Get Faster Code Faster! Intel® Advisor
Vectorization Optimization

Have you:
- Recompiled for AVX2 with little gain
- Wondered where to vectorize?
- Recoded intrinsics for new arch.?
- Struggled with compiler reports?

Data Driven Vectorization:
- What vectorization will pay off most?
- What's blocking vectorization? Why?
- Are my loops vector friendly?
- Will reorganizing data increase performance?
- Is it safe to just use pragma simd?

"Intel® Advisor’s Vectorization Advisor permitted me to focus my work where it really mattered. When you have only a limited amount of time to spend on optimization, it is invaluable."

Gilles Civario
Senior Software Architect
Irish Centre for High-End Computing
The Right Data At Your Fingertips
Get all the data you need for high impact vectorization

Filter by which loops are vectorized!
Trip Counts
What prevents vectorization?

Focus on hot loops
What vectorization issues do I have?
Which Vector instructions are being used?
How efficient is the code?

Get Faster Code Faster!

<table>
<thead>
<tr>
<th>Function/Cell Sizes and Loops</th>
<th>Vector Issues</th>
<th>Self Time(s)</th>
<th>Total Time(s)</th>
<th>Trip Counts</th>
<th>Loop Type</th>
<th>Why No Vectorization?</th>
<th>Vectorized Loops</th>
<th>Efficiency</th>
<th>Vector L...</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop at std::algohdu2470.in...</td>
<td></td>
<td>0.170s</td>
<td>0.170s</td>
<td></td>
<td>Scal</td>
<td>non-vectorizable loop in...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop at loopstlcpp2449.in.s...</td>
<td></td>
<td>0.170s</td>
<td>0.170s</td>
<td></td>
<td>Scal</td>
<td>non-vectorizable loop ins...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop at loopstlcpp2449.in.s...</td>
<td></td>
<td>0.150s</td>
<td>0.150s</td>
<td>12; 4</td>
<td>Vctorized</td>
<td>Vectorization possible but...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop at loopstlcpp2449.in.s...</td>
<td></td>
<td>0.000s</td>
<td>0.000s</td>
<td></td>
<td>Remainder</td>
<td>Vectorization possible but...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop at loopstlcpp2449.in.s...</td>
<td></td>
<td>0.000s</td>
<td>0.000s</td>
<td></td>
<td>Scal</td>
<td>Vectorization possible but...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop at loopstlcpp2399.in.s...</td>
<td></td>
<td>0.160s</td>
<td>0.160s</td>
<td>12</td>
<td>Expand</td>
<td>Expand</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop at loopstlcpp3381.in.s...</td>
<td></td>
<td>0.150s</td>
<td>0.150s</td>
<td>12</td>
<td>Expand</td>
<td>Expand</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop at loopstlcpp3381.in.s...</td>
<td></td>
<td>0.150s</td>
<td>0.150s</td>
<td>12</td>
<td>Expand</td>
<td>Expand</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop at loopstlcpp3381.in.s...</td>
<td></td>
<td>0.150s</td>
<td>0.150s</td>
<td>12</td>
<td>Expand</td>
<td>Expand</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4 Steps to Efficient Vectorization

Intel® Advisor – Vectorization Advisor

1. Compiler diagnostics + Performance Data + SIMD efficiency information

Function Call Site C/C++ Compile Time Test Time Compiler Vectorization

1. Loop diagnoses
   - Source: Code optimization
   - Test: Performance data
   - Compiler: SIMD efficiency

2. Guidance: detect problem and recommend how to fix it

Issue: Peeled/Remainder loops present

All or some source loops are not executing in the kernel loop. Improve performance by moving source loop iterations from peeled/remainder loops to the kernel loop. Read more at Vector Essentials, Utilizing Full Vectors.

Recommendation: Align memory access

Projected maximum performance gain: High
Projection confidence: Medium

The compiler creates a peeled loop because one of the memory accesses in the source loop does not start at a data boundary. Align the memory access and tell the compiler your memory access is aligned. Example: Aligns memory using a 32-byte boundary:

```c
float ***array;
array = (float **)%_malloc(ARRAY_SIZE*SIZE); // test wide loop
```

3. Loop-Carried Dependency Analysis

Problems and Messages

<table>
<thead>
<tr>
<th>ID</th>
<th>Type</th>
<th>Site Name</th>
<th>Sources</th>
<th>Modules</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Parallel site information</td>
<td>site2</td>
<td>digit2.cpp</td>
<td>digit2</td>
<td>Not a problem</td>
</tr>
<tr>
<td>P2</td>
<td>Read after write dependency</td>
<td>site2</td>
<td>digit2.cpp</td>
<td>digit2</td>
<td>New</td>
</tr>
<tr>
<td>P3</td>
<td>Read after write dependency</td>
<td>site2</td>
<td>digit2.cpp</td>
<td>digit2</td>
<td>New</td>
</tr>
<tr>
<td>P4</td>
<td>Write after write dependency</td>
<td>site2</td>
<td>digit2.cpp</td>
<td>digit2</td>
<td>New</td>
</tr>
<tr>
<td>P5</td>
<td>Write after write dependency</td>
<td>site2</td>
<td>digit2.cpp</td>
<td>digit2</td>
<td>New</td>
</tr>
<tr>
<td>P6</td>
<td>Write after read dependency</td>
<td>site2</td>
<td>digit2.cpp</td>
<td>digit2</td>
<td>New</td>
</tr>
<tr>
<td>P7</td>
<td>Write after read dependency</td>
<td>site2</td>
<td>digit2.cpp</td>
<td>digit2</td>
<td>New</td>
</tr>
</tbody>
</table>

4. Memory Access Patterns Analysis

<table>
<thead>
<tr>
<th>ID</th>
<th>Stride</th>
<th>Type</th>
<th>Source</th>
<th>Modules</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>P22</td>
<td>0:0</td>
<td>Unit stride</td>
<td>runCpuLoops.cpp:637</td>
<td>intel.exe</td>
<td></td>
</tr>
</tbody>
</table>
1. Compiler diagnostics + Performance
   Data + SIMD efficiency information

<table>
<thead>
<tr>
<th>Function Call Site and Logic</th>
<th>Self Time</th>
<th>Test Time</th>
<th>Compiler Vectorization</th>
<th>Loop Type</th>
<th>Why No Vectorization?</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop in std::complextimeofday()</td>
<td>1.56 MB</td>
<td>0.01 MB</td>
<td>Fails</td>
<td>Suboptimal</td>
<td>inside loop is already vectorized</td>
</tr>
<tr>
<td>loop in std::complextimeofday()</td>
<td>5.16 MB</td>
<td>5.16 MB</td>
<td>Fails</td>
<td>Suboptimal</td>
<td>inside loop is already vectorized</td>
</tr>
<tr>
<td>loop in std::complextimeofday()</td>
<td>5.16 MB</td>
<td>5.16 MB</td>
<td>Fails</td>
<td>Suboptimal</td>
<td>inside loop is already vectorized</td>
</tr>
<tr>
<td>loop in std::complextimeofday()</td>
<td>5.16 MB</td>
<td>5.16 MB</td>
<td>Fails</td>
<td>Suboptimal</td>
<td>inside loop is already vectorized</td>
</tr>
<tr>
<td>loop in std::complextimeofday()</td>
<td>5.16 MB</td>
<td>5.16 MB</td>
<td>Fails</td>
<td>Suboptimal</td>
<td>inside loop is already vectorized</td>
</tr>
<tr>
<td>loop in std::complextimeofday()</td>
<td>5.16 MB</td>
<td>5.16 MB</td>
<td>Fails</td>
<td>Suboptimal</td>
<td>inside loop is already vectorized</td>
</tr>
<tr>
<td>loop in std::complextimeofday()</td>
<td>5.16 MB</td>
<td>5.16 MB</td>
<td>Fails</td>
<td>Suboptimal</td>
<td>inside loop is already vectorized</td>
</tr>
<tr>
<td>loop in std::complextimeofday()</td>
<td>5.16 MB</td>
<td>5.16 MB</td>
<td>Fails</td>
<td>Suboptimal</td>
<td>inside loop is already vectorized</td>
</tr>
</tbody>
</table>
Is Most Execution in the Fast Part of the Vector?

Intel Advisor shows you

Optimization Notice

Copyright © 2016, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
Vector Efficiency: All The Data In One Place

My “performance thermometer”

Survey: Find out if your code is “under vectorized” and why

- Auto-vectorization: affected <3% of code
  - With moderate speed-ups
- First attempt to simply put #pragma simd:
  - Introduced slow-down
- Look at Vector Issues and Traits to find out why
  - All kinds of “memory manipulations”
  - Usually an indication of “bad” access pattern

Achieved Efficiency

Original (scalar) code efficiency. Corresponds to 1x speed-up.

Upper bound: 100% efficiency 4x gain (VL=4)

Achieved Efficiency

Original (scalar) code efficiency. Corresponds to 1x speed-up.

Upper bound: 100% efficiency 4x gain (VL=4)
1. Compiler diagnostics + Performance Data + SIMD efficiency information

<table>
<thead>
<tr>
<th>Function Call Used in Loop</th>
<th>Self Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(loop in another function)</td>
<td>126.3 µs</td>
<td>126.3 µs</td>
</tr>
<tr>
<td>(loop in another function)</td>
<td>126.3 µs</td>
<td>126.3 µs</td>
</tr>
<tr>
<td>(loop in another function)</td>
<td>126.3 µs</td>
<td>126.3 µs</td>
</tr>
<tr>
<td>(loop in another function)</td>
<td>126.3 µs</td>
<td>126.3 µs</td>
</tr>
<tr>
<td>(loop in another function)</td>
<td>126.3 µs</td>
<td>126.3 µs</td>
</tr>
</tbody>
</table>

2. Guidance: detect problem and recommend how to fix it

- **Issue:** Peeled (Remainder loops) present
  - All or some source loop iterations are not executing in the kernel loop. Improve performance by moving source loop iterations from peeled remainder loops to the kernel loop. Read more at Vector Essentials.
  - Using Full Vectors...

- **Recommendation:** Align memory access

  **Projected maximum performance gain:** High

  **Projection confidence:** Medium

  The compiler created a peeled loop because one of the memory accesses in the source loop does not start at a data boundary. Align the memory access and tell the compiler your memory access is aligned. This example aligns memory using a 32-byte boundary:

  ```c
  float *array = (float *) malloc(ALIGNED_SIZE(sizeof(float), 32));
  // Somewhere else
  __assume_aligned(array, 32);
  // Use array in loop
  ```
Get Specific Advice For Improving Vectorization
Intel® Advisor – Vectorization Advisor

Advisor shows hints to move iterations to vector body.
Critical Data Made Easy
Loop Trip Counts

Knowing the time spent in a loop is not enough!

Check actual trip counts
Loop is iterating 101 times but called > million times
Since the loop is called so many times it would be a big win if we can get it to vectorize.
1. Compiler diagnostics + Performance Data + SIMD efficiency information

2. Guidance: detect problem and recommend how to fix it

3. Loop-Carried Dependency Analysis
Is It Safe to Vectorize?

Loop-carried dependencies analysis verifies correctness

Select loop for Correct Analysis and press play!

Vector Dependence prevents Vectorization!
Correctness – Is It Safe to Vectorize?

Loop-carried dependencies analysis

Received recommendations to force vectorization of a loop:

1. Mark-up loop and check for REAL dependencies
2. Explore dependencies with code snippets

In this example 3 dependencies were detected:

- RAW – Read After Write
- WAR – Write After Read
- WAW – Write After Write

This is NOT a good candidate to force vectorization!
1. Compiler diagnostics + Performance Data + SIMD efficiency information

2. Guidance: detect problem and recommend how to fix it

3. Loop-Carried Dependency Analysis

4. Memory Access Patterns Analysis
Improve Vectorization

Memory Access pattern analysis

2.2 Check Memory Access Patterns
Identify and explore complex memory accesses for marked loops. Fix the reported problems.

Run Memory Access Patterns analysis, just to check how memory is used in the loop and the called function
Find vector optimization opportunities
Memory Access pattern analysis

All memory accesses are uniform, with zero unit stride, so the same data is read in each iteration.

We can therefore declare this function using the omp syntax:

```cpp
#pragma omp declare simd uniform(x0)
```

Stride distribution
Quickly Find Loops with Non-optimal Stride
Memory Access pattern analysis

- Quickly identify loops that are good, bad or mixed.
- Unit stride memory accesses are preferable.
- Find unaligned data
Roofline model: Am I bound by VPU/CPU or by Memory?

Optimization Notice
Copyright © 2016, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

What makes loops A, B, C different?
Roofline ingredient #1

a. FLOPS and b. Memory throughput peaks

- DP FMA ~35 Peak GFLOP/sec
- L1 per core 255 Gb/sec
- DRAM per core 10 Gb/sec

Peaks ("Roofs") obtained via benchmarking given system (highly optimized benchmarks)
Roofline ingredient #2
Axis Y: FLOP/S data

Y coordinate = FLOP/S measured for given point (loop or function)
Roofline ingredient #3: Axis X: Arithmetic Intensity (AI)

Putting
- memory utilization/demand
- CPU utilization altogether

\[ AI = \frac{\text{# FLOP}}{\text{# BYTE}} \]

What makes loops A, B, C different?
3 ingredients => Intel Advisor Roofline automation

1. Roofs (benchmark-based)
2. FLOP/S (AVX-512 mask aware)
3. AI (Cumulative traffic)

Each point corresponds to some loop or function with FP.
Size (and color) of point = Time spent in loop/function
Roofline Automation in Intel Advisor 2017+
answer the questions

1. How far are we from peak?
2. Can we do any better?
3. Where we are? What are the limiting factors?
   - Memory subsystem?
   - Lack of CPU/Vectorization/Threading?
   - Both? Anything else?

- Interactive mapping to source and performance profile
- Synergy between Vector Advisor and Roofline: FMA example
- Customizable chart
FLOP/s data measurement

1. **Seconds** are given by Survey run

2. **#FLOP** is currently given by “Trip Counts” run
   - Additionally provides cumulative memory traffic and (AVX-512) **Mask Register Utilization profile**

Works from Nehalem to KNL, technology mostly invariant to target platform

- Do not depend on PMU capabilities
- Good FLOPS/Mask PMU doesn't exist for KNL

Currently mapped to loops, functions, workload
Cache-Aware vs. Classic Roofline

\[ \text{AI} = \frac{\# \text{ FLOP}}{\# \text{ BYTE}} \]

- **AI_DRAM** (often referred to as Operational Intensity)

  \[ = \frac{\# \text{ FLOP}}{\# \text{ BYTES}} \text{ (CPU & Cache }\leftrightarrow\text{ DRAM)} \]
  
  - “DRAM traffic”-based
  
  - Variable for the same code/platform (varies with dataset size/trip count)
  
  - Can be measured relative to different memory hierarchy levels – cache level, HBM, DRAM

- **AI_CARM**

  \[ = \frac{\# \text{ FLOP}}{\# \text{ BYTES}} \text{ (CPU }\leftrightarrow\text{ Memory Sub-system)} \]
  
  - “Algorithmic”, “Cumulative (L1+L2+LLC+DRAM) traffic-based”
  
  - Invariant for the given code on given platform
  
  - Typically AI_CARM < AI_DRAM
Acknowledgments/References

Classic Roofline formulated by Williams, Waterman, Patterson, (Berkeley)

http://www.eecs.berkeley.edu/~waterman/papers/roofline.pdf

“Cache-aware Roofline model: Upgrading the loft” (Illic, Pratas, Sousa, INESC-ID/IST, Thec Uni of Lisbon)
http://www.inesc-id.pt/ficheiros/publicacoes/9068.pdf

Done by Intel Advisor and PathFinding Teams, Roman Belenov, Igor Kaleturin, Julia Fedorova, Zakhar Matveev

in collaboration with Philippe Thierry and his colleagues.

Some implementation aspects were inspired by Intel SDE and Hugh Caffey M

To Register for Advisor “Roofline” Alpha Evaluation:
Send request to vector_advisor@intel.com
Intel® Parallel Studio XE
Faster code faster!

Intel Advisor is part of Intel Parallel Studio XE:

**Vectorizing Compiler**
Squeeze all the performance out of the latest instruction set

**Threaded Performance Libraries**
Pre-vectorized, pre-threaded, pre-optimized

**High Level Parallel Models**
Productive solutions for thread, process & vector parallelism

**Parallel Performance Profilers**
Quickly discover bottlenecks and tune for high performance

**Thread Debugger**
Find and debug non-deterministic threading errors

**Vectorization Optimization and Thread Prototyping**
Data driven design tools help you vectorize & thread effectively

---

**Download Today**

Google:
“Intel Parallel Studio”

Or go directly to:

---

Copyright © 2016, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
Additional Resources

All links start with: https://software.intel.com/


**Explicit Vector Programming in Fortran:**
https://software.intel.com/articles/explicit-vector-programming-in-fortran


**Beta Registration & Download:**  https://software.intel.com/en-us/articles/intel-parallel-studio-xe-2016-beta

**For Intel® Xeon Phi™ coprocessors, but also applicable:**

**Intel® Composer XE User and Reference Guides:**
https://software.intel.com/compiler_15.0_ug_c
https://software.intel.com/compiler_15.0_ug_f

**Compiler User Forums:**  http://software.intel.com/forums
Legal Disclaimer & Optimization Notice

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.

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.

Copyright © 2015, 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.

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

Notice revision #20110804
### Platform Hardware and Software Configuration

<table>
<thead>
<tr>
<th>Platform</th>
<th>Core Frequency</th>
<th>Cores/Socket</th>
<th>Num Sockets</th>
<th>L1 Data Cache</th>
<th>L1 I Cache</th>
<th>L2 Cache</th>
<th>L3 Cache</th>
<th>Memory</th>
<th>Memory Frequency</th>
<th>Memory Access</th>
<th>H/W Prefetchers Enabled</th>
<th>HT Enabled</th>
<th>Turbo Enabled</th>
<th>C States</th>
<th>O/S Name</th>
<th>Operating System</th>
<th>Compiler Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Xeon™ P4710 Processor</td>
<td>3.0 GHZ</td>
<td>4</td>
<td>2</td>
<td>32K</td>
<td>32K</td>
<td>12 MB</td>
<td>None</td>
<td>32 GB</td>
<td>800 MHZ</td>
<td>UMA</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Disable d</td>
<td>Fedora</td>
<td>3.11.10-301.fc20</td>
<td>icc version 14.0.1</td>
</tr>
<tr>
<td>Intel® Xeon™ P5570 Processor</td>
<td>2.93 GHZ</td>
<td>4</td>
<td>2</td>
<td>32K</td>
<td>32K</td>
<td>256K</td>
<td>8 MB</td>
<td>48 GB</td>
<td>1333 MHZ</td>
<td>NUMA</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Disable d</td>
<td>Fedora</td>
<td>3.11.10-301.fc20</td>
<td>icc version 14.0.1</td>
</tr>
<tr>
<td>Intel® Xeon™ P5680 Processor</td>
<td>3.33 GHZ</td>
<td>6</td>
<td>2</td>
<td>32K</td>
<td>32K</td>
<td>256K</td>
<td>12 MB</td>
<td>48 MB</td>
<td>3200 MHZ</td>
<td>NUMA</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Disable d</td>
<td>Fedora</td>
<td>3.11.10-301.fc20</td>
<td>icc version 14.0.1</td>
</tr>
<tr>
<td>Intel® Xeon™ E5 2690 Processor</td>
<td>2.9 GHZ</td>
<td>8</td>
<td>2</td>
<td>32K</td>
<td>32K</td>
<td>256K</td>
<td>20 MB</td>
<td>64 GB</td>
<td>1600 MHZ</td>
<td>NUMA</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Disable d</td>
<td>Fedora</td>
<td>3.11.10-301.fc20</td>
<td>icc version 14.0.1</td>
</tr>
<tr>
<td>Intel® Xeon™ E5 2697v2 Processor</td>
<td>2.7 GHZ</td>
<td>12</td>
<td>2</td>
<td>32K</td>
<td>32K</td>
<td>256K</td>
<td>30 MB</td>
<td>64 GB</td>
<td>1867 MHZ</td>
<td>NUMA</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Disable d</td>
<td>Fedora</td>
<td>3.11.10-301.fc20</td>
<td>icc version 14.0.1</td>
</tr>
</tbody>
</table>

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](http://www.intel.com/performance)