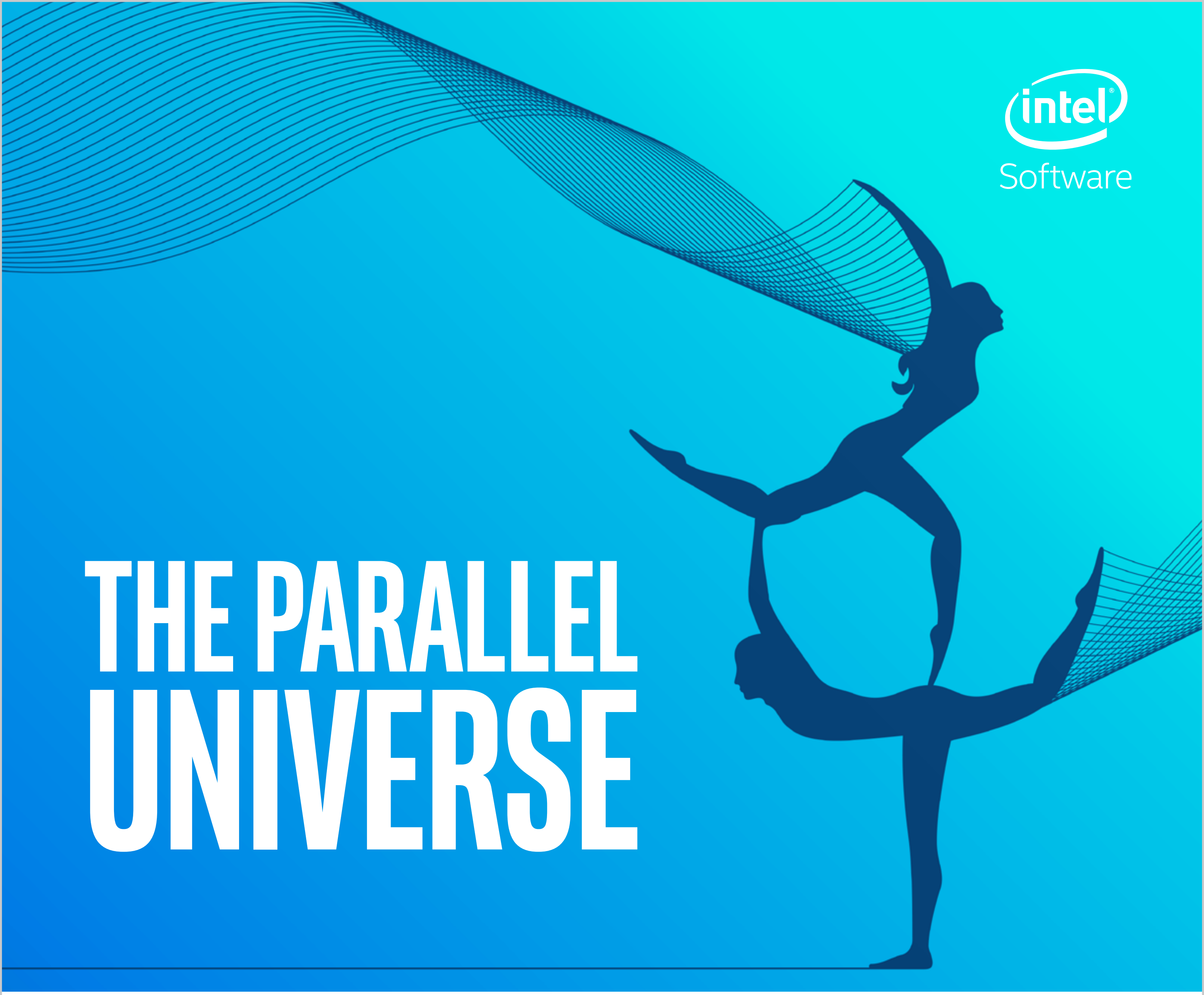


THE PARALLEL UNIVERSE

Improve Productivity and Boost C++ Performance

Intel® C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology

OpenMP* API Version 4.5: A Standard Evolves

Issue 24 2016

CONTENTS

CONTENTS CONT...

LETTER FROM THE EDITOR

James Reinders, an expert on parallel programming, is coeditor of the recent ***High Performance Parallel Programming Pearls Volumes One*** and ***Two***. His other book credits include ***Multithreading for Visual Effects*** (2014), ***Intel® Xeon Phi™ Coprocessor High Performance Programming*** (2013), ***Structured Parallel Programming*** (2012), ***Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism*** (2007), and ***VTune™ Performance Analyzer Essentials*** (2005).

Time-Saving Tips as Spring Begins in the Northern Hemisphere

I love springtime. I carve out time so I can plant seeds to get our garden going. Nature gets busy, and everyday life seems to pick up as well. With so much to do, we all look for ways to make life easier, more efficient, and more productive. Software development can have its fair share of menial, time-consuming tasks. I definitely want to fast-forward past the tedious so there is more time for fun stuff.

In this issue, our authors offer some suggestions on how to provide less time-intensive ways to achieve better performance with some very specific solutions: AoS to SoA in C++, bi-endian compilation, and richer OpenMP* support. It's fair to say that working to get vectorization can be tedious when reasoning which layout of data in memory will yield the best performance. Should AoS become SoA?

Our first feature, "**Improve Productivity and Boost C++ Performance**," discusses how the new Intel® SIMD Data Layout Template (SDLT) helps with memory layout optimization to potentially increase performance of SIMD-ready C++ code.

We do not make tools to settle Lilliputian disagreements about eggs, but we can help deal with compiling code for Intel® processors even if the code was written originally for big-endian processors. "**Intel® C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology**" describes how this edition of Intel® compilers can help easily migrate legacy applications from big-endian to little-endian architectures. Rather than extensively rewriting code, we can reuse code for both architectures with this helpful tool.

"**OpenMP* API Version 4.5: A Standard Evolves**" is an in-depth look at the new version of OpenMP and how it helps us better express parallelism in applications and offloaded code. OpenMP 4.5 offers task-generating loops that relieve us from having to find solutions to tame load imbalances and resolve composability issues caused by parallel loops, among other benefits.

We may never escape life's seemingly endless to-do lists. But, we do offer some ways, in this issue, to save time in our pursuits of high performance for our applications. What you do with your time savings is up to you!

James Reinders
March 2016

IMPROVE PRODUCTIVITY AND BOOST C++ PERFORMANCE

Introducing the Intel® SIMD Data Layout Template (Intel® SDLT) to Boost Efficiency in Your Vectorized C++ Code

Udit Patidar, ***Product Marketing Engineer, Developer Products Division,*** **Intel Corporation**

Intel provides many tools to help you find hotspots in your code (e.g., Intel® VTune™ Amplifier XE) and offer advice on optimizing your code (e.g., Intel® Advisor XE). **Intel® C++ Compiler** generates detailed optimization reports that can tell you whether vectorization of a particular C++ loop was successful.[1] Intel is introducing a new template library included in Intel C++ Compiler, a component in Intel® Parallel Studio XE or Intel® System Studio. If you are an application developer using C++, you know it takes considerable effort to hand-optimize memory access layout. The **Intel® SIMD Data Layout Template** (Intel SDLT) library optimizes C++ codes by enabling the programmer to switch from an array of structures (AOS) representation to a structure of arrays (SOA) representation with minimal effort or overhead. Such an SOA representation improves memory utilization and facilitates vectorization avoiding data-structure layout conversions.

The latest Intel® processors and coprocessors offer vector instructions and support the single instruction/multiple data (SIMD) programming model. With Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instructions, vectorized codes theoretically become capable of delivering 8x more peak performance for double-precision (or 16x single-precision) floating-point computations over the performance of nonvectorized codes. Unfortunately, the theoretical limit is rarely reached, especially on legacy codes that were not written with **vectorization** in mind. Without addressing the memory layout of data, you may end up leaving a lot of performance on the table.

In C++, the choice of how to lay data out in memory is crucial to achieving efficient vectorization. This is especially true when dealing with structures and arrays. It is common for developers to represent an array with a container from the C++ Standard Template Library, like `std::vector`.[2]

```
struct Point3s

  float x;
  float y;
  float z;
};

std::vector<Point3s> inputDataSet(count);
std::vector<Point3s> outputDataSet(count);

for(int i=0; i < count; ++i) {
  Point3s inputElement = inputDataSet[i];
  Point3s result = // loop iteration independent algorithm
                   // that transforms the inputElement.
  // algorithm uses high level objects & helper methods.
  outputDataSet[i] = result;
}
```

Although such a data layout might feel natural for a C++ programmer, the overhead of loading this AOS data set into vector registers can negate the performance gains of vectorizing. Converting the AOS representation to an SOA representation might be better suited for vectorization, but is very counterintuitive to the C++ programmer.[3]

Enter Intel SDLT.

The SDLT library provides an AOS interface to the user but stores the data in SOA format in memory. It abstracts the problem of **SIMD**-friendly data layout away from the programmer. Intel® SDLT offers a high-level interface using standard ISO C++11 features and does not require special compiler support to work. Because of its SIMD-friendly layout, it can better take advantage of the Intel® compiler's performance features such as OpenMP* SIMD extensions and Intel® Cilk™ Plus SIMD extensions.

To use the library, you declare your data types as primitives, describing them to SDLT. Then you can use your primitives with generic SDLT containers (instead of `std::vector`). Use the container's data accessors instead of array pointers or iterators. When used with an explicit vector programming model, the accessors from Intel SDLT containers handle data transformation and let the compiler generate efficient SIMD code.

```
SDLT_PRIMITIVE(Point3s, x, y, z)

sdlt::soa1d_container<Point3s> inputDataSet(count);
sdlt::soa1d_container<Point3s> outputDataSet(count);

auto inputData = inputDataSet.const_access();
auto outputData = outputDataSet.access();

#pragma forceinline recursive
#pragma omp simd
for(int i=0; i < count; ++i) {
  Point3s inputElement = inputData[i];
  Point3s result = // loop iteration independent algorithm
                   // that transforms the inputElement.
  // Keep algorithm high level using object helper methods.
  outputData[i] = result;
}
```

When an accessor is used with the loop's index to export to or import from a local variable (inside loop body), the compiler's vectorizor can transparently access the underlying SIMD-friendly data format and, when possible, perform unit stride loads. In many cases, the compiler can optimize its private representation of local objects inside loop to be SOA. In the example above, because the container's underlying memory layout is SOA as well as the compiler's private representation of local objects, the compiler can generate efficient unit stride loads.

Representing data in memory SOA enables the compiler to directly load array elements into vector registers. This greatly helps SIMD vectorization reach its potential. In contrast, computing directly on AOS layouts can necessitate extra instructions to populate vector registers, essentially consuming execution slots but not contributing any results.

In summary, you can use SDLT to increase your productivity by letting it worry about optimal memory layout, as well as potentially increase performance of your SIMD-ready C++ code, as shown in the table below.

Increase productivity for developers using C++	Rather than stop using C++ objects and revert to C arrays when enabling SIMD code, use generic containers with minimal effort. Let SDLT handle data layout transformations for you.
Improve performance	By making memory access contiguous, the compiler can generate more efficient SIMD code and, in some cases, utilize vectorization where the overhead was previously too high.
Integrate simply	SDLT containers provide a similar interface to `std::vector`. Data accessors are compatible with existing Intel vector programming models and fit right in with other Intel® performance libraries.

When evaluating whether SDLT might be suitable for you, consider this rule of thumb:[4]
It is almost always better to vectorize than not to vectorize on Intel SIMD-capable hardware.

Given the increased number of data lanes in the latest Intel coprocessors, not carefully considering SDLT might prevent you from modernizing your code and exaggerate the shortcomings and downsides of traditional C++ applications. Try the new SDLT feature to enable an SOA layout for your C++ application. It might just improve your SIMD efficiency and give you a pleasant surprise.

SDLT Helps DreamWorks Animation Advance the State of the Art

The AOS-to-SOA problem is widely studied. And SDLT has already gained considerable traction in many industries. As an example, DreamWorks Animation has a mathematical library which was written using the standard C++ programming principles (i.e., without explicit vectorization in mind). It would be disruptive and difficult to manually change the data structure layout to an SOA on such a large scale. It was a large barrier to enabling SIMD, since the mathematical library affects almost all areas of character animation. Once refactored with SDLT, existing codes continued to compile (without changes) and loops could now be vectorized, showing great benefits.

"We used [SDLT] to vectorize the deformer code in Premo, the in-house animation tool for DreamWorks Animation. The performance improvements we were able to achieve were dramatic, and these improvements will translate directly into higher-quality characters that will be seen on-screen in future movies. Also, the library itself was easy to use and integrate into our existing codebase."*

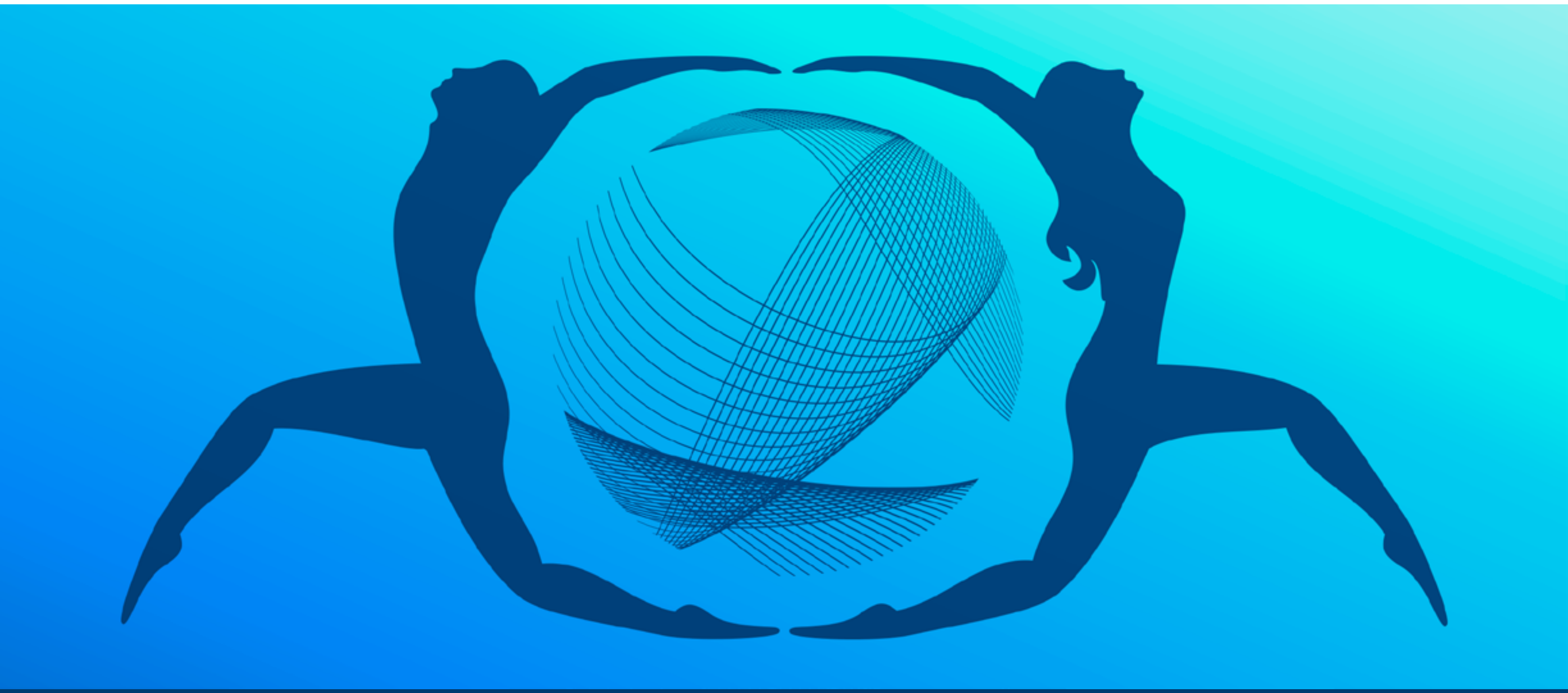
Martin Watt, DreamWorks Animation

Code samples and compiler documentation:

1. **Averaging Filter implementation using Intel® SDLT**
2. **SDLT Code examples**
3. **More Intel SDLT code examples**

References

1. **"Get a Helping Hand from the Vectorization Advisor,"** ***The Parallel Universe*** **Issue 23.**
2. **Introduction to the Intel® SIMD Data Layout Templates (Intel® SDLT).**
3. **How to Manipulate Data Structure to Optimize Memory Use on 32-Bit Intel® Architecture.**
4. **Case Study: Comparing Arrays of Structures and Structures of Arrays Data Layouts for a Compute-Intensive Loop.**



INTEL® C++ COMPILER STANDARD EDITION FOR EMBEDDED SYSTEMS WITH BI-ENDIAN TECHNOLOGY

Avoid Legacy Platform Lock-in from Big-E Platform and Software Dependencies and Easily Migrate to Little-Endian Architectures

Kittur Ganesh, *Software Technical Consulting Engineer,* Intel Corporation

Intel® C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology is a productivity tool for developers locked into legacy platforms due to big-endian software dependencies who want to easily migrate their legacy applications from big-endian to little-endian architectures. Rather than rewrite the entire application code base, developers can speed migration with reduced code changes, code reuse, and one code base for both big-endian and little-endian architectures.

The Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology also offers the performance benefits of the Intel® C++ Compiler, delivering outstanding application performance.

This article presents a comprehensive overview of the key features including the top issues to watch for when porting legacy applications to little-endian architecture systems.

Overview

In the early to mid-1990s, big-endian RISC architectures (e.g., SPARC*, MIPS*, PowerPC*) dominated the embedded market segment and were used widely in computer networking, telecommunications, set-top boxes, DSL, cable modems, etc. In fact, most applications were developed for these systems. However, with the release of several new generations of faster, more scalable versions of x86 architecture processors, many developers faced a dilemma on how best to port these applications to the little-endian x86 architecture systems.

Although the applications are developed in standardized high-level languages (C/C++), traditional compilers can compile only to a single-endian architecture. Porting from one endian architecture to another becomes problematic if there are endianness byte-order dependencies in the source code bases, since it can result in severe runtime issues when executed on the target architecture. To address these dependencies, developers have to locate the hard-to-find, byte-order-dependent code and manually convert it to endian-neutral code—or satisfy the endianness of the target architecture.

If performance is a higher priority than time to market (TTM), migration through manual conversion can be very error-prone and expensive. If the application is already endian-neutral, it may just be a matter of recompiling the application, such as the EEMBC* benchmark samples, which are endian-agnostic.

For applications that contain endianness dependencies, using the **Intel C++ Compiler** Standard Edition for Embedded Systems with Bi-Endian Technology can result in correct code generation while maintaining higher performance. In addition, porting is all about handling data endian types such as function prototypes and variable declarations, while the manual porting process involves handling bit-field accesses and shift operations, etc., which is tedious.

Unlike the Intel C++ Compiler, the Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology enables developers to compile and build applications with the byte order semantics in the source code bases intact, as long as the intended endianness of the byte order of dependent data is specified using the language extensions, a part of the compiler usage model (described later in this article). The compiler inserts byte-swap instructions where necessary on the designated byte-order-dependent code so that the native endian architecture data in memory is converted into the target endian architecture byte order semantics for proper runtime execution.

The Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology is a command-line, Linux* hosted, Linux target, stand-alone compiler specifically designed to enable developers to migrate applications from big-endian to little-endian architectures.

Benefits include:

- **Faster and easier migration.** Enables faster migration to little-endian Intel® architecture.
- **Minimal code changes.** Instead of requiring a rewrite of the entire application code base, the compiler allows reuse of the big-endian code base with minimal code changes, reducing implementation and validation effort.
- **Common code base.** Allows maintaining one code base for both big-endian and little-endian architectures.

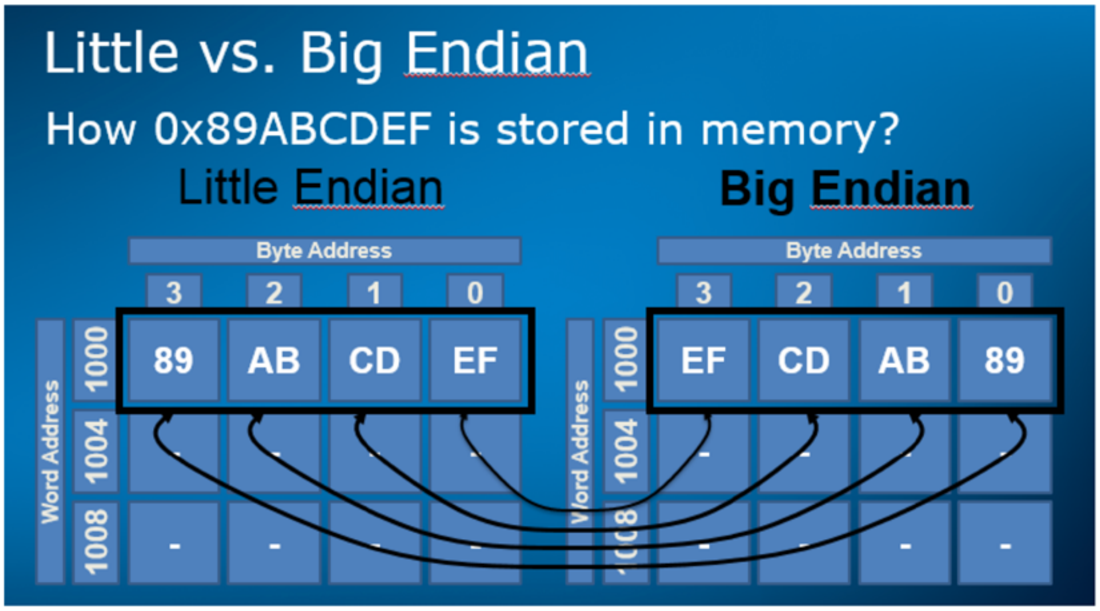
The Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology is similar to the Intel C/C+ Compiler for Linux* and shares a common core of technology related to optimization and performance models. The key difference between the compilers is the bi-endian technology, which is supported only in the Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology.

Usage model key features are outlined in the subsequent sections of this article through easy-to-understand code snippets, including some common issues to watch for during the porting process. Understanding these key features should help developers quickly get started with the Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology to seamlessly migrate legacy applications to Intel architecture.

A Solution for Byte Endianness and Byte-Order Basics

Byte endianness is the system architecture attribute specifying the representation of data layout in memory for multiple byte access. The little-endian variables are stored with the least significant byte in the lowest memory byte address location, while the big-endian variables are stored with the least significant byte in the highest memory byte address. In this context, the big-endian model means that both big-endian and the little-endian data are allowed in memory accordingly.

Figure 1 shows the representation of the data 0x89ABCDEF in memory, with the big-endian variables using the opposite ordering of bytes within memory compared to little-endian variables, which have the least significant byte (EF) stored in the highest memory byte address.

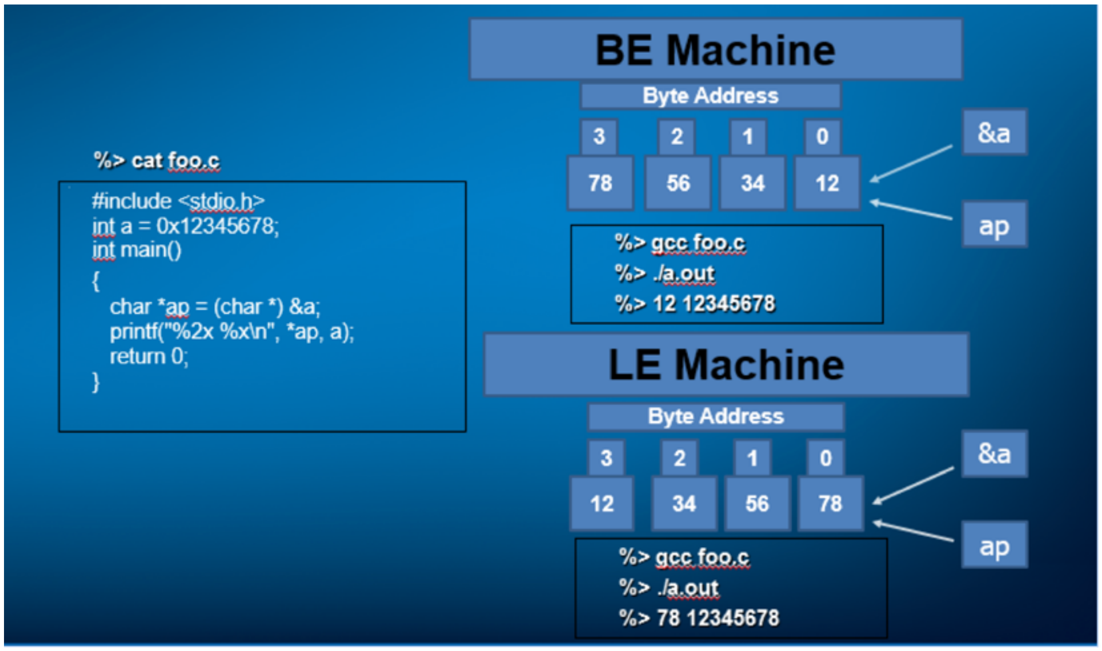


1 Little vs. big-endian

Figure 2 shows a sample code snippet containing byte-order dependency, which produces different results when executed on big-endian versus little-endian architecture systems. As noted, on the little-endian system, the pointer ap points to 78, the least significant byte, while it points to 12 on a big-endian system. To fix this issue, the programmer can manually transform the byte-order code into endian-neutral code or reflect the endianness of the target architecture. But the manual process consumes enormous time and effort to identify byte-order-dependent code in legacy applications consisting of millions of lines of code—and can be error-prone as well.

Figure 3 shows the Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology solution to the endianness dependency issue in **Figure 2**. The programmer needs to only identify the byte order of the data using the language construct (big-endian switch) for the compiler to enforce the big-endian byte order semantics—producing identical results on both architectures.

In the code snippet in **Figure 3**, value "a" is stored in big-endian format independent on which system the code is executed. On little-endian systems, the value "a" is byte-swapped by the compiler when necessary. The Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology inserts the byteswap (bswap) instruction to emulate the big-endian execution only when necessary, to minimize the generated instructions whenever possible to reduce performance overhead.

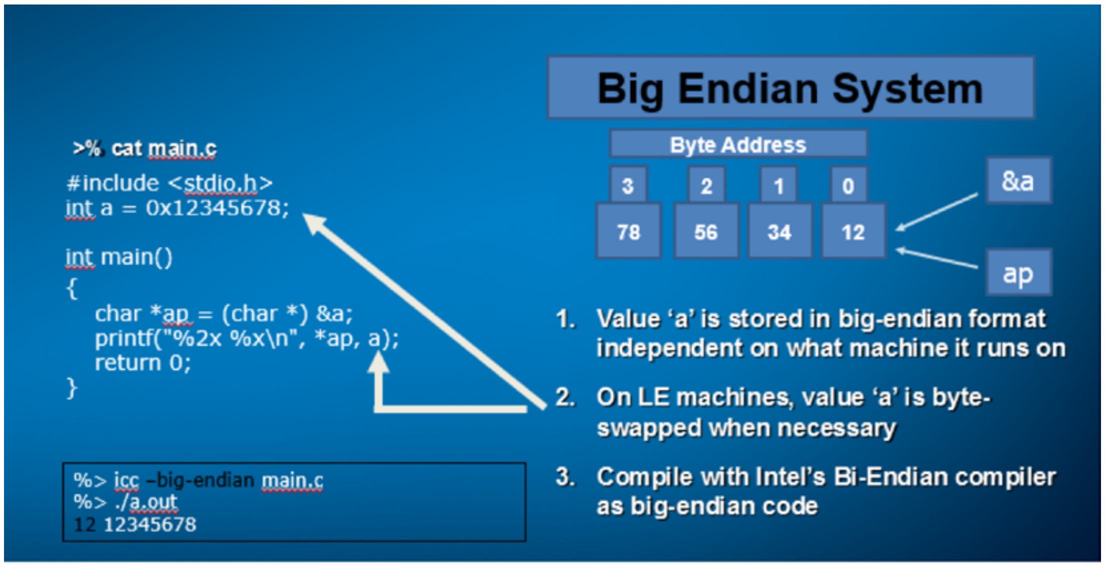


```
%> cat foo.c
#include <stdio.h>
int a = 0x12345678;
int main()
{
  char *ap = (char *) &a;
  printf("%2x %x\n", *ap, a);
  return 0;
}
```

```
%> gcc foo.c
%> ./a.out
%> 12 12345678
```

```
%> gcc foo.c
%> ./a.out
%> 78 12345678
```

2 Byte-order-dependent code execution



```
>% cat main.c
#include <stdio.h>
int a = 0x12345678;

int main()
{
   char *ap = (char *) &a;
   printf("%2x %x\n", *ap, a);
   return 0;
}
```

```
%> icc –big-endian main.c
%> ./a.out
12 12345678
```

1. Value 'a' is stored in big-endian format independent on what machine it runs on
2. On LE machines, value 'a' is byte-swapped when necessary
3. Compile with Intel's Bi-Endian compiler as big-endian code

3 Identical results on big-endian and little-endian systems

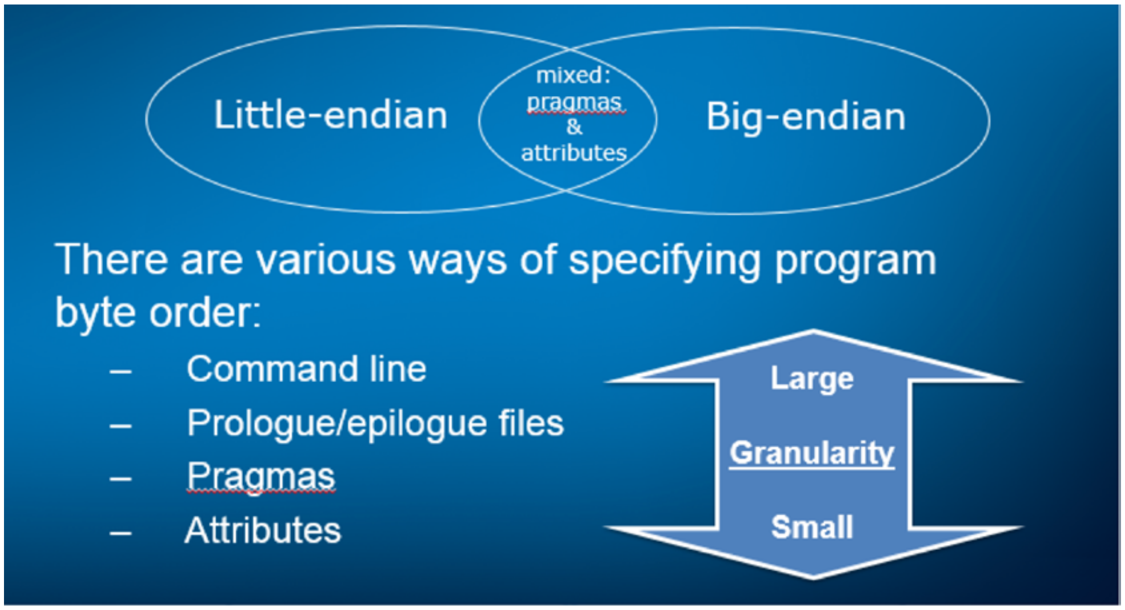
Usage Model and Language Support

The Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology provides bi-endian language extensions, enabling the developer to indicate the byte order (a type attribute) of data types including built-in types or typedef, code regions, and variable declarations. These language extensions enforce appropriate byte-order semantics on the assigned data types during execution on the target architecture.

As shown in **Figure 4**, the Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology provides various ways of designating the program code byte order, such as command line switches, prolog/epilog files mechanism, pragmas, and attributes support.

The Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology resolves the data accesses using big-endian or little-endian conventions as supported through the implicit and explicit usage models. The implicit model is designed to designate the byte order for a file or an entire application through the use of command line options such as –big-endian or –little-endian switches or using the prolog and epilog files mechanism to set the implicit endian mode for the entire file or directories as well. It is advisable to use this implicit model where possible during the migration process for simplifying and accelerating the porting process.

The explicit model supports mixed-endian development through pragma byte-order attributes features. The programmer can designate the byte order of variables and functions at varying levels of granularity. The explicit model, due to this control on granularity levels, enables mixed-endian development.

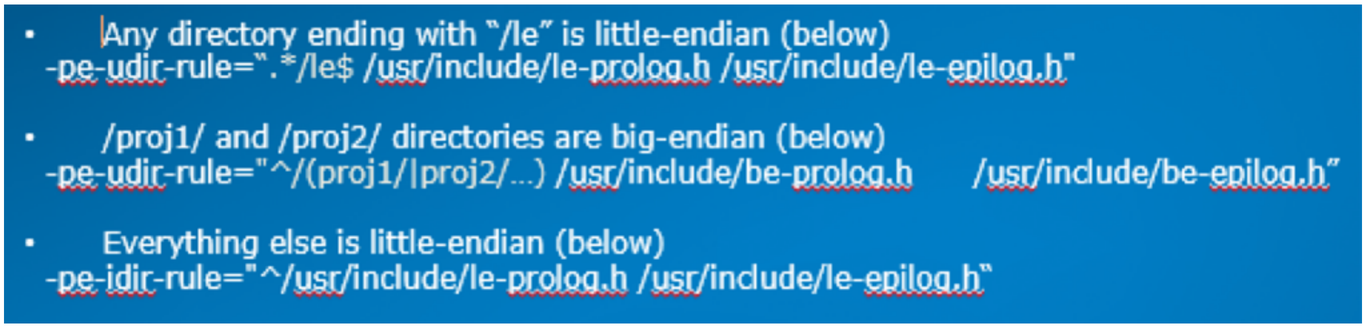


There are various ways of specifying program byte order:

- Command line
- Prologue/epilogue files
- Pragmas
- Attributes

4 Mixed-endian code development

Figure 5 shows a a code snippet sample specifying endian byte order for directories using the prolog and epilog switches. Likewise, **Figure 6** shows a code sample using the command line switch –big-endian, #pragma byte_order and attribute features at a finer granularity through explicit declarations. The command line option –big-endian sets the endian order for the whole program to big-endian byte order at a higher granularity. The #pragma byte_order (push, little-endian) specifies that all the declarations following the # pragma are of little-endian byte order including any include files, except for the variable b, which is specified as big-endian through the endian attribute applied to a specific variable. Similarly, the developer can assign the byte order to different data types such as float, arrays, structures, and structures with bit-fields.

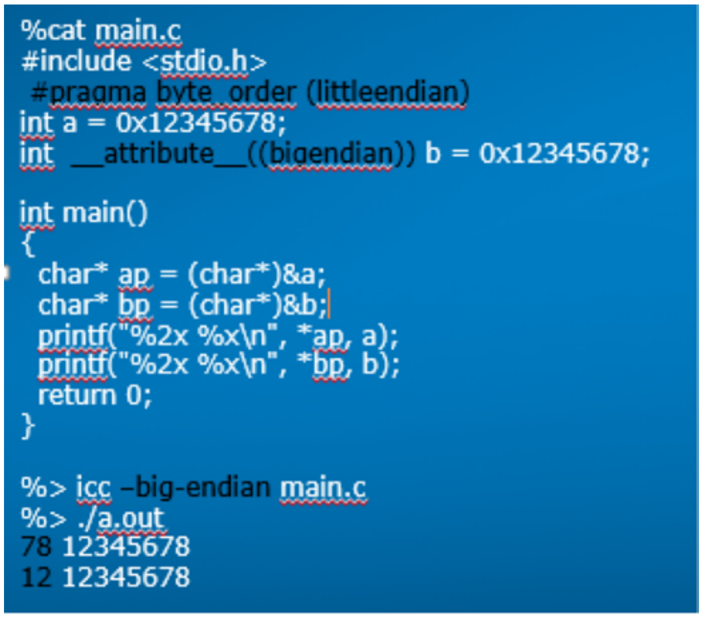


- Any directory ending with "/le" is little-endian (below)
  ```
  -pe-udir-rule=".*/le$ /usr/include/le-prolog.h /usr/include/le-epilog.h"
  ```
- /proj1/ and /proj2/ directories are big-endian (below)
  ```
  -pe-udir-rule="^/(proj1/|proj2/...) /usr/include/be-prolog.h      /usr/include/be-epilog.h"
  ```
- Everything else is little-endian (below)
  ```
  -pe-idir-rule="^/usr/include/le-prolog.h /usr/include/le-epilog.h"
  ```

5 Implicit mode using prolog and epilog feature

In general, the bi-endian functionality in the Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology can be summarized as follows: If the code is specified as big-endian, the compiler will treat the code as though it executes on the big-endian system and insert byte-swap instructions (bswap) at crucial points and, where possible, eliminate code to increase performance. Likewise, code specified as little-endian will run as native code, and the compiler will not insert any bswap instructions, since they are not necessary. The programmer can mix little-endian and big-endian code in the application.

Figure 6 shows a code sample involving incompatible endian data type (msg) mismatch. To fix this issue, the user can use the –symcheck option, which then invokes the *bepostld* tool to perform a type match of all symbols used in the application. If it detects a type mismatch, it prints an error message with detailed information useful for the developer to fix the error. The *bepostld* tool is the bi-endian post link utility. It enables the programmer to identify incompatible type definitions for global variables and functions, in addition to performing the initialization of static big-endian pointers.



```
%cat main.c
#include <stdio.h>
 #pragma byte_order (littleendian)
int a = 0x12345678;
int  __attribute__((bigendian)) b = 0x12345678;

int main()
{
  char* ap = (char*)&a;
  char* bp = (char*)&b;
  printf("%2x %x\n", *ap, a);
  printf("%2x %x\n", *bp, b);
  return 0;
}

%> icc –big-endian main.c
%> ./a.out
78 12345678
12 12345678
```

6 Byte order attribute usage at a finer granularity

Adopting Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology

As mentioned earlier, the 1990s saw many applications for big-endian architectures like SPARC, MIPS, and PowerPC. Many of these legacy applications are still used as part of the mixed-endian development environment. Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology can help accelerate the porting of old legacy code to Intel architecture systems.

Figure 7 shows a typical use case in such an embedded environment.

Embedded Operating System
- System library and GNU* Libc (little-endian)
- HW interface (little-endian)
- Routing code in the user space (big-endian)
- Bi-Endian Interface (Linux* OS and Libc)
    - System/libc calls are through an abstraction layer
    - Wrapper for used Libc and system service calls (customized)

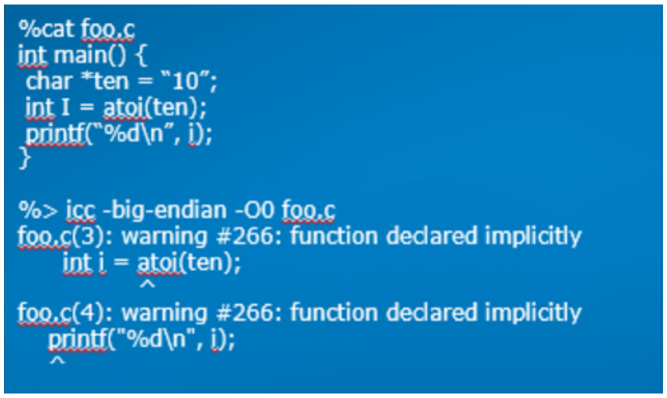
Bare metal
- Own operating system and routing code (big-endian)
- HW Interface (little-endian)
- Bi-Endian interface is through the HW interface

7 Typical usage example

Top Issues to Watch For

Porting large legacy applications can be easier if the endianness of the directories is set correctly, eliminating all warnings on byte-order mismatch reported by the compiler, getting rid of endianness consistency errors reported by the `bepostld` tool across modules, and ensuring correct implementation of the function prototypes. Common issues during the migration process include:

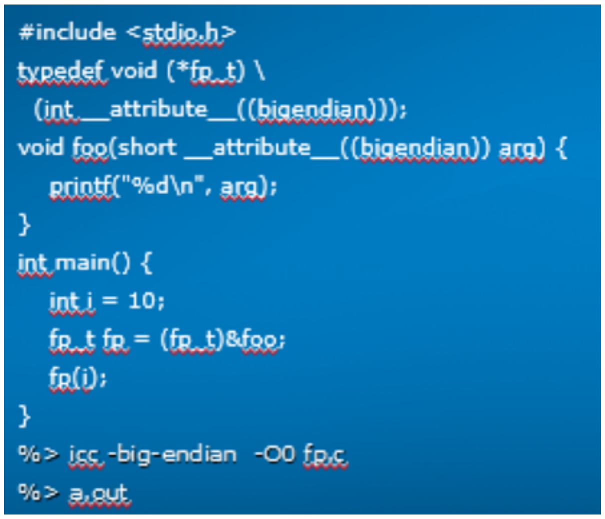
1. **Missing prototype. Figure 8** shows an example. When the sample is executed, the program will crash, since the compiler has no indication that atoi() and printf() are little-endian and therefore incorrectly passes byte-swapped arguments. **Solution:** Add stdio.h and stdlib.h include files.



```
%cat foo.c
int main() {
 char *ten = "10";
 int I = atoi(ten);
 printf("%d\n", i);
}

%> icc -big-endian -O0 foo.c
foo.c(3): warning #266: function declared implicitly
      int i = atoi(ten);
              ^
foo.c(4): warning #266: function declared implicitly
    printf("%d\n", i);
    ^
```

8 Missing prototype

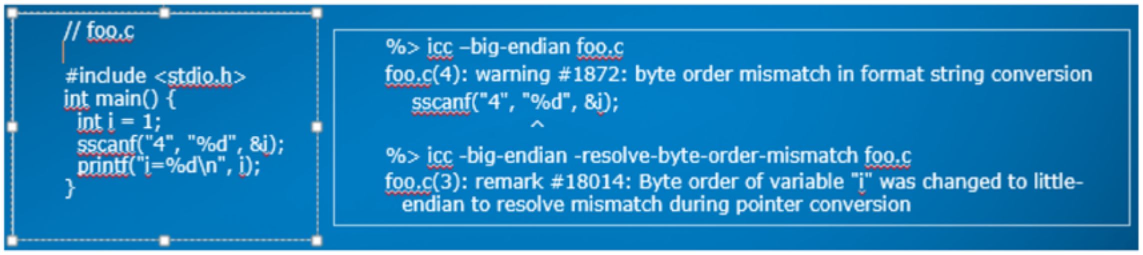
2. **Wrong casting. Figure 9** shows a sample code snippet where the formal and actual arguments have different sizes. **Solution:** Compile with –param-byte-order=little-endian.



```
#include <stdio.h>
typedef void (*fp_t) \
  (int __attribute__((bigendian)));
void foo(short __attribute__((bigendian)) arg) {
    printf("%d\n", arg);
}
int main() {
    int i = 10;
    fp_t fp = (fp_t)&foo;
    fp(i);
}
%> icc -big-endian  -O0 fp.c
%> a.out
```

9 Wrong casting

3. **Pointed-to type mismatch. Figure 10** shows type mismatch as sscanf() assigns little-endian value in big-endian var 'i'. **Solution:** Declare 'i' as little-endian; swap value inside a wrapper for 'sscanf'. Or compile with –resolve-byte-order-mismatch to let the compiler fix the problem automatically.

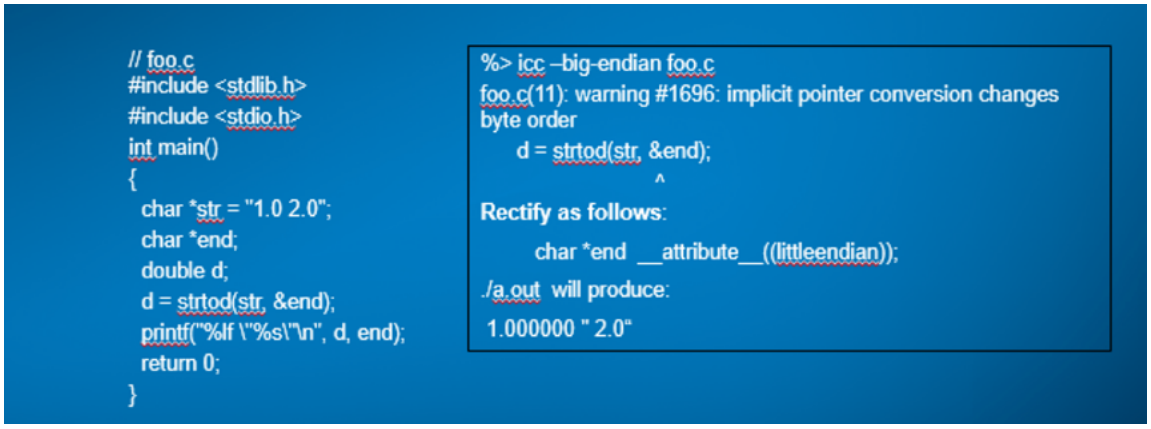


```
// foo.c

#include <stdio.h>
int main() {
  int i = 1;
  sscanf("4", "%d", &i);
  printf("i=%d\n", i);
}
```

```
%> icc –big-endian foo.c
foo.c(4): warning #1872: byte order mismatch in format string conversion
    sscanf("4", "%d", &i);
                      ^

%> icc -big-endian -resolve-byte-order-mismatch foo.c
foo.c(3): remark #18014: Byte order of variable "i" was changed to little-
   endian to resolve mismatch during pointer conversion
```

10 Pointed-to type mismatch

4. **Missing byte order on the pointer type. Figure 11** shows a sample code snippet involving missing byte order on the pointer. The solution is to assign the correct endianness of little endian to the "end" pointer using the attribute feature as shown in **Figure 11**.

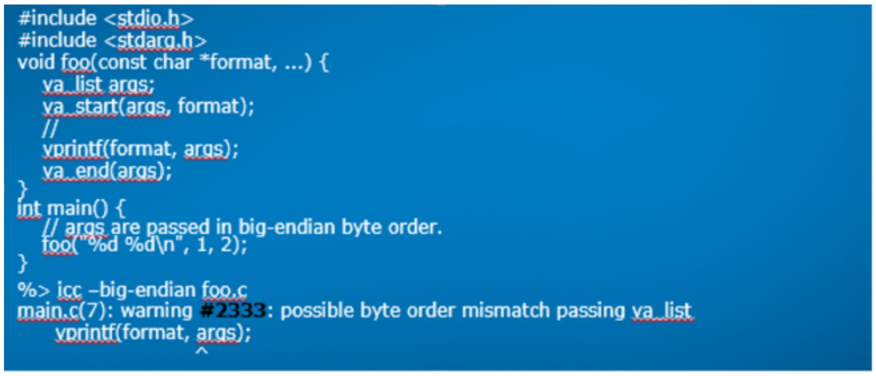


```
// foo.c
#include <stdlib.h>
#include <stdio.h>
int main()
{
  char *str = "1.0 2.0";
  char *end;
  double d;
  d = strtod(str, &end);
  printf("%lf \"%s\"\n", d, end);
  return 0;
}
```

```
%> icc –big-endian foo.c
foo.c(11): warning #1696: implicit pointer conversion changes
byte order
     d = strtod(str, &end);
                     ^
Rectify as follows:
       char *end __attribute__((littleendian));
./a.out  will produce:
 1.000000 " 2.0"
```

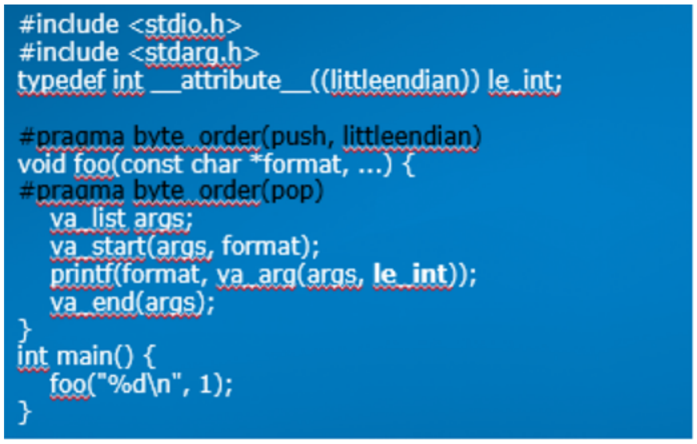
11 Missing byte order on pointer type

5. **Variable length arguments. Figure 12** shows a code snippet involving variable length arguments. vprintf() retrieves vargs in little-endian byte order, but 'foo' is called with big-endian args, resulting in byte-order mismatch. Note that the compiler doesn't automatically adjust the byte order of arguments passed via va_list. Also, the variable length arguments should be passed and retrieved in the same byte order (**Figure 13**).



```
#include <stdio.h>
#include <stdarg.h>
void foo(const char *format, ...) {
   va_list args;
   va_start(args, format);
   //
   vprintf(format, args);
   va_end(args);
}
int main() {
   // args are passed in big-endian byte order.
   foo("%d %d\n", 1, 2);
}
%> icc –big-endian foo.c
main.c(7): warning #2333: possible byte order mismatch passing va_list
     vprintf(format, args);
     ^
```

12 Variable-length args



```
#include <stdio.h>
#include <stdarg.h>
typedef int __attribute__((littleendian)) le_int;

#pragma byte_order(push, littleendian)
void foo(const char *format, ...) {
#pragma byte_order(pop)
   va_list args;
   va_start(args, format);
   printf(format, va_arg(args, le_int));
   va_end(args);
}
int main() {
   foo("%d\n", 1);
}
```

13 Variable-length args (continued)

6. **main().** A few key points to note regarding the `main()` function arguments and byte order:

• Arguments to main are treated as little-endian by default:

```
typedef __attribute__((littleendian)) int leint;
int main( leint argc, char __attribute__((littleendian)) **argv )  { }
```

• If big-endian arguments are required:

```
typedef __attribute__((bigendian)) int beint;
typedef  __attribute__((bigendian)) char  *pIntel C++ Compiler Standard
   Edition for Embedded Systems with Bi-Endian Technologyhar;
int main (beint argc,  pIntel C++ Compiler Standard Edition for Embedded
   Systems with Bi-Endian Technologyhar *argv __attribute__((bigendian)) {}
```

Note: Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology will not diagnose or warn for the above, as big-endian type is explicitly set in the code.

7. **Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology porting tips:**

    - Ensure correct endianness of files and directories.
    - Eliminate all warnings on byte-order mismatch.
    - Get rid of the errors reported by the `bepostld` utility.
    - Watch out for big-endian issues caused by size mismatch (16 versus 32), variable arguments, function pointers, type casting, etc.
    - Ensure correct function prototypes with proper endianness byte order.

Conclusion

Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology is a productivity tool for developers looking to overcome platform lock-in due to big endian software dependencies and migrate large legacy applications to little-endian architectures. We discussed some top issues to watch for when porting byte-order-dependent applications from big-endian to little-endian architectures. Also, the Intel C++ Compiler Standard Edition for Embedded Systems with Bi-Endian Technology shares a common core of technology related to optimization and performance models with the classic Intel C/C+ Compiler for Linux—delivering outstanding performance.

OPENMP* API VERSION 4.5: A STANDARD EVOLVES

Support for Heterogenous Programming with Easier Parallel Execution of Loops

Michael Klemm, *Senior Application Engineer,* Intel Corporation
and Christian Terboven, *HPC Group,* RWTH Aachen University

The OpenMP* API specification is a well-known and widely used standard for multithreading on shared-memory systems. Version 4.5 is the next step in the standard's evolution, introducing new concepts for parallel programming as well as additional features for offload programming.

History

The previous version of the OpenMP API specification (4.0) was released in July 2013. A major feature added was basic support for heterogeneous programming by offloading computation from the host to coprocessors. Due to the growing demand of the high-performance computing community for compute power, coprocessing gained a lot of interest. It became clear that OpenMP 4.0 lacked critical features to support heterogeneous programming optimally. At the same time, programmers are always required to look for new parallelization opportunities and to better express parallelism in their codes.

OpenMP 4.5 strives to improve offloading to coprocessors, as well as to add useful features to make **parallel programming** easier. In this article, we will describe three key new features of OpenMP 4.5:

- Task-generating loops for easier parallel execution of loops
- Locks with hints to expose more parallelism when using locks
- Improvements to offloading to make use of coprocessors in the system

A German version of this article was published in the online magazine *Heise Developer*.

Task-Generating Loops

Parallel loops are one of the most important parts of OpenMP applications. The traditional worksharing constructs (for in C/C++ and do in Fortran) are very simple ways of expressing thread-parallel loops by dividing loop iterations into chunks and distributing them across the threads in a parallel team. However, these constructs have several restrictions that complicate the life of a parallel programmer when dealing with large application codes. One of the most problematic issues is that nesting worksharing constructs in other worksharing constructs is prohibited by OpenMP. Because of that, one cannot nest a parallel loop inside another without creating a new team of threads to execute the inner parallel loop.

```
void taskloop_example() {
#pragma omp taskgroup
    {
#pragma omp task
        long_running_task()  // can execute concurrently

#pragma omp taskloop collapse(2) grainsize(500) nogroup
        for (int i = 0; i < N; i++)
            for (int j = 0; j < M; j++)
                loop_body();
   }
}
```

1 Example code using the new taskloop construct with an OpenMP* task

The new taskloop construct solves the problem by using OpenMP tasks to execute the loop chunks instead of assigning them directly to worker threads. Since OpenMP tasks can be nested arbitrarily, none of the nesting restrictions of worksharing constructs apply. **Figure 1** shows how to mix OpenMP tasks and the taskloop construct. The OpenMP task in the example calls the function long_running_task() to mimic some long-running activity that can execute concurrently with the following loop. The taskloop construct then cuts the loop's iteration space into chunks and creates one OpenMP task for each chunk. Any free worker thread picks up one of these tasks and executes it. If the long-running task finishes early, its executing thread can also pick up tasks created by the taskloop. Because tasks may be nested arbitrarily, each of the executing tasks may, in turn, create new ones to further increase the level of concurrency in the application. The OpenMP runtime system takes care of load balancing the tasks across workers until all tasks have been executed.

```
template<class K, class V>
struct hash_map {

  hash_map() {
    omp_init_lock(&lock);
  }

  ~hash_map() {
    omp_destroy_lock(&lock);
   }

  V& find(const K& key) const {
    V* ret = 0;
    omp_set_lock(lock);
    ret = internal_find(key);
    omp_unset_lock(lock);
    return *ret;
  }

  void insert(const K& key, const V& value) {
    omp_set_lock(lock);
    internal_insert(key, value);
    omp_unset_lock(lock);
  }
  //...

private:
  mutable omp_lock_t lock;
  hash_buckets *buckets;
  // ...
};
```

2 Example of a hash map to map keys to values

The taskloop construct inherits its syntax from both the worksharing constructs and the tasking constructs. Besides the usual clauses to control visibility and sharing of data (shared, private, firstprivate, and lastprivate), it also supports the task clauses (final and mergeable). In addition, the construct accepts the nogroup clause that deactivates the implicit task group that automatically synchronizes all tasks created by the construct. In the example, we use an explicit taskgroup construct to synchronize execution of the long-running task and the taskloop construct.

The size of the generated tasks can be controlled by the grainsize clause. It defines how many loop iterations are assigned to each created task. If the programmer prefers to specify the number of tasks, the num_tasks clause can be used instead. Finally, collapse can be used to create a product loop out of perfectly nested loops. This is similar to what the collapse clause achieves for worksharing constructs.

OpenMP 4.5 also extends the OpenMP task constructs `task` and `taskloop` with the `priority` clause. It can be used to influence the order of task execution by the runtime system. The clause takes a positive integer value: the higher the value, the higher the priority of the created task. It is used as a hint to the runtime system to suggest that it should execute tasks with higher priority before tasks with lower priority. However, the OpenMP implementation is not required to adhere to the hint, so one cannot assume that using priorities guarantees a certain ordering of task execution. If the programmer does not specify any priority, it is assumed to be zero by default.

Locks, Locks, Locks

Mutual exclusion by acquiring and releasing locks is an inevitable evil in many parallel programs. To avoid race conditions and data corruption, locks usually have to be taken before entering a code region that accesses a shared resource. This protection of the resource comes with a price to pay in terms of serializing execution and thus limiting parallelism. In some applications, locks are placed for safety reasons, despite the fact that the probability of a conflict due to concurrent access to the shared resource is very low, but not zero.

```
template<class K, class V>
struct hash_map {

   hash_map() {
      omp_init_lock_with_hint(&lock,
         kmp_lock_hint_hle);
   }

   ~hash_map() {
      omp_destroy_lock(&lock);
   }

   V& find(const K& key) const {
      V* ret = 0;
      omp_set_lock(lock);
      ret = internal_find(key);
      omp_unset_lock(lock);
      return *ret;
   }

   void insert(const K& key, const V& value) {
      omp_set_lock(lock);
      internal_insert(key, value);
      omp_unset_lock(lock);
   }

private:
     mutable omp_lock_t lock;
     hash_buckets *buckets;
};
```

3 Using speculative locks to optimistically execute a critical region

The example code in **Figure 2** shows a very simplistic and inefficient implementation of a hash map used to map a key of type `K` to a value of type `V`. Despite being a short example without error handling or other optimizations, the code shows one particular locking issue. The mutual exclusion is activated by acquiring the lock immediately after entering each of the methods of the `hash_map` class. The effect is that execution is serialized to avoid potential race conditions on the individual hash buckets. However, this is frequently unnecessary, as data structures with hash functions are designed to avoid access conflicts as much as possible. It would thus be perfectly safe for multiple threads to enter the hash map code if each thread works on a different hash bucket or element. In typical implementations you will find locks to protect individual buckets or elements, or even lock-free data structures. In any case, getting to a scalable and efficient solution requires a lot of engineering work.

OpenMP 4.5 offers a new feature to alleviate this burden. Programmers can use a new API to pass a hint for each lock to the OpenMP runtime and inform it about the intended usage of the locks in the application code. Two new functions to initialize a lock are defined: `omp_init_lock_with_hint` and `omp_init_nest_lock_with_hint`. These functions accept an additional argument of type `omp_lock_hint_t` (see **Table 1** for possible lock hints). As with other hints in OpenMP, the implementation can use the hint to optimize the lock implementation for the intended usage. For instance, the implementation can then replace a test-and-set lock with a lock based on futexes (short for "fast userspace mutex"). Or it can make use of special hardware instructions, e.g., Intel® Transaction Synchronization Extensions (Intel® TSX). In all circumstances, the lock semantics are preserved so that the observable behavior of the program is not affected.

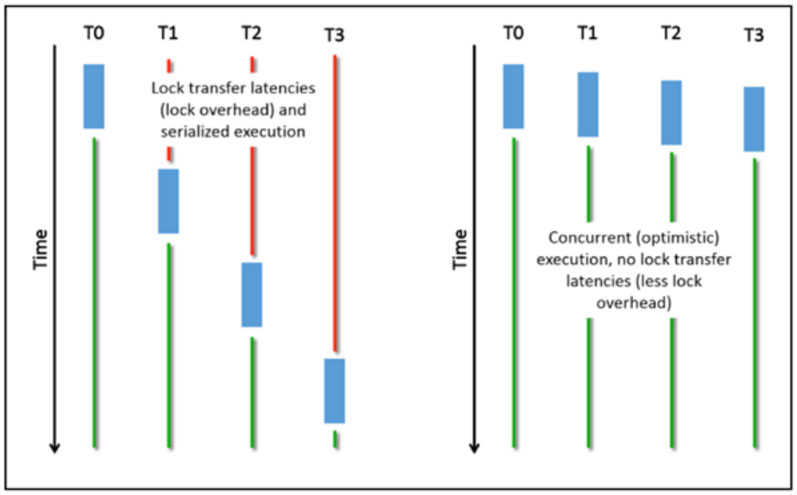
The hints are integer expression, so that they can be combined by the | operator in C/C++ or by the + operator in Fortran. This gives a programmer more flexibility in expressing the desired usage of a particular lock. The OpenMP standard also explicitly allows an implementation to extend the predefined hints with additional hints. The Intel OpenMP runtime makes use of this by defining the additional hints listed in **Table 2**.

Hint	Semantics
omp_lock_hint_none	The OpenMP runtime can freely choose the lock implementation.
omp_lock_hint_uncontended	Threads rarely access the lock concurrently, so expected contention is low.
omp_lock_hint_contended	Optimize the lock for frequent conflicts by multiple threads trying to acquire the lock at the same time.
omp_lock_hint_nonspeculative	Do not use optimistic locking; there are too many conflicts due to overlapping working sets of the acquiring threads.
omp_lock_hint_speculative	Use optimistic locking; the working set of the threads is expected to be distinct so that conflicts are unlikely.

Table 1. Supported lock hints in OpenMP* 4.5

Hint	Semantics
kmp_lock_hint_hle	Use hardware lock elision feature of Intel® TSX for the lock.
kmp_lock_hint_rtm	Use a lock implemented with the restricted transactional memory feature of Intel® TSX.
kmp_lock_hint_adaptive	Use a speculative, adaptive lock that checks for contention and falls back to a traditional test-and-set lock in case of too many conflicts.

Table 2. Additional lock hints as defined by the Intel® OpenMP* runtime


T0
T1
T2
T3
Lock transfer latencies (lock overhead) and serialized execution
Time
T0
T1
T2
T3
Concurrent (optimistic) execution, no lock transfer latencies (less lock overhead)
Time

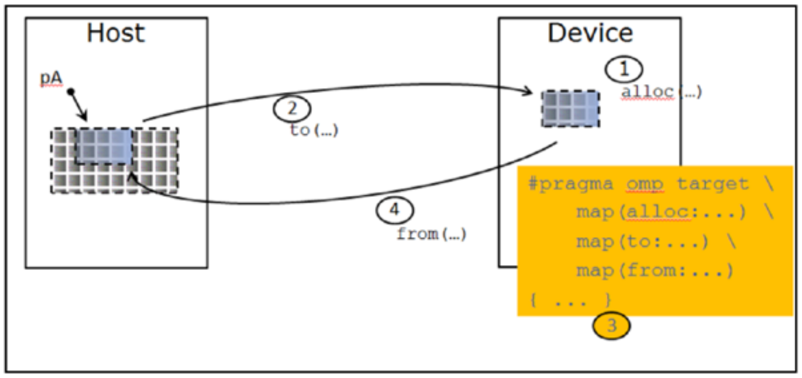
4 Traditional mutual exclusion (left) and optimistic mutual exclusion (right)

In **Figure 3**, the code uses the new feature to pass information to the OpenMP runtime that the lock should be executed speculatively, as the programmer assumes that thread will not conflict while accessing the hash map. On a processor with Intel TSX, the hardware will optimistically ignore the lock instead of acquiring and releasing the lock (see **Figure 4**). Only if the hardware detects that lock semantics have been violated—for instance, if one thread modified a hash bucket while another thread was reading from it—will the processor revert and re-execute code with traditional locking for mutual exclusion.

As a rough guideline for when to use speculative locking, one can consider two fundamental cases. First, Intel TSX and speculative locking pay off if a lock is uncontended and does not produce a lot of conflicts (as in the hash table example). Second, if many readers but only a few writers compete for a lock and thus produce a lot of contention, but almost no conflicts, then a speculative lock can reduce the lock overhead as well.

Offloading

OpenMP 4.0 extended the multithreading paradigm of OpenMP with heterogeneous offloading to attached compute devices such as coprocessors. The `target` construct sends the control flow from a host thread to the coprocessor. Programmers can use the `map` clause to specify data objects to be transferred and the direction of the data transfer (see **Figure 5**). Usually, the offloaded code regions are known as kernels and are massively parallel fragments that make use of the respective properties of the target device.



5 Execution model for offloaded code regions including data transfers

OpenMP 4.0 introduced device data environments that can keep data alive across different offload regions to save expensive data transfers between the host and target. **Figure 6** shows an example of such a data region that keeps `var1` on the target device during the invocation of `kernel1` and `kernel2`. The duration of the data region is bound to the lexical scope structured block associated with the `target data` construct. When the code reaches the opening curly brace, the data environment is created and data is transferred. Once the code reaches the closing curly brace, the data environment is destroyed. While this allows the implementation to keep the environment across multiple kernel invocations, it prohibits the mapping of new data or the mapping of data in an unstructured way, e.g., in the constructor or destructor of the C++ class.

```
double var1[N];

void offload_example() {
#pragma omp target data map(tofrom:var1[:N])
    {

        C *c = new C();

#pragma omp target
        c.kernel1();   // uses var1 and members of C

#pragma omp target
        c.kernel2();   // uses var1 and members of C

        delete c;
    }
}
```

6 Device data environments bound to the lexical scope of the `target data` construct

OpenMP 4.5 provides new constructs to overcome these restrictions. The target enter data and target exit data directives create and destroy data mapping on the target device:

```
#pragma omp target enter data map(map-type: var-list) [clauses]
#pragma omp target exit data map(map-type: var-list) [clauses]
```

Figure 7 shows how the new directives can be used to create and destroy data environments in a C++ object's constructor and destructor. When an instance of the `C` class is constructed, the values member is mapped to the target device. When the object is destroyed, the destructor also disposes of the data environment associated with values.

Another extension concerns the mapping of elements of structured data types. OpenMP 4.0 allowed only scalar variables, arrays, or bitwise copyable data structures to be mapped. OpenMP 4.5 extends data transfers to cover partial mapping of members of structured data types. **Figure 8** shows a few of the new possibilities.

Finally, OpenMP 4.5 contains support for asynchronous offloading. Previously the host thread waited for the completion of the offloaded code region before it continued with execution. With the `nowait` clause, programmers can turn a `target` construct into an OpenMP task that is executed concurrently with the encountering host thread. The `target enter data` and `target exit data` directives also support the new clause to allow for asynchronous data transfers.

Because the asynchronous offloads and data transfers are regular OpenMP tasks, the new feature inherits the `depend` clause to synchronize asynchronous execution with other OpenMP tasks executing on the host. The example code in **Figure 9** performs an asynchronous

```
class C {

 public:
    C() {
#pragma omp target enter data map(alloc:values[M])
    }

    ~C() {
#pragma omp target exit data map(delete:values[M])
    }
 private:
    double *values;
};
```

7 Creating and destroying device data environments in C++ objects

```
struct A {
   int field;
   double array [N];
} a;

#pragma omp target map(a.field)
#pragma omp target map(a.array[23:42])
```

8 Mapping of elements in structured data types

```
double data[N];

void synchronization_example() {
#pragma omp target enter data map(to:data[N]) \
            depend(out:data[0]) nowait

    do_something_on_the_host_1();

#pragma omp target depend(inout:data[0]) nowait
    perform_kernel_on_device();

    do_something_on_the_host_2();

#pragma omp target exit data map(from:data[N]) \
            depend(inout:data[0])

#pragma omp task depend(in:data[0])
    task_on_the_host(data);

    do_something_on_the_host_3();
}
```

9 Asynchronous offloading and data transfers and synchronization with host threads

data transfer that overlaps with host execution. The `depend` clause defers the kernel invocation until the data transfer has completed. The same kind of dependency avoids premature data transfer back from the device, unless kernel execution has finished. Finally, the host thread creates a task that is awaiting completion of the last data transfer, before it starts concurrently to execute the host code in `do_something_on_the_host_3()`.

Conclusion

Besides several corrections, OpenMP 4.5 brings new features that enable programmers to better express parallelism and to increase the performance of both host applications and offloaded code. Task-generating loops relieve programmers from cumbersome solutions to tame load imbalances and resolve composability issues caused by parallel loops. Support for locks with hints provides an easy way for programmers to optimize the locking behavior of their applications and to exploit modern processor support for hardware transactional memory in a portable manner. Finally, the extensions to offloading allow for asynchronous execution to overlap computation and communication on both the host and the attached coprocessors.

INTEL® MPI LIBRARY: SUPPORTING THE HADOOP* ECOSYSTEM

MPI Often Outperforms Hadoop MapReduce in Tasks with Heavy Computations

Mikhail Smorkalov, *Software Development Engineer,* Intel Corporation

For decades, **MPI** has dominated as the model to use in distributed calculations. However, with high-performance computing (HPC) incorporating workloads requiring processing of huge volumes of input data, new approaches and frameworks have appeared. The most popular ones are the Apache Hadoop* MapReduce paradigm[1] in general and the Hadoop software stack (including all tools and frameworks running on top of it) in particular.

Vanilla Hadoop is composed of several modules[2] and implies certain constraints on the scope of problems it can solve efficiently. For example, MapReduce is not good at iterative algorithms since it requires dumping the intermediate results to the storage between the iterations. These shortcomings triggered development of other frameworks on top of Hadoop, e.g., Apache Spark*, Apache Storm*, which allow for efficient in-memory data caching between the iterations. Moreover, YARN*, as a cluster management framework, allows for arbitrary paradigms, not only MapReduce, thus broadening the area of Hadoop applicability to the fields where mostly MPI could be found previously.

While some tasks can be solved much more easily and efficiently using the Hadoop stack, even machine learning that often requires a lot of heavy computations and intensive internode communication gains popularity on the Hadoop platform—although multiple studies[3, 4] demonstrate that MPI outperforms Hadoop frameworks significantly in this sphere. With some, there is a misconception that MPI is just for HPC. This article helps shed some light on benefits and challenges connected with using MPI in the Hadoop ecosystem, complementing or supplementing the tools commonly used in this area.

New HPC Challenges

Historically, high-performance computations dealt with compute-intensive areas, like weather modeling, physics, and chemistry, where input is relatively small but the number of calculations—and sometimes the volume of output data—is huge. MPI is the best fit for such problems, providing a number of communication patterns, and usually MPI implementations, that use hardware capabilities in a highly efficient way.

However, new application domains require creation of data-intensive applications that operate on terabyte and petabytes of input data. Frequently, they do not require a lot of internode communication, so efficient, reliable, and scalable input/output (IO), becomes a very significant aspect. This aspect is addressed by HDFS* in Hadoop.

Another challenge is that the duration of data analytics tasks can be quite significant—making transparent fault management a must. Moreover, streaming tasks may run permanently, so a node crash is inevitable at some point. The Hadoop software stack encapsulates a mechanism for graceful task handover without the need to restart the whole job. Only failed subtasks are handed over to healthy nodes. Also, the Hadoop scheduler does its best to run tasks as close to data as possible, thus securing the best possible data locality.

All these features make the Hadoop platform attractive for data scientists and analysts. And this has stimulated development of multiple analytics frameworks and libraries in the Hadoop ecosystem. These frameworks allow for using many programming languages (e.g., Python*, Java*, Scala*), thus lowering barriers to entry. MPI implementations traditionally provide only C and Fortran bindings required by the standard. It is worth mentioning that some MPI implementations provide Java and Python interfaces.

MPI Repertoire

First, let's consider the rich and comprehensive set of MPI communication patterns that can be used to implement the MapReduce program. Obviously, the Map phase can be implemented via MPI_Scatter(v) functions, while the natural choice for the Reduce phase is either MPI_Reduce[5] or, if the reduction needs all data in place to proceed, MPI_Alltoall(v), followed by some merge operation. Moreover, the MPI implementations usually provide several algorithms for each collective operation and select the optimal one depending on the scale, message size, and hardware architecture. So the main operations used in the Hadoop world have natural analogs in MPI. But what about fault management and data management functionalities, which are vital for the big data world? Does MPI provide something to address those challenges?

Indeed, Hadoop scalability is great due to the data locality aspect that means getting data from the storage closest to the process whenever possible. The HPC world uses parallel file systems accessed via network, thus putting serious requirements on the network bandwidth and load balancing when working with huge input data. However, modern parallel file systems (e.g., Lustre* or GPFS*) are supposed to provide near-linear scalability with increasing the system size. This means they can be used for scalable data management. Moreover, Intel's distribution of Lustre takes care of running Hadoop over the Lustre file system with optimal data access.[6] While it is not always possible to keep absolute data locality when running Hadoop applications, parallel file systems can provide applications with fast parallel access to remote data.

Even in the Hadoop world, people tend to file systems different from HDFS.[7, 8] The advantages come from the parallel nature of those storage systems and high-bandwidth interconnects that are used in HPC clusters. For example,[8] shows that Lustre FS* can provide much faster data access than even local drive access on commodity nodes.

Moreover, MPI-IO includes collective IO operations that can be tuned flexibly. For example, they allow for collective buffering so that the only process on the node is writing/reading to the FS, thus decreasing concurrency when doing IO.

As for fault tolerance, it is definitely one of the strongest advantages of the Hadoop platform. It allows for almost transparent recovery in case of hardware failure. At the same time, fault tolerance is a rather weak side of the MPI standard. Some implementations may allow for writing fault-tolerant MPI programs by following special workflow,[9] but it requires significant efforts from application engineers. User-level fault tolerance that may be introduced in the MPI 4.0 standard also does not introduce any transparent fault management, but at least should make the life of MPI developers easier. The only transparent fault management mechanism available in MPI is check-pointing.[10] It is based on global snapshots, though, implying that absolutely all tasks (even healthy ones) will be restarted from the checkpoint. So, this is one of the few aspects where MPI implementations are obviously inferior to Hadoop—making MPI less than optimal for long-running services such as streaming processing apps.

Another advantage of the Hadoop ecosystem is the wide community that contributes to the Hadoop ecosystem in different areas. A number of tools and technologies provide convenient access to different types of data sources such as files, relational databases, and streams. On the other hand, MPI is a relatively low-level technology focused on performance, so it does not provide a lot of syntax sugar in API. However, initiatives like Intel® Data Analytics Acceleration Library (Intel® DAAL)[11] introduce a way to efficiently implement the whole dataflow, from reading data from different data sources to transformations and calculations on them, including a broad range of algorithms. This may attract developers who previously did not want to bother with implementing all dataflow by themselves but would like to get the performance advantages of MPI.

Also, it is worth mentioning that vanilla Hadoop uses TCP/IP for communication between the nodes in the shuffling phase, making getting benefits from fast interconnects complicated. There are initiatives related to using RDMA for intercommunication,[12] though.

Finally, many people consider the high scalability of Hadoop when selecting the tools for solving their task, but MPI is generally also very good in this. For example, **Intel® MPI Library**'s proven scalability is up to 340,000 processes.[13]

Running MPI in the Hadoop Ecosystem

As the previous sections show, MPI can address some of the challenges of emerging HPC, being a competitor to Apache Spark in certain areas. However, using MPI with Hadoop tools and frameworks is complicated, since HPC and Hadoop used to progress in parallel, following the needs of most common applications in corresponding areas, thus utilizing different ecosystems, including resource managers. Current activities related to merging HPC and Hadoop worlds are mostly focused on moving Hadoop tools into the HPC environment (or even implementing new MapReduce frameworks atop MPI, such as MR-MPI[14]) to utilize performance advantages provided by high-end systems. However, while speeding up data analytics and machine learning applications by running them on HPC clusters is one of the options (and maybe the most promising one),[15] its value is not that obvious for companies that already have Hadoop infrastructure and successfully use it.

Another option is running MPI applications in the Hadoop environment without maintaining two different infrastructures on the cluster or setting up two different clusters.[16] This section describes the mechanism for integrating MPI into the Hadoop ecosystem built on top of Cloudera Distribution Including Apache Hadoop* (CDH*).[17]

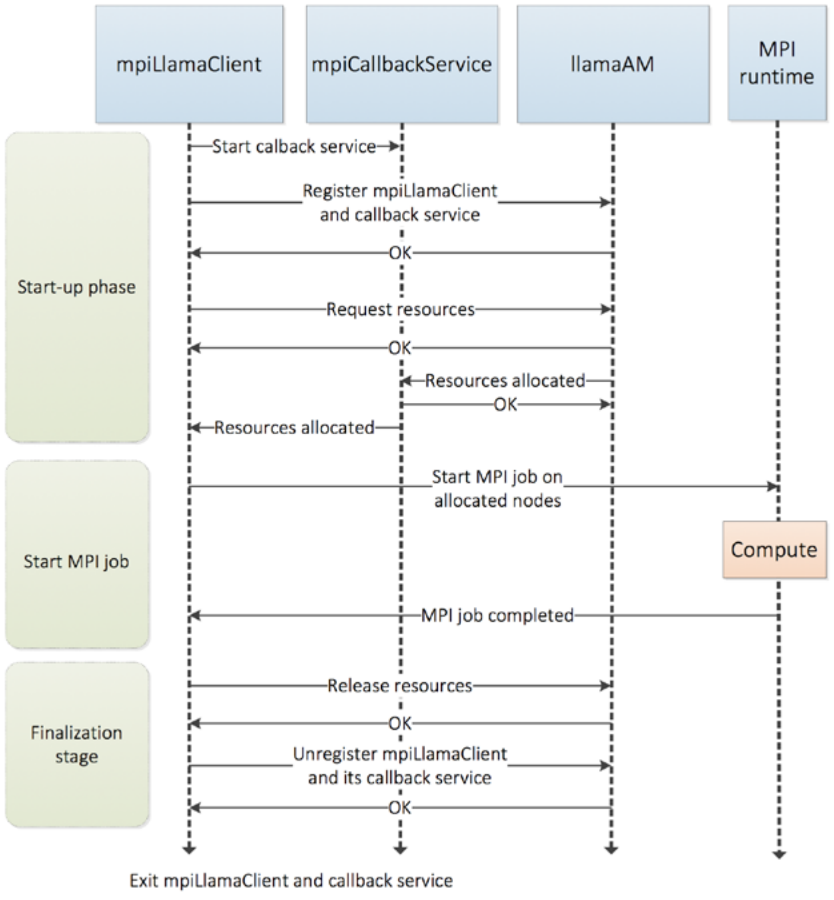
The CDH package includes Llama*,[18] an application master for YARN, originally designed to be used by Impala*.[19] Llama provides a cross-language Apache Thrift* API for requesting and releasing resources and some additional functionality (e.g., gang scheduling, which is vital for running MPI). With this set of features, it becomes possible to programmatically obtain information about Hadoop cluster nodes and request the resources for an MPI job when needed, and thus to share the same infrastructure dynamically between MPI and Hadoop jobs.

To make use of Llama functionality and gracefully run MPI on a Hadoop cluster, two complementary services need to be implemented:

- **MPI Llama client:** Entity that queries Llama to get required information (e.g., cluster node names) and request/release resources.
- **MPI Llama callback service:** Daemon that waits for notification from Llama on certain events (e.g., node allocation).

The actual workflow consists of three independent phases (**Figure 1**):

- **Start-up:** Launching complementary services (client and callback), registering them in Llama, and requesting resources based on MPI job needs. This phase is finished as soon as the callback service gets notification that resources are allocated.
- **MPI job start:** Natively running the MPI job on Hadoop cluster nodes, based on the resource list provided by Llama.
- **Finalization phase:** Releasing resources and shutting down complementary services when the MPI job has finished.



1 Running MPI in a Hadoop* cluster

One obvious advantage of this mechanism is that Llama communication time does not depend on the MPI application complexity, so it just adds constant overhead compared to a pure MPI run. This means that all results related to a performance comparison between MPI and Hadoop/Spark applications are still relevant, since the contribution of Llama communication overhead into the total wall time would be vanishingly small for real applications.

Another advantage of this approach over the others (e.g., the academic mpich2-yarn[20] project) is that it is not tied to a certain MPI implementation, since requesting resources and launching MPI jobs are independent phases of the workflow. Thus, moving to another MPI implementation is a matter of changing MPI launch command. For example, using BDMPI[21] as an MPI implementation provides a way to use MPI for efficient execution of out-of-core algorithms, making it a flexible alternative to native Hadoop frameworks for big data problems.

The functionality described above has been implemented in the Intel® MPI Library 5.1 Update 2 (please see the Intel MPI reference guide for an exact usage model).

It's also important to realize that running MPI on a Hadoop cluster generally imposes some limitations:

- MPI-IO is available on shared folders only or locally, as MPI implementations do not support HDFS. This means the cluster admin may want to set up an NFS folder in addition to HDFS.
- YARN does not provide information about container CPU affinity, so pinning functionality should be used carefully when more than one application is running on the same node.

Performance Evaluation

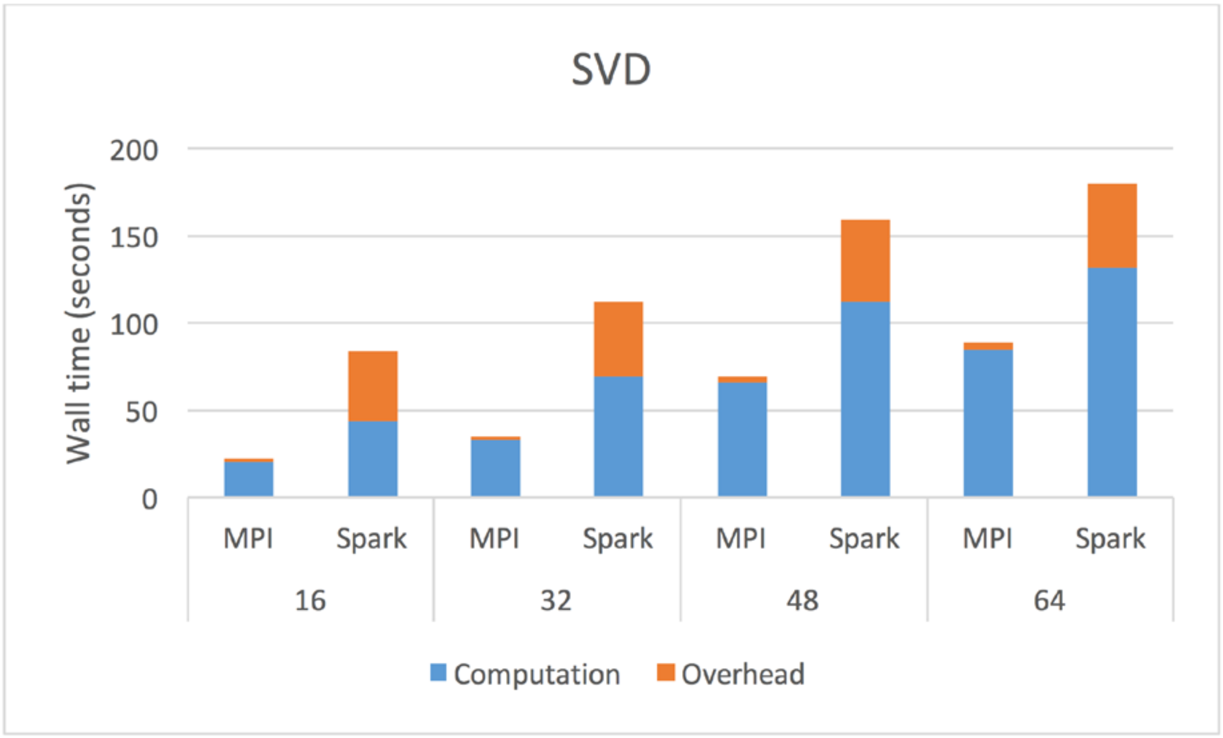
Two distributed algorithms commonly used in data analytics are selected as a target:

1. Singular value decomposition (SVD)
2. Principal component analysis (PCA, using the correlation method)

Spark- and MPI-backed samples for each of the algorithms are taken from the Intel DAAL distribution and composed of the same building blocks, so the performance difference is explained by different distribution frameworks (Spark versus MPI) and languages (Java versus C), but not by implementation details. The Spark-backed SVD sample has also been modified to avoid collecting the left orthogonal matrix on driver node as that is generally excessive and doesn't benefit from the Spark paradigm of RDDs (so the complexity of the Spark-backed algorithm is a bit lower than for the MPI-backed one).

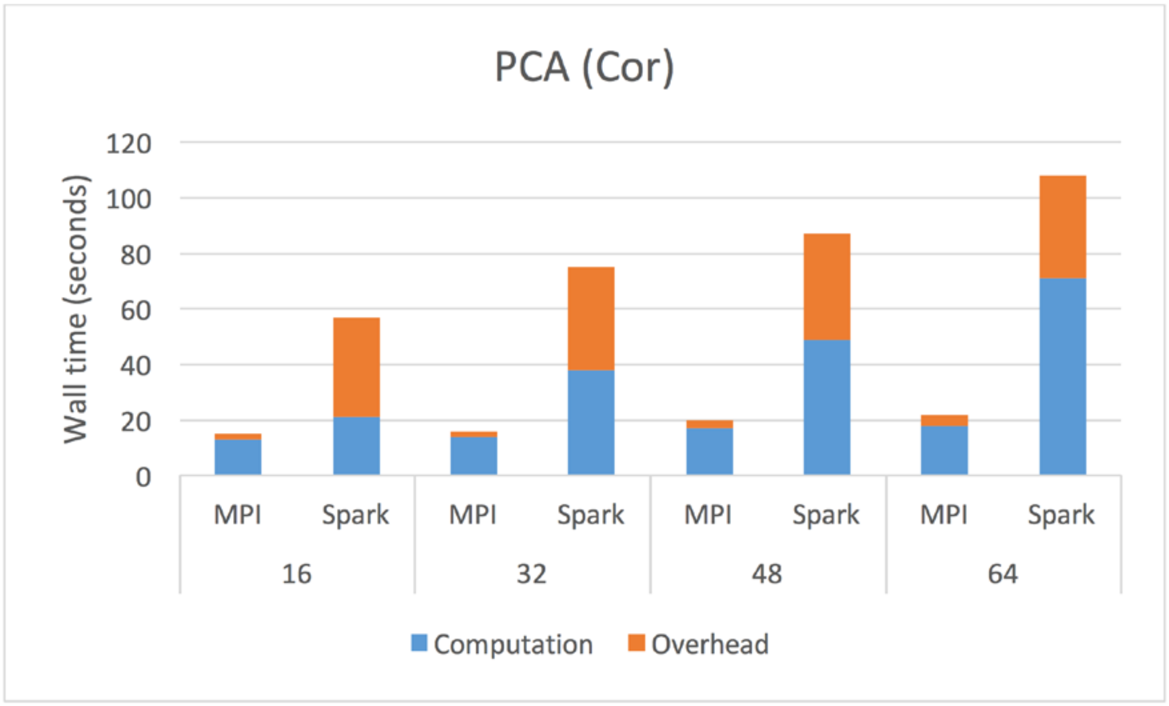
In our performance studies, we used an eight-node cluster in which each node is equipped with one Intel® Xeon® processor X5570, one NetEffect* NE020 10 Gb accelerated Ethernet adapter, and 12 GB of RAM. The cluster was running the SLES* 11.0 Linux* operating system and the Cloudera Express* 5.4.6 version of CDH (Spark v.1.3.0). We used Intel MPI Library Version 5.1.2, Intel DAAL 2016 Update 1, Intel® C++ Compiler Version 15.0.4, JDK v.1.7.0_67, and Scala v. 2.10.4.

Performance was measured on 16, 32, 48, and 64 data blocks, each containing 10,000 x 1,000 elements of the input matrix. The full dataset of 64 blocks was 4.2 GB in size. When running the MPI-backed sample, one MPI process per block was started. For Spark, the desired number of executors was set to the number of blocks via the spark.executor.instances property, which is not always respected since Spark follows its own heuristics when it defines the number of executors. Also, since Spark is quite sensitive to JVM and its own configurations, memory settings were selected empirically (listed under **Figures 2** and **3**) to facilitate the best performance.



2 Singular value decomposition

spark.executor.memory = 1300m
spark.yarn.executor.memoryOverhead = 1024m
spark.kryoserializer.buffer.max.mb = 512m



3 Principal component analysis

spark.executor.memory = 1024m
spark.yarn.executor.memoryOverhead = 768m

Total wall time in the measurements above consists of two summands—actual calculation and the overhead. The latter, for the MPI sample, is defined as the time required for Intel MPI to negotiate the resources with Llama and start the MPI job, and for Spark—as a difference between the wall time and sum of durations of all computation stages.

Note that since MPI is sensitive to running in oversubscription mode, performance may be a bit degraded when the number of MPI ranks is higher than the number of physical cores. However, the wall time is still much shorter than for Spark.

Conclusions and Outlook

The measurements previously mentioned, together with other published results,[3, 4] demonstrate that MPI can be a good fit for some algorithms used in data analytics, and MPI integration into the Hadoop ecosystem proposed in this article allows for broader MPI usage in this area.

It is important to understand that although MPI can significantly outperform and replace Hadoop-based frameworks for some problems, it is not as well suited for others. So, it is more natural to use them together, providing a new level of synergy. Some long-running services may be implemented with Hadoop tools, while relatively short tasks, which involve a lot of computations and tricky communication, can be moved to MPI.

Besides the performance that MPI can provide, there are millions of lines of MPI code created over the years on which Hadoop developers can piggyback. Furthermore, with data analytics breaking into HPC world and emergence of the high-performance data analytics domain, the question of using MPI and big data frameworks in the same ecosystem should get more attention. Integration of MPI into the Hadoop ecosystem is one of the options to address it.

BLOG HIGHLIGHTS

Three Pieces of Advice for Code Modernization Success

BY CLAY BRESHEARS >

I got an email request to write a blog about three things I would advocate to a programmer that could be used to speed up her program. My first flippant thought was, "Location! Location! Location!" That got me thinking about real estate and led my meandering mind to answer the original query with a paraphrase of Blake (played by Alec Baldwin) from the film *Glengarry Glen Ross*: "A-B-C. A-Always, B-Be, C-Concurrent. Always be concurrent."

I quickly realized such a simple quote was packed with so much more than just three simple things. I've written dozens of IDZ blog posts on individual items (granularity, load balance, task decomposition, parallelizing loops, etc.) that would be much more relevant. (If you can find them online, feel free to pick three and don't bother to finish reading the rest of this post. Or, better yet, finish reading this one now and search online later for more focused recommendations.) I even wrote a book on the topic of concurrent and parallel programming. If you've got a copy, take out any three pages from the latter half of the book and you'll likely have three different things you can do to speed up code. (Also, now that you've ruined your copy by tearing out those three pages, feel free to buy another copy.)

Read more >

References

1. J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *Comm. ACM,* 51(1): 107–113, January 2008.

2. Apache Hadoop Wiki, **hadoop.apache.org/#What+Is+Apache+Hadoop%3F**.

3. S. Jha, J. Qiu, A. Luckow, P. Mantha, and G.C. Fox. "A Tale of Two Data-Intensive Paradigms: Applications, Abstractions, and Architectures," eprint arXiv:1403.1528, March 2014.

4. F. Liang, C. Feng, X. Lu, and Z. Xu. "Performance Benefits of DataMPI: A Case Study with BigDataBench." In *The 4th Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware,* BPOE-4, Salt Lake City, Utah, 2014.

5. T. Hoefler, A. Lumsdaine, and J. Dongarra. "Towards Efficient MapReduce Using MPI." *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 240–249, 2009.

6. "Intel® Enterprise Edition for Lustre*," **intel.com/content/www/us/en/software/intel-enterprise-edition-for-lustre-software.html**.

7. A. Woodie. "What Can GPFS on Hadoop Do For You?" Datanami, February 2014, **datanami.com/2014/02/18/what_can_gpfs_on_hadoop_do_for_you_/**.

8. N. Rutman. "Map/Reduce on Lustre: Hadoop Performance in HPC Environments" (technical white paper). Xyratex, **xyratex.com/sites/default/files/Xyratex_white_paper_MapReduce_1-4.pdf**.

9. W. Gropp and E. Lusk. "Fault Tolerance in MPI Programs." The International Journal of High Performance Computing Applications, Volume 18, No. 3, Fall 2004, pp. 363–372.

10. J. Hursey, J.M. Squyres, and A. Lumsdaine. "A Checkpoint and Restart Service Specification for Open MPI" (technical report), **open-mpi.org/papers/iu-cs-tr635/iu-cs-tr635.pdf.**

11. "Announcing Intel® Data Analytics Acceleration Library 2016 Beta," **software.intel.com/en-us/articles/announcing-intel-data-analytics-acceleration-library-2016-beta**.

12. High-Performance Big Data Project, Network-Based Computing Laboratory, Ohio State University. "RDMA-Based Apache Hadoop," **hibd.cse.ohio-state.edu**.

13. Intel® MPI Library, **software.intel.com/en-us/intel-mpi-library**.

14. S. Plimpton. "MapReduce and MPI." SOS 17 - *Intersection of HPC & Big Data*, March 2013.

15. J. Dursi. "HPC Is Dying, and MPI Is Killing It." **dursi.ca/hpc-is-dying-and-mpi-is-killing-it**.

16. The Nielsen Company. "Bridging the Worlds of High Performance Computing and Big Data," **sites.nielsen.com/newscenter/bridging-the-worlds-of-high-performance-computing-and-big-data**.

17. "CDH Components," **cloudera.com/content/cloudera/en/products-and-services/cdh.html**.

18. Cloudera, Inc. "Llama," **cloudera.github.io/llama**.

19. Cloudera, Inc. "Apache Impala," **impala.io**.

20. GitHub, Inc. "mpich2-yarn," **github.com/alibaba/mpich2-yarn**.

21. Karypis Lab. "BDMPI - Big Data Message Passing Interface," **glaros.dtc.umn.edu/gkhome/bdmpi/overview**.

FINDING YOUR MEMORY ACCESS PERFORMANCE BOTTLENECKS

Improve Application Performance Quickly and Simply with the New Memory Access Analysis Feature of Intel® VTune™ Amplifier XE

Kevin O'Leary, *Software Technical Consulting Engineer;* Dmitry Ryabtsev, *Software Development Engineer;* and Alexey Budankov, *Software Development Engineer;* Intel Corporation

How your application accesses memory can dramatically impact performance. It's not enough to parallelize your application by adding threads and **vectorization**. Memory bandwidth is just as important, but is often not as well understood by software developers. Tools that help minimize memory latency and increase bandwidth can help developers pinpoint performance bottlenecks and diagnose their causes.

Today's modern processors have many different types of memory accesses. For example, the latency of an L1 cache hit is vastly different from the latency of an access that misses all of your memory caches and needs to access DRAM. There are additional complexities brought about by non-uniform memory access (NUMA) architectures.

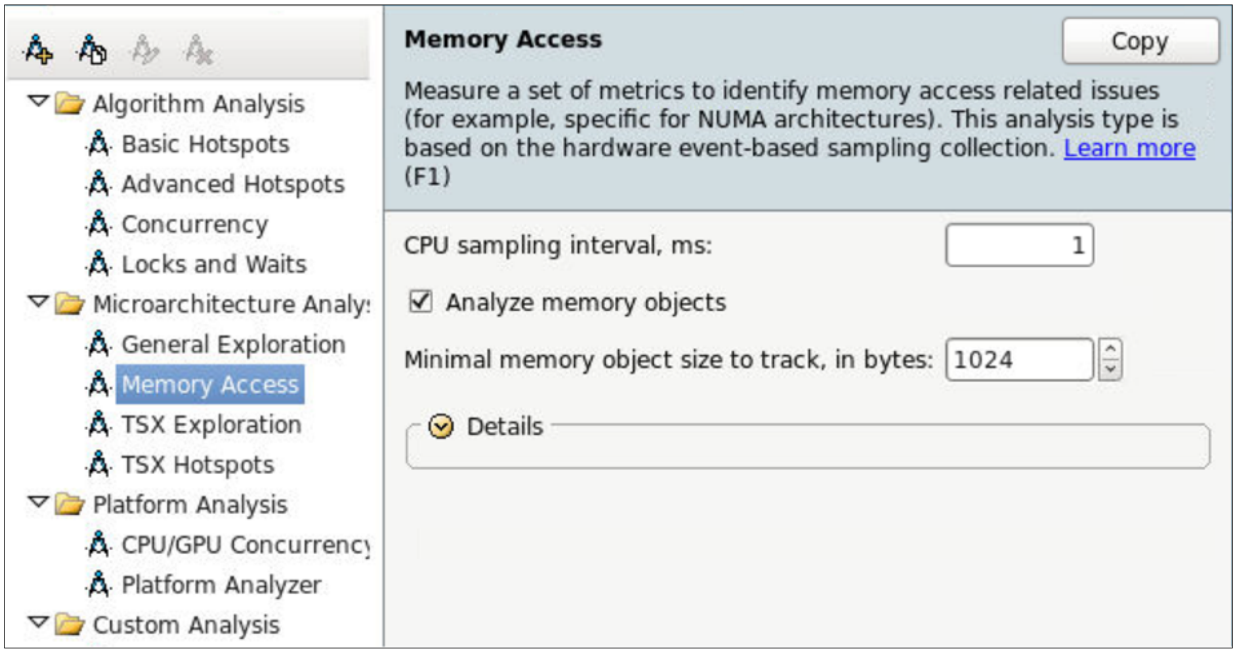
Intel® VTune™ Amplifier XE is a performance profiler that has many features you can use to analyze memory accesses. These features are contained in the new Memory Access analysis type, which lets you:

- **Detect performance problems** by memory hierarchy (e.g., L1-, L2-, LLC-, DRAM-bound).
- **Track memory objects** and attribute the latency these objects cause to their appropriate code and data structures.
- **Analyze bandwidth-limited accesses** (including DRAM and Intel® QuickPath Interconnect [Intel® QPI] bandwidth) and quickly see graphs and histograms of your DRAM and QPI that show you bandwidth over the timeline of your program.
- Identify **NUMA-related issues** contributing to performance problems.

This article provides an overview of the new Memory Access feature and how it can help solve several tough memory problems and greatly increase an application's performance.

Overview

To access Intel VTune Amplifier's Memory Access feature, click on the new "Memory Access" analysis type, and then click "Start" (**Figure 1**).



Memory Access

Measure a set of metrics to identify memory access related issues (for example, specific for NUMA architectures). This analysis type is based on the hardware event-based sampling collection. Learn more (F1)

CPU sampling interval, ms: 1

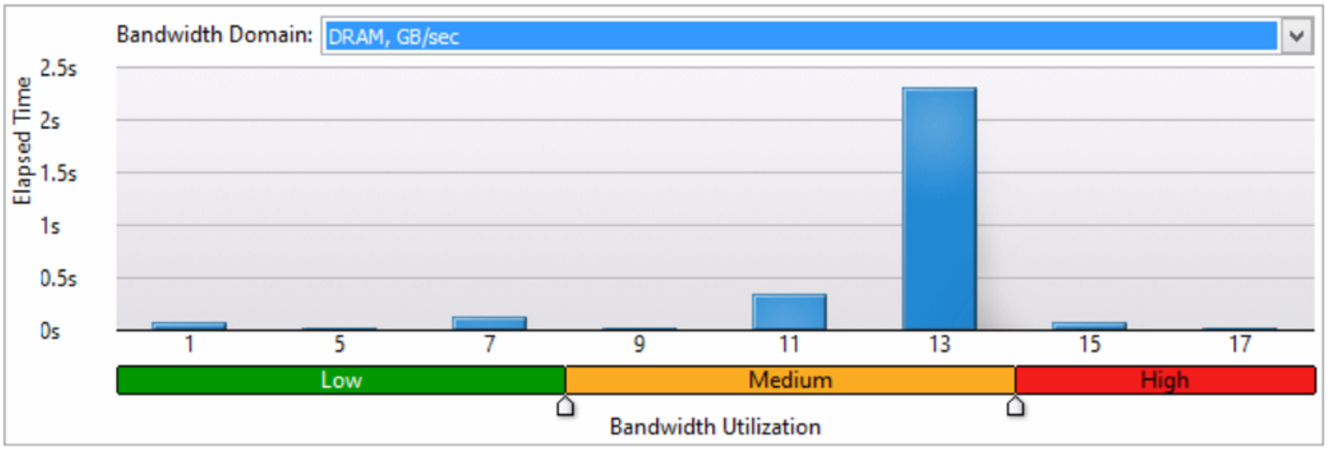
☑ Analyze memory objects

Minimal memory object size to track, in bytes: 1024

1 Access the Memory Access feature

View Bandwidth Utilization

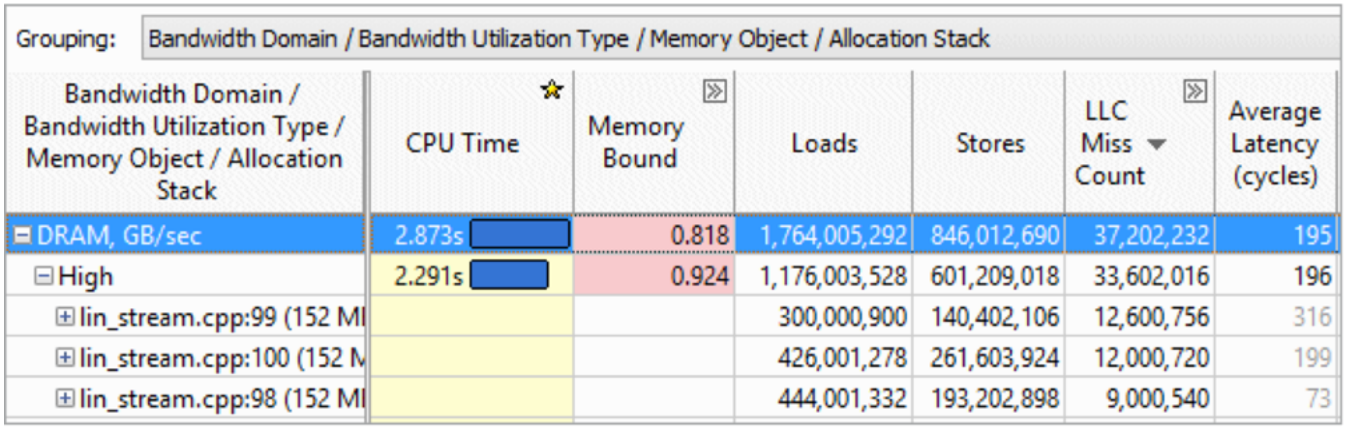
You can see how effectively your DRAM and QPI bandwidth are being utilized. You need to be concerned about high bandwidth utilization. To help fix this, you can find the places in your code contributing to bandwidth (**Figure 2**).



2 Bandwidth histogram

View Memory Objects Inducing Bandwidth

Identify the code source and memory objects that are inducing bandwidth. Grouping by the Bandwidth Domain allows you to identify memory objects that are contributing the most to your memory bandwidth (**Figure 3**). You can see the sections of code that have more DRAM issues, QPI issues, etc.



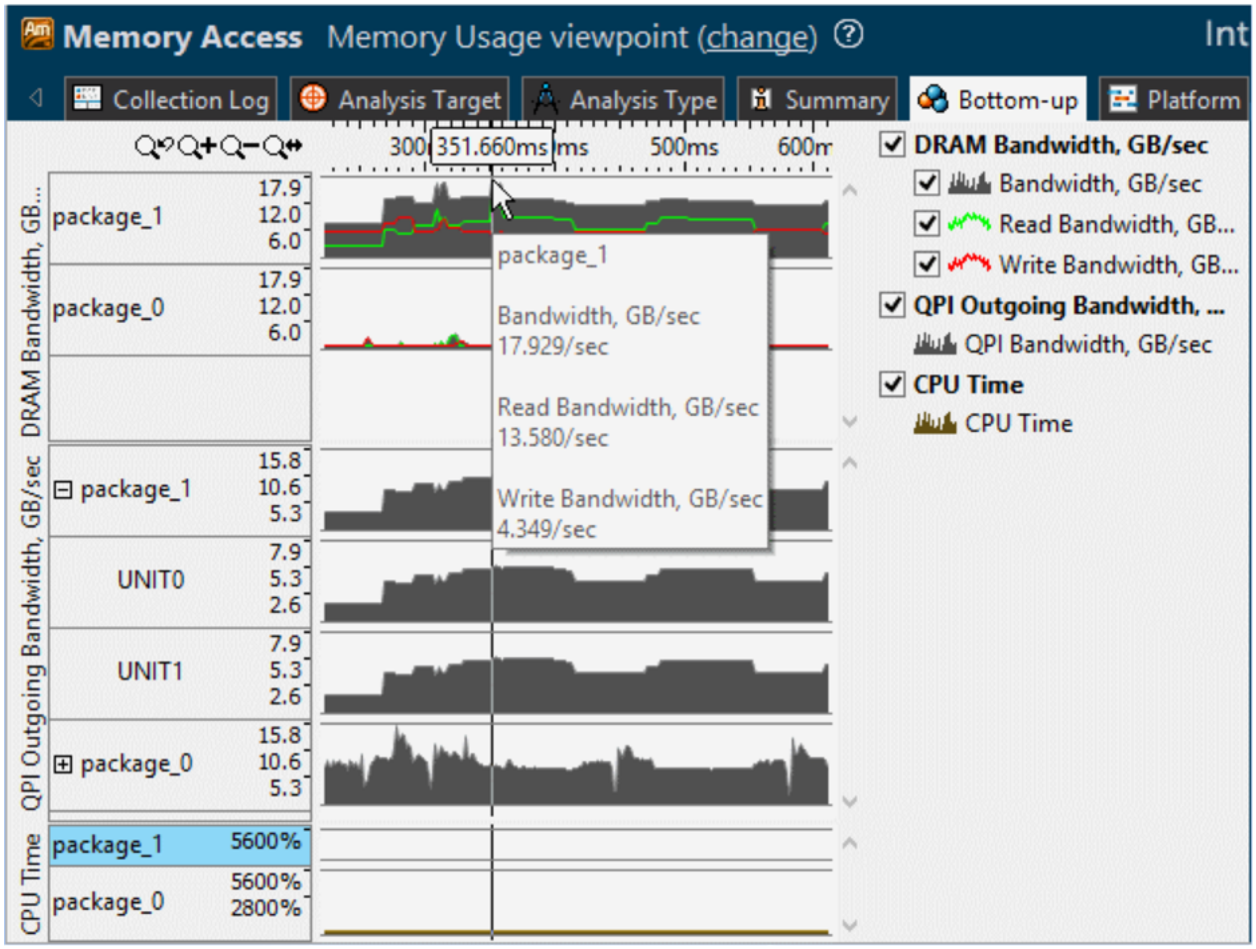
Grouping: Bandwidth Domain / Bandwidth Utilization Type / Memory Object / Allocation Stack

Bandwidth Domain / Bandwidth Utilization Type / Memory Object / Allocation Stack	CPU Time	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency (cycles)
⊟ DRAM, GB/sec	2.873s	0.818	1,764,005,292	846,012,690	37,202,232	195
⊟ High	2.291s	0.924	1,176,003,528	601,209,018	33,602,016	196
⊞ lin_stream.cpp:99 (152 MI			300,000,900	140,402,106	12,600,756	316
⊞ lin_stream.cpp:100 (152 M			426,001,278	261,603,924	12,000,720	199
⊞ lin_stream.cpp:98 (152 MI			444,001,332	193,202,898	9,000,540	73

3 Memory bandwidth

Graph Memory Bandwidth over the Timeline of Your Application

Your memory bandwidth will, in general, vary as your program runs. By viewing the bandwidth in a graph that shows your read/write bandwidth in GB/sec, you can see where in your application spikes in memory usage and target the section of your application where the extra memory usage occurs (**Figure 4**). You can then filter by selecting the area in the timeline where the spike was occurring and see only the code that was active during that time.



package_1

Bandwidth, GB/sec
17.929/sec

Read Bandwidth, GB/sec
13.580/sec

Write Bandwidth, GB/sec
4.349/sec

4 Memory usage

The ability to track down the code sections in your application that are inducing memory bandwidth is a powerful feature. Average latency is critical when tuning for memory accesses. Viewing bandwidth in the timeline graph is a simple way to characterize your memory usage as your application runs. In the latest version of Intel VTune Amplifier, the bandwidth graph is relative to the maximum possible that your platform is capable of achieving, so you can clearly see how much performance you are leaving on the table.

Solving Memory Problems

Tough Problem No. 1: False Sharing

First, some quick definitions:

Sharing: If more than one thread accesses the same piece of memory, then they are said to "share" the memory. Because of the way that modern computers are organized, this sharing can cause all sorts of performance penalties. These performance penalties are necessary because all of the different threads/cores need to agree what is stored at a memory address and synchronize all of the various caches due to this contention.

False sharing: This is when two different threads access a piece of memory that is located on the same cache line. They don't actually share the same piece of memory, but because the memory references are located close together, they just happen to be stored together on the same cache line. When multiple threads have false sharing, they have the same type of performance penalties as threads that are actually sharing the same piece of memory—but they are taking the performance hit that is completely unnecessary.

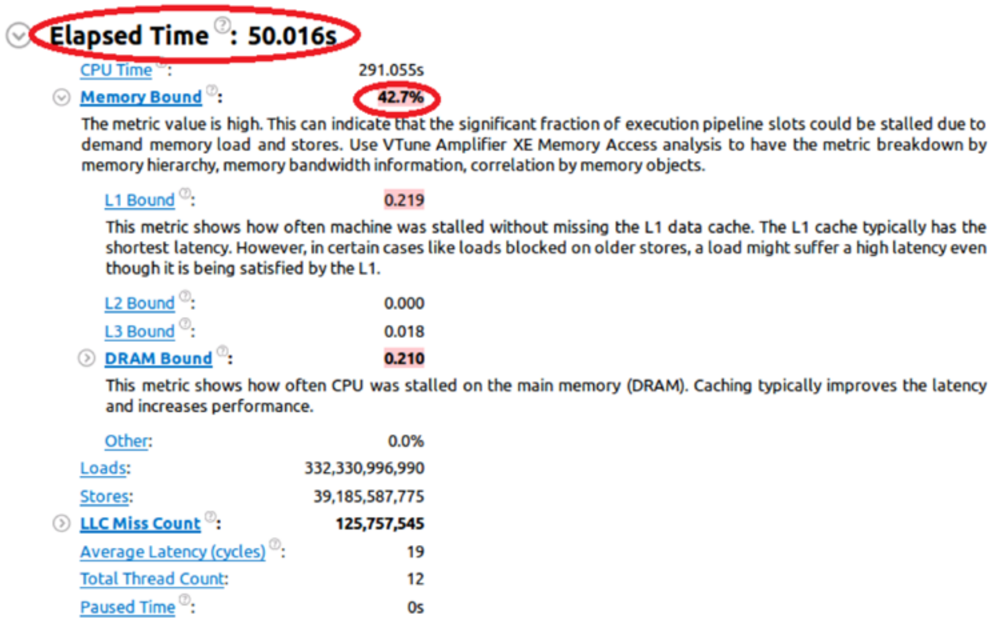
For this case, we'll study the linear_regression application from the Phoenix System* (**csl.stanford.edu/~christos/sw/phoenix**).

Step No. 1: Run Memory Access Analysis to Uncover Potential Memory Issues

Run Memory Access analysis with these options enabled:

- Select Memory Objects Analysis.
- Set Object Size Threshold to 1 to capture all memory allocations.

The Summary view shows some key metrics (**Figure 5**).



Elapsed Time: 50.016s

CPU Time:	291.055s
Memory Bound:	42.7%

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use VTune Amplifier XE Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

L1 Bound:	0.219

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1.

L2 Bound:	0.000
L3 Bound:	0.018
DRAM Bound:	0.210

This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

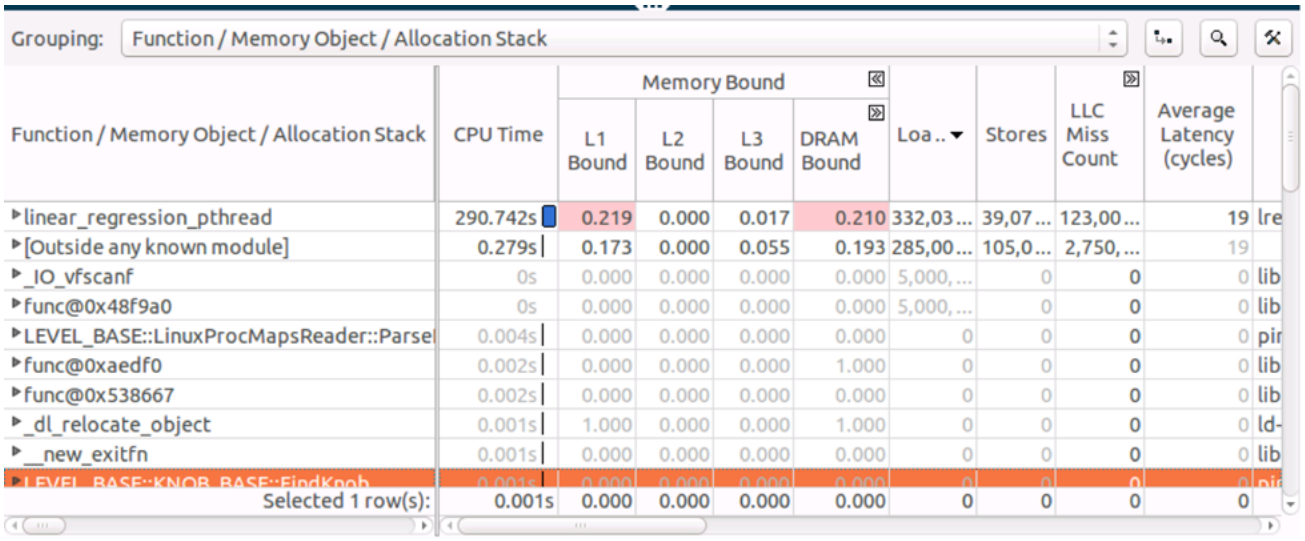
Other:	0.0%
Loads:	332,330,996,990
Stores:	39,185,587,775
LLC Miss Count:	125,757,545
Average Latency (cycles):	19
Total Thread Count:	12
Paused Time:	0s

5 Key metrics

For our first run, the elapsed time is 50 seconds. We can also see that the application is "Memory Bound" and that more than 42 percent of CPU resources are wasted waiting for memory operations to complete. Note that the Memory Bound metric is colored pink; this indicates that a potential performance issue needs to be addressed.

Step No. 2: Investigate the Memory Issue Identified

Switch to the Bottom-Up tab (**Figure 6**) to see more details.



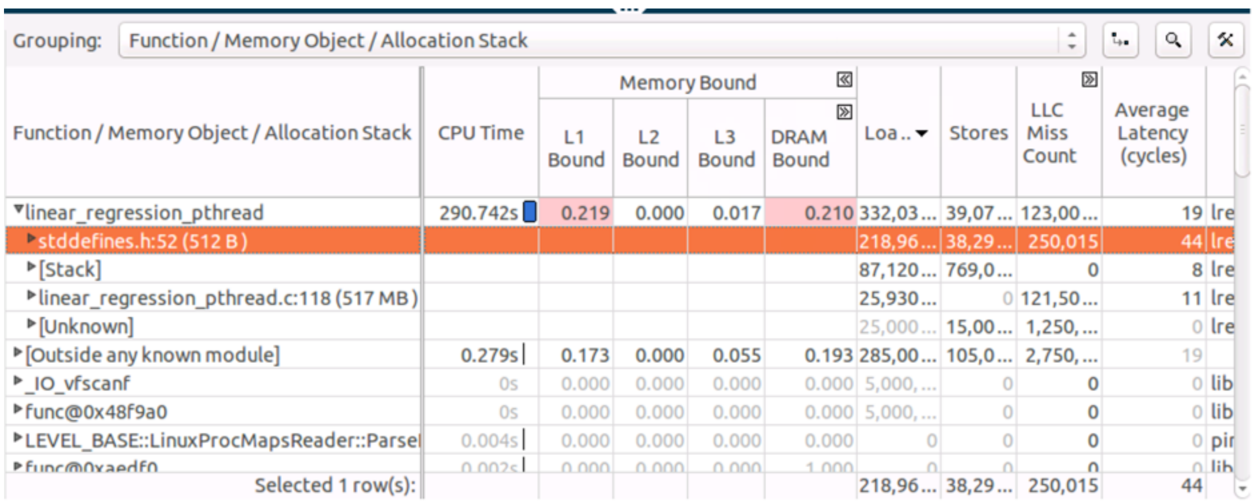
Grouping: Function / Memory Object / Allocation Stack

Function / Memory Object / Allocation Stack	CPU Time	Memory Bound				Loa..	Stores	LLC Miss Count	Average Latency (cycles)	
		L1 Bound	L2 Bound	L3 Bound	DRAM Bound					
linear_regression_pthread	290.742s	0.219	0.000	0.017	0.210	332,03...	39,07...	123,00...	19	lre
[Outside any known module]	0.279s	0.173	0.000	0.055	0.193	285,00...	105,0...	2,750,...	19	
_IO_vfscanf	0s	0.000	0.000	0.000	0.000	5,000,...	0	0	0	lib
func@0x48f9a0	0s	0.000	0.000	0.000	0.000	5,000,...	0	0	0	lib
LEVEL_BASE::LinuxProcMapsReader::Parsel	0.004s	0.000	0.000	0.000	0.000	0	0	0	0	pir
func@0xaedf0	0.002s	0.000	0.000	0.000	1.000	0	0	0	0	lib
func@0x538667	0.002s	0.000	0.000	0.000	0.000	0	0	0	0	lib
_dl_relocate_object	0.001s	1.000	0.000	0.000	1.000	0	0	0	0	ld-
__new_exitfn	0.001s	0.000	0.000	0.000	0.000	0	0	0	0	lib
LEVEL_BASE::KNOB_BASE::FindKnob	0.001s	0.000	0.000	0.000	0.000	0	0	0	0	pi
Selected 1 row(s):	0.001s	0.000	0.000	0.000	0.000	0	0	0	0	

6 Bottom-Up tab

We see that almost all of our time is spent in a single function, linear_regression_pthread. We can also see that this function is L1 and DRAM bound.

Expand the grid row for the linear_regression_pthread function to see what memory objects it accessed and sort by Loads (**Figure 7**).



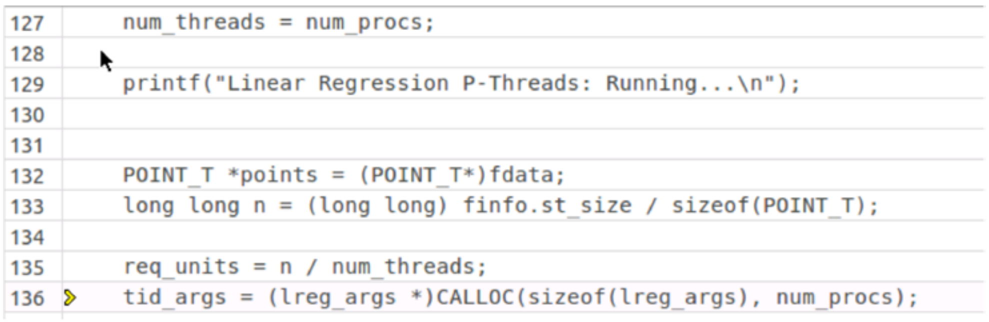
Grouping: Function / Memory Object / Allocation Stack

Function / Memory Object / Allocation Stack	CPU Time	Memory Bound				Loa..	Stores	LLC Miss Count	Average Latency (cycles)	
		L1 Bound	L2 Bound	L3 Bound	DRAM Bound					
linear_regression_pthread	290.742s	0.219	0.000	0.017	0.210	332,03...	39,07...	123,00...	19	lre
stddefines.h:52 (512 B)						218,96...	38,29...	250,015	44	lre
[Stack]						87,120...	769,0...	0	8	lre
linear_regression_pthread.c:118 (517 MB)						25,930...	0	121,50...	11	lre
[Unknown]						25,000...	15,00...	1,250,...	0	lre
[Outside any known module]	0.279s	0.173	0.000	0.055	0.193	285,00...	105,0...	2,750,...	19	
_IO_vfscanf	0s	0.000	0.000	0.000	0.000	5,000,...	0	0	0	lib
func@0x48f9a0	0s	0.000	0.000	0.000	0.000	5,000,...	0	0	0	lib
LEVEL_BASE::LinuxProcMapsReader::Parse	0.004s	0.000	0.000	0.000	0.000	0	0	0	0	pir
func@0xaedf0	0.002s	0.000	0.000	0.000	1.000	0	0	0	0	lib
Selected 1 row(s):						218,96...	38,29...	250,015	44	

7 Expand Grid row and sort by Loads

We see that the hottest object—stddefines.h:52 (512 B)—is quite small, only 512 bytes. It should fit fully into the L1 cache, but the Average Latency metric shows a latency of 44 cycles. This far exceeds the normal L1 access latency of four cycles, which often means we have some contention issues that could be either true or false sharing.

By examining the allocation stack for the "stddefines.h:52 (512B)" object (**Figure 8**), we can see source location where the object was allocated.



```
    num_threads = num_procs;

    printf("Linear Regression P-Threads: Running...\n");


    POINT_T *points = (POINT_T*)fdata;
    long long n = (long long) finfo.st_size / sizeof(POINT_T);

    req_units = n / num_threads;
    tid_args = (lreg_args *)CALLOC(sizeof(lreg_args), num_procs);
```

8 Source location

In this example, num_procs is the number of threads and the structure being allocated is lreg_args.

```
typedef struct
{
    pthread_t tid;
    POINT_T *points;
    int num_elems;
    long long SX;
    long long SY;
    long long SXX;
    long long SYY;
    long long SXY;
} lreg_args;
```

The threads are accessing the lreg_args structure as follows:

```
 // ADD UP RESULTS
for (i = 0; i < args->num_elems; i++)
{
   //Compute SX, SY, SYY, SXX, SXY
   args->SX  += args->points[i].x;
   args->SXX += args->points[i].x*args->points[i].x;
   args->SY  += args->points[i].y;
   args->SYY += args->points[i].y*args->points[i].y;
   args->SXY += args->points[i].x*args->points[i].y;
}
```

We can see that each thread is independently accessing its element in the array, so it does look like false sharing.

Step No. 3: Modify the Code to Remove the False Sharing

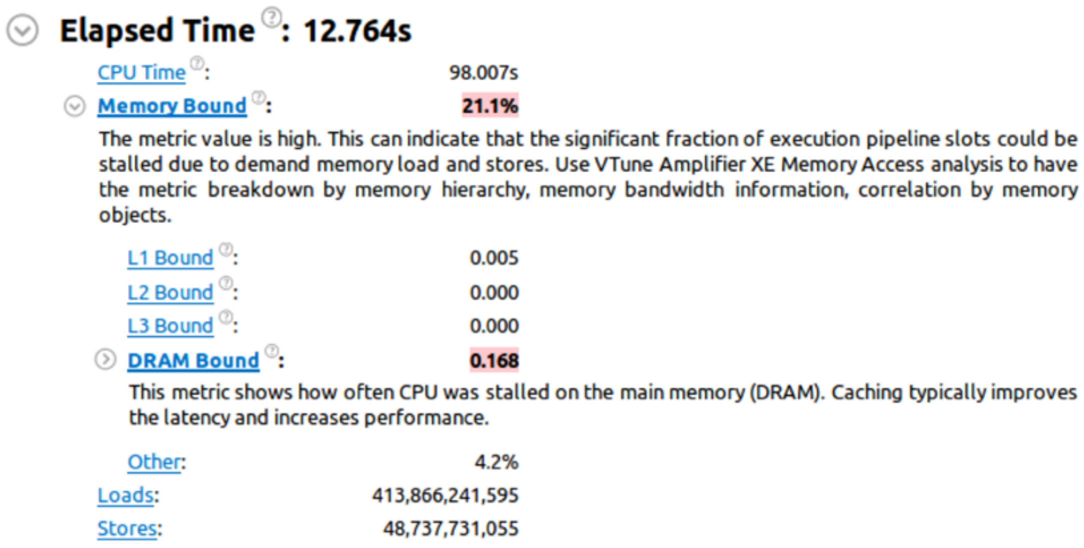
False sharing can typically be easily avoided by adding padding so that threads always access different cache lines.

Modify the lreg_args structure by adding a `char pad[80]` field as follows:

```
typedef struct
{
    char pad[80];
    pthread_t tid;
    POINT_T *points;
    int num_elems;
    long long SX;
    long long SY;
    long long SXX;
    long long SYY;
    long long SXY;
} lreg_args;
```

Step No. 4: Rerun Memory Access Analysis

Figure 9 shows the new result.



Elapsed Time: 12.764s

CPU Time:	98.007s
Memory Bound:	**21.1%**

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use VTune Amplifier XE Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

L1 Bound:	0.005
L2 Bound:	0.000
L3 Bound:	0.000
DRAM Bound:	**0.168**

This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

Other:	4.2%
Loads:	413,866,241,595
Stores:	48,737,731,055

9 Elapsed time

The new elapsed time is 12 seconds. We improved the application's performance by approximately 4x just by making a one line code change that padded a structure. The Memory Bound metric is also much lower and the L1 Bound issue has been resolved.

Tough Problem No. 2: NUMA Issues

In a processor that supports NUMA, it is not enough to know that you missed a cache on the CPU where you are running. In NUMA architectures, you could also be referencing the cache and DRAM on another CPU. The latencies for this type of access are an order of magnitude greater than the local case. You need the ability to identify and optimize these remote memory accesses.

For this case, we'll study a simple triad application parallelized using OpenMP* and running a dual-socket Intel® Xeon® processor.

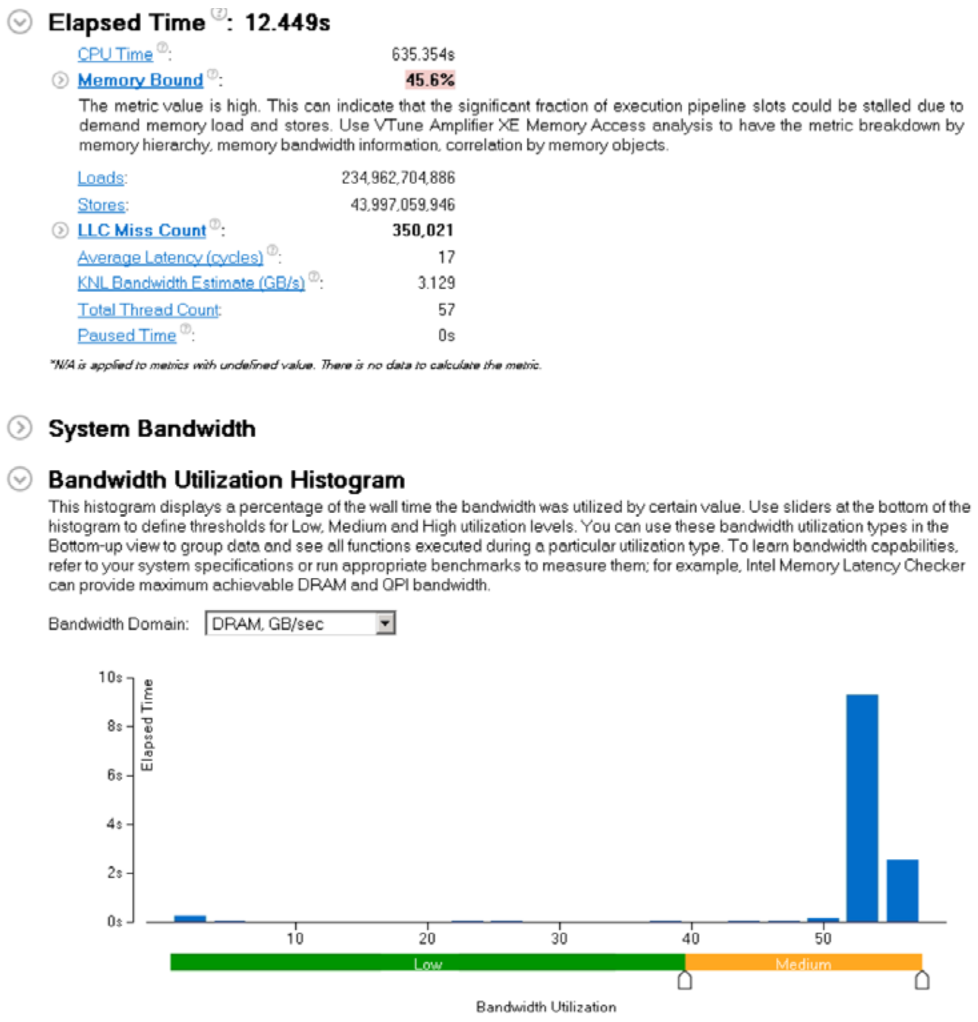
Here is the code:

```
static double   a[N];
static double   b[N];
static double   c[N];

void doTriad(double x)
{
#pragma omp parallel for
    for (int i = 0; i < N; i++)
        a[i] = b[i] + x*c[i];
}

int main() {

    for (int i = 0; i < N; ++i)
    {
        a[i] = 1.0;
        b[i] = 2.0;
        c[i] = 0.0;
    }

    for (int n = 0; n < NTIMES; ++n)
    {
        doTriad(3.0);
    }

    return 0;
}
```

First, we initialize the arrays and then call the Triad function that uses “omp parallel for.”

Step No. 1: Run Memory Access Analysis

Run the Memory Access analysis on this application. The expectation is for it to be DRAM bandwidth-bound, without utilizing the system bandwidth up to the maximum (**Figure 10**).



Elapsed Time: 12.449s

CPU Time:	635.354s
Memory Bound:	45.6%

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use VTune Amplifier XE Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

Loads:	234,962,704,886
Stores:	43,997,059,946
LLC Miss Count:	350,021
Average Latency (cycles):	17
KNL Bandwidth Estimate (GB/s):	3.129
Total Thread Count:	57
Paused Time:	0s

*N/A is applied to metrics with undefined value. There is no data to calculate the metric.

System Bandwidth

Bandwidth Utilization Histogram

This histogram displays a percentage of the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and QPI bandwidth.

Bandwidth Domain: DRAM, GB/sec

10 Memory Access analysis

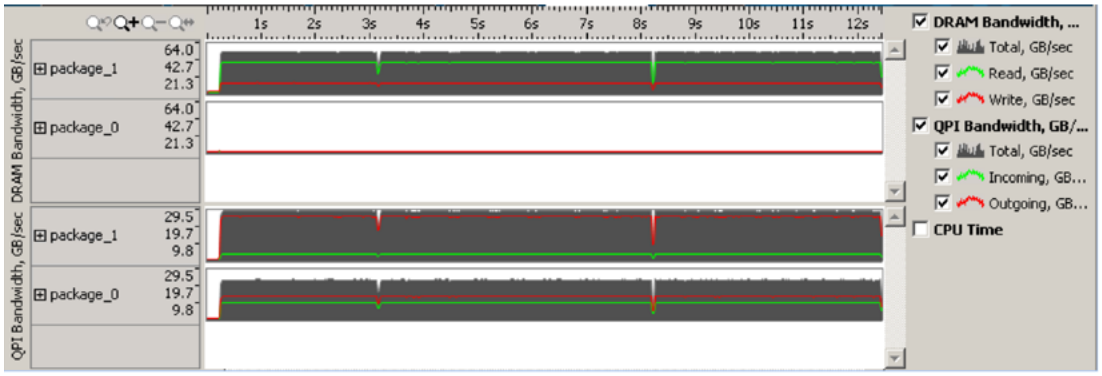
The summary section includes some very useful metrics. We can see the elapsed time is 12.449 seconds. The Memory Bound metric is high and highlighted (as we expected). What is puzzling is the Bandwidth Utilization histogram shows only a medium DRAM bandwidth utilization level of 50 to 60 GB/s. This will need to be investigated.

Some other useful metrics are:

- **Average Latency.** This is the average number of cycles our memory accesses are taking. Note: An L1 memory access can usually be done in four cycles, but a remote DRAM access can take approximately 300 cycles.
- **KNL Bandwidth Estimate.** This is an estimate of the expected per-core bandwidth if run on next-generation Intel® Xeon Phi™ coprocessors. This is useful for users who will be moving to this platform and would like to know if the memory access portion of their code is ready.

Step No. 2: Investigate Bandwidth Utilization

Switch to the Bottom-Up tab (**Figure 11**) to see more details.



11 Bottom-Up tab

From the timeline graph, we see that DRAM bandwidth is utilized on only one of the sockets, package_1. In addition, we see high QPI (intra-socket) traffic, up to 30 GB/s. This is a typical issue on NUMA machines, where memory is allocated on one node and the work is split among multiple nodes. This forces some of them to have to load the data remotely over QPI links, which is much slower than accessing local memory.

BLOG HIGHLIGHTS

An Intro to MCDRAM (High Bandwidth Memory) on Knights Landing

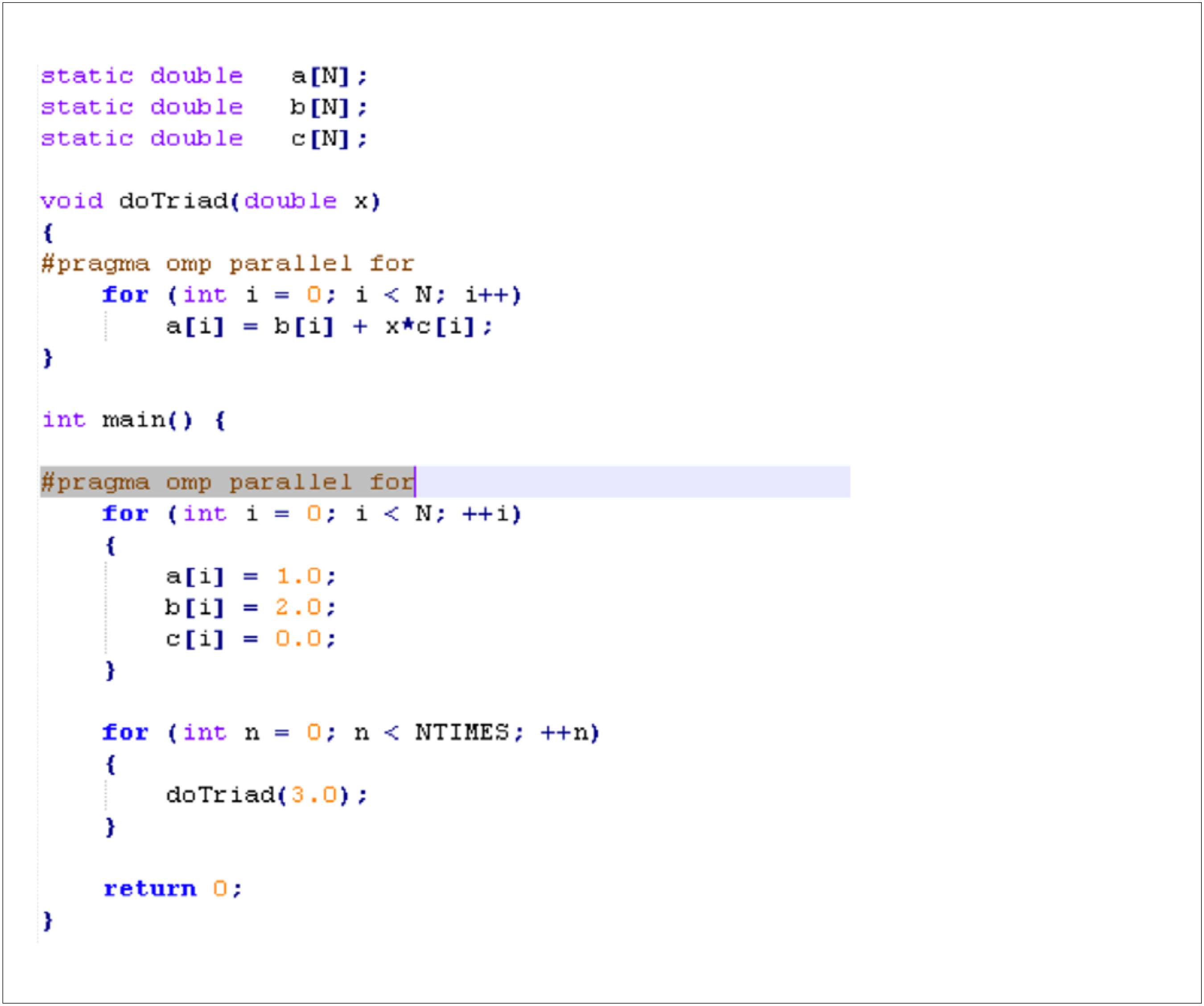
BY MIKE PEARCE >

The next-generation Intel® Xeon Phi™ coprocessor family x200 product (code-name Knights Landing) brings in new memory technology, a high bandwidth on package memory called Multi-Channel DRAM (MCDRAM) in addition to the traditional DDR4. MCDRAM is a high bandwidth (~4x more than DDR4), low-capacity (up to 16GB) memory, packaged with the Knights Landing Silicon. MCDRAM can be configured as a third-level cache (memory-side cache) or as a distinct NUMA node (allocatable memory) or somewhere in between. With the different memory modes by which the system can be booted, it becomes very challenging from a software perspective to understand the best mode suitable for an application.

Read more >

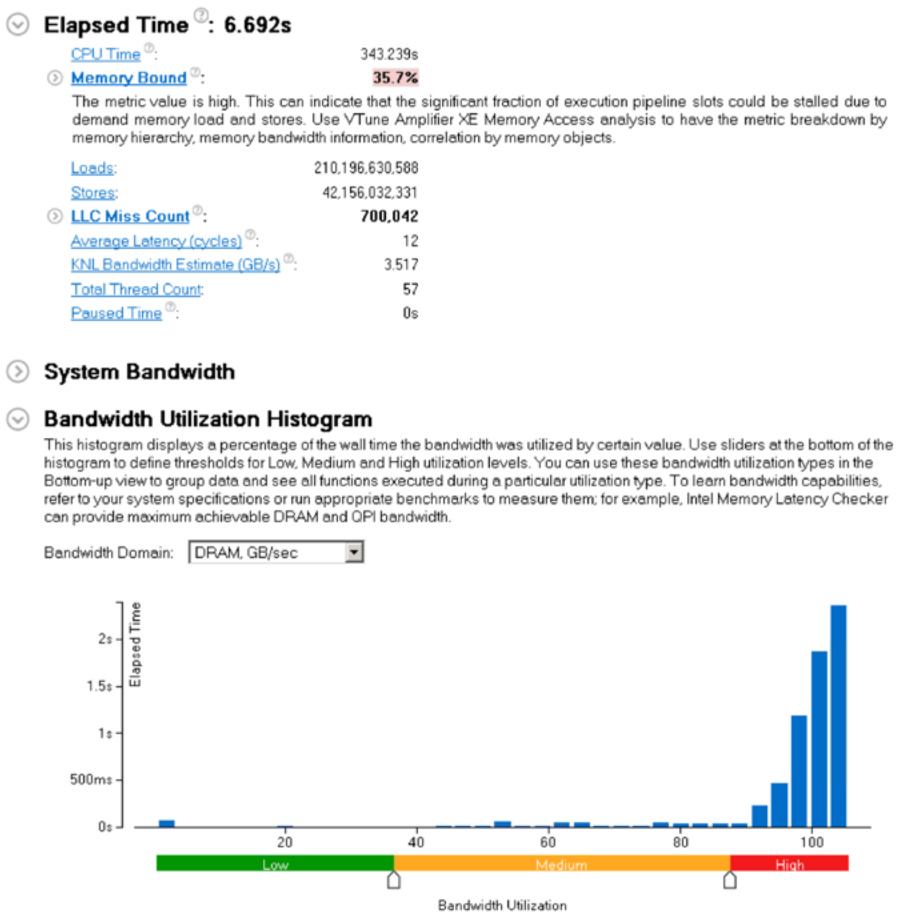
Step No. 3: Modify the Code to Avoid Remote Memory Access

If we change the code to make both sockets access only local memory, thus avoiding remote node accesses, it should run faster. On Linux*, memory pages are allocated on first access. So the solution for our case is simple: We should initialize the memory on the same nodes where we'll be working with them. We can accomplish by adding an `omp parallel for` pragma to our initialization loop:



```
static double   a[N];
static double   b[N];
static double   c[N];

void doTriad(double x)
{
#pragma omp parallel for
    for (int i = 0; i < N; i++)
        a[i] = b[i] + x*c[i];
}

int main() {

#pragma omp parallel for
    for (int i = 0; i < N; ++i)
    {
        a[i] = 1.0;
        b[i] = 2.0;
        c[i] = 0.0;
    }

    for (int n = 0; n < NTIMES; ++n)
    {
        doTriad(3.0);
    }

    return 0;
}
```

Step No. 4: Rerun Memory Analysis with KMP_AFFINITY Variable



Elapsed Time: 6.692s

CPU Time:	343.239s
Memory Bound:	35.7%

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use VTune Amplifier XE Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

Loads:	210,196,630,588
Stores:	42,156,032,331
LLC Miss Count:	700,042
Average Latency (cycles):	12
KNL Bandwidth Estimate (GB/s):	3.517
Total Thread Count:	57
Paused Time:	0s

System Bandwidth

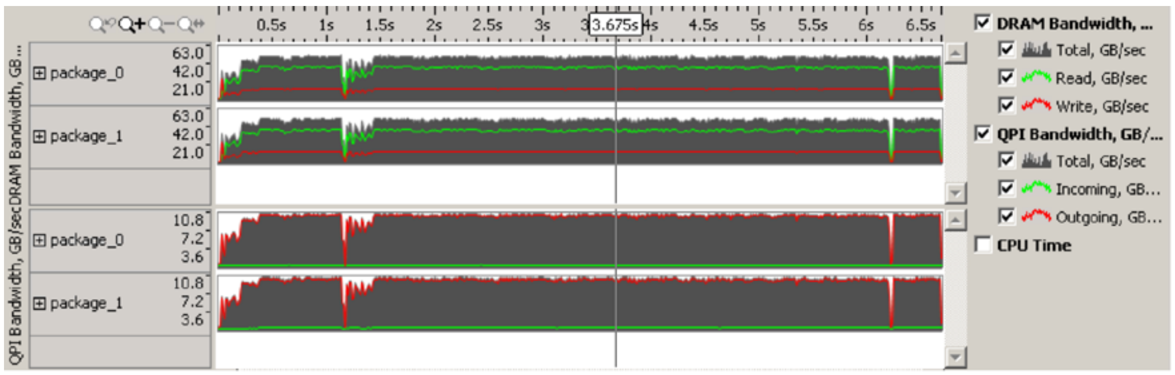
Bandwidth Utilization Histogram

This histogram displays a percentage of the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and QPI bandwidth.

Bandwidth Domain: DRAM, GB/sec

12 Bandwidth utilization

Our elapsed time has decreased from 12.449 to 6.69 seconds, almost a 2x speed-up. Also, our DRAM and width utilization moved to high levels, aligned with expectations (**Figure 12**).



Bandwidth is now equally split between the sockets and QPI traffic is 3x lower (**Figure 13**).

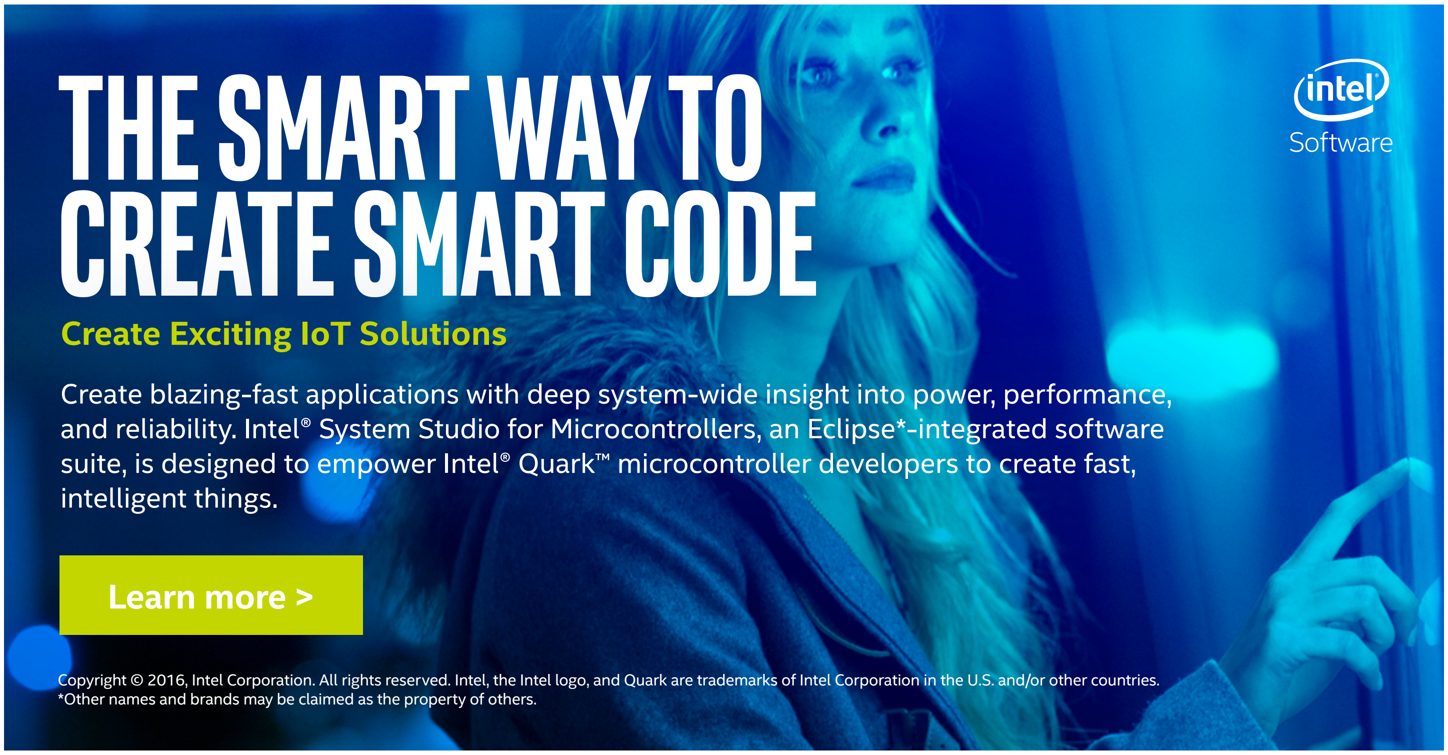
The complexities of NUMA architectures necessitate greater attention to the details of your memory accesses. By optimizing the memory accesses in your application that have the greatest latencies, you can get the biggest potential performance gains.

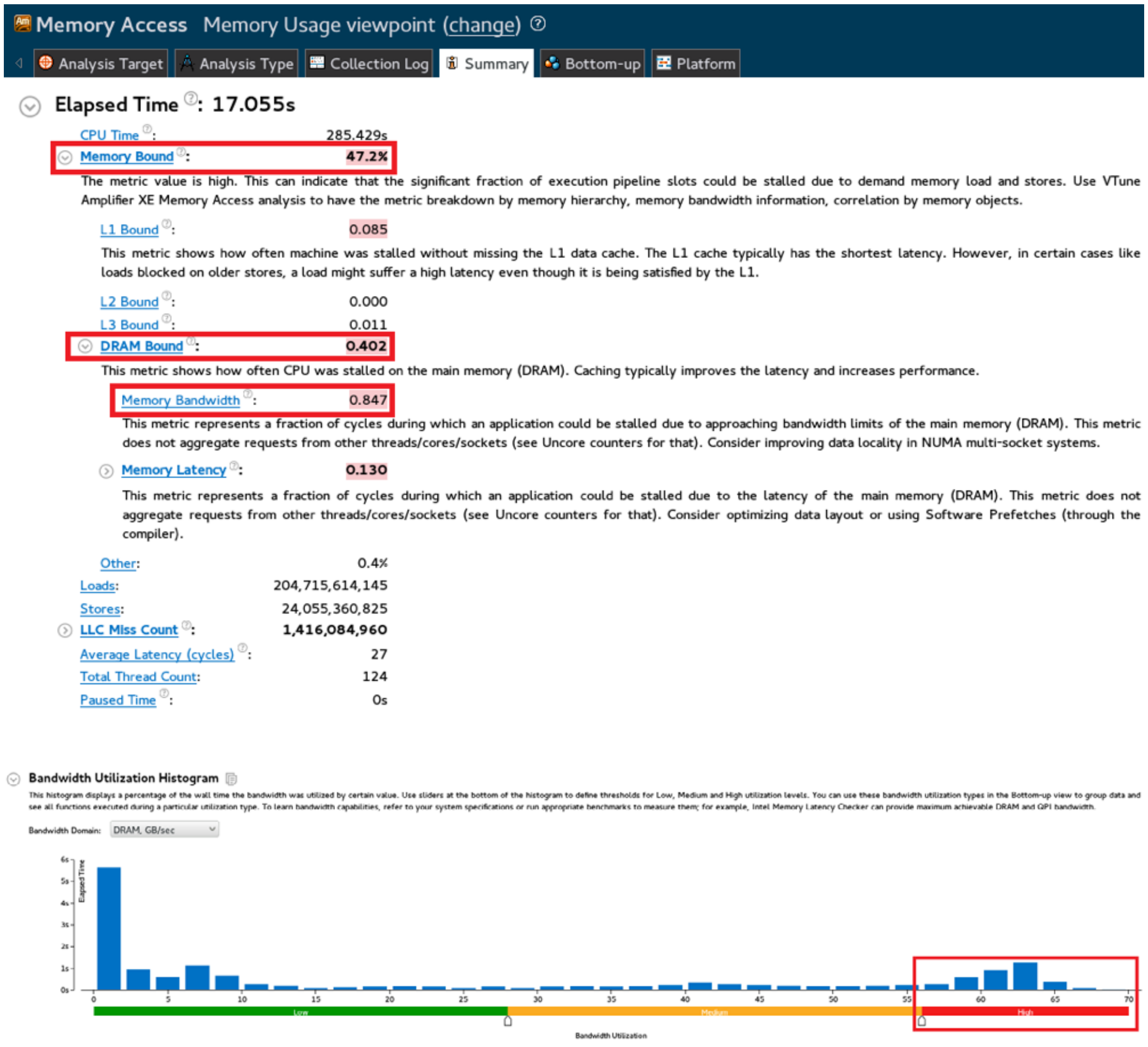
Tough Problem No. 3: Optimizing for Next-Generation Intel® Xeon Phi™ Coprocessors

Memory bandwidth often becomes a limiting factor for application performance. The new generation of Intel Xeon Phi coprocessors features a special on-package MCDRAM memory that aims to alleviate this physical bandwidth limit problem. MCDRAM memory can deliver much greater bandwidth speed-up in addition to the 90 GB/s for the traditional DRAM (DDR4) memory that you can also access. However, MCDRAM memory has limited size, so it is important to determine which data objects should be placed to this high-bandwidth type of memory to benefit the most.

Step No. 1: Run Memory Access Analysis

In this example, we'll be using the miniFE* benchmark from the Mantevo Suite* (**mantevo.org/**). Profiling the benchmark without modifications using the Intel VTune Amplifier Memory Access analysis features, we observe that the benchmark code is memory bound. This means it induces a significant amount of traffic in the traditional memory and may benefit from employing the high-bandwidth capabilities of MCDRAM (**Figure 14**).





Elapsed Time: 17.055s

CPU Time:	285.429s
Memory Bound:	47.2%

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use VTune Amplifier XE Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

L1 Bound:	0.085

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1.

L2 Bound:	0.000
L3 Bound:	0.011
DRAM Bound:	0.402

This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

Memory Bandwidth:	0.847

This metric represents a fraction of cycles during which an application could be stalled due to approaching bandwidth limits of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider improving data locality in NUMA multi-socket systems.

Memory Latency:	0.130

This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider optimizing data layout or using Software Prefetches (through the compiler).

Other:	0.4%
Loads:	204,715,614,145
Stores:	24,055,360,825
LLC Miss Count:	1,416,084,960
Average Latency (cycles):	27
Total Thread Count:	124
Paused Time:	0s

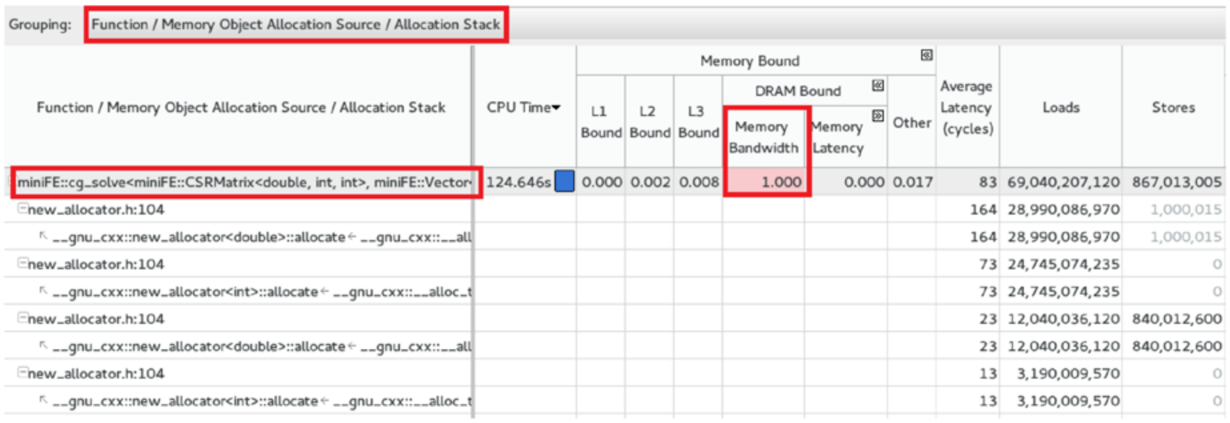
Bandwidth Utilization Histogram

This histogram displays a percentage of the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and QPI bandwidth.

14 Memory access

Step No. 2: Investigate the Memory Allocation-Inducing Bandwidth

Applying Function/Memory Object/Allocation Stack grouping in the Bottom Up view of Memory Access analysis results, we find that the majority of memory accesses are produced in the objects of miniFE::CSRMatrix and miniFE::Vector classes (**Figure 15**).

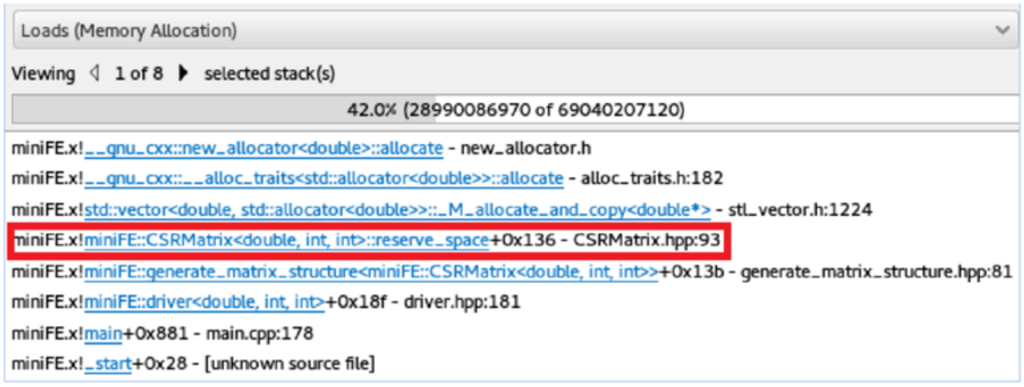


Grouping: Function / Memory Object Allocation Source / Allocation Stack

Function / Memory Object Allocation Source / Allocation Stack	CPU Time	Memory Bound						Average Latency (cycles)	Loads	Stores
		L1 Bound	L2 Bound	L3 Bound	DRAM Bound		Other			
					Memory Bandwidth	Memory Latency				
miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector	124.646s	0.000	0.002	0.008	1.000	0.000	0.017	83	69,040,207,120	867,013,005
new_allocator.h:104								164	28,990,086,970	1,000,015
__gnu_cxx::new_allocator<double>::allocate ← __gnu_cxx::__all								164	28,990,086,970	1,000,015
new_allocator.h:104								73	24,745,074,235	0
__gnu_cxx::new_allocator<int>::allocate ← __gnu_cxx::__alloc_t								73	24,745,074,235	0
new_allocator.h:104								23	12,040,036,120	840,012,600
__gnu_cxx::new_allocator<double>::allocate ← __gnu_cxx::__all								23	12,040,036,120	840,012,600
new_allocator.h:104								13	3,190,009,570	0
__gnu_cxx::new_allocator<int>::allocate ← __gnu_cxx::__alloc_t								13	3,190,009,570	0

15 Function/Memory Object/Allocation Stack grouping

In addition, the Stack Pane provides the full allocation call stack of miniFE::CSRMatrix class data, where we can see the source code location of the allocation operation (**Figure 16**).



Loads (Memory Allocation)

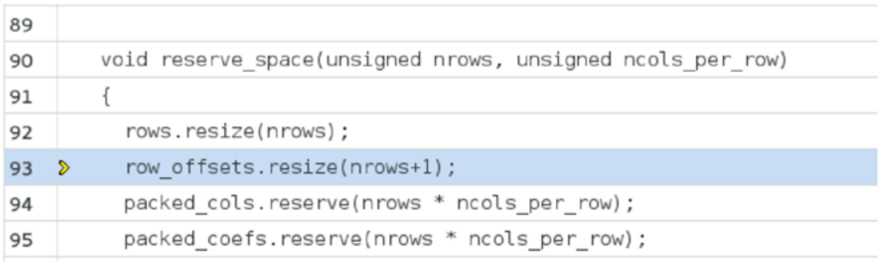
Viewing 1 of 8 selected stack(s)

42.0% (28990086970 of 69040207120)

```
miniFE.x!__gnu_cxx::new_allocator<double>::allocate - new_allocator.h
miniFE.x!__gnu_cxx::__alloc_traits<std::allocator<double>>::allocate - alloc_traits.h:182
miniFE.x!std::vector<double, std::allocator<double>>::_M_allocate_and_copy<double*> - stl_vector.h:1224
miniFE.x!miniFE::CSRMatrix<double, int, int>::reserve_space+0x136 - CSRMatrix.hpp:93
miniFE.x!miniFE::generate_matrix_structure<miniFE::CSRMatrix<double, int, int>>+0x13b - generate_matrix_structure.hpp:81
miniFE.x!miniFE::driver<double, int, int>+0x18f - driver.hpp:181
miniFE.x!main+0x881 - main.cpp:178
miniFE.x!_start+0x28 - [unknown source file]
```

16 Stack Pane

Clicking on the line associated with the CSRMatrix.hpp:93 string, the Source View is opened at the place of allocation operation in CSRMatrix.hpp file at line 93 (**Figure 17**).



```
89
90      void reserve_space(unsigned nrows, unsigned ncols_per_row)
91      {
92        rows.resize(nrows);
93        row_offsets.resize(nrows+1);
94        packed_cols.reserve(nrows * ncols_per_row);
95        packed_coefs.reserve(nrows * ncols_per_row);
```

17 Source View

The miniFE::CSRMatrix class data is managed by STL vector containers that have the flexibility of specifying a custom memory allocator class for stored vector elements:

```
template<typename Scalar,
         typename LocalOrdinal,
         typename GlobalOrdinal,
         typename ComputeNode>
struct CSRMatrix {
…
  typedef Scalar        ScalarType;
  typedef LocalOrdinal  LocalOrdinalType;
  typedef GlobalOrdinal GlobalOrdinalType;
  typedef ComputeNode   ComputeNodeType;

  bool                       has_local_indices;
  std::vector<GlobalOrdinal> rows;
  std::vector<LocalOrdinal>  row_offsets;
  std::vector<LocalOrdinal>  row_offsets_external;
  std::vector<GlobalOrdinal> packed_cols;
  std::vector<Scalar>        packed_coefs;
  LocalOrdinal               num_cols;
  ComputeNode&               compute_node;
…
}
```

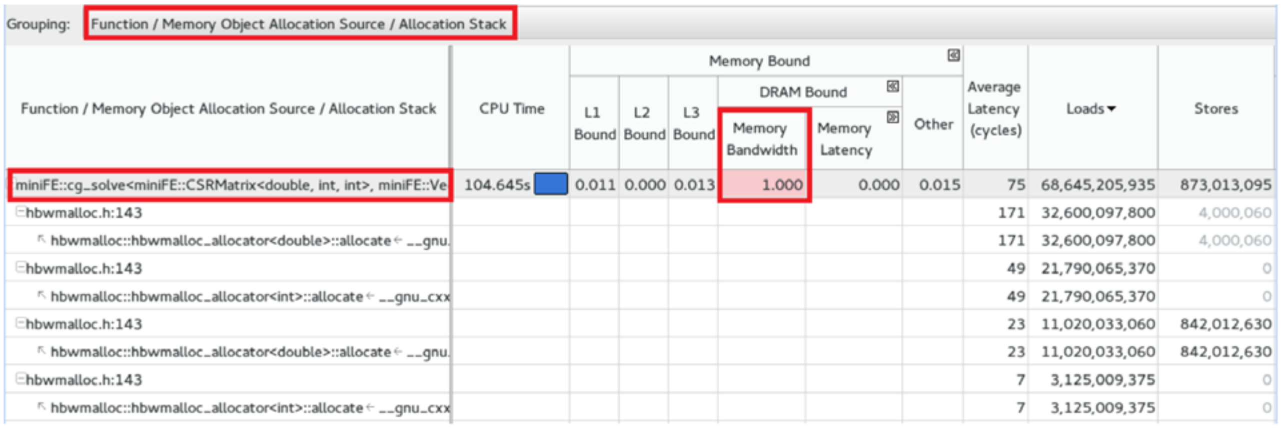
Step No. 2: Allocate Objects Using High-Bandwidth Memory

Moving miniFE::CSRMatrix and miniFE::Vector data objects to MCDRAM memory is possible by employing the memkind library API (**https://github.com/memkind/memkind**). The hbwmalloc.h header file provides the implementation of the hbwmalloc::hbwmalloc_allocator class, which may be used to parameterize the STL vector container with MCDRAM (high-bandwidth) type of memory.

For our case, the modifications look like this:

```
…
#include "/opt/mk-0.3.0/include/hbwmalloc.h"
…
template<typename Scalar,
         typename LocalOrdinal,
         typename GlobalOrdinal,
         typename ComputeNode>
struct CSRMatrix {
…
  typedef Scalar        ScalarType;
  typedef LocalOrdinal  LocalOrdinalType;
  typedef GlobalOrdinal GlobalOrdinalType;
  typedef ComputeNode   ComputeNodeType;

  bool                       has_local_indices;
  std::vector<GlobalOrdinal, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> > rows;
  std::vector<LocalOrdinal, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> >  row_offsets;
  std::vector<LocalOrdinal, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> >  row_offsets_external;
  std::vector<GlobalOrdinal, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> > packed_cols;
  std::vector<Scalar, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> >        packed_coefs;
  LocalOrdinal               num_cols;
  ComputeNode&               compute_node;
…
}
```

Rebuilding the modified source code and reapplying the Memory Access analysis to the new version of the benchmark code, we observe that miniFE::CSRMatrix and miniFE::Vector data objects are now created using hbwmalloc::hbwmalloc_allocator class provided by the memkind library (**Figure 18**).



Grouping: Function / Memory Object Allocation Source / Allocation Stack

Function / Memory Object Allocation Source / Allocation Stack	CPU Time	Memory Bound						Average Latency (cycles)	Loads▼	Stores
		L1 Bound	L2 Bound	L3 Bound	DRAM Bound		Other			
					Memory Bandwidth	Memory Latency				
miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Ve	104.645s 🟦	0.011	0.000	0.013	1.000	0.000	0.015	75	68,645,205,935	873,013,095
hbwmalloc.h:143								171	32,600,097,800	4,000,060
hbwmalloc::hbwmalloc_allocator<double>::allocate ← __gnu.								171	32,600,097,800	4,000,060
hbwmalloc.h:143								49	21,790,065,370	0
hbwmalloc::hbwmalloc_allocator<int>::allocate ← __gnu_cxx								49	21,790,065,370	0
hbwmalloc.h:143								23	11,020,033,060	842,012,630
hbwmalloc::hbwmalloc_allocator<double>::allocate ← __gnu.								23	11,020,033,060	842,012,630
hbwmalloc.h:143								7	3,125,009,375	0
hbwmalloc::hbwmalloc_allocator<int>::allocate ← __gnu_cxx								7	3,125,009,375	0

18 Memory Access analysis

Step No. 3: Re-Run the Benchmark

Running the modified version of the benchmark on Intel Xeon Phi coprocessor memory with MCDRAM we see that it executes almost four times faster than the original version of the benchmark allocating its intensively processed data in the traditional DRAM memory:

```
[vtune@nntvtune46 src]$ /usr/bin/time /tmp/miniFE-2.0_openmp_ref_ORIG/src/miniFE.x.sh
MiniFE Mini-App, OpenMP Peer Implementation
Creating OpenMP Thread Pool...
Counted: 12 threads.
Running MiniFE Mini-App...
      creating/filling mesh...0.197327s, total time: 0.197329
generating matrix structure...13.5858s, total time: 13.7832
         assembling FE data...13.3513s, total time: 27.1345
      imposing Dirichlet BC...2.61192s, total time: 29.7464
      imposing Dirichlet BC...1.11535s, total time: 30.8617
making matrix indices local...1.19209e-06s, total time: 30.8617
Starting CG solver ...
Initial Residual = 201.001
Iteration = 20   Residual = 0.0609161
…
Iteration = 200   Residual = 0.00112011
Final Resid Norm: 0.00112011
2671.76user 9.53system 3:55.02elapsed 1140%CPU (0avgtext+0avgdata 1511308maxresident)k0inputs+8out-
puts (0major+49614minor)pagefaults 0swaps

[vtune@nntvtune46 src]$ /usr/bin/time /tmp/miniFE-2.0_openmp_ref_KNL/src/miniFE.x.sh
MiniFE Mini-App, OpenMP Peer Implementation
Creating OpenMP Thread Pool...
Counted: 12 threads.
Running MiniFE Mini-App...
      creating/filling mesh...0.198685s, total time: 0.198686
```

```
generating matrix structure...13.9371s, total time: 14.1358
         assembling FE data...13.1823s, total time: 27.3181
      imposing Dirichlet BC...2.51502s, total time: 29.8331
      imposing Dirichlet BC...1.10896s, total time: 30.942
making matrix indices local...9.53674e-07s, total time: 30.942
Starting CG solver ...
Initial Residual = 201.001
Iteration = 20   Residual = 0.0609161
…
Iteration = 200   Residual = 0.00112011
Final Resid Norm: 0.00112011
475.87user 2.09system 0:52.25elapsed 914%CPU (0avgtext+0avgdata 2598752maxresident)k0inputs+8outputs
(0major+23143minor)pagefaults 0swaps
```

This example demonstrates how bandwidth-bound code can benefit from placing its most intensively processed data in the MCDRAM memory available on the latest Intel Xeon Phi platform, and how the memkind library can greatly simplify this task.

Conclusion

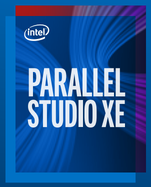
It is crucial to optimize the memory accesses of your program. Understanding how your program is accessing memory by using a tool such as Intel VTune Amplifier XE can greatly assist you in getting the most out of your hardware.

We showed an overview of the new Intel VTune Amplifier XE Memory Access analysis feature. We also showed how some tough memory problems could be resolved by using this feature.

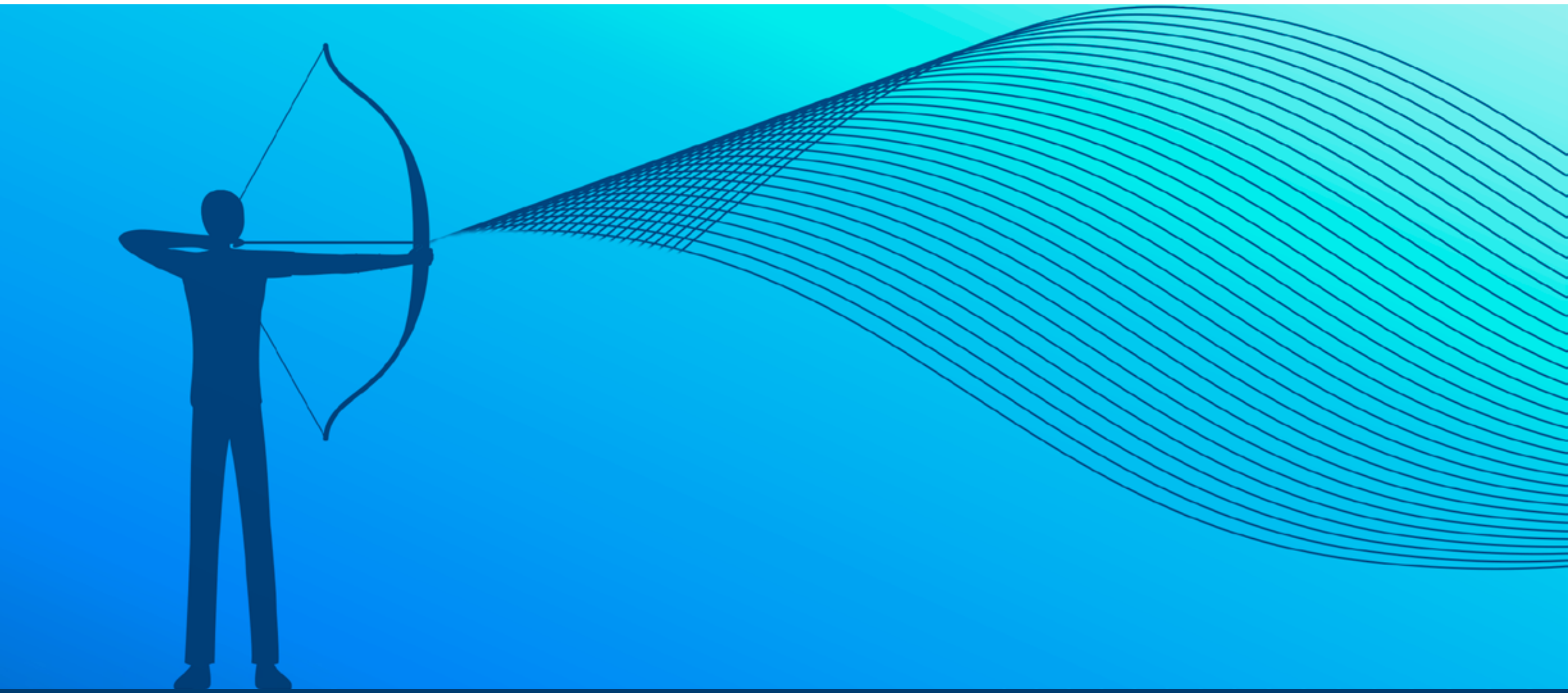
We showed how users could detect false sharing problems by seeing high average latency values for relatively small memory objects. We improved application performance by 4x with a trivial one-line code change by just padding a structure.

We showed how users could help detect NUMA issues with a significant amount of remote memory accesses and improved application performance by 2x after removing the remote access.

Finally, we demonstrated how finding the portions of your code that can take advantage of the memory technologies available on the latest Intel Xeon Phi platform can help increase the speed of the benchmark application 4x.



TRY INTEL® VTUNE™ AMPLIFIER XE
Part of Intel® Parallel Studio XE >



OPTIMIZING IMAGE IDENTIFICATION WITH INTEL® INTEGRATED PERFORMANCE PRIMITIVES

Tencent Speeds MD5 Image Identification by 2x

Yueqiang Lu, *Application Engineer,* Intel APAC R&D Ltd., and
Ying Hu, *Technical Consulting Engineer*, Intel APAC R&D Ltd.

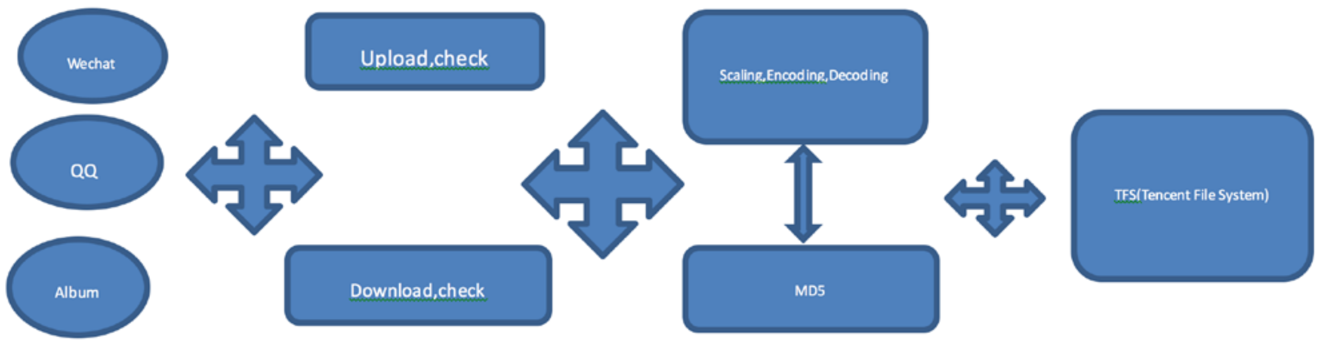
Tencent, Inc., is China's largest and most-used Internet service portal. It owns both the largest online game community and the largest Web portal (**qq.com**), as well as the No. 1 and No. 2 applications (WeChat*, QQ*) in China.

Every day, Tencent needs to process billions of new user-generated images from WeChat, QQ, and QQ Album*. Some hot applications even have hundreds of millions of images to be uploaded, stored, processed, and downloaded in a single day—which consumes vast computing resources.

To manage, store, and process these images, Tencent developed Tencent File System* (TFS*). But even with compression, the image volume reached hundreds of petabytes. Moreover, it is still growing explosively—and the supported cluster has more than 20,000 servers.

Technical Background

Based on TFS, the image processing system provides uploading, scaling, encoding, and downloading services. As an image uploads, TFS scales it into a different resolution and creates the related ID by Message Digest Algorithm 5 (MD5).[1] Next, the image is transcoded into WebP* format for storage. While downloading an image, the system must find the right place to read the image, and then transcode it into the user-required image format and resolution (**Figure 1**).



1 Tencent File System* image processing

Because the website has tons of visits each second, there's a small possibility that the image download component will read the wrong image. Avoiding this kind of error requires an MD5 calculation and check. However, this is a huge computing workload—so Tencent needed to maximize MD5 computing performance.

Originally, Tencent used the md5sum* utility tool along with the Operator* OS to compute the MD5 value for each image file. Intel worked closely with Tencent engineers to help them optimize performance with Intel® Integrated Performance Primitives (**Intel® IPP**)—which helped Tencent achieve a 100 percent performance improvement on the Intel® architecture-based platform.

Intel® Streaming SIMD Extensions and Software Optimization

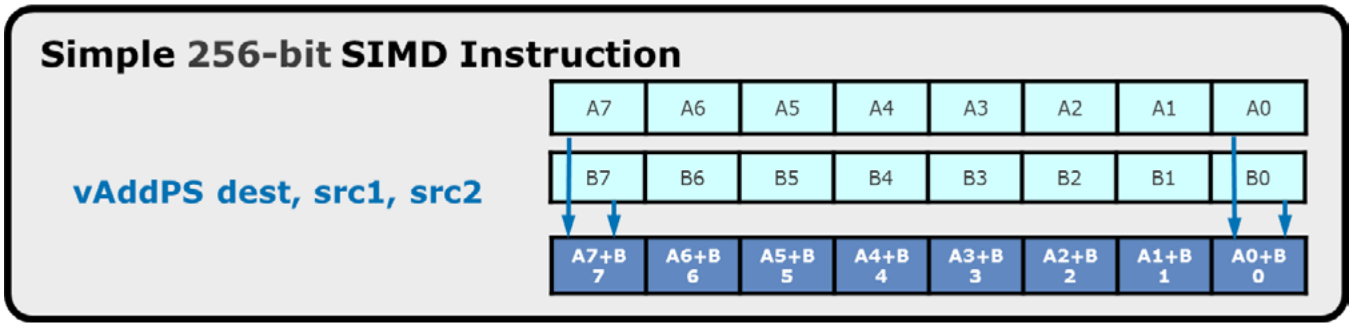
Intel introduced an instruction set extension with the Intel® Pentium® III processor called Intel® Streaming SIMD Extensions (Intel® SSE). This was a major redesign of an earlier single instruction, multiple data (**SIMD**) instruction set called MMX®, introduced with the Intel Pentium processor.

Intel evolved the Intel SSE instruction set along with Intel architecture, extending it by wider vectors and adding a new extensible syntax and rich functionality. The latest SIMD instruction set, Intel® Advanced Vector Extensions 2 (Intel® AVX2), can be found in the Intel® Core™ i7 processor.

Most of the Intel® Xeon® processors in the TFS system support Intel SSE2, one of the Intel® SIMD processor supplementary instruction sets. Intel SSE2 is supplemented by Intel SSE3, Intel SSE4.x, and Intel Advanced Vector Extensions (Intel AVX).

Intel AVX is a 256-bit instruction set extension to Intel SSE, designed to provide even higher performance for applications that are compute-intensive. Intel AVX adds new functionality to the Intel SIMD instruction set (based on Intel SSE) on floating-point and integer computing, and it includes a more compact SIMD instruction set.

Figure 2 shows one SIMD operation on 8 data (32-bit integer type, floating point type) instructions.



2 SIMD operation on 8 data instructions

Intel AVX improves performance by extending the breadth of vector processing capability across floating-point and integer data domains. This results in higher performance and more efficient data management across a wide range of applications such as image and audio/video processing, scientific simulations, financial analytics, and 3D modeling and analysis.

Algorithms That Benefit from Intel SSE

Algorithms that can benefit from Intel SSE[2] include those that employ logical or mathematical operations on data sets larger than a single 32-bit or 64-bit word. Intel SSE uses vector instructions, or SIMD architecture, to complete operations such as bitwise XOR, integer or floating-point multiply-and-accumulate, and scaling in a single clock cycle for multiple 32-bit or 64-bit words. Speed-up comes from the parallel operation and the size of the vector (multiword data) to which each mathematical or logical operator is applied.

Examples of algorithms that can significantly benefit from SIMD vector instructions include:

- **Image processing and graphics.** Both scale in terms of resolution (pixels per unit area) and the pixel encoding (bits per pixel to represent intensity and color) and both benefit from speedup relative to processing frame rates.
- **Digital signal processing (DSP).** Samples digitized from sensors and instrumentation have resolution-like images as well as data acquisition rates. Often, a time series of digitized data that is one-dimensional will still be transformed using algorithms, like a DFT (Discrete Fourier Transform), that operate over a large number of time series samples.
- **Digest, hashing, and encoding.** Algorithms used for security, data corruption protection, and data loss protection such as simple parity, CRC (cyclic redundancy check), MD5, SHA (secure hash algorithm), Galois math, Reed-Solomon encoding, and CBC (cypher-block-chaining) all make use of logical and mathematical operators over blocks of data, often many kilobytes in size.
- **Data transformation and data compression.** Most often, simulations in engineering and scientific computing involve data transformation over time and can include grids of data that are transformed. For example, in physical thermodynamic, mechanical, fluid-dynamic, or electrical-field models, a grid of floating-point values is used to represent the physical fields as finite elements. These finite element grids are then updated through mathematical transformations over time to simulate a physical process.

Intel IPP Optimized for Intel AVX

Intel IPP is a performance building block for all kinds of image and signal processing, data compression, and cryptography needs. These ready-to-use, royalty-free functions are highly optimized using Intel SSE, Intel AVX, Intel AVX2, and Intel AVX-512 instruction sets, which often outperform what an optimized compiler can produce alone.[3]

Table 1 summarizes the features of Intel IPP.

Optimized for Performance and Power Efficiency	**Intel Engineered and Future-Proofed to Shorten Development Time**	**Wide Range of Cross-Platform and OS Functionalitiesy**
Highly tuned routines	Fully optimized for current and past processors	Thousands of highly optimized signal, data, and media functions
Highly optimized using SSSE4, SSSE3, Intel® SSE, and Intel® AVX, Intel® AVX2, Intel® AVX-512 instruction sets	Saves development, debug, and maintenance time	Broad domain support
Performance beyond what an optimized compiler produces alone	Code once now, receive future optimizations later	Supports Intel® Quark™, Intel® Core™, Intel® Xeon®, and Intel® Xeon Phi™ platforms

Table 1. Intel® IPP features

AVX = Advanced Vector Extensions
SSE = Streaming SIMD Extensions
SSSE = Supplemental Streaming SIMD Extensions

The Intel IPP library is optimized for a variety of SIMD instruction sets. Besides the optimization, Intel IPP also provides an automatic "dispatching" mechanism, which can detect the SIMD instruction set that is available on the running processor and select the optimal SIMD instructions for that processor.

Table 2 shows processor-specific codes that Intel IPP uses.

Associated with Processor-Specific Libraries		
IA-32 Architecture	**IA- 64 Architecture**	**Description**
px	mx	Generic code optimized for processors with Intel® Streaming SIMD Extensions (Intel® SSE)
w7		Optimized for processors with Intel SSE2
	m7	Optimized for processors with Intel SSE3
v8	u8	Optimized for processors with Supplemental Streaming SIMD Extensions 3 (SSSE3), including the Intel® Atom™ processor
p8	y8	Optimized for processors with Intel SSE4.1
g9	e9	Optimized for processors with Intel® Advanced Vector Extensions (Intel® AVX) and Intel® Advanced Encryption Standard New Instructions (Intel® AES-NI)
h9	l9	Optimized for processors with Intel AVX2
	n0	Optimized for Intel AVX-512 on Knights Landing (F, CD, ER, PF)
	k0	Optimized for Intel AVX-512 on Intel® Xeon® (F, CD<BW, DQ, VL)

Table 2. Processor-specific codes

See **Understanding CPU Dispatching in the Intel® IPP Library** for more information on dispatching. For more information on Intel IPP functions optimized for Intel AVX, read the article **Intel® IPP Functions Optimized for Intel® AVX**.

MD5 in Intel IPP

Hash functions are used in cryptography with digital signatures and for ensuring data integrity. When used with digital signatures, a publicly available function hashes the message and signs the resulting hash value. The party that receives the message can then hash the message and check if the block size is authentic for the given hash value.

Hash functions are also referred to as “message digests” and “one-way encryption functions.” To ensure data integrity, hash functions are used to compute the hash value that corresponds to a particular input. Then, if necessary, you can check if the input data has remained unmodified. You can recompute the hash value again using the available input and compare it to the original hash value. Intel IPP has implemented the following hash algorithms for streaming messages:

- MD5 [RFC 1321]
- SHA-1
- SHA-224
- SHA-256
- SHA-384
- SHA-512 [FIPS PUB 180-2]

These algorithms are widely used in enterprise applications.

A Closer Look at MD5

MD5 is a widely used cryptographic hash function producing a 128-bit (16-byte) hash value, typically expressed in text format as a 32-digit hexadecimal number.

Although MD5 was considered as “cryptographically broken and unsuitable” in a strict environment, it has been widely used in the software world to provide some assurance that a transferred file has arrived intact. For example, file servers often provide a precomputed MD5 (known as md5sum) checksum for the files, so that a user can compare the checksum of the downloaded file to it. Most Linux*-based operating systems include md5sum utilities in their distribution packages.

The Intel IPP MD5 functions apply hash algorithms to digesting streaming messages. It uses a state context (for example, ippsSHA1State) as an operational vehicle to carry all necessary variables to manage the computation of the chaining digest value. For example, the primitive implementing the MD5 hash algorithm must use the ippsMD5State context. The function Init initializes (MD5Init) the context and sets up specified initialization vectors. Once initialized, the function Update (MD5Update) digests the input message stream with the selected hash algorithm until it exhausts all message blocks. The function Final (MD5Final) is designed to pad the partial message block into a final message block with the specified padding scheme. It then uses the hash algorithm to transform the final block into a message digest value.

Here is an example illustrating how the application code can apply the implemented MD5 hash standard to digest the input message stream:

1. Call the function MD5GetSize to get the size required to configure the ipps MD5State context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the MD5Init function to set up the initial context state with the MD5-specified initialization vectors.
3. Keep calling the function MD5Update to digest the incoming message stream in the queue until its completion. To determine the current value of the digest, call MD5GetTag between the two calls to MD5Update.
4. Call the function MD5Final for padding the partial block into a final MD5-1 message block and transform it into a 160-bit message digest value.
5. Clean up secret data stored in the context.
6. Call the operating system memory free service function to release the ippsMD5State context.

Intel engineers optimized Intel IPP functions mainly by the vectorization, or SSE instruction, and by extracting Intel architecture such as cache utilization, registers reutilization, etc.

With respect to Intel IPP MD5 implementation, the optimized technique is used:

- Fully unrolled code instead of tiny loop
- Using cyclic registers permutation instead of memory operations
- Coding rotations immediately instead of general parameterized 32-bit rotation

"Through close collaboration with Intel engineers, we adopted the Intel® Integrated Performance Primitives library for the image identification component in our online image storage and processing application. The application's performance improved significantly, and our cost of operations reduced greatly. We really appreciate the collaboration with Intel and are looking forward to more collaboration."

Nicholas, Leader of the TFS-Based Image Storage and Processing Team

The Intel IPP code replaced the md5sum. With the code shown in **Table 3**, no more manual optimization was needed.

md5sum code	IPP MD5

`md5sum  performance.PNG > hash.md5`
`cat hash.md5`
`0e5e74555b68db366c85b1b194f258fe`
`performance.PNG.`
 	
`The md5sum is along with Cent OS`
`distribution.`
	`char* intel_md5sum(const char *p,`
	`unsigned uiLen)`
	` {`
	`     static Ipp8u MD[32];`
	`     int size;`
	`     IppsMD5State *ctx;`
	`     ippsMD5GetSize(&size);`
	`     ctx = (IppsMD5State*)malloc(size);`
	`     IppStatus st = ippsMD5Init(ctx);`
	`     st = ippsMD5Update((const Ipp8u *)`
	`p, (int)uiLen, ctx);`
	`     st = ippsMD5Final(MD, ctx);`
	`     free(ctx);`
	`     return (char*)MD;`
	`}`

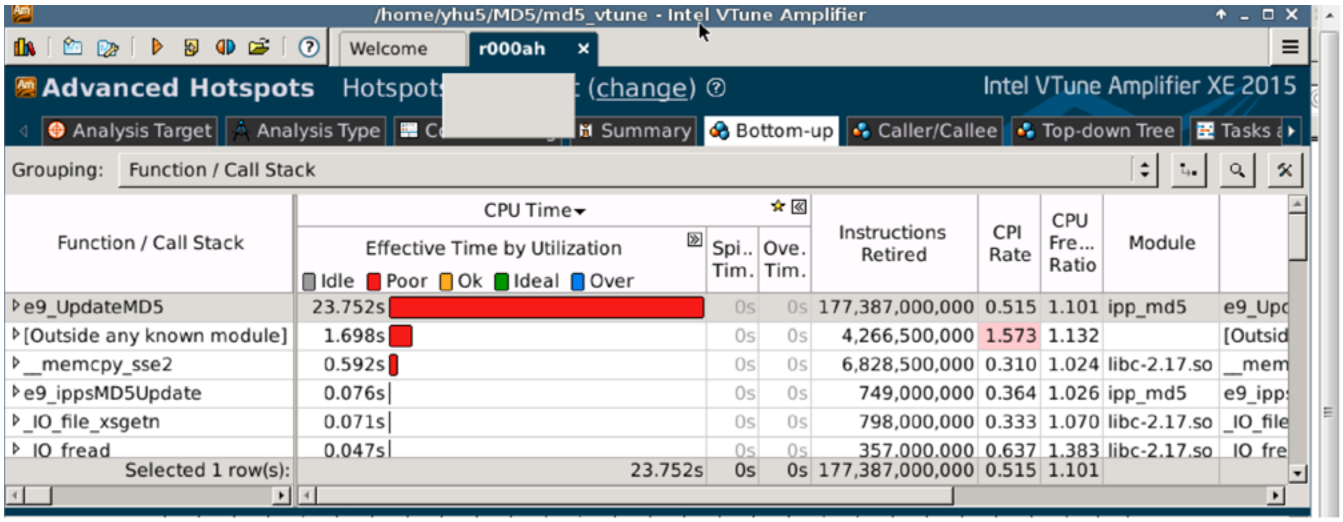
Table 3. Code example

Invoking Intel IPP to Accelerate MD5

The Intel IPP MD5 code, `md5test.cpp`, is compiled using gcc as follows:

```
[root@localhost code]# make -f Makefile.gcc
g++ -O2 ipp_md5.cpp —o ipp_md5
-I/opt/intel/compilers_and_libraries_2016.0.109/linux/ipp/include
/opt/intel/compilers_and_libraries_2016.0.109/linux/ipp/lib/intel64/libippcp.a
/opt/intel/compilers_and_libraries_2016.0.109/linux/ipp/lib/intel64/libippcore.a
```

This integrates the IPP crypto library into the program and extracts performance from the computing resources automatically. **Figure 3** is a screen shot of Intel® VTune™ Amplifier XE running the ipp_md5 program. It shows the ipp function e9_ippsMD5Update takes most of the CPU time of the program where e9 (AVX-optimized) code was running.


/home/yhu5/MD5/md5_vtune - Intel VTune Amplifier
Welcome
r000ah
Advanced Hotspots Hotspots [redacted] (change)
Intel VTune Amplifier XE 2015
Analysis Target
Analysis Type
Summary
Bottom-up
Caller/Callee
Top-down Tree
Tasks

Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (▨ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	CPU Time: Spi.. Tim.	CPU Time: Ove. Tim.	Instructions Retired	CPI Rate	CPU Fre... Ratio	Module	
e9_UpdateMD5	23.752s	0s	0s	177,387,000,000	0.515	1.101	ipp_md5	e9_Upc
[Outside any known module]	1.698s	0s	0s	4,266,500,000	1.573	1.132		[Outsid
__memcpy_sse2	0.592s	0s	0s	6,828,500,000	0.310	1.024	libc-2.17.so	__mem
e9_ippsMD5Update	0.076s	0s	0s	749,000,000	0.364	1.026	ipp_md5	e9_ipp
_IO_file_xsgetn	0.071s	0s	0s	798,000,000	0.333	1.070	libc-2.17.so	_IO_file
_IO_fread	0.047s	0s	0s	357,000,000	0.637	1.383	libc-2.17.so	_IO_fre
Selected 1 row(s):	23.752s	0s	0s	177,387,000,000	0.515	1.101		

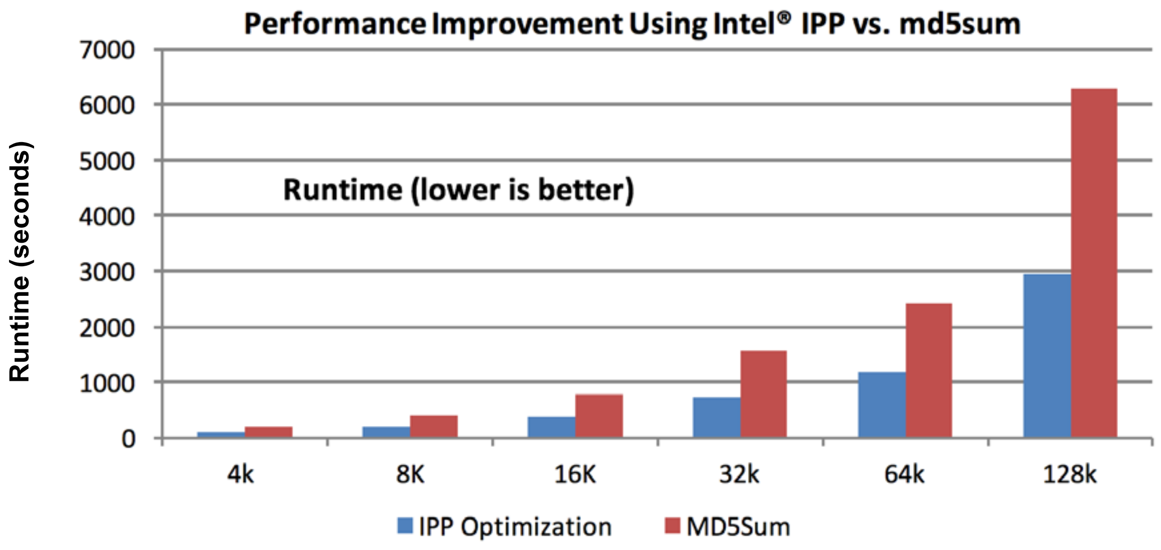
3 Intel® VTune™ Amplifier XE running the ipp_md5 program

Performance Data

The test was run based on different sizes of image files using Intel IPP and md5sum provided by the Linux OS. Using 10,000 iterations resulted in the performance shown in **Table 4** and **Figure 4**.

Processor OS	Test Images Size	md5sum (Average Time for 10,000 Iterations)	IPP_Md5 (Average Time for 10,000 Iterations)
Intel® Xeon® Processor E5-2620 (15M Cache, 2.00 GHz, 7.20 GT/s Intel® QuickPath Interconnect, Intel® Advanced Vector Extensions-supported CentOS 6.5)	4K	206ms	95ms
	8K	406ms	189ms
	16K	789ms	369ms
	32K	1574ms	740ms
	64K	2420ms	1183ms
	128K	6273ms	2943ms

Table 4. Test run performance results



Configuration Info - Versions: Intel® IPP 9.0.0, Hardware: Intel® Xeon® Processor E5-2620 (15M Cache, 2.00 GHz, 7.20 GT/s Intel® QPI) Memory: 64G Operating System: CentOS8 6.5 x86_64; Benchmark Source: Test Image files.
*Other brands and names are the property of their respective owners.

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.

4 Test run performance

On the Intel® Xeon® processor E5-2620 (15M Cache, 2.00 GHz, 7.20 GT/s Intel® QuickPath Interconnect, Intel AVX-supported), comparing the md5sum along with Linux showed a 100 percent performance improvement. Tencent engineers also implemented Intel IPP MD5 for their online system. Their test showed about a 60 percent performance improvement compared to the original MD5.

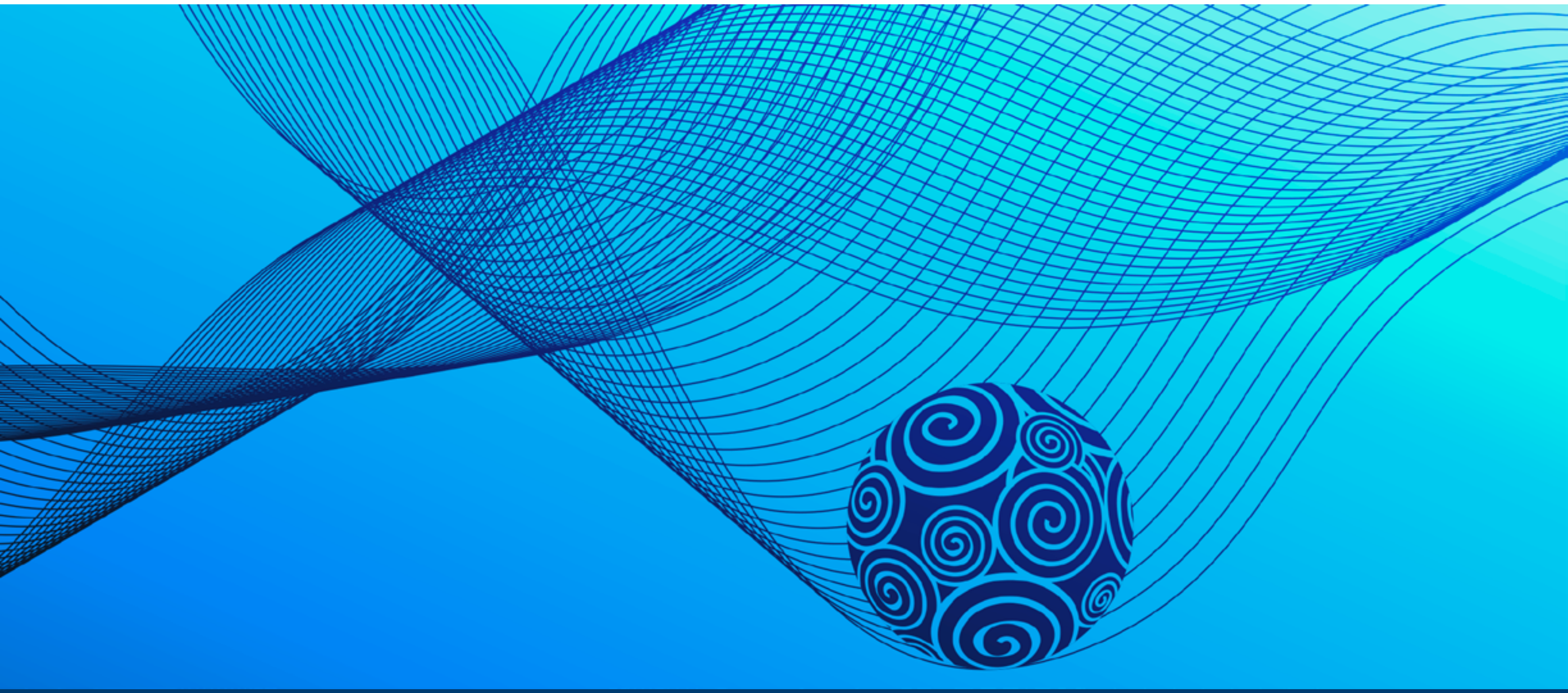
Conclusion

Tencent has billions of new user-generated images to process every day from WeChat, QQ, and QQ Album. All images are handled by the TFS-based image storage and processing system. Tencent has to give each image a unique ID by MD5 hash. Intel worked with Tencent engineers to optimize this function component using Intel IPP, achieving a 2x performance improvement.

Methods for improving the speed of computing the md5sum of images is straightforward with Intel IPP. This work demonstrates significant progress toward being able to handle these computationally intensive methods by optimizing them for the latest Intel® hardware using the Intel IPP and performance-tuning methodologies.

References

1. **md5sum on Wikipedia.**
2. **Using Intel® Streaming SIMD Extensions and Intel® Integrated Performance Primitives to Accelerate Algorithms.**
3. **Intel® AVX Realization of IIR Filter for Complex Float Data.**



DEVELOP SMARTER USING THE LATEST IOT AND EMBEDDED TECHNOLOGY

Go from Great Idea to Great Product with Intel® System Studio

Noah Clemons, ***Technical Consulting Engineer, Parallel Programming Products,*** **Intel Corporation**

It takes a lot of work to turn a great idea into a great product. And having the right tools for the job makes the whole process much easier. Intel is providing those tools to meet all your embedded system development needs with Intel® System Studio and Intel® System Studio for Microcontrollers. This unified set of **software development tools** for coding, analysis, and debugging works with all Intel® microcontrollers, Internet of Things (IoT) devices, and embedded platforms.

In this article, we'll:

- Provide a quick overview of Intel System Studio and Intel System Studio for Microcontroller components
- Discuss how the components work across all platforms
- Describe where the components have been adapted to accommodate the new microcontroller platform
- Explain how Intel System Studio addresses IoT needs as well as embedded development

Who Needs Intel System Studio?

Intel System Studio is for:

- Device manufacturers who need the right tools to bring life into platforms
- System integrators who rely on existing platforms and need to establish a full system software stack
- Embedded software developers who need to build and optimize dedicated applications

Build Exciting Products

Intel System Studio focuses on providing all the tools you need to develop exciting products on Intel® hardware, supporting Intel's embedded platforms from Intel® Quark™ X1000 and Intel® Atom™ processor-based IoT gateways through Intel® Core™ and Intel® Xeon® processor-based servers.

Intel System Studio for Microcontrollers is specifically built to provide you with a customized development environment focused on microcontrollers for the Intel® Quark™ microcontroller D1000, D2000, and SE (coming soon).

BLOG HIGHLIGHTS

Device Selection

BY ALEX KATRANOV >

This post continues a series of articles that describes the opencl_node, a new node available in the Intel® Threading Building Blocks (Intel® TBB) library since version 4.4 Update 2. This node allows OpenCL™ powered devices to be more easily utilized and coordinated by an Intel TBB flow graph. The first article in this series can be found here.

In the previous article, I described the basic interfaces. In this posting I discuss selecting devices to use for execution of a kernel.

Read more >

Table 1 shows some of the features these suites provide.

	Intel System Studio	Intel System Studio for Microcontrollers
Platforms	Intel® Xeon® processor, Intel® Core™ processor, Intel® Atom™ processor, Intel® Quark™ U-series SoC	Intel® Quark™ D-series microcontroller
Software Development Environment	Eclipse-based integrated development environment (IDE), Command Line*, Intel® Graphics Performance Analyzers	Eclipse-based IDE, Command Line
Host Systems	Linux*, Windows*, OS X*[1]	Linux*, Windows*
Compilers	Intel® C++ Compiler	Intel® C++ Compiler or GNU* C Compiler[2]
Target Platform OS	Linux, Android*, Windows, FreeBSD*, VxWorks*	Bare metal, Real Time* OS
Target Platform Software	Samples, debugger, and profiler support drivers	Board support package, Intel® Quark™ microcontroller software interface, samples
Libraries	Intel® Math Kernel Library, Intel® Performance Primitives, and Intel® Threading Building Blocks (Image, Signal Math, Data Processing, Multithreading)	C Runtime, Floating-Point Emulation, and DSP libraries
Analyzers	Intel® VTune™ Amplifier, Energy Profiler, Inspector (Memory Analyzer)	Power Analyzer[3]
Debuggers	Applications and OS, WinDbg* Kernel debugger, Intel®-enhanced GDB, Intel® System Debugger, JTAG, JTAG over USB, UEFI Agent	Application and OS, Embedded System Registers View, MCU Flashing Intel®-enhanced GDB, OpenOCD*-based JTAG

1. Intel® System Studio supports some, but not all OS X* features.
2. Intel® Quark™ D1000 MCU is supported by the LLVM-based Intel® Compiler; Intel® Quark™ D2000 and Intel Quark SE MCUs are supported by GCC.
3. Power Analyzer for MCUs is coming soon.

Table 1. Intel® System Studio and Intel® System Studio for Microcontrollers capabilities

Intel System Studio

With Intel System Studio, you can choose between using the command line or GUI-based tools including full Eclipse* or Microsoft Visual Studio* integration. These tools support targeting Windows*, Linux*, VxWorks*, Wind River Linux*, FreeBSD*, and Android*, and support the latest versions of Intel® processors including Intel Quark, Intel® Edison platform, Intel Atom x3 processors (formerly code-named SoFI), Intel Atom x5, x7 processors (formerly code-named Cherry Trail), 6th generation Intel Core processors (formerly code-named Skylake), Microsoft Windows 10, and FreeBSD.

The backbone capability of Intel System Studio is an optimizing compiler, the Intel® C/C++ Compiler, and libraries with enhanced C++11 and C++14 (-std=c++14) feature support—generating tailored code for your architecture as well as the ability to analyze your system. The three main libraries are Intel® Integrated Performance Primitives (Intel® IPP), Intel® Math Kernel Library (Intel® MKL), and Intel® Threading Building Blocks (Intel® TBB).

Intel IPP provides the performance building blocks for image, signal, and string processing; data compression; cryptography; and computer vision through an extensive library of software functions. Supported by both Intel System Studio and Intel System Studio for Microcontrollers, these libraries provide additional optimizations for Intel Quark, Intel Atom, and Intel Core processors.

These ready-to-use, royalty-free functions are highly optimized using Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX, Intel® AVX2, and Intel® AVX-512) instruction sets, which often outperform what an optimized compiler can produce alone.

Optimized for Performance and Power Efficiency	Intel Engineered and Future-Proofed to Shorten Development Time	Wide Range of Cross-Platform and OS Functionalities
Highly tuned routines	Fully optimized for current and past processors	Thousands of highly optimized signal, data, and media functions
Highly optimized using SSSE4, SSSE3, SSE, and AVX, AVX2, AVX-512 instruction sets	Save development, debug, and maintenance time	Broad domain support
Performance beyond what an optimized compiler produces alone	Code once now, receive future optimizations later	Supports Intel® Quark™, Intel® Core™, Intel® Xeon®, and Intel® Xeon® platforms

Table 2. Benefits of Intel® System Studio

Intel MKL speeds math processing in scientific, engineering, and financial applications. Version 11.2 added Parallel Direct Sparse Solver for Clusters, Verbose mode for BLAS, and LAPACK, S/C/Z/DGEMM improvements on small matrix sizes, significant SVD and eigensolvers performance improvements, and other features and optimizations to expand this library further.

Intel TBB is a widely used C++ template library for task parallelism. It provides parallel algorithms and data structures, threads and synchronization primitives, and scalable memory allocation and task scheduling. Version 4.3 also provides Memory Allocator improvements (improved tbbmalloc to increase performance and scalability for threaded applications), improved Intel® Transactional Sychronization Extensions (Intel® TSX) support (applications that use read-write locks can take additional advantage of Intel TSX via tbb::speculative_spin_rw_mutex), improved compatibility with the C++ 11 standard, tasks arenas (improved control over workload isolation and the degree of concurrency with new class tbb::task_arena), and support for the latest Intel® architecture. (See the Intel TBB **release notes** for a hardware support matrix.)

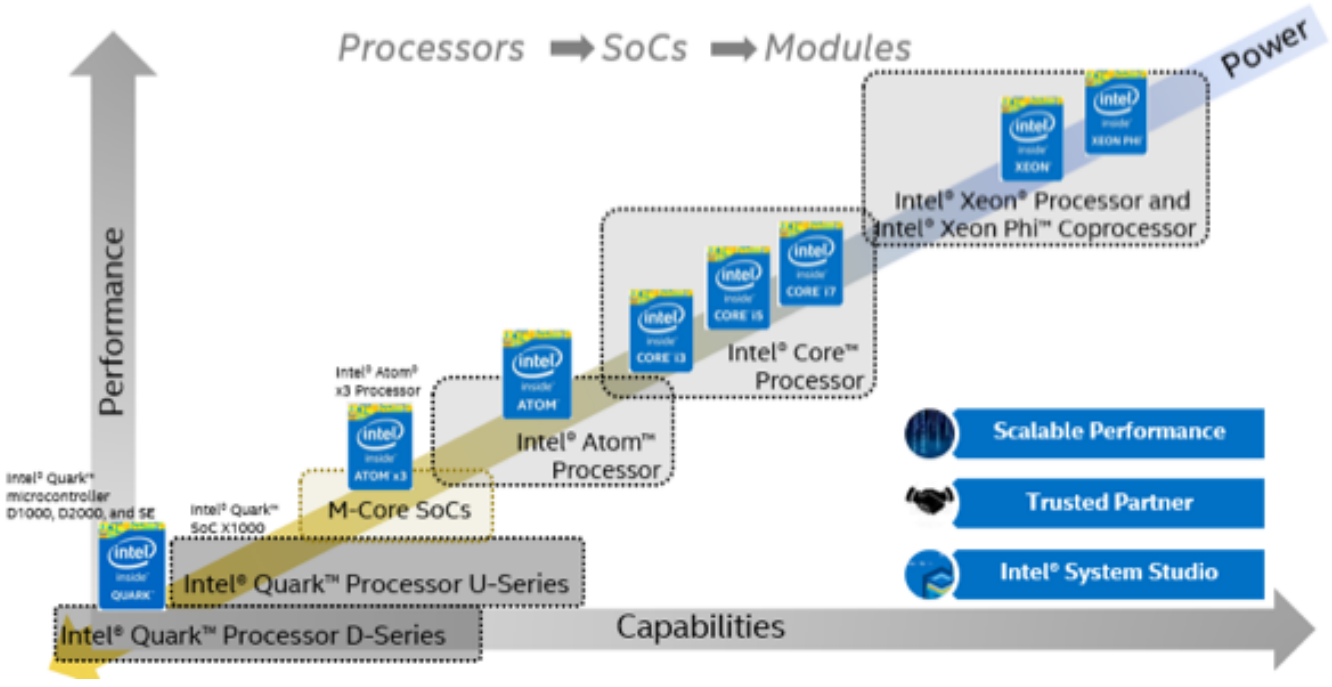
Intel® VTune™ Amplifier is an essential performance profiling and power analysis tool. It enables you to quickly and easily get the tuning data you need for a wide array of analysis types, such as performing a power analysis on an IoT gateway or a modem-based platform.

Intel® System Debugger offers JTAG debugging support through USB, providing an array of tools to meet your debugging needs. The first tool is the JTAG debug and instruction trace to Microsoft WinDbg* kernel debugger (Intel System Studio for Windows target version). This helps to isolate tricky Windows driver issues during board bring-up and includes Intel® Processor Trace support in the WinDbg kernel debugger to help isolate complex runtime issues.

Intel® Trace Hub performs system tracing to capture system-wide hardware and software events. (It supports 6th generation Intel Core processors.) This helps you understand complex interactions between hardware and software faster and offers time-stamp correlated trace information.

Closed Chassis Debug is JTAG-based debug and trace over a low-cost USB connection. This lets you develop with the form factor that meets your product goals without exposing a debug port. Instead, a USB port can be wired to support all your debugging needs.

Figure 1 shows the tools' end-to-end scalability for IoT projects.


Processors ➡ SoCs ➡ Modules
Power
Performance
Capabilities
Intel® Xeon® Processor and Intel® Xeon Phi™ Coprocessor
Intel® Core™ Processor
Intel® Atom™ Processor
Intel® Atom® x3 Processor
M-Core SoCs
Intel® Quark™ microcontroller D1000, D2000, and SE
Intel® Quark™ SoC X1000
Intel® Quark™ Processor U-Series
Intel® Quark™ Processor D-Series
Scalable Performance
Trusted Partner
Intel® System Studio

1 IoT end-to-end scalability

Intel® System Studio for Microcontrollers

Intel System Studio for Microcontrollers is a new tool suite supporting Intel Quark D1000, Intel Quark D2000, and Intel Quark SE microcontrollers. The suite offers an Eclipse-based integrated development environment. Developers can also run the tools in Intel System Studio for Microcontrollers from the command line using the Make utility.

Intel System Studio for Microcontrollers supports creating code to run on bare-metal systems or on select real-time operating systems (on D2000 and SE microcontrollers). It includes a board support package (BSP) that eliminates the need for writing the bootstrap code and simplifies I/O functions.

In addition to BSP for Intel Quark D2000 and Intel Quark SE microcontrollers, Intel System Studio for Microcontrollers includes the Quark™ Microcontroller Software Interface (QMSI) package, which includes implementation for most I/O interfaces (e.g., GPIOs, analog input, I2C, SPI, UART) and also includes device drivers for peripherals found on customer reference boards (e.g., Bosch BMC150* accelerometer).

Intel System Studio for Microcontrollers comes with floating-point emulation and DSP libraries, which are highly optimized for code size (typically less than 1KB per function) with an emphasis on performance, accuracy, and low power consumption. The DSP library, based on the popular CMSIS-DSP* library, includes basic math, fast math, complex math, statistics, transform, interpolation, and matrix functions. The Intel-optimized LibM includes some most frequently used single-precision functions such as sqrtf, expf, logf, sinf, cosf, sincosf, tanf, asinf, acosf, atanf, floorf, ceilf, and truncf. Compared to the GNU* standard C math library, it offers up to 10x better performance and up to 5x smaller code size.

Intel System Studio integrates debugger and MCU firmware flashing support using open source GDB and OpenOCD* software. On the hardware side, the JTAG interfacing is done using simple and cost-effective FTDI FT232H and FTDI FT2232H USB to JTAG/UART adapters.

Cross-Platform Tools

Regardless of what platform you're targeting, Intel System Studio has a well-defined set of tools that work seamlessly across the wide variety of embedded, mobile, wearable, IoT, and now microcontroller platforms. You can use these tools from very small to large many-core applications. While this article focused on the compiler and libraries, the analysis tools work across the same range of platforms and are worth understanding as well.

Learn more

- **Intel® System Studio**
- **Intel® System Studio for Microcontrollers**
- **Intel® Math Kernel Library**
- **Intel® Performance Primitives**
- **Intel® System Debugger**
- **Intel® Threading Building Blocks**
- **Intel® vTune Amplifier**

TUNING HYBRID APPLICATIONS WITH INTEL® CLUSTER TOOLS

Understanding MPI Utilization Inefficiencies and Balancing Thread Level Loads

Alexey Malhanov, *Software Development Engineer,* and Dmitry Prohorov, *Software Engineering Manager,* Intel Corporation

Modern many-core processors such as Intel® Xeon Phi™ products can provide balanced performance-per-watt numbers to build efficient high-performance computing systems. But the growing number of cores—with the relatively slow growth of memory size and increasing complexity of memory hierarchy—might be a limiting scalability factor for pure **MPI** applications. The need for data replication in MPI ranks and increasing MPI buffers, plus intensive intra-node communications for hundreds of cores, can limit application performance. Hybrid programming models of MPI + X, where "X" utilizes shared memory models for intra-node communications, can reduce the bottlenecks and increase the scalability of the application.

One of the most popular hybrid programming models is MPI + OpenMP*, since pragma-based OpenMP is relatively easy to use to introduce thread-level **parallelism** and is based on industry standards. On the other hand, the challenge of adding a programming model requires performance analysis tools that are MPI and "X"-aware to understand scalability issues.

This article describes the capabilities of Intel® VTune™ Amplifier XE and Intel® Trace Analyzer and Collector, which are part of Intel® Parallel Studio XE Cluster Edition, for tuning MPI + OpenMP applications. As an example, we'll consider a life science application, heart_demo, which simulates electrophysiological heart activity with the help of Runge-Kutta and finite elements methods.

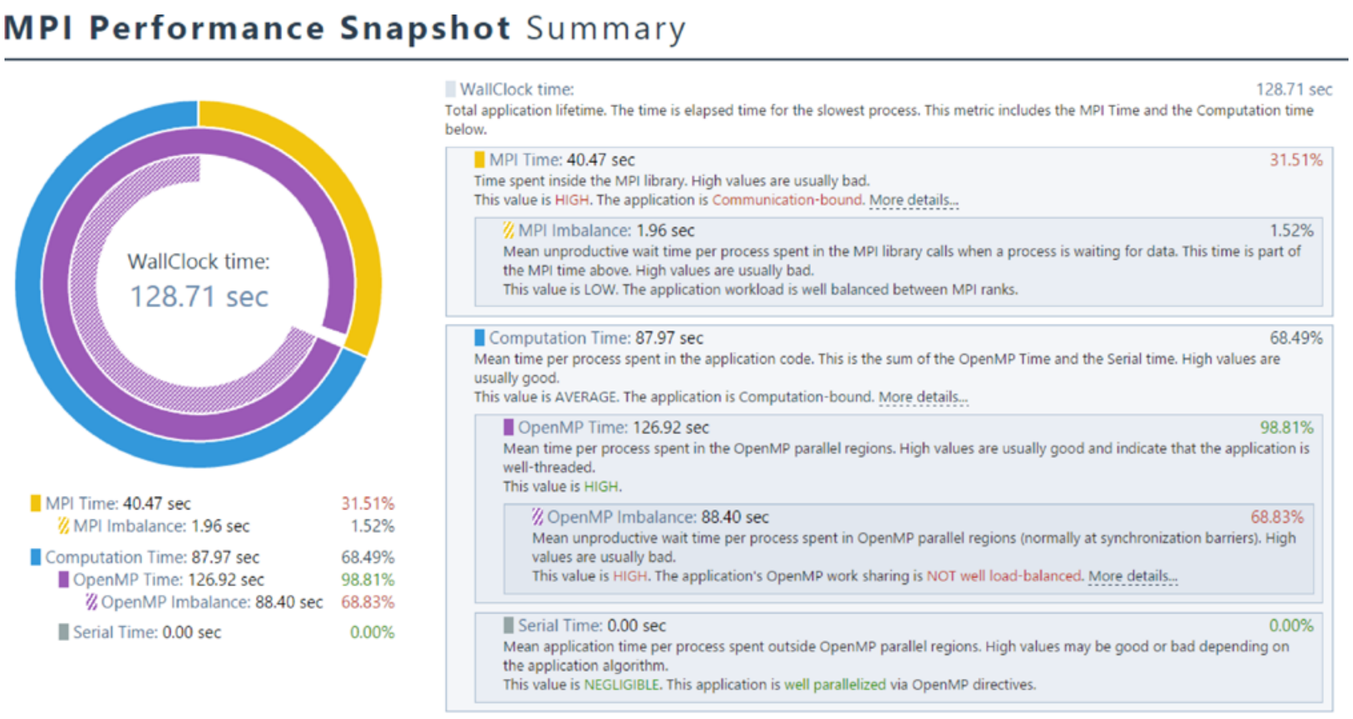
Hybrid Application Tuning: Where to Begin?

For any developer who creates applications that use both MPI and OpenMP, the goal is to develop a performant application that will use the cluster time allotted to the job as efficiently as possible. The Intel® MPI Performance Snapshot utility, a feature of Intel® Trace Analyzer and Collector, can be considered as an entry point for that. Here, we eliminate the aspect of optimal processes/threads ratio identification and concentrate on application efficiency improvement for a given amount of resources. For this article, the cluster we are using has four nodes with two sockets and 36 CPUs per socket and can launch one process per socket to avoid NUMA effects. Thus, we have eight MPI processes with 36 OpenMP threads each.

The MPI Performance Snapshot utility is a lightweight utility that gives us the overview of the application performance. It depicts inefficiencies connected with both MPI and OpenMP and suggests ways to explore further and deeper with the additional analysis capabilities of ITAC in cases where an application is MPI-bound, or to Intel VTune Amplifier XE if the application has weaknesses in OpenMP utilization. To enable the statistics-gathering with the help of the MPI Performance Snapshot utility, simply add the "-mps" option in a command line for `mpirun` execution:

```
source mpivars.sh
source mpsvars.sh
mpirun -mps -n 8 -ppn 2 -hosts host1,…,host4 ./heart_demo [parameters]
mps stats.txt app_stat.txt -O report_initial.html
```

The first two lines are needed for setting up the necessary environment. The third launches the application with the MPI Performance Snapshot utility profiling. And the last forms the report.



MPI Performance Snapshot Summary

▮ MPI Time: 40.47 sec	31.51%
▨ MPI Imbalance: 1.96 sec	1.52%
▮ Computation Time: 87.97 sec	68.49%
▮ OpenMP Time: 126.92 sec	98.81%
▨ OpenMP Imbalance: 88.40 sec	68.83%
▮ Serial Time: 0.00 sec	0.00%

WallClock time: 128.71 sec
Total application lifetime. The time is elapsed time for the slowest process. This metric includes the MPI Time and the Computation time below.

MPI Time: 40.47 sec 31.51%
Time spent inside the MPI library. High values are usually bad.
This value is HIGH. The application is Communication-bound. More details...

MPI Imbalance: 1.96 sec 1.52%
Mean unproductive wait time per process spent in the MPI library calls when a process is waiting for data. This time is part of the MPI time above. High values are usually bad.
This value is LOW. The application workload is well balanced between MPI ranks.

Computation Time: 87.97 sec 68.49%
Mean time per process spent in the application code. This is the sum of the OpenMP Time and the Serial time. High values are usually good.
This value is AVERAGE. The application is Computation-bound. More details...

OpenMP Time: 126.92 sec 98.81%
Mean time per process spent in the OpenMP parallel regions. High values are usually good and indicate that the application is well-threaded.
This value is HIGH.

OpenMP Imbalance: 88.40 sec 68.83%
Mean unproductive wait time per process spent in OpenMP parallel regions (normally at synchronization barriers). High values are usually bad.
This value is HIGH. The application's OpenMP work sharing is NOT well load-balanced. More details...

Serial Time: 0.00 sec 0.00%
Mean application time per process spent outside OpenMP parallel regions. High values may be good or bad depending on the application algorithm.
This value is NEGLIGIBLE. This application is well parallelized via OpenMP directives.

1 MPI Performance Snapshot Summary

For the application, we get the result shown in **Figure 1**.

From this summary, what conclusions can we make? First, our application spends one-third of the overall execution time in the MPI library. This might not be efficient, since part of this time could be spent on calculation, thus reducing overall execution time. Secondly, despite the fact that the MPI Performance Snapshot utility indicated our application is well-threaded, the OpenMP imbalance is high. This means the workload is distributed irregularly among the threads in parallel regions.

Exploring an MPI-Bound Application with the Help of ITAC

There are three key reasons for an application to be MPI-bound:

1. High wait times inside the MPI library. This occurs when a process waits for the data from other processes. This case is characterized with high values of MPI Imbalance indicator (**Figure 1**).
2. Active communications.
3. Poor or incorrectly set optimization settings of the library.

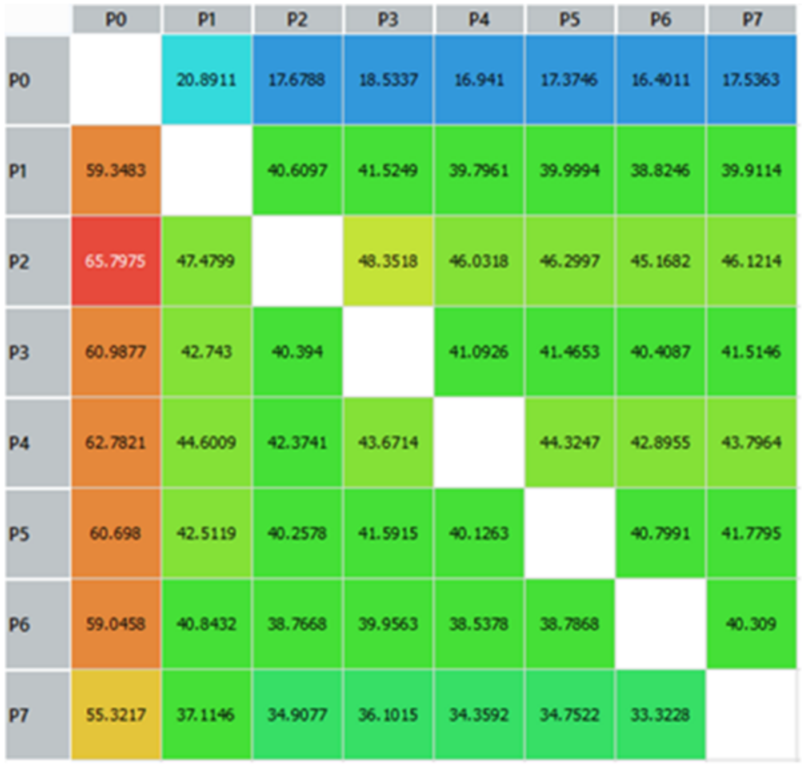
To reduce the impact of the first and second items, the user may consider the communication pattern for restructuring. The third item could be improved with the help of the mpitune utility, which is a part of Intel® MPI Library package.

Now, for further analysis on the MPI communications, let us use ITAC to dive deep into MPI-related problems. To enable the statistics-gathering within ITAC, the user should add the `-trace` option in a command line for `mpirun` execution:

```
source mpivars.sh
source itacvars.sh
mpirun -trace -n 8 -ppn 2 -hosts host1,…,host4 ./heart_demo [parameters]
```

The commands to set up ITAC are similar to those used for the MPI Performance Snapshot utility in the first round of analysis performed. The change in the third line represents the command line for the application execution with ITAC profiling enabled. Once the profiling has been completed, you can see the results in the ITAC graphical user interface.

ITAC is broad and rich with functionality for exploring the efficiency of MPI utilization. **Figure 2** shows the message profile chart.

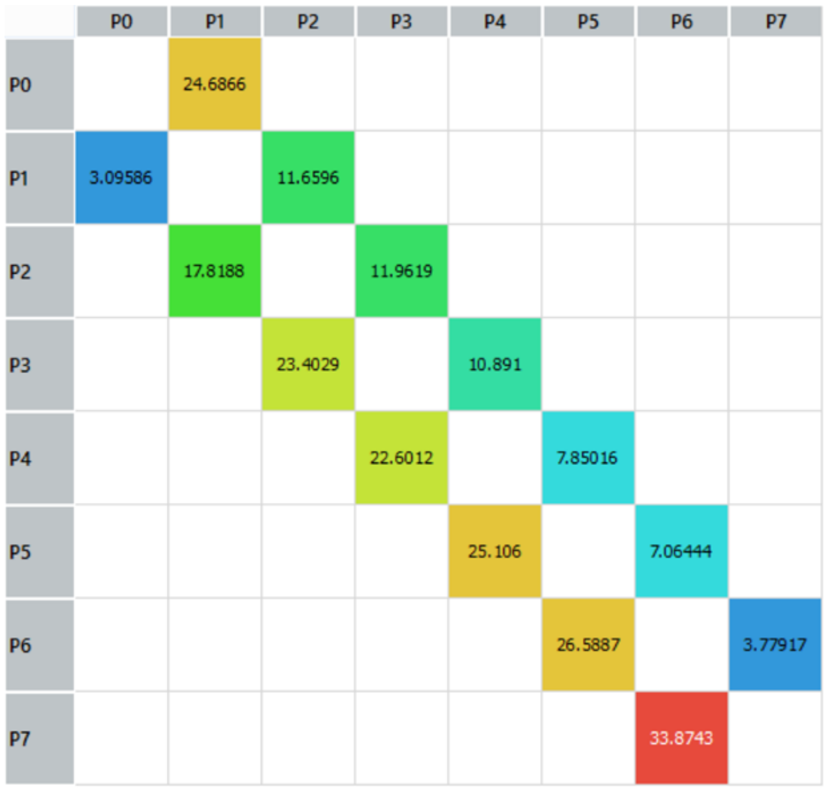


	P0	P1	P2	P3	P4	P5	P6	P7
P0		20.8911	17.6788	18.5337	16.941	17.3746	16.4011	17.5363
P1	59.3483		40.6097	41.5249	39.7961	39.9994	38.8246	39.9114
P2	65.7975	47.4799		48.3518	46.0318	46.2997	45.1682	46.1214
P3	60.9877	42.743	40.394		41.0926	41.4653	40.4087	41.5146
P4	62.7821	44.6009	42.3741	43.6714		44.3247	42.8955	43.7964
P5	60.698	42.5119	40.2578	41.5915	40.1263		40.7991	41.7795
P6	59.0458	40.8432	38.7668	39.9563	38.5378	38.7868		40.309
P7	55.3217	37.1146	34.9077	36.1015	34.3592	34.7522	33.3228	

2 Message profile chart

The cell with i-th row and j-th column in the chart depicts the communication time between processes with numbers I and j respectively. From the chart, we can conclude that each of the processes communicates with the others. Processes with numbers 1, 2, 3, 4, 5, 6, and 7 have relatively large communication time with process 0. Such a picture is typical for a communication pattern where one of the processes (in this case, with number 0) is a so-called "master" process that distributes the workload between others and gather the results of calculations.

From **Figure 2**, we can conclude that the reason for inefficient MPI utilization may be a too-active communication scheme. To overcome the problem, let us reconsider the communication pattern.



	P0	P1	P2	P3	P4	P5	P6	P7
P0		24.6866						
P1	3.09586		11.6596					
P2		17.8188		11.9619				
P3			23.4029		10.891			
P4				22.6012		7.85016		
P5					25.106		7.06444	
P6						26.5887		3.77917
P7							33.8743	

3 Message profile chart after improvements

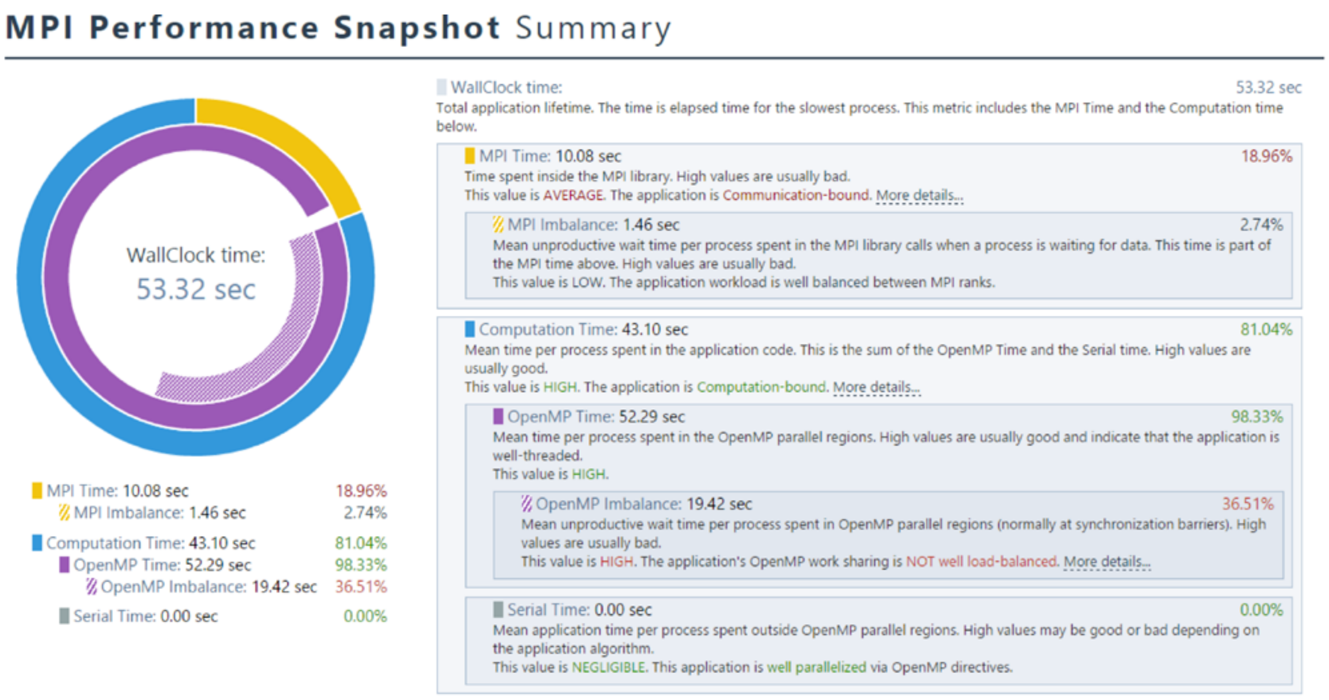
For this case, we managed to reorganize the data and change the communication pattern so that each process commutates only with the next and previous processes.

After the improvements, the message profile chart looks like **Figure 3**.

The communication matrix changed, becoming diagonal, and communication time between processes drastically improved.

The MPI Performance Snapshot utility results show the dramatic improvement in the efficiency: reorganizing the data and optimizing the communication patterns, we managed to speed up the application more than two times (**Figure 4**).

However, the tool still indicates that the application is MPI-bound, which means there is still room for refinement from the MPI point of view. Nevertheless, it is enough for our purposes and we further concentrate on OpenMP utilization tuning in the application.



MPI Performance Snapshot Summary

🟨 MPI Time: 10.08 sec	18.96%
▨ MPI Imbalance: 1.46 sec	2.74%
🟦 Computation Time: 43.10 sec	81.04%
🟪 OpenMP Time: 52.29 sec	98.33%
▨ OpenMP Imbalance: 19.42 sec	36.51%
■ Serial Time: 0.00 sec	0.00%

WallClock time: 53.32 sec
Total application lifetime. The time is elapsed time for the slowest process. This metric includes the MPI Time and the Computation time below.

MPI Time: 10.08 sec 18.96%
Time spent inside the MPI library. High values are usually bad.
This value is AVERAGE. The application is Communication-bound. More details...

MPI Imbalance: 1.46 sec 2.74%
Mean unproductive wait time per process spent in the MPI library calls when a process is waiting for data. This time is part of the MPI time above. High values are usually bad.
This value is LOW. The application workload is well balanced between MPI ranks.

Computation Time: 43.10 sec 81.04%
Mean time per process spent in the application code. This is the sum of the OpenMP Time and the Serial time. High values are usually good.
This value is HIGH. The application is Computation-bound. More details...

OpenMP Time: 52.29 sec 98.33%
Mean time per process spent in the OpenMP parallel regions. High values are usually good and indicate that the application is well-threaded.
This value is HIGH.

OpenMP Imbalance: 19.42 sec 36.51%
Mean unproductive wait time per process spent in OpenMP parallel regions (normally at synchronization barriers). High values are usually bad.
This value is HIGH. The application's OpenMP work sharing is NOT well load-balanced. More details...

Serial Time: 0.00 sec 0.00%
Mean application time per process spent outside OpenMP parallel regions. High values may be good or bad depending on the application algorithm.
This value is NEGLIGIBLE. This application is well parallelized via OpenMP directives.

4 Dramatic improvements in efficiency

Reducing OpenMP Imbalance with the Help of Intel VTune Amplifier XE

The MPI Performance Snapshot utility shows several metrics that allow assessing OpenMP parallelization efficiency. The collection uses statistics generated from the Intel OpenMP runtime library. It includes the ability to calculate serial time—how much wall time the application spends in execution outside parallel regions—as well as the time spent by OpenMP threads on barriers waiting for other threads to finish their work on barrier synchronization points. The latter can be a result of load imbalance on an implicit region or loop or explicit user barriers. In our case, we have significant time identified by the MPI Performance Snapshot utility as OpenMP imbalance. The tool can use the Intel VTune Amplifier XE performance analysis tool to explore this in more detail.

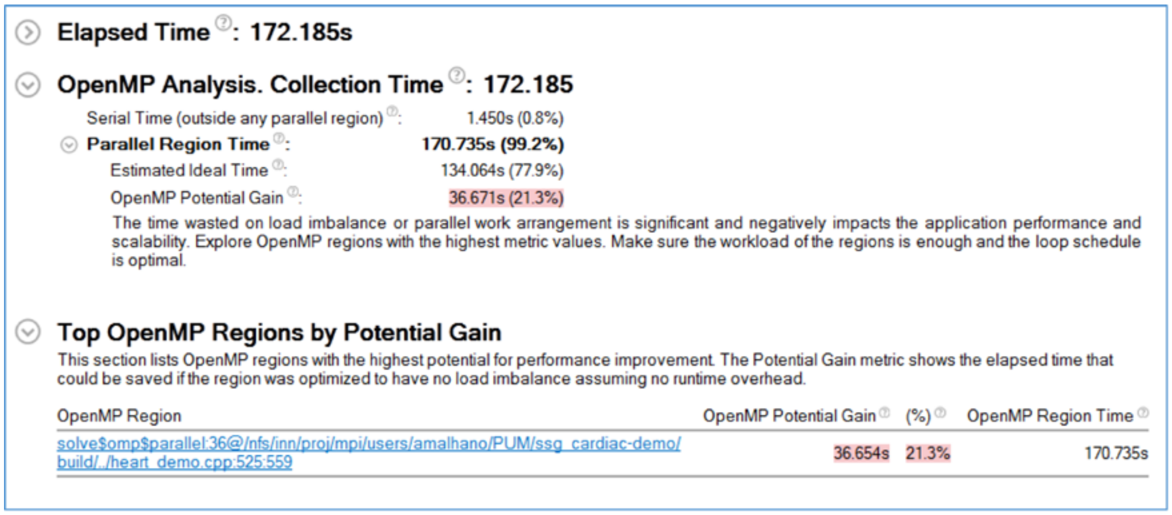
As the MPI Performance Snapshot utility, Intel VTune Amplifier XE uses Intel OpenMP runtime statistics generated in application runtime to find:

- OpenMP region/barrier markup
- Imbalanced barriers
- Parallel loop attributes (e.g., scheduling, chunking, loop iteration count)

The instrumentation in OpenMP runtime is used carefully so as not to spoil the performance picture of the application. Mostly, these are global fork-join and barrier points, avoiding costly per-thread instrumentation. To estimate the cost of scheduling, locks, atomic operations, reduction, and overhead on parallel work arrangement, Intel VTune Amplifier XE uses a CPU sampling and statistical approach that normally produces accurate results on compute-intensive workloads.

Let’s use a basic hotspots analysis (**Figure 5**) on a rank detected by ITAC as the most CPU-bound to see the OpenMP efficiency information.

```
mpirun -gtool “amplxe-cl -collect hotspots -r result:1” -hosts
host1,…,host3 -n 8 -ppn 2 ./heart_demo [parameters]
```



Elapsed Time: 172.185s

OpenMP Analysis. Collection Time: 172.185

Serial Time (outside any parallel region):	1.450s (0.8%)
Parallel Region Time:	**170.735s (99.2%)**
Estimated Ideal Time:	134.064s (77.9%)
OpenMP Potential Gain:	36.671s (21.3%)

The time wasted on load imbalance or parallel work arrangement is significant and negatively impacts the application performance and scalability. Explore OpenMP regions with the highest metric values. Make sure the workload of the regions is enough and the loop schedule is optimal.

Top OpenMP Regions by Potential Gain

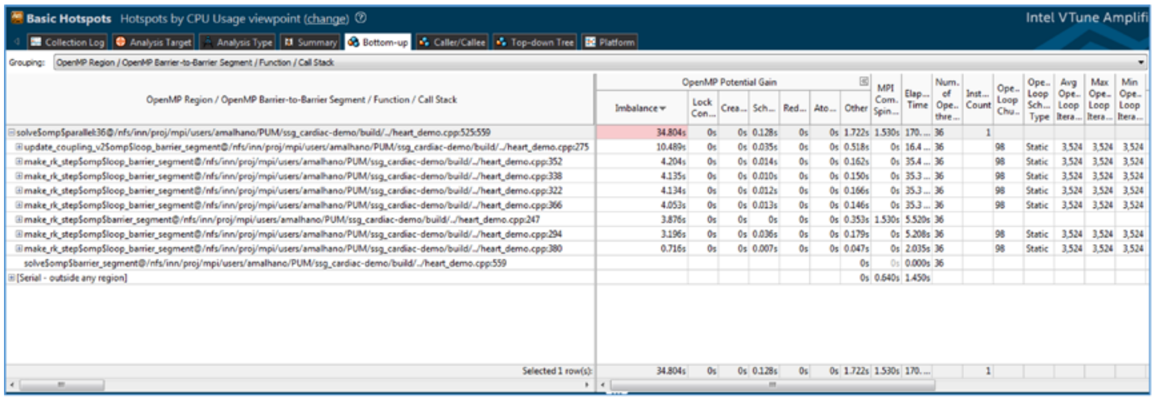
This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	OpenMP Potential Gain	(%)	OpenMP Region Time
solve$omp$parallel:36@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo/build/../heart_demo.cpp:525:559	36.654s	21.3%	170.735s

5 Basic hotspots analysis

From the summary view, we can see that the application has tuning potential for OpenMP parallel code with a maximum gain of 21 percent of the application wall time. So it is worth exploring further. Since the application has only one parallel region construct, the Top OpenMP Regions view contains the only region.

Drilling down to a grid view and expanding the region by barriers of work-sharing constructs, we can see the information in **Figure 6**.

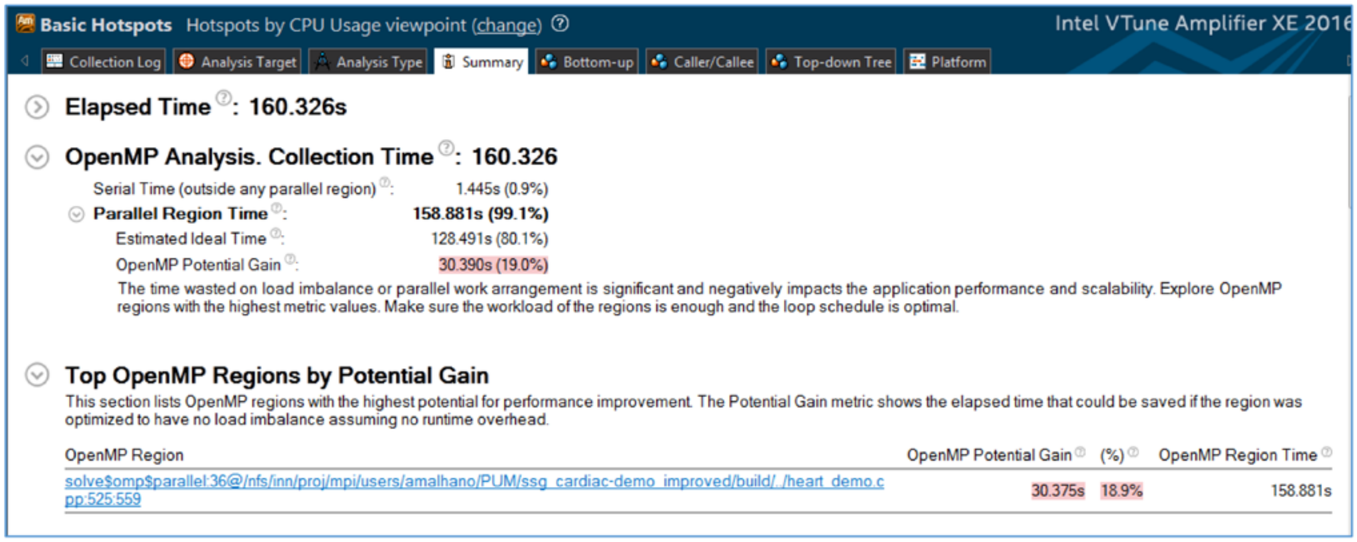


Grouping: OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack

OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack	OpenMP Potential Gain							MPI Com. Spin...	Elap... Time	Num. of Ope.. thre...	Inst... Count	Ope.. Loop Chu..	Ope.. Loop Sch... Type	Avg Ope.. Loop Itera...	Max Ope.. Loop Itera...	Min Ope.. Loop Itera...
	Imbalance	Lock Con...	Crea...	Sch...	Red...	Ato...	Other									
solve$omp$parallel:36@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo/build/../heart_demo.cpp:525:559	34.804s	0s	0s	0.128s	0s	0s	1.722s	1.530s	170. ...	36	1					
update_coupling_v2$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo/build/../heart_demo.cpp:275	10.489s	0s	0s	0.035s	0s	0s	0.518s	0s	16.4 ...	36		98	Static	3,524	3,524	3,524
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo/build/../heart_demo.cpp:352	4.204s	0s	0s	0.014s	0s	0s	0.162s	0s	35.4 ...	36		98	Static	3,524	3,524	3,524
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo/build/../heart_demo.cpp:338	4.135s	0s	0s	0.010s	0s	0s	0.150s	0s	35.3 ...	36		98	Static	3,524	3,524	3,524
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo/build/../heart_demo.cpp:322	4.134s	0s	0s	0.012s	0s	0s	0.166s	0s	35.3 ...	36		98	Static	3,524	3,524	3,524
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo/build/../heart_demo.cpp:366	4.053s	0s	0s	0.013s	0s	0s	0.146s	0s	35.3 ...	36		98	Static	3,524	3,524	3,524
make_rk_step$omp$barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo/build/../heart_demo.cpp:247	3.876s	0s	0s	0s	0s	0s	0.353s	1.530s	5.520s	36						
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo/build/../heart_demo.cpp:294	3.196s	0s	0s	0.036s	0s	0s	0.179s	0s	5.208s	36		98	Static	3,524	3,524	3,524
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo/build/../heart_demo.cpp:380	0.716s	0s	0s	0.007s	0s	0s	0.047s	0s	2.035s	36		98	Static	3,524	3,524	3,524
solve$omp$barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo/build/../heart_demo.cpp:559							0s	0s	0.000s	36						
[Serial - outside any region]							0s	0.640s	1.450s							
Selected 1 row(s):	34.804s	0s	0s	0.128s	0s	0s	1.722s	1.530s	170. ...		1					

6 Expanding the region by barriers of work-sharing constructs

Sorting the grid by the Imbalance metric, we can choose a loop of interest to play with dynamic scheduling to eliminate the imbalance impact. For simplicity, let's apply dynamic scheduling for all parallel loops, choosing nondefault chunking to eliminate scheduling overhead (20 iterations). After recollecting the profile with dynamic scheduling, we can see results in **Figure 7**.



Elapsed Time: 160.326s

OpenMP Analysis. Collection Time: 160.326

Serial Time (outside any parallel region):	1.445s (0.9%)
Parallel Region Time:	**158.881s (99.1%)**
Estimated Ideal Time:	128.491s (80.1%)
OpenMP Potential Gain:	30.390s (19.0%)

The time wasted on load imbalance or parallel work arrangement is significant and negatively impacts the application performance and scalability. Explore OpenMP regions with the highest metric values. Make sure the workload of the regions is enough and the loop schedule is optimal.

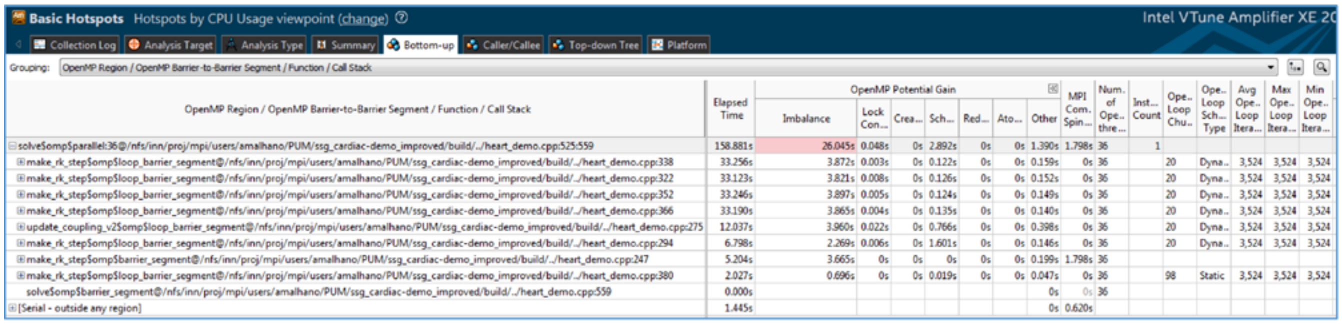
Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	OpenMP Potential Gain	(%)	OpenMP Region Time
solve$omp$parallel:36@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved/build/../heart_demo.cpp:525:559	30.375s	18.9%	158.881s

7 Recollecting the profile with dynamic scheduling

Figure 7 shows that the application runtime is improved by 12 seconds. The imbalance numbers became better. The grid view (**Figure 8**) shows us the parallel loops that benefited most.



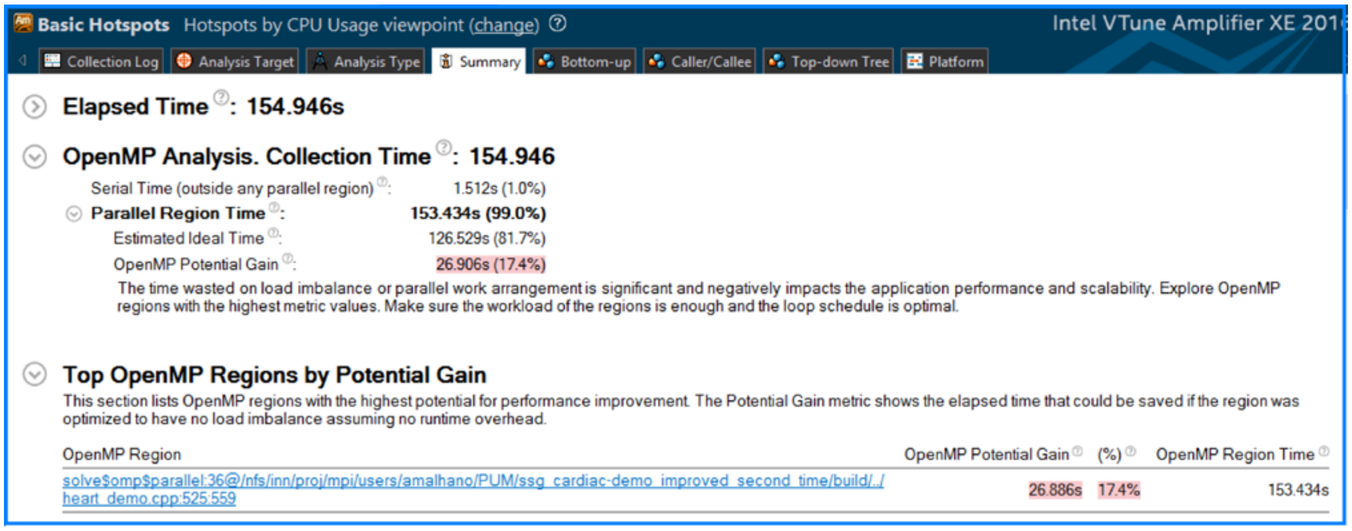
Grouping: OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack

OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack	Elapsed Time	OpenMP Potential Gain							MPI Com. Spin...	Num. of Ope.. thre...	Inst... Count	Ope.. Loop Chu..	Ope.. Loop Sch... Type	Avg Ope.. Loop Itera...	Max Ope.. Loop Itera...	Min Ope.. Loop Itera...
		Imbalance	Lock Con...	Crea...	Sch...	Red...	Ato...	Other								
solve$omp$parallel:36@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved/build/../heart_demo.cpp:525:559	158.881s	26.045s	0.048s	0s	2.892s	0s	0s	1.390s	1.798s	36	1					
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved/build/../heart_demo.cpp:338	33.256s	3.872s	0.003s	0s	0.122s	0s	0s	0.159s	0s	36		20	Dyna..	3,524	3,524	3,524
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved/build/../heart_demo.cpp:322	33.123s	3.821s	0.008s	0s	0.126s	0s	0s	0.152s	0s	36		20	Dyna..	3,524	3,524	3,524
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved/build/../heart_demo.cpp:352	33.246s	3.897s	0.005s	0s	0.124s	0s	0s	0.149s	0s	36		20	Dyna..	3,524	3,524	3,524
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved/build/../heart_demo.cpp:366	33.190s	3.865s	0.004s	0s	0.135s	0s	0s	0.140s	0s	36		20	Dyna..	3,524	3,524	3,524
update_coupling_v2$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved/build/../heart_demo.cpp:275	12.037s	3.960s	0.022s	0s	0.766s	0s	0s	0.398s	0s	36		20	Dyna..	3,524	3,524	3,524
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved/build/../heart_demo.cpp:294	6.798s	2.269s	0.006s	0s	1.601s	0s	0s	0.146s	0s	36		20	Dyna..	3,524	3,524	3,524
make_rk_step$omp$barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved/build/../heart_demo.cpp:247	5.204s	3.665s	0s	0s	0s	0s	0s	0.199s	1.798s	36						
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved/build/../heart_demo.cpp:380	2.027s	0.696s	0s	0s	0.019s	0s	0s	0.047s	0s	36		98	Static	3,524	3,524	3,524
solve$omp$barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved/build/../heart_demo.cpp:559	0.000s							0s	0s	36						
[Serial - outside any region]	1.445s							0s	0.620s							

8 Grid view

The parallel loop at line 275 in heart_demo.cpp shows 4 seconds of elapsed time improvement. Loops in heart_demo.cpp lines 322, 338, 352, and 366 show 2 seconds of improvement per loop, even though the imbalance did not completely vanish. The loop in heart_demo.cpp line 294 became 1.5 seconds worse because scheduling overhead annihilated the imbalance improvement. This means it makes sense to keep it static. The results in this case will look like **Figure 9**.

Playing with scheduling, we need to be careful with cache usage (**Figure 10**). Cache reuse can become worse with the dynamic nature of work distribution by worker threads. That, in turn, can make parallel loop time even longer.



Elapsed Time: 154.946s

OpenMP Analysis. Collection Time: 154.946

Serial Time (outside any parallel region):	1.512s (1.0%)
Parallel Region Time:	**153.434s (99.0%)**
Estimated Ideal Time:	126.529s (81.7%)
OpenMP Potential Gain:	26.906s (17.4%)

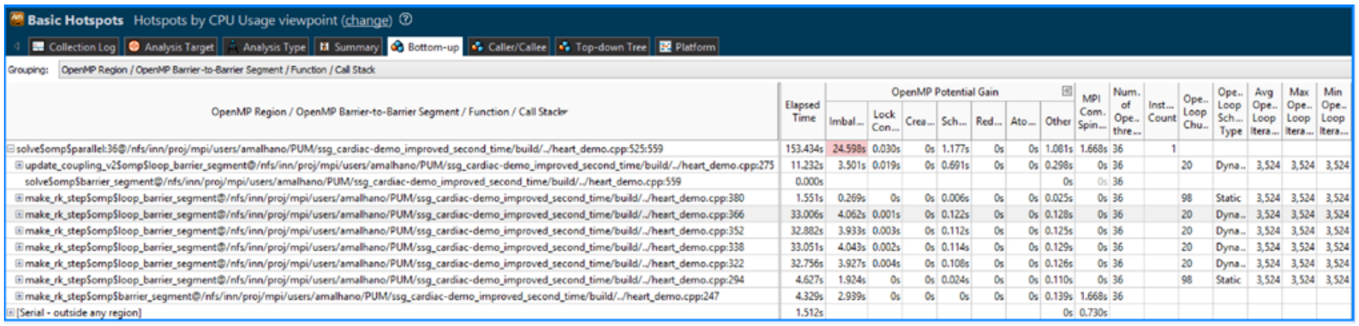
The time wasted on load imbalance or parallel work arrangement is significant and negatively impacts the application performance and scalability. Explore OpenMP regions with the highest metric values. Make sure the workload of the regions is enough and the loop schedule is optimal.

Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	OpenMP Potential Gain	(%)	OpenMP Region Time
solve$omp$parallel:36@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:525:559	26.886s	17.4%	153.434s

9 Hotspots by CPU usage



Grouping: OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack

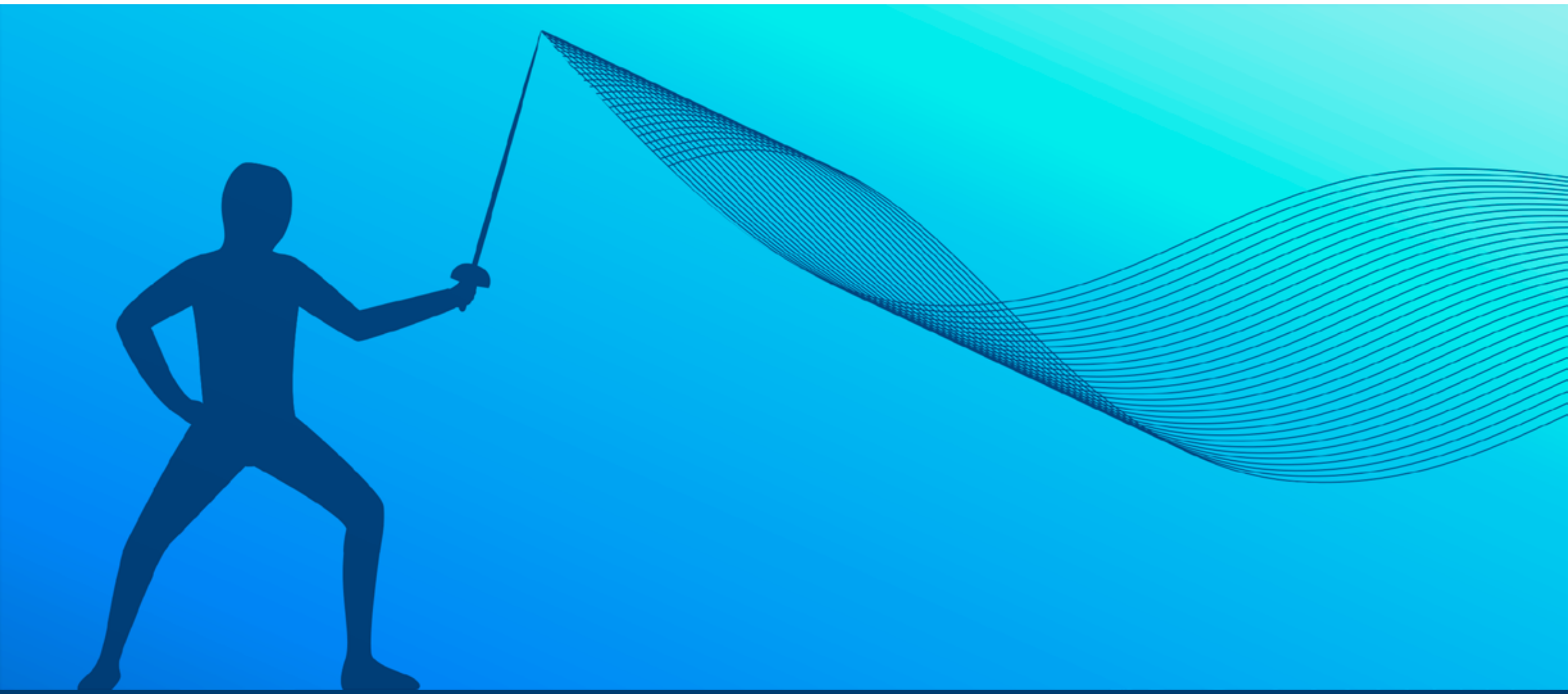
OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack	Elapsed Time	OpenMP Potential Gain							MPI Com. Spin...	Num. of Ope.. thre...	Inst... Count	Ope.. Loop Chu..	Ope.. Loop Sch... Type	Avg Ope.. Loop Itera...	Max Ope.. Loop Itera...	Min Ope.. Loop Itera...
		Imbal...	Lock Con...	Crea...	Sch...	Red...	Ato...	Other								
solve$omp$parallel:36@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:525:559	153.434s	24.598s	0.030s	0s	1.177s	0s	0s	1.081s	1.668s	36	1					
update_coupling_v2$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:275	11.232s	3.501s	0.019s	0s	0.691s	0s	0s	0.298s	0s	36		20	Dyna..	3,524	3,524	3,524
solve$omp$barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:559	0.000s							0s	0s	36						
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:380	1.551s	0.269s	0s	0s	0.006s	0s	0s	0.025s	0s	36		98	Static	3,524	3,524	3,524
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:366	33.006s	4.062s	0.001s	0s	0.122s	0s	0s	0.128s	0s	36		20	Dyna..	3,524	3,524	3,524
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:352	32.882s	3.933s	0.003s	0s	0.112s	0s	0s	0.125s	0s	36		20	Dyna..	3,524	3,524	3,524
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:338	33.051s	4.043s	0.002s	0s	0.114s	0s	0s	0.129s	0s	36		20	Dyna..	3,524	3,524	3,524
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:322	32.756s	3.927s	0.004s	0s	0.108s	0s	0s	0.126s	0s	36		20	Dyna..	3,524	3,524	3,524
make_rk_step$omp$loop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:294	4.627s	1.924s	0s	0s	0.024s	0s	0s	0.110s	0s	36		98	Static	3,524	3,524	3,524
make_rk_step$omp$barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:247	4.329s	2.939s	0s	0s	0s	0s	0s	0.139s	1.668s	36						
[Serial - outside any region]	1.512s							0s	0.730s							

10 Cache usage

Conclusions

We have described a step-by-step workflow for hybrid application analysis and tuning. The Message Profile chart from ITAC helped us to understand MPI utilization inefficiencies and eliminate them by changing the communication pattern. Intel VTune Amplifier XE gave insight into OpenMP parallelization efficiency and helped to better balance thread level loads.

We have not covered all possible strategies for hybrid application analysis and tuning, since the tools have rich functionality, which is out of our scope. Tool utilization scenarios can vary from application to application. You should refer to the appropriate user manuals to learn about tool functionality and choose the best approach for your project.



VECTORIZE YOUR CODE USING INTEL® ADVISOR XE 2016

Solve Common Problems When Increasing Performance

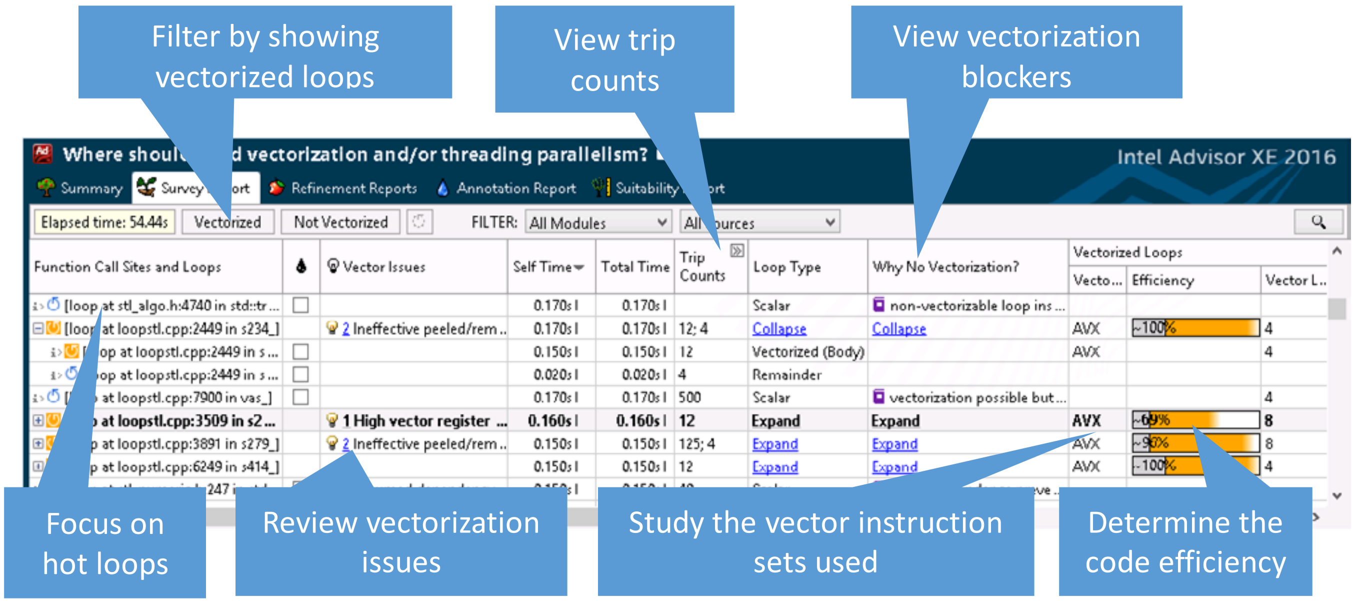
Kevin O'Leary, *Software Technical Consulting Engineer,* Kirill Rogozhin, *Software Development Manager,* and Vadim Kartoshkin, *Technical Writer,* Intel Corporation

Many factors can make programs difficult for automatic vectorization. In this article, we will examine some of the factors that can make vectorizing code problematic without providing the compiler with some additional hints. Vectorizing loops is critical for increasing your applications' performance, and Intel® Advisor XE is the tool that can guide you through the process of vectorization.

Intel Advisor XE 2016 is a dynamic analysis tool that now contains a Vectorization Advisor feature (**Figure 1**). Using Vectorization Advisor, you can survey all the loops in your application and see:

- Which loops were vectorized and which loops were not
- What prevented **vectorization** for the non-vectorized loops
- The speedup and vectorization efficiency for the vectorized loops
- Any issues that decreased efficiency of the vectorized loops
- The vectorized and non-vectorized loops that were limited by the memory layout

In this article, we will provide an overview of the Vectorization Advisor and show some new features that can assist you with vectorization on the next generation of Intel® Xeon Phi™ (formerly Knights Landing). We will also provide some examples of common problems and show how you can utilize the Vectorization Advisor to vectorize them.

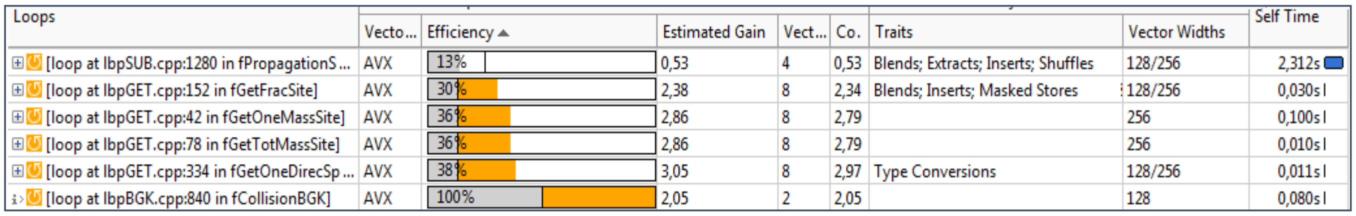

Filter by showing vectorized loops
View trip counts
View vectorization blockers
Where should d vectorization and/or threading parallelism?
Intel Advisor XE 2016
Summary
Survey ort
Refinement Reports
Annotation Report
Suitability ort
Elapsed time: 54.44s
Vectorized
Not Vectorized
FILTER: All Modules
All ources
Focus on hot loops
Review vectorization issues
Study the vector instruction sets used
Determine the code efficiency

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Trip Counts	Loop Type	Why No Vectorization?	Vectorized Loops		
								Vecto...	Efficiency	Vector L...
[loop at stl_algo.h:4740 in std::tr...	☐		0.170s	0.170s		Scalar	non-vectorizable loop ins...			
[loo at loopstl.cpp:2449 in s234_]		2 Ineffective peeled/rem...	0.170s	0.170s	12; 4	Collapse	Collapse	AVX	~100%	4
op at loopstl.cpp:2449 in s...	☐		0.150s	0.150s	12	Vectorized (Body)		AVX		4
op at loopstl.cpp:2449 in s...	☐		0.020s	0.020s	4	Remainder				
at loopstl.cpp:7900 in vas_]	☐		0.170s	0.170s	500	Scalar	vectorization possible but...			4
at loopstl.cpp:3509 in s2...		**1 High vector register ...**	**0.160s**	**0.160s**	**12**	**Expand**	**Expand**	**AVX**	~69%	**8**
p at loopstl.cpp:3891 in s279_]		2 Ineffective peeled/rem...	0.150s	0.150s	125; 4	Expand	Expand	AVX	~96%	8
p at loopstl.cpp:6249 in s414_]			0.150s	0.150s	12	Expand	Expand	AVX	~100%	4

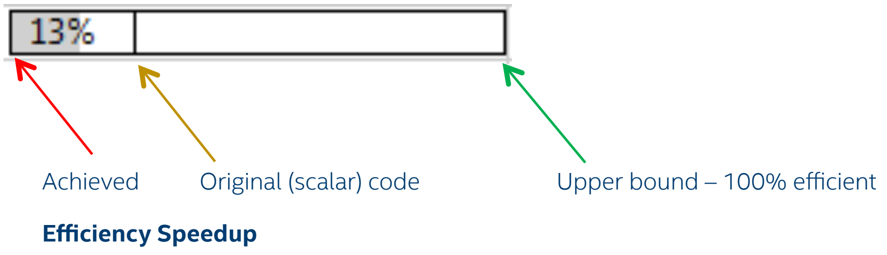
1 Vectorization Advisor: All the data you need, at your fingertips

Five Steps to Improve Your Vector Efficiency

1. **Survey.** The first step is surveying the application. During this survey step, you can see the loops where your application is spending time. The `hot` loops are where you will get the greatest benefit from optimizing. **Figure 2** is a Survey Report of an application. In Intel Advisor XE, you can filter by the type of loop: vectorized or non-vectorized. Non-vectorized loops show what is preventing vectorization.
2. **View recommendations.** Get specific advice for improving your vectorization efficiency. Also, see issues that prevent vectorization.
3. **Trip counts.** We collect loop iteration trip counts as a separate collection step. It is very important to know not just that a loop is hot, but also the trip count. If your trip count is low, there may not be enough iterations to vectorize efficiently. You can also see if your trip count is divisible by your vector length and will not require a remainder loop.
4. **Dependency analysis.** To generate correct code, the compiler must take a conservative view with respect to the semantics of the language it is compiling for. If it is possible for a dependency to exist based on the rules of the language, then the compiler must assume the dependency exists. By using a dynamic tool such as Intel Advisor XE, you can check if the assumed dependency is real.
5. **Memory access pattern (MAP) analysis.** You may greatly increase the vectorization efficiency of your application if you know how your data structures are laid out in memory and accessed in your loop. It is important that memory references are aligned properly, accessed in unit stride manner, etc. There are several techniques related to memory access that can assist with vectorization, such as converting your data structures from `Arrays of Structures` to `Structures of Arrays`. Using a MAP analysis, you can uncover the patterns that are inherently vector-inefficient.



Loops	Vecto...	Efficiency	Estimated Gain	Vect...	Co.	Traits	Vector Widths	Self Time
[loop at lbpSUB.cpp:1280 in fPropagationS...	AVX	13%	0,53	4	0,53	Blends; Extracts; Inserts; Shuffles	128/256	2,312s
[loop at lbpGET.cpp:152 in fGetFracSite]	AVX	30%	2,38	8	2,34	Blends; Inserts; Masked Stores	128/256	0,030s
[loop at lbpGET.cpp:42 in fGetOneMassSite]	AVX	36%	2,86	8	2,79		256	0,100s
[loop at lbpGET.cpp:78 in fGetTotMassSite]	AVX	36%	2,86	8	2,79		256	0,010s
[loop at lbpGET.cpp:334 in fGetOneDirecSp...	AVX	38%	3,05	8	2,97	Type Conversions	128/256	0,011s
[loop at lbpBGK.cpp:840 in fCollisionBGK]	AVX	100%	2,05	2	2,05		128	0,080s



2 Vector efficiency: Your performance thermometer

Vectorization Advisor can estimate your code's vectorization efficiency. By examining the efficiency metric, you can see which loops have issues that need to be addressed. If a loop is vectorized but at low efficiency, you can first check if Vectorization Advisor gives any recommendations for improving the code efficiency. Generally, data structure layout can affect vector efficiency greatly. In this case, you can run a MAP analysis to determine if you are referencing memory in a vector-friendly manner.

No Access to Intel® AVX-512 Hardware Yet?

Using Intel Advisor XE, you can get your code ready for the next generation Intel Xeon Phi coprocessors even if you don't have access to the hardware yet. Enable this functionality by generating code for multiple vector instruction sets (including AVX-512) using the Intel® Compiler `—ax` option, and then analyze the resulting binary with Vectorization Advisor.

Compile the Code with –ax Flags

First, order the compiler to generate binaries with alternative code paths (besides the default ones). You can do that by specifying the –ax options. For example, generate code for both SSE2 and AVX2 instruction set analyses (ISAs) in the same binary using the following compilation flag:

```
—axCORE-AVX2
```

With this option, the compiler generates assembly for the SSE2 instruction set (this is the default if you don't specify a different default instruction set with the `—x` or `—m` flag) and also generates alternative code paths for the AVX2 instruction set that will be used if the system on which the binary executes has the hardware with the corresponding ISA.

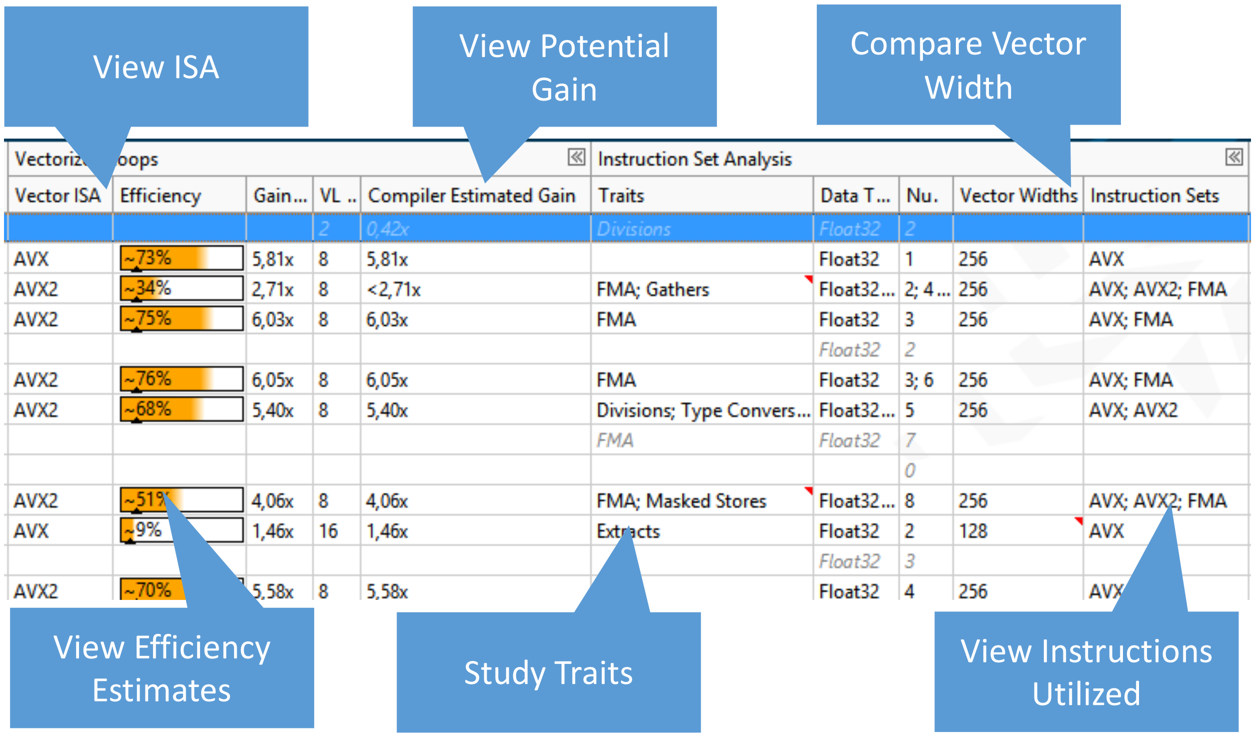
If you want to generate the code that targets some high-end hardware (including Intel Xeon Phi machines, for example), you can order the compiler to generate code that will use the highest ISA available.

Consider this example, where we target minimum ISA as SSE4.1 and expect the code to use corresponding instructions on the machines with AVX2 and AVX-512 instruction set architectures:

```
-axCORE-AVX512, —axCORE-AVX2, —xsse4.1
```

- `–xsse4.1` changes the default code path to SSE4.1 if the hardware does not support the alternative ones.
- `–axCORE-AVX2` sets the first alternative, which will be used on a machine with AVX2-enabled hardware.
- `–axCORE-AVX512` sets the second alternative, enabling a machine with AVX-512 ISA to use corresponding instructions.

If you are interested in viewing the difference between the codes generated for different vector ISAs, refer to the Intel Advisor Survey report to take a deeper look at the non-executed loops. The report provides information about various traits used for different ISAs and enables you to compare estimated gains (**Figure 3**).



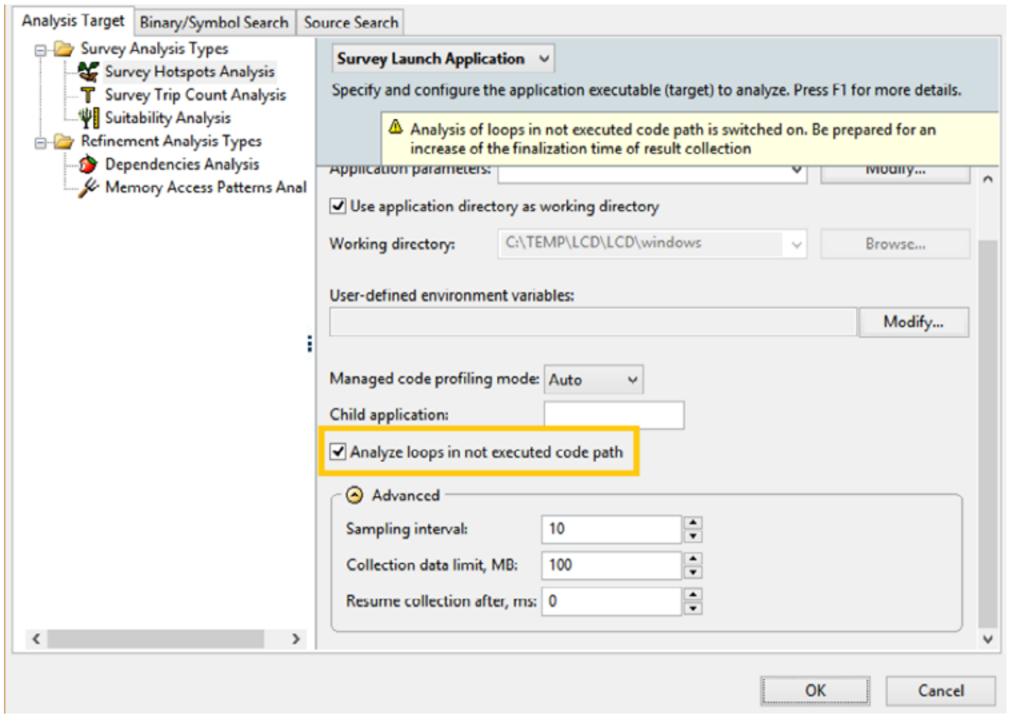
Vectoriz oops					Instruction Set Analysis				
Vector ISA	Efficiency	Gain...	VL ..	Compiler Estimated Gain	Traits	Data T...	Nu.	Vector Widths	Instruction Sets
			2	0,42x	Divisions	Float32	2		
AVX	~73%	5,81x	8	5,81x		Float32	1	256	AVX
AVX2	~34%	2,71x	8	<2,71x	FMA; Gathers	Float32...	2; 4...	256	AVX; AVX2; FMA
AVX2	~75%	6,03x	8	6,03x	FMA	Float32	3	256	AVX; FMA
						Float32	2		
AVX2	~76%	6,05x	8	6,05x	FMA	Float32	3; 6	256	AVX; FMA
AVX2	~68%	5,40x	8	5,40x	Divisions; Type Convers...	Float32...	5	256	AVX; AVX2
					FMA	Float32	7		
							0		
AVX2	~51%	4,06x	8	4,06x	FMA; Masked Stores	Float32...	8	256	AVX; AVX2; FMA
AVX	~9%	1,46x	16	1,46x	Extracts	Float32	2	128	AVX
						Float32	3		
AVX2	~70%	5,58x	8	5,58x		Float32	4	256	AVX

3 Intel® Advisor XE Survey report

Enable Non-Executed Code Paths Analysis

Once you have your binary compiled to use different and most suitable instruction sets depending on the hardware capabilities, you need to enable Intel Advisor to analyze all versions of the vector loops residing in your binary, as follows.

1. Run Intel Advisor XE.
2. In **Project Properties (Ctrl+P)**:
   a. Specify path to the binary.
   b. Check "Analyze loops in not executed code path."



Specify and configure the application executable (target) to analyze. Press F1 for more details.

Analysis of loops in not executed code path is switched on. Be prepared for an increase of the finalization time of result collection

Application parameters:

☑ Use application directory as working directory

Working directory: C:\TEMP\LCD\LCD\windows

User-defined environment variables:

Managed code profiling mode: Auto

Child application:

☑ Analyze loops in not executed code path

Advanced

Sampling interval: 10

Collection data limit, MB: 100

Resume collection after, ms: 0

3. Click **OK**.

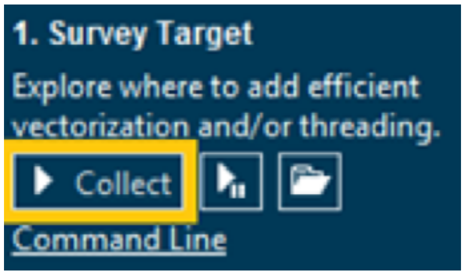
If you work in command line (e.g., an MPI application on a cluster node), use CLI syntax:

```
mpirun -n 2 -gtool "advixe-cl -collect survey -support-multi-isa-binaries
-no-auto-finalize --project-dir=/tmp/my_proj" /tmp/bin/my_app
```

Viewing Loops Residing in Non-Executed Code Paths

Survey the Binary

Click the Collect button at the workflow tab.



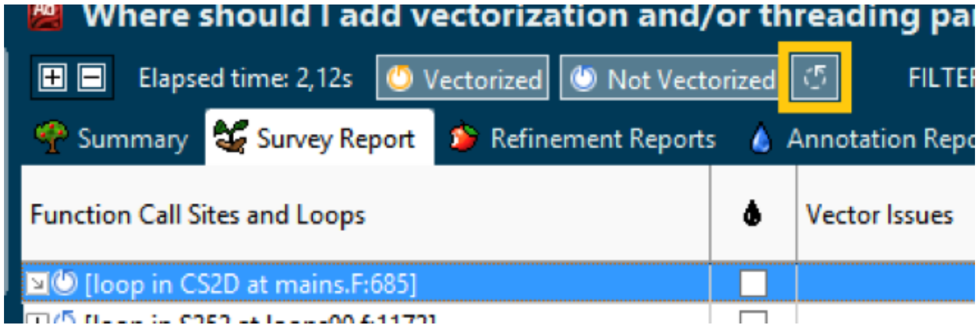
1. Survey Target

Explore where to add efficient vectorization and/or threading.

Warning: *Finalization of Intel® Advisor XE results for all loops (including the non-executed ones for ISAs different from those available with the current hardware) might take more time than usual.*

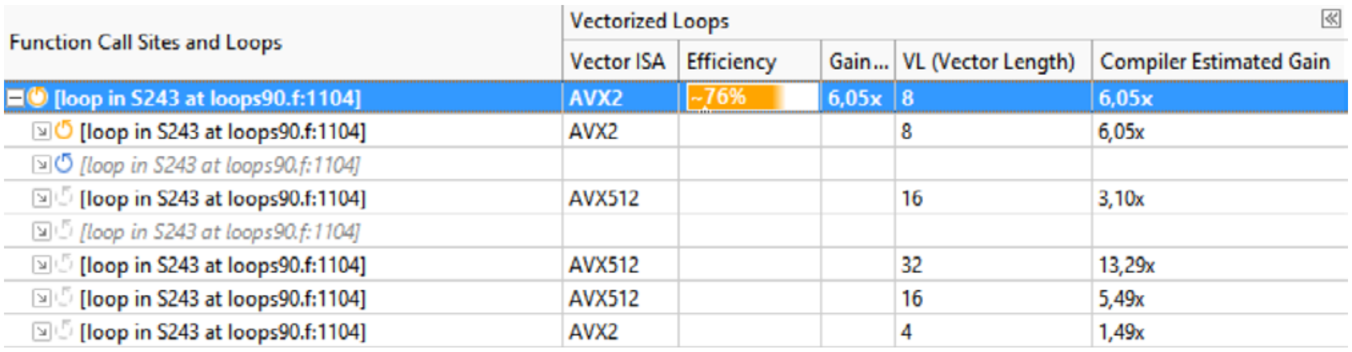
Turn on Viewing the Non-Executed Loops in the Survey Report

Once the Survey analysis results are collected, refer to the Survey report. You need to enable viewing of non-executed loops in the Survey grid, which can be done by clicking the corresponding button:



Where should I add vectorization and/or threading pa

Once you click the button, Intel Advisor will refresh the grid and add non-executed loops under “parent” loops. To see them, expand a vectorized loop . You will see the non-executed loops among those that did execute (**Figure 4**).



Function Call Sites and Loops	Vectorized Loops				
	Vector ISA	Efficiency	Gain...	VL (Vector Length)	Compiler Estimated Gain
[loop in S243 at loops90.f:1104]	AVX2	~76%	6,05x	8	6,05x
[loop in S243 at loops90.f:1104]	AVX2			8	6,05x
[loop in S243 at loops90.f:1104]					
[loop in S243 at loops90.f:1104]	AVX512			16	3,10x
[loop in S243 at loops90.f:1104]					
[loop in S243 at loops90.f:1104]	AVX512			32	13,29x
[loop in S243 at loops90.f:1104]	AVX512			16	5,49x
[loop in S243 at loops90.f:1104]	AVX2			4	1,49x

4 Non-executed loops among executed loops

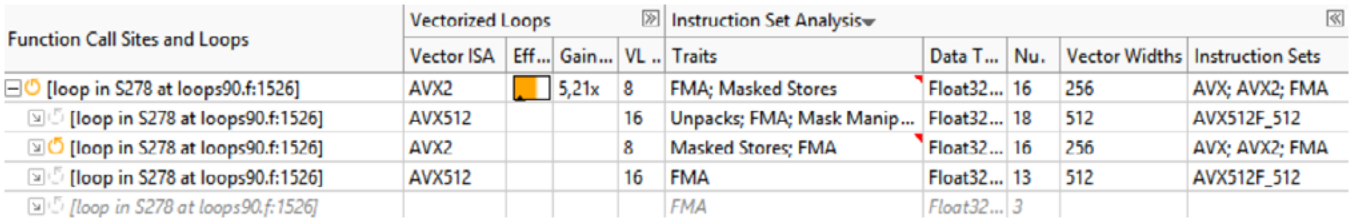
Survey Columns to Study

Now look at the Vectorized Loops column (extend using with the button):

Intel Advisor also shows compiler diagnostics for the non-executed loops , which include:

- The **Vector ISA** column, which simply notifies you which ISA the particular code path targets.
- The **VL (Vector Length)** column, which reports the vector length. So, for the case of running the sample code used while writing this article on an Intel Xeon Phi-enabled machine, vector loop needs 16 to 32 (depending on a loop) operations fewer than its scalar version, which leading to potentially higher performance.
- The **Compiler Estimated Gain** column, which shows the performance predictions made by the compiler. The estimated gain is compared to the scalar version of the same loop running on the hardware supporting the target ISA. In other words, the compiler estimates a 6.05X gain for the vectorized version of the loop against its scalar version, with both running on the same AVX2-enabled machine. Another example is the 13.29X gain predicted by the compiler in the case of running the same loop on an AVX-512 machine.

You definitely want to view the **Instruction Set Analysis** column to compare AVX2 and AVX-512 loops:

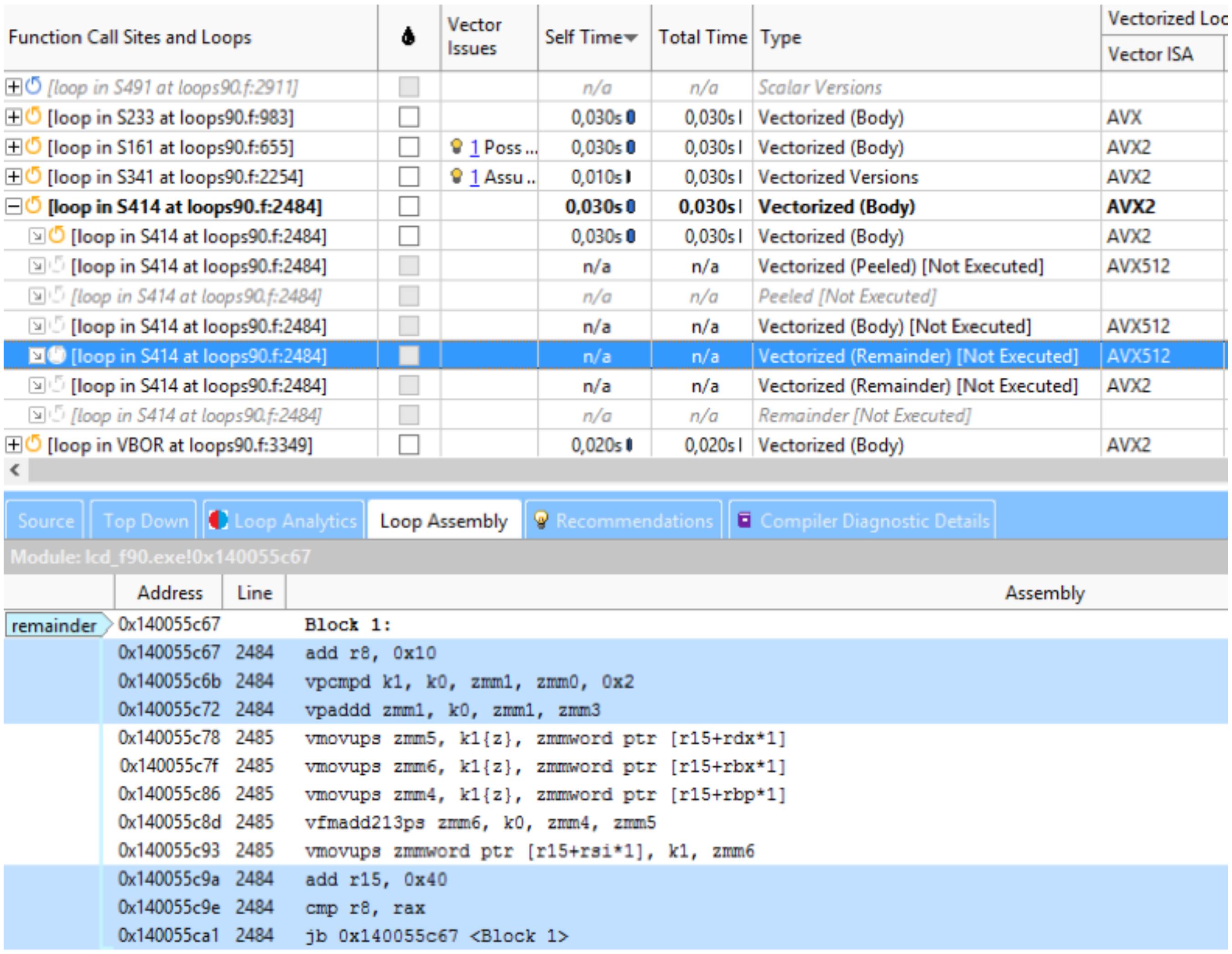


Function Call Sites and Loops	Vectorized Loops				Instruction Set Analysis				
	Vector ISA	Eff...	Gain...	VL ..	Traits	Data T...	Nu.	Vector Widths	Instruction Sets
[loop in S278 at loops90.f:1526]	AVX2	🟧	5,21x	8	FMA; Masked Stores	Float32...	16	256	AVX; AVX2; FMA
[loop in S278 at loops90.f:1526]	AVX512			16	Unpacks; FMA; Mask Manip...	Float32...	18	512	AVX512F_512
[loop in S278 at loops90.f:1526]	AVX2			8	Masked Stores; FMA	Float32...	16	256	AVX; AVX2; FMA
[loop in S278 at loops90.f:1526]	AVX512			16	FMA	Float32...	13	512	AVX512F_512
[loop in S278 at loops90.f:1526]					*FMA*	*Float32...*	*3*		

- The **Traits** column reports instructions, whose presence may impact code performance significantly (in both negative and positive ways). For AVX-512-enabled loops, Intel Advisor may indicate such traits as Gather, Compress, SQRT reciprocal, Mask Manipulations, among others, enabled with the AVX-512 ISA only.
- The **Vector Width** column, which shows the vector register width in bits, which is hardware-specific.
- The **Instruction Sets** column, which reports instruction sets used for individual instructions.

View Assembly Representation

To view the assembly of a code path with a different ISA (e.g., AVX-512), select the loop in the Survey grid, then click on the Loop Assembly tab:



Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Vectorized Loc Vector ISA
[loop in S491 at loops90.f:2911]			n/a	n/a	Scalar Versions	
[loop in S233 at loops90.f:983]			0,030s	0,030s	Vectorized (Body)	AVX
[loop in S161 at loops90.f:655]		1 Poss ...	0,030s	0,030s	Vectorized (Body)	AVX2
[loop in S341 at loops90.f:2254]		1 Assu ..	0,010s	0,030s	Vectorized Versions	AVX2
[loop in S414 at loops90.f:2484]			**0,030s**	**0,030s**	**Vectorized (Body)**	**AVX2**
[loop in S414 at loops90.f:2484]			0,030s	0,030s	Vectorized (Body)	AVX2
[loop in S414 at loops90.f:2484]			n/a	n/a	Vectorized (Peeled) [Not Executed]	AVX512
[loop in S414 at loops90.f:2484]			n/a	n/a	Peeled [Not Executed]	
[loop in S414 at loops90.f:2484]			n/a	n/a	Vectorized (Body) [Not Executed]	AVX512
[loop in S414 at loops90.f:2484]			n/a	n/a	Vectorized (Remainder) [Not Executed]	AVX512
[loop in S414 at loops90.f:2484]			n/a	n/a	Vectorized (Remainder) [Not Executed]	AVX2
[loop in S414 at loops90.f:2484]			n/a	n/a	Remainder [Not Executed]	
[loop in VBOR at loops90.f:3349]			0,020s	0,020s	Vectorized (Body)	AVX2

Module: lcd_f90.exe!0x140055c67

```
             Address      Line   Assembly
remainder    0x140055c67         Block 1:
             0x140055c67  2484   add r8, 0x10
             0x140055c6b  2484   vpcmpd k1, k0, zmm1, zmm0, 0x2
             0x140055c72  2484   vpaddd zmm1, k0, zmm1, zmm3
             0x140055c78  2485   vmovups zmm5, k1{z}, zmmword ptr [r15+rdx*1]
             0x140055c7f  2485   vmovups zmm6, k1{z}, zmmword ptr [r15+rbx*1]
             0x140055c86  2485   vmovups zmm4, k1{z}, zmmword ptr [r15+rbp*1]
             0x140055c8d  2485   vfmadd213ps zmm6, k0, zmm4, zmm5
             0x140055c93  2485   vmovups zmmword ptr [r15+rsi*1], k1, zmm6
             0x140055c9a  2484   add r15, 0x40
             0x140055c9e  2484   cmp r8, rax
             0x140055ca1  2484   jb 0x140055c67 <Block 1>
```

Using this feature, you can view the difference between AVX2 and AVX-512 code paths.

More Background on Loop Vectorization

A typical vectorized loop consists of:

- **Main vector body:** Fastest among the three.
- **Optional peel part:** Used for unaligned references in your loop. Uses Scalar or slower vector.
- **Remainder part:** Due to the number of iterations (trip count) not being divisible by vector length. Uses a scalar or slower vector.

A larger vector register means more iterations in peel/remainder, so:

- Make sure you align your data (and you tell the compiler it is aligned)
- Make the number of iterations divisible by the vector length

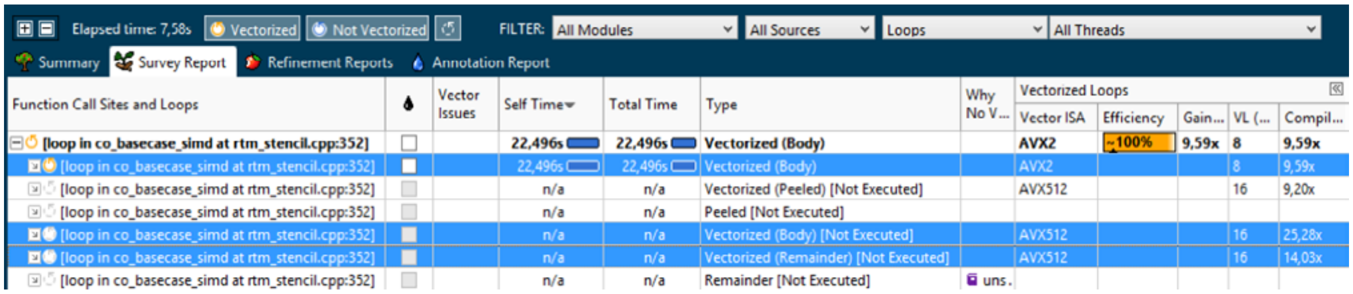
AVX-512 Diagnostic Examples

RTM Stencil Project

Stencil computation is the basis for the Reverse Time Migration algorithm in seismic computing. The underlying mathematical problem is to solve the wave equation using a finite difference method. This sample computes a 3D 25-point stencil. Generating AVX-512 code for an **RTM Stencil** sample project shows that the compiler estimates an AVX-512 speedup of 25.28x and AVX2 speedup of 9.59x (**Figure 5**).

Since the vector length difference is 2x, we would expect the AVX-512 code to be 2x faster, but it is estimated to be 2.63x faster.

Using Intel Advisor XE you can also see a possible answer for the discrepancy. The AVX2 code has scalar remainder, but in AVX-512 the remainder is vectorized.



Function Call Sites and Loops	💧	Vector Issues	Self Time	Total Time	Type	Why No V...	Vectorized Loops				
							Vector ISA	Efficiency	Gain...	VL (...	Compil...
[loop in co_basecase_simd at rtm_stencil.cpp:352]	☐		22,496s	22,496s	Vectorized (Body)		AVX2	~100%	9,59x	8	9,59x
[loop in co_basecase_simd at rtm_stencil.cpp:352]	☐		22,496s	22,496s	Vectorized (Body)		AVX2			8	9,59x
[loop in co_basecase_simd at rtm_stencil.cpp:352]	☐		n/a	n/a	Vectorized (Peeled) [Not Executed]		AVX512			16	9,20x
[loop in co_basecase_simd at rtm_stencil.cpp:352]	☐		n/a	n/a	Peeled [Not Executed]						
[loop in co_basecase_simd at rtm_stencil.cpp:352]	☐		n/a	n/a	Vectorized (Body) [Not Executed]		AVX512			16	25,28x
[loop in co_basecase_simd at rtm_stencil.cpp:352]	☐		n/a	n/a	Vectorized (Remainder) [Not Executed]		AVX512			16	14,03x
[loop in co_basecase_simd at rtm_stencil.cpp:352]	☐		n/a	n/a	Remainder [Not Executed]	uns.					

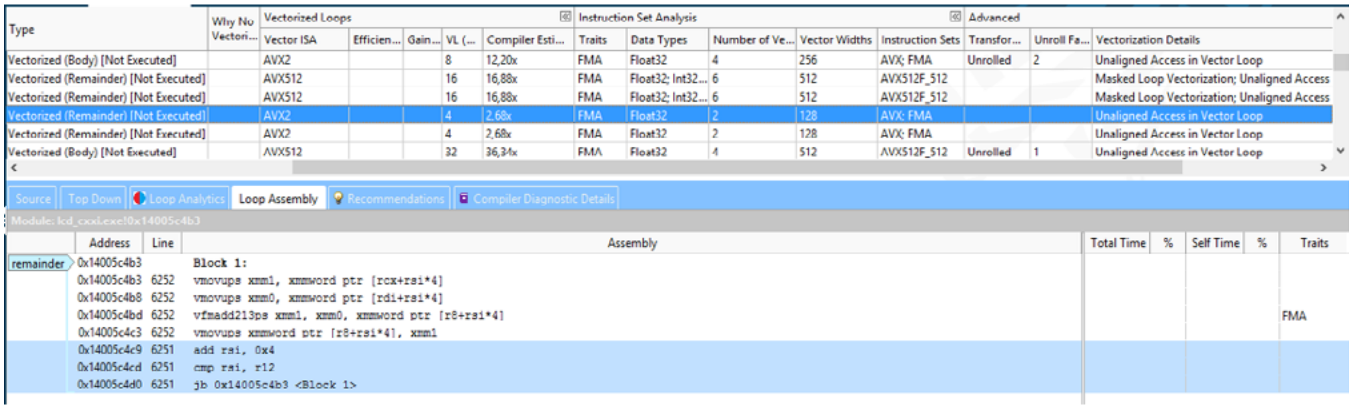
5 Speedup for RTM Stencil sample project

LCD Vectorization Benchmark

In this example project, the compiler estimated gains for the two versions: 12.20x (AVX2) vs. 36.34x (AVX-512). So the AVX-512 code is 2.97x faster but the vector length is only twice as big. In the RTM Stencil example, the compiler was not able to vectorize the AVX2 remainder loop.

In this case, both AVX2 and AVX-512 code paths have vectorized remainders. This is likely explained by using masked operations in AVX-512.

For AVX2:



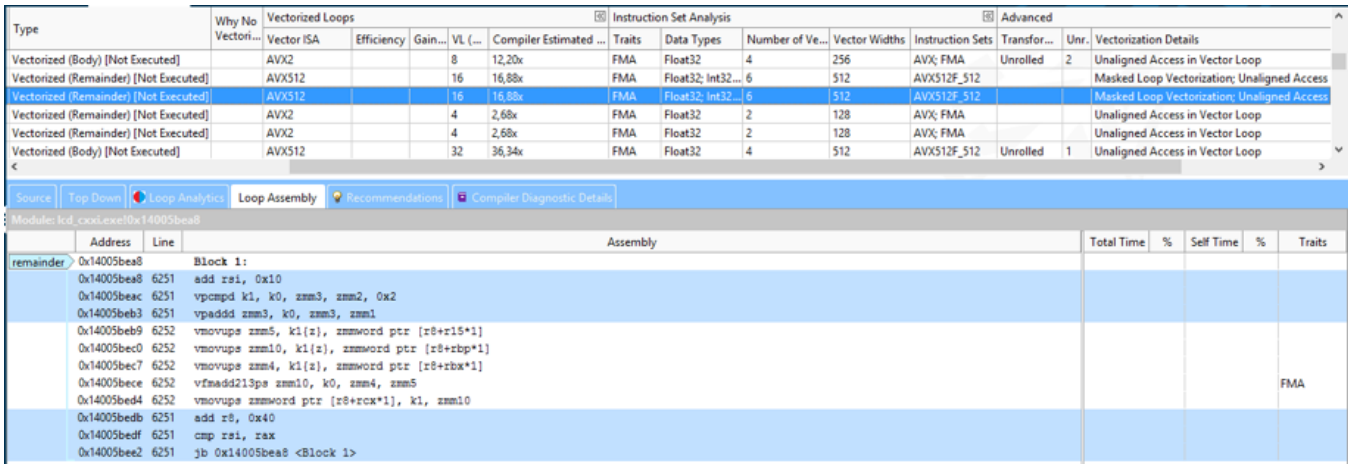
Type	Why No Vectori...	Vectorized Loops					Instruction Set Analysis					Advanced		
		Vector ISA	Efficien...	Gain...	VL (...	Compiler Esti...	Traits	Data Types	Number of Ve...	Vector Widths	Instruction Sets	Transfor...	Unroll Fa...	Vectorization Details
Vectorized (Body) [Not Executed]		AVX2			8	12,20x	FMA	Float32	4	256	AVX; FMA	Unrolled	2	Unaligned Access in Vector Loop
Vectorized (Remainder) [Not Executed]		AVX512			16	16,88x	FMA	Float32; Int32...	6	512	AVX512F_512			Masked Loop Vectorization; Unaligned Access
Vectorized (Remainder) [Not Executed]		AVX512			16	16,88x	FMA	Float32; Int32...	6	512	AVX512F_512			Masked Loop Vectorization; Unaligned Access
Vectorized (Remainder) [Not Executed]		AVX2			4	2,68x	FMA	Float32	2	128	AVX; FMA			Unaligned Access in Vector Loop
Vectorized (Remainder) [Not Executed]		AVX2			4	2,68x	FMA	Float32	2	128	AVX; FMA			Unaligned Access in Vector Loop
Vectorized (Body) [Not Executed]		AVX512			32	36,34x	FMA	Float32	4	512	AVX512F_512	Unrolled	1	Unaligned Access in Vector Loop

Source | Top Down | Loop Analytics | Loop Assembly | Recommendations | Compiler Diagnostic Details

Module: lcd_cxxi.exe!0x14005c4b3

```
            Address     Line   Assembly                                          Total Time  %  Self Time  %  Traits
remainder   0x14005c4b3        Block 1:
            0x14005c4b3 6252   vmovups xmm1, xmmword ptr [rcx+rsi*4]
            0x14005c4b8 6252   vmovups xmm0, xmmword ptr [rdi+rsi*4]
            0x14005c4bd 6252   vfmadd213ps xmm1, xmm0, xmmword ptr [r8+rsi*4]                                 FMA
            0x14005c4c3 6252   vmovups xmmword ptr [r8+rsi*4], xmm1
            0x14005c4c9 6251   add rsi, 0x4
            0x14005c4cd 6251   cmp rsi, r12
            0x14005c4d0 6251   jb 0x14005c4b3 <Block 1>
```

For AVX-512:



Type	Why No Vectori...	Vectorized Loops					Instruction Set Analysis					Advanced		
		Vector ISA	Efficiency	Gain...	VL (...	Compiler Estimated ...	Traits	Data Types	Number of Ve...	Vector Widths	Instruction Sets	Transfor...	Unr.	Vectorization Details
Vectorized (Body) [Not Executed]		AVX2			8	12,20x	FMA	Float32	4	256	AVX; FMA	Unrolled	2	Unaligned Access in Vector Loop
Vectorized (Remainder) [Not Executed]		AVX512			16	16,88x	FMA	Float32; Int32...	6	512	AVX512F_512			Masked Loop Vectorization; Unaligned Access
Vectorized (Remainder) [Not Executed]		AVX512			16	16,88x	FMA	Float32; Int32...	6	512	AVX512F_512			Masked Loop Vectorization; Unaligned Access
Vectorized (Remainder) [Not Executed]		AVX2			4	2,68x	FMA	Float32	2	128	AVX; FMA			Unaligned Access in Vector Loop
Vectorized (Remainder) [Not Executed]		AVX2			4	2,68x	FMA	Float32	2	128	AVX; FMA			Unaligned Access in Vector Loop
Vectorized (Body) [Not Executed]		AVX512			32	36,34x	FMA	Float32	4	512	AVX512F_512	Unrolled	1	Unaligned Access in Vector Loop

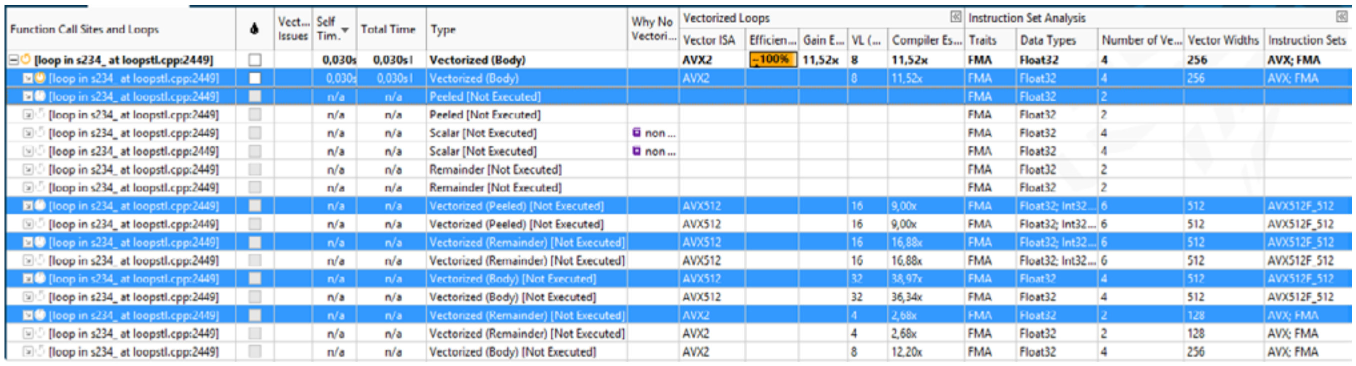
Source | Top Down | Loop Analytics | Loop Assembly | Recommendations | Compiler Diagnostic Details

Module: lcd_cxxi.exe!0x14005bea8

```
            Address     Line   Assembly                                          Total Time  %  Self Time  %  Traits
remainder   0x14005bea8        Block 1:
            0x14005bea8 6251   add rsi, 0x10
            0x14005beac 6251   vpcmpd k1, k0, zmm3, zmm2, 0x2
            0x14005beb3 6251   vpaddd zmm3, k0, zmm3, zmm1
            0x14005beb9 6252   vmovups zmm5, k1{z}, zmmword ptr [r8+r15*1]
            0x14005bec0 6252   vmovups zmm10, k1{z}, zmmword ptr [r8+rbp*1]
            0x14005bec7 6252   vmovups zmm4, k1{z}, zmmword ptr [r8+rbx*1]
            0x14005bece 6252   vfmadd213ps zmm10, k0, zmm4, zmm5                                             FMA
            0x14005bed4 6252   vmovups zmmword ptr [r8+rcx*1], k1, zmm10
            0x14005bedb 6251   add r8, 0x40
            0x14005bedf 6251   cmp rsi, rax
            0x14005bee2 6251   jb 0x14005bea8 <Block 1>
```

According to the data in the Vectorization details, it turns out that the AVX-512 code used masked operations in the remainder loop, leading to a 16.88x speedup, while the AVX2 remainder loop had a speedup of only 2.60x.

Using masked operations enables the AVX-512 version to vectorize the “peel” loop as well. Note that in the loop below, the AVX-512 version has vectorized peel, body, and remainder loops.



Function Call Sites and Loops		Vect... Issues	Self Tim.	Total Time	Type	Why No Vectori...	Vectorized Loops					Instruction Set Analysis				
							Vector ISA	Efficien...	Gain E...	VL (...	Compiler Es...	Traits	Data Types	Number of Ve...	Vector Widths	Instruction Sets
[loop in s234_ at loopstl.cpp:2449]	☐		**0,030s**	**0,030s**	**Vectorized (Body)**		**AVX2**	**~100%**	**11,52x**	**8**	**11,52x**	**FMA**	**Float32**	**4**	**256**	**AVX; FMA**
[loop in s234_ at loopstl.cpp:2449]	☐		0,030s	0,030s	Vectorized (Body)		AVX2			8	11,52x	FMA	Float32	4	256	AVX; FMA
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Peeled [Not Executed]							FMA	Float32	2		
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Peeled [Not Executed]							FMA	Float32	2		
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Scalar [Not Executed]	non ...						FMA	Float32	4		
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Scalar [Not Executed]	non ...						FMA	Float32	4		
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Remainder [Not Executed]							FMA	Float32	2		
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Remainder [Not Executed]							FMA	Float32	2		
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Vectorized (Peeled) [Not Executed]		AVX512			16	9,00x	FMA	Float32; Int32...	6	512	AVX512F_512
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Vectorized (Peeled) [Not Executed]		AVX512			16	9,00x	FMA	Float32; Int32...	6	512	AVX512F_512
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Vectorized (Remainder) [Not Executed]		AVX512			16	16,88x	FMA	Float32; Int32...	6	512	AVX512F_512
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Vectorized (Remainder) [Not Executed]		AVX512			16	16,88x	FMA	Float32; Int32...	6	512	AVX512F_512
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Vectorized (Body) [Not Executed]		AVX512			32	38,97x	FMA	Float32	4	512	AVX512F_512
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Vectorized (Body) [Not Executed]		AVX512			32	36,34x	FMA	Float32	4	512	AVX512F_512
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Vectorized (Remainder) [Not Executed]		AVX2			4	2,68x	FMA	Float32	2	128	AVX; FMA
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Vectorized (Remainder) [Not Executed]		AVX2			4	2,68x	FMA	Float32	2	128	AVX; FMA
[loop in s234_ at loopstl.cpp:2449]	☐		n/a	n/a	Vectorized (Body) [Not Executed]		AVX2			8	12,20x	FMA	Float32	4	256	AVX; FMA

The AVX-512 vectorized remainder has a “full” VL of 16, while the AVX2 version has a VL of 4—less than the VL of the AVX2 body. The AVX2 version has only a scalar peel loop, while AVX-512 peel is vectorized with an estimated speedup 9.0x.

Key Takeaways

A couple of things to summarize:

- Now you can generate and analyze the code that targets multiple ISAs at once on a single machine. In addition, you can make performance predictions based on the compiler reports.
- With Intel Advisor, you can now view ISA-specific “families” of instructions, used for individual instructions, and also view traits that are code-path specific.

Also see:

- **Intel® C++ Compiler Code Generation Options**
- **Intel® Fortran Compiler Code Generation Options**
- **x, Qx compiler options of the Intel® C++ Compiler**
- **x, Qx compiler options of the Intel® Fortran Compiler**
- **ax, Qax compiler options of the Intel® C++ Compiler**
- **ax, Qax compiler options of the Intel® Fortran Compiler**

Conclusion

In this article, we presented an overview of Intel Advisor XE 2016 and also the feature that enables you to analyze the code that targets AVX-512 ISAs while running this code on a machine with only an AVX2-enabled processor. We also presented examples that demonstrate how Vectorization Advisor can assist you in vectorizing C++ STL code.

To get the most out of your hardware, you need to modernize your code with vectorization and threading. Taking a methodical approach such as the one outlined in this paper, and taking advantage of the powerful tools in Intel Parallel Studio XE, can make the modernization task dramatically easier.

Software

INTEL® SOFTWARE PRODUCTS TECHNICAL WEBINARS

Spring 2016 Series

Watch Weekly for Free Advice from Intel Experts

Build better data analysis applications, vectorize effectively, and boost performance. Register now for each session you'd like to attend.

DATE	TOPIC	PRESENTER	
Wednesday, March 16 9:00 to 10:00 a.m. PDT	Have a Heart: Love Your Hybrid Programs	James Tullos	REGISTER
Tuesday, March 22 9:00 to 10:00 a.m. PDT	We Are Family: Harnessing Heterogeneous Systems with Intel® Threading Building Blocks	Mike Voss	REGISTER
Tuesday, March 29 9:00 to 10:00 a.m. PDT	Effective Parallel Optimizations with Intel® Fortran	Martyn Corden	REGISTER
Tuesday, April 5 9:00 to 10:00 a.m. PDT	Faster Data Applications on Spark* Clusters Using Intel® Data Analytics Acceleration Library	Zhang Zhang	REGISTER
Tuesday, April 12 9:00 to 10:00 a.m. PDT	A New Era for OpenMP*: Beyond Shared Memory Parallel Programming	Xinmin Tian	REGISTER
Tuesday, April 19 9:00 to 10:00 a.m. PDT	Improving Vectorization Efficiency Using Intel® Standard Data Layout Template Library	Anoop Madhusoodhanan Prabha	REGISTER
Tuesday, April 26 9:00 to 10:00 a.m. PDT	Vectorize or Performance Dies: Tune for the Latest AVX SIMD Instructions—Even without the Latest Hardware	Kevin O'Leary	REGISTER
Tuesday, May 3 9:00 to 10:00 a.m. PDT	Boost Python* Performance with Intel® Math Kernel Library	Ricardo Covarrubias Carreno	REGISTER
Tuesday, May 10 9:00 to 10:00 a.m. PDT	Intermittent Multithreading Bugs: Find and Squash Races, Deadlocks, and Memory Bugs	Kevin O'Leary	REGISTER
Tuesday, May 17 9:00 to 10:00 a.m. PDT	Understanding the Effect of NUMA on Your Workloads: Intel® VTune™ Amplifier with Memory Analysis	Bhanu Shankar	REGISTER
Tuesday, May 24 9:00 to 10:00 a.m. PDT	Performance Analysis of Python* Applications with Intel® VTune™ Amplifier	Vasilij Litvinov	REGISTER
Tuesday, May 31 9:00 to 10:00 a.m. PDT	Building Fast Code for Data Compression and Protection in Intel® Integrated Performance Primitives	Yu Chao	REGISTER

For more complete information about compiler optimizations, see our **Optimization Notice**.

Software

THE PARALLEL UNIVERSE