Intel® Advisor
Vectorization Optimization and Thread Prototyping

Munara Tolubaeva
Software Technical Consulting Engineer
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 © 2016, 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.

Notice revision #20110804
Have you:

- Tried threading an app, but seen little performance benefit?
- Hit a “scalability barrier”? Performance gains level off as you add cores?
- Delayed a release that adds threading because of synchronization errors?

Breakthrough for threading design:

- Quickly prototype multiple options
- Project scaling on larger systems
- Find synchronization errors before implementing threading
- Separate design and implementation - Design without disrupting development

Add Parallelism with Less Effort, Less Risk and More Impact

http://intel.ly/advisor-xe
Have you:

- Recompiled with AVX2, but seen little benefit?
- Wondered where to start adding vectorization?
- Recoded intrinsics for each new architecture?
- Struggled with cryptic compiler vectorization messages?

**Breakthrough for vectorization design**

- What vectorization will pay off the most?
- What is blocking vectorization and why?
- Are my loops vector friendly?
- Will reorganizing data increase performance?
- Is it safe to just use pragma simd?
Factors that prevent Vectorizing your code

1. Loop-carried dependencies

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

2. Function calls (incl. indirect)

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

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[j][i] += 1;
    }
}
```

5. Cost-benefit (compiler specific..)

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

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. 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];
}
```

3. 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..
Intel® Advisor helps you increase performance!
Recommended 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.
   - Memory-bound loops
   - SIMD Loops


Done with all low-hanging impactful parts of your code?
5 Steps to Efficient Vectorization - Vector Advisor
(part of Intel® Advisor, Parallel Studio, Cluster Studio 2016)

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 Dependency Analysis

5. Memory Access Patterns Analysis
1. Compiler diagnostics + Performance
   Data + SIMD efficiency information

<table>
<thead>
<tr>
<th>Function Call</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::compress(compute_6_double_4x2) )</td>
<td>1.43s</td>
<td>0.83s</td>
<td>Compiler</td>
<td>Sub 1</td>
<td>vector dependence prevents vectorization.</td>
</tr>
<tr>
<td>(Loop in std::compress(compute_6_double_4x2) )</td>
<td>1.31s</td>
<td>0.74s</td>
<td>Compiler</td>
<td>Sub 2</td>
<td>inner loop was already vectorized.</td>
</tr>
<tr>
<td>(Loop in std::compress(compute_6_double_complex) )</td>
<td>6.86s</td>
<td>1.31s</td>
<td>Vectorized (C3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Vectorized: The loop is vectorized for improved performance.
- Non-vectorized: The loop is not vectorized due to vector incompatibilities or other reasons.
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 use?
- How efficient is the code?
All the data in one place

<table>
<thead>
<tr>
<th>Top Down</th>
<th>Source</th>
<th>Assembly</th>
<th>Recommendations</th>
<th>Compiler Diagnostics</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Function Call Sites and Loops</th>
<th>Total Time %</th>
<th>Total Time</th>
<th>Self Time</th>
<th>Loop Type</th>
<th>Why No Vectorization?</th>
<th>Vectorized Loops</th>
<th>Vector Length</th>
<th>Compiler Estimated Gain</th>
<th>Traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>100.0%</td>
<td>15.043s</td>
<td>0s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>func@0x0f82dadcf</td>
<td>100.0%</td>
<td>15.043s</td>
<td>0s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>func@0x0f82dacf0</td>
<td>100.0%</td>
<td>15.043s</td>
<td>0s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BaseThreadInitThunk</td>
<td>100.0%</td>
<td>15.043s</td>
<td>0s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_mainCRTStartup</td>
<td>100.0%</td>
<td>15.043s</td>
<td>0s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>99.9%</td>
<td>15.035s</td>
<td>0s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Loop at Driver.c:145]</td>
<td>99.9%</td>
<td>15.035s</td>
<td>0s</td>
<td>Scalar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>printf</td>
<td>0.0%</td>
<td>0.001s</td>
<td>0.0006s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>func@0x10150e0f0</td>
<td>0.1%</td>
<td>0.008s</td>
<td>0.0076s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Background on loop vectorization

A typical vectorized loop consists of

Main vector body
• Fastest among the three!

Optional peel part
• Used for the unaligned references in your loop. Uses Scalar or slower vector

Remainder part
• Due to the number of iterations (trip count) not being divisible by vector length. Uses Scalar or slower vector.

Larger vector register means more iterations in peel/remainder
• Make sure you Align your data!
• Make sure the number of iterations divisible by the vector length!
Efficiently Vectorize your code
Intel Advisor – Vectorization Advisor
1. Compiler diagnostics + Performance Data + SIMD efficiency information

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

- **Issue:** Pooled/Remainder loops present
  - All or some source loop iterations are not executing in the kernel loop. Improve performance by moving source loop iterations from pooled/remainder loops to the kernel loop. Read more at Vector Essentials, Utilizing Full Vectors.

- **Recommendation:** Align memory access
  - **Projection confidence:** Medium
  - The compiler created a pooled 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 array = malloc(ARRAY_SIZE* sizeof(float), 32);
// Somewhere else
__assume_aligned(array, 32);
// Use the array in loop
```
Get Specific Advice For Improving Vectorization
Intel® Advisor – Vectorization Advisor

Advisor shows hints to move iterations to vector body.
1. **Compiler diagnostics + Performance Data + SIMD efficiency information**

- Runtime Call Site and Debug Time:
  - Self Time
  - Total Time
  - Compiler Time

- Loop Type:
  - Performance
  - Data
  - SIMD efficiency information

- “Accurate” Trip Counts + FLOPs: understand utilization, parallelism granularity & overheads

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

- Issue: Pooled/Remainder loops are present.

- Recommendation: Align memory access.
  - Projected maximum performance gain: High
  - Projection confidence: Medium

- The compiler created a pooled loop because one of the memory accesses in the source loop does not start at a data boundary. Align the memory access and all the compiler’s memory access is aligned.
  - This example aligns memory using a 32-byte boundary.
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. “Accurate” Trip Counts + FLOPs: understand utilization, parallelism granularity & overheads

4. 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!
Data Dependencies – Tough Problem #1

Is it safe to force the compiler to vectorize?

Data dependencies

```c
for (i=0;i<N;i++) // Loop carried dependencies!
    A[i] = A[i-1]*C[i]; // Need the ability to check if it
                      // is safe to force the compiler
```

---

**Issue:** Assumed dependency present

The compiler assumed there is an anti-dependency (Write after read – WOR) 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><code>-DIR SIMD or ISOMP SIMD</code></td>
<td>Ignores all dependencies in the loop</td>
</tr>
<tr>
<td>#pragma ivdep</td>
<td><code>-DIR SIMD</code></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
Correctness – Is It Safe to Vectorize?

Loop-carried dependencies analysis

Received recommendations to force vectorization of a loop:

1. Mark-up the loop and check for the presence of REAL dependencies

2. Explore dependencies in more details 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. “Accurate” Trip Counts + FLOPs: understand utilization, parallelism granularity & overheads

4. Loop-Carried Dependency Analysis

5. Memory Access Patterns Analysis
Non-Contiguous Memory – Tough Problem #2

Potential to vectorize but may be inefficient

- Non-unit strided access to arrays

  ```c
  for (i=0; i<N; i+=2) //Incrementing “i” by 2 is not unit stride
      //We need a way to check how we are
      //accessing memory.
  ```

- Indirect reference in a loop

  ```c
  for (i=0; i<N; i++)
      A[B[i]] = C[i]*D[i]; //We have to decode B[i] to find out
      //which element of A to reference
  ```
Improve Vectorization

Memory Access pattern analysis

Select loops of interest

Run Memory Access Patterns analysis, just to check how memory is used in the loop and the called function.
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: `pragma omp declare simd uniform(x0`
Advisor 2017 Update 2 Features

1. Cache aware Roofline
2. Improved Trip Counts and FLOPS
   1. **Call Count** metric for functions
3. Filtering by module
4. Re-finalization
5. Dynamic Instruction Mixes
   1. -report survey -mix
Vectorized loops with high efficiency

Are we done??..
Vectorized loops with high efficiency

...It depends.

If code is not SIMD bound, then Speedup $\leq$ Vectorization Gain

In addition (instead of) VPU-bound code could be Memory Bound
Am I bound by VPU/CPU or by Memory?
Quick and Dirty check with Survey Loop Analytics.

The types of instructions in your loop will be a rough indicator of whether you are doing more memory or computational work.
Am I bound by VPU/CPU or by Memory?

ROOFLINE ANALYSIS

Better optimized – smaller potential (gap)

Big optimization gap
Roofline Automation in Intel® (Vectorization) Advisor 2017

- Interactive mapping to source and performance profile
- Synergy between Vector Advisor and Roofline: FMA example
- Customizable chart

Each Roof (slope) Gives peak CPU/Memory throughput of your PLATFORM (benchmarked)

Each Dot represents loop or function in YOUR APPLICATION (profiled)
Advisor Roofline: under the hood

Roofline application profile:

Axis Y: FLOP/S = #FLOP (mask aware) / #Seconds
Axis X: AI = #FLOP / #Bytes

Seconds
User-mode sampling
Root access not needed

Roofs
Microbenchmarks
Actual peak for the current configuration

#FLOP
Binary Instrumentation
Does not rely on CPU counters

Bytes
Binary Instrumentation
Counts operands size (not cachelines)
## Getting Roofline in Advisor

<table>
<thead>
<tr>
<th>FLOP/S = #FLOP/Seconds</th>
<th>Seconds</th>
<th>#FLOP Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>- Mask Utilization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- #Bytes</td>
</tr>
</tbody>
</table>

### Step 1: Survey
- Non intrusive. *Representative*
- Output: Seconds (+much more)

### Step 2: FLOPS
- Precise, instrumentation based
- Physically count Num-Instructions
- Output: #FLOP, #Bytes
Survey+FLOPs Report on AVX-512: FLOP/s, Bytes and AI, Masks and Efficiency
General efficiency (FLOPS) vs. VPU-centric efficiency (Vector Efficiency)

<table>
<thead>
<tr>
<th>Function Call Sites and Loops</th>
<th>Total Time</th>
<th>Type</th>
<th>Vectorized Loops</th>
<th>Gain...</th>
<th>VL...</th>
<th>Vector Issues</th>
<th>FLOPS And AVX-512 Mask Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vectorized (Remai..</td>
<td>44%</td>
<td>3.53x</td>
<td>8</td>
<td>0.847</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vectorized (Remai..</td>
<td>26%</td>
<td>5.79x</td>
<td>16; 8</td>
<td>3,666</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vectorized (Remai..</td>
<td>44%</td>
<td>3.53x</td>
<td>8</td>
<td>1,482</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vectorized (Remai..</td>
<td>23%</td>
<td>1,84x</td>
<td>8</td>
<td>0,768</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vectorized (Remai..</td>
<td>38%</td>
<td>3,05x</td>
<td>8</td>
<td>0,724</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vectorized (Remai..</td>
<td>24%</td>
<td>1,94x</td>
<td>8</td>
<td>1,529</td>
</tr>
</tbody>
</table>

High Vector Efficiency
Low FLOPS

Low Vector Efficiency
High FLOPS
Interpreting Roofline Data: advanced ROI analysis.

**Final Limits**
(assuming perfect optimization)
Long-term ROI, optimization strategy

**Current Limits**
(what are my current bottlenecks)
Next step, optimization tactics

- **Finally compute-bound**
  Invest more into effective CPU/VPU (SIMD) optimization

- **Finally memory-bound**
  Invest more into effective cache utilization

L1/L2 or CPU bound
LLC or CPU bound

Check your Advisor Survey and MAP results
Command line usage

- `advixe-cl -collect survey -project-dir ./your_project -no-auto-finalize -search-dir src=./srcPath -search-dir bin=./binPath -- ./yourExecutable`

- `advixe-cl -collect map -mark-up-list=10,12,15 -project-dir ./your_project -search-dir bin=./binPath -search-dir src=./srcPath -- ./yourExecutable`

- `advixe-cl -collect dependencies --project-dir ./your_project --loops="loop-height=0,total-time>2" -- ./yourExecutable`

- `advixe-cl -report survey -project-dir ./yourProject --search-dir src:r=./src`

- `advixe-cl -collect survey -module-filter-mode=include -module-filter=AnalyzeMyApp.exe,AnalyzeThisToo.dll -project-dir MyProject -- AnalyzeMyApp.exe`

- `advixe-cl -collect survey -module-filter-mode=exclude -module-filter=DoNotAnalyze.so -project-dir MyProject -- MyApplication`
Roofline access and how-to

• For 2017 Update 1
  (!) Requires env variable set before running command line or GUI:

  ```
  export ADVIXE_EXPERIMENTAL=roofline
  ```

• Starting from 2017 Update 2

  Just available by default
Roofline access and how-to command line example

(optional) > source advixe-vars.sh

(optional) > export ADVIXE_EXPERIMENTAL=roofline

> advixe-cl --collect survey --no-auto-finalize --project-dir ./your_project
   -- <your-executable-with-parameters>

> advixe-cl --collect tripcounts -flops-and-masks --project-dir ./
   your_project -- <your-executable-with-parameters>

> advixe-gui ./your_project