'icc -O3 –xavx –ipo -g -fPIC -fp-model strict -fomit-
frame-pointer -openmp –DMKL_ILP64'
```

Here we use `-O3` optimizations for speed and it enables more aggressive loop transformations such as Fusion, Block-Unroll-and-Jam, and collapsing IF statements. We use `-openmp` option for OpenMP threading and the `-xavx` option tells the compiler to generate Intel® AVX instructions. You may also set the `–xHost option` in case you are not aware of the processor architecture, and it will use the highest instruction set available on the compilation host processor. If you are using the ILP64 interface, add the `-DMKL_ILP64` compiler flag.

Run icc --help for more information on processor-specific options, and refer to the Intel compiler documentation for more details on the various compiler flags.

Modify the Fortran compiler configuration in numpy-x.x.x/numpy/distutil/fcompiler/intel.py to use the following compiler options for the Intel Fortran Compiler.

On 32- and 64-bit use the following options:

```
ifort -xhost -openmp -fp-model strict –i8 -fPIC
```

Compile and install NumPy with the Intel compiler: (on 32-bit platforms replace "`intelem`" with "`intel`") by running the following command.

```
$python setup.py config --compiler=intelem build_clib --
compiler=intelem build_ext --compiler=intelem install
```

Compile and install SciPy with the Intel compiler (on 32-bit platforms replace "`intelem`" with "`intel`") as shown below:

```
$python setup.py config --compiler=intelem --fcompiler=intelem
build_clib --compiler=intelem --fcompiler=intelem build_ext --
compiler=intelem --fcompiler=intelem install
```

We have to set up the library paths for Intel MKL and Intel compiler by exporting `LD_LIBRARY_PATH` environment variable as shown below:

On 64-bit platforms :

```
$export
LD_LIBRARY_PATH=/opt/intel/composer_xe_2013_sp1/mkl/lib/intel64:/opt/
intel/composer_xe_2013_sp1/lib/intel64:$LD_LIBRARY_PATH
```

On 32-bit platforms:

```
$export
LD_LIBRARY_PATH=/opt/intel/composer_xe_2013_sp1/mkl/lib/ia32:/opt/intel
/composer_xe_2013_sp1/lib/ia32:$LD_LIBRARY_PATH
```

The `LD_LIBRARY_PATH` variable may cause a problem if you have installed Intel MKL and Intel® Composer XE in directories other than the standard ones. A solution we have found that always works is to build Python, NumPy, and SciPy inside an environment where you've set the `LD_RUN_PATH` variable. For example, on a 32-bit platform, you can set the path as follows:

```
$export LD_RUN_PATH=/opt/intel/composer_xe_2013_sp1/lib/ia32:/opt/
intel/composer_xe_2013_sp1/mkl/lib/ia32
```

Note: We recommend using arrays with the default 'C' ordering style, which is row-major, rather than the Fortran Style, which is column-major, because NumPy uses CBLAS, and also because you get better performance this way.

Installing R with Intel Compiler and Intel MKL Software Tools

R is a high-level programming language with which one can perform complex calculations, frequently used in statistical computing, computational biology, and big data analytics. R has high-level functions to operate on matrices, perform numerical integration, advanced statistics, etc., which can be utilized in many scientific computing applications.

This section will show you how to use the BLAS and LAPACK libraries within Intel MKL to improve the performance of R. If you want to use other Intel MKL functions, you can do so by using wrappers. In our runs of a standard R benchmark, we could get nearly 20 times speedup for Cholesky decomposition of 3Kx3X matrices on an Intel® Core™ i7 machine with four cores, SMT on, with Intel MKL and Intel compiler optimizations compared to the default R installation done using GNU tools.

Use the –with-blas option during the configuration step to configure R to use the Intel MKL BLAS and LAPACK libraries.

To take advantage of Intel compiler optimizations, initially, we set the environment variables for Intel linker and library archive tools from the Intel compiler by exporting the AR and LD variables that will be used in the configure process:

```
$export AR="xiar"

$export LD="xild"

Set the Intel MKL and OpenMP library paths

$MKL_LIB_PATH='/opt/intel/composer_xe_2013_sp1/mkl/lib/intel64'

$OMP_LIB_PATH='/opt/intel/lib/intel64'

$export LD_LIBRARY_PATH=${MKL_LIB_PATH}:${OMP_LIB_PATH}

$MKL=" -L${MKL_LIB_PATH} -L${OMP_LIB_PATH} -lmkl_intel_lp64 -
lmkl_intel_thread -lmkl_core  -liomp5 -lpthread"
```

The **config.site** file, located in the root folder of R.x.x.x source, has to be modified to use the Intel compiler and its optimization features:

1. Edit config.site by making changes for C/C++ and Fortran compilers and its options-related lines as shown below:

```
CC='icc –std=c99'
CFLAGS='-O3 –ipo –xavx –openmp'
F77='ifort'
FFLAGS='-O3 –ipo –xavx –openmp'
CXX='icpc'
CXXFLAGS='-O3 –ipo –xavx –openmp'
$./configure --with-blas="$MKL" --with-lapack
```

   The default number of threads will equal the number of physical cores on the system, but can be controlled by setting OMP_NUM_THREADS or MKL_NUM_THREADS.

2. Check the **config.log** to see if Intel MKL was working during the configuration test.

```
configure:29075: checking for dgemm_ in -lmkl_intel_lp64 -
lmkl_intel_thread -lmkl_core -liomp5 -lpthread

configure:29096: icc -std=c99 -o conftest -O3 -ipo -openmp -xHost  -
I/usr/local/include  -L/usr/local/lib64 conftest.c -lmkl_intel_lp64 -
lmkl_intel_thread -lmkl_core -liomp5 -lpthread  -lifport -lifcoremt -
limf -lsvml -lm -lipgo -liomp5 -lirc -lpthread -lirc_s -ldl -lrt -ldl -lm  >&5

conftest.c(210): warning #266: function "dgemm_" declared implicitly
  dgemm_()
  ^

configure:29096: $? = 0
configure:29103: result: yes
configure:29620: checking whether double complex BLAS can be used
configure:29691: result: yes
configure:29711: checking whether the BLAS is complete
```

3. Now, run **make** to build and install R with the Intel compiler and Intel MKL:

```
$make && make install
```

4. If Intel MKL library was dynamically linked in R, use the ldd command to verify that Intel MKL and compiler libraries are linked to R. You should get outputs as seen below on successful installation of R with Intel MKL using the Intel compiler.

```
$ ldd ./bin/exec/R
        linux-vdso.so.1 =>  (0x00007fff2a1ff000)
        libmkl_intel_lp64.so =>
/opt/intel/composer_xe_2013_sp1.0.080/mkl/lib/intel64/libmkl_intel_lp64.so
(0x00007f7d89bf9000)
        libmkl_intel_thread.so =>
/opt/intel/composer_xe_2013_sp1.0.080/mkl/lib/intel64/libmkl_intel_thread.so
(0x00007f7d88b5e000)
        libmkl_core.so =>
/opt/intel/composer_xe_2013_sp1.0.080/mkl/lib/intel64/libmkl_core.so
(0x00007f7d87630000)
        libiomp5.so =>
/opt/intel/composer_xe_2013_sp1.0.080/compiler/lib/intel64/libiomp5.so
(0x00007f7d87317000)
        libpthread.so.0 => /lib64/libpthread.so.0
(0x00000036ec600000)
        libifport.so.5 =>
/opt/intel/composer_xe_2013_sp1.0.080/compiler/lib/intel64/libifport.so.5
(0x00007f7d870c8000)
        libifcoremt.so.5 =>
/opt/intel/composer_xe_2013_sp1.0.080/compiler/lib/intel64/libifcoremt.so.5
(0x00007f7d86d59000)
        libimf.so =>
/opt/intel/composer_xe_2013_sp1.0.080/compiler/lib/intel64/libimf.so
(0x00007f7d8689a000)
        libsvml.so =>
/opt/intel/composer_xe_2013_sp1.0.080/compiler/lib/intel64/libsvml.so
(0x00007f7d85e3f000)

. . . . . .
```

When asked to provide some indication of the performance improvements that installing R can provide, I ran R-benchmark-25.R (from **R benchmarks** site) on a system with a 32-core Intel® Core™ i7-975 processor Extreme Edition (8MB LLC, 3.33GHz) with 6GB of RAM and running RHEL 6 x86_64. I first installed R-2.15.3 and ran the benchmark without Intel MKL.

We saw significant improvement in performance when we ran the standard R benchmark R-2.15.3 on an Intel Core i7 3770K quad-core machine (8MB LLC, 3.50 GHz) with 16GB RAM running on Ubuntu 13.1 with and without Intel MKL. In the results shown below for some of the tests, the total time is in seconds for all the tests in the benchmark.

	GNU	**Intel® MKL & Intel® Compilers**
2800x2800 cross-product matrix (b = a' * a)	9.4276667	0.2966667
Cholesky decomposition of a 3000x3000 matrix	3.7346666	0.1713333
Total time for all 15 tests	29.1816667	5.4736667

Installing Julia with Intel MKL

Finally, we will see how to build and install Julia language, which is gaining greater momentum in the HPC community. Julia is a high-level, dynamic programming language addressing high-performance numerical and scientific computing. But, it's also effective for general purpose programming. Though Julia is also an interpreted language, it is designed to provide high performance on par with the compiled languages. The implementation design of Julia includes multiple dispatch as the core language platform, exposing a sophisticated type system, dynamic type inference to generate fast code, and aggressive specialization of generated code for types encountered at runtime. In the level 3 BLAS performance test runs, we saw around 1.5x to 2x improvement with Intel MKL over the default installation on an Intel Core i7, four-core machine.

Julia's source code can be downloaded from the GIT hub (**https://github.com/JuliaLang/julia/**).

To set up the Intel MKL environment, let us build for 64-bit with 8 bytes integer support (ILP64).

Run the mklvars.sh script to set the Intel MKL environment variables passing arguments for Intel 64 and ilp64 as shown below:

```
$source /opt/intel/mkl/bin/mklvars.sh intel64 ilp64
```

For 32-bit architecture, use ia32 as the argument to set the environment variable:

```
$source /opt/intel/mkl/bin/mklvars.sh ia32
```

In our case, to use 64-bit and ILP64 architecture, we also need to set the Intel MKL Interface layer variable as:

```
$export MKL_INTERFACE_LAYER=ILP64
```

Next, we have to edit Make.inc file located in the Julia root folder or we can create a Make.user file to include the changes mentioned here.

Change `USE_MKL = 0` variable to `USE_MKL = 1`

Add `USE_INTEL_IMF=1`

Once you have set the Intel MKL environment and modified the Make.inc file to use Intel MKL, building Julia with Intel MKL support is straightforward and can be done by running the following command:

```
$make install
```

You can also build Julia with Intel MKL support by passing arguments to make:

```
$make USE_MKL=1 install
```

To utilize Intel compilers and their optimizations features, build Julia with the following command:

```
$make -j 4 CC=icc CFLAGS="-O3 -xavx" CXX=icpc CXXFLAGS="-O3 –
xavx"  FC=ifort FFLAGS="-O3 –xavx" USE_MKL=1 USE_INTEL_IMF=1
```

Note that in this case, we are running the build in four threads by passing –j 4. You may change this, depending on the number of threads available in your system.

To rebuild a prebuilt Julia source installed with Intel MKL support, delete the OpenBLAS, ARPACK, and SuiteSparse dependencies from the /deps folder of Julia and run the command as follows:

```
$make cleanall testall
```

You may run the various performance tests provided in the Julia/test/perf /blas and Julia/test/perf/lapack folder to see the performance benefits of Julia with Intel software tools.

Summary

Python, R, and Julia are popular languages that provide faster programming and readable code. Developers in the HPC, analytics, and scientific computing domains can take advantage of the latest Intel architecture features by enabling these languages with Intel software tools. Intel compiler features allow you to exploit SIMD/AVX vectors and instructions by using vectorization flags and OpenMP support to utilize all the available cores in the system. The compilers offer multifile optimization using interprocedural optimization and many other performance optimization features. The hand-tuned Intel MKL library is a high performing, industry-standard math library optimized for the latest Intel architectures by using better algorithms, vectorization, and OpenMP threading, and also provides reproducible results when its conditional numerical reproducibility feature is used. With a few simple steps Python, R, and Julia can be built and installed with Intel MKL support using Intel compilers that give out-of-the-box performance improvement with these languages. If your systems have Intel Xeon Phi coprocessors, automatically enabled functions in MKL will add further performance improvement by dividing computation between the host CPU and the coprocessor. ●

With a few simple steps Python, R, and Julia can be built and installed with Intel® Math Kernel Library support using Intel® compilers that give out-of-the-box performance improvement.

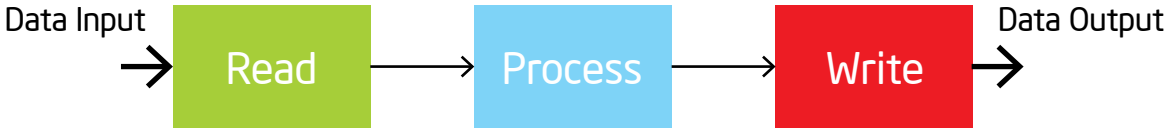
Multithreading and Task Analysis with Intel® VTune™ Amplifier XE 2013

By Vladimir Tsymbal, *Technical Consulting Engineer, Intel Software and Services*

Developers looking for performance opportunities may consider a class of applications that can be organized to exploit pipelining, typically through a combination of intrinsically serial and parallel stages of execution. Managing threads in such circumstances can be tricky. Intel® Threading Building Blocks (Intel® TBB) may help to save time and effort in the design and support of parallel algorithms such as pipeline building by taking care of thread management work for better parallelism. With this library, a programmer can avoid the drudgery of mapping execution stages to threads and taking care of the work balance between them. A problem just needs to be represented as a set of execution tasks to be completed, and Intel TBB will take care of dynamic distribution of the tasks to hardware threads available in a current system. Task management can be nontrivial depending on the complexity of an application. Intel® VTune™ Amplifier XE embedded task analysis can help a programmer to organize user tasks by providing a convenient visual instrument for problem investigation, saving additional development time. Here, we will consider a simplified version of a real problem, going step by step through parallelization, pipeline building, and task analysis for performance improvement.

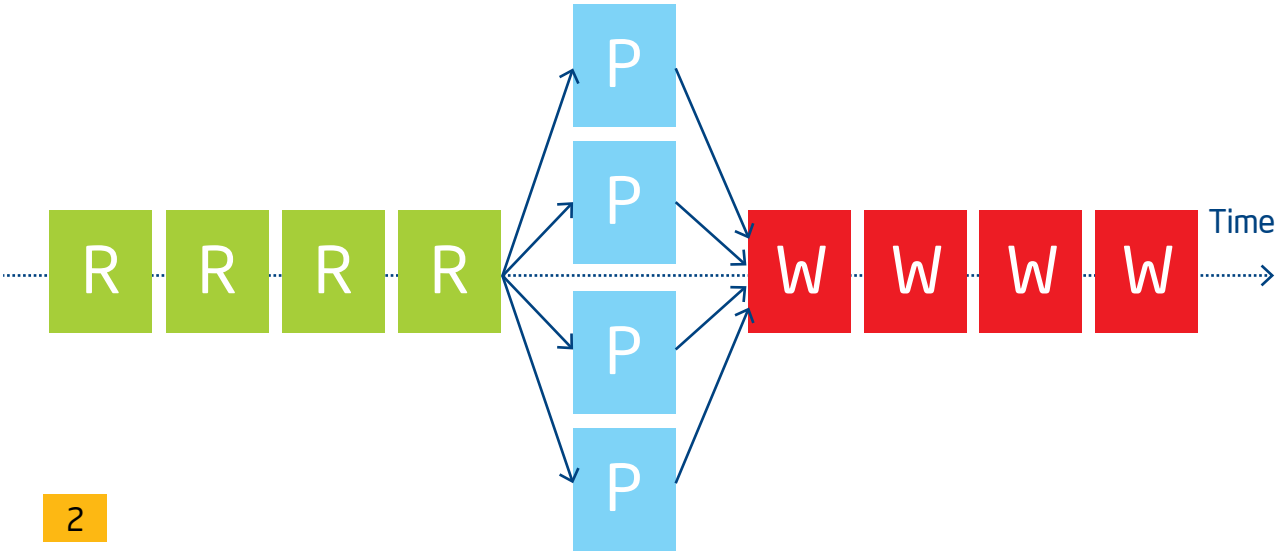
It's well known that an execution task can be parallelized using a decomposition intended to distribute a given amount of work between a set of execution units. Depending on specifics of the execution problem or its input data structure, you can apply either data decomposition or task decomposition. Simply put, *data decomposition* for parallelization can be described as breaking down data arrays into chunks to be executed in the same processing unit in parallel. *Task decomposition* for parallel execution can be considered as several different processing units that work in parallel and handle the same data in some way. The common goal for decomposition of an execution problem is to provide all available processing units with work, and to make sure they are busy the entire time that any part of the problem is still being solved. Failure to appropriately distribute tasks among processing units leads to inefficient computation and makes execution time suboptimal. Thus, analysis of tasks execution is crucial for performance optimization.

Consider a simplified case of what might be found in any real task case where contiguous input data are being acquired from a data source, processed in a functional unit, and then stored to a sink (**Figure 1**). This is a very common data flow case that can be found in applications like codecs, filters, or communication layers.



1

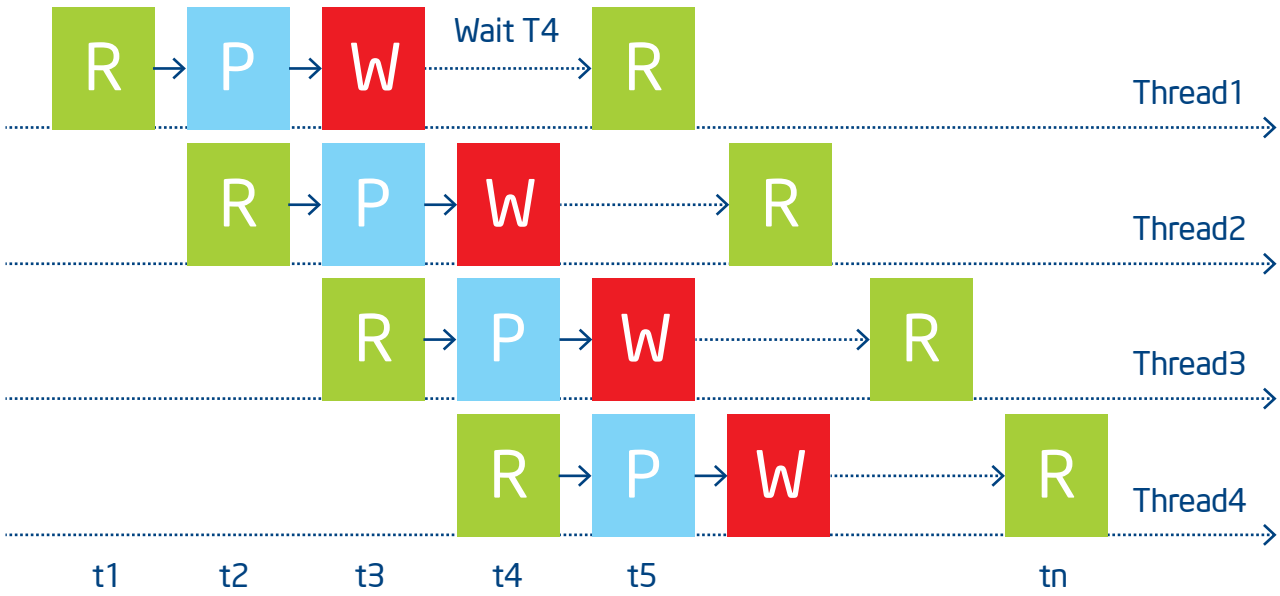
For the sake of simplicity, we will not be considering dependency between data chunks in the incoming data. Usually, there is a dependency, as in multimedia decoders when picture blocks are dependent on their neighboring pixels, and frames depend on subsequent ones. But even in those cases, we can identify blocks of data that can be considered as independent and processed in parallel. Thus, we apply a data decomposition model and distribute the independent chunks of data between the processing units which handle the data in parallel, reducing execution time. Once data are processed, they go to the unit that writes to the next stage (**Figure 2**).



2

If your data sources are devices, such as hard disks, streaming devices, or the network, the process of reading and writing data is intrinsically serial. And, even though you have a number of hardware execution threads in your system, the only stage for which you can decrease execution time is the processing stage: According to Amdahl's law, your time optimization is limited by the serial part of the execution. The reading and writing stages are serial parts, which leave the rest of the hardware threads waiting in a sleep mode while they are executing.

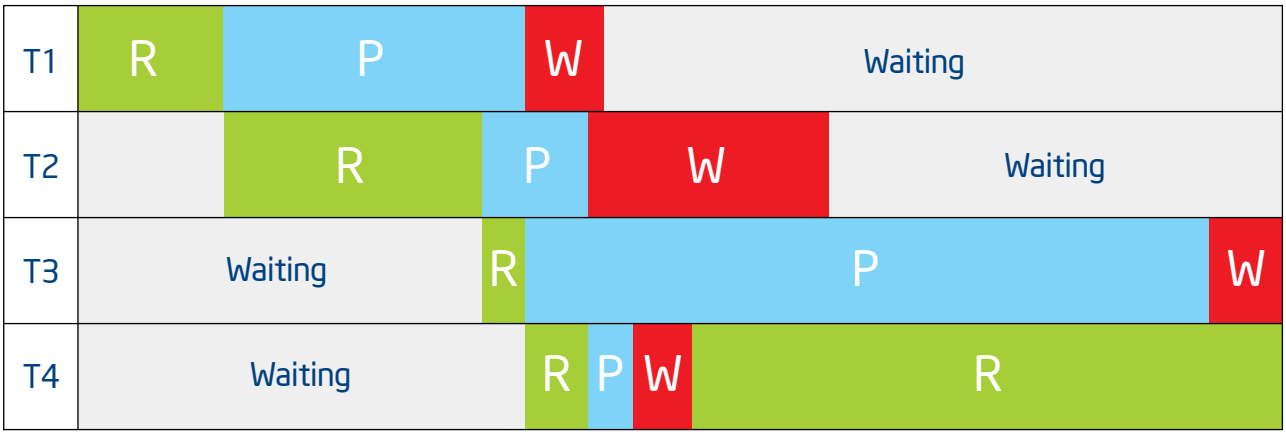
To overcome this limitation, redesign your system with the goal to make your hardware threads busy all the time when the input data blocks are independent and the order of data out doesn't matter, or it can be reordered later. You build a simple pipeline that consists of the read/process/write stages executed by each thread (**Figure 3**). The read and write stages are still sequential, but distributed between the executing treads. Note that no thread can start a Read or Write stage while any other thread is still executing that stage, but two different threads can execute read and write stages simultaneously (one each).



3

By restructuring your design this way you introduce parallel task decomposition: threads are executing different tasks (functions) at the same time. You are using data decomposition in this new design, as well. Together, these methods help keep hardware threads busy all the time, assuming there are no delays in data reading/writing. There is sufficient processing work to cover the I/O overlaps, and processing units dispose available CPUs for execution.

In real-world systems and applications, you can't be guaranteed that read-write operations are stable, CPUs are not all busy in a certain moment, or that data is consistent and homogeneous. Input data can be delayed from their supplier or insufficient to fill up an input buffer. CPUs may be busy with other tasks of higher priority or share the resources with other processes. Some data may require much more CPU time to handle. For example, codecs need to do drastically more work on detailed and dense scenes than with sparse or background scenes. Write operations to the I/O subsystem are usually not a problem as it can be buffered to memory, but it only works until your algorithm needs some written data as input. Given all the issues above, you may end up with as ineffective pipeline execution as the initial parallel implementation (**Figure 4**).



4

To avoid problems induced by work imbalance, try to dynamically distribute the workload between threads. This requires monitoring of non-busy threads and managing tasks so that they can be broken down into smaller tasks and delegated to the available hardware threads for execution. Implementation of the threads and tasks management infrastructure can be fairly complicated. You can avoid the burden of developing and supporting such infrastructure by using threading libraries like Intel TBB. In regards to this example, the Intel TBB library contains an embedded algorithm that is called a "pipeline class." It's a fairly good fit for the problem we're discussing.

We are not going to dive deeply into details of the pipeline implementation in Intel TBB here (you can take a closer look at the documentation available or the library source code at: **http://threadingbuildingblocks.org/**). Next, we'll look at a quick explanation of how to create a pipeline for parallel execution of the tasks. The whole project with source code can be found in the Intel TBB product installation directory or on the website.

Basically, the pipeline is the application of a series of executing stages to a stream of items. The executing stages can be the tasks that we need to execute on the data stream. In the Intel TBB library, those stages can be defined as instances of class *filter*. So you build your pipeline as a sequence of filters. Some stages (like processing) can be executed simultaneously or in parallel on different items, so you define these stages as *parallel* filter class. Other stages like read or write must be executed serially and in order, so you define them as *serial_in_order* filter class. The library contains abstract classes for such filters, and you just need to derive your own classes from them. For example (for the sake of simplicity, not all required definitions are provided in these code snippets):

```
class MyReadFilter: public tbb::filter {
    FILE* input_file;
    DataItem* next_item;
    /*override*/ void* operator()(void*);
public:
    MyReadFilter( FILE* in );
};
```

```
MyReadFilter:: MyReadFilter( FILE* in ) :
    filter(serial_in_order),
    input_file(in),
    next_item(DataItem*)
{
}
```

```
class MyWriteFilter: public tbb::filter {
    FILE* output_file;
    /*override*/ void* operator()(void*);
public:
    MyWriteFilter( FILE* out );
};
```

```
MyWriteFilter:: MyWriteFilter( FILE* out ) :
    filter(serial_in_order),
    output_file(out),
{
}
```

Because the data is being read from a file, you need to define a private member for file pointer in the class. Similarly, you define a MyWriteFilter class for the write stage and assign an output file pointer in an appropriate class member. The classes are responsible for allocating memory and passing data items through the pipeline. Main execution for a stage is done in the *operator()* method defined in the base class. Simply override the *operator()* methods and implement reading data items from a file to a data container, and writing data from the container to an output file accordingly.

```
void* MyReadFilter::operator()(void*) {
// ALLOCATEMEMORY
// READ A DATA ITEM FROM FILE
// PUT DATA ITEM TO CONTAINER
}
```

```
void* MyWriteFilter::operator()(void*) {
// GET DATA ITEM FROM CONTAINER
// WRITE THE DATA ITEM TO FILE
// DEALLOCATE MEMORY
}
```

Your process stage can be executed in parallel, so it needs to be defined as a *parallel* filter class, and the *operator()* method should process the streaming data according to your algorithm.

```
class MyProcessFilter: public tbb::filter {
public:
    MyProcessFilter();
    /*override*/void* operator()( void* item );
};
MyProcessFilter:: MyProcessFilter() :
    tbb::filter(parallel)
{}
```

```
void* MyProcessFilter::operator()( void*
item ) {
// FIND A CURRENT DATA ITEM IN CONTAINER
// PROCESS THE ITEM
}
```

The final step is to construct the whole pipeline: create the filter class objects and the pipeline class object, and link the stages. After the pipeline is built, you call a *run()* method of the pipeline class, specifying a maximum number of tokens. The number of tokens in this case is the number of data items to be handled simultaneously by the pipeline. Selection of this number will be left for a later discussion. We follow the recommendation from the Intel TBB guidelines and choose the number equal to twice the number of available threads. This will ensure that each stage has a data item to handle, and that there won't be a data item queue growing because the first stage could handle data much faster than the next one.

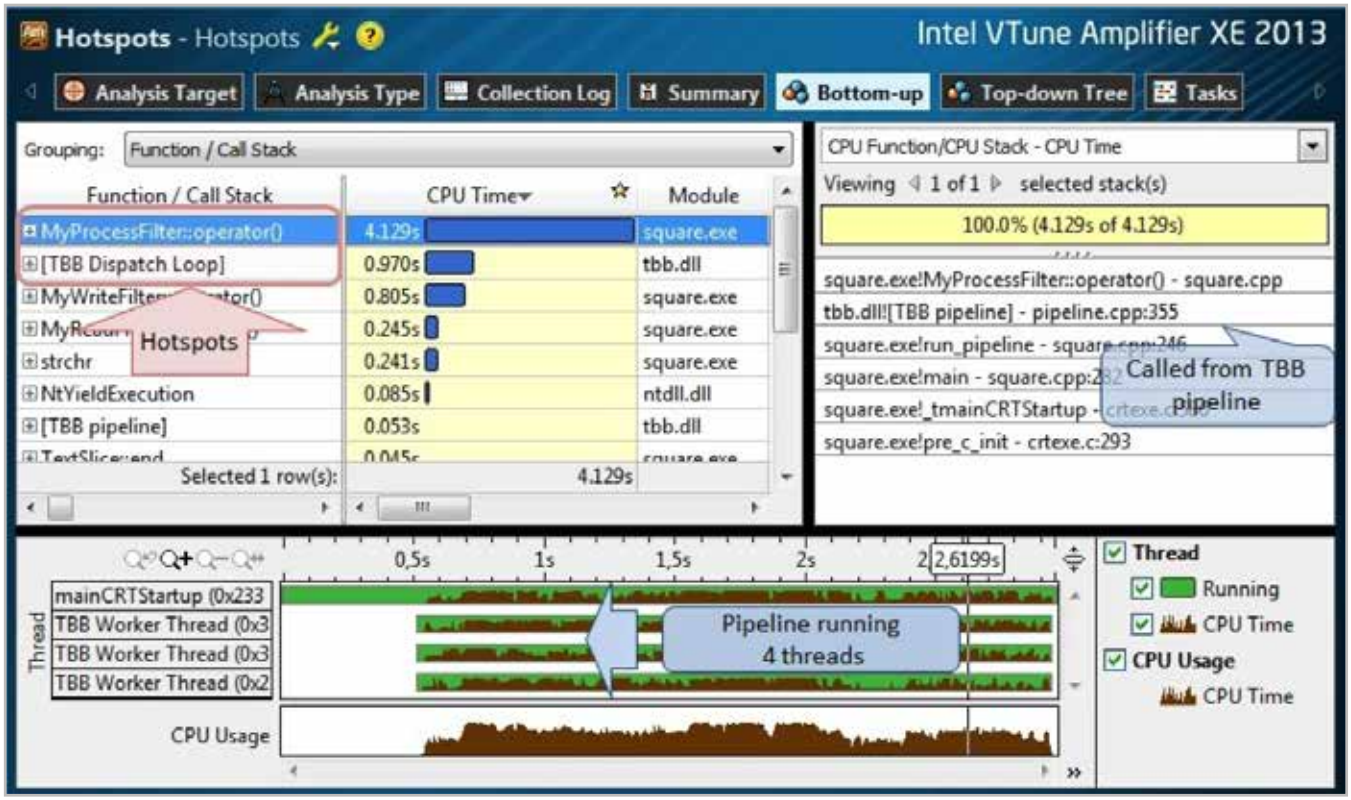
```
tbb::pipeline pipeline;
 MyReadFilter r_filter( input_file );
 pipeline.add_filter( r_filter );

 MyProcessFilter p_filter;
 pipeline.add_filter(p_filter );

 MyWriteFilter w_filter( output_file );
 pipeline.add_filter( w_filter );

 pipeline.run(2*nthreads)
```

Now, you have three pipelined tasks which will be handled by the Intel TBB library in a way that maximizes utilization of CPU resources. The process stage will be parallelized and, along with the rest of the stages, can be dynamically reassigned to different hardware threads depending on their availability. As you can see, you don't need to care about thread management; the only thing requiring your attention is properly creating and linking tasks in the pipeline.

There are a few questions that may be unresolved. How effective is the constructed pipeline? How are those stages being executed in the pipeline—is there any library overhead or execution gap between stages?

To answer these questions, we need to analyze the pipeline execution. Profiling the process of data handling with Intel VTune Amplifier XE should help us estimate the effectiveness of utilizing CPU cores, understanding details of task management in the Intel TBB library, and identifying possible pitfalls that might prevent better parallelism, and thus, performance.

Following a general profiling scenario, we start analysis with hotspot profiling in Intel VTune Amplifier XE. We analyze a sample program that reads a file containing decimal integers [Read] in text format, changes each to its square [Process], and finally writes the data [Write] into a new file.

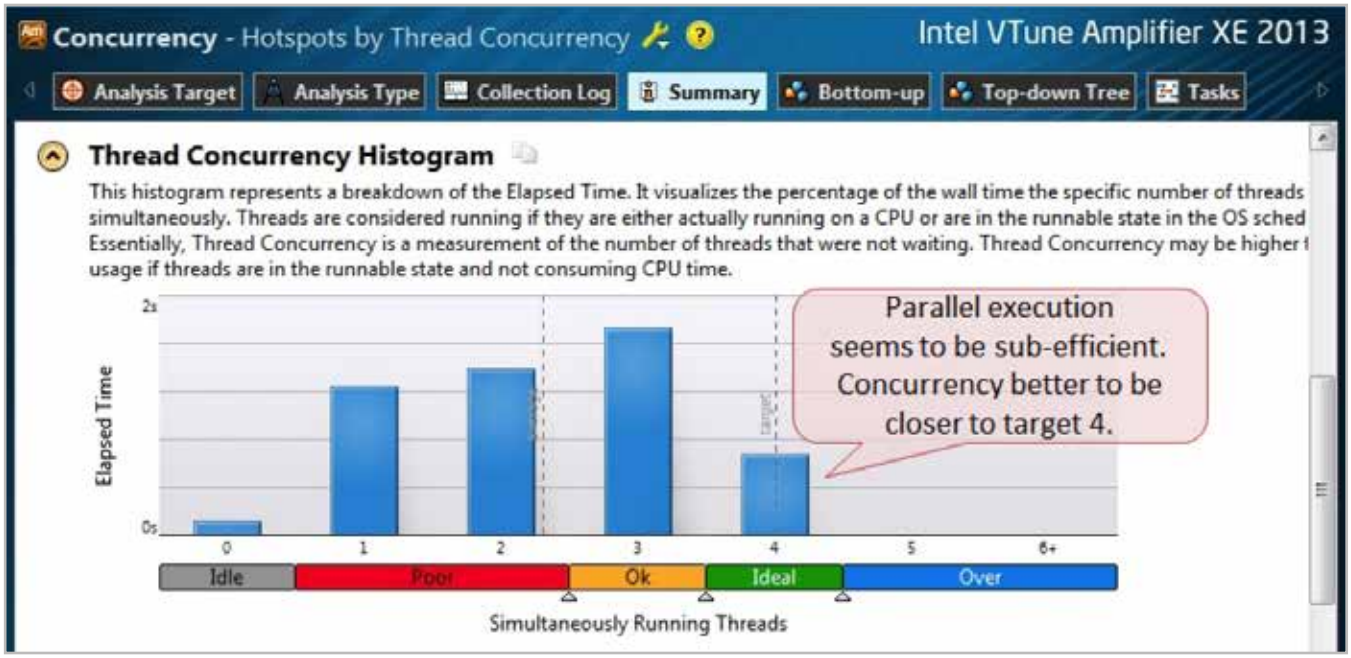


5

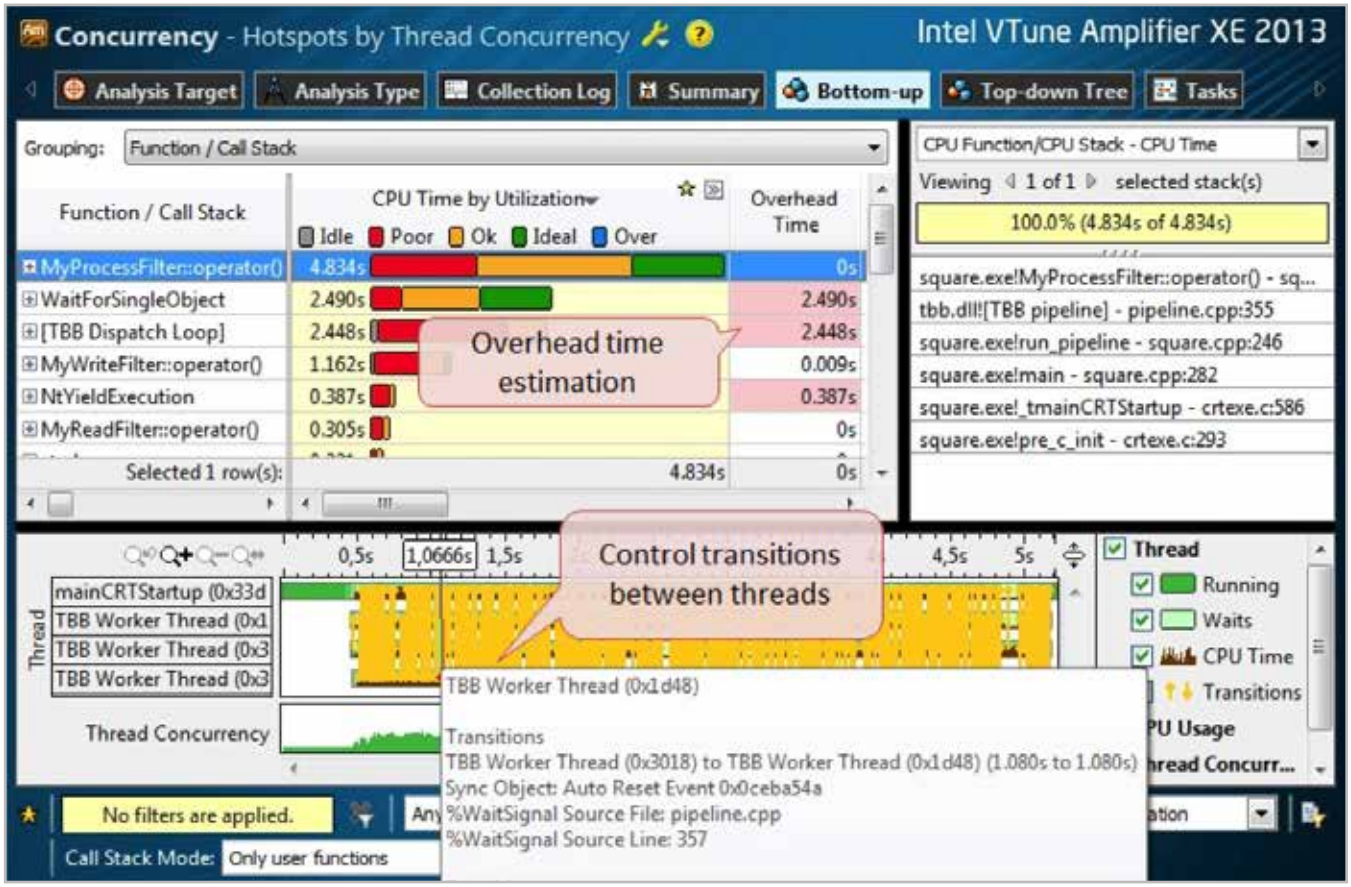
The result of the hotspot analysis is what we'd expect (**Figure 5**). The test was executed in four threads available on the system. The *MyProcessClass::operator()* method called from the Intel TBB pipeline is the hottest function, as it performs a text-to-integer conversion, calculates the square, and converts the result back to text. What we observe is that the (Intel) [TBB Dispatch Loop] in the hotspot list may be the Intel TBB task dispatcher overhead exposed during execution. Let's continue with a concurrency analysis and determine the parallel execution efficiency of the application.

The results of the concurrency analysis expose inefficiency in this parallel execution, as most of the time the application was running less than the optimal 4 threads simultaneously (**Figure 6**). (Note: the blue concurrency level bars show how much time an application spent running given threads in parallel. Ideally, we'd like to see a single bar at concurrency level 4, which means all 4 threads running all the application execution time on a 4-core CPU.)

A quick look into the bottom-up view gives us a picture of excessive synchronization overhead in the Intel TBB pipeline (**Figure 7**). You can hover over the yellow transition lines to find source of this excessive synchronization.



6



7

At this point, it might be tempting to give up searching for a reason for the parallelization inefficiency and blame Intel TBB pipeline implementation. However, our investigation would be incomplete without checking task execution. A trace of tasks may help to find more information on task sequences and timing. A custom instrumentation of tasks in the source code seems to be the easiest way to collect the traces. The only problem might be the burden of trace handling and representation for analysis. Intel VTune Amplifier XE provides you with a powerful task analysis instrument—which helps to collect the traces and represent them graphically for quicker investigation.

In order to employ task analysis, you need to instrument your task execution code using a specific task API that is available in the tool. Here are the steps you would likely follow:

1. Include the API header in your source file.

```
#include "ittnotify.h"
```

2. Define a task domain in your program. It's useful to distinguish between different threading domains used in your project.

```
__itt_domain* domain = __itt_domain_create("PipelineTaskDomain");
```

3. Define your task handles to match the stages.

```
__itt_string_handle* hFileReadSubtask = __itt_string_handle_create("Read");
__itt_string_handle* hFileWriteSubtask = __itt_string_handle_create("Write");
__itt_string_handle* hDoSquareSubtask = __itt_string_handle_create("Do Square");
```

4. Wrap each stage's execution source code with the *__itt_task_begin* and *__itt_task_end* instrumentation calls. For example, the read and write stages can be done as follows:

```
void* MyReadFilter::operator()(void*) {
__itt_task_begin(domain, __itt_null, __itt_null, hFileReadSubtask);
// ALLOCATE MEMORY
// READ A DATA ITEM FROM FILE
// PUT DATA ITEM TO CONTAINER
__itt_task_end(domain);
}
```

```
void* MyWriteFilter::operator()(void*) {
__itt_task_begin(domain, __itt_null, __itt_null, hFileWriteSubtask);
// GET DATA ITEM FROM CONTAINER
// WRITE THE DATA ITEM TO FILE
// DEALLOCATE MEMORY
__itt_task_end(domain);
}
```

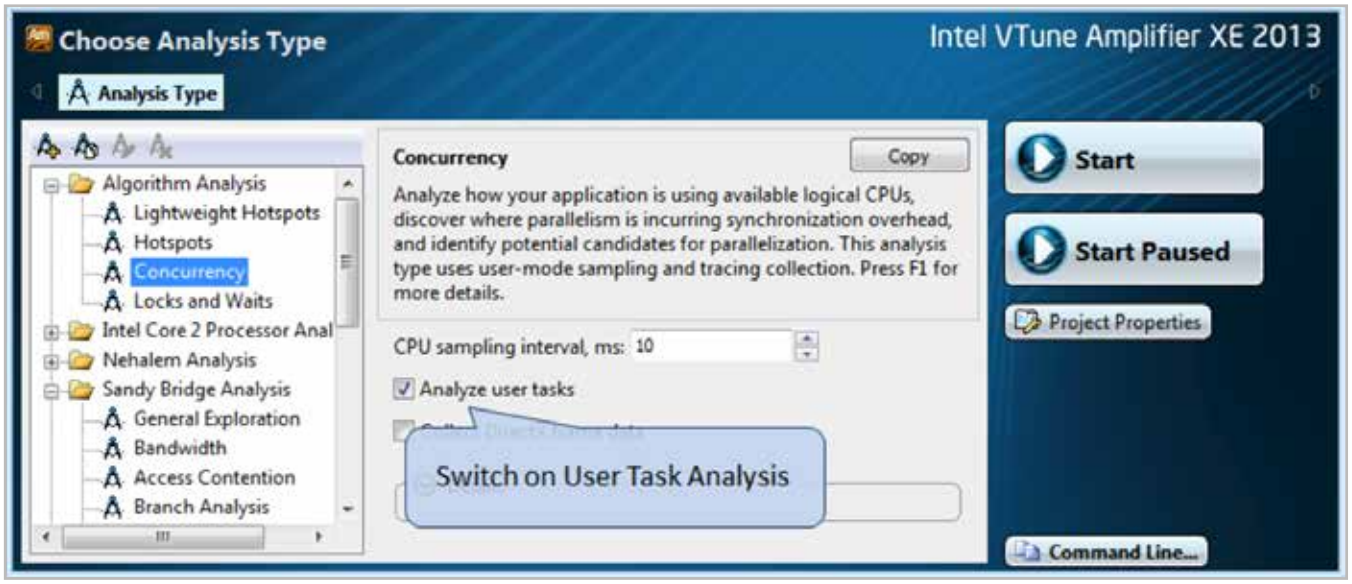
This also applies to the process stage (more information on the API calls can be found in the product documentation).

```
void* MyProcessFilter::operator()( void* item ) {
__itt_task_begin(domain, __itt_null, __itt_null, hDoSquareSubtask);
// FIND A CURRENT DATA ITEM IN CONTAINER
// PROCESS THE ITEM
__itt_task_end(domain);
}
```

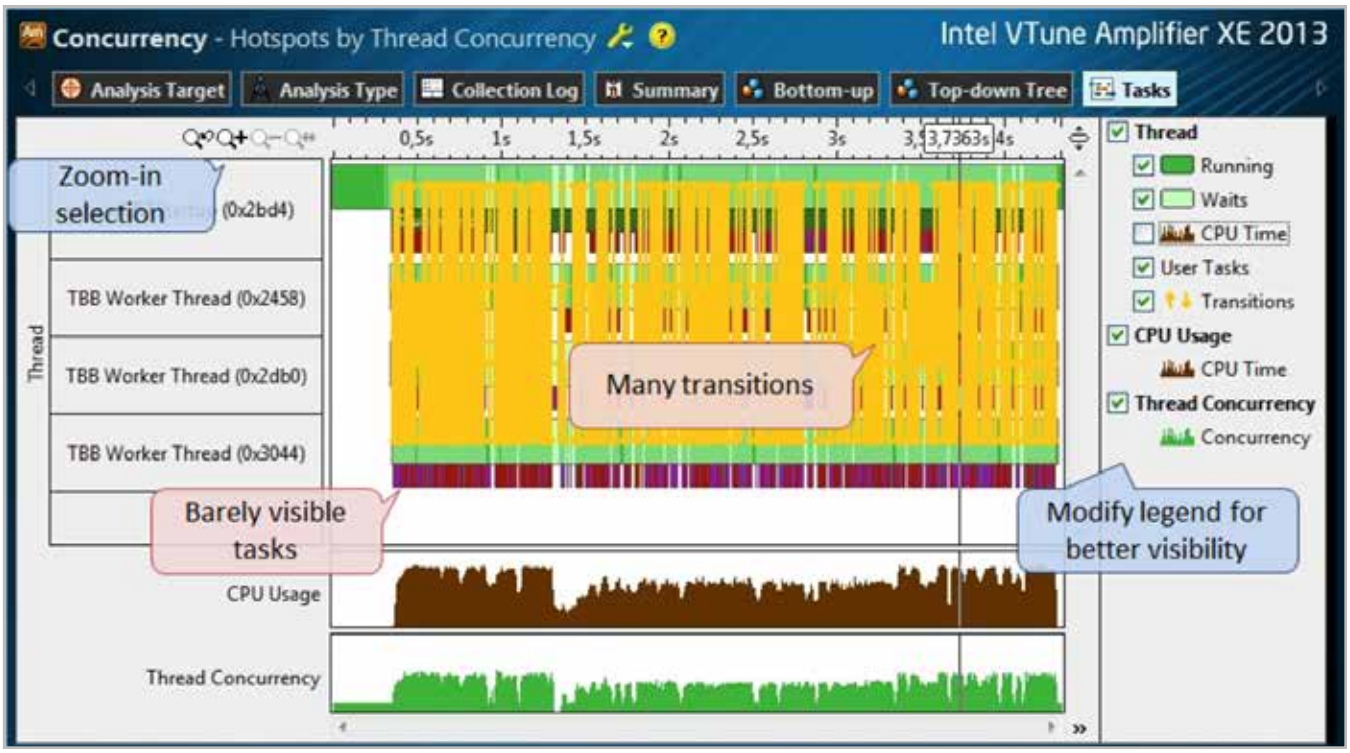
5. Add the path to Intel VTune Amplifier XE headers into your project: $(VTUNE_AMPLIFIER_XE_2013_DIR)
6. Statically link your binary with the *libittnotify.lib* that can be found in the path $(VTUNE_AMPLIFIER_XE_2013_DIR) lib[32|64], depending on the word size of your system.

Finally, you need to switch on the user task analysis in the Intel VTune Amplifier XE analysis configuration window (**Figure 8**). Now, you can run any analysis type and you will get results that can be mapped to your tasks.

After running the concurrency analysis collection, now switch to the Tasks View tab (**Figure 9**). You may see that the whole timeline is crowded with the yellow transition lines. These can be switched off in the legend control on the right pane, which can also control the brown CPU time graph. However, the tasks are rarely visible on the timeline since the colored bars are too thin to be distinguished. In this case, you can select any time region and zoom in from a context menu or with the zooming tools.



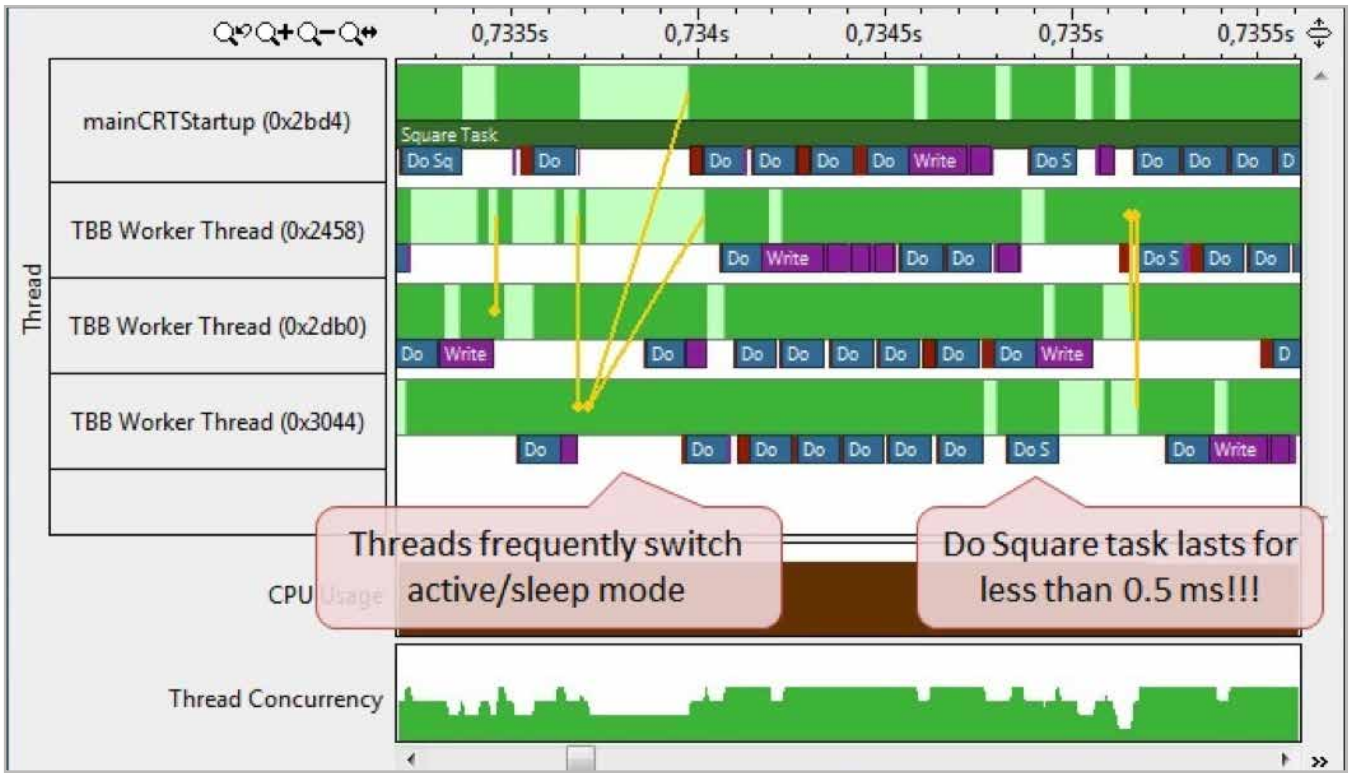
8



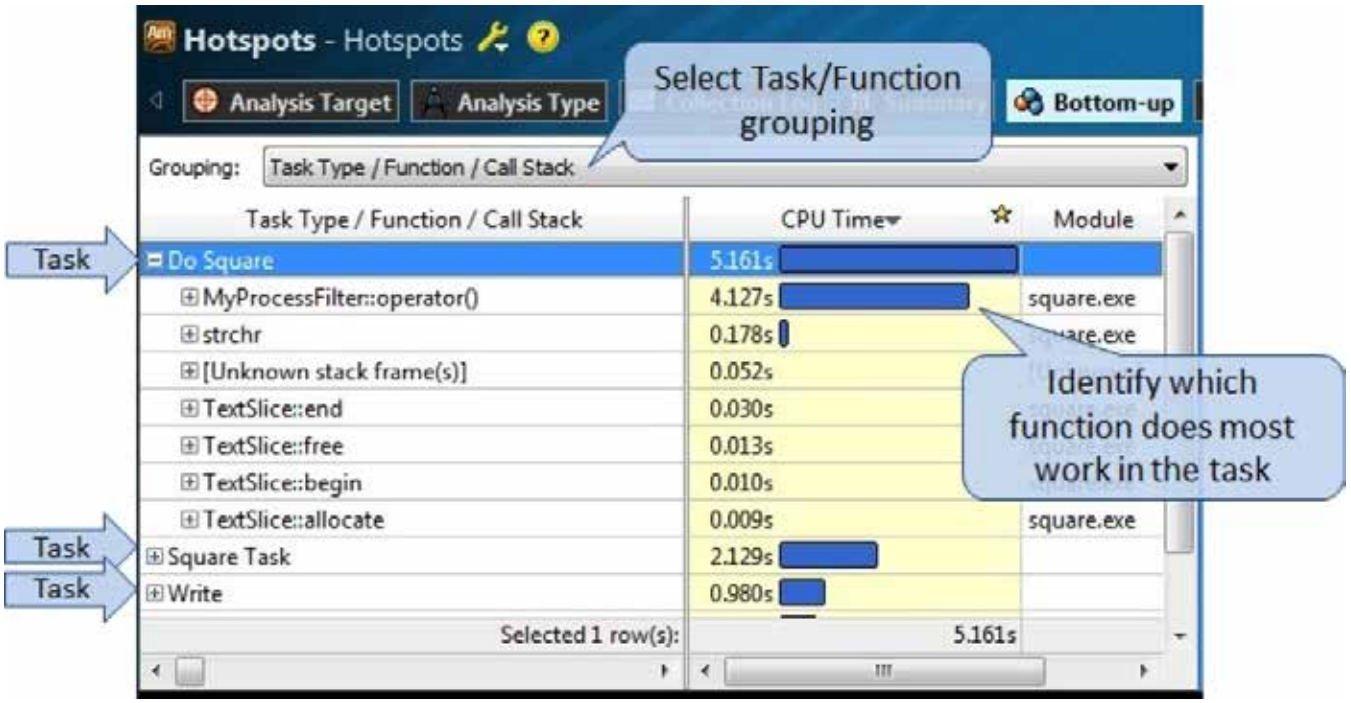
9

When you zoom in on the timeline, two problems are evident (**Figure 10**). First, there are huge gaps between the tasks, and the threads are not active in those gaps—obviously waiting for synchronization. Second, is the task duration: "Do Square" tasks are being executed in approximately 0.1 ms. This is a critically small time period if we take into consideration that tasks are managed by a task scheduler in Intel TBB, and it makes the scheduler do its job too frequently (overhead). The solution is to increase the amount of work to be done in each task in order to decrease the overhead of task management.

In this simple example, it's clear which functions are performing which tasks. In real applications, tasks may be executed in many functions. To identify which functions there are and decide where to increase task load, you can use the bottom-up view on either concurrency or hotspot analysis results. Simply change the grouping to TaskType/Function, and observe your task list in the grid. Expanding a task in this grouping, you'll see a function tree showing individual contributions to the CPU task time (**Figure 11**).

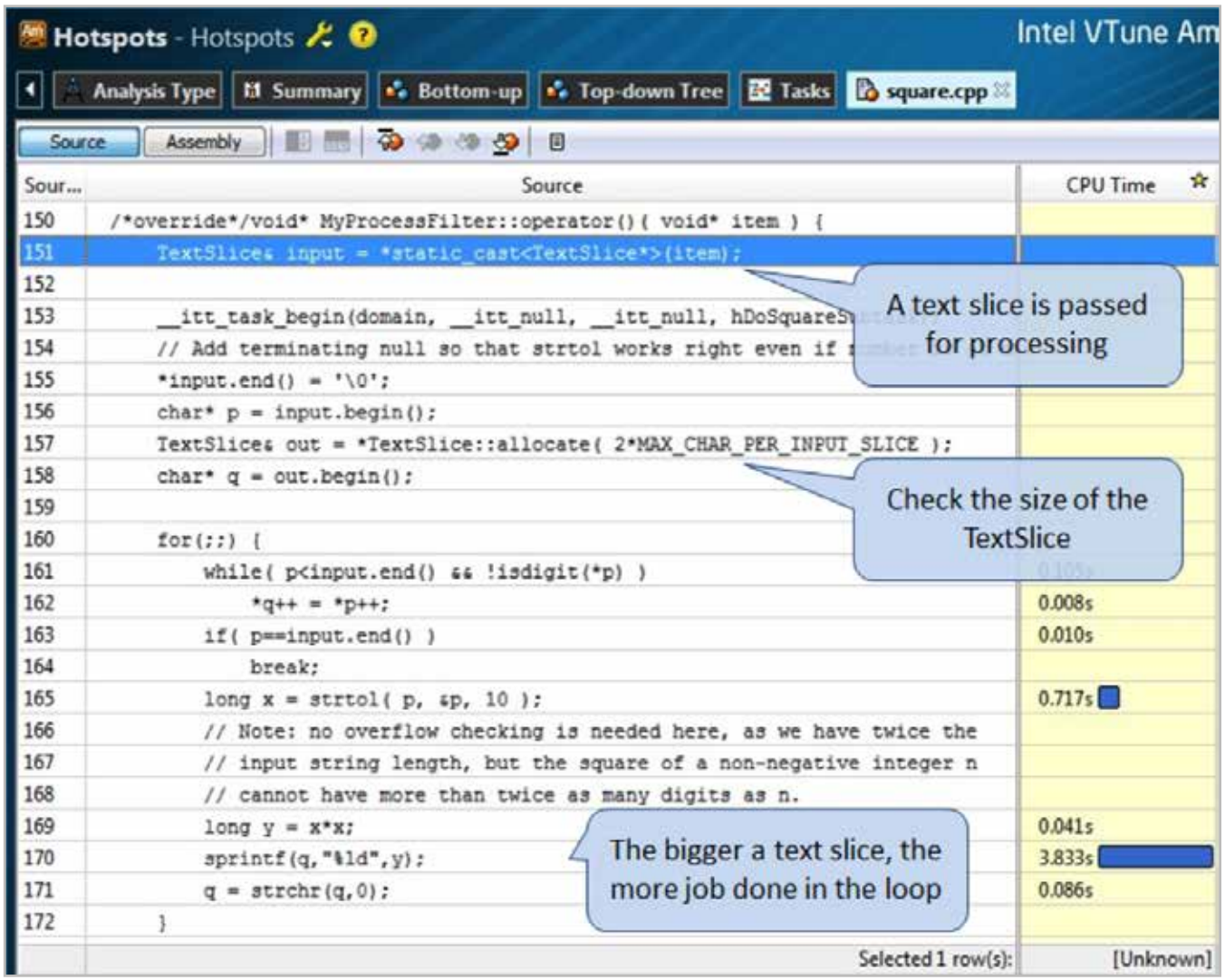


10



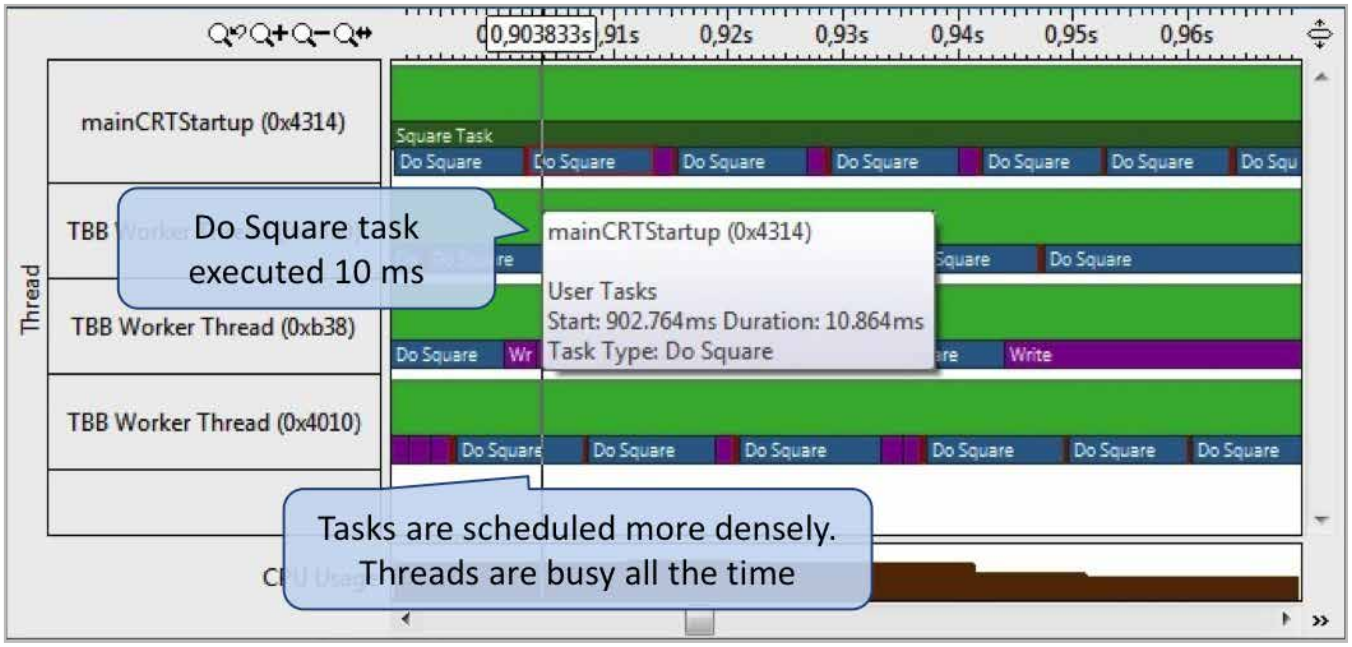
11

Next, we drill down to the *MyProcessFilter::operator()* function source code and identify that a text slice item is being passed to the function for processing (**Figure 12**). This text data is iterated. Each string is converted to a long type value, the value is multiplied by itself to get a square value, and, finally, the value is converted back to a string. The easiest way to increase a workload for the function would be to increase the allowable size of the text slice. This will linearly increase a number of processing operations. We simply select a new size limit, MAX_CHAR_PER_INPUT_SLICE, 100 times bigger than the original one (based on the knowledge of the task execution timing). And, we assume that the read and write operations will take advantage of the bigger slice as well.



12

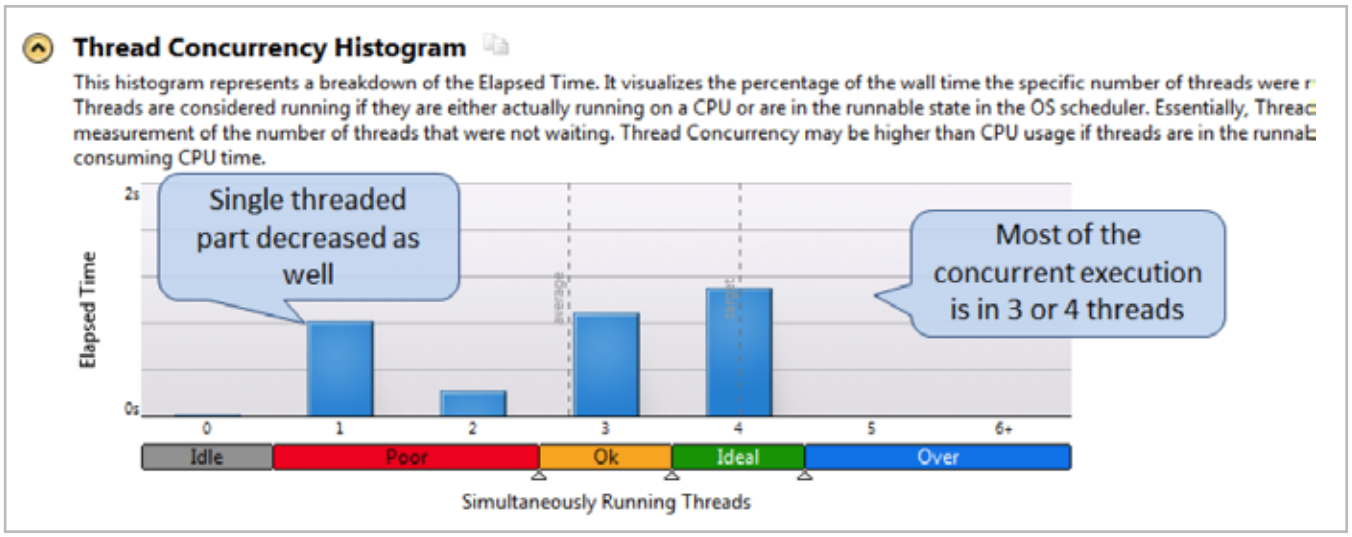
The recompiled application reveals much better results in the task analysis, at least visually (**Figure 13**). The Do Square task is being executed for ~10ms (hover a mouse over task block to see a callout with duration information). There are almost no gaps between the tasks, and the threads seem to be mostly busy. You might also observe how the pipeline schedules similar tasks, like the write task, into one stack on a thread in order to minimize overall scheduling and switching overhead and keep threads busy.



13

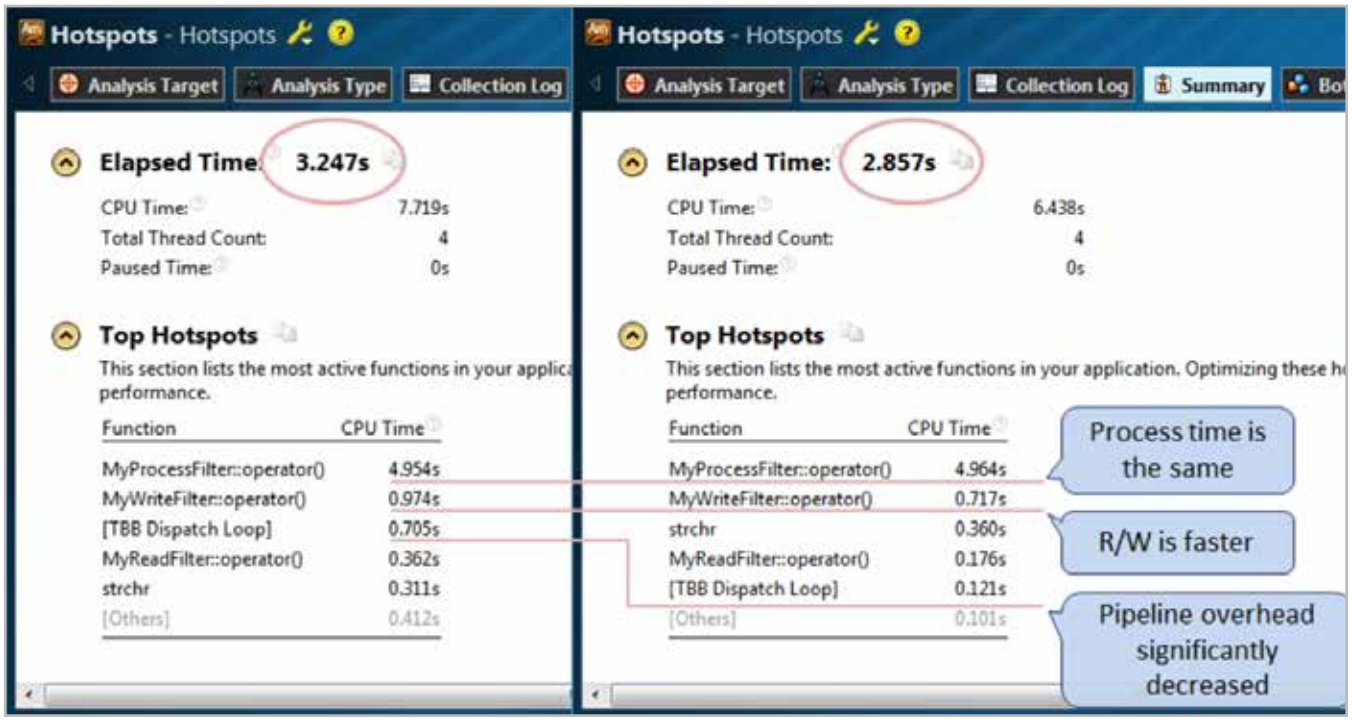
You may check an overall concurrency improvement by observing that the thread concurrency histogram is the summary view. As you can see (**Figure 14**), the parallel execution is mostly done in three or four simultaneously running threads. This is a noticeable improvement compared to the histogram we had for the initial analysis. You may also compare the average concurrency numbers, but be aware that this value is calculated taking into account a serial portion of execution as well.

A single threaded part of the execution remains, but is decreased to a third of its original value. This means that we achieved better parallelism in the pipeline, although there is still a serial execution portion of the program (you can observe this in the beginning of the timeline view). If selected and filtered in, it can be identified as an initialization phase in the main thread of the application. If you can clearly distinguish a portion of your code that is responsible for application initialization, you may want to use the start paused API and resume collection at a point when the application does actual work.



14

If you need to see numbers that characterize the improvement, use the compare feature of the tool, which is available for any view. For instance, look at two side-by-side screenshots of the summary panes corresponding to initial and final hotspot analysis (**Figure 15**). Though the elapsed time improved, you may notice that the duration of the processing stage *MyProcessFilter::operator()* did not change, as we did not change the amount of work. However, Intel TBB task dispatching overhead significantly decreased. The reading and writing stages timing decreased as well, taking advantage of bigger data slices.



15

Summary

Data and task level decomposition is widely used for achieving effective parallelism of algorithms. Many applications would gain from pipelining a combination of intrinsically serial and parallel stages of execution. The Intel TBB library may help dramatically save time and effort spent designing and supporting parallel algorithms such as pipeline, by taking care of thread management work for better parallelism. With the library, a programmer has no need to manage thread assignment and execution. A problem just needs to be represented as tasks to be completed, and Intel TBB provides dynamic distribution of tasks to execution threads available in your current system. Task creation may be nontrivial depending on the complexity of an application. Intel VTune Amplifier XE offers task analysis to provide programmers with powerful performance traces of user tasks in a visual and convenient way for problem investigation, saving additional development time. ●

BLOG HIGHLIGHTS

Intel® Xeon Phi™ Coprocessor Power Management Configuration: Using the micsmc Command-Line Interface

BY TAYLOR KIDD »

The descriptions of the various micsmc command line options presented here cover the purpose and interpretation of the data obtained from these sensors. Since the coprocessor has both passive and active cooling SKUs, there are inlet, outlet, and fan-related sensors. These sensors exist on both types of coprocessors, but their meaning only applies to the active versions. In passive versions, the meaning and usefulness of the sensors is going to depend upon the cooling provided by the housing of the host containing the coprocessors.

Command Line Usage: Measuring Power

Not unexpectedly, you can do everything that can be done using the graphical tool and more by using the command line version. For a full list of commands, see Table FULL. Table POWER below that shows the most relevant options related to power.

More ›

```
twkidd@knightscorner1:~> micsmc --help

Intel(R) Xeon Phi(TM) Coprocessor Platform Status Panel
VERSION: 3.1-0.1.build0
Developed by Intel Corporation. Intel, Xeon, and Intel Xeon Phi are trademarks
of Intel Corporation in the U.S and/or other countries.

This application monitors device performance, including driver info,
temperatures, core usage, etc.

This program is based in part on the work of the Qwt project
(http://qwt.sf.net).

The Status Panel User Guide is available in all supported languages, in PDF and
HTML formats, at:

   "/opt/intel/mic/sysmgmt/docs"+
```

Speed Threading Performance:
Enabling Intel® TSX Using Intel® C++ Compiler

By Anoop Madhusoodhanan Prabha, *Software Engineer, Intel*

Introduction

Processors were originally developed with only a single core. In the last decade, Intel and other processor manufacturers developed processors with multiple cores (multiple processing units). This paradigm shift in processor design from single-core to multiple cores made multithreaded programming all the more important. Software applications now have the opportunity to make use of all available cores on the target machine by distributing the workload across multiple threads.

Though all applications split a dataset across multiple threads in a mutually exclusive manner, there is always at least one resource that is shared by all application threads; for example, a simple reduction variable or a histogram (an array or hash table). This shared resource—being a critical resource—needs to be protected within a critical section to avoid data races. Traditionally, the critical section is executed in a serial fashion, since a lock controls the access to the critical section code. Any thread that executes the critical section needs to acquire the lock (acquiring involves read and write operations on the lock object). If the lock is acquired by thread A, for example, any other thread trying to execute the critical section protected by that lock needs to wait until that lock is released. This approach is called coarse-grained locking, since no matter which location of the shared resource is accessed, a single lock protects access to the shared resource as a whole.

Another approach is fine-grained locking. This is where a shared resource is divided into smaller chunks and each chunk's access is protected using separate locks. This approach offers an advantage over the coarse-grained approach, because now two threads can update the shared resource simultaneously, as long as they are updating two different chunks of the shared resource. But this approach demands extra code logic to be inserted, in order to decide which lock needs to be acquired (based on which shared resource chunk to be updated).

Another alternative solution is Intel® Transactional Synchronization Extensions (Intel® TSX). Intel TSX is a transactional memory solution implemented in a hardware layer (L1 cache). Two software interfaces are provided to make use of this hardware feature in programs. These software interfaces are available as a part of the Intel® C++ Composer XE package and have the same syntax as the coarse-grained locks, but do the job of fine-grained locks (chunk size being equal to L1 cache line size).

Intel TSX

Intel TSX is targeted to improve the performance of the lock-protected critical sections, while maintaining the lock-based programming paradigm. It allows the processor to determine dynamically if the critical section access needs to be serialized or not. In Intel TSX, the lock is elided, instead of fully acquired (the lock object is only read and watched, but not written to). This enables concurrency, because another thread can execute the same critical section since the lock is not acquired. The critical sections are defined as transactional regions. The memory operations inside the transactional region are committed atomically only when the transaction completes successfully.

Success and Failure of Transactional Execution

A transactional execution completes successfully when two or more threads that elide the lock of a critical section update the shared resource, such that no two threads update the shared resource locations within the same L1 cache line. Every memory address read or written to is maintained in the read-set and write-set by Intel TSX at cache line granularity. The failure of transactional execution will lead to transactional aborts.

Transactional aborts can occur for multiple reasons:

- Data conflict occurs when another logical processor either reads a memory location that is part of the transactional region's write-set, or writes to a location that is part of the read-set or write-set of the transactional region. Since Intel TSX detects data conflicts at the granularity of a cache line, unrelated locations placed in the same cache line will be identified as a conflict.
- Transactional aborts also occur when there are limited transactional resources. For instance, L1 data caches miss because the shared resource is too big to fit in the L1 cache.
- Usage of CPUID (processor query instruction) inside the transactional region will lead to a transactional abort.
- Interrupts generated during transactional execution will lead to transactional abort.

For more details on transactional aborts and how to minimize them in your program, refer to section 12.2.3 in **Intel® 64 and IA-32 Architectures Optimization Reference Manual**.

Software Interfaces for Intel TSX

The two software interfaces which expose the Intel TSX feature are:

- **Hardware Lock Elision (HLE):** This is a legacy-compatible instruction set extension. The instruction prefixes used are XACQUIRE and XRELEASE. This interface can be used by a programmer who wants to use Intel TSX targeting legacy hardware. The code paths for both transactional execution and non-transactional execution are the same. In Intel TSX mode, the program will first try the transactional execution and, if it is successful, an atomic commit is done on memory operations. If a transactional abort happens, all the memory operations are rolled back in the buffer. In this case, execution of the same transactional region will happen, but this time in a non-transactional mode (acquires the lock instead of eliding). Intel® C++ Compiler provides **HLE intrinsics** which can be used by programmers to implement lock elision logic in their threading library.
- **Restricted Transactional Memory (RTM):** This is a new instruction set supported by 4th generation Intel® Core™ processors. Check **http://ark.intel.com** for processors that support Intel TSX new instructions (**Figure 1**).

  The new instructions are XBEGIN, XEND, XABORT, and XTEST. Unlike the HLE interface, the transactional region is different from a non-transactional region (fallback path). If a transactional abort happens, all the memory operations are rolled back in the buffer and execution transfers to the fallback path where the execution happens in non-transactional mode. Intel C++ Compiler provides **RTM intrinsics**, which can be used by programmers to enable Intel TSX in their program.

Intel® TSX-NI	Yes

1

More information on the software interface for Intel TSX is explained in Chapter 12 of **Intel® Architecture Instruction Set Extensions Programming Reference**.

Intel TSX-Enabled locks

- **Pthread library in GNU C library:** glibc POSIX thread mutex and rwlocks are enabled with a lock elision code path, too. Applications using pthread locks will be automatically Intel TSX-enabled by linking with this new version of glibc (**glibcrtm-2.17 branch**) and setting certain environment variables. More information on the Intel TSX enabling effort and how to use them in applications is explained in the **Lock Elision in the GNU C library**.
- **OpenMP* library shipped with the Intel C++ Compiler:** Traditionally, programmers either use #pragma critical { } block or OpenMP explicit locks (omp_set_lock()/omp_unset_lock()) to protect the critical section in OpenMP programs. The explicit locks are enabled with an Intel TSX code path. Any application using OpenMP explicit locks can be run in transactional mode, without any code change, by setting the environment variable **KMP_LOCK_KIND=adaptive**.

Enabling Intel TSX Using RTM Intrinsics

A simple histogram example is used to demonstrate the use of RTM intrinsics to enable Intel TSX. This program reads an input .bmp (RGB format) image and creates a histogram that tracks the distribution of blue pixel values from 0 through 255. In this example, the dataset (image payload or data) is split in a mutually exclusive manner across multiple OpenMP threads. But the histogram (histogramblue in program) is a shared resource used in the transactional region. In the code sample (**Figure 2**), the "TSX" macro enables the Intel TSX code path. The transaction region begins with a call to _xbegin() compiler intrinsic and ends with a call to _xend() compiler intrinsic. Any code encapsulated between _xbegin() and _xend() is the one executed in transactional mode. In this case, transactional region has the relevant histogram location accessed and incremented by 1. If the call to _xbegin() returns _XBEGIN_STARTED, then execution switches to transactional mode, or else the control goes to the fallback path, which executes in non-transactional mode (this happens when the lock is acquired because of a transactional abort).

Download the sample code that demonstrates enabling Intel TSX.

```
104 #pragma omp parallel for shared(in, histogramblue) schedule(dynamic, 5000)
105     for(int i = 0; i < numofdatapoints; i+=3)
106     {
107         int j = i+2;
108 #ifdef TSX
109         unsigned int status = _xbegin();
110         if(status == _XBEGIN_STARTED){
111             histogramblue[in[j]]++;
112             _xend();
113         } else {
114             #pragma omp atomic
115             histogramblue[in[j]]++;
116         }
117 #else
118 #pragma omp atomic
119             histogramblue[in[j]]++;
120 #endif
```

2

Enabling Intel TSX Using OpenMP Library Shipped With the Intel C++ Compiler

The same histogram program can be programmed using OpenMP explicit locks which are Intel TSX-enabled. Intel C++ Compiler 14.0 includes the OpenMP library which is also Intel TSX-enabled. The procedure to make the above program run in TSX mode using OpenMP explicit locks follows:

1. Change the code, as shown below:

   This section defines and initializes an OpenMP lock.

```
48      omp_lock_t writelock;
49      omp_init_lock(&writelock);
```

   The OpenMP parallel for the block shown above is replaced with:

```
104 #pragma omp parallel for shared(in, histogramblue) schedule(dynamic, 5000)
105     for(int i = 0; i < numofdatapoints; i+=3)
106     {
107         int j = i+2;
108         omp_set_lock(&writelock);
109         histogramblue[in[j]]++;
110         omp_unset_lock(&writelock);
111     }
```

   Finally, the code for destroying the OpenMP lock is:

```
122     omp_destroy_lock(&writelock);
```

2. Set the environment variable KMP_LOCK_KIND=adaptive, as shown below:

```
KMP_LOCK_KIND    adaptive
```

3. Link to OpenMP library shipped with the Intel C++ Compiler.

In order to run the program in normal mode (non-transactional mode), just reset the environment variable KMP_LOCK_KIND. No code change is required.

Enabling Intel TSX in Intel® Threading Building Blocks (Intel® TBB)

Intel Threading Building Blocks 4.2 introduced **speculative_spin_mutex** which enables Intel TSX using HLE instructions. In a future release of Intel® TBB, speculative_rw_spin_mutex will be introduced with RTM instructions and will be available as a community preview feature. Intel TBB 4.2 is available as a part of the **Intel C++ Composer XE 2013 SP1 Update 2** package or as a standalone library at official **Intel TBB** website.

Conclusion

Intel TSX is the implementation of transactional memory in hardware. It provides two software interfaces to enable Intel TSX: HLE, for legacy processors, and RTM, for 4th generation Intel Core processors and above. The technology helps to exploit the inherent concurrency of the program by allowing concurrent execution of a critical section, unless a conflict is detected at runtime.

The efficiency of any application with Intel TSX can be evaluated using the **Intel® PCM tool** and **Linux* perf**. The tool provides insight into how many transactions happened successfully and the number of times transactional aborts happened. You can try Intel TSX by downloading the **evaluation version** of Intel C++ Composer XE. ●

Intel® TSX is the implementation of transactional memory in hardware. It provides two software interfaces to enable Intel TSX: HLE, for legacy processors, and RTM, for 4th generation Intel® Core™ processors and above. The technology helps to exploit the inherent concurrency of the program by allowing concurrent execution of a critical section, unless a conflict is detected at runtime.

Tachyon Ray Tracer:
Porting on the Intel® Xeon Phi™ Coprocessor

By Roman Lygin, *Sr. Technical Support Engineer, Intel*
and Dmitry Durnov, *Software Development Engineer, Intel*

Introduction

Introduction of the Intel® Xeon Phi™ coprocessor has generated strong interest within the development community. However, developers often need to adjust their applications in order to efficiently retrieve available compute power. The great advantage of Intel Xeon Phi coprocessor architecture is that it allows developers to leverage existing programming models and tools, and to maintain one source base across CPU and coprocessor.

This article presents a practical case study of porting Tachyon—an open source ray tracer[1] and part of the Spec MPI* suite—to the Intel Xeon Phi coprocessor. The porting project's objective was not to achieve the best possible performance, but rather to outline a workflow using development tools, motivation, and code optimizations. We planned to limit the effort invested and to avoid significant code rewrites. This was done on purpose—to better emulate a real project where developers always act within project constraints, at least of time and manpower.

The development community has already made some effort to test the feasibility of porting other ray tracers to the Intel Xeon Phi coprocessor.[2,3] Intel Labs, the dedicated research arm of Intel Corporation, has been developing Embree[4], a collection of ray-tracing compute kernels highly optimized for Intel® architecture, including the Intel Xeon Phi coprocessor.

Tachyon

Tachyon reads the definition of a scene from a file. A scene contains light sources and a 3-D model, defined as a collection of geometric primitives (triangles, spheres, etc.) with associated materials that determine light reflection properties. Tachyon also reads in the path of a camera moving around the scene.

Once data has been loaded, the Tachyon algorithm computes a frame for each camera position in the form of NxM image of RGB pixel values. The color of each pixel is computed using an initial intersection of an issued ray (direction depends on pixel coordinates and camera position) with the 3-D model, and further processing of the reflected ray(s) by a parameterized shader.

1 One frame computed for a sample 3-D model.

Figure 1 shows one frame computed for a sample 3-D model (teapot).

Algorithm performance is measured in frames per second (FPS). Tachyon can be used for both for immediate rendering and for throughput computing (when generated images are dumped to files), so our efforts were focused on increasing this metric, up to and beyond a reasonable interactive FPS threshold.

Tachyon supports hybrid parallelism, using MPI and OpenMP*. For each frame, the work is distributed among P MPI ranks using equal chunks: each MPI rank computes 1/P of the frame (more precisely, each rank computes M/P frame lines in a round-robin manner, where M is a frame height). The master rank 0 performs the same number of computations, collects computed chunks from workers, and post-processes the buffer (e.g., displays on the screen or outputs to a file). OpenMP threads are used to run parallel loop processing lines in a frame chunk delegated to an MPI rank.

There is no strict synchronization (MPI_BARRIER or alike) before each frame. However, due to frequent communications, the communication channel gets saturated and workers start to wait for the master before proceeding to the following frame. Thus, a priori, the algorithm is prone to poor scalability. The master is a bottleneck (due to performing the same amount of computations as workers and extra work) and synchronization is implicit.

Test Environment

We chose a cluster of four nodes, each equipped with two Intel® Xeon® processors E5-2680 and one Intel Xeon Phi coprocessor 7120. We used the latest Tachyon version 0.99 beta 6[1] with slight modification, which replaced line-by-line sends of each computed frame chunk with the single buffered send by each worker to the master. This modification stemmed from our prior analysis of the Tachyon code. It helps reduce the number of communications between the MPI ranks and significantly improves scalability.

Single floating precision mode was used to represent coordinates and normal. As a test workload, we used the model named teapot.dat (representing a famous Utah teapot model) (**Figure 1**), and the camera file teapot.cam concatenated 10 times (representing sufficient workload).

Performance was measured using an application mechanism that computes an average FPS by dividing a total number of computed frames by total elapsed computation time.

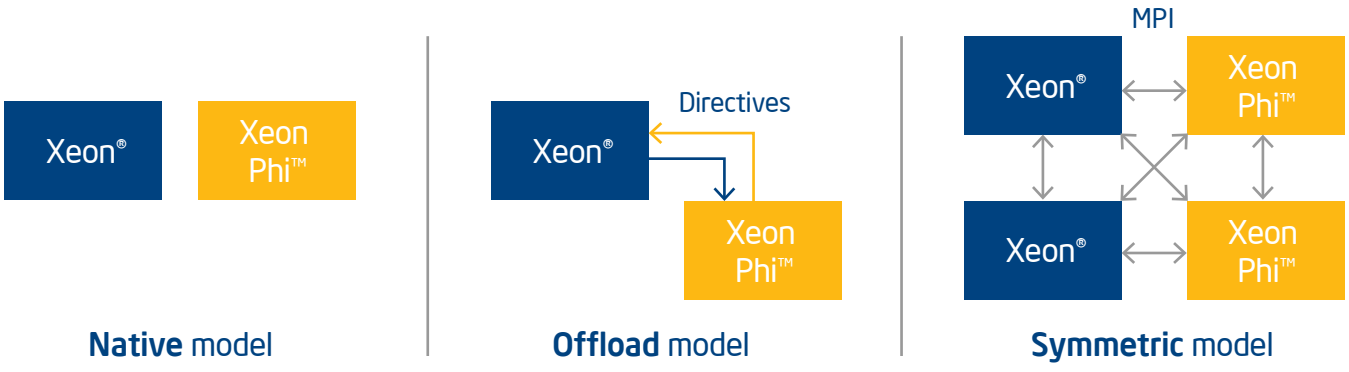
For compilation, execution, and performance analysis we used Intel® Cluster Studio XE 2013 SP1,[5] released in September 2013. Intel Cluster Studio XE is a software suite for efficient creation and analysis of parallel applications. It includes compilers, programming libraries, and tools for development on the Intel Xeon processor and Intel Xeon Phi coprocessor.[6]

Full specification of the environment follows:

- RHEL* 6.2; MPSS* 3.1.1
- Intel Cluster Studio XE 2013 SP1
- Four nodes with two Intel Xeon processors E5-2680 and one Intel Xeon Phi coprocessor 7120

We also tested a few other models ranging in complexity. The selected models used triangles as the most generic way to describe a 3-D model. Nonetheless, the ideas described here should be equally applicable to other primitives (e.g., spheres).

For heterogeneous Intel Xeon and Intel Xeon Phi coprocessor runs, a symmetric model (**Figure 2**) was used, since it required no up-front code modifications.



2 Execution models supported on Intel® Xeon Phi™ coprocessor.

Initial Baseline

Porting the Tachyon application to the Intel Xeon Phi coprocessor was straightforward. Essentially, the arguments -mmic and -fp-model fast=2 were added to Intel® compiler command line. The former is used to specify a target platform as Intel Xeon Phi coprocessor the latter, to choose a floating point computation model favoring performance over higher accuracy (which is acceptable for multiple applications, including ray tracing).

Initially, measured performance (in FPS) is shown in **Table 1**.

	Baseline
Intel® Xeon® processor only	141.8
Intel® Xeon Phi™ coprocessor only	38
Intel Xeon processor and Intel Xeon Phi coprocessor	39

1 Initial performance, in FPS (frame per seconds).

Thus, the heterogeneous run revealed a performance drop compared to a run with only an Intel Xeon processor. This required further analysis.

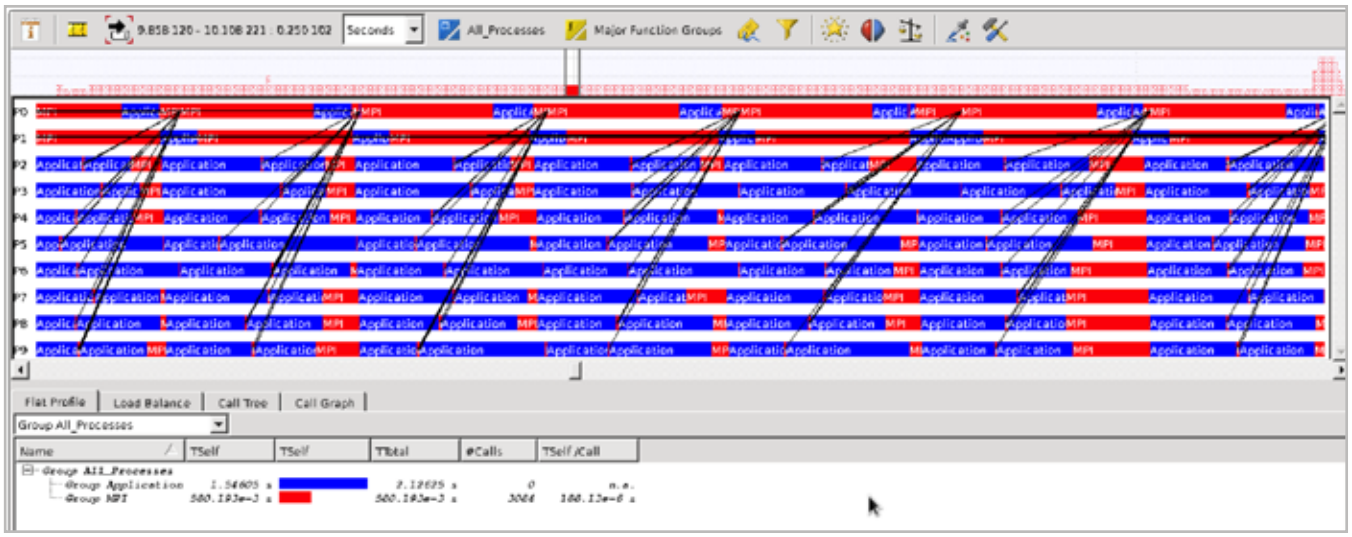
Performance Analysis

Intel Cluster Studio XE includes tools for performance analysis, working with both shared memory and distributed memory parallelism. These are:

- Intel® Trace Analyzer and Collector (or ITAC): Performance profiler and correctness checker of MPI-based applications;
- Intel® VTune™ Amplifier XE: Performance profiler for single-node applications

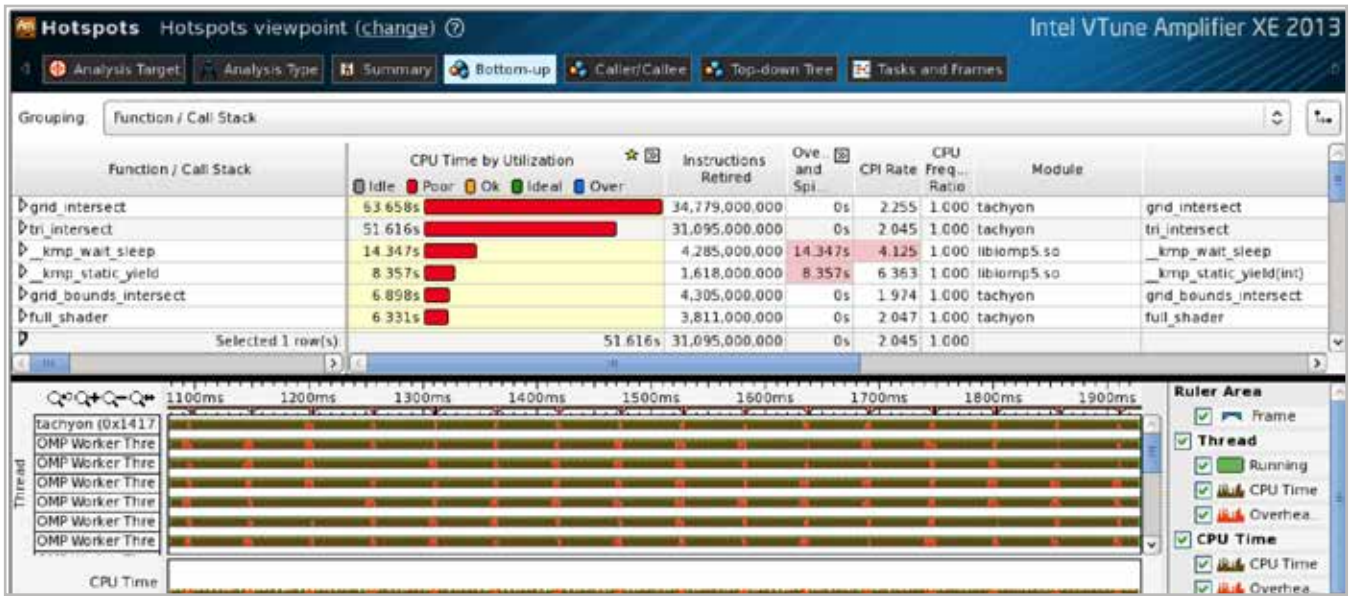
Working in combination, these tools allow developers to pinpoint performance issues in parallel applications, including hybrid ones.

Analysis with ITAC revealed that significant time is wasted by MPI ranks in waiting. Time spent in MPI functions (marked in red in **Figure 3**) is significant compared to payload. This is a consequence of the communication and implicit synchronization scheme of the algorithm: an inherent work imbalance (different complexities of equally split-frame chunks) combined with the different performance of the Intel Xeon processor and Intel Xeon Phi coprocessor. The ITAC timeline reveals that Intel Xeon processes (on top) suffer more (more time spent in waiting, in red color at the end of each frame).



3 ITAC shows that frame computations are quite synchronized (note the send/receive messages in black between workers and master; frames start almost simultaneously) and processes suffer from work imbalance (significant wait time marked in red).

Intel VTune Amplifier basic hotspot analysis reveals that each OpenMP parallel region (in each MPI process) also suffered from significant wasted time. Intel VTune Amplifier automatically highlights this issue both in the grid view (pink) and on the timeline (red) (**Figure 4**). Intel VTune Amplifier recognizes OpenMP parallel regions[7] and marks them on the timeline, thereby significantly simplifying analysis.

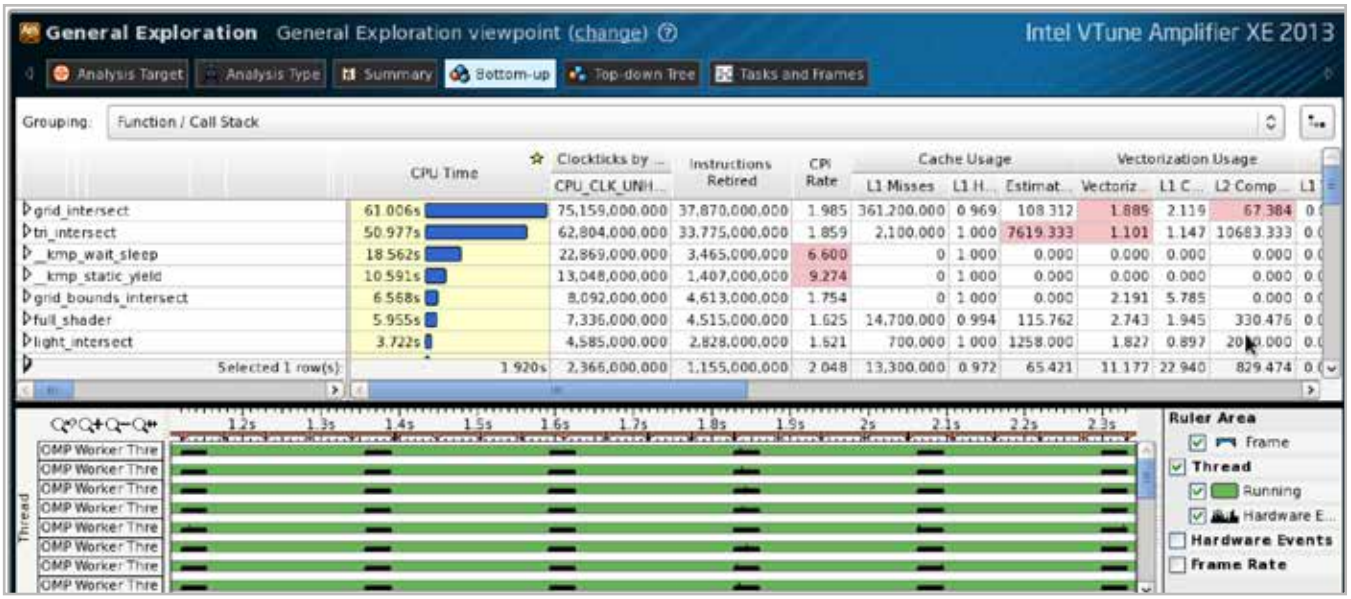


4 Intel® VTune™ Amplifier detects wasted time in OpenMP* parallel regions.

The wasted time at the end of each OpenMP parallel region also stems from the issue of work imbalance. Work imbalance appears due to different complexities of lines in each frame chunk being processed by OpenMP threads in each MPI rank. Note that the best performance was achieved using dynamic balancing and the minimum grain size of 1 (one line of frame chunk, i.e., with OMP_SCHEDULE=dynamic,1). As expected, use of static scheduling or greater chunk sizes caused even worse imbalances.

The results shown here have been collected using 61 threads (1/4 of available 244 threads) simulating a run of four MPI processes on the Intel Xeon Phi coprocessor. Using fewer MPI processes, and hence more OpenMP threads, produces even worse performance as the amount of work available for each OpenMP thread diminishes. This is obvious as the number of available parallel work items equals the number of lines in a frame. With a frame size of 512 x 512 and 61 OpenMP threads, each thread would be given eight to nine lines on average (=512/61). This might provide insufficient parallel slack for good work balance.

For deeper analysis of applications running on Intel Xeon Phi, Intel VTune Amplifier offers a command line mode with specific extra knobs. We followed expert recommendations[8] and, in particular, used the knob enable-vpu-metrics=true. This allowed us to detect poor usage of vector registers by hotspot functions (**Figure 5**).



5 Intel® VTune™ Amplifier highlights poor usage of vector registers by hotspot functions (coefficients of 1.9 and 1.1 out of a possible 16, are highlighted).

Combining these three analyses, we reached the following conclusions:

- Vector units are underutilized.
- 244 threads are underutilized as the application does not expose sufficient threading parallelism.
- Therefore, given the same work pile, MPI processes on the Intel Xeon Phi coprocessor ran slower than on the Intel Xeon processor. Regular synchronization forces faster Intel Xeon processor processes to wait for slower Intel Xeon Phi coprocessor processes, leading to total slowdown.

These deficiencies defined our next course of action:

1. To reduce synchronization between MPI processes by enabling dynamic scheduling of MPI ranks.
2. To retrieve sufficient threading (OpenMP) parallelism.
3. To apply vectorization.

Next, we'll take a close look at each of these modifications.

Modifications

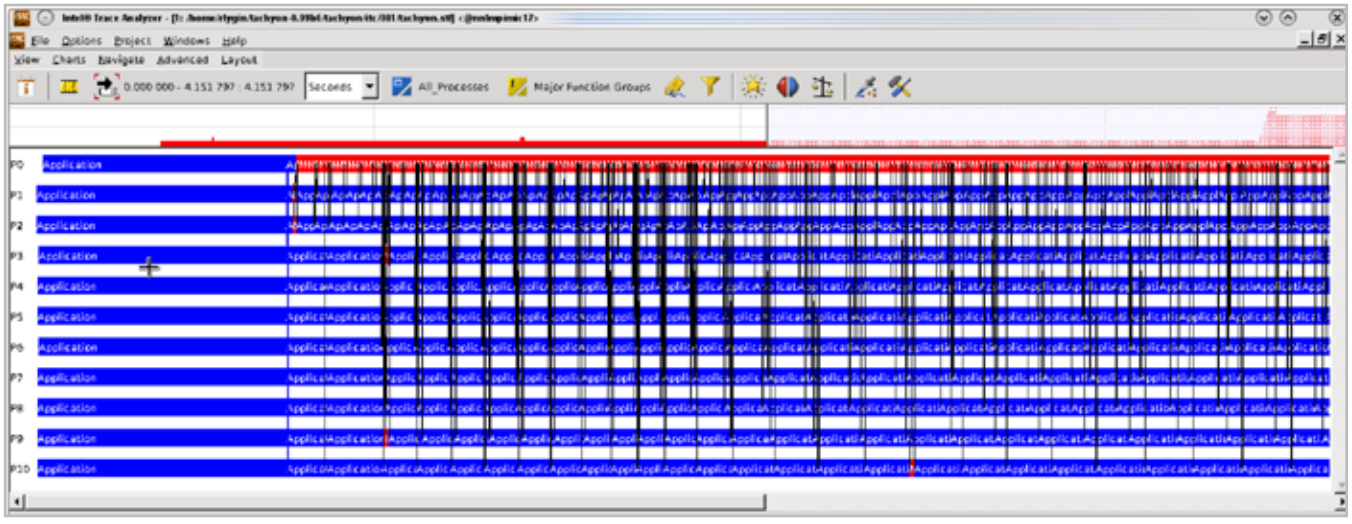
Enabling Dynamic Balancing with MPI

To reduce synchronizations between MPI ranks, we revisited the scheme that previously required computing frames strictly one-by-one by MPI ranks. Instead, we chose a scheme where each MPI rank computes an entire frame. An available MPI rank would just receive a frame number from the master rank, compute, and send back the computed frame, wait for the number of the next frame to compute, and so on. In this scheme, a strict order of frames to compute is no longer required: a master would ensure the order when outputting computed frames (on the screen or dumping to a file). The number of simultaneous frames in the fly is limited by memory capacity.

Implementation required about 250 lines of code changes. In this new scheme, the master no longer bears the same computational workload. Instead, it only coordinates computations and output.

This new approach allows us to compensate differences between processes running on Intel Xeon processors and Intel Xeon Phi coprocessors. Moreover, it helps to increase scalability to a larger number of ranks (including the case of only Intel Xeon processor processes). Thus, this is an example of optimization with dual benefits for Intel Xeon processors and Intel Xeon Phi coprocessors.

Rerunning our analysis with ITAC, we observed a much more structured and balanced execution (**Figure 6**).



6 New computation scheme shows no global synchronization between MPI ranks and excellent work balance.

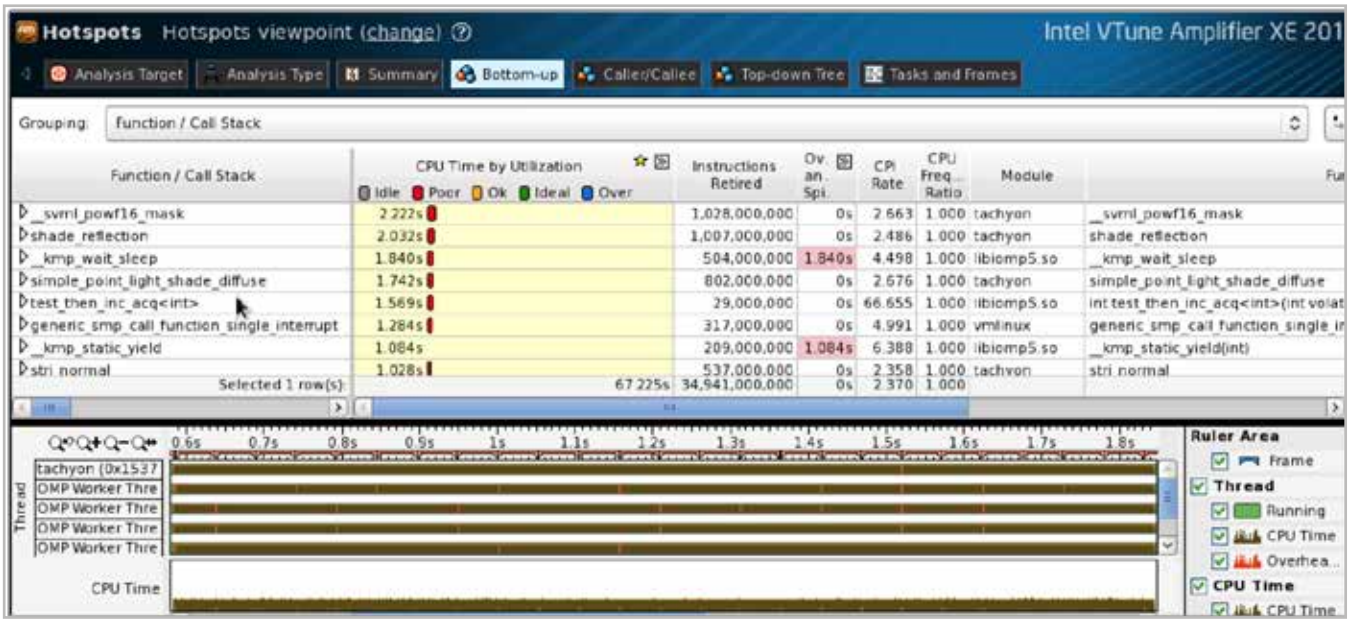
As expected, the master spends most of its time in waiting mode (shown in red) awaiting results from the workers. Outputting was put in a separate master's thread so it would not block computations.

Enabling Greater Threading Parallelism

To create parallel slack for OpenMP threads, we decreased the grain size. As previously mentioned, the grain size in terms of frame lines was already at its lowest point; we had to split the line and use its part as a grain. This required changing the loop so it would not use the Y coordinate, but rather a global pixel index. A code snippet (**Figure 7**) shows the modification, which only required changing six lines of code.

```
#pragma omp for schedule(runtime) nowait
#if defined(SINGLE_VAR_LOOP)
    for (p = 0; p < total_pixel; p += grain_size) {
      for (pp = 0; pp < grain_size; pp++) {
        int tp = p + pp;
        y = starty + (tp / xcount) * yinc;
        x = startx + (tp % xcount) * xinc;
        addr = hsize * (y - 1) + (3 * (x - 1));
#else /* SINGLE_VAR_LOOP */
    for (y=starty; y<=stopy; y+=yinc) {
      addr = hsize * (y - 1) + (3 * (startx - 1));
      for (x=startx; x<=stopx; x+=xinc,addr+=hskip) {
#endif
        primary.frng = cachefrng; /* each pixel uses
        col=scene->camera.cam_ray(&primary, x, y);
```

7 Code change in an OpenMP*-enabled for loop to reduce the grain size.



8 Intel® VTune™ Amplifier confirms significantly improved work balance across OpenMP* threads.

As the number of computations for each pixel is significant compared to parallelism overhead, the lower grain sizes yielded better performance. We chose 8 as the minimum value to prevent "false sharing" (having multiple threads writing into the same cache line).

We can see from the timeline and grid view that rerunning Intel VTune Amplifier achieved significantly improved work balance and reduced OpenMP overhead (**Figure 8**).

As with the first modification, this also helps with better scalability on Intel Xeon processors, although the benefit is lower, since the number of available threads differs significantly (e.g., 16 vs. 61).

Enabling Vectorization

Finding an opportunity for vectorization was the greatest challenge, because:

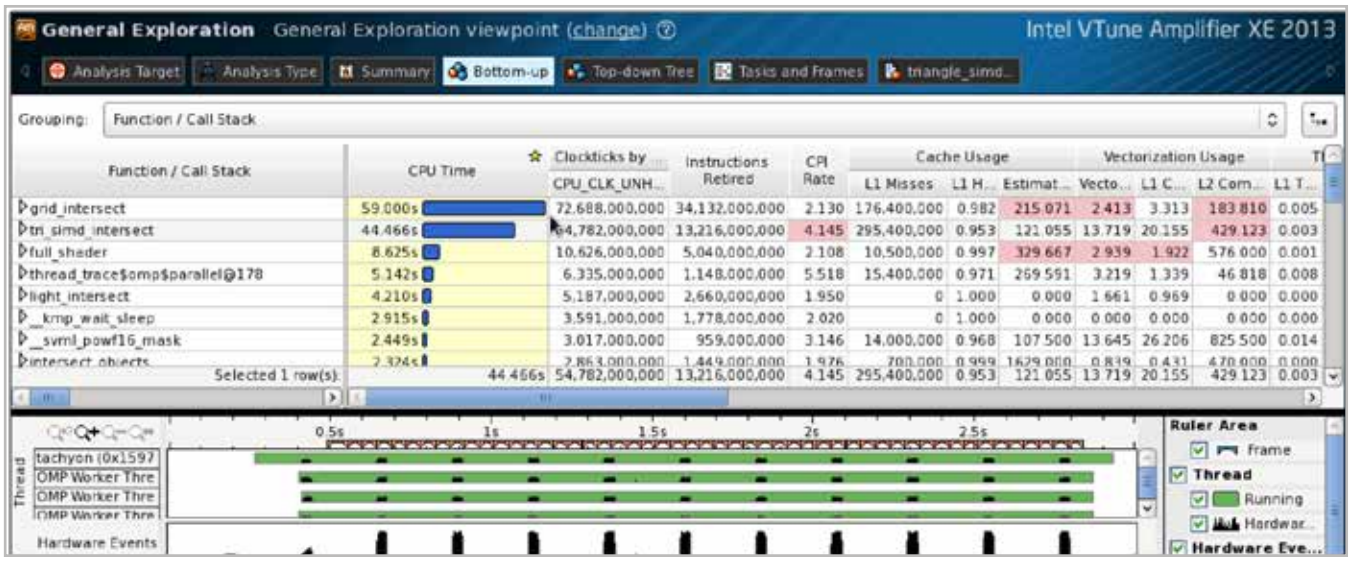
- A hotspot function [tri_intersect(), which computes ray-triangle intersection] has no loop that could be vectorized by the compiler, and
- The hotspot function is applied on a linked list [in the second hotspot function grid_intersect()], which does not allow us to write an elemental function that can be applied to an array of elements.

To enable vectorization in function tri_intersect(), we had to change the data layout. We defined an array of 16 triangles and implemented a tri_simd_intersection() function that would compute intersection of a ray with 16 triangles at once, using vectorized arithmetic operations. Our size choice (16) was driven by the size of vector register on the Intel Xeon Phi coprocessor, which is 512-bit and allows storing of all 16 single precision floating numbers.

This same idea allowed us to enable vectorization on Intel Xeon processors, using 4 triangles for SSE registers and 8 triangles for AVX registers. To minimize code changes and to avoid duplication of tri_simd_intersect() versions for each architecture, we implemented tri_simd_intersect() using a C++ template mechanism. Here, we used data structure as a template parameter and overloaded arithmetic operations (+,-, dot- and cross-products, etc.) for each data structure. To enforce vectorization in overloaded operations, we chose explicit intrinscis (prefixed with _mm512, _mm256, or _mm depending on the target architecture). The implementation for SSE and AVX was based on Embree[4] architecture-specific data structures; the Intel Xeon Phi coprocessor was implemented using similar ideas. An alternative implementation could be based on plain loops, along with relying on compiler auto-vectorization (or #pragma simd or similar hints).

Creation of these composite objects occurs only once in the algorithm, after a scene load. This extra cost is negligible in comparison to computation time. As the number of triangles is not necessarily a multiple of a respective number (16, 8, or 4), a bit mask is maintained to distinguish real triangles in the array. Many vector intrinsics accept this mask as an extra parameter.

Enabling vectorization allowed us to increase the intensity of vector instructions and to make tri_intersect() a second hotspot (**Figure 9**).



9 Intel® VTune™ Amplifier confirms vector instruction efficiency increase.

The impact of vectorization depends strongly on a 3-D model and is especially beneficial for complex models, where triangles are more efficiently packed into arrays.

Performance Results

Results collected on the Tachyon version that includes all three modifications are shown in **Table 2**, along with initial baseline and speed-up.

	Baseline	Final version	Speedup
Intel® Xeon® processor only	141.8	218.3	1.5x
Intel® Xeon Phi™ coprocessor only	38	173.2	4.6x
Intel Xeon processor and Intel Xeon Phi coprocessor (symmetric mode)	39	378.7	9.7x

2 Resulting performance, in FPS, along with speed-up vs. baseline.

Conclusion

We have selected an application that initially demonstrated poor performance on Intel Xeon Phi coprocessors (compared to Intel Xeon processors), and used a structured workflow to analyze and address performance issues. This case study demonstrates key benefits of the Intel Xeon Phi coprocessor as a programming platform, including:

- Allowing developers to use the same programming models and tools (MPI, OpenMP, and performance profilers)
- Achieving dual benefits for efforts spent on optimization for one platform (Intel Xeon Phi coprocessor or Intel Xeon processor)

Final performance of a combined run is about 1.73x of an improved Intel® Xeon® processor-optimized version (378 FPS vs 218 FPS), and speedup is 9.7x vs. initial baseline. Although achieving these results required code modifications, they have been limited. Required efforts were about 5 weeks, which seems affordable in real-world projects and a reasonable investment given the return.

You've seen a representative example of porting existing applications to Intel Xeon Phi coprocessors—we hope it will be of interest in your software development efforts.

Acknowledgements

The authors would like to thank John E. Stone for the Tachyon application design, and providing interim versions; our colleague at Intel, Andrew Tananakin, for initial analysis and correction of the OpenMP-related issues in the earlier version of Tachyon; the Embree team for implementation of data structures enabling vectorization; and Hans Pabst for his inputs into this case study and delivering reviews at user events. ●

References

1. J. Stone, *Tachyon Ray Tracing System*, **http://jedi.ks.uiuc.edu/~johns/raytracer/**
2. **http://www.slideshare.net/NakataMaho/some-experiences-for-porting-application-to-intel-xeon-phi**
3. **http://www.theismus.de/HPCBlog/?page_id=114**
4. *Embree* Ray Tracing Kernels*, **http://embree.github.io/**
5. Intel® Cluster Studio XE: **http://software.intel.com/en-us/intel-cluster-studio-xe**
6. Intel® Xeon Phi™ Coprocessor Developer Resources: **http://software.intel.com/en-us/mic-developer**
7. K. Rogozhin, *Profiling OpenMP* applications with Intel® VTune™ Amplifier XE*, **http://software.intel.com/en-us/articles/profiling-openmp-applications-with-intel-vtune-amplifier-xe**
8. S. Cepeda, *Optimization and Performance Tuning for Intel® Xeon Phi™ Coprocessors, Part 2: Understanding and Using Hardware Events*, **http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding**

Building Native Android Apps With Intel® C++ Compiler

By Anoop Madhusoodhanan Prabha, *Software Engineer, Intel*

Increasingly, manufacturers are offering Android devices, especially phones and tablets, based on Intel® Atom™ processors.

Most Android apps are written in Java*. However, a large number of developers use C or C++ and use Android NDK* to build the shared library and Java Native Interface* (JNI). App performance, including a smooth user experience, is one of the key reasons for using native programming over Java.

Intel® C++ Compiler for Android is compatible with the gcc compiler in the Android NDK. App developers looking for possible performance advantages should consider using the Intel® compiler. Because it's compatible, most existing programs written in C or C++ can be compiled using the Intel compiler, and the resulting shared library (.so) can be shipped with an Android apk file. In other words, developers can build and ship an app using the Intel C++ Compiler in the same way they currently use NDK tools, to prepare it for distribution through stores such as Google Play*.

Some of the main features of Intel C++ Compiler 14.0 for Android are listed below:

1. **Multiple Development Platform Support and Automatic Integration With Android NDK:** Intel C++ Compiler 14.0 supports development on Linux*, Windows*, and OS X*. The compiler installer will integrate the compiler with Android NDK based on the NDK path specified by the user. This integration helps in triggering the Intel tool chain through ndk-build script. The build can be triggered using icc/icpc directly or from the Eclipse* IDE. More information is available in the **Getting Started Guide**.
2. **GNU Compatibility:** Compatible with the GNU C++ compiler in the Android NDK for multiarchitecture support. More information on benchmark information is published at **Intel Developer Zone**.
3. **Intel Atom Processor Optimizations:** Optimization switch –xatom_ssse3 for Saltwell-based Intel Atom processor and –xatom_sse4.2 for Silvermont-based Intel Atom processor.
4. **Interprocedural Optimization and Profile Guided Optimization:** Intel C++ Compiler for Android is based on the classic Intel® C++ Composer XE. Thus, most of the advanced optimizations such as interprocedural optimization and profile-guided optimization are supported for an Android target.
5. **Intel® Cilk™ Plus:** This is a programming model which simplifies implementing task parallelism and data parallelism. For more information, see **Using Advanced Intel® C++ Compiler Features for Android Applications**.

A typical, high-level lifecycle of Android apps starts with development, moves to building/packaging them as .apk files, and then to testing them on physical devices or emulators. When physical devices aren't available, emulators are a practical solution. X86 emulators are now shipped with the Android SDK. For more information on how to install the x86 emulator using Android SDK Manager*, refer to **Installing the Intel® Atom™ x86 System Image for Android Emulator* Add-on from the Android SDK Manager***.

To improve the user experience and performance of x86-based emulators, the two steps below are highly recommended:

1. Turn on Intel® Virtualization Technology in the BIOS.
2. Install **Intel® Hardware Accelerated Execution Manager (Intel® HAXM)**. More information on how to use Intel HAXM on Linux* is explained in **How to Start Intel® Hardware-Assisted Virtualization (hypervisor) on Linux* to Speedup Intel Android x86 Gingerbread* Emulator**.

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

BLOG HIGHLIGHTS

Intel® Xeon Phi™ coprocessor Power Management Configuration: Using the micsmc GUI Interface

BY TAYLOR KIDD »

How Do We Configure Coprocessor Power Management

Depending upon your suite of workloads, the size of your data center, its configuration, as well as other considerations, you may want to configure your power management (PM) differently from the default. See the previous blogs in this series. You can do this in two ways, either at boot time or at runtime using the micsmc application.

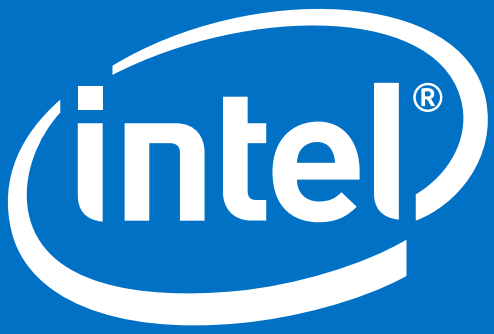
micsmc is the Intel® MIC System Management and Configuration application. It allows a user on the host to monitor and configure coprocessors attached to the host. It also has a command line version allowing scripts to do much the same.

To graphically view power status, and to enable and disable power management states on coprocessors attached to a host, you need to use a graphical window manager, such as FVWM* or GNOME*, to log into the host system. Log in as root, run the usual "compilervars.sh intel64", and execute micsmc. You can run micsmc from a user account to monitor coprocessors, but you cannot change PM states. If you want to be able to change PM states, you need to have superuser privileges.

This brings up the monitoring console. See Figure GUI below. Not surprisingly, clicking on 'Advanced' => 'Settings' brings up the "Advanced Settings" window. The advanced settings window allows you to change the PM behavior of the cards, as well as reset and reboot the cards.

```
# source /opt/intel/composer_xe_2013_sp1/bin/compilervars.sh intel64
# micsmc &
#
```

More >



Software

The Parallel Universe