OPTIMIZE USING ROOFLINE AUTOMATION AND SIMD ANALYSIS TOOLS
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 showing the workflow with labels: Run Roofline analysis, Investigate loops, Improve app performance, Run Dependencies analysis, Mark for deeper analysis, Run MAP analysis, 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

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

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

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.

```c
vector<double> X(SIZE);
typedef struct AoS {
    double a;
    double b;
    double pad1;
    double pad2;
} AoS;
AoS Y[SIZE];

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.

- vector dependence prevents vectorization
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.

<table>
<thead>
<tr>
<th>Site Location</th>
<th>Strides Distribution</th>
<th>Recommendations</th>
</tr>
</thead>
<tbody>
<tr>
<td>[loop in main at rofile.cpp:53]</td>
<td>50% / 50% / 0%</td>
<td>1 Inefficient memory access patterns present</td>
</tr>
</tbody>
</table>

Optimization Notice

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

```
#pragma unroll(8)
#pragma omp simd
for (int i = 0; i < SIZE; i++)
{
    X[i] = ((7.4 * Y.a[i] + 14.2) + Y.b[i] * 3.1) * Y.a[i] + 42.0;
}
```
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

<table>
<thead>
<tr>
<th></th>
<th>Effect</th>
<th>Utilization</th>
<th>Hardware Peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed Time</td>
<td>154.92s</td>
<td>10%</td>
<td>11.89</td>
</tr>
<tr>
<td>Vector Instruction Set</td>
<td>AVX512, AVX2, AVX, SSE2, SSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of CPU Threads</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Effective Program Characteristics

- **GFLOPS**: 10.16 (10%) out of 100.1 (DP) FLOPS, 201.7 (SP) FLOPS
- **GINTOPS**: 1.723 (3.2%) out of 53.94 (Int64) INTOPS, 106.2 (Int32) INTOPS
- **CPU <-> Memory (I1+NTS GB/s)**: 34.71 (1.2e+3%) out of 450.6 GB/s [bytes]

#### Performance characteristics

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU time</td>
<td>154.55s</td>
<td>100%</td>
</tr>
<tr>
<td>Time in 3 vectorized loops</td>
<td>142.89s</td>
<td>92.5%</td>
</tr>
<tr>
<td>Time in scalar code</td>
<td>11.66s</td>
<td>7.5%</td>
</tr>
</tbody>
</table>

### Vectorization Gain/Efficiency

- Vectorized Loops Gain/Efficiency: 3.37x (4.2% gain)
- Program Approximate Gain: 3.19x
Roofline compare
Performance results are based on testing as of 2/22/2019 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure.

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, the Intel logo, Pentium, Xeon, Core, VTune, OpenVINO, Cilk, are trademarks of Intel Corporation or its subsidiaries 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.

Notice revision #20110804