OPTIMIZATION WORKSHOP

Intel® VTune™ Amplifier and Intel® Advisor

Kevin O'Leary, Technical Consulting Engineer
Changing Hardware Affects Software Development

More cores and wider vector registers mean more threads and more maximum performance! ... but you need to need to write software that takes advantage of those cores and registers.

More threads means more potential speedup.

<table>
<thead>
<tr>
<th>Intel® Xeon® Processor</th>
<th>64-bit</th>
<th>5100 series</th>
<th>5500 series</th>
<th>5600 series</th>
<th>E5-2600</th>
<th>E5-2600 V2</th>
<th>E5-2600 V3</th>
<th>E5-2600 V4</th>
<th>Platinum 8180</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>12</td>
<td>18</td>
<td>22</td>
<td>28</td>
</tr>
<tr>
<td>Threads</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>24</td>
<td>36</td>
<td>44</td>
<td>56</td>
</tr>
<tr>
<td>SIMD Width</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>512</td>
</tr>
</tbody>
</table>
The Agenda

- Optimization 101
- Threading
- The uOp Pipeline
- Tuning to the Architecture
- Vectorization
- Q & A
Optimization 101

Take advantage of compiler optimizations with the right flags.

<table>
<thead>
<tr>
<th>Linux*</th>
<th>Windows*</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-xCORE-AVX512</td>
<td>/QxCORE-AVX512</td>
<td>Optimize for Intel® Xeon® Scalable processors, including AVX-512.</td>
</tr>
<tr>
<td>-xCOMMON-AVX512</td>
<td>/QxCOMMON-AVX512</td>
<td>Alternative, if the above does not produce expected speedup.</td>
</tr>
<tr>
<td>-fma</td>
<td>/Qfma</td>
<td>Enables fused multiply-add instructions. (Warning: affects rounding!)</td>
</tr>
<tr>
<td>-O2</td>
<td>/O2</td>
<td>Optimize for speed (enabled by default).</td>
</tr>
<tr>
<td>-g</td>
<td>/Zi</td>
<td>Generate debug information for use in performance profiling tools.</td>
</tr>
</tbody>
</table>

Use optimized libraries, like Intel® Math Kernel Library (MKL).

<table>
<thead>
<tr>
<th>Linear Algebra</th>
<th>Fast Fourier Transforms</th>
<th>Vector Math</th>
<th>Summary Statistics</th>
<th>Deep Neural Networks</th>
<th>And More...</th>
</tr>
</thead>
<tbody>
<tr>
<td>• BLAS</td>
<td>• Multidimensional</td>
<td>• Trigonometric</td>
<td>• Kurtosis</td>
<td>• Convolution</td>
<td>• Splines</td>
</tr>
<tr>
<td>• LAPACK</td>
<td>• FFTW interfaces</td>
<td>• Hyperbolic</td>
<td>• Variation coefficient</td>
<td>• Pooling</td>
<td>• Interpolation</td>
</tr>
<tr>
<td>• ScalAPACK</td>
<td>• Cluster FFT</td>
<td>• Exponential</td>
<td>• Order statistics</td>
<td>• Normalization</td>
<td>• Trust Region</td>
</tr>
<tr>
<td>• Sparse BLAS</td>
<td></td>
<td>• Log</td>
<td>• Min/max</td>
<td>• ReLU</td>
<td>• Fast Poisson</td>
</tr>
<tr>
<td>• Sparse Solvers</td>
<td></td>
<td>• Power</td>
<td>• Variance-</td>
<td>• Softmax</td>
<td>Solver</td>
</tr>
<tr>
<td>• Iterative</td>
<td></td>
<td>• Root</td>
<td>covariance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• PARDISO*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Cluster Sparse Solver</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary Statistics
• Kurtosis
• Variation coefficient
• Order statistics
• Min/max
• Variance-covariance

Deep Neural Networks
• Convolution
• Pooling
• Normalization
• ReLU
• Softmax

And More...
• Splines
• Interpolation
• Trust Region
• Fast Poisson Solver
Adding Threading with Intel® Advisor

- Find good threading sites with the **Survey** analysis, then annotate the code to tell Advisor how to simulate threading and locking.

- Use the **Suitability** analysis to predict threading performance and the **Dependencies** analysis to check for correctness problems.

Set hypothetical environment details to see effects.

Experiment with what would happen if you changed the number or duration of parallel tasks without re-running the analysis.

See how each parallel site would scale on a given number of CPUs.

Predicted program speedup.
Using Intel® VTune™ Amplifier for Threading Optimization

Use **Threading** analysis to see how well your program is using its threads.

Each thread is displayed on the timeline, with color coded activity.

- Coarse-grain locks indicate that your program is effectively single threaded.
- Thread imbalance is when the application isn't using all the threads all the time.
- Lock contention means your program is spending more time swapping threads out than actually working.
What is the uop Pipeline?

There are multiple steps to executing an instruction.

Modern CPUs pipeline instructions rather than performing all the steps for one instruction before beginning the next instruction.

The pipeline can be divided into two sections.
- The Front End, which fetches instructions, decodes them, and allocates them to...
- The Back End, which executes the uops. Once completed, a uop is considered “retired.”

A Pipeline Slot is a representation of the hardware needed to process a uop.

The Front End can only allocate (and the Back End retire) so many uops per cycle. This determines the number of Pipeline Slots. In general, there are four.
Pipeline Slot Categorization

Pipeline slots can be sorted into four categories on each cycle.
- Retiring
- Back End Bound
- Bad Speculation
- Front End Bound

Each category has an expected range of values in a well tuned application.

<table>
<thead>
<tr>
<th>App. Type: Category</th>
<th>Client/Desktop</th>
<th>Server/Database/Distributed</th>
<th>High Performance Computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retiring</td>
<td>20-50%</td>
<td>10-30%</td>
<td>30-70%</td>
</tr>
<tr>
<td>Bad Speculation</td>
<td>5-10%</td>
<td>5-10%</td>
<td>1-5%</td>
</tr>
<tr>
<td>Front End Bound</td>
<td>5-10%</td>
<td>10-25%</td>
<td>5-10%</td>
</tr>
<tr>
<td>Back End Bound</td>
<td>20-40%</td>
<td>20-60%</td>
<td>20-40%</td>
</tr>
</tbody>
</table>
Pipeline Slot Categorization: Retiring

This is the good category! You want as many of your slots in this category as possible. However, even here there may be room for optimization.
Pipeline Slot Categorization: Bad Speculation

This occurs when a uop is removed from the back end without retiring; effectively, it’s cancelled, most often because a branch was mispredicted.
Pipeline Slot Categorization: Back End Bound

This is when the back end can’t accept uops, even if the front end can send them, because it already contains uops waiting on data or long execution.
Pipeline Slot Categorization: Front End Bound

This is when the front end can’t deliver uops even though the back end can take them, usually due to delays in fetching code or decoding instructions.
The Tuning Process

**Find Hotspots**
- Intel® VTune™ Amplifier
- Hotspots Analysis

**Determine Efficiency**
- Intel® VTune™ Amplifier
- Microarchitecture Exploration

**If Inefficient:**
- Diagnose Bottleneck
- Intel® VTune™ Amplifier
- Microarchitecture Exploration
- Memory Access Analysis
- Intel® Advisor
- Vectorization Advisor

**Implement Solution**
Finding Hotspots

Use **Hotspots** analysis. Find where your program is spending the most time to ensure optimizations have a bigger impact.

- The Summary tab shows a high-level overview of the most important data.

- The Bottom-up tab provides more detailed analysis results.
  - The total amount of time spent in a function is divided up by how many CPUs were active during the time the function was running.
  - Low confidence metrics are grayed out: VTune uses statistical sampling and may miss extremely small, fast portions of the program.
Determining Efficiency

Use **Microarchitecture Exploration** analysis. It's preconfigured with:

- appropriate events and metric formulae for the architecture
- hardware-specific thresholds for highlighting potential problems in pink

Inefficiency can be caused by:

- Not retiring enough necessary instructions.
  - Look for retiring rate lower than expected value.
- Retiring too many unnecessary instructions.
  - Look for underuse of AVX or FMA instructions.

<table>
<thead>
<tr>
<th>Address</th>
<th>Source Line</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1400010f5</td>
<td>58</td>
<td>xor eax, eax</td>
</tr>
<tr>
<td>0x1400010f7</td>
<td>63</td>
<td>vmmovd xmm0, edx</td>
</tr>
<tr>
<td>0x1400010fb</td>
<td>63</td>
<td>vpbroadcastd ymm1, xmm0</td>
</tr>
</tbody>
</table>
Diagnosing the Bottleneck

Intel® VTune™ Amplifier has hierarchical expanding metrics categorized by the four slot types. You can follow the pink highlights down the subcategories to identify the root cause. You can hover over a column to see a helpful tooltip.

We can’t cover all solutions today, but there’s more information in the Tuning Guides: https://software.intel.com/en-us/articles/processor-specific-performance-analysis-papers
## Solutions Sampler

### Back End Bound

#### Core Bound

**Divider**
- Use reciprocal-multiplication where possible.

#### Memory Bound

**Contested Access/Data Sharing**
- Solve false sharing by padding variables to cache line boundaries.
- Try to reduce actual sharing requirements.

**Remote Memory Access**
- Affinitize/pin threads to cores.
- Use NUMA-efficient thread schedulers like Intel® Threading Building Blocks.
- Test whether performance improves using Sub-NUMA Cluster Mode.

**Cache Misses**
- Block your data.
- Use software prefetches.
- Consider Intel® Optane™ DC Persistent Memory.
Understanding the Memory Hierarchy

Data can be in any level of any core's cache, or in the shared L3, DRAM, or on disk.

Accessing data from another core adds another layer of complexity.

Cache coherence protocols – beyond the scope of today's lecture. But we will cover some issues caused by this.
Cache Misses

**Why:** Cache misses raise the CPI of an application. Focus on long-latency data accesses coming from 2nd and 3rd level misses

**What Now:** If either metric is highlighted for your hotspot, consider reducing misses:
- Change your algorithm to reduce data storage
- Block data accesses to fit into cache
- Check for sharing issues (See Contested Accesses)
- Align data for vectorization (and tell your compiler)
- Use streaming stores
- Use software prefetch instructions

“<memory level> Bound” = Percentage of cycles when the CPU is stalled, waiting for data to come back from <memory level>
Categorizing Inefficiencies in the Memory Subsystem

- Back End bound is the most common bottleneck type for most applications.
- It can be split into Core Bound and Memory Bound
  - **Core Bound** includes issues like not using execution units effectively and performing too many divides.
  - **Memory Bound** involves cache misses, inefficient memory accesses, etc.
    - Store Bound is when load-store dependencies are slowing things down.
    - The other sub-categories involve caching issues and the like. Memory Access Analysis may provide additional information for resolving this performance bottleneck.
VTune Amplifier Workflow Example - Summary View

High percentage of L3 Bound cycles
VTune Amplifier Workflow Example- Bottom-Up View

- Over-Time DRAM Bandwidth
- Over-Time QPI/UPI Bandwidth
- Grid Breakdown by Function (configurable)
VTune Amplifier Workflow Example - Bottom-Up View

Focus on areas of interest with "Zoom In and Filter"

Fine-grained details in Zoomed-in view
VTune Amplifier Workflow Example - Bottom-Up View

Memory Bound function. 44% of pipeline slots are stalled.

DRAM and UPI Bandwidth are low.

Double-click a function for source view.
VTune Amplifier Workflow Example - Source View

Metrics at a source line granularity

Inefficient array access pattern in nested loop
### Intel® Optane™ DC Persistent Memory

Determine whether your application can benefit from Intel® Optane™ DC Persistent Memory without the hardware using **Memory Consumption** analysis. Identify frequently accessed objects using a **Memory Access** analysis.

<table>
<thead>
<tr>
<th>Memory Mode</th>
<th>App Direct Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires no special programming. Just turn it on and see if it helps!</td>
<td>Requires the use of an API to manually control memory allocation.</td>
</tr>
<tr>
<td>Not actually persistent. Acts like an extra layer of cache between DRAM and disk.</td>
<td>Comes in Volatile (non-persistent) and Non-Volatile (persistent) modes.</td>
</tr>
<tr>
<td>Hottest data should remain in DRAM while the rest goes to persistent memory instead of disk.</td>
<td>Hottest and/or store-heavy objects should reside in DRAM and the rest in persistent memory.</td>
</tr>
</tbody>
</table>

**Non-Volatile Persistent Memory** may not behave as expected. Errors can be detected early using **Intel® Inspector – Persistence Inspector.**

*Note: Memory Consumption analysis is not currently available on Windows* operating systems.
Solutions Sampler

**Back End Bound**

**Core Bound**
- Divider
  - Use reciprocal-multiplication where possible.

**Memory Bound**
- Contested Access/Data Sharing
  - Solve false sharing by padding variables to cache line boundaries.
  - Try to reduce actual sharing requirements.
- Remote Memory Access
  - Affinitize/pin threads to cores.
  - Use NUMA-efficient thread schedulers like Intel® Threading Building Blocks.
  - Test whether performance improves using Sub-NUMA Cluster Mode.
- Cache Misses
  - Block your data.
  - Use software prefetches.
  - Consider Intel® Optane™ DC Persistent Memory.

**Front End Bound**

**Front End Latency**
- Use switches to reduce code size, such as `/O1` or `/Os`.
- Use Profile-Guided Optimization (PGO) with the compiler.
- For dynamically generated code, try co-locating hot code, reducing code size, and avoiding indirect calls.

**Bad Speculation**

**Branch Mispredicts**
- Avoid unnecessary branching.
- Hoist popular branch targets.
- Use PGO with the compiler.

**Machine Clears**
- Check for lock contention or 4k aliasing.

**Retiring**
- You're doing more work than you need to.
  - Use FMAs. Compile with `–fma` or `/Qfma` and the appropriate `–x` or `/Qx` option.
  - Take advantage of vectorization with AVX-512!
Vectorization 101

Vector registers and SIMD (Single Instruction Multiple Data) instructions allow a CPU to do multiple operations at once.

- **Use /QxCORE–AVX512 or –xCORE–AVX512 compiler flags.**
  - If you don't see the expected improvement, try COMMON–AVX512 instead.

“Automatic” Vectorization Is Not Enough

Explicit pragmas and optimization are often required

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, 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. For more information go to [http://www.intel.com/performance](http://www.intel.com/performance).
Intel® Advisor

Intel® Advisor is a thread prototyping and vectorization optimization tool. Start with a **Survey** analysis.

- Are your loops vectorized?
- Vectorized Loop
- Unvectorized Loop

- What’s dragging your performance down?
- What should you do next?

- How much time is a given loop taking?

- Are you using the latest instruction set?
- What’s preventing vectorization?

- How efficient is your vectorization?

---

**Performance Issues**

<table>
<thead>
<tr>
<th>Function Call Sites and Loops</th>
<th>Self Time</th>
<th>Total Time</th>
<th>Type</th>
<th>Why No Vectorization?</th>
<th>Vect...</th>
<th>Efficien...</th>
<th>Gain E...</th>
</tr>
</thead>
<tbody>
<tr>
<td>[loop in main at rofile.cpp:247]</td>
<td>7.59s</td>
<td>7.59s</td>
<td>Vectorized (Body)</td>
<td>AVX2 31% 1.22x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in main at rofile.cpp:247]</td>
<td>7.51s</td>
<td>7.51s</td>
<td>Vectorized (Body)</td>
<td>AVX2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in main at rofile.cpp:247]</td>
<td>0.078s</td>
<td>0.078s</td>
<td>Remainder</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in main at rofile.cpp:260]</td>
<td>3.01s</td>
<td>3.01s</td>
<td>Vectorized (Body)</td>
<td>AVX2 99% 3.98x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in main at rofile.cpp:273]</td>
<td>2.48s</td>
<td>2.48s</td>
<td>Vectorized (Body)</td>
<td>AVX2 99% 3.98x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in main at rofile.cpp:256]</td>
<td>0.016s</td>
<td>3.03s</td>
<td>Scalar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Other names and brands may be claimed as the property of others.
Trip Counts analysis shows you loop trip counts and call counts. High call counts amplify the importance of tuning a loop. Scalar trip counts that aren’t divisible by vector length cause remainder loops.

Loops with peels and/or remainders can be expanded.

This loop’s scalar trip count was 1326, which doesn’t divide evenly by 4. \(1326/4 = 331.5\)

You can see which component loops are what type in this column.

Poor efficiency + high call count = major performance penalty!

This is especially important with the long vector registers of AVX-512!
Trip Counts analysis can also collect FLOP and Mask Utilization data.

- **Floating-point Operations** are used to calculate FLOPS (Floating Point Operations Per Second)... but Integer operations are also supported!

FLOPS and IntOPS are computation-specific performance measurements. Collecting them produces a Roofline chart, a visual representation of performance relative to hardware limits.

- The horizontal axis is Arithmetic Intensity, a measurement of FLOPs per byte accessed. The vertical axis is performance.
- The dots are loops. The lines are hardware limitations; horizontal lines are compute limits and diagonal lines are memory limits.
VNNI usage verification by Intel Advisor

Easily identify VNNI usage in the functions/loops
Roofline

The Roofline chart can be an effective means of identifying bottlenecks, and determining what optimizations to make where, for maximum effect. It is a good indicator of:

▪ How much performance is left on the table
▪ Which loops take the most time
▪ Which loops are worth optimizing
▪ Likely causes of performance bottlenecks
▪ What to investigate next

Roofline Article: https://software.intel.com/en-us/articles/intel-advisor-roofline
Memory Access Patterns & Dependencies

Memory Access Patterns (MAP) and Dependencies are specialized analysis types. Use them when Advisor recommends.

- MAP detects inefficient strides and mask utilization information.
- Dependencies determines whether it's safe to force vectorization in a loop that was left scalar due to the compiler detecting a potential dependency.
Intel® Advisor GUI
Typical Vectorization Optimization Workflow

There is no need to recompile or relink the application, but the use of `-g` is recommended.

**In a rush:** Collect Survey data and analyze loops iteratively

**Looking for detail:**

1. Collect survey and tripcounts data [*Roofline*]
   - Investigate application place within roofline model
   - Determine vectorization efficiency and opportunities for improvement

2. Collect memory access pattern data
   - Determine data structure optimization needs

3. Collect dependencies
   - Differentiate between real and assumed issues blocking vectorization

![Diagram of Typical Vectorization Optimization Workflow]

NOTE: Roofline analysis = Survey analysis + Trip Counts & FLOP analysis
What is the Roofline Model?
Characterization of your application performance in the context of the hardware

It uses two simple metrics
- Flop count
- Bytes transferred

\[ a_i = b_i + c_i \times d_i \]

2 Operations

1W+3R = 4*4 bytes = 16 bytes

Vectorization, Threading
Optimization of Memory Access

Roofline first proposed by University of California at Berkeley:
Cache-aware variant proposed by University of Lisbon:
*Cache-Aware Roofline Model: Upgrading the Loft*, 2013
Roofline Model in Intel® Advisor

Intel® Advisor implements a Cache Aware Roofline Model (CARM)
- “Algorithmic”, “Cumulative (L1+L2+LLC+DRAM)” traffic-based
- Invariant for the given code / platform combination

How does it work?
- Counts every memory movement
- Instrumentation - Bytes and Flops
- Sampling - Time

<table>
<thead>
<tr>
<th>Advantage of CARM</th>
<th>Disadvantage of CARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Hardware counters</td>
<td>Only vertical movements !</td>
</tr>
<tr>
<td>Affordable overhead (at worst =~10x)</td>
<td>Difficult to interpret</td>
</tr>
<tr>
<td>Algorithmic (cumulative L1/L2/LLC)</td>
<td>How to improve performance ?</td>
</tr>
</tbody>
</table>
Roofline Chart in Intel® Advisor

- Roof values are measured
- Dots represent profiled loops and functions
- High level of customization
TUNING A SMALL EXAMPLE WITH ROOFLINE

A Short Walk Through the Process
Example Code
A Short Walk Through the Process

The example loop runs through an array of structures and does some generic math on some of its elements, then stores the results into a vector. It repeats this several times to artificially pad the short run time of the simple example.

```cpp
for (int r = 0; r < REPEAT; r++)
{
    for (int i = 0; i < SIZE; i++)
    {
        X[i] = ((7.4 * Y[i].a + 14.2) + Y[i].b * 3.1) * Y[i].a + 42.0;
    }
}
```
Finding the Initial Bottleneck
A Short Walk Through the Process

The loop is initially under the Scalar Add Peak. The Survey confirms the loop is not vectorized.

The “Why No Vectorization?” column reveals why.
Overcoming the Initial Bottleneck
A Short Walk Through the Process

The recommendations tab elaborates: the dependency is only assumed.

Running a Dependencies analysis confirms that it's false, and recommends forcing vectorization with a pragma.
The Second Bottleneck
A Short Walk Through the Process

Adding a pragma to force the loop to vectorize successfully overcomes the Scalar Add Peak. It is now below L3 Bandwidth.

The compiler is not making the same algorithmic optimizations, so the AI has also changed.

```c
for (int r = 0; r < REPEAT; r++)
{
    #pragma omp simd
    for (int i = 0; i < SIZE; i++)
    {
        X[i] = ((7.4 * Y[i].a + 14.2) + Y[i].b * 3.1) * Y[i].a + 42.0;
    }
```
Diagnosing Inefficiency
A Short Walk Through the Process

While the loop is now vectorized, it is inefficient. Inefficient vectorization and excessive cache traffic both often result from poor access patterns, which can be confirmed with a MAP analysis.

Array of Structures is an inefficient data layout, particularly for vectorization.
A New Data Layout
A Short Walk Through the Process

Changing Y to SoA layout moved performance up again.

```
26  vector<double> X(SIZE);
27  typedef struct SoA
28  {
29     double a[SIZE];
30     double b[SIZE];
31     double pad1[SIZE];
32     double pad2[SIZE];
33  } SoA;
34  SoA Y;
```

Either the Vector Add Peak or L2 Bandwidth could be the problem now.
Improving the Instruction Set
A Short Walk Through the Process

Because it’s so close to an intersection, it’s hard to tell whether the Bandwidth or Computation roof is the bottleneck. Checking the Recommendations tab guides us to recompile with a flag for AVX2 vector instructions.
Assembly Detective Work
A Short Walk Through the Process

The dot is now sitting directly on the Vector Add Peak, so it is meeting but not exceeding the machine’s vector capabilities. The next roof is the FMA peak. The Assembly tab shows that the loop is making good use of FMAs, too.

The Code Analytics tab reveals an unexpectedly high percentage of scalar compute instructions.

The only scalar math op present is in the loop control.
One More Optimization
A Short Walk Through the Process

Scalar instructions in the loop control are slowing the loop down.

Unrolling a loop duplicates its body multiple times per iteration, so control makes up proportionately less of the loop.
Recap
A Short Walk Through the Process

17.156s
Original scalar loop.

9.233s
Vectorized with a pragma.

4.250s
Switched from AoS to SoA.

3.217s
Compiled for AVX2.

2.406s
Unrolled with a pragma.
INTEGRATED ROOFLINE
Beyond CARM: Integrated Roofline

New capability in Intel® Advisor: use simulation based method to estimate specific traffic across memory hierarchies.

- Record load/store instructions
- Use knowledge of processor cache structure and size
- Produce estimates of traffic generated at each level by individuals loops/functions
Integrated Roofline Representation

Choose memory level

Hover for details
New and improved summary

Program metrics

Elapsed Time 154.92s
Vector Instruction Set AVX512, AVX2, AVX, SSE2, SSE
Number of CPU Threads 1

<table>
<thead>
<tr>
<th>Effective Program Characteristics</th>
<th>Utilization</th>
<th>Hardware Peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOPS</td>
<td>10.16</td>
<td>out of 100.1 (DP) FLOPS</td>
</tr>
<tr>
<td></td>
<td>10%</td>
<td>201.7 (SP) FLOPS</td>
</tr>
<tr>
<td>GINTOPS</td>
<td>1.723</td>
<td>out of 53.94 (Int64) INTOPS</td>
</tr>
<tr>
<td></td>
<td>3.2%</td>
<td>106.2 (Int32) INTOPS</td>
</tr>
<tr>
<td>CPU &lt;-&gt; Memory [L1+NTS GB/s]</td>
<td>34.71</td>
<td>out of 450.6 GB/s [bytes]</td>
</tr>
<tr>
<td></td>
<td>1.2e+3%</td>
<td></td>
</tr>
</tbody>
</table>

Performance characteristics

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU time</td>
<td>154.55s</td>
</tr>
<tr>
<td>Time in 3 vectorized loops</td>
<td>142.89s</td>
</tr>
<tr>
<td>Time in scalar code</td>
<td>11.66s</td>
</tr>
</tbody>
</table>

Vectorization Gain/Efficiency

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorized Loops Gain/Efficiency</td>
<td>3.37x</td>
</tr>
<tr>
<td>Program Approximate Gain</td>
<td>3.19x</td>
</tr>
</tbody>
</table>
Roofline compare
Flow Graph Analyzer

Workflows: Create, Debug, Visualize and Analyze

Design mode
- Allows you to create a graph topology interactively
- Validate the graph and explore what-if scenarios
- Add C/C++ code to the node body
- Export C++ code using Threading Building Blocks (TBB) flow graph API

Analysis mode
- Compile your application (with tracing enabled)
- Capture execution traces during the application run
- Visualize/analyze in Flow Graph Analyzer
- Works with TBB and OpenMP
Summary

2nd gen Intel® Xeon® Scalable processors have more performance capacity than ever before, but code needs to be written to take advantage of it!

- **Build a good foundation**
  - Use the right compiler flags and libraries
  - Write your application to make good use of multithreading
    - Use **Intel® Advisor** to plan your threading
    - Use Intel® VTune™ Amplifier’s **Threading** analysis to optimize your threading

- **Tune to the architecture with performance profiling tools.**
  - Find your hotspots with VTune™ Amplifier’s **Hotspots** analysis type.
  - Diagnose your bottlenecks with the **Microarchitecture Exploration** analysis type
    - Dig deeper with a **Memory Access** analysis or **Intel® Advisor**
  - Implement solutions based on your findings
    - Use **Intel® Inspector** to make good use of Intel® Optane™ DC Persistent Memory
Get the Most from Your Code Today with Intel® Tech.Decoded

Visit TechDecoded.intel.io to learn how to put key optimization strategies into practice with Intel development tools.

**Big Picture Videos**
Discover Intel’s vision for key development areas.

**Essential Webinars**
Gain strategies, practices and tools to optimize application and solution performance.

**Quick Hit How-To Videos**
Learn how to do specific programming tasks using Intel® tools.

**TOPICS:**
- Visual Computing
- Code Modernization
- Systems & IoT
- Data Science
- Data Center & Cloud
Legal Disclaimer & Optimization Notice

Performance results are based on testing from 2010 thru 2017 and may not reflect all publicly available security updates. See configuration disclosure for details. No product or component can be absolutely secure.

Benchmark results were obtained prior to implementation of recent software patches and firmware updates intended to address exploits referred to as "Spectre" and "Meltdown". Implementation of these updates may make these results inapplicable to your device or system.

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. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Copyright © 2019, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

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.
Configurations for 2010-2017 Benchmarks

Platform Hardware and Software Configuration

| Platform | Unscaled Core Frequency | Cores/ Socket | Num Sockets | L1 Cache | L2 Cache | L3 Cache | Memory | Memory Frequency | Memory Access | H/W Prefetchers Enabled | HT Enabled | Turbo Enabled | C States | O/S Name | Operating System | Compiler Version |
|----------|-------------------------|---------------|-------------|----------|----------|----------|--------|------------------|--------------|-----------------------|-------------|---------------|----------|----------|------------|------------------|-----------------|
| WSM      | Intel® Xeon™ X5680 Processor 3.33 GHZ | 6          | 2          | 32K      | 256K     | 12 MB    | 48 MB  | 1333 MHz         | NUMA         | Y         | Y         | Y        | Disabled   | Fedora 20 | 3.11.10-301.fc20 | icc version 17.0.2 |
| SNB      | Intel® Xeon® E5 2690 Processor 2.9 GHZ | 8          | 2          | 32K      | 256K     | 20 MB    | 64 GB  | 1600 MHz         | NUMA         | Y         | Y         | Y        | Disabled   | Fedora 20 | 3.11.10-301.fc20 | icc version 17.0.2 |
| IVB      | Intel® Xeon® E5 2697v2 Processor 2.7 GHZ | 12         | 2          | 32K      | 256K     | 30 MB    | 64 GB  | 1867 MHz         | NUMA         | Y         | Y         | Y        | Disabled   | RHEL 7.1  | 3.10.0-229.el7.x86_64 | icc version 17.0.2 |
| HSW      | Intel® Xeon® E5 2600v3 Processor 2.2 GHZ | 18         | 2          | 32K      | 256K     | 46 MB    | 128 GB | 2133 MHz         | NUMA         | Y         | Y         | Y        | Disabled   | Fedora 20 | 3.15.10-200.fc20.x86_64 | icc version 17.0.2 |
| BDW      | Intel® Xeon® E5 2600v4 Processor 2.3 GHZ | 18         | 2          | 32K      | 256K     | 46 MB    | 256 GB | 2400 MHz         | NUMA         | Y         | Y         | Y        | Disabled   | RHEL 7.0  | 3.10.0-123.el7.x86_64 | icc version 17.0.2 |
| BDW      | Intel® Xeon® E5 2600v4 Processor 2.2 GHZ | 22         | 2          | 32K      | 256K     | 56 MB    | 128 GB | 2133 MHz         | NUMA         | Y         | Y         | Y        | Disabled   | CentOS 7.2 | 3.10.0-327.el7.x86_64 | icc version 17.0.2 |
| SKX      | Intel® Xeon® Platinum 81xx Processor 2.5 GHZ | 28         | 2          | 32K      | 1024K    | 40 MB    | 192 GB | 2666 MHz         | NUMA         | Y         | Y         | Y        | Disabled   | CentOS 7.3 | 3.10.0-514.10.2.el7.x86_64 | icc version 17.0.2 |

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. For more information go to http://www.intel.com/performance

Copyright © Intel Corporation 2019
*Other names and brands may be claimed as the property of others.