SUPPORTING OPTIMIZATION WITH INTEL® ADVISOR

Dr. Michael Klemm
Software and Services Group
Developer Relations Division
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 © 2017, 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
Finding Vectorization Candidates

**Hard way:** Stare hard and long enough at the code

**Easier way:** Augment code with logging/profiling code

**Right way:** Use an external profiling tool to measure
5 Steps to Efficient Vectorization - Vector Advisor
(part of Intel® Advisor, Parallel Studio, Cluster Studio 2016 and beyond)

1. Compiler diagnostics + Performance Data + SIMD efficiency information

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

3. "Accurate" Trip Counts + FLOPs: understand utilization, parallelism granularity & overheads

4. Loop-Carried Dependencies Analysis

5. Memory Access Patterns Analysis

Optimization Notice

Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
Factors that prevent Vectorizing your code

1. Loop-carried dependencies

```c
DO I = 1, N
   A(I+1) = A(I) + B(I)
ENDDO
```

1. A Pointer aliasing (compiler-specific)

```c
void scale(int *a, int *b)
{
   for (int i = 0; i < 1000; i++)
      b[i] = z * a[i];
}
```

2. Function calls (incl. indirect)

```c
for (i = 1; i < nx; i++)
{
    x = x0 + i * h;
    sumx = sumx + func(x, y, xp);
}
```

3. Loop structure, boundary condition

```c
struct _x { int d; int bound; };
void doit(int *a, struct _x *x)
{
   for(int i = 0; i < x->bound; i++)
      a[i] = 0;
}
```

4. Outer vs. inner loops

```c
for(i = 0; i <= MAX; i++)
{
   for(j = 0; j <= MAX; j++)
   {
      D[i][j] += 1;
   }
}
```

5. Cost-benefit (compiler specific..)

And others......
Factors that slow-down your Vectorized code

1. A. Indirect memory access

```c
for (i=0; i<N; i++)
    A[B[i]] = C[i]*D[i]
```

1. B Memory sub-system Latency / Throughput

```c
void scale(int *a, int *b)
{
    for (int i = 0; i < VERY_BIG; i++)
        c[i] = z * a[i][j];
    b[i] = z * a[i];
}
```

2. Serialized or “sub-optimal” function calls

```c
for (i = 1; i < nx; i++) {
    sumx = sumx +
    serialized_func_call(x, y, xp);
}
```

3. Small trip counts not multiple of VL

```c
void doit(int *a, int *b, int unknown_small_value)
{
    for(int i = 0; i < unknown_small_value; i++)
        a[i] = z*b[i];
}
```

4. Branchy codes, outer vs. inner loops

```c
for(i = 0; i <= MAX; i++) {
    if ( D[i] < N)
        do_this(D);
    else if (D[i] > M)
        do_that();
    //...
}
```

5. MANY others: spill/fill, fp accuracy trade-offs, FMA, DIV/SQRT, Unrolling, even AVX throttling..
Vectorization Analysis Workflow

1. Run Survey Analysis
2. Check Trip Counts
3. Check Dependencies
4. Check Memory Access Patterns

Data "Refinement"
Mark for Deeper Analysis
3. Run Dependencies Analysis
3. Run MAP Analysis

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others.
Using Advisor and SIMD methodology

1. **Characterize** your code (e.g., scalar vs. vector, efficiency). **Focus** on most impactful parts.
   - Scalar Loops
   - SIMD Loops

2. **Explore root cause preventing** (compilers) from Vectorization. Implement low-hanging fix.
   - Localize memory/memory-access-bound cases.

3. **Check if Dependencies** are real. Resolve dependencies.

2. **Root cause vectorized code slow-down factors. Implement low-hanging fix.**
   - Memory-bound loops
   - SIMD Loops

3. **Explore Memory Access Pattern and Memory Locality. Refactor for Memory Layout.**

**Done with all low-hanging impactful parts of your code?**
- yes
  - Scalar Loops
  - Memory-bound loops
  - SIMD Loops
- no
  - Scalar Loops
  - SIMD Loops

*Other names and brands may be claimed as the property of others.*
Using Advisor to address these factors

1. Run **Advisor Survey** Analysis to quickly **characterize** your code and **focus** on most impactful parts.

   - **Scalar Loops**
   - **SIMD Loops**

2. Use **Survey** to find **Vectorizable loops** and get tips **how to vectorize**

3. **Dependencies Analysis:** Check if Dependencies are real. Resolve dependencies.

3. **MAP Analysis:** Root cause **SIMD and Memory slow-down. Refactor** for better memory layout.

2. Use **Survey/TripCounts** and **Recommendations** to root cause vectorized code slow-down factors. Get tips for speed-up.

   - **2.1 Use TripCounts/FLOPs/Roofline** data to localize memory/memory-access-bound cases*

Done with all **low-hanging impactful** loops? **Need more data?**

   - yes
   - no

   - **Scalar Loops**
   - **Memory-bound loops**
   - **SIMD Loops**
Using Advisor to address these factors

1. Run Advisor Survey Analysis to quickly characterize your code: scalar vs. vector, vector efficiency.

2. Use Survey to find Vectorizable loops and get tips on how to vectorize.
3. Dependencies Analysis: Check if Dependencies are real. Resolve dependencies.

2.1 Use TripCounts/FLOPs/Roofline data to basically localize memory/memory-access-bound cases.

3. MAP Analysis: Root cause SIMD and Memory slow-down. Refactor for better memory layout.

Done with all low-hanging impactful loops? Need more data?

Yes

Scalar Loops

Memory-bound loops

SIMD Loops

No

Scalar Loops

SIMD Loops
Quickly **characterize** the efficiency of your code:

Advisor Summary.

Use the summary view to quickly characterize your program.

Time in Scalar vs. Vector loops.

SIMD Efficiency.

Focus on Hottest kernels.
Advisor Survey: **Focus + Characterize.**

One stop shop. Get all the data you need for high impact vectorization

- Filter by which loops are vectorized!
- What prevents vectorization?
- Focus on hot loops
- What vectorization issues do I have?
- Which Vector instructions are being used?
- How efficient is the code?
Advisor Survey: **Focus + Characterize.**

**Focus and order non-vectorized loops**

- “Compiler diagnostics” sub-tab to explain root cause
Advisor Survey: **Focus + Characterize.**

Focus and order vectorized loops

- **Efficiency** – my performance thermometer
- **Recommendations** – get tip on how to improve performance
  - (also apply to scalar loops)

### Function Call Sites and Loops

<table>
<thead>
<tr>
<th>Vectorized Loops</th>
<th>Efficiency</th>
<th>Gain</th>
<th>VL Traits</th>
<th>Data T.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVX</td>
<td>1.07x</td>
<td>8</td>
<td>FMA</td>
<td>Float32</td>
</tr>
<tr>
<td>AVX2</td>
<td>1.05x</td>
<td>8</td>
<td>FMA</td>
<td>Float32</td>
</tr>
<tr>
<td>AVX3</td>
<td>1.05x</td>
<td>8</td>
<td>FMA</td>
<td>Float32</td>
</tr>
</tbody>
</table>

### Issue

- **Assumed dependency present**

### Problem Description

All or some `source` loop iterations are not executing in the `loop body`. Improve performance by moving source loop iterations inside the loop body.

### Recommendation

- **Add data padding**

  The `trip count` is not a multiple of `vector length`. To fix, do one of the following:
  - Increase the size of objects and add iterations so the trip count is a multiple of vector length.
  - Increase the size of static and automatic objects, and use a compiler option to add padding.

**Windows® OS**

```
-QOPT-ASSUME-SAFE-PADDING
```

**Linux® OS**

```
-QOPT-ASSUME-SAFE-PADDING
```

**Note:** These compiler options apply only to Intel® Many Integrated Core Architecture (Intel® MIC Architecture) and some compiler options do not add padding for static and automatic objects. To satisfy this assumption, you must increase the size of static and automatic objects in your application.

**Optional:** Specify the trip count, if it is not constant, using a `directive`:

```
#pragma loop_count
```

**Read More:**

- `#pragma loop_count`
Vector Efficiency: my “performance thermometer”

<table>
<thead>
<tr>
<th>Vector Issues</th>
<th>Self Time</th>
<th>Total Time</th>
<th>Type</th>
<th>Why No Vectorization?</th>
<th>Vectorized Loops</th>
<th>Trip Counts</th>
<th>Instruction Set Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inefficient</td>
<td>5.281s</td>
<td>5.281s</td>
<td>Vectorized (Both)</td>
<td>AVX</td>
<td>55%</td>
<td>2.19x</td>
<td>4</td>
</tr>
<tr>
<td>Ineffective</td>
<td>2.828s</td>
<td>2.828s</td>
<td>Vectorized (Both)</td>
<td>AVX</td>
<td>91%</td>
<td>3.65x</td>
<td>4</td>
</tr>
</tbody>
</table>

-91%: Achieved Vectorization Efficiency

Achieved Vectorization Efficiency = (Estimated Gain/Vector Length) * 100%
Estimated Gain = 3.65x
Vector Length = 4

Orange color = Achieved vectorization efficiency is higher than reference efficiency for original scalar loop

Efficiency is approximately 91%, which means actual efficiency may be lower

(25%): Reference Efficiency for original scalar loop
Reference Efficiency = (1x/Vector Length) * 100%

(100%): Theoretical Maximum Vectorization Efficiency
Maximum Vectorization Efficiency = (Theoretical Maximum Gain/Vector Length) * 100%
Theoretical Maximum Gain = Currently selected Vector Length = 4

-91%: Achieved Vectorization Efficiency

Achieved Vectorization Efficiency = (Estimated Gain/Vector Length) * 100%
Estimated Gain = 3.65x
Vector Length = 4

Orange color = Achieved vectorization efficiency is higher than reference efficiency for original scalar loop

Efficiency is approximately 91%, which means actual efficiency may be lower

(25%): Reference Efficiency for original scalar loop
Reference Efficiency = (1x/Vector Length) * 100%

(100%): Theoretical Maximum Vectorization Efficiency
Maximum Vectorization Efficiency = (Theoretical Maximum Gain/Vector Length) * 100%
Theoretical Maximum Gain = Currently selected Vector Length = 4

-91%: Achieved Vectorization Efficiency

Achieved Vectorization Efficiency = (Estimated Gain/Vector Length) * 100%
Estimated Gain = 3.65x
Vector Length = 4

Orange color = Achieved vectorization efficiency is higher than reference efficiency for original scalar loop

Efficiency is approximately 91%, which means actual efficiency may be lower

(25%): Reference Efficiency for original scalar loop
Reference Efficiency = (1x/Vector Length) * 100%

(100%): Theoretical Maximum Vectorization Efficiency
Maximum Vectorization Efficiency = (Theoretical Maximum Gain/Vector Length) * 100%
Theoretical Maximum Gain = Currently selected Vector Length = 4

Intel microprocessors. Performance tests, such as SYSmark and MobileMark, changes 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. See FTC disclaimer.
Data Dependencies – Tough Problem #1
Is it safe to force the compiler to vectorize?

DO I = 1, N  
   A(I) = A(I-1) * B(I)  
ENDDO

or

void scale(int *a, int *b)  
{  
   for (int i = 0; i < 1000; i++)  
      b[i] = z * a[i];  
}

**Issue: Assumed dependency present**
The compiler assumed there is an anti-dependency (Write after read – WAR) or true dependency (Read after write – RAW) in the loop. Improve performance by investigating the assumption and handling accordingly.

- **Enable vectorization**
  Potential performance gain: Information not available until Beta Update release
  Confidence this recommendation applies to your code: Information not available until Beta Update release

The Correctness analysis shows there is no real dependency in the loop for the given workload. Tell the compiler it is safe to vectorize using the restrict keyword or a directive.

<table>
<thead>
<tr>
<th>ICL/ICC/ICPC Directive</th>
<th>IFORT Directive</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma simd or #pragma omp simd</td>
<td>IDIRS SIMD or ISOMP SIMD</td>
<td>Ignores all dependencies in the loop</td>
</tr>
<tr>
<td>#pragma ivdep</td>
<td>IDIRS IVDEP</td>
<td>Ignores only vector dependencies (which is safest)</td>
</tr>
</tbody>
</table>

Read More:
- User and Reference Guide for the Intel C++ Compiler 15.0 > Compiler Reference > Pragmas > Intel-specific
  Pragma Reference >
  - ivdep
  - omp simd
Check if it is safe to vectorize with Advisor Dependencies

Select loop for Dependency Analysis and press play!

Vector Dependence prevents Vectorization!
Why is my code so slow?

![Intel Advisor XE 2017](image)

**Where should I add vectorization and/or threading parallelism?**

<table>
<thead>
<tr>
<th>Function Call Sites and Loops</th>
<th>Vector Issues</th>
<th>Self Time</th>
<th>Total Time</th>
<th>Type</th>
<th>Why No Vectorization?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(loop in matvec at Multiply.c72)</td>
<td>1 Inefficient</td>
<td>5.281s</td>
<td>5.281s</td>
<td>Vectorized (Bo...</td>
<td></td>
</tr>
<tr>
<td>(loop in matvec at Multiply.c66)</td>
<td>1 Inefficient</td>
<td>2.828s</td>
<td>2.828s</td>
<td>Vectorized (Bo...</td>
<td></td>
</tr>
<tr>
<td>(loop in matvec at Multiply.c49)</td>
<td>1 Inefficient</td>
<td>0.531s</td>
<td>5.812s</td>
<td>Scalar</td>
<td></td>
</tr>
<tr>
<td>(loop in matvec at Multiply.c49)</td>
<td>1 Assumed d...</td>
<td>0.516s</td>
<td>3.344s</td>
<td>Scalar</td>
<td></td>
</tr>
<tr>
<td>(loop in matvec at Multiply.c85)</td>
<td>1 Assumed d...</td>
<td>0.063s</td>
<td>0.063s</td>
<td>Scalar</td>
<td>vector dependence...</td>
</tr>
<tr>
<td>(loop in main at Driver.c151)</td>
<td>1 Assumed d...</td>
<td>0.016s</td>
<td>9.297s</td>
<td>Scalar</td>
<td>loop with function...</td>
</tr>
<tr>
<td>(loop in matvec at Multiply.c49)</td>
<td>1 Inefficient</td>
<td>0.000s</td>
<td>0.000s</td>
<td>Scalar</td>
<td></td>
</tr>
</tbody>
</table>

**Optimization Notice**

Copyright © 2017, Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.*
Am I bound by VPU/CPU or by Memory?: ROOFLINE ANALYSIS

New dev: Roofline as a function of time

An "extreme" application doing both Stream and HPL would have WRONG averaged scalar AI. Dynamic Roofline helps to identify such different phases.
Am I bound by VPU/CPU or by Memory?:
ROOFLINE ANALYSIS
Am I bound by VPU/CPU or by Memory?: Advisor ROOFLINE ANALYSIS

Advisor Roofline Planned:
• CARM
• Classic
• Mask-aware

Interactively mapped to other data sources