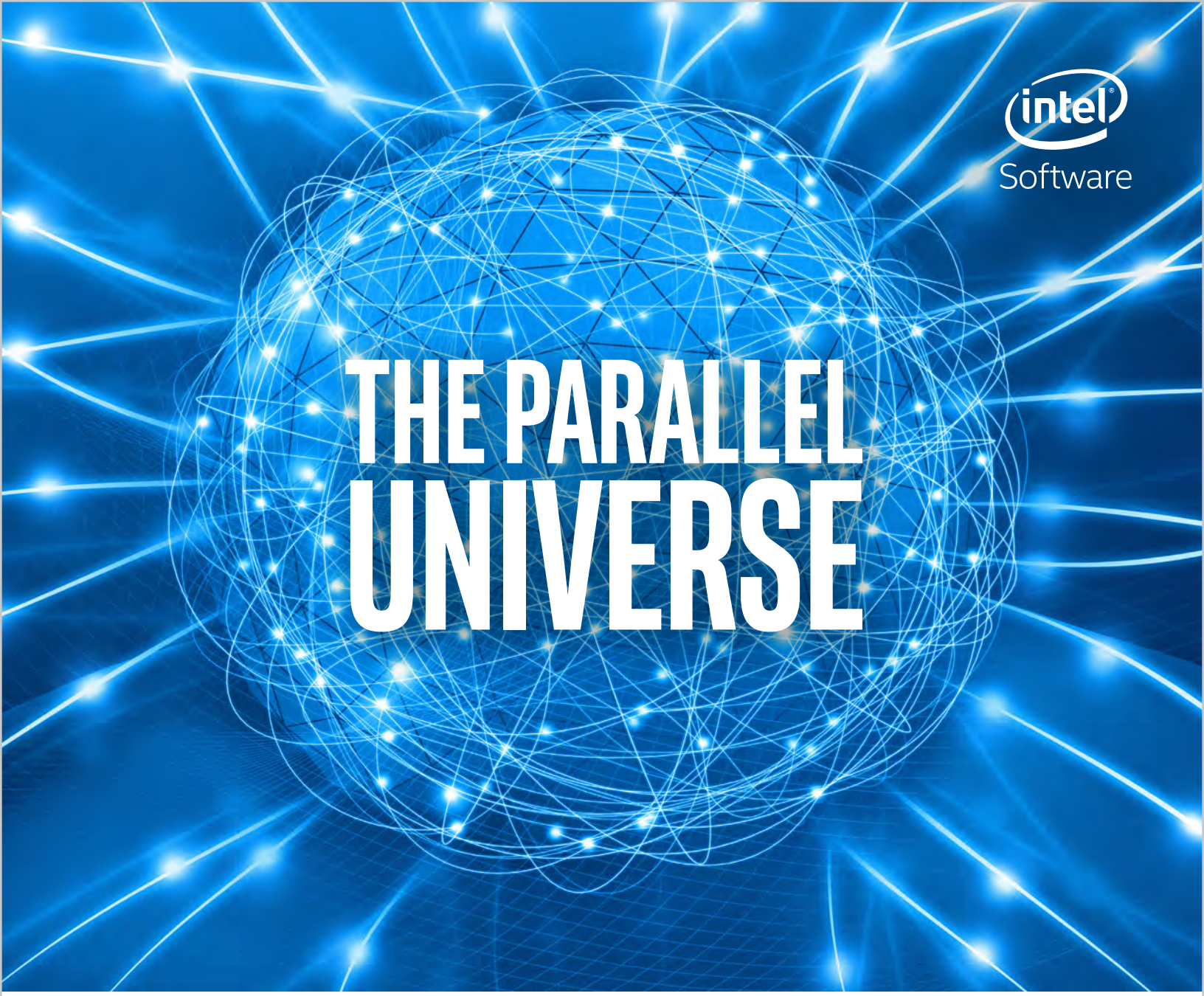


The Present and Future of OpenMP*

How the Gold Standard Parallel Programming Language Has Improved

Improve Performance on Multicore and Many-Core Intel® Architectures

Identify Scalability Problems in Parallel Applications

Issue
27
2017

CONTENTS

Letter from the Editor 3

The Changing HPC Landscape Still Looks the Same

by Henry A. Gabb, *Principal Engineer* at Intel Corporation

FEATURE

The Present and Future of the OpenMP* API Specification 5

How the Gold Standard Parallel Programming Language Has Improved with Each New Version

Reducing Packing Overhead in Matrix-Matrix Multiplication 21

Improve Performance on Multicore and Many-Core Intel® Architectures, Particularly for Deep Neural Networks

Identify Scalability Problems in Parallel Applications 26

How to Improve Scalability for Intel® Xeon and Intel® Xeon Phi™ Processors Using New Intel® VTune™ Amplifier Memory Analysis

Vectorization Opportunities for Improved Performance with Intel® AVX-512 47

Examples of How Intel® Compilers Can Vectorize and Speed up Loops

Intel® Advisor Roofline Analysis 56

A New Way to Visualize Performance Optimization Trade-Offs

Intel-Powered Deep Learning Frameworks 74

Your Path to Deeper Insights

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

THE CHANGING HPC LANDSCAPE STILL LOOKS THE SAME

Henry A. Gabb, *Principal Engineer* at Intel Corporation, is a long-time high-performance and parallel computing practitioner and has published numerous articles on parallel programming. He was editor/coauthor of "Developing Multithreaded Applications: A Platform Consistent Approach" and was program manager of the Intel/Microsoft Universal Parallel Computing Research Centers.

Celebrating 20 Years of OpenMP*

The OpenMP* application programming interface turns 20 this year. To celebrate, we tapped Michael Klemm (the current CEO of the OpenMP Architecture Review Board, or ARB) and some of his colleagues to give an overview of the newest features in the specification—particularly, enhancements to task-based parallelism and offloading computations to specialized accelerators.

Our feature article covers "**The Present and Future of the OpenMP API Specification**," so I'll say a little about its past. I half-jokingly refer to the early to mid-1990s as the "bad old days" of high-performance computing (HPC). There were many, many different parallel programming models and parallel architectures dotting a fast-changing HPC landscape. For distributed-memory architectures, there were low-level, message-passing methods like SHMEM, high-level, methods like PVM or **MPI**, and even higher levels of abstraction with High Performance Fortran and Unified Parallel C. For shared-memory architectures, there were low-level threading methods like Pthreads or higher-level compiler-directed threading. One thing was clear: There were no magic compilers that could automatically parallelize real applications. Parallel compiler directives were the next best thing.

For those of us who remember parallel compiler directives before OpenMP, there were many vendor-specific sets to choose from (e.g., Cray, SGI, Intel, Kuck and Associates, Inc.), each doing the same thing but with different syntaxes. In exasperation, several large governmental HPC facilities demanded a unified syntax for parallel compiler directives.

OpenMP was born in 1997. Most of the original vendors are still on the ARB, and many more members have been added since (the ARB currently has 29 members). It remains the gold standard for portable, vendor-neutral parallel programming directives because it never lost sight of its original purpose.

Today, MPI and OpenMP cover most application requirements in HPC. There are still challenges. Memory subsystems are as unbalanced as ever, different processor architectures now commonly exist within the same system, and keeping data coherent among these different processing elements is an additional burden on the programmer. But MPI and OpenMP continue to evolve with these challenges, so the HPC future looks bright.

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

New Tools for Tuning Serial Performance

Parallelism is great, but would you parallelize code that has not been properly tuned? No, you wouldn't. So this issue of *The Parallel Universe* also looks at tuning serial performance. My first supercomputer was a Cray X-MP, so I learned early the importance of vectorization. "**Vectorization Opportunities for Improved Performance with Intel® AVX-512**" gives a good overview of tuning code with the new Intel AVX-512 instruction set and shows how to use these instructions to expose vectorization opportunities that were not previously possible. The new Intel® Advisor Roofline and Intel® VTune™ Amplifier Memory Analysis features help visualize performance optimization tradeoffs and how memory access is affecting an application's performance. These features are demonstrated in "**Intel® Advisor Roofline Analysis**" and "**Identify Scalability Problems in Parallel Applications**." We round out this issue with tips for optimizing general matrix-matrix multiplication operations in the Intel® Math Kernel Library ("**Reducing Packing Overhead in Matrix-Matrix Multiplication**") and a brief overview of Intel software support for machine learning ("**Intel-Powered Deep Learning Frameworks**").

Hello, I'm New Here

Finally, I'd like to introduce myself as the new editor of *The Parallel Universe*. I've been doing HPC since about 1990, but I was originally doing research in computational life science. Each successive research project required more computing power. To stay relevant, I had to learn about **performance tuning** and parallel programming. My academic background is in biochemistry and genetics, so I resented the intrusion of computer science into my scientific domain. But my initial resistance gave way to fascination when I saw how HPC could change my research and make it possible to answer new and bigger research questions. Hardware and software advances allow me to quickly run simulations on my laptop that once took days on a circa 1995 supercomputer. I used to dread the heterogeneous parallel computing future. Now, I welcome it with the same fascination I had as a young graduate student.

Henry A. Gabb
January 2017

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

THE PRESENT AND FUTURE OF THE OPENMP* API SPECIFICATION

How the Gold Standard Parallel Programming Language Has Improved with Each New Version

Michael Klemm, *Senior Staff Application Engineer,* Intel Deutschland GmbH; Alejandro Duran, *Application Engineer,* Intel Corporation Iberia; Ravi Narayanaswamy, *Senior Staff Development Engineer;* Xinmin Tian, *Senior Principal Engineer;* and Terry Wilmarth, *Senior Development Engineer,* Intel Corporation

There are two decades of history associated with the OpenMP* API and, since its inauguration, OpenMP features have been added to keep up with developments in hardware and software to ensure that you can use it to program the hardware that you have. Since the release of version 4.0 in 2013, the OpenMP language has supported heterogeneous and SIMD programming. Similarly, support for programs with irregular parallelism was improved in 2008 with the addition of tasking constructs. OpenMP Technical Report 4: Version 5.0 Preview 1 (TR4 for short) is the next step in the evolution of the OpenMP language. It adds task reductions, extends SIMD parallel programming, and considerably extends the productivity of heterogeneous programming. In this article, we review existing OpenMP features and provide a preview of what will be coming soon in implementations supporting TR4.

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

Tasking: Express Yourself with Tasks

Tasking, or task-based programming, is an important concept for applications that require irregular parallelism (e.g., recursive algorithms, graph traversals, and algorithms operating on unstructured data). Since OpenMP version 3.0, the task construct has provided a convenient way to express the concurrent execution of small units of work that are handed to a scheduler in the OpenMP runtime system.

```
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 Simple example using the new taskloop construct together with an OpenMP* task

Figure 1 illustrates the creation of an OpenMP task to execute a long-running function and then a loop that has been parallelized using the `taskloop` construct. This construct appeared with OpenMP 4.5 and provides syntactic sugar to allow programmers to easily parallelize loops using OpenMP tasks. It divides the loop iteration space into chunks and creates one task for each chunk. The construct supports several clauses to allow fine control (e.g., `grainsize` to control the amount of work per task and `collapse` to create a product loop out of the `i` and `j` loops). TR4 extends the expressiveness of OpenMP tasks by defining new clauses for the `taskgroup`, `task`, and `taskloop` constructs to perform reductions across the generated tasks.

Figure 2 illustrates the creation of tasks to process a linked list and find the minimum value of all elements in the list. The `parallel` construct creates a parallel region to have worker threads available for task execution. The `single` construct then restricts execution to one thread that traverses the linked list and generates one task for each list item via `omp` task. This is a common way to implement a producer-consumer pattern in OpenMP.

Task reductions in TR4 use the `taskgroup` construct that was introduced in OpenMP version 4.0. It was designed to group tasks logically and to provide a way to await completion of all the tasks in the group. TR4 extends the `taskgroup` construct to perform reductions through the `task_reduction` clause, as illustrated in **Figure 2**. If this clause is added to the construct, all partial results gathered by the individual tasks are aggregated to form the final result at the end of the `taskgroup` region. Tasks that contribute to the reduction operation must have an `in_reduction` clause that matches the reduction clause of their `taskgroup`.

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

Starting with TR4, the `taskloop` construct supports the `reduction` and `in_reduction` clauses with their task reduction semantics. If a `reduction` clause appears on the `taskloop` construct, an implicit task group is created, which performs the requested reduction operation at the end of the loop. If an `in_reduction` clause is added, the tasks generated by the `taskloop` construct participate in the reduction of an outer `taskgroup` region.

```
int find_minimum(list_t * list) {
    int minimum = INT_MAX;
    list_t * ptr = list;
#pragma omp parallel
#pragma omp single
#pragma omp taskgroup task_reduction(min:minimum)
    {
        for (ptr = list; ptr ; ptr = ptr->next) {
#pragma omp task firstprivate(ptr) in_reduction(min:minimum)
            {
                int element = ptr->element;
                minimum = (element < minimum) ? element : minimum;
            }
        }
    }
    return minimum;
}
```

2 Traversing a linked list and computing the minimum using task reductions

Offloading: Making the Most of Coprocessors

The OpenMP API strives to improve the usability of offloading pragmas based on user feedback. To that end, new features have been added to TR4 and some existing features have been enhanced. One of the key new features is the ability to automatically detect functions used in offload regions and treat them as if they appeared in a `declare target` directive. Previously, all functions called in an offload region had to be explicitly tagged using `declare target` directives. This was hard work, especially if the routines were in header files not owned by the programmer (e.g., the Standard Template Library), and would require `declare target` directives on the header file itself, which would create a copy of every function in the header file for the device even if some functions were not used in the offload region.

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

```
#pragma omp declare target
void foo() {
    // ...
}
#pragma omp end declare target

void bar()  {
#pragma omp target
    {
        foo();
    }
}
```

```
void foo() {
    // ...
}


void bar() {
#pragma omp target
    {
        foo();
    }
}
```

3 Automatically detect functions used in offload regions

In **Figure 3**, the code on the left shows what was required in OpenMP version 4.5. Starting with TR4, the code on the right side is sufficient due to the implicit detection and creation of the device function.

Automatic detection also extends to variables with static storage duration in TR4. The examples in **Figure 4** are equivalent.

```
int x;
#pragma omp declare target to (x)

void bar() {
    #pragma omp target
    {
        x = 5;
    }
}
```

```
int x;


void bar() {
    #pragma omp target
    {
        x = 5;
    }
}
```

4 Automatic detection for variables with static storage duration

OpenMP version 4.5 introduced the `use_device_ptr` clause. The variable in `use_device_ptr` must be mapped before it can be used. To achieve this, the programmer would need to use a separate `#pragma target data` clause, as a variable can appear in only one data clause. Thus, the OpenMP directives in **Figure 5** are needed.

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

```
#pragma omp target data map(buf)
#pragma omp target data use_device_ptr(buf)
```

5 Mapping the variable

In TR4, an exception has been made so that the variable can appear in both the map and `use_device_ptr` clauses in a single construct, as shown in **Figure 6**.

```
#pragma omp target data map(buf) use_device_ptr(buf)
```

6 Variable can appear in both the map and `use_device_ptr` clauses

Static data members are now permitted in a class inside an omp declare target construct. Class objects with static members can also be used in a map clause (**Figure 7**).

```
#pragma omp declare target
class C {
    static int x;
    int y;
}
class C  myclass;
#pragma omp end declare target

void bar() {
#pragma omp target map(myclass)
    {
        myclass.x = 10
    }
}
```

7 Class objects with static members used in a `map` clause

In addition, virtual member functions are allowed in classes inside an `omp declare target` construct or objects used in a `map` clause. The only caveat is that the virtual member functions can be invoked only on a device if the object is created on the same device.

In OpenMP 4.5, scalar variables used in a `reduction` or `lastprivate` clause on a combined construct for which the first construct is target are treated as `firstprivate` for the target construct. That results in the host value never being updated, surprisingly. To update the value on the host, the programmer had to separate the `omp` target directive from the combined construct and explicitly map the scalar variable. In TR4, such variables are automatically treated as if they had a `map(tofrom:variable)` applied to them.

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

If a section of a named array is mapped using `omp target` data, any nested `omp target` inside the `omp target data` construct that references the array would require an implicit mapping to either the same section or a subsection of the array used in the outer `omp target data map` clause. If the explicit mapping is omitted on the inner `omp target` region, the implicit mapping rule kicks in, which would imply that the entire array is mapped according to OpenMP version 4.5. This would result in a runtime error from mapping a larger-sized array when a subsection of the array is already mapped. Similarly, mapping a field of a structure variable in the outer `omp target data` construct and using the address of the structure variable inside a nested `omp` target construct would result in an attempt to map the entire structure variable when part of the structure is already mapped. TR4 has fixed these cases to give the behavior that programmers typically expect **(Figure 8)**.

```
struct {int x,y,z} st;
int A[100];
#pragma omp target data map(s.x A[10:50])
{
#pragma omp target
    {
        A[20] = ;       // error in OpenMP 4.5, Ok in TR4
        foo(&st);       // error in OpenMP 4.5, OK in TR4
    }
#pragma omp target map(s.x, A[10:50])
    {
        A[20] = ;       // Ok OpenMP 4.5 and TR4
        foo(&st);       // Ok OpenMP 4.5 and TR4
    }
}
```

8 Improved mapping

The new features in TR4 improve the programmability of offloading using OpenMP, requiring fewer modifications to the application. The automatic detection of variables and functions used in `target` regions removes the need for explicit specification. Similarly, the elimination of the need to repeat `map` clauses inside nested regions and allowing variables to appear in both `map` and `use_device_ptr` reduces the number of OpenMP directives required. The changes to the behavior of reduction variables aligns the language with programmer expectations. Overall, the cleaner semantics make the use of offload devices within OpenMP applications simpler and more intuitive.

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

Efficient SIMD Programming

SIMD Loops with Cross-Iteration Dependencies

OpenMP version 4.5 extends the `ordered` construct by adding a new `simd` clause. The `ordered simd` construct declares that a structured block in the SIMD loop or SIMD function must be executed in iteration order or in the order of function calls, respectively. **Figure 9** shows the use of the `ordered simd` block to preserve read-write, write-read, and write-write ordering within each iteration and among iterations, while the entire loop can be executed concurrently using SIMD instructions. In the first `ordered simd` block, the index `ind[i]` of array a may have a write-write conflict (e.g., `ind[0] = 2, ind[2] = 2`), so it needs to be serialized by the `ordered simd` to allow **vectorization** of the entire loop. In the second `ordered simd` block, the `myLock(L)` and `myUnlock(L)` operations must be in a single `ordered simd` block. Otherwise, as part of the loop vectorization (e.g., for a vector length of two), the calls to `myLock(L)` and `myLock(L)` will be expanded to two calls as follows: `{myLock(L); myLock(L); …; myUnlock(L); myUnlock(L);}`. Nesting the lock functions will typically result in a deadlock. The `ordered simd` construct shown in the example creates the proper sequence `{myLock(L); …; myUnlock(L); …; myLock(L); myUnlock(L);}`.

```
#pragma omp simd
for (i = 0; i < N; i++) {
    // ...
#pragma omp ordered simd
    {
        // write-write conflict
        a[ind[i]] += b[i];
    }
    // ...
#pragma omp ordered simd
    {
       // atomic update
       myLock(L)
       if (x > 10) x = 0;
       myUnlock(L)
    }
    // ...
}
```

```
#pragma omp simd
for (i = 0; i < N; i++) {
    // ...
#pragma omp ordered simd
    {
        if (c[i]) > 0) q[j++] = b[i];
    }
    // ...
#pragma omp ordered simd
    {
        if (c[i] > 0) q[j++] = d[i];
    }
    // ...
}
```

9 Preserving read-write, write-read, and write-write ordering within each iteration and among iterations

When using the `simd` clause on the `ordered` construct, caution is required to not violate inherent dependencies between two ordered `simd` blocks. **Figure 9** shows incorrect uses of `#pragma omp ordered simd`, as the order of stores is changed under SIMD execution with respect to its serial execution. Assume `c[0] = true` and `c[1] = true`. When the above loop is executed serially, the order of stores is: `q[0] = b[0], q[1] = d[0], q[2] = b[1], q[3] = d[1]`, and so forth. However, when the loop is executed concurrently with a

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

vector length of two, the order of stores is: `q[0] = b[0], q[1] = b[1], q[2] = d[0], q[3] = d[1], ...` The change in store ordering is due to a violation of the write-to-read dependency on the variable `j` between the two `ordered simd` blocks in the loop. The correct use is to merge the two `ordered simd` blocks into a single `ordered simd` block.

REF/UVAL/VAL Modifier Extensions to the Linear Clause

The `linear` clause provides a superset of the functionality provided by the `private` clause. When a `linear` clause is specified on a construct, the value of the new list item on each iteration of the associated loop(s) corresponds to the value of the original list item before entering the construct, plus the logical number of the iterations multiplied by the linear step. The value corresponding to the sequentially last iteration of the associated loop(s) is assigned to the original list item. When a `linear` clause is specified on a declarative directive, all list items must be formal parameters (or, in Fortran, dummy arguments) of a function that will be invoked concurrently on each SIMD lane.

The rationale behind adding `ref/uval/val` modifiers to the `linear` clause is to provide a way for programmers to precisely specify the `linear` or `uniform` property of memory references with respect to address and data value so the compiler can leverage the information to generate efficient SIMD code using unit-stride loads/stores instead of gathers/scatters. Essentially, for implicitly referenced linear arguments, it would be better to have reference as linear. The semantics of `uval/val/ref` is described as:

- `linear(val(var):[step])` indicates that the value is linear even if the `var` is passed by its reference. The vector of addresses is passed for passed by reference. In this case, the compiler must generate gathers or scatters.
- `linear(uval(var):[step])` indicates that the value passed by reference is linear while the reference itself is uniform. So the reference to the first lane is passed, but other values can be constructed using step. The compiler can use general-purpose registers to pass the base address and compute its linear value.
- `linear(ref(var):step)` indicates that the parameter is passed by reference, the underlying reference is linear, and the memory access will be linear unit-stride or nonstrided depending on step. The compiler can use general-purpose registers to pass the base address and compute its linear address.

Figure 10 shows a function `FOO` with arguments X and Y, which are pass-by-reference in Fortran. The "`VALUE`" attribute does not change this behavior. It says only that the updated value will not be visible to the caller per the Fortran 2008 language specification. Since the references of X and Y are not annotated as linear, the compiler must generate gather instructions to load (X0, X1, X2, X3) and (Y0, Y1, Y2, Y3), assuming the vector length is four. In **Figure 11**, the references to X and Y are annotated as linear, so the compiler can generate unit-stride SIMD loads for much better performance.

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

```
      REAL FUNCTION FOO(X, Y)
!$omp declare simd(FOO)
      REAL, VALUE :: Y    !! pass by reference
      REAL, VALUE :: X    !! pass by reference
      FOO = X + Y         !! gathers generated
                          !! based on vector
                          !! of addresses
      END FUNCTION FOO
      ! ...
!omp$ simd private(X,Y)
      DO I= 0, N
        Y = B(I)
        X = A(I)
        C(I) += FOO(X, Y)
      ENDDO
```

10 Linearity of reference X and Y is unknown at compile time

```
      REAL FUNCTION FOO(X, Y)
!$omp declare simd(FOO) linear(ref(X), ref(Y))
      REAL, VALUE :: Y    !! pass by reference
      REAL, VALUE :: X    !! pass by reference
      FOO = X + Y         !! unit stride
                          !! SIMD loads
      END FUNCTION FOO
      ! ...
!omp$ simd private(X,Y)
      DO I= 0, N
        Y = B(I)
        X = A(I)
        C(I) += FOO(X, Y)
      ENDDO
```

11 References to X and Y annotated as linear

In **Figure 12**, the function `add_one` is annotated as a SIMD function. It has a C++ reference argument `const int &p`. Assuming a vector length of four, if `p` is annotated as `linear(ref(p)),` the compiler can generate the unit-stride load instruction with the base address `p` in the `rax` register to load `p[0], p[1], p[2],` and `p[3]` to the `xmm0` register. In that case, the `add_one` function requires only three instructions.

```
#pragma omp declare simd notinbranch // linear(ref(p))
__declspec(noinline)
int add_one(const int& p) {
    return (p + 1);
}
```

12 SIMD code comparison with and without `linear(ref(p))` annotation

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

However, if `p` is not annotated as `linear(ref(p)),` the compiler has to assume that four different addresses `p0, p1, p2,` and `p3` are passed in via two `xmm` registers, and the gather operation is emulated with a sequence of scalar load and packing instructions. As a result, the `add_one` function now requires 16 instructions rather than three.

Overall, the additional SIMD features in OpenMP version 4.5 allow the user to provide more information to the compiler, which allows vectorization of more loops and the generation of better vector code in many circumstances.

Affinity: Thread Placement Made Easy

The OpenMP version 4.0 specification gave users a standard way to control thread affinity for the first time. It introduced two new concepts to the language:

1. Binding policy
2. Place partition

The binding policy, specified by the `bind-var` Internal Control Variable (ICV), determines where the threads of a team will be bound relative to the parent thread's place. The place partition, specified by the `place-partition-var` ICV, is the set of places to which threads can be bound. Once a thread is bound to a place for a given team, it should not be moved from that place.

There are three binding policies defined by the specification: `master, close,` and `spread`. In describing these policies, we will consider a set of four places, each one a core with two threads. We will show examples of placing three threads and six threads on those places, and we assume that the parent thread will always be on the third place. In the `master` policy, the master thread is bound to the parent thread's place, and then the remaining threads in the team are assigned to the same place as the master thread **(Table 1)**.

	Place 1: {0,1}	Place 2: {2,3}	Place 3: {4,5} (parent)	Place 4: {6,7}
Three threads			0, 1, 2	
Six threads			0, 1, 2, 3, 4, 5	

Table 1. Thread placement for the `master` policy

The `close` policy starts by placing the master thread on the parent thread's place, and then proceeds in a round-robin fashion with the remaining threads in the team. To place `T` threads on `P` places, the master's place gets roughly the first `T/P` threads, then the next place in the place partition gets the next `T/P` threads, and so on, wrapping around in the place partition as needed, giving a distribution (**Table 2**).

	Place 1: {0,1}	Place 2: {2,3}	Place 3: {4,5} (parent)	Place 4: {6,7}
Three threads	2		0	1
Six threads	4	5	0,1	2,3

Table 2. Thread placement for the close policy

With the `spread` policy, things get very interesting. The placement of threads will be such that they are spread out over the available places. This is accomplished by forming `T` roughly even subpartitions of the place partition, or `P` partitions if `T >= P`. If `T <= P`, each thread gets its own subpartition, starting with the master thread, which will get the subpartition containing the place to which the parent thread is bound. Each subsequent thread is bound to the first place in each subsequent subpartition, wrapping around as needed. If `T > P`, sets of consecutive threads get the same subpartition, which in this case will consist of a single place. Thus, all the threads in the set will be bound to the same place. We show the subpartitions formed in **Table 3** in curly braces. These are important if nested parallelism is used, since they affect the available resources used by each nested parallel region.

	Place 1: {0,1}	Place 2: {2,3}	Place 3: {4,5} (parent)	Place 4: {6,7}
Three threads	1 {{0,1}}	2 {{2,3}}	0 {{4,5},{6,7}} Note: 0 is bound to {4,5}	
Six threads	4 {{0,1}}	5 {{2,3}}	0,1 {{4,5}}	2,3 {{6,7}}

Table 3. Thread placement and subpartitions for the spread policy

OpenMP version 4.0 also provides a query function for the thread affinity binding policy: `omp_proc_bind_t omp_get_proc_bind()`. It returns the binding policy to be used in the next `parallel` region (assuming that no `proc_bind` clause is specified on that region).

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

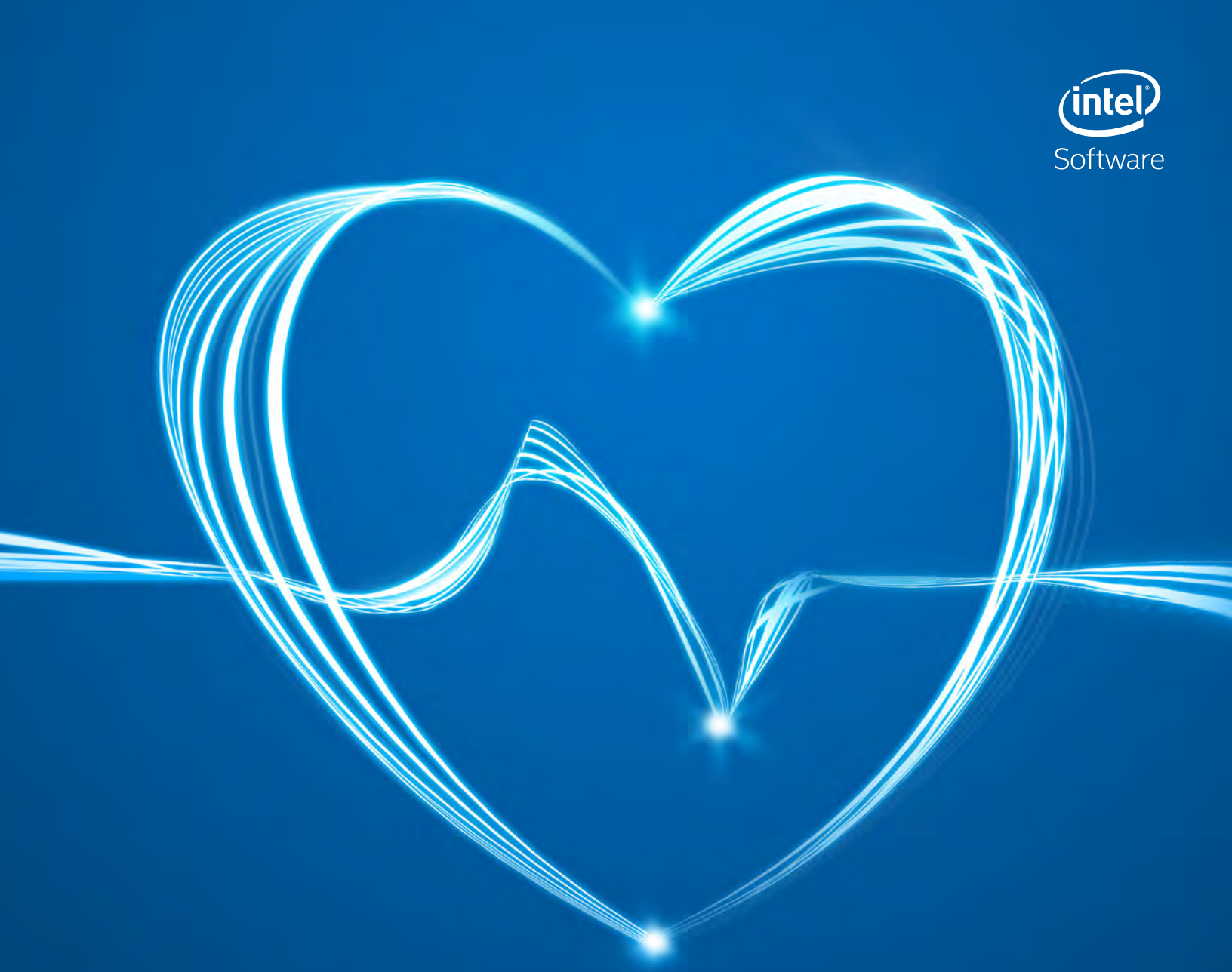
What is really interesting about the `spread` policy is what happens with the subpartition. With the `master` and `close` policies, each implicit task inherits the place partition of the parent implicit task. But, in the `spread` policy, implicit tasks get their `place-partition-var` ICV set to the subpartition instead. This means that a nested `parallel` construct will have all of its threads placed within the subpartition of its parent.

The value of `bind-var` can be initialized via the environment variable `OMP_PROC_BIND`. The value of `bind-var` can also be overridden by the addition of a `proc_bind` clause to a `parallel` construct. Specifying the `place-partition-var` is accomplished via the `OMP_PLACES` environment variable. Places can be hardware threads, cores, sockets, or specific quantities of those. They can also be explicit processor lists. More details can be found in the OpenMP API specification.

The OpenMP 4.5 specification enhanced the language's affinity capabilities by providing a set of functions capable of querying aspects of the place partition and binding place of the current thread. These new API functions are useful for confirming the correctness of the settings to achieve the programmer's desired thread affinity. This is particularly important when the complexity of the code is high and nested parallelism is used in conjunction with the `spread` binding policy to place threads in nested parallel regions such that they share lower-level caches. These API functions are:

- `int omp_get_num_places();` Returns the number of places in the `place-partition-var` in the execution environment of the initial task.
- `int omp_get_place_num_procs(int place_num);` Returns the number of processors available to the execution environment in the place specified by `place_num` in the place partition.
- `void omp_get_place_proc_ids(int place_num, int *ids);` Gets the processors available to the execution environment in the place specified by `place_num` in the place partition, allocates an array to hold them, and puts that array at `ids`.
- `int omp_get_place_num(void);` Returns the number of the place in the place partition to which the encountering thread is bound.
- `int omp_get_partition_num_places(void);` Returns the number of places in the place partition of the innermost implicit task. Note that this differs from `omp_get_num_places()` in that it will show the effects of the `spread` binding policy as the place partition gets broken into subpartitions, whereas `omp_get_num_places()` will always show the full original place partition.
- `void omp_get_partition_place_nums(int *place_nums);` Gets the list of place numbers corresponding to the place partition of the innermost implicit task and allocates an array in `place_nums` to store them. Note that the place numbers are the numbers of the places in the full original place partition. This function is particularly useful to see which places from the original place partition appear in a subpartition resulting from the use of the spread binding policy.

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



CODE CAN SAVE LIVES

Your code can inspire medical breakthroughs for improved healthcare

Do more with Intel® Parallel Studio XE. It's designed to optimize your parallel code, so you can fast-track your high-performance computing needs to find the right solutions—and diagnosis—quicker.

Code amazing things ›

© 2017 Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.

Outlook on the OpenMP API Specification, Version 5.0

The OpenMP Architecture Review Board (ARB) is discussing other features that could appear in version 5.0 of the OpenMP specification. The most likely candidates are described in this section.

Memory Management Support

How to support increasingly complex memory hierarchies within OpenMP applications is an area of active discussion. This complexity comes from multiple directions: new memories with different characteristics (such as MCDRAM on Intel® Xeon Phi™ Processors or Intel® 3D XPoint™ memory), the need to request certain characteristics of the allocated memory to ensure good performance (e.g., certain alignments or page sizes), the need for special compiler support for some memories, NUMA effects, etc. Furthermore, many new memory technologies are being investigated, so any proposal needs to be extensible to handle future technologies.

The current working direction is based on two key concepts that try to model the different technologies and operations: memory spaces and allocators. Memory spaces represent system memory with a set of traits (e.g., page size, capacity, bandwidth, etc.) that programmers can specify to find the memory that they want their program to use. Allocators are objects that allocate memory from a memory space and can also have traits that alter their behavior (e.g., the alignment of allocations).

New APIs are defined to manipulate memory spaces and allocators, and to allocate and deallocate memory. Separating the calls for creating an allocator and allocating memory allows the construction of maintainable interfaces where decisions about where memory should be allocated are taken in a common "decision" module. **Figure 13** shows how the proposal can be used to select the memory with the highest bandwidth in the system that uses 2 MB pages and defines two different allocators from that memory: one that ensures allocations are 64-byte-aligned and one that does not.

```
omp_memtrait_set_t  trait_set;
omp_memtrait_t  traits[] = {{OMP_MTK_BANDWIDTH,OMP_MTK_HIGHEST},
                            {OMP_MTK_PAGESIZE, 2*1024*1024}};
omp_init_memtrait_set(&trait_set,2,traits);

omp_memspace_t *amemspace = omp_init_memspace(&trait_set);

omp_alloctrait_t trait = {{OMP_ATK_ALIGNMENT},{64}};
omp_alloctrait_set_t trait_set;
omp_init_alloctrait_set(&trait_set,1,&trait);

omp_allocator_t *aligned_allocator = omp_init_allocator(amemspace,
                                                        &trait_set);
omp_allocator_t *unaligned_allocator = omp_init_allocator(amemspace, NULL);

double *a = (double *) omp_alloc( aligned_allocator, N * sizeof(double) );
double *b = (double *)omp_alloc( unaligned_allocator, N * sizeof(double) );
```

13 Selecting the memory with the highest bandwidth

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

A new `allocate` directive is proposed to affect the underlying allocation of variables that are not allocated with an API call (e.g., automatic or static variables). A new allocate clause can be used to affect the allocations done by OpenMP directives (e.g., private copies of variables). **Figure 14** shows how to use the directive to change the allocation of variables `a` and `b` to a memory with the highest bandwidth that also uses 2 MB pages. The private copies of `b` in the parallel region of the example are allocated on a memory with the lowest latency.

```
int a[N], b[M];
#pragma omp allocate(a,b) memtraits(bandwidth=highest, pagesize=2*1024*1024)

void example() {
#pragma omp parallel private(b) allocate(memtraits(latency=lowest):b)
    {
        // ...
    }
}
```

14 Changing the allocation of variables `a` and `b` to a memory with the highest bandwidth that also uses 2 MB pages

Improvements for Heterogeneous Programming

Several features are being considered to improve OpenMP's device support:

- Currently, structures in a `map` clause are bitwise copied, including pointer fields in a structure. If the programmer requires the pointer field to point to valid device memory, that would require creating the device memory and explicitly updating the pointer field on the device. The committee is discussing extensions that would enable the programmer to specify the automatic attaching/detaching of the pointer fields in a structure using a `map` clause by extending it to support pointer fields in structures.
- The ARB is considering allowing function pointers to be used in a `target` region and also allowing a function pointer to appear in `declare target`.
- New device `memcpy` routines that can execute asynchronously.
- Support to enable an "execute on device or fail" semantics of the `target` construct. Currently, target regions can execute silently on the host when the device is not available.
- Support for variables and functions that only exist on devices and are not copies of host-based ones.
- Support for multiple types of devices in a single application.

Tasking Improvements

Beyond the features discussed in this article, other features being considered for OpenMP 5.0 are:

- Enabling data dependencies between `taskloop` constructs.
- Enabling data dependencies in both the `task` construct and `taskloop` construct to contain expressions that could be expanded to multiple values to generate more than dependence from a single `depend` clause.
- Providing support to express task to thread affinity with patterns similar to those of `OMP_PROC_BIND`.

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

Other Changes

Other areas where there are active discussions of additional features for OpenMP 5.0 include:

- Upgrade the OpenMP base language specification to C11, C++11, or C++14, and Fortran 2008.
- Relax the restrictions of the `collapse` clause to allow nonrectangular loop shapes and allow code to appear between the loops in the nest.
- Allow reductions to happen in the middle of a parallel region without being associated with a work-sharing construct.

The OpenMP API is the gold standard of a portable and vendor-neutral parallel programming language for shared memory parallelization of C, C++, and Fortran applications in high-performance computing and beyond. And with upcoming developments in version 5.0, it promises to offer much more for developers to fully utilize the capabilities of modern processors.

BLOG HIGHLIGHTS

Code Modernization: Powering Scientific Discovery and Fostering Innovation Globally at CERN (Part 2)

BY RUSS BEUTLER (INTEL) ›

Intel recently talked to CERN openlab CTO Maria Girone to discuss how CERN and Intel work together to deliver improvements in processing speed, sometimes by factors, and how that impacts CERN's research on the basic constituents of matter. This conversation is meant to help developers understand how a Modern Code approach can help advance research and breakthroughs globally.

In Part 2 of the interview, Maria offers advice for developers around building a code modernization strategy and discusses the programs available to enable developers and students to develop their skills, advance their careers, and bring large-factor improvements to the applications they work with.

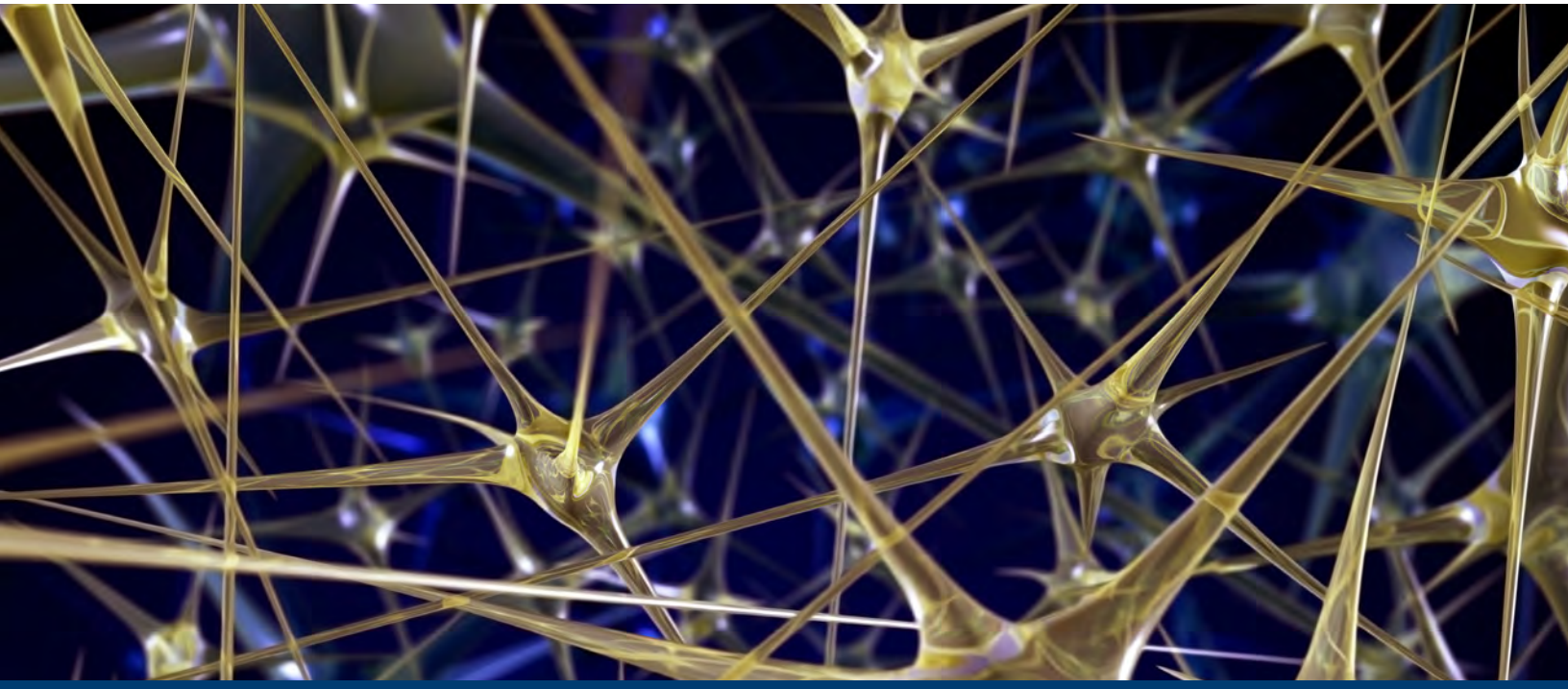
What advice do you have for developers and companies building a code modernization strategy and looking to ensure that their applications take full advantage of modern server hardware?

First, it's important to recognize that there's room for significant improvement when it comes to legacy code. And that it represents a good career opportunity for developers who are good at updating it.

Then, specifically, they should work to understand the gap between where their legacy software currently stands performance-wise, and the gains achievable through efficient parallelization and vectorization. Next, it's important to demonstrate the improvements that can be made by undertaking a code modernization effort.

Read more ›

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



REDUCING PACKING OVERHEAD IN MATRIX-MATRIX MULTIPLICATION

Improve Performance on Multicore and Many-Core Intel® Architectures, Particularly for Deep Neural Networks

Kazushige Goto, Murat Efe Guney, and Sarah Knepper, *Software Development Engineers*, Intel Corporation

General matrix-matrix multiplication (GEMM) is a fundamental operation in many scientific, engineering, and machine learning applications and is one of the key routines in the BLAS (basic linear algebra subprograms) domain. Four precisions (real single, real double, complex single, and complex double) of GEMM exist. In this article, we focus on SGEMM (real single precision).

The Fortran API for SGEMM is:

```
SGEMM(transa, transb, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc)
```

It performs the computation:

`C := alpha * op(A) * op(B) + beta * op(C), where op(X) = X or` X^T
(depending on the value of the `transx` parameter).

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

The arrays A and B are inputs, while C is both input and output. Array A contains an `m`-by-`k` matrix, array B a `k`-by-`n` matrix, and array C an `m`-by-`n` matrix. The leading dimensions (`lda`, `ldb`, and `ldc`) determine the stride from one column to the next, allowing GEMM to work on portions of a larger matrix. Leading dimensions can also impact the performance by causing subsequent columns to be cache line-aligned or to map to the same set on the 8-way level-1 cache.

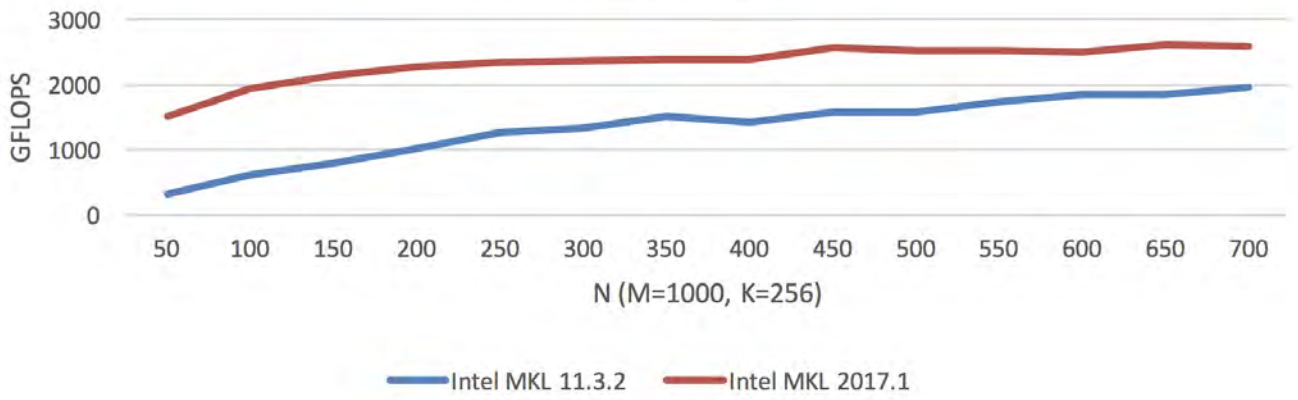
Intel® Math Kernel Library (Intel® MKL) offers high-performing GEMM implementations. The typical approach for optimizing matrix-matrix multiplication is to transform blocks of the original input matrices into an internal data format (such as a packed format), multiply transformed blocks via a handwritten assembly kernel, and then update the output matrix.[1] Block sizes are chosen to maximize cache and register usage. The reasons to pack are numerous:

- The ability to fit more data from A and B into the caches, which allows for bigger blocking and more data reuse
- Contiguous, aligned, and predictable accesses, which minimize cache and data translation lookaside buffer (DTLB) misses
- A reduction in loop overhead

For conventional sizes in high-performance computing, this packing-based approach works well. In general here, `m` and `n` are relatively large, while `k` may be moderate (outer product) or also relatively large (square), so that the amount of time spent packing the input matrices is small, relative to the time spent in the computational kernel. However, for sizes where one of `m` or `n` is relatively small, as is common for some machine learning applications, the packing overhead can become significant. As a result, a GEMM implementation that does not rely on explicit packing can outperform a conventional, packing-based GEMM implementation. Intel MKL 11.3 Update 3 includes {S,D}GEMM kernels that are optimized for some of these skewed sizes for **Intel® Advanced Vector Extensions 2 (Intel® AVX2)** and Intel® Advanced Vector Extensions 512 (Intel® AVX-512) and on the 2nd generation of the **Intel® Xeon Phi™** processor. Later versions of Intel MKL continue to improve these kernels.

To see an example of the benefit these new kernels can provide, **Figure 1** compares the performance of SGEMM in Intel MKL 2017 Update 1 with that of Intel MKL 11.3 Update 2 for sizes that may arise in machine learning. Here, `m` and `k` are fixed to 1000 and 256, respectively, while `n` varies. The performance is given in gigaflops (billions of floating-point operations per second), so higher is better.

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



Configuration Info – Versions: Intel® Math Kernel Library (Intel® MKL) 11.3.2 and Intel® MKL 2017.1; hardware: Intel® Xeon® processor E5-2699v4, two 22-core CPUs (55 MB LLC, 2.2 GHz), 64 GB of RAM; operating system: RHEL 7.2 GA x86_64.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. *Other names and brands may be claimed as the property of others. Benchmark Source: Intel Corporation

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.

1 Improved Intel® MKL SGEMM performance for skewed sizes

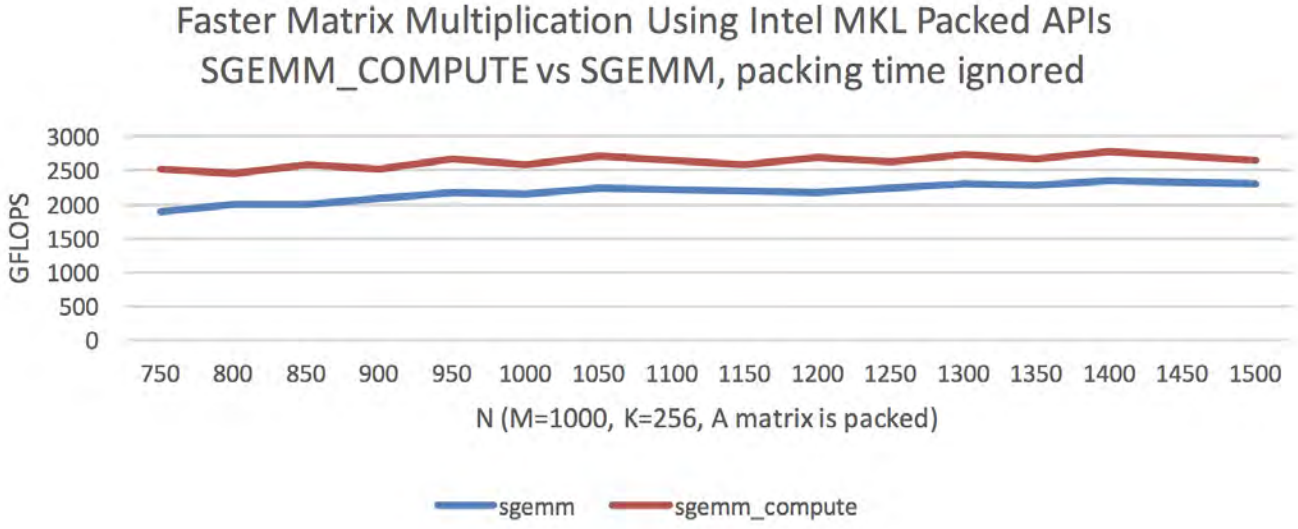
Based on the particulars of the SGEMM call (including processor type and thread count, problem size and leading dimensions, and transposition parameters), Intel MKL chooses to use either the conventional kernels or the new packing-free kernels. Deep learning frameworks that rely on Intel MKL for SGEMM performance benefit from these optimizations without requiring modification to the frameworks.

Another way to minimize the packing overhead arises when one or more of the input matrices (A or B) is reused in multiple matrix multiplications; for example, this can arise during recurrent neural networks. Here, we can pay the cost to pack the reused matrix once and then use the packed version in multiple SGEMM computations. Intel MKL 2017 introduced packed APIs for {S,D}GEMM that allow the packing overhead to be amortized over multiple matrix multiplications. For single precision, the four new packed APIs are:

```
dest = sgemm_alloc (identifier, m, n, k)

sgemm_pack (identifier, trans, m, n, k, alpha, src, ld, dest)

sgemm_compute (transa, transb, m, n, k, A, lda, B, ldb, beta, C, ldc)

sgemm_free (dest)
```

The parameters are similar to those of SGEMM, with the addition of the `identifier` character, which identifies which matrix (A or B) is to be packed. The `transa` and `transb` parameters of `sgemm_compute` can be set to "T" (transpose) or "N" (no transpose) as usual; additionally, a value of `P` indicates that the corresponding matrix is in the internal packed format. The packed APIs provide benefit if an input matrix is used multiple times; thus, `sgemm_alloc` and `sgemm_pack` would each be called once to allocate memory and pack the desired matrix in the internal packed format, respectively, followed by multiple calls to `sgemm_compute` where the packed matrix is passed as one of the input matrices. Finally, `sgemm_free` is called to release the memory. (For further details, please see the **Intel® Math Kernel Library Developer Reference**.)

Figure 2 shows a comparison of the performance of `sgemm` and `sgemm_compute` (where the A matrix is packed and the packing time is ignored) in gigaflops.



Configuration Info - Versions: Intel® Math Kernel Library (Intel® MKL) 2017.1; Hardware: Intel® Xeon® Processor E5-2699v4, 2 22-core CPUs (55MB LLC, 2.2GHz), 64GB of RAM; Operating System: RHEL 7.2 GA x86_64.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. *Other names and brands may be claimed as the property of others. Benchmark Source: Intel Corporation

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.

2 Faster matrix multiplication using Intel® MKL packed APIs

Applications that reuse the larger of the A and B input matrices between GEMM calls have the greatest potential to benefit from packed APIs. If both `m` and `n` parameters are large, or if the smaller of the two input matrices is reused, the performance improvements from switching to packed APIs may not be significant enough to justify the programming effort. Therefore, we recommend measuring the performance of the particular use case before modifying the application code to employ packed APIs.

As we have seen, the internal packing operation as found in a conventional GEMM implementation can have a noticeable overhead, particularly for sizes with one or more small dimensions. This overhead can be reduced by using kernels that avoid explicitly packing the input matrices or by amortizing the cost of packing over multiple GEMM computations. Starting from Intel MKL 11.3 Update 3, {S,D}GEMM can choose to use kernels that operate directly on the input matrices without first packing to internal buffers; which kernels are used is determined at runtime based on problem characteristics and processor information. Alternatively, Intel MKL 2017 introduced packed APIs that allow one or both of the input matrices to be explicitly packed and then reused in multiple matrix-matrix computations. These two approaches help to achieve high GEMM performance on multicore and many-core Intel® architectures, particularly for sizes arising from deep neural networks.

1. Kazushige Goto and Robert A. van de Geijn. 2008. “Anatomy of High-Performance Matrix Multiplication.” ACM Transactions on Mathematical Software, 34, 3, Article 12, May 2008.

TRY INTEL® MATH KERNEL LIBRARY (INTEL® MKL) TODAY >

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

Sign up for future issues | Share with a friend

IDENTIFY SCALABILITY PROBLEMS IN PARALLEL APPLICATIONS

How to Improve Scalability for Intel® Xeon and Xeon Phi™ Processors Using New Intel® VTune™ Amplifier Memory Analysis

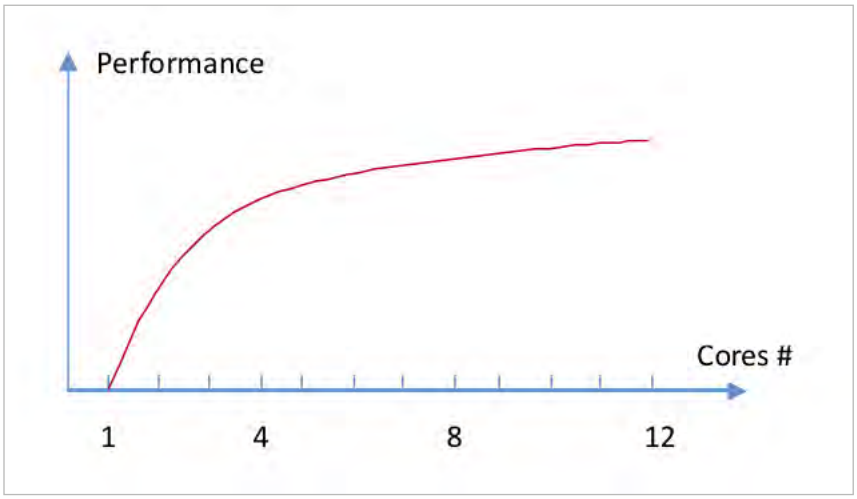
Vladimir Tsymbal, *Software Technical Consulting Engineer*, Intel Corporation

With the constantly increasing number of computing cores in modern systems, we expect well-parallelized software to increase performance—preferably linearly—with the number of cores. However, there are some factors limiting parallelism and scalability on multicore systems. We are not going to cover all of them in this article. But in most cases, the limitation is due to the implementation of parallelism:

- **Load imbalance** that leads to idle threads and CPU cores.
- **Excessive synchronization** and, as a result, wasted CPU time in spin-waiting and other nonproductive work.
- **Parallel runtime library overhead**, which might be due to misuse of the library API.

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

When those limiting factors are eliminated, parallel efficiency improves significantly—with all CPU cores busy doing useful work. Near-linear speedup is observed on well-tuned benchmarks like STREAM or LINPACK. However, with the increasing number of cores on your system (or as you run your code on a newer system with more cores), you might notice that the performance of your application is not increasing linearly—or that parallelism begins to plateau (**Figure 1**).



1 Performance changes according to the number of cores

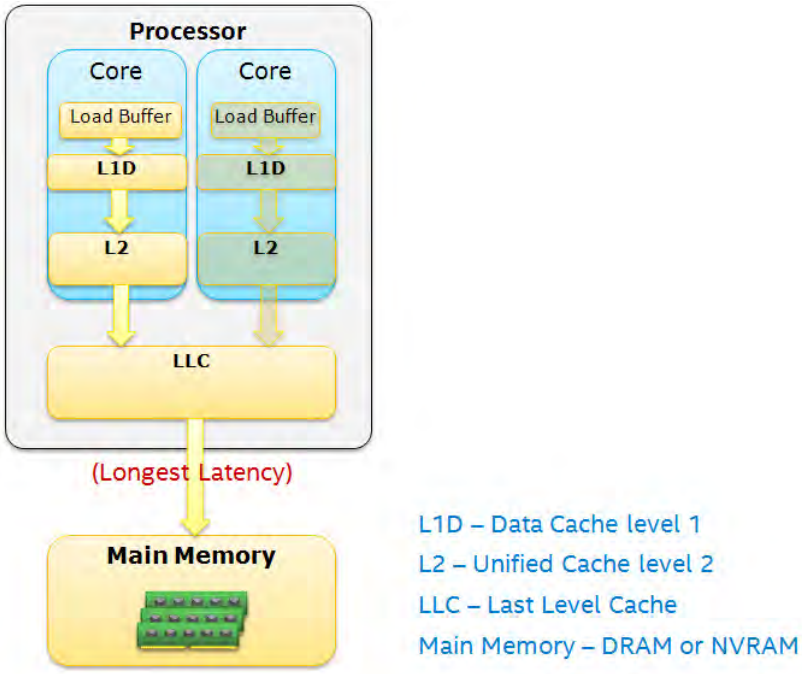
According to the top-down performance analysis approach,[1] you should first check if other components are limiting performance. Make sure that:

- **Your system is not constantly busy with something else that might consume resources,** such as other applications or services consuming compute time.
- **Your application is not bound to system I/O** (e.g., waiting for disk or other file system or network system operations to complete).
- **Your system has enough physical memory** to avoid frequent swapping to the hard disk drive.

As a common recommendation, you are expected to be aware if your hardware is configured properly and the memory subsystem provides expected performance characteristics. For example, you have all memory slots filled with DIMMs that correspond to the motherboard characteristics (e.g., number of channels, memory speed). You can easily check performance of your hardware with known benchmarks. It's important to do such a check, since it's easier to fix the problem with hardware than with software optimizations.

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

Once all these checks are done, look at memory latency as one of the main reasons for poor parallel scaling. In the x86 systems architecture, the CPU retrieves data from its cache subsystem. Ideally, data resides in the cache closest to the CPU (the L1 data cache) by the time it is needed by instructions (**Figure 2**). The farther requested data is from the CPU, the longer it takes to travel to the CPU core execution units. A CPU hardware prefetcher should help to bring data in faster, but it's not always possible. Often, data is delayed, which can stall the CPU.[2]



Basically, there are two reasons why data is late:

1. When data is requested by an instruction being executed in an EXE unit of the CPU, data bits make the long trip from the main memory or other caches to the CPU's L1D (i.e., the prefetcher didn't work). This creates a *memory latency* problem.
2. Data is requested in advance (i.e., the prefetcher did its work), but the bits got stuck in traffic on the way to the CPU because of the transport infrastructure capacity. This creates a *memory bandwidth* problem.

2 Retrieving data from the memory subsystem

Of course, there might be a combination of both problems if several requests are made from several sources. To avoid these problems, it's important to make smart usage of data. To solve the memory latency problem, ensure that data is accessed incrementally by its address. Sequential data access (or even unit stride, with a constant small distance) makes the prefetcher's life easier—and data access faster. To solve the memory bandwidth problem, reuse data and keep it hot in cache as much as possible. Either solution requires reconsidering data access patterns or even the whole algorithm implementation.

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

What's Limiting the Scalability of Your Application?

Once we have identified that our code execution is inefficient on a CPU, and we have observed that most stalls are memory bound, we need to define the specific memory problem because the solution is different depending on whether the problem is due to memory latency or bandwidth. We will use **Intel® VTune™ Amplifier's** embedded memory access analysis for a detailed investigation of memory problems.

Let's consider several iterations of improving a simplified matrix multiplication benchmark. In spite of its simplicity, it effectively demonstrates the possible memory problems that can occur depending on how the algorithm is implemented. For measurements, we will be using an **Intel® Xeon® processor E5-2697 v4** (code-named Broadwell, 36 cores) system with known theoretical parameters of memory bandwidth = 76.8 GB/s and double-precision (DP) floating point operations per second (FLOPS) = 662 GFLOPS/s.

Naïve Implementation of the Matrix Multiplication Algorithm

The naïve matrix multiplication implementation (multiply1, in **Figure 3**) will never scale linearly to a large number of CPU cores. Nevertheless, for educational purposes, it's a good example to illustrate how to identify causes of inefficient performance. The only improvement we would make is adding the –no-alias compiler option in order to allow **vectorization**. Otherwise, a scalar implementation would be roughly 10 times slower. The results of running the vectorized benchmark multiply1 on matrix size 9216 x 9216 can be found in **Table 1**. Note that the best performance is well below the theoretical maximum FLOPS.

```
void multiply2(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE c[][NUM], TYPE t[]
[NUM])
{
   int i,j,k;

// Loop interchange
  for(i=tidx; i<msize; i=i+numt) {
     for(k=0; k<msize; k++) {
#pragma ivdep
        for(j=0; j<msize; j++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }}}}
```

3 Optimized implementation of the matrix multiplication algorithm (multiply2)

No. Threads	Elapsed Time, Seconds	DP FLOPS, GFLOPS/Second
4	208	7.7
8	102	15.1
16	59	26.8
36	42	37.8
72 HT	24	66.1

Table 1. Performance and scaling of the naïve matrix multiplication (36 cores, Intel® Xeon® processor E5-2697 v4, two sockets @ 2300 MHz)

As **Table 1** shows, the parallel benchmark is scaling almost linearly with increasing numbers of threads. Scaling begins to plateau when more than 30 cores are involved. The data in **Table 1** might create a false confidence regarding the performance and scaling of the multiply1 benchmark. It's extremely important to understand how much your benchmark is using the compute power of a machine. In our case, the reported FLOPS (determined in the benchmark) is far from the theoretical number calculated for the machine earlier (approximately 10x smaller). The parallel scalability is not limited but the serial performance is. Note that Intel VTune Amplifier indicates the code execution within the loop is inefficient (**Figure 4**). The low *Retiring* and high *CPI* rates help estimate how far we are from practical limits.

S. Li.	Source	Cloc...	Inst... Ret...	CPI Rate	Front-... Bound	Bad Specu...	Back-... Bound	Retiring
66	void multiply1(int msize, int tidx, int numt, TYPE a[][NUM]							
67	{							
68	int i,j,k;							
69								
70	// Naive implementation							
71	for(i=tidx; i<msize; i=i+numt) {							
72	for(j=0; j<msize; j++) {	38,488...	35,...	1.077	1.1%	2.2%	76.6%	20.1%
73	for(k=0; k<msize; k++) {	71,300...	69,...	1.033	0.0%	0.0%	100.0%	14.7%
74	c[i][j] = c[i][j] + a[i][k] * b[k][j];	1,867,...	903...	2.067	0.7%	0.0%	87.1%	12.1%
75	}							
76	}							
77	}							
78	}							

4 Performance of the naïve, parallel matrix multiplication benchmark

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

Next, we'll look at an optimized implementation of the matrix multiplication algorithm (multiply2 in **Figure 3**). If the algorithm is simple enough, and if your compiler is smart enough, it will recognize the inefficient index strides and generate a version with interchanged loops automatically (or you can do that manually).

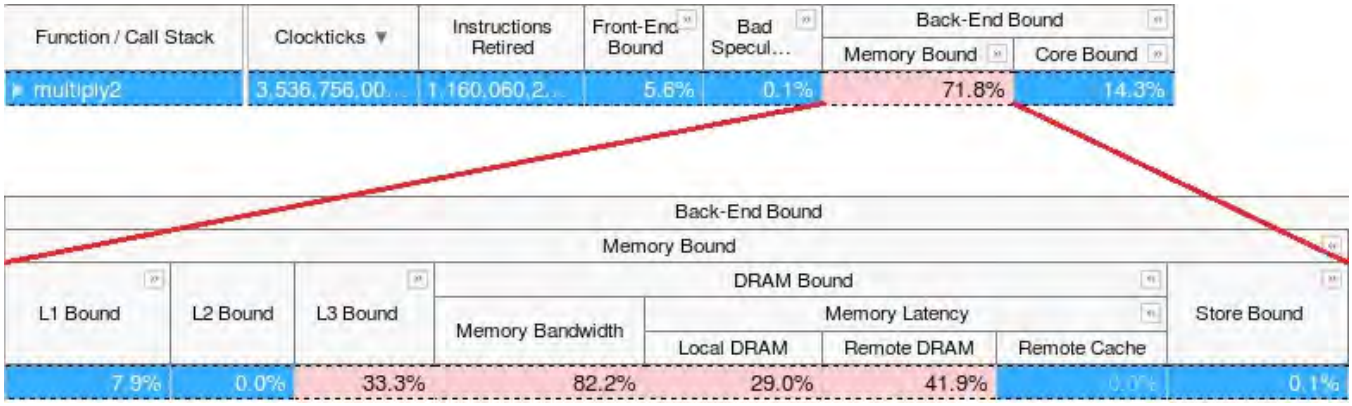
No. Threads	Elapsed Time, Seconds	DP FLOPS, GFLOPS/Second
4	208.8	7.8
8	103.3	15.1
16	58.8	26.4
36	38.4	40.5
72 HT	24.7	63.0

Table 2. Performance and scaling of the optimized matrix multiplication (36 cores, Intel® Xeon® processor E5-2697 v4, two sockets @ 2300 MHz)

As you might have noticed from **Table 2**, the absolute numbers are slightly better, but still far from ideal.

Let's try to understand what's limiting performance and scalability. The *General Exploration* profiling results (**Figure 5**) implement yet another top-down analysis approach, this time for CPU microarchitecture.[3] We might notice a couple of interesting things.

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

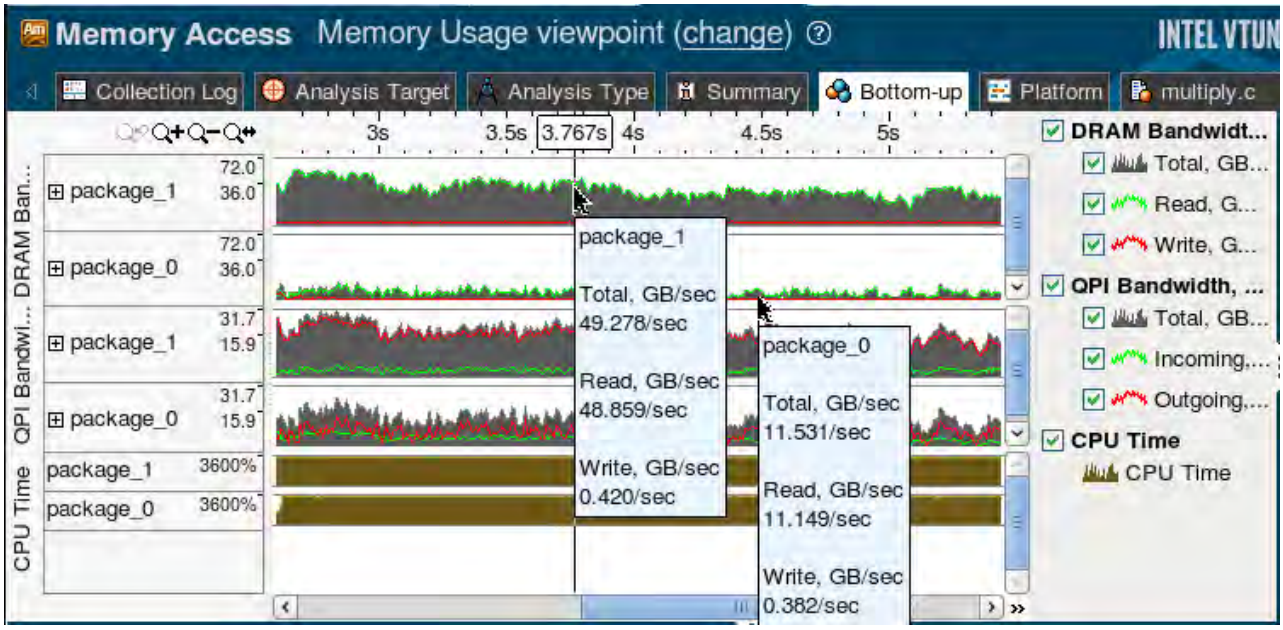


5 General Exploration profiling results

First, notice that the memory latency for bringing data from DRAM to CPU decreased. This is expected, since we implemented contiguous address access in the algorithm. But the memory bandwidth metric is very high. With that in mind, we should check the bandwidth numbers of the main data paths to make sure the DRAM controller and **Intel® QuickPath Interconnect (QPI)** are not bottlenecks. Second, notice the L3 latency is high as well, even though the data access has a contiguous pattern. This requires additional considerations. High L3 latency meant that we frequently have L2 misses, which is strange because the hardware L2 prefetcher should work (and does work, since the DRAM latency does not decrease with contiguous access). Third, the remote DRAM latency is significant. This indicates that there are nonuniform memory access (NUMA) effects, and some portion of data is fetched from remote DRAM for each node. So, to make the whole picture of data transfers clearer, we need to measure data traffic on the DRAM memory controller and the QPI bus between sockets. For that purpose, we use VTune memory access profiling.

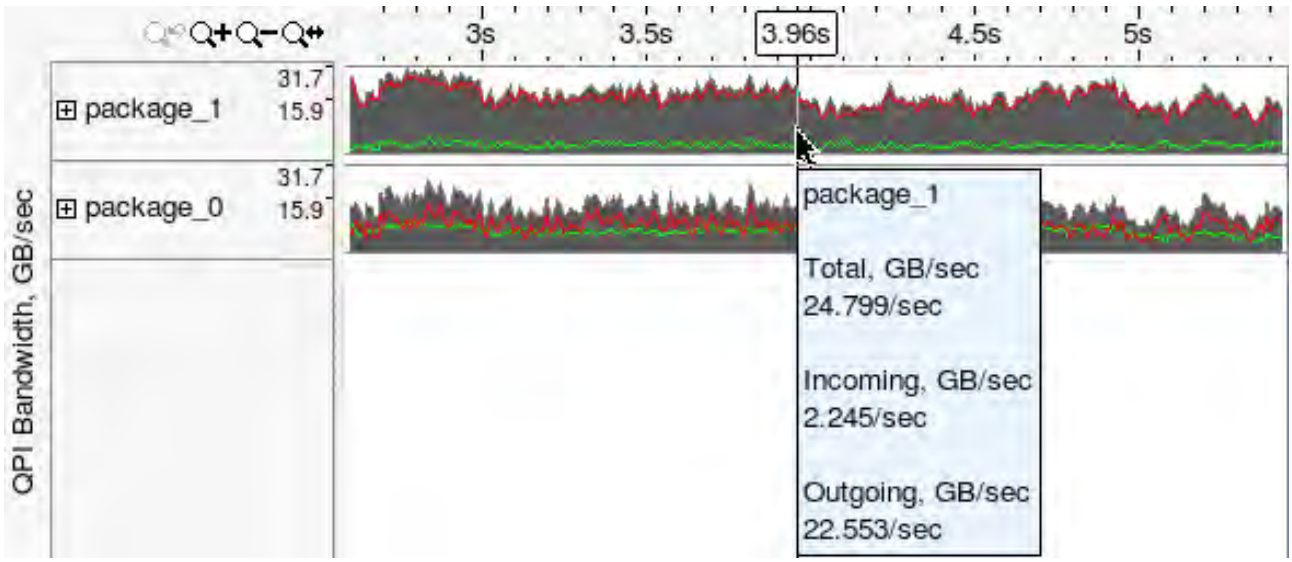
Figure 6 shows the profiling results for the example with 72 threads. Only one DRAM controller is loaded with data (package_1), and the average data rate is almost 50 GB/second, which is roughly two-thirds of the maximum bandwidth. On the memory controller of package_0, the traffic is negligible.

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



6 Collecting the Memory Access profile on multiply2 with 72 threads

In the same time period, we observe that half of the data traffic in the outgoing QPI lane formed package_1. This explains how the data gets from package_1 DRAM to the package_0 CPU cores (**Figure 7**). This cross-QPI traffic creates extra latency for data that is being fetched either from remote DRAM by the prefetcher or from remote LLC by a CPU core. Eliminating the NUMA effect might be easy for the benchmark, since the data is well structured and evenly distributed among threads. We just set thread affinities to the CPU cores and have each thread initialize the `a`, `b`, and `c` matrices. But we need to be careful in assuming that allocating data within each thread would eliminate all NUMA effects.



7 Cross-QPI traffic

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

Figure 8 shows an example that fails to improve performance under the previous assumption, and a way to detect the problem using Intel VTune Amplifier. In the benchmark source code, we introduce a function that represents threads pinned to enumerated CPUs. **Figure 8** shows a part of the code.

```
CreateThreadPool( … )
{
  pthread_t ht[NTHREADS];
  pthread_attr_t attr;
  cpu_set_t cpus;
  pthread_attr_init(&attr);

  for (tidx=0; tidx<NTHREADS; tidx++) {
       CPU_ZERO(&cpus);
       CPU_SET(tidx, &cpus);
       pthread_attr_setaffinity_np(&attr, sizeof(cpu_set_t), &cpus);
    pthread_create( &ht[tidx], &attr, (void*)start_routine, (void*) &par[tidx]);

  }
  for (tidx=0; tidx<NTHREADS; tidx++)
    pthread_join(ht[tidx], (void **)&status);
}
```

8 Threads pinned to enumerated CPUs

In a data initialization function, the arrays should be distributed between threads in the same way the arrays are multiplied in the multiplication function. **Figure 9** shows the modification done in the functions to simplify NUMA awareness. In the initialization function, the data array is divided by chunks of size `msize/numt`, which is the size of the matrix divided by the number of threads. The same is done in the multiplication function shown in **Figure 10**. Surprisingly, the runtime for the benchmark is not much better than the NUMA-unaware version, so let's analyze with an VTune memory access profile (**Figure 11**).

```
InitMatrixArrays (int msize, int tidx, int numt,  … )
{
    int i,j,k,ibeg,ibound,istep;
    istep = msize / numt;
    ibeg = tidx * istep;
    ibound = ibeg + istep;

    for(i=ibeg; i<ibound; i++) {
      for (j=0; j<msize;j++) {
            a[i][j] = 1.0*i+2.0*j+3.0;
            b[i][j] = 2.0*i+1.0*j+3.0;
            c[i][j] = 0.0;
        }
    }
}
```

9 Simplifying NUMA awareness

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

```
multiply2(int msize, int tidx, int numt,  … )
{
    int i,j,k,ibeg,ibound,istep;
    istep = msize / numt;
    ibeg = tidx * istep;
    ibound = ibeg + istep;

        for(i=ibeg; i<ibound; i++) {
                for(k=0; k<msize; k++) {
                        for(j=0; j<msize; j++) {
                                c[i][j] = c[i][j] + a[i][k] * b[k][j];
}}}}
```

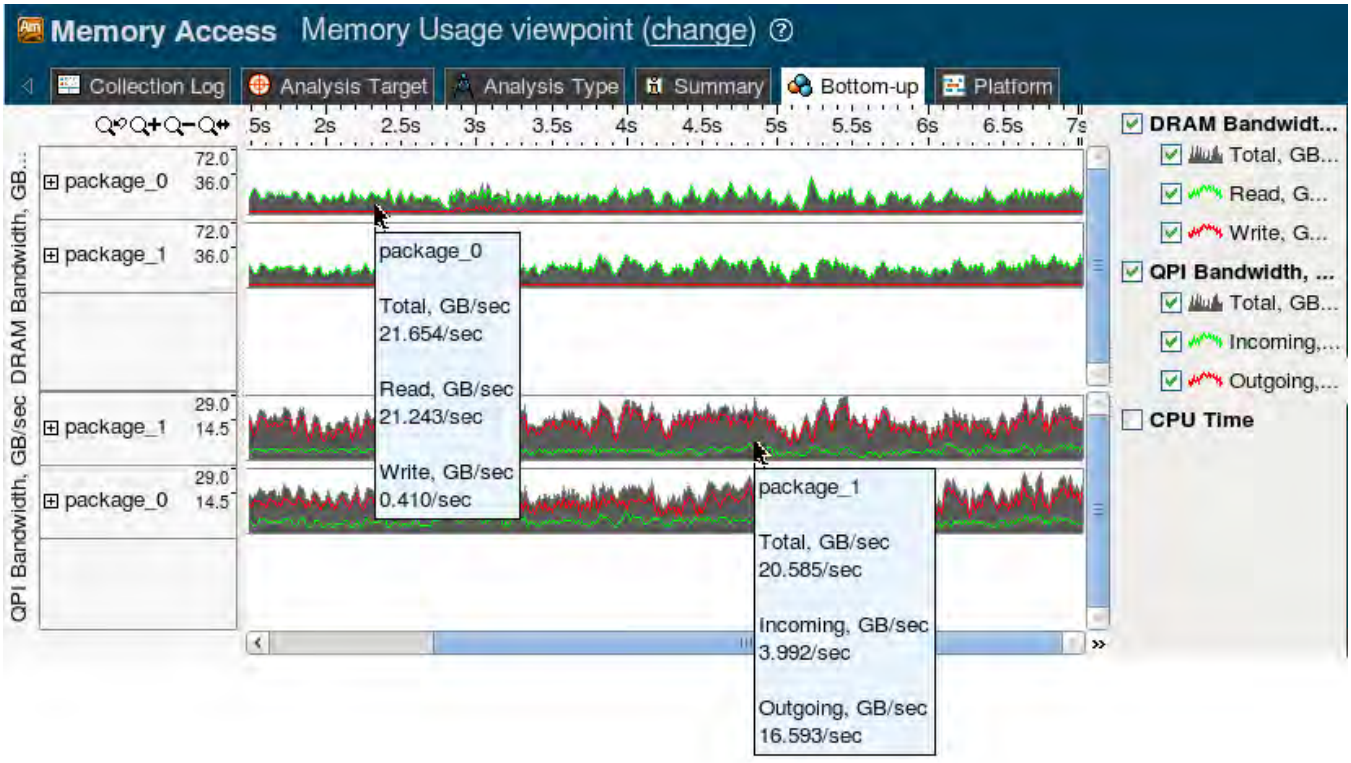
```
Threads #: 72 Pthreads
Matrix size: 9216
Using multiply kernel: multiply2
Freq = 2.30100 GHz
Execution time = 20.162 seconds
MFLOPS: 72826.877 mflops
```

10 Multiplication function

11 Memory access profiling

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

The summary page notifies us that the application is still memory bound (with stalls due to data latencies from memory and data traffic), but the latencies are mostly caused by LLC and less by DRAM. Also, the ratio between local and remote access is very high, which means that the NUMA awareness approach didn't work. If we check the timeline for traffic over the DRAM controller and QPI (**Figure 12**), we see that the data stream from DRAM is hardly reaching 30 percent of peak bandwidth, but the QPI is saturated at approximately 90 percent of its capacity in each direction (the practical limit for QPI is 29.2 GB/s for this system).



12 Checking the timeline for traffic over the DRAM controller and QPI

Remote access (whether DRAM or LLC) is increasing latency for reading memory blocks and making the CPU stall. Those latencies can be measured by Intel VTune Amplifier's memory access, which allows us to identify which data (matrix) is still being accessed in an inefficient, remote way. If we examine the memory analysis summary (**Figure 13**), we can observe which memory objects created most of the latency.

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

Top Memory Objects by Latency

This section lists memory objects that introduced the highest latency to the overall application execution.

Memory Object	Total Latency	Loads	Stores	LLC Miss Count
_mm_malloc (648 MB)	63.5%	112,553,776,512	25,600,384	339,210,176
_mm_malloc (648 MB)	36.2%	129,952,698,464	118,683,380,224	8,000,240
[Unknown]	0.2%	1,149,634,488	606,409,096	5,600,168
[Stack]	0.0%	120,803,624	201,603,024	0
_mm_malloc (648 MB)	0.0%	276,808,304	88,001,320	0
[Others]	0.2%	47,201,416	0	0

13 Top memory objects by latency

Among the three top memory objects (represented by their allocation function), we notice that one clearly represents the biggest portion of latencies and is responsible for a large number of load operations (**Figure 14**). Note that only one object has an average latency high enough to conclude that the data is from the remote DRAM of LLC. We can confirm this conclusion by the numbers in the Remote DRAM Access columns.

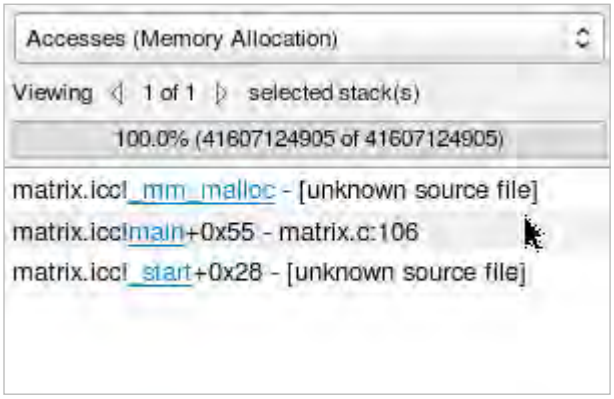
Grouping: Memory Object / Function / Allocation Stack

Memory Object / Function / Allocation Stack	Loads	Stores	LLC Miss Count	
			Local DRAM Acces...	Remote DRAM Access Count
▶ [Unknown]	1,149,634,488	606,409,096	3,200,096	3,200,096
▶ _mm_malloc (648 MB)	129,952,698,464	118,683,380,224	8,800,264	0
▶ _mm_malloc (648 MB)	112,553,776,512	25,600,384	117,603,528	136,804,104
▶ _mm_malloc (648 MB)	276,808,304	88,001,320	0	0

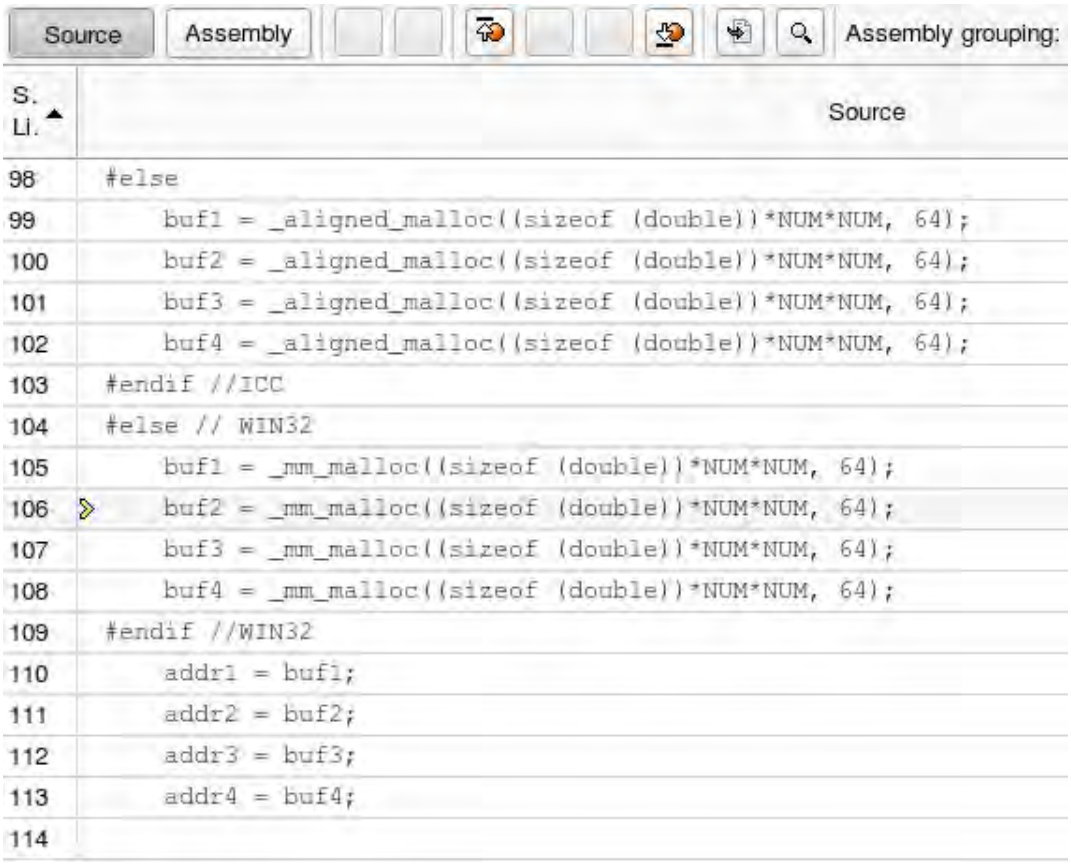
14 Memory objects by allocation function

It's easy to figure out that those three objects are the three a, b, and c matrices. The one with high Stores is matrix c. To identify which matrix data is creating high latencies, you need to check the stack for the memory object in the Intel VTune Amplifier stack pane (**Figure 15**). Going by the stack in the user code, we can drill down to the source line of data allocation presented in the Intel VTune Amplifier Source View (**Figure 16**). In this case, it's a matrix b data that creates latency chatter and an increased number of loads. Now we need to understand why it's happening despite the fact that the data arrays were allocated and initialized within pinned threads.

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

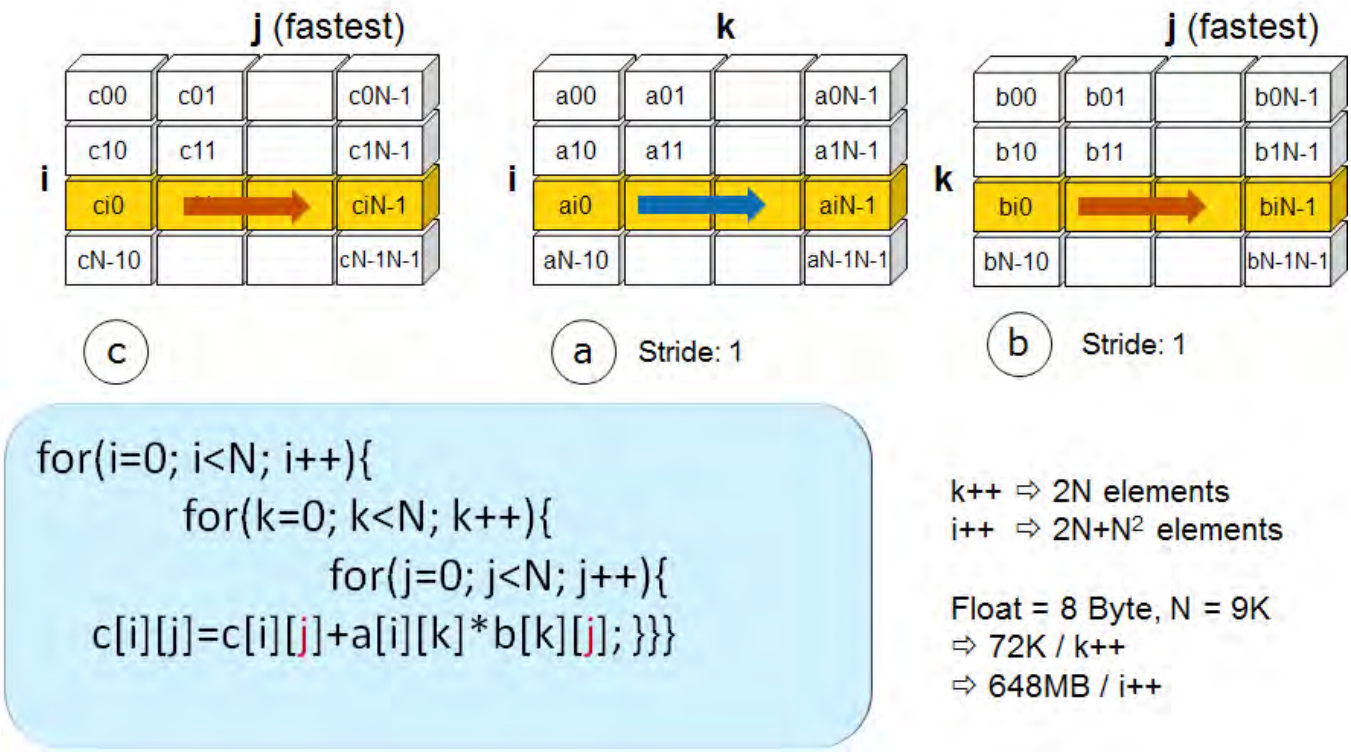


15 Memory objects by stack pane



16 Intel® VTune™ Amplifier source view

A quick investigation of the algorithm with transposed matrices reveals a fundamental inefficiency in the data access pattern (**Figure 17**). For each matrix a row, the whole matrix b has to be read entirely from memory.

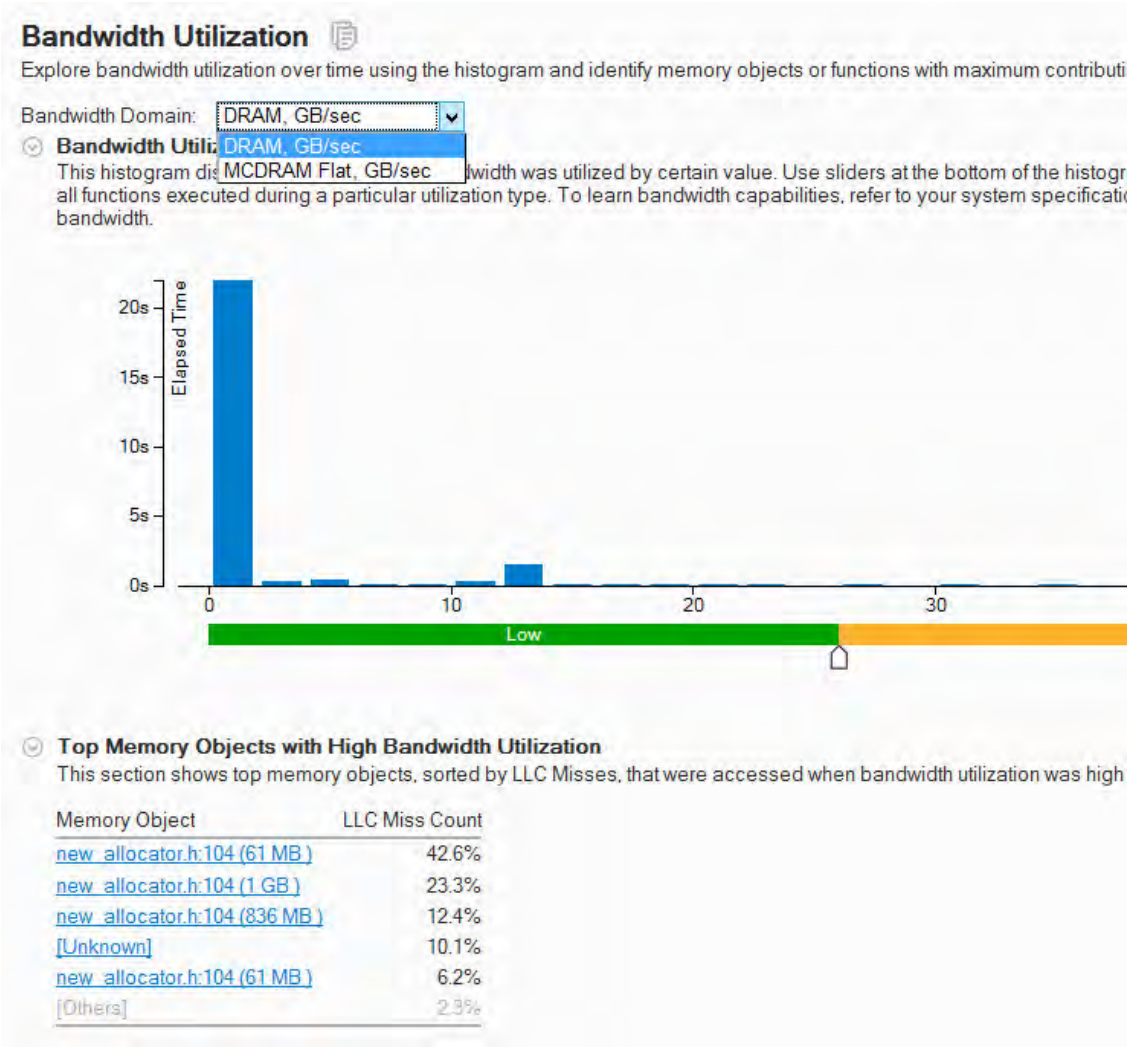


17 Algorithm with transposed matrix

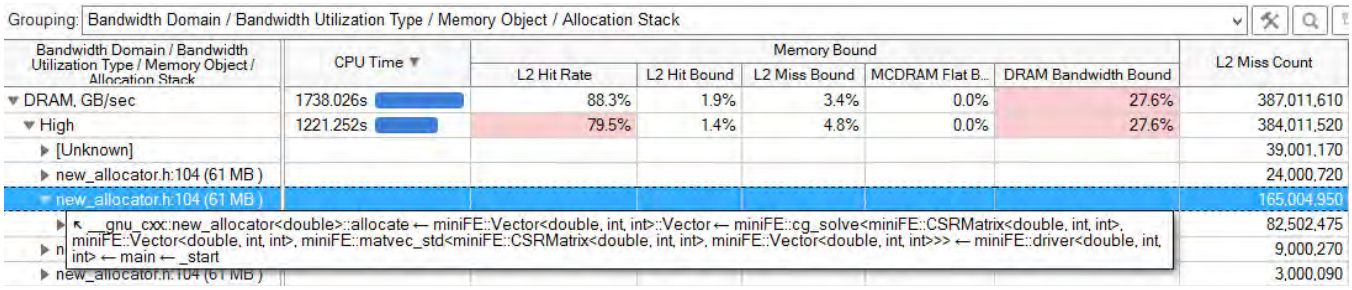
The matrices include about 9K elements in a column/row. So, the whole matrix memory block size will exceed any CPU cache capacity, generating constant cache data eviction and reloads from DRAM. Even though the distributed rows of matrices c and a are accessed by threads on the CPU cores on which they were allocated, it doesn't completely apply to matrix b. Half of the matrix b data will be read by threads from a remote socket in this implementation of the algorithm. Even worse, reading the whole matrix b for each row of matrix a creates a redundant data load operation (N times more than needed) and generates excessive traffic on QPI for accessing remote data.

Similarly, you can define which data objects were contributing to increased traffic for DRAM or MCDRAM on Intel Xeon Phi processor-based systems. You just need to select which memory domain traffic you want to analyze. You can get the objects' reference and allocating stack information (**Figure 18**) and, when grouped by bandwidth Domain and bandwidth Utilization Type, you can observe the objects and identify those that contribute most to the L2 Miss Count (**Figure 19**).

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



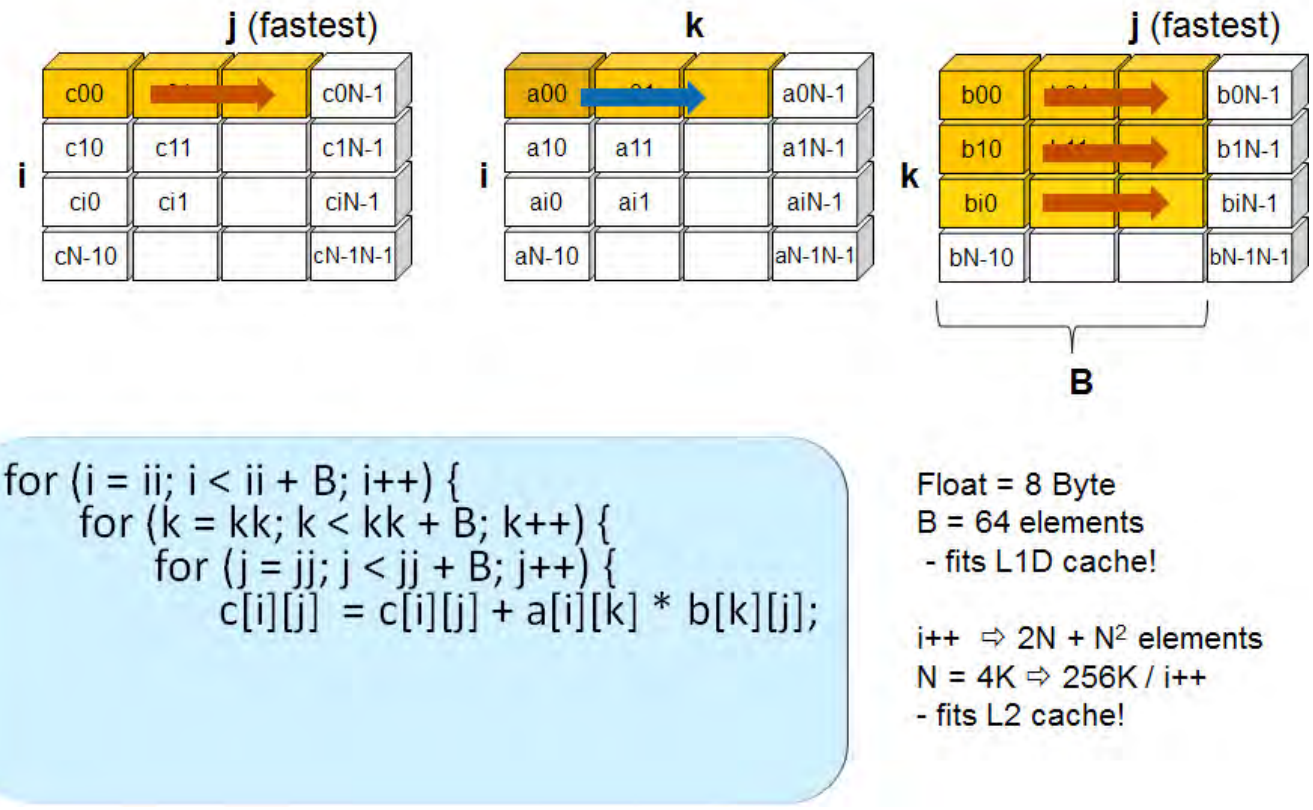
18 Analyzing memory domain traffic



19 Bandwidth domain

Data Blocking

We can decrease the data latencies for eliminating CPU stalls by yet another modification of the multiplication algorithm. We want all data in the three matrixes being accessed by threads running on a local socket. One of the well-known and frequently used modifications is data blocking (**Figure 20**). It allows working with smaller blocks of arrays from each of the matrices, keeping them hot in caches and reused by CPU (which, in turn, gives opportunities for further performance improvements through optimizing the blocks for CPU cache sizes). Also, this makes it easier to distribute the blocks among threads and prevent massive remote accesses and reloads.



20 Matrix data blocking

If we look at the results of the cache blocking modification (**Figure 21**), we can observe that even without fixing NUMA effects, memory latencies are much smaller and execution is much faster.

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

```
./matrix.icc
Threads #: 72 Pthreads
Matrix size: 9216
Using multiply kernel: multiply3
Freq = 2.3 GHz
Execution time = 12.08 seconds
MFLOPS: 128710.367 mflops
```

21 Cache blocking modification (multiply3)

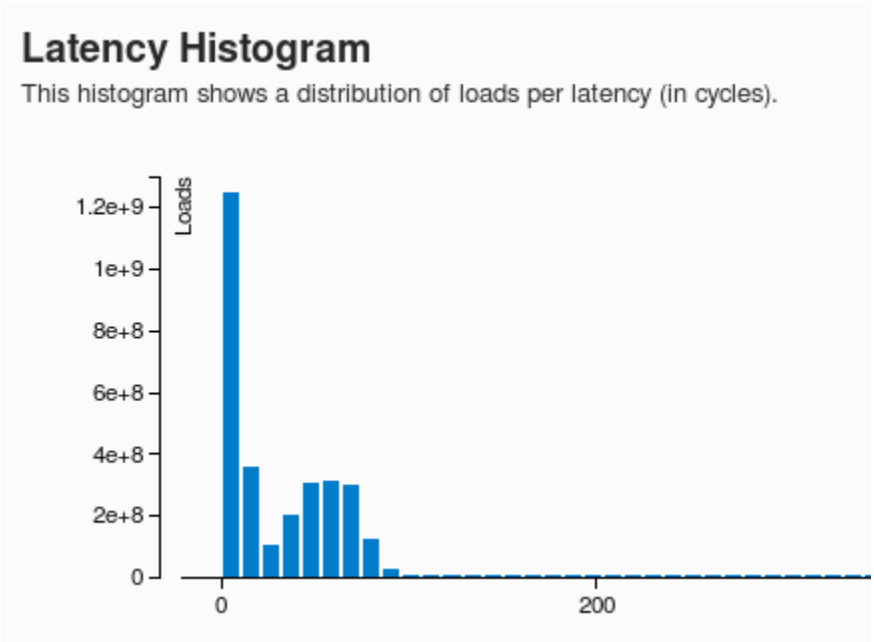
According to the *General Exploration* profile (**Figure 22**), the Retiring pipeline slots increased up to 20 percent, while the rest of CPU stalls are shared between Memory Bound and Core Bound execution.

Clockticks:	2,268,184,100,000	
Instructions Retired:	1,851,470,100,000	
CPI Rate:	1.225	
MUX Reliability:	0.994	
Front-End Bound:	**9.4%**	**of Pipeline Slots**
Bad Speculation:	**0.0%**	**of Pipeline Slots**
Back-End Bound:	**70.9%**	**of Pipeline Slots**
Memory Bound:	**45.6%**	**of Pipeline Slots**
L1 Bound:	**0.7%**	**of Clockticks**
L2 Bound:	5.6%	of Clockticks
L3 Bound:	**31.9%**	**of Clockticks**
Contested Accesses:	0.0%	of Clockticks
Data Sharing:	0.0%	of Clockticks
L3 Latency:	100.0%	of Clockticks
SQ Full:	0.0%	of Clockticks
DRAM Bound:	**3.0%**	**of Clockticks**
Memory Bandwidth:	26.9%	of Clockticks
Memory Latency:	**69.5%**	**of Clockticks**
Local DRAM:	36.6%	of Clockticks
Remote DRAM:	6.5%	of Clockticks
Remote Cache:	0.0%	of Clockticks
Store Bound:	**1.2%**	**of Clockticks**
Core Bound:	**25.3%**	**of Pipeline Slots**
Retiring:	**19.6%**	**of Pipeline Slots**
Total Thread Count:	73	
Paused Time:	0s	

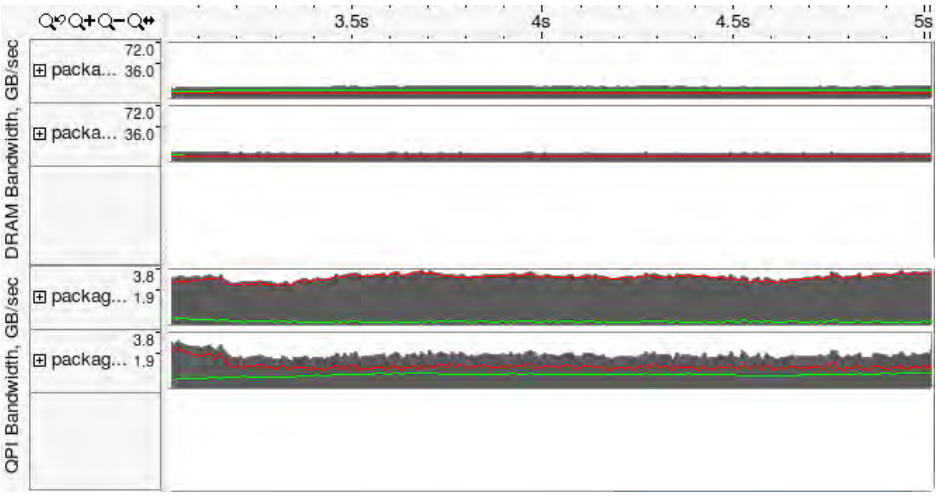
22 General Exploration profile (multiply3)

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

According to the Latency Histogram (**Figure 23**), most of the latencies are concentrated around L2 access values, while the rest are in the zone of 50 to 100 cycles, which is in the area of LLC Hit latency numbers. The Bandwidth timeline diagram (**Figure 24**) shows that most of data is taken from a local DRAM, and traffic on QPI is slightly increased. This is still a smaller performance than the **Intel® Math Kernel Library (Intel® MKL)** implementation of the double-precision matrix multiplication (dgemm), but closer to it for this size of matrix (**Figure 25**). So, the final optimization that we could do is modifying the algorithm to be blocked and fully NUMA aware. The final performance is shown in **Table 3** and **Figure 26**.



23 Latency histogram (multiply3)



24 Bandwidth timeline diagram (multiply3)

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

```
$./matrix.mkl
Threads #: 72 requested OpenMP threads
Matrix size: 9216
Using multiply kernel: multiply5
Freq = 2.799980 GHz
Execution time = 2.897 seconds
MFLOPS: 540032.936 mflops
```

25 Performance measurement for Intel® MKL-based multiply5

No. Threads	Elapsed Time, Seconds	DP FLOPS, GFLOPS/Second
4	104.8	14
8	60.1	25
16	31.3	49
36	17.85	87
72 HT	12.08	128

Table 3. Performance of the final matrix3 optimization (36 cores, Intel® Xeon® processor E5-2697 v4, two sockets @ 2300 MHz)

BLOG HIGHLIGHTS

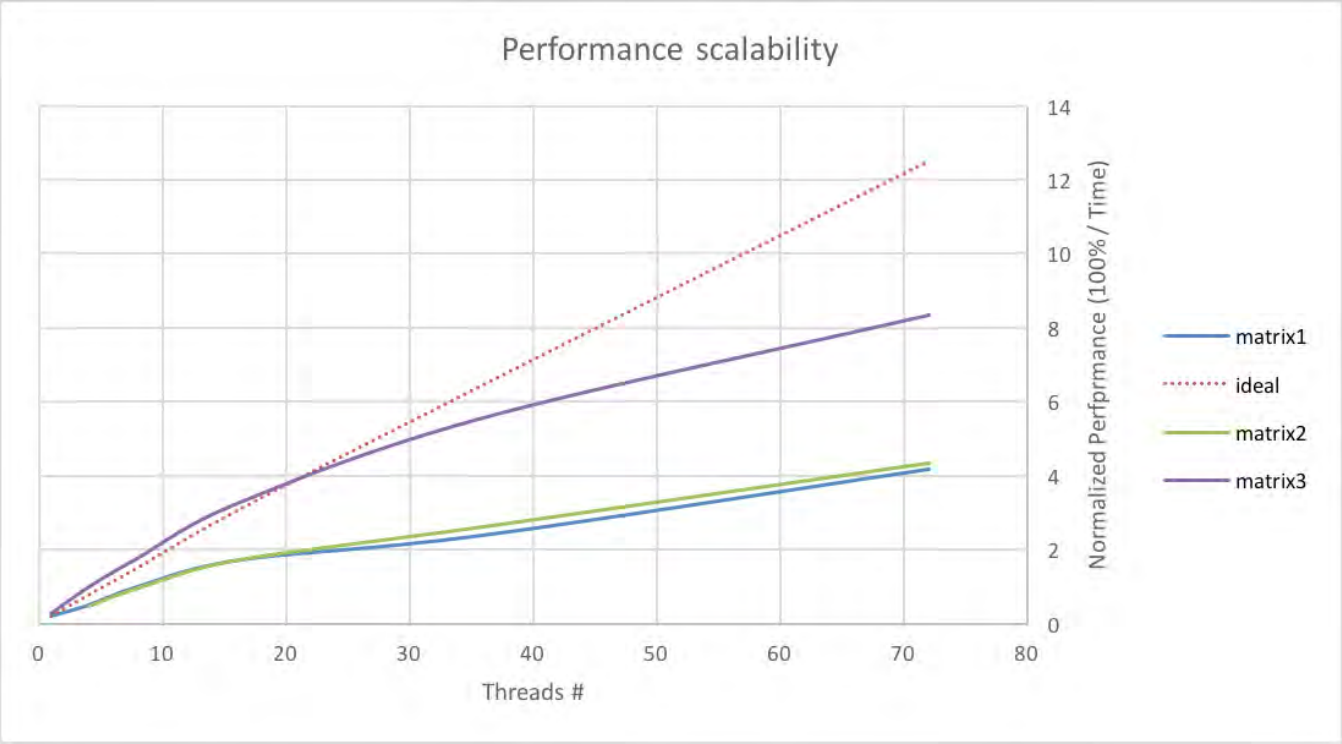
How to Get Started as a Developer in AI

BY NIVEN SINGH (INTEL) ›

The promise of artificial intelligence has captured our cultural imagination since at least the 1950s—inspiring computer scientists to create new and increasingly complex technologies, while also building excitement about the future among regular everyday consumers. What if we could explore the bottom of the ocean without taking any physical risks? Or ride around in driverless cars on intelligent roadways? While our understanding of AI—and what's possible—has changed over the past few decades, we have reason to believe that the age of artificial intelligence may finally be here. So, as a developer, what can you do to get started? This article will go over some basics of AI, and outline some tools and resources that may help.

Read more ›

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



26 Matrix multiplication benchmark results

Note on the scalability graph:

- The matrix3 line goes beyond the ideal line due to cache blocking effects, which make the single-threaded version execute faster than a naïve implementation.
- Until the number of threads is equal to the number of physical cores, the matrix3 line goes closer to the ideal line, while adding **hyper-threading** does not improve scaling.

Conclusion

Some memory access patterns might appear to be causing poor scalability in parallel applications due to CPU microarchitecture constraints. To avoid the constraints, you need to identify exactly which data arrays cause the CPU to stall while waiting for data. With Intel VTune Amplifier memory access profiling, you can identify the data objects that cause the biggest delays, as well as the amount of this delay measured in CPU clock ticks, the level of the cache subsystem in which the data reside, and the source code for data object allocation and delayed access. This information should help you reconsider your algorithm to deliver better memory access patterns.

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

References

1. Charlie Hewett. **Top Down Methodology for Software Performance Analysis**.

2. **Intel® 64 and IA-32 Architectures Optimization Reference Manual**.

3. Ahmad Yasin. *"A Top-Down Method for Performance Analysis and Counters Architecture."* IEEE Xplore: 26 June 2014. Electronic ISBN: 978-1-4799-3606-9.

TRY INTEL® VTUNE™ AMPLIFIER TODAY >

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

VECTORIZATION OPPORTUNITIES FOR IMPROVED PERFORMANCE WITH INTEL® AVX-512

Examples of How Intel® Compilers Can Vectorize and Speed Up Loops

Martyn Corden, *Software Technical Consulting Engineer,* Intel Corporation

Editor's note: **The last issue** *of* The Parallel Universe *touched on Intel® Parallel Studio XE 2017 support for enhanced vectorization. In this issue, Martyn Corden dives deeper into Intel® AVX-512 instructions to show developers how to expose vectorization that was not previously possible.*

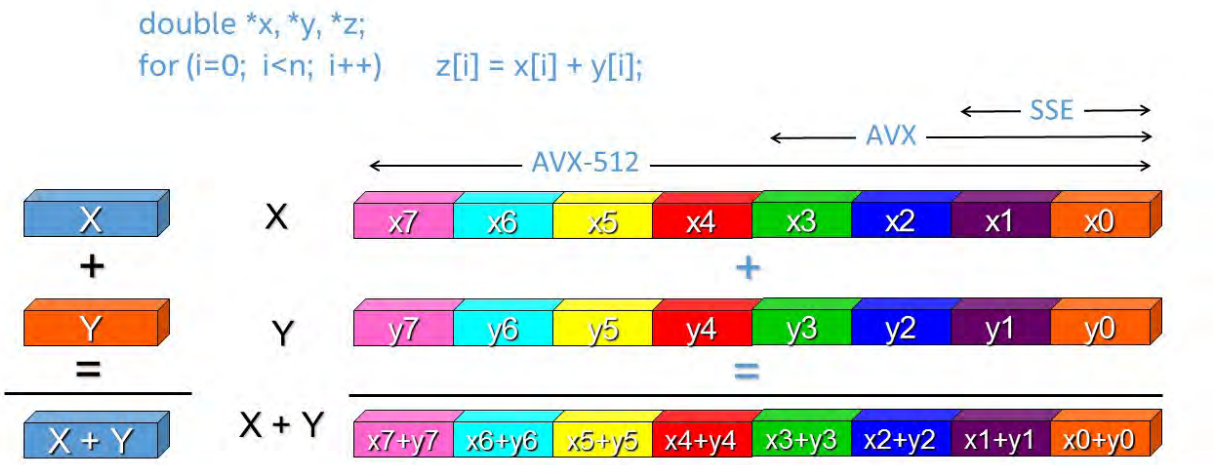
The days of easy performance improvements from steadily increasing clock speeds are long gone. Moore's Law instead provides increased parallelism through additional cores and more and wider SIMD registers. Besides the increased SIMD vector width of 512 bits, **Intel® Advanced Vector Extensions 512 (Intel® AVX-512)** includes new instructions that enable **vectorization** of some loops that could not be vectorized with older instruction sets and more efficient vectorization of others.

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

In this article, we'll explain, with examples, how **Intel® C/C++ and Fortran compilers** can vectorize and speed up loops that compress or expand arrays and loops that fill histograms or perform scatters with potential address conflicts. We will also show how the compiler can convert strided loads or gathers into more efficient unit stride SIMD loads for certain types of loops over arrays of structures. Finally, we'll show how to recognize this conversion using new features of the optimization report in the Intel® Compiler version 17.0 from **Intel® Parallel Studio XE 2017**. These optimizations can benefit a broad range of applications running not only on the **Intel® Xeon Phi™** X200 processor, but also on future Intel® Xeon® processors that support the Intel AVX-512 instruction set.

A Vectorization Refresher

Figure 1 illustrates a simple, double-precision floating-point loop. In scalar mode, one instruction produces one result. After vectorization, a single Intel AVX-512 instruction can produce eight results, compared to four for Intel® AVX, or two for Intel® Streaming SIMD Extensions (Intel® SSE). The Intel Xeon Phi x200 processor supports Intel AVX-512 instructions for a wide variety of operations on 32- and 64-bit integer and floating-point data. This may be extended to 8- and 16-bit integers in future Intel Xeon processors.



1 Scalar and vectorized loop versions with Intel® SSE, AVX and AVX-512

Automatic vectorization is enabled by default in the Intel® compiler. However, one of the switches in **Figure 2** must be set in order to target the Intel AVX-512 instructions.

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

Linux* and OS X*	Windows*	Functionality
-xmic-avx512	/Qxmic-avx2	Intel® Xeon Phi™ x200 processor family only
-xcore-avx512	/Qxcore-avx512	Future Intel® Xeon® processor only
-xcommon-avx512	/Qxcommon-avx512	Intel® AVX-512 subset common to both. Not a fat binary.
-axmic-avx512	/Qaxmic-avx512	Fat binary. Targets Intel® Xeon Phi™ x200 and also other Intel® Xeon® processors
-xhost	/Qxhost	Targets the compilation host

Table 1. Compiler switches to enable Intel® AVX-512 instructions

Compress and Expand Loops

The Fortran example in **Figure 2A** illustrates array compression. Only those elements of a large source array that satisfy a condition are copied into a smaller target array. The C example in **Figure 2B** illustrates the inverse operation (i.e., array expansion), where elements of a smaller source array are copied back into the larger, sparse array.

```
nb = 0
do ia=1, na                ! line 23
  if (a(ia) > 0.) then
    nb = nb + 1          ! dependency
    b(nb) = a(ia)         ! compress
  endif
enddo
```

2A Array compression

```
int j = 0
for (int i=0; i <N; i++) {
    if (a[i] > 0) {
        c[i] = a[k++];   // expand
    }
}
// Cross-iteration dependencies via j and k
```

2B Array expansion

The conditional increment of the dense array index introduces a dependence between loop iterations. In the past, this would have prevented automatic vectorization. For example, when compiling the loop in **Figure 2A** targeting Intel® AVX2, the optimization report shows:

```
ifort -c -xcore-avx2 -qopt-report-file=stderr -qopt-report=3 compress.f90
…
LOOP BEGIN at compress.f90(23,3)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization.  …
  remark #15346: vector dependence: assumed ANTI dependence between nb (25:7) and nb (25:7)
LOOP END
```

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

The C example in **Figure 2B** behaves in the same way. An OpenMP* SIMD directive cannot be used either, because the dependency would lead to incorrect results.

Intel AVX-512 overcomes this dependency with the new vcompress instructions that write selected elements of one SIMD register into contiguous elements of another, or to memory. Likewise, the vexpand instructions write contiguous elements from a source register or memory to selected (sparse) elements of a destination SIMD register. These new instructions allow the compiler to vectorize the compression example (**Figure 2A**) when Intel AVX-512 is enabled:

```
ifort -c -xmic-avx512 -qopt-report-file=stderr -qopt-report=3 compress.f90
…
LOOP BEGIN at compress.f90(23,3)
   remark #15300: LOOP WAS VECTORIZED
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15457: masked unaligned unit stride stores: 1
…
   remark #15497: vector compress: 1
LOOP END
```

3 Array compression when Intel(r) AVX-512 is enabled

All elements of the source array are loaded, so the load is unmasked. Only selected elements are stored, so the effective store is masked. The optimization report contains remark #15497 to show that a compress idiom was recognized and vectorized. An assembly listing, obtained with the -S option, shows instructions such as:

```
vcompressps %zmm4, -4(%rsi,%rdx,4){%k1}
```

Similar results are obtained for the array expansion loop in **Figure 2B**.

Compressing a single-precision array of 1,000,000 random values by a factor of two, repeated 1,000 times on an Intel Xeon Phi 7250 processor, resulted in an approximately 16x speedup of Intel AVX-512 over Intel AVX2, which corresponds to the width of the SIMD registers and instructions.

Intel AVX-512 Conflict Detection Instructions

Loops containing a store with indirect addressing have a potential dependency that typically prevents vectorization. For example:

```
for (i=0; i<n; i++)  a[index[i]] = …
```

If `index[i]` has the same value for two different values of *`i`*, the stores conflict and cannot be executed safely at the same time. The vpconflict instruction from Intel AVX-512 resolves this conflict by providing a mask for those SIMD lanes (values of `i`) that are conflict-free (i.e., no values of `index[i]` are duplicated). After the SIMD computation has been safely performed for these lanes, the loop is re-executed for any lanes that were masked out.

Histogramming

Filling a histogram is a common operation in many applications (e.g., image processing). The code fragment in **Figure 4** fills a histogram of `sin(x)` in the array `h` for an array `x` of input values. With Intel AVX2, this does not vectorize because of a dependency, since two input values could contribute to the same histogram bin, *`ih`*:

```
for (i=0; i<n; i++) {
      y     = sinf(x[i]*twopi);
      ih    = floor((y-bot)*invbinw);
      ih    = ih > 0     ? ih : 0;
      ih    = ih < nbin-1 ? ih : nbin-1;
      h[ih] = h[ih] + 1;                      //  line 25
   }
```

4 Loop to fill a histogram of sin(x)

```
icc -c -xcore-avx2 histo.c -qopt-report-file=stderr -qopt-report-phase=vec
…
LOOP BEGIN at histo2.c(20,4)
      remark #15344: loop was not vectorized: vector dependence prevents vectorization…
      remark #15346: vector dependence: assumed FLOW dependence between h[ih] (25:7) and h[ih] (25:7)
LOOP END
```

Intel AVX-512 conflict detection instructions allow this loop to be vectorized safely:

```
ifort -c -xmic-avx512 histo2.f90 -qopt-report-file=stderr -qopt-report=3 -S
…
LOOP BEGIN at histo2.c(20,4)
   remark #15300: LOOP WAS VECTORIZED
   remark #15458: masked indexed (or gather) loads: 1
   remark #15459: masked indexed (or scatter) stores: 1
   remark #15499: histogram: 2
LOOP END
```

In the assembly code (not shown), `x` and `index` are loaded, and `ih` is calculated for all SIMD lanes. This is followed by a `vpconflict` instruction and a masked gather of the unique bins (i.e., elements of `h`) into a register, which are then incremented. If some values of `ih` were duplicated, the code loops back and reincrements the corresponding bins. Finally, there is a masked store (scatter) of the incremented bins back into `h`.

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

A simple test that filled a histogram of 200 bins from a single-precision array of 100,000,000 random values between 0 and 1, repeated 10 times, was compiled first for Intel AVX2 and then for Intel AVX-512 and run on an Intel Xeon Phi 7250 processor. A nearly 9x speedup was observed. The speedup depends heavily on the problem details. It comes mostly from vectorization of the loop computations, such as the sine function in this example, not from the gather and scatter themselves. The speedup can be large in the common case where conflicts are relatively infrequent. However, it may be smaller if there are many conflicts, as in a histogram containing a singularity or narrow spike.

Gather-to-Shuffle Optimization

Large arrays of small structures or of short vectors occur frequently but present a challenge to efficient vectorization. Vectorization over the structure content or short vectors may be impossible or inefficient due to the low trip count. Vectorization over the large array index is also inefficient because data used by consecutive iterations are not adjacent in memory, so efficient SIMD loads of contiguous data cannot be used. This is illustrated in **Figure 5**, which simply calculates the sum of squares of the components of an array of 3-vectors.

```
struct Point { float x; float y; float z; };

float sumsq( struct Point *ptvec, int n) {
    float   t_sum = 0;

    for (int i=0; i<n; i++) {
        t_sum += ptvec[i].x * ptvec[i].x;
        t_sum += ptvec[i].y * ptvec[i].y;
        t_sum += ptvec[i].z * ptvec[i].z;
    }
    return  t_sum;
}
```

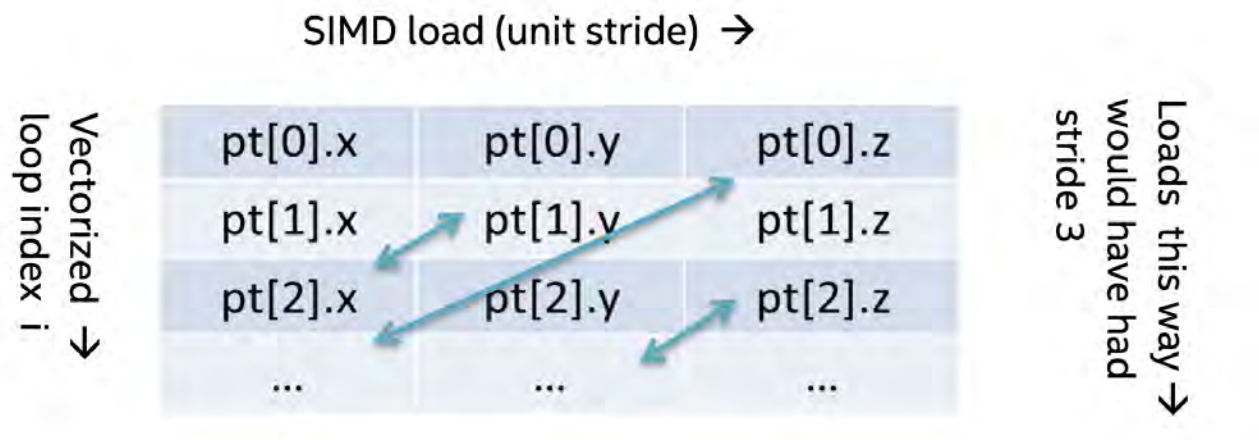
5 Handling large arrays of small structures

The older version 15 compiler vectorizes this loop using strided loads or gathers to load each component of the struct:

```
icc -std=c99 -xmic-avx512 -qopt-report=4 -qopt-report-file=stderr -qopt-report-phase=vec,cg
…
LOOP BEGIN at g2s.c(6,5)
   remark #15415: vectorization support: gather was generated for the variable ptvec:  strided by 3
…
   remark #15300: LOOP WAS VECTORIZED
   remark #15460: masked strided loads: 6
```

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

However, the compiler can do better by noting that the `x, y,` and `z` components of the struct are adjacent in memory. The compiler can perform SIMD loads of the components, and then perform permutations and shuffles to transpose the data so it is laid out for vectorization of the loop over `i`, as illustrated in **Figure 6**.



6 Data transpose to enable vectorization

When compiled with the newer version 17 compiler, the optimization report contains a "code generation" component:

```
Report from: Code generation optimizations [cg]
sumsq.c(10,22):remark #34030: adjacent sparse (strided) loads optimized for speed. Details: stride {
12 }, types { F32-V512, F32-V512, F32-V512 }, number of elements { 16 }, select mask { 0x000000007 }.
```

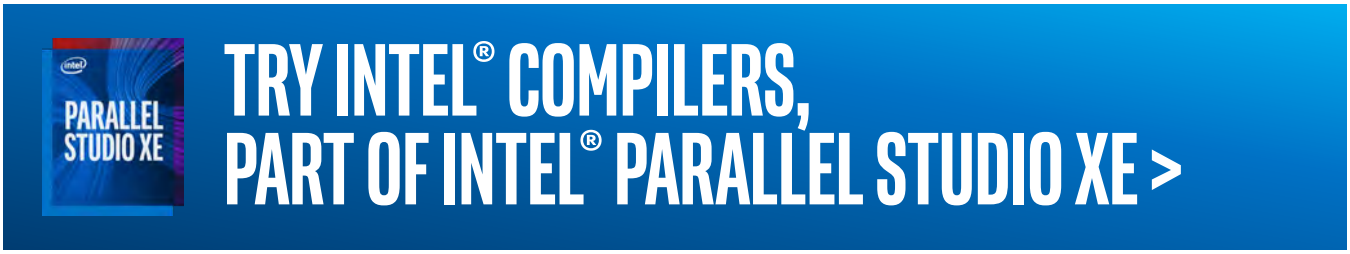
This shows that the compiler was able to convert the original strided loads into contiguous SIMD loads followed by shuffles and permutes. A simple test loop over 10,000 random points, repeated 100,000 times, was compiled for Intel AVX-512 and run on an Intel Xeon Phi 7250 processor. The observed speedup using the version 17 compiler, compared to the version 15 compiler, was more than a factor of 2. The same optimization is performed if pt is a two-dimensional array `pt[10000][3]`, or in Fortran, if `pt` is a two-dimensional array `pt(3,10000)` or a derived type array.

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

The compiler can sometimes perform this “gather to shuffle” optimization when targeting other instruction sets. However, the powerful new shuffle and permute instructions result in it being generated much more frequently when targeting Intel AVX-512.

Summary

Powerful new instructions in Intel AVX-512 enable improved application performance through the vectorization of some loops that could not be vectorized previously, and more efficient vectorization of others. The compiler optimization report shows when and where these optimizations are performed.



Additional Resources

Intel® Xeon Phi™ processor

A Guide to Auto-Vectorization with Intel® C++ Compilers

Explicit Vector Programming in Fortran

Initially written for Intel® Xeon Phi™ x100 coprocessors but also applicable elsewhere:
Vectorization Essentials

Fortran Array Data and Arguments and Vectorization

Intel® Compiler User Forums

Webinars

Getting the Most out of Your Compiler with New Optimization Reports

Further Vectorization Features of the Intel® Compiler

From Serial to Awesome: Advanced Code Vectorization and Optimizations

Data Alignment, Padding, and Peel/Remainder Loops

BLOG HIGHLIGHTS

Direct N-body Simulation

BY **MIKE P. (INTEL)** ›

Exercise in performance optimization on Intel® architecture, including Intel® Xeon Phi™ processors.

NOTE: This lab is an overview of various optimizations discussed in Chapter 4 in the book *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*, second edition (2015). The book can be obtained at **xeonphi.com/book**.

In this step, we will look at how to modernize a piece of code through an example application. The provided source code is an N-body simulation, which is a simulation of many particles that gravitationally or electrostatically interacting with each other. We keep track of the position and the velocity of each particle in the structure "Particle." The simulation is discretized into timesteps. In each timestep, first, the force on each particle (stored in the structure) is calculated with a direct all-to-all algorithm (O(n^2) complexity). Next, the velocity of each particle is modified using the explicit Euler method. Finally, the positions of the particles are updated using the explicit Euler method.

Read more ›

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

INTEL® ADVISOR ROOFLINE ANALYSIS

A New Way to Visualize Performance Optimization Trade-offs

Kevin O'Leary, *Technical Consulting Engineer*; Ilyas Gazizov, *Senior Software Developer*; Alexandra Shinsel, *Technical Consulting Engineer*; Zakhar Matveev, *Product Architect*; and Dmitry Petunin, *Technical Consulting Engineer*, Intel Corporation

Software must be both threaded and vectorized to fully utilize today's and tomorrow's hardware. Data-driven **vectorization** design can yield long-term performance growth with less risk and more impact. Even with perfect vector and thread parallelism, developers often have to additionally balance CPU/vector/thread utilization versus memory subsystem data bottlenecks. This aspect of optimization could often be addressed by using a roofline "bounds and bottlenecks" performance model.

This article provides an overview of **Intel® Advisor 2017** and discusses the new Intel Advisor Roofline Analysis feature. The roofline model provides an intuitive and powerful representation of how to best address performance issues in your application. Finally, a case study shows the optimization process on a real example.

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

Roofline Model

Roofline modeling was first proposed by University of California at Berkeley researchers Samuel Williams, Andrew Waterman, and David Patterson in the paper **Roofline: An Insightful Visual Performance Model for Multicore Architectures** in 2009. Recently, it was extended to address all levels of the memory subsystem, as described by Aleksandar Ilic, Frederico Pratas, and Leonel Sousa in their **Cache-Aware Roofline Model: Upgrading the Loft** paper.

A roofline model provides insight into how your application works by helping you answer these questions:

- Does my application work optimally on the current hardware? If not, what is the most underutilized hardware resource?
- What limits performance? Is my application workload memory or compute bound?
- What is the right strategy to improve application performance?

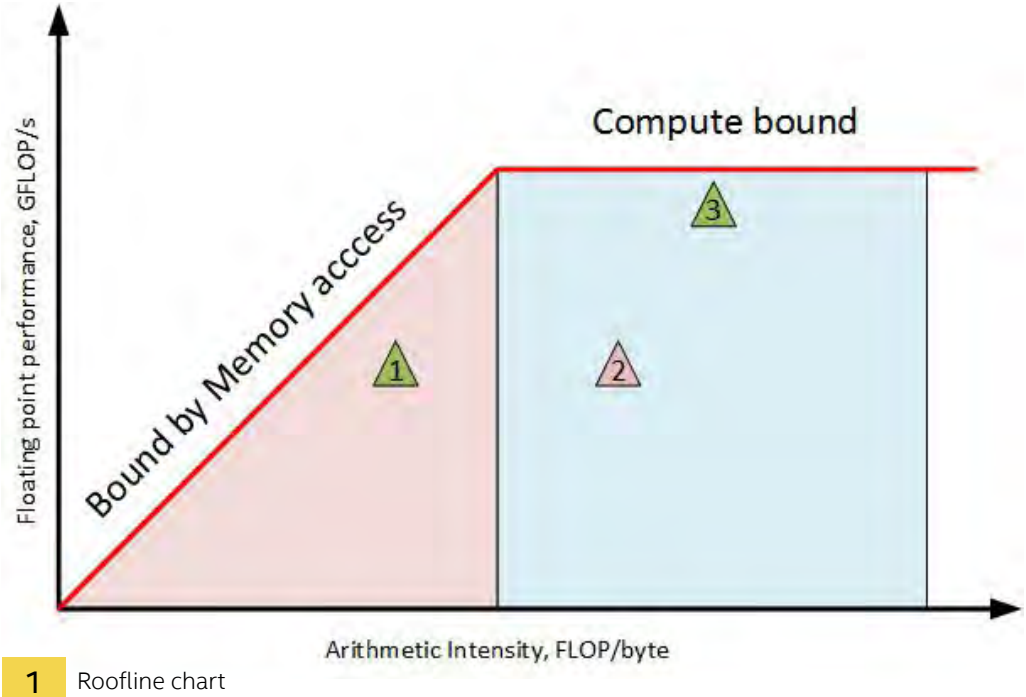
The model plots data to help you visualize application compute and memory bandwidth ceilings by measuring two parameters:

1. Arithmetic intensity, the number of floating-point operations per byte transferred between CPU and memory
2. Floating-point performance measured in billions of floating-point operations per second (GFLOPS)

The proximity of the data points to the model lines (rooflines) shows the degree of optimization (**Figure 1**). The kernels on the right-hand side, in the blue region, are more compute bound. As you move up the Y axis, they get close to the floating-point peak. The performance of these kernels is bounded by the compute capabilities of the platform. To improve the performance of kernel 3, consider migrating this kernel to a highly parallel platform, such as the **Intel® Xeon Phi™** processor, where the compute ceiling and memory throughput are higher. For kernel 2, vectorization can be considered as a performance improvement strategy, since it is far away from the ceiling.

Toward the left-hand side of the plot, in the red region, the kernels are memory bound. As you move up the Y axis, they are limited by the DRAM and cache peak bandwidth of the platform. To increase the performance of these kernels, consider improving the algorithm or its implementation to perform more computations per data item, thereby shifting the plot position to the right, where the performance ceiling is higher. These kernels may also run faster on an Intel Xeon Phi processor because of the higher memory bandwidth.

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

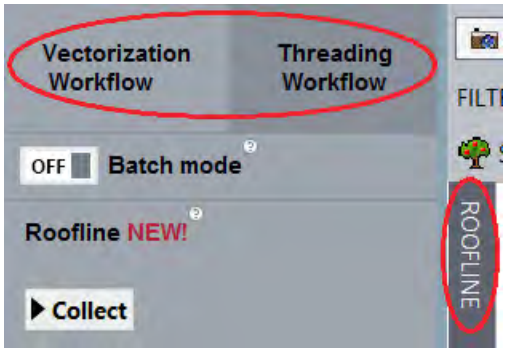


1 Roofline chart

Intel Advisor Overview

Intel Advisor is a software analysis tool offering a powerful software design and performance characterization platform for applications. Intel Advisor incorporates thread parallelism prototyping (Threading Advisor), vector parallelism optimization (Vectorization Advisor), and memory-versus-compute characterization (Advisor Roofline Automation) capabilities.

In this article, we mainly focus on the Advisor Roofline and Vectorization analyses. When using the Intel Advisor GUI, you can switch between Vectorization and Threading Advisor flows using the "Workflow" toggle. The Roofline chart can be accessed by using the Roofline sidebar (**Figure 2**).



2 Intel® Advisor workflow selector

Vectorization Advisor can help you increase the performance of your code using these steps:

- **Survey** shows which loops consume the most time with detailed SIMD statics.
- **FLOPS** and **Trip Counts** measure iteration counts, call counts, and the precise number of floating-point operations per second for each loop and function.
- **Recommendations** gives specific advice on how to fix performance issues.
- **Dependencies Analysis** provides a dynamic dependency analysis to verify if a loop has cross-iteration dependencies that can limit vectorization and parallelization.
- **Memory Access Patterns Analysis** checks if you are accessing memory in a vector-friendly manner.

You can discover many important performance and design insights by combining Vectorization Advisor and Roofline analyses. For example, knowing the Vectorization Efficiency metric provided by the Vectorization Advisor Survey Report is often crucial when interpreting the data on a roofline chart.

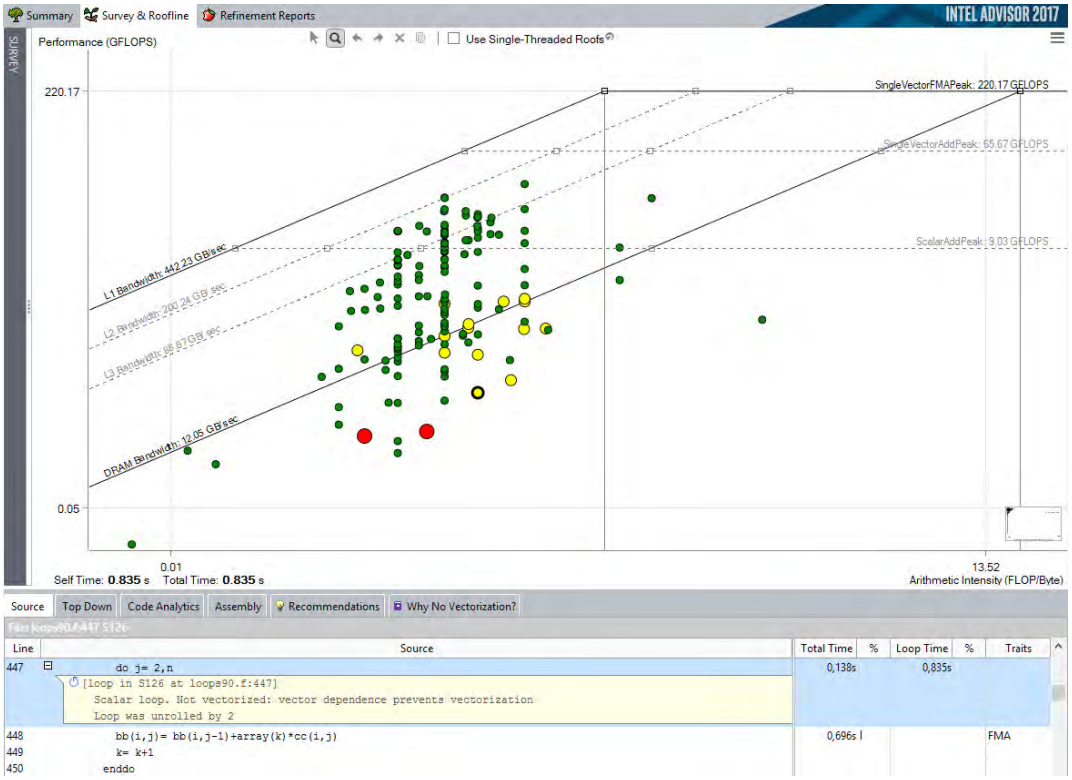
Intel Advisor Roofline Analysis

Intel Advisor implements the "cache-aware" flavor of the roofline model, which provides additional insight by addressing all levels of memory/cache hierarchy:

- Sloped rooflines illustrate peak performance levels if all the data fits into the respective cache.
- Horizontal lines show the peak achievable performance levels if vectorization and other CPU resources are used effectively.

Intel Advisor places a dot for every loop in the roofline plot **(Figure 3)**. The circle sizes and colors denote the relative execution time of the loops. Most of the loops require further optimizations to better utilize cache memory. Some, such as the green dot sitting on the dotted ScalarAddPeak line, just to the right of the vertical line in the middle, may be a loop that is poorly vectorized. As you can see, the roofline chart makes it easy to locate opportunities to improve application performance.

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



3 Intel® Advisor roofline chart with source tab

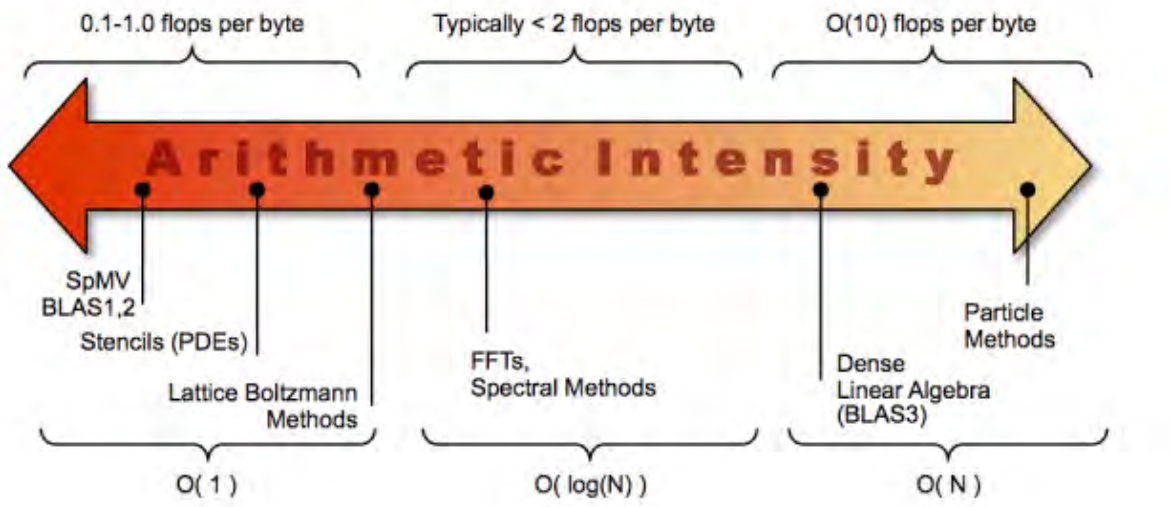
How to Interpret the Intel Advisor Roofline Chart

A roofline plot provides useful information but is not a reference table, in which one simply locates their input and reads the corresponding output. It is a guide that suggests what factors to investigate. It requires interpretation.

The lines on a roofline chart, such as the ones in **Figure 3**, are representative of hardware limits on kernel performance based on benchmarks run by Intel Advisor to establish baselines and performance limits on the host system. The uppermost lines form the roofs that are representative of the maximum performance of the machine. In **Figure 3**, the uppermost lines are "L1 Bandwidth" and "Single Vector FMA Peak." Not every kernel can achieve this performance and may be ultimately limited by lower roofs, depending on the nature of the algorithm (e.g., a kernel that cannot be vectorized would be limited by the maximum performance of scalar computations).

A kernel's horizontal position on the plot is its arithmetic intensity (the number of floating-point operations per byte transferred between CPU and memory, as measured by operand size), which is primarily determined by its algorithm, though this can be altered somewhat by optimizations that occur during compilation. **Figure 4** gives examples of different algorithms and their relative arithmetic intensities. Redesigning a kernel's algorithm to increase its arithmetic intensity pushes it to the right in the roofline chart, which may be helpful in raising its maximum potential performance, due to the slopes of the memory bandwidth roofs.

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



4 Arithmetic Intensity

A kernel's vertical position relative to the various roofs reveals bottlenecks. If a kernel is placed above a roof, then that roof is not the primary performance bottleneck, although it can still affect performance. The roofs above a kernel are potential bottlenecks, each corresponding to an issue that can be overcome using a particular type of optimization. If a kernel is below the scalar computation peak line, then it is worth investigating the kernel's vectorization status. If it is unvectorized, or inefficiently vectorized, it is likely that this roof represents the bottleneck and suggests that it would be prudent to improve or implement the kernel's vectorization if possible. If, on the other hand, this kernel is efficiently vectorized, the scalar computation peak can be ruled out as a bottleneck, and you can move on to investigating the other roofs above the kernel.

Solving Performance Problems Using Intel Advisor

Some Intel Advisor tips are provided in this section.

Tip No. 1: Use the summary view to see the top time-consuming loops (**Figures 5** and **6**) and tuning recommendations **(Figure 7)**.

Top time-consuming loops

Loop	Source Location	Self Time	Total Time	Trip Counts
flessparseapngtc	lesSparse.f:387	5.9143s	5.9143s	6
flessparseapg	lesSparse.f:232	4.9664s	4.9664s	1; 2; 4; 1; 1
flessparseapkg	lesSparse.f:286	4.3634s	4.3634s	2; 1; 4; 1
fmtxblkdot2	ftools.f:445	1.0166s	1.0166s	9162
fmtxvdimvecmult	ftools.f:45	0.8999s	0.8999s	572

5 Summary of top loops

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

6 Summary of top loops with top loop vectorized

Top time-consuming loops

Loop	Self Time	Total Time	Trip Counts
[loop in flessparseapngtc at lesSparse.f:389]	5.879s	5.879s	1; 2
[loop in flessparseapg at lesSparse.f:232]	5.247s	5.247s	1; 2; 4; 1; 1
[loop in flessparseapkg at lesSparse.f:287]	4.247s	4.247s	2; 1; 4; 1
[loop in fmtxblkdot2 at ftools.f:445]	1.158s	1.158s	9162; 3; 1
[loop in fmtxblkdmaxpy at ftools.f:874]	1.148s	1.148s	9162; 3; 1

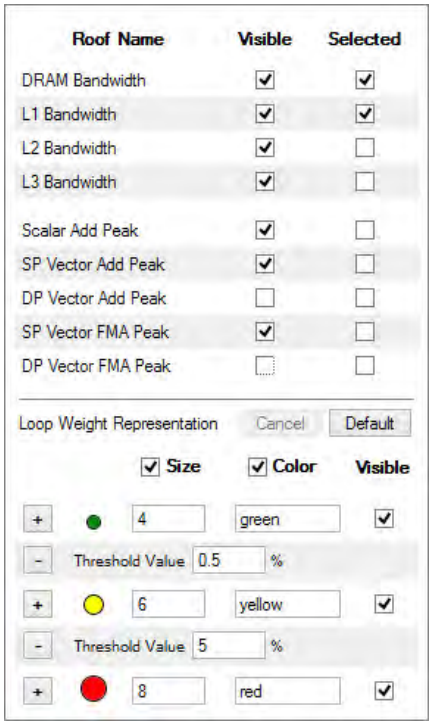
7 Summary of top recommendations

Recommendations

Loop	Self Time	Recommendations
[loop in flessparseapngtc at lesSparse.f:389]	5.879s	Enforce vectorized remainder
[loop in flessparseapg at lesSparse.f:232]	5.247s	Add data padding
[loop in flessparseapkg at lesSparse.f:287]	4.247s	Enforce vectorized remainder Add data padding
[loop in flessparseapsclr at lesSparse.f:514]	0.261s	Disable unrolling Enforce vectorized remainder Align data Add data padding
[loop in flessparseapfull at lesSparse.f:448]	0.190s	Enforce vectorized remainder Add data padding

Tip No. 2: Use roofline customization to remove the roofs you don't need **(Figure 8)**.

For example, if you are operating on only single-precision data, you can safely remove the double-precision peaks from your roofline.



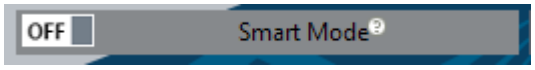
Tip No. 3: Use Smart Mode to find the best optimization candidates **(Figure 9)**.

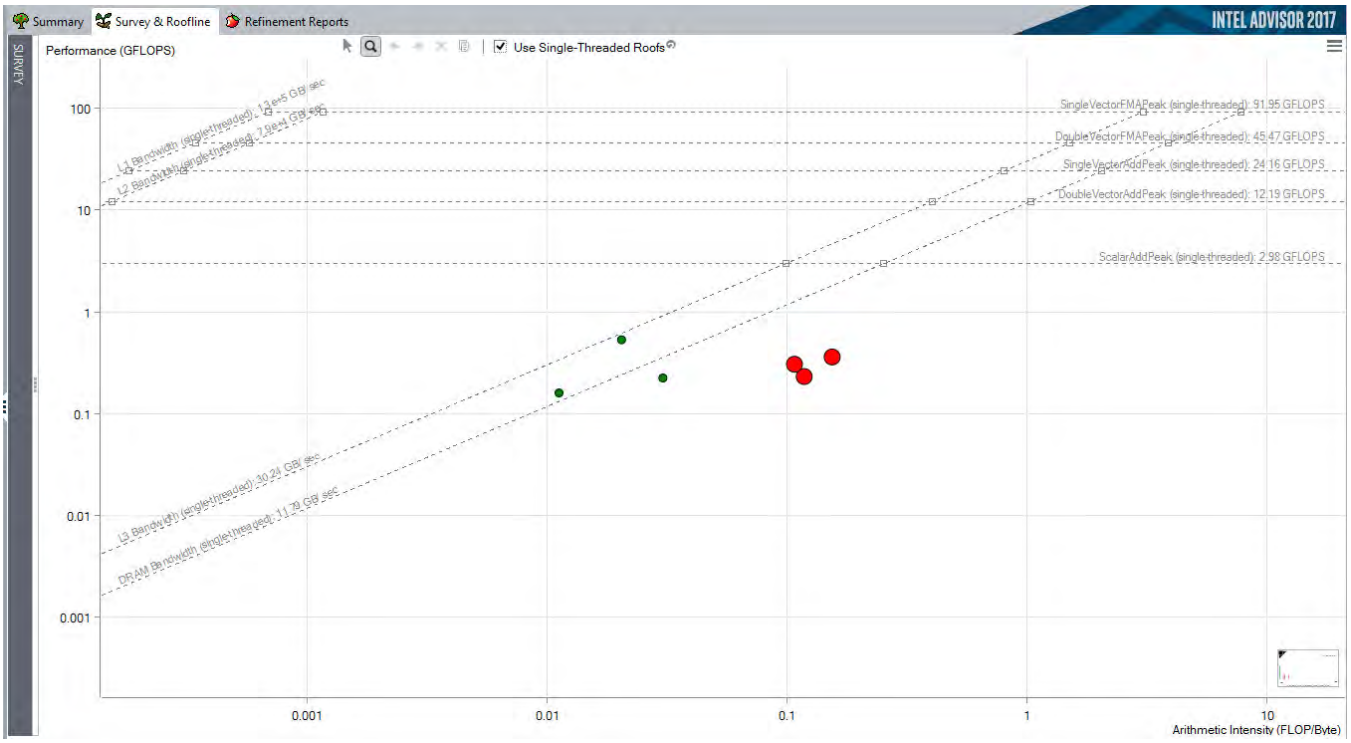
Loops are ordered on the roofline by their Elapsed Self Time, but by activating Smart Mode, you can identify loops that have high total time. The more total time that is spent in a loop, the larger the overall effect of optimizing it can be.

8 Intel® Advisor roofline chart customization

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

9a Intel® Advisor Smart Mode selector





9b Intel® Advisor roofline chart filtered using Smart Mode

Tip No. 4: Use some of the other Intel Advisor features to supplement the information in the roofline chart.

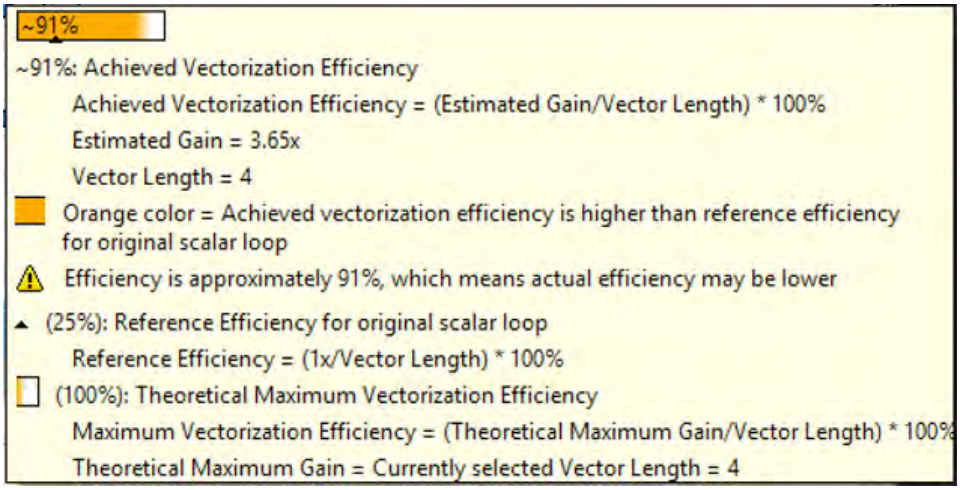
Vectorization efficiency is your vectorization thermometer **(Figures 10** and **11)**. Under Instruction Set Analysis, look at the Traits column to see factors that could be affecting vectorization **(Figure 10)**. Consider running the Intel Advisor memory access pattern collection if you suspect you're referencing memory in a non-vector-friendly fashion. *[Editor's note: Vladimir Tsymbal's article* ***"Identifying Scalability Problems in Parallel Applications on Multicore Systems"*** *in this issue of* The Parallel Universe *describes some techniques to analyze memory access.]*

Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops				Trip Counts	Instruction Set Analysis		
					Vect...	Efficiency	Gain...	VL (...		Traits	Data T...	Num.
1 Inefficient ...	5.281s	5.281s	Vectorized (Bo...		AVX	~55%	2.19x	4	25; 2	Inserts	Float64	3; 7
1 Ineffective ...	2.828s	2.828s	Vectorized (Bo...		AVX	~91%	3.65x	4	25; 2		Float64	3

10 Intel® Advisor survey highlighting vectorization efficiency

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

11 Intel® Advisor Vectorized Efficiency explanation



Tip No. 5: Isolate vectorized from nonvectorized loops using the loop toggle **(Figure 12)**.

12 Intel® Advisor Vectorized/ Not Vectorized loop selector

Tip No. 6: Use the source window together with the roofline chart **(Figure 13)**.

Intel Advisor seamlessly integrates your source code into the performance profile.

BLOG HIGHLIGHTS

Intel® Xeon Phi™ Product Family x200 (KNL) User mode (ring 3) MONITOR and MWAIT

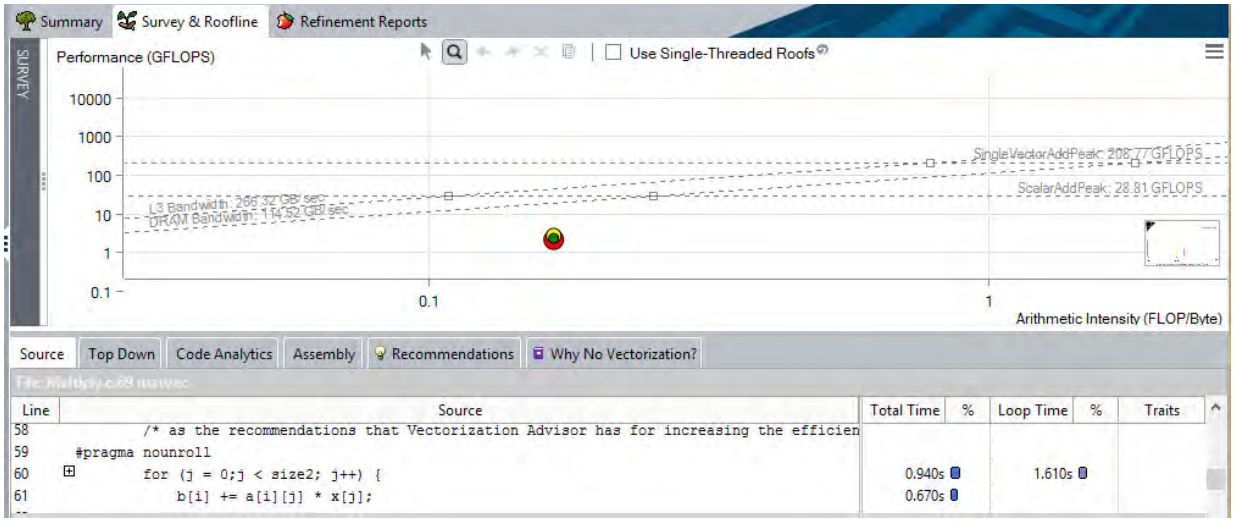
BY JAMES C. (INTEL) › and BENJAMIN C. (INTEL) ›

The Intel® Xeon Phi™ Product Family x200 series processors (formerly known as Knights Landing) contain a model-specific feature, which allows the MONITOR and MWAIT[1] instructions to be executed in rings other than ring 0, whereas architecturally, these instructions are restricted to ring 0 (kernel code). Specifically, this feature allows them to be executed in ring 3, which is normal user mode.

The feature can be enabled by setting bit 1 (as below) in MSR 140H (the MISC_FEATURE_ENABLES model-specific register). The register can also be read to determine whether the instructions are enabled at other than ring 0.

Read more ›

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



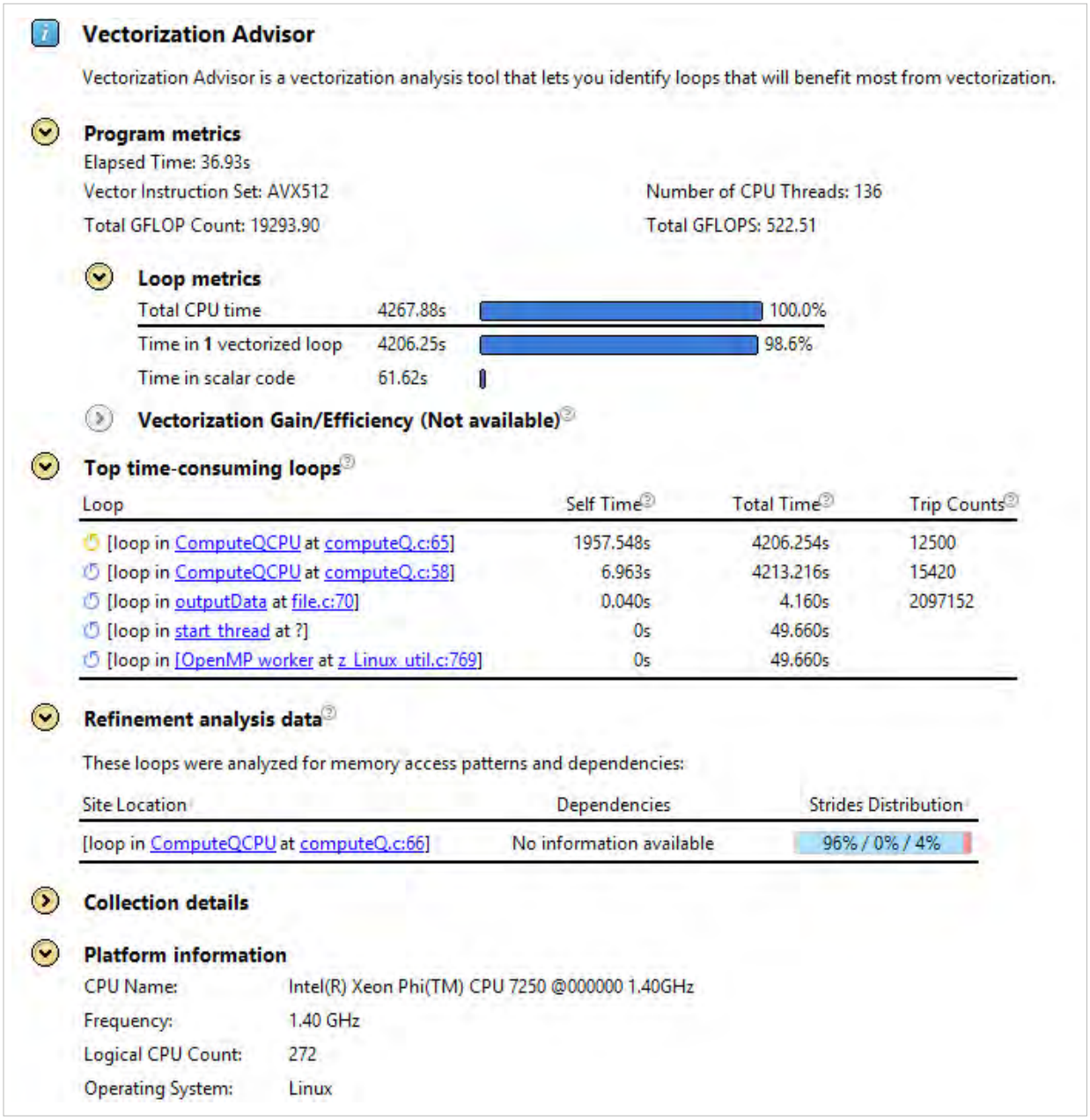
13 Intel® Advisor roofline chart highlighting source integration

Case Study: Using Roofline Analysis to Tune an MRI Image Reconstruction Benchmark

The 514.pomriq SPEC ACCEL Benchmark is an MRI image reconstruction kernel described in Stone et al. (2008). MRI image reconstruction is a conversion from sampled radio responses to magnetic field gradients. The sample coordinates are in the space of magnetic field gradients, or K-space. The Q matrix in the MRI image reconstruction is a precomputable value based on the sampling trajectory, the plan of how points in K-space will be sampled. The algorithm examines a large set of input, representing the intended MRI scanning trajectory and the points that will be sampled. Each element of the Q matrix is computed by a summation of contributions from all trajectory sample points. Each contribution involves a three-element vector dot product of the input and output 3-D location plus a few trigonometric operations. The output Q elements are complex numbers but the inputs are multielement vectors. The kernel is fundamentally compute bound because trigonometric functions are expensive, and the regularity of the problem allows for easy management of memory bandwidth. Therefore, once tiling and data layout remove any artificial memory bandwidth bottleneck, the most important optimizations are low-level sequential code optimizations and improving the instruction stream efficiency, such as loop unrolling.

The input to 514.pomriq consists of one file containing the number of K-space values; the number of X-space values; and the list of K-space coordinates, X-space coordinates, and Phi-field complex values for the K-space samples. Each set of coordinates and the complex values are stored as arrays with each field written contiguously. The 514.pomriq output consists of the resulting Q matrix of complex values in “real, imaginary” format for each line. This case study will analyze the 514.pomriq compute kernel and focus on its optimization. The Intel Advisor summary for 514.pomriq run on an Intel Xeon Phi 7250 processor is shown in **Figure 14**, where it is easy to see that loops involved in computing the Q matrix are the hotspot.

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



Vectorization Advisor

Vectorization Advisor is a vectorization analysis tool that lets you identify loops that will benefit most from vectorization.

Program metrics

Elapsed Time: 36.93s

Vector Instruction Set: AVX512 — Number of CPU Threads: 136

Total GFLOP Count: 19293.90 — Total GFLOPS: 522.51

Loop metrics

Total CPU time	4267.88s	100.0%
Time in **1** vectorized loop	4206.25s	98.6%
Time in scalar code	61.62s	

Vectorization Gain/Efficiency (Not available)

Top time-consuming loops

Loop	Self Time	Total Time	Trip Counts
[loop in ComputeQCPU at computeQ.c:65]	1957.548s	4206.254s	12500
[loop in ComputeQCPU at computeQ.c:58]	6.963s	4213.216s	15420
[loop in outputData at file.c:70]	0.040s	4.160s	2097152
[loop in start_thread at ?]	0s	49.660s	
[loop in [OpenMP worker at z_Linux_util.c:769]	0s	49.660s	

Refinement analysis data

These loops were analyzed for memory access patterns and dependencies:

Site Location	Dependencies	Strides Distribution
[loop in ComputeQCPU at computeQ.c:66]	No information available	96% / 0% / 4%

Collection details

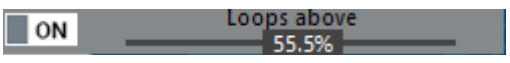
Platform information

CPU Name:	Intel(R) Xeon Phi(TM) CPU 7250 @000000 1.40GHz
Frequency:	1.40 GHz
Logical CPU Count:	272
Operating System:	Linux

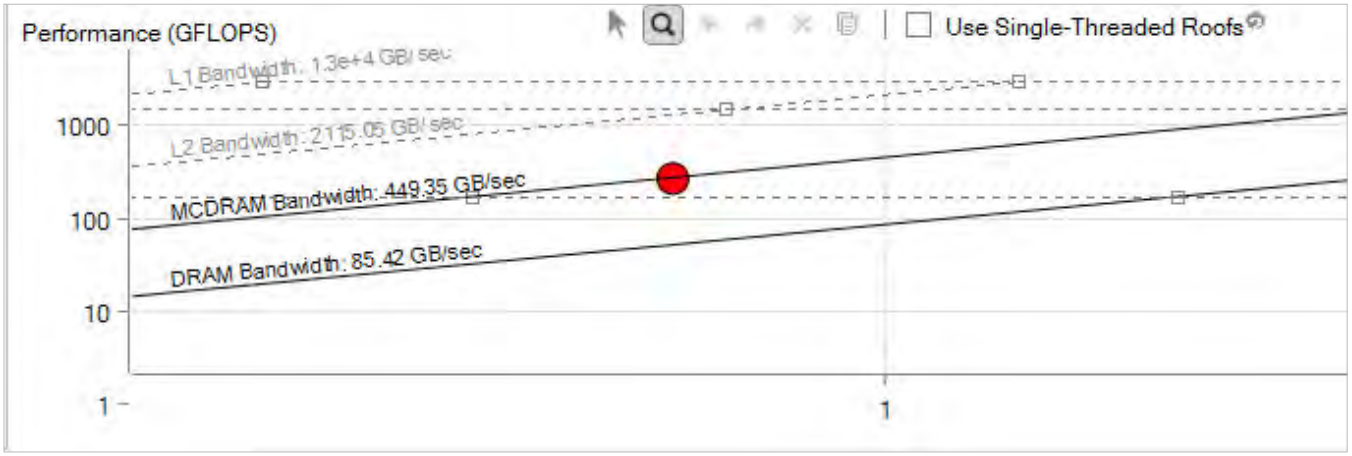
14 Intel® Advisor summary view

We use Intel Advisor Smart Mode to narrow down the best optimization candidates **(Figure 15a)**. The loop where we are spending the most time is vectorized, but it is still below the MCDRAM roof **(Figure 15b)**. This possibly indicates issues with memory use. Let's examine the loop using Code Analytics and Recommendations.

15a Intel® Advisor smart mode selector



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



15b Intel® Advisor roofline chart filtered using smart mode

16 Intel® Advisor Code Analytics tab highlighting gathers

We can see heavy gather instructions here and advice to explore the memory access pattern of the loop **(Figures 16** and **17)**. After running a memory access pattern analysis, we observe a 4 percent gather stride, which gives us a tip as to where there might be the potential bottleneck: nonoptimal memory access **(Figure 18)**. After looking at the details, we can verify that there is no need to use gather instructions since the stride is constant **(Figure 19a)**. We can also see the same in the Intel Advisor recommendations **(Figure 19b)**.

17 Intel® Advisor memory access patterns recommendation

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Max. Site Footprint	Site Name	Recon
[loop in ComputeQCPU at computeQ.c:66]	No information available	96% / 0% / 4%	Mixed strides	3MB	loop_site_17	

Memory Access Patterns Report | Dependencies Report | Recommendations

ID	Stride	Type	Source	Nested Function	Variable references	Access Footprint
P1		Gather stride	computeQ.c:66		block 0x7f0045867010 allocated at main.c:99	3MB
P2		Parallel site information	computeQ.c:66			
P4	0	Uniform stride	omriq_exe_nog2s:0x29a8	__svml_sincosf16	__svml_sincosf16_chosen_core_func	8B
P5	0	Uniform stride	omriq_exe_nog2s:0x29cf	__svml_sincosf16_b3		64B
P6	0	Uniform stride	omriq_exe_nog2s:0x29de	__svml_sincosf16_b3		64B
P7	0	Uniform stride	omriq_exe_nog2s:0x29f3	__svml_sincosf16_b3	__svml_ssincos_data	64B
P8	0	Uniform stride	omriq_exe_nog2s:0x29fa	__svml_sincosf16_b3	__svml_ssincos_data	64B
P9	0	Uniform stride	omriq_exe_nog2s:0x2a07	__svml_sincosf16_b3	__svml_ssincos_data	64B
P10	0	Uniform stride	omriq_exe_nog2s:0x2a14	__svml_sincosf16_b3	__svml_ssincos_data	64B
P11	0	Uniform stride	omriq_exe_nog2s:0x2a1b	__svml_sincosf16_b3	__svml_ssincos_data	64B

18 Intel® Advisor Memory Access Patterns Report

19a Intel® Advisor Memory Access Pattern Report details

Details View

Gather (irregular) access

Operand Size (bits): 32
Operand Type: bit*16;float32*16
Vector Length: 16
Memory access footprint: 3MB

Gather/scatter details

Pattern: "Constant (non-unit)"

Instruction accesses values with constant offset from the base:
- stride within instruction = X
- stride between iterations = X*vector length

Horizontal stride (bytes): 16
Vertical stride (bytes): 256

Mask is constant
Mask: [1111111111111111]
Active elements in the mask: 100.0%

Variable references

Names: block 0x7f0045867010 allocated at main.c:99

Issue: Inefficient gather/scatter instructions present

The compiler assumes indirect or irregular stride access to data used for vector operations. Improve memory access by alerting the compiler to detected regular stride access patterns, such as:

Pattern	Description
Invariant	The instruction accesses values in the same memory throughout the loop.
Uniform (Horizontal Invariant)	The instruction accesses values in the same memory within the vector iteration.
Vertical Invariant	The instruction accesses the memory locations using the same offset across all vector iterations.
Unit	The instruction accesses values in contiguous memory throughout the loop, and the stride between vector iterations = vector length.

Recommendation: Refactor code with detected regular stride access patterns — Confidence: Low

The Memory Access Patterns Report shows the following regular stride access(es):

19b Intel® Advisor gather/scatter recommendation

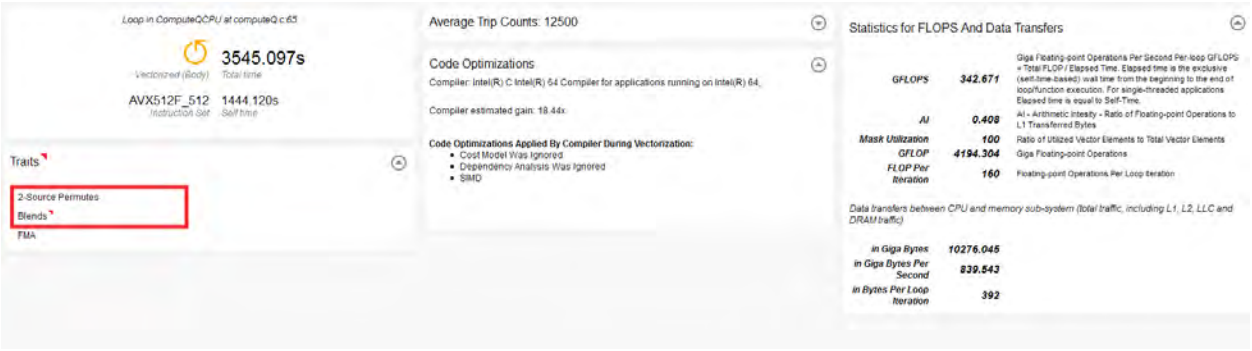
```
#pragma omp simd private(expArg, cosArg, sinArg) reduction(+:QrSum, QiSum)
      for (indexK = 0; indexK < numK; indexK++) {
        expArg = PIx2 * (kVals[indexK].Kx * x[indexX] +
        kVals[indexK].Ky * y[indexX] +
        kVals[indexK].Kz * z[indexX]);

        cosArg = cosf(expArg);
        sinArg = sinf(expArg);

        float phi = kVals[indexK].PhiMag;
        QrSum += phi * cosArg;
        QiSum += phi * sinArg;
      }
```

20 Source code of the rate-limiting kernel

Everything becomes clear after checking the source code of the kernel (**Figure 20**). The code uses an array of structures, which become "gathers" after vectorization. However, newer versions of the Intel® compiler can recognize the access pattern and apply optimizations to get rid of gathers in order to use more lightweight instructions. "Gathers" replacement is performed by the "Gather to Shuffle/Permutes" compiler transformation and can often be profitable on modern CPUs, especially on platforms with Intel AVX-512 support. *[Editor's note: Martyn Corden's article "***Vectorization Opportunities for Improved Performance with Intel® AVX-512***" in this issue of* The Parallel Universe *describes how the Intel® Compiler 2017 takes advantage of Intel AVX-512 to create new opportunities for loop vectorization.]*

Let's take a look at the roofline after recompilation using the new Intel compiler (e.g., Intel Compiler 2017 Update 1) with "Gather to Shuffle/Permutes" support. We can see that the dot is above MCDRAM now, and there are no gather instructions (replaced with Intel AVX-512 "2-source permutes"), as well as an increased number of floating-point operations per second **(Figures 21** and **22)**.

21 Intel® Advisor roofline chart showing loop now above MCDRAM bandwidth



22 Intel® Advisor Code Analytics tab

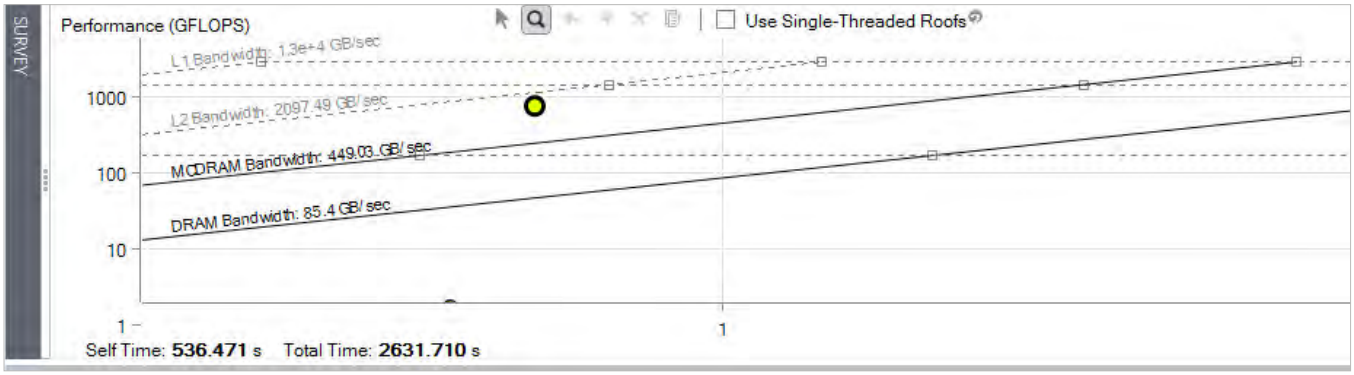
However, there is a more effective way to resolve such issues: AOS (array of structures) to SOA (structure of arrays) conversion. This optimization allows us to use more convenient data containers to improve efficiency during vector processing. In the past, it involved manually reworking the underlying data structures. Now, using the Intel® SIMD Data Layout Templates library (**Figure 23**), we can simply improve the performance by adding a few lines of code where the `kValues` structure is declared, where the structure is initialized, and where the K-values are computed.

```
#include <sdlt/sdlt.h>
struct kValues {
  float Kx;
  float Ky;
  float Kz;
  float PhiMag;
};

SDLT_PRIMITIVE(kValues, Kx, Ky, Kz, PhiMag)


sdlt::soa1d_container<kValues> inputKValues(numK);
auto kValues = inputKValues.access();

  for (k = 0; k < numK; k++) {
    kValues [k].Kx() = kx[k];
    kValues [k].Ky() = ky[k];
    kValues [k].Kz() = kz[k];
    kValues [k].PhiMag() = phiMag[k];
  }


auto kVals = inputKValues.const_access();
#pragma omp simd private(expArg, cosArg, sinArg) reduction(+:QrSum, QiSum)
      for (indexK = 0; indexK < numK; indexK++) {
        expArg = PIx2 * (kVals[indexK].Kx() * x[indexX] +
        kVals[indexK].Ky() * y[indexX] +
        kVals[indexK].Kz() * z[indexX]);

        cosArg = cosf(expArg);
        sinArg = sinf(expArg);

        float phi = kVals[indexK].PhiMag();
        QrSum += phi * cosArg;
        QiSum += phi * sinArg;
      }
```

23 Using the SIMD Data Layout Templates library

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

Let's check the new roofline chart **(Figure 24)**. After applying this optimization, the dot is no longer red. This means it takes less time now, and it has more GFLOPS, putting it close to the L2 roof. Additionally, the loop now has unit stride access and, as a result, no special memory manipulations. The total performance improvement is almost 3x for the kernel and 50 percent for the entire application. Additionally, the loop now has unit stride access and, as a result, no special memory manipulations (**Figure 25a** and **25b**).



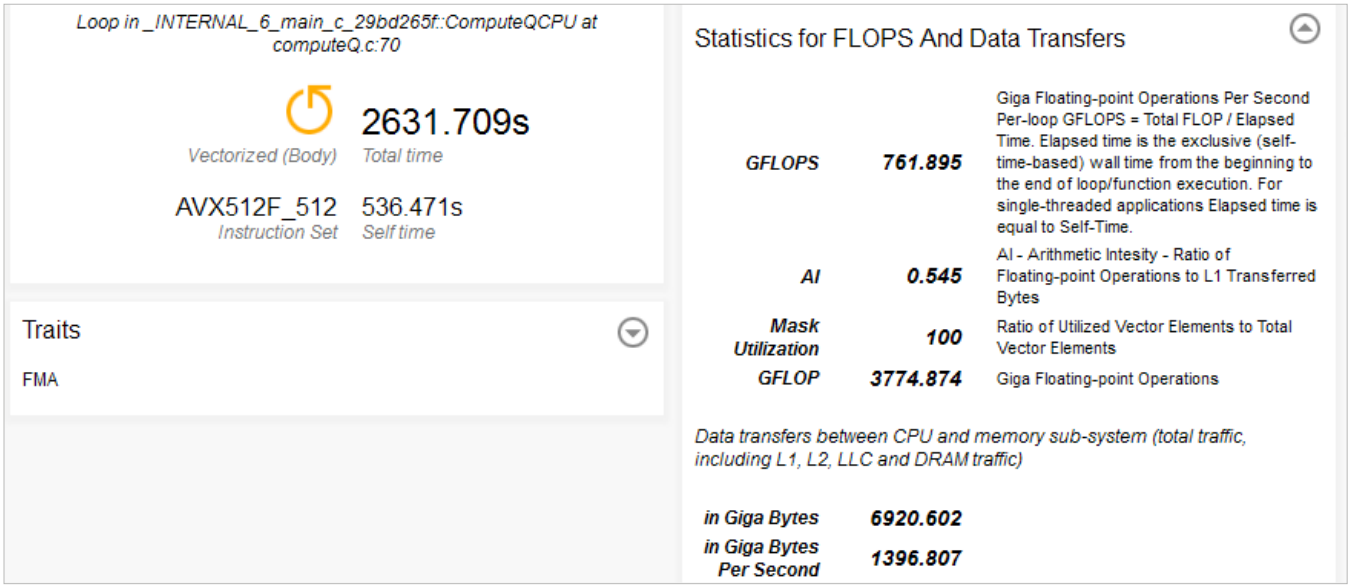
24 Intel® Advisor final optimized roofline chart

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Max. Site Footprint	Site Name	Recommendations
[loop in _ZN4sd...	No information available	100% / 0% / 0%	All unit strides	769KB	loop_site_27	

Memory Access Patterns Report | Dependencies Report | Recommendations

ID	Stride	Type	Source	Nested Function	Variable references	Access Footprint	Modules	Site Name
P1		Parallel site information	rvalue_binary_operator_proxy_other_xmacro.h:50				omriq_exe_soa	loop_site_27
P3	0	Uniform stride	omriq_exe_soa:0x2d28	__svml_sincosf16	__svml_sincosf16_chosen_core_func	8B	omriq_exe_soa	loop_site_27
P4	0	Uniform stride	omriq_exe_soa:0x2d4f	__svml_sincosf16_b3		64B	omriq_exe_soa	loop_site_27
P5	0	Uniform stride	omriq_exe_soa:0x2d5e	__svml_sincosf16_b3		64B	omriq_exe_soa	loop_site_27
P6	0	Uniform stride	omriq_exe_soa:0x2d73	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P7	0	Uniform stride	omriq_exe_soa:0x2d7a	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P8	0	Uniform stride	omriq_exe_soa:0x2d87	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P9	0	Uniform stride	omriq_exe_soa:0x2d94	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P10	0	Uniform stride	omriq_exe_soa:0x2d9b	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P11	0	Uniform stride	omriq_exe_soa:0x2da2	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P12	0	Uniform stride	omriq_exe_soa:0x2db6	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P13	0	Uniform stride	omriq_exe_soa:0x2dcb	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P14	0	Uniform stride	omriq_exe_soa:0x2dd2	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P15	0	Uniform stride	omriq_exe_soa:0x2ddf	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P16	0	Uniform stride	omriq_exe_soa:0x2de6	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P17	0	Uniform stride	omriq_exe_soa:0x2df4	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P18	0	Uniform stride	omriq_exe_soa:0x2e13	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P19	0	Uniform stride	omriq_exe_soa:0x2e1a	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P20	0	Uniform stride	omriq_exe_soa:0x2e21	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P21	0	Uniform stride	omriq_exe_soa:0x2e28	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P22	0	Uniform stride	omriq_exe_soa:0x2e41	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P23	0	Uniform stride	omriq_exe_soa:0x2e48	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P24	0	Uniform stride	omriq_exe_soa:0x2e55	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P25	0	Uniform stride	omriq_exe_soa:0x2e62	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P26	0	Uniform stride	omriq_exe_soa:0x2e7b	__svml_sincosf16_b3	__svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P27	0	Uniform stride	omriq_exe_soa:0x2e96	__svml_sincosf16_b3		64B	omriq_exe_soa	loop_site_27
P28	0	Uniform stride	omriq_exe_soa:0x2e9e	__svml_sincosf16_b3		64B	omriq_exe_soa	loop_site_27
P29	1	Unit stride	computeQ.c:72	ComputeQCPU		769KB	libiomp5.so; omriq_exe_soa	loop_site_27
P30	1	Unit stride	computeQ.c:73	ComputeQCPU		769KB	libiomp5.so; omriq_exe_soa	loop_site_27
P31	1	Unit stride	computeQ.c:79	ComputeQCPU		769KB	libiomp5.so; omriq_exe_soa	loop_site_27
P32	1	Unit stride	computeQ.c:80	ComputeQCPU		769KB	libiomp5.so; omriq_exe_soa	loop_site_27
P33	1	Unit stride	rvalue_binary_operator_proxy_other_xmacro.h:50			769KB	libiomp5.so; omriq_exe_soa	loop_site_27

25a Intel® Advisor final optimized Memory Access Patterns Report



25b Intel® Advisor final optimized Code Analytics tab

Conclusion

The roofline model provides a new, visually intuitive, and powerful representation of your application's performance. By using the proper optimization techniques, as indicated by the region of the roofline chart your application is in, you can avoid wasting valuable time on optimizations that will have minimal impact on your performance. The roofline model can answer the following questions:

- Can I get better performance?
- What are the key performance bottlenecks: memory or CPU?
- How much speedup can I get if I optimize a particular bottleneck?
- How much speedup can I get if I use another platform?

As systems get bigger and more complex, getting these answers is nontrivial, but roofline analysis can save you time and effort.

Modernize Your 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 article, and taking advantage of the powerful tools in **Intel® Parallel Studio XE**, can make the modernization task dramatically easier.
- Use Intel® Advisor roofline analysis, now available in Intel® Parallel Studio XE 2017 Update 1.
- Send an email to **vector_advisor@intel.com** to get the latest information on some exciting new capabilities that are currently under development.

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

Useful Intel Advisor Links

Get started with the **Intel® Advisor roofline feature**

Selftime-based FLOPS computing with an important explanation of how to interpret roofline results for nested loops

Analyzing Intel MPI applications with Intel Advisor

References

Stone, S. S.; J. P. Haldar, S. C. Tsao, W. W. Hwu., Z. Liang, and B. P. Sutton. *"Accelerating advanced MRI reconstructions on GPUs."* In International Conference on Computing Frontiers, pages 261–272, 2008.

INTEL-POWERED DEEP LEARNING FRAMEWORKS

Your Path to Deeper Insights

Pubudu Silva, *Senior Software Engineer,* Machine Learning and Computer Vision, Intel Corporation

Artificial intelligence (AI)—the concept of intelligent machines that are able to perform tasks such as visual understanding, speech perception, language processing, and decision-making that otherwise require human intelligence—continues to be the next big thing, at least since the introduction of computers.

Machine learning is proving to be very effective in performing some of the key AI tasks. Artificial neural networks (ANNs), a loose model of the mammalian cerebral cortex neuronal structure, were especially promising for AI due to their ambitious design and general applicability to a wide variety of tasks. The strength of ANNs lies in their ability to learn and maintain hidden transient states (hidden nodes). This makes it possible for them to learn a wide range of mappings, from the input to the desired output, by cascading several nonlinear functions.

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

In a learned ANN, hidden layers represent the internal abstraction of the data in hierarchical stages, with deeper layers representing higher levels of abstraction. It is believed that mammalian brains also process information with multiple hierarchical processing layers (e.g., in primate visual systems, processing is done in a sequence of stages from edge detection, primitive shape detection, and moving up to gradually more complex visual shapes).[1] Therefore, multilayer, "deeper" ANNs are naturally desired for AI research.

Networks that process data in a sequence of multiple stages with deep cascaded layers are typically called "deep networks." Most of the widely used machine learning algorithms—such as support vector machines (SVMs), mixture of Gaussian (MoG), k-nearest neighbors (kNN), principal component analysis (PCA), and kernel density estimation (KDE)—don't contain more than three layers of processing. Hence, they can be considered "shallow" architectures. ANNs with two to three layers can be successfully trained. There were several unsuccessful attempts in training deeper ANNs during the last decades of the twentieth century. They faced two main issues:

1. Vanishing gradient issues
2. Over-fitting arising from the increased number of weights introduced by the additional layers

With the advances in computing, researchers were able to train machine learning algorithms with millions of data samples in a relatively shorter time, effectively resolving the over-fitting issues. Convolutional neural networks (CNNs) are deep networks with multiple layer types, but they contain many fewer weights (than a fully connected ANN of equivalent depth) due to their weight-sharing philosophy. Hence, CNNs are much easier to train than ANNs. With CNNs, theoretical best performance is only slightly worse than that of the ANNs. They have become very popular for supervised image learning tasks. Thanks to the breakthrough discovery of Hinton et al. in 2006,[2] successful deep networks—such as deep belief networks and deep auto-encoders—also made their way to unsupervised learning.

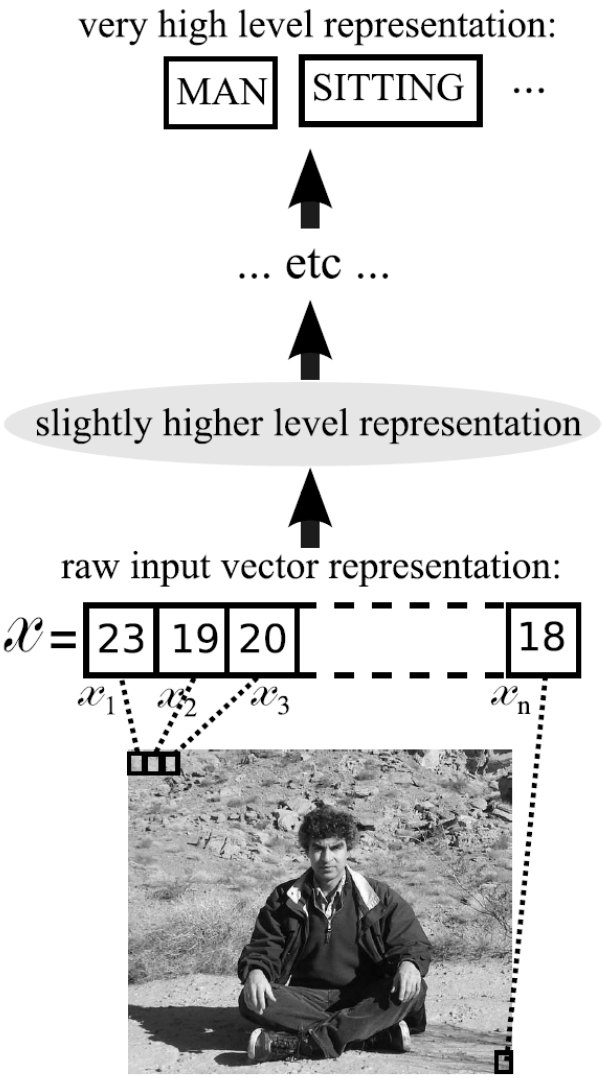
In general, deep networks have outperformed almost all other machine learning algorithms in most AI-related tasks such as classification, regression, image captioning, and natural language processing. The stunning success of deep networks can be attributed to the way they autonomously learn feature hierarchies.

The key contrast between deep learning and traditional statistical learning methods is that the latter learns on human-engineered features of the data while the former learns on raw data itself. Deep networks autonomously generate the features best suited to a given task in their early levels as DNNs learn. This effectively removes the guesswork and human bias from the learning process, leaving the whole learning task to a cost function–based optimization process on the given original data.

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

Deep, layered structure allows learning hierarchies of features, where deeper levels learn higher-level features based on the lower-level features learned by prior levels of the network.

Figure 1 shows how an input image is transformed into gradually higher levels of representation in a deep network. The deeper the image goes into the network, the more and more abstract the representation becomes. For example, learned feature hierarchy, from initial layers to deeper ones, could be edges, shapes, parts of objects, total objects, the scene, etc. However, in practice, it is hard to speculate what the "right" feature vectors should be for each of these hierarchical layers of abstraction without going through the learning. This underscores the key issue in learning on human engineered features: generally, in deeper networks, the output layer gets to process very high-level features, enabling it to learn much higher-level concepts than is possible with shallower networks.



As the various deep learning techniques have become standard tools for developers, data scientists, and researchers, a number of deep learning frameworks (such as Caffe, Tensorflow*, Theano*, and Torch), and libraries (MatConvNet, CNTK, Pylearn2, and Deeplearning4j) have been developed to help easily train and score deep networks. These frameworks and libraries are immensely helpful in reducing tedious boilerplate work. The user can focus effort on the deep learning aspects rather than implementing the individual components.

1 Deep network processing of an image input by transforming it to gradually higher levels of representation
Source: Yoshua Bengio, Learning Deep Architectures for AI, 2009

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

Additionally, users have access to the codebase of most of the frameworks and libraries, since they are often launched as open source projects, with active contributions from the developer community. Since deep learning typically involves training on super-large data sets, for days and even weeks, performance optimization of the commonly used deep learning software is critical to the advancement of the technology in general.

Intel consistently contributes to the open source deep learning frameworks especially by optimizing them for Intel® architecture. Its **machine learning site** contains up-to-date information on Intel's involvement in machine learning and deep learning. More information about the performance optimization tools and techniques can be found there. Some of the optimization efforts are published as case studies to guide software developers in their own deep learning applications and any frameworks or libraries they may use in the development cycle.

For example, the process followed in optimizing Caffe is presented in the case study **Caffe Optimized for Intel® Architecture: Applying Modern Code Techniques**. **Intel® VTune™ Amplifier** is a powerful profiling tool that provides valuable insights, which can be used as the initial guidance for performance optimization process, such as CPU and cache usage, CPU core utilization, memory usage, threading load balance, and thread locks. Libraries such as **Intel® Math Kernel Library (Intel® MKL)**, **Intel® Threading Building Blocks (Intel® TBB)**, and OpenMP* have proved to be very instrumental in optimizing deep learning software.

In order to accelerate the deep learning development and research, Intel recently announced the **Intel® Deep Learning SDK**, which is a free set of tools for data scientists and software developers to develop, train, and deploy deep learning solutions. The SDK is designed as a Web-based client connected to an Ubuntu*/CentOS server. The simple installation wizard installs the SDK with the popular deep learning frameworks that are optimized for Intel architecture on the server. The training tool of the SDK greatly simplifies the preparation of training data, model design, and model selection with its simple graphical user interface and advanced visualization techniques. Deployment tools can be used to optimize trained deep learning models to specific target devices via model compression and weight quantization techniques.

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

References

1. Serre, T.; Kreiman, G.; Kouh, M.; Cadieu, C.; Knoblich, U.; and Poggio, T. 2007. "A quantitative theory of immediate visual recognition." *Progress in Brain Research, Computational Neuroscience: Theoretical Insights into Brain Function,* 165, 33–56.

2. Hinton, G. E.; Osindero, S.; and Teh, Y. 2006. A fast learning algorithm for deep belief nets. *Neural Computation,* 18, 1527–1554.

Your Path to Deeper Insight

Find up-to-date information on Intel-optimized deep learning frameworks, libraries, and exciting deep learning tools such as the Intel Deep Learning SDK at **intel.com**.

LEARN MORE ABOUT INTEL® DEEP LEARNING SDK AND OTHER AI SOFTWARE TOOLS >

intel Software

ACCELERATE LEARNING WITH INTEL TECHNICAL WEBINARS

Get ready to improve code performance with the latest tips and tricks from Intel® software experts. Watch these free webinars on these hot topics.

Intel® Parallel Studio XE 2017: Create Faster Code—Faster ›

Optimize for AVX-512 with or without AVX-512 Hardware ›

Scale Your Application across Shared and Distributed Memory! ›

Remove Python* Performance Barriers for Machine Learning ›

Code for Speed with High Bandwidth Memory on Intel® Xeon Phi™ Processors ›

Vectorization, the "Other" Parallelism You Need to Take Advantage Of! ›

Roofline Analysis: A New Way to Visualize Performance Optimization Tradeoffs ›

Happy Together: Develop Media Solutions & Apps for the Latest Intel® Processors ›

Accelerating Deep Learning and Machine Learning with Intel Libraries ›

Intel® HPC Orchestrator: A System Software Stack Providing Key Building Blocks of Intel® Scalable System Framework ›

Keep an eye out for more all-new webinars happening in Spring 2017.

Missed anything? **See all archived webinars here ›**

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

© 2017 Intel Corporation. All rights reserved. Intel, the Intel logo, and Intel Xeon Phi are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

THE PARALLEL UNIVERSE

© 2017 Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Inside, the Intel Inside logo, Intel Atom, Intel Cilk, Intel Core, VTune, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.