WHY DO WE CARE?
Motivation (instead of Agenda)

• Starting from 4.x, OpenMP introduces support for both levels of parallelism:
  • Multi-Core (think of “pragma/directive omp parallel for”)
  • SIMD (think of “pragma/directive omp simd”)
  • 2 pillars of OpenMP SIMD programming model

• Hardware with Intel ® AVX-512 support gives you theoretically 8x speed-up over SSE baseline (less or even more in practice)

• Intel® Advisor is here to assist you in:
  • Enabling SIMD parallelism with OpenMP (if not yet)
  • Improving performance of already vectorized OpenMP SIMD code
  • And will also help to optimize for Memory Sub-system (Advisor Roofline)
Don’t use a single Vector lane!
Un-vectorized and un-threaded software will underperform
Permission to Design for All Lanes

Threading and Vectorization needed to fully utilize modern hardware
Vector parallelism in x86

Intel® microarchitecture code name ...
(theoretically) 8x more SIMD FLOP/S compared to your (–O2) optimized baseline

- Significant leap to 512-bit SIMD support for processors
- Intel® Compilers and Intel® Math Kernel Library include AVX-512 support
- Strong compatibility with AVX
- Added EVEX prefix enables additional functionality

Don’t leave it on the table!
Two level parallelism decomposition with **OpenMP**: image processing example

```c
#pragma omp parallel for
for (int y = 0; y < ImageHeight; ++y){
    #pragma omp simd
    for (int x = 0; x < ImageWidth; ++x){
        count[y][x] = mandel(in_vals[y][x]);
    }
}
```
Two level parallelism decomposition with OpenMP: fluid dynamics processing example

```c
#pragma omp parallel for
for (int i = 0; i < X_Dim; ++i){
#pragma omp simd
    for (int m = 0; x < n_velocities; ++m){
        next_i = f(i, velocities(m));
        X[i] = next_i;
    }
}
```
Key components of Intel® Advisor

Step 1. Compiler diagnostics + Performance Data + SIMD efficiency information

Guidance: detect problem and recommend how to fix it

Step 2. “Precise” Trip Counts & FLOPs. Roofline analysis.
Characterize your application.

Step 3. Loop-Carried Dependency Analysis

Step 4. Memory Access Patterns Analysis

More Advise ("Recommendations") for OpenMP SIMD

What's new in "2019" release

• Roofline for INT OP/S
• Integrated Roofline (exp)
• Interactive(!) HTML export

• MAC OS viewer
• Python API..
<table>
<thead>
<tr>
<th>Vectorization</th>
<th>Threading</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;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.&quot;</td>
<td>&quot;Intel® Advisor has been extremely helpful in identifying the best pieces of code for parallelization. We can save several days of manual work by targeting the right loops and we can use Advisor to find potential thread safety issues to help avoid problems later on.&quot;</td>
</tr>
<tr>
<td>Gilles Civario</td>
<td>Carlos Boneti</td>
</tr>
<tr>
<td>Senior Software Architect</td>
<td>HPC software engineer, Schlumberger</td>
</tr>
<tr>
<td>Irish Centre for High-End Computing</td>
<td></td>
</tr>
<tr>
<td>“Intel® Advisor’s Vectorization Advisor fills a gap in code performance analysis. It can guide the informed user to better exploit the vector capabilities of modern processors and coprocessors.”</td>
<td>“Intel® Advisor has allowed us to quickly prototype ideas for parallelism, saving developer time and effort, and has already been used to highlight subtle parallel correctness issues in complex multi-file, multi-function algorithms.”</td>
</tr>
<tr>
<td>Dr. Luigi Iapichino</td>
<td>Simon Hammond</td>
</tr>
<tr>
<td>Scientific Computing Expert</td>
<td>Senior Technical Staff</td>
</tr>
<tr>
<td>Leibniz Supercomputing Centre</td>
<td>Sandia National Laboratories</td>
</tr>
</tbody>
</table>
TWO PILLARS OF OPENMP SIMD
Pillar 1: SIMD Pragma Notation

OpenMP 4.0: 
\#pragma omp simd [clause [,clause] ...]

- **Targets loops**
  - Can target inner or outer canonical loops

- **Developer asserts loop is suitable for SIMD**
  - The Intel Compiler will vectorize if possible (will ignore dependency or efficiency concerns)
  - Use when you **KNOW** that a given loop is safe to vectorize
  - Can choose from lexicon of clauses to modify behavior of SIMD directive

- **Developer should validate results (correctness)**
  - Just like for race conditions in OpenMP* threading loops

- **Minimizes source code changes needed to enforce vectorization**
OMP SIMD Pragma Clauses

reduction(operator:v1, v2, …)

- v1 etc are reduction variables for operation “operator”
- Examples include computing averages or sums of arrays into a single scalar value: `reduction (+:sum)`

linear(v1:step1, v2:step2, …)

- declares one or more list items to be private to a SIMD lane and to have a linear relationship with respect to the iteration space of a loop: `linear (i:2)`

safelen (length)

- no two iterations executed concurrently with SIMD instructions can have a greater distance in the logical iteration space than this value
- `Typical values are 2, 4, 8, 16`

simdlen(length)
OMP SIMD Pragma Clauses cont...

aligned(v1:alignment, v2:alignment)

- declares that the object to which each list item points is aligned to the number of bytes expressed in the optional parameter of the aligned clause.

collapse(number of loops)

- Nested loop iterations are collapsed into one loop with a larger iteration space.

private(v1, v2, ...), lastprivate (v1, v2, ...)

- declares one or more list items to be private to an implicit task or to a SIMD lane, lastprivate causes the corresponding original list item to be updated after the end of the region..
Pillar 2: SIMD-enabled functions

Write a function for one element and add `pragma` as follows

```c
#pragma omp declare simd
float foo(float a, float b, float c, float d)
{
    return a * b + c * d;
}
```

Call the scalar version:

```c
e = foo(a, b, c, d);
```

Call vector version via SIMD loop:

```c
#pragma omp simd
for(i = 0; i < n; i++) {
    A[i] = foo(B[i], C[i], D[i], E[i]);
}
```

```c
A[:] = foo(B[:], C[:], D[:], E[:]);
```
Many Ways to Vectorize

- **Use Performance Libraries**
  - (MKL, IPP)

- **Compiler:**
  - Auto-vectorization (no change of code)

- **Compiler:**
  - Auto-vectorization hints (#pragma vector, ...)

- **Explicit (user mandated) Vector Programming:**
  - OpenMP 4.x, Intel Cilk Plus

- **SIMD intrinsic class**
  - (e.g.: F32vec, F64vec, ...)

- **Vector intrinsic**
  - (e.g.: _mm_fmadd_pd(...), _mm_add_ps(...), ...)

- **Assembler code**
  - (e.g.: [v]addps, [v]addss, ...)

**Ease of use**

- implicit

- explicit

**Programmer control**

- instruction aware

- Cilk Plus Array Notation (CEAN)
  - (a[:] = b[:] + c[:])

- Use Performance Libraries
  - (MKL, IPP)
VECTORIZATION ANALYSIS WITH INTEL® ADVISOR
Factors that prevent Vectorizing your code

1. Loop-carried dependencies

```c
DO I = 1, N
   A(I + M) = 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[j][i] += 1;
   }
}
```

5. Cost-benefit (compiler specific..)

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

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

1. A. Indirect memory access

\[
\text{for (i=0; i<N; i++)}
\]
\[A[B[i]] = C[i]*D[i]\]

1. B Memory sub-system Latency / Throughput

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

2. Serialized or “sub-optimal” function calls

\[
\text{for (i = 1; i < nx; i++) }
\]
\[
\text{sumx = sumx + serialized_func_call(x, y, xp); }
\]

3. Small trip counts not multiple of VL

\[
\text{void doit(int *a, int *b, int unknown_small_value) }
\]
\[
\{ \\
\quad \text{for (int i = 0; i < unknown_small_value; i++)} \\
\quad \quad a[i] = z*b[i]; \\
\}\n\]

4. Branchy codes, outer vs. inner loops

\[
\text{for(i = 0; i <= MAX; i++) }
\]
\[
\text{if ( D[i] < N) do_this(D); }
\]
\[
\text{else if (D[i] > M) do_that(); }
\]
\[
\text{//…}
\]

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

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

Start
Edit & Compile
Take Snapshot
Deeper-dive analysis
(Mark-up Loops)
Use the same target binary within every cycle
Intel® Advisor Survey

Why no vectorization? How to improve AVX performance?

- **Efficiency** – my performance thermometer
- **Recommendations** – get tip on how to improve performance
  - (also apply to scalar loops)
Actionable advice: introducing OpenMP SIMD
INTEL® ADVISOR “MAP” AND DEPENDENCIES ANALYSIS
Memory access pattern analysis

How should I access data?

Unit stride access are faster

\[
\text{for } (i=0; \ i<N; \ i++) \\
A[i] = B[i]*d
\]

Constant stride are usually worse

\[
\text{for } (i=0; \ i<N; \ i+=2) \\
A[i] = B[i]*d
\]

Non predictable access are usually bad

\[
\text{for } (i=0; \ i<N; \ i++) \\
A[i] = B[C[i]]*d
\]

For B, 1 cache line load computes 4 DP

For B, 2 cache line loads compute 4 DP with reconstructions

For B, 4 cache line loads compute 4 DP with reconstructions, prefetching might not work
Intel®Advisor Memory Access Pattern (MAP)

know your access pattern

Unit-Stride access

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

Constant stride access

```
for (i=0; i<N; i++)
    point[i].x = x[i]
```

Variable stride access

```
for (i=0; i<N; i++)
    A[B[i]] = C[i]*D[i]
```
Gather/Scatter Analysis

Advisor MAP detects gather “offset patterns”.

---

<table>
<thead>
<tr>
<th>Pattern #</th>
<th>Pattern Name</th>
<th>Horizontal Stride Value</th>
<th>Vertical Stride Value</th>
<th>Example of Corresponding Fix(es)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Invariant</td>
<td>0</td>
<td>0</td>
<td>OpenMP uniform clause, simd pragma/directive, reftacing</td>
</tr>
<tr>
<td>2</td>
<td>Uniform (horizontal invariant)</td>
<td>0</td>
<td>Arbitrary</td>
<td>OpenMP uniform clause, simd pragma/directive</td>
</tr>
<tr>
<td>3</td>
<td>Vertical Invariant</td>
<td>Constant</td>
<td>0</td>
<td>OpenMP private clause, simd pragma/directive</td>
</tr>
<tr>
<td>4</td>
<td>Unit</td>
<td>1 or -1</td>
<td></td>
<td>OpenMP linear clause, simd pragma/directive</td>
</tr>
<tr>
<td>5</td>
<td>Constant</td>
<td>Constant = X</td>
<td>Constant = X * VectorLength</td>
<td>Subject for AAs -&gt; SaxA transformation</td>
</tr>
</tbody>
</table>
Enable vectorization of your scalar code: Assumed Dependency

- More data is needed to confirm if the loop can be vectorized
- Select the given loop and run dependency analysis to see if it can be vectorized using an OpenMP* 4.0 simd pragma.
Check if it is safe to vectorize: Advisor Dependencies

Select loop for Dependency Analysis and press play!

Vector Dependence prevents Vectorization!
Almost half loops checked did not have actual dependencies
Plotting a Roofline Chart

A Roofline Chart uses AI as its X axis and FLOPS as its Y axis.

The maximum FLOPS as a product of ops/byte (AI) and maximum bytes supplied per second is a diagonal line.

The CPU's maximum FLOPS can be plotted as a horizontal line.

A loop or function can be plotted as a point on the graph.

A Roofline Chart uses AI as its X axis and FLOPS as its Y axis.
**Old approach – pen and paper**

**Run STREAM**

**Run DGEMM**

**Read the source, count FP ops, loads & stores**

---

"3D stencil performance evaluation and auto-tuning on multi and many-core computers", C. Andreolli et.al.

**Cumbersome – but people still did it!**
The Roofline Chart in Intel® Advisor

1. **Run Roofline**

2. A single button or CLI command runs the Survey and FLOPS analyses to generate the Roofline chart.

3. **Focus on big dots with headroom.**
   - Intel® Advisor sizes and color-codes dots by relative time taken.
   - Space between dots and their uppermost limits is room to improve.
   - Roofs above a dot indicate potential sources of bottlenecks.
Roofline Automation in Intel® Advisor

Each Dot represents loop or function in YOUR APPLICATION (profiled)

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

NEW: INTOP/S based Roofline for your Machine Learning codes

Automatic and integrated – first class citizen in Intel® Advisor
Questions to answer with Roofline: for your loops / functions

1. Am I doing well? How far am I from the pick? (do I utilize hardware well or not?)

2. Final Bottleneck? (where will be my limit after I done all optimizations?)
   Long-term ROI, optimization strategy

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
Feature Synergy
Overcoming the Scalar Add Peak

- Survey and Code Analytics tabs indicate vectorization status with colored icons. ⛤ = Scalar   ⛧ = Vectorized
- “Why No Vectorization” tab and column in Survey explain what prevented vectorization.
- Recommendations tab may help you vectorize the loop.
- Dependencies determines if it’s safe to force vectorization.
Feature Synergy
Overcoming the Vector Add Peak

Survey and Code Analytics display the vector efficiency and presence of FMAs.

- Recommendations may help improve efficiency or induce FMA usage.

The Assembly tab* is useful for determining how well you are making use of FMAs.

*Color coding added for clarity.
Feature Synergy
Overcoming the Memory Bandwidth Roofs

• Memory Access Patterns (MAP) identifies inefficient access patterns.
• Intel® SIMD Data Layout Templates (Intel® SDLT) allows code written as AOS to be stored as efficient SOA.
• Intel® VTune™ Amplifier can be used to further optimize cache usage.
• If cache usage cannot be improved, try re-working the algorithm to increase the AI (and slide up the roof).

Performance is limited by minimum of intercepts (L2, LLC, DRAM, CPU)

In this example: by DRAM
How to generate **CARM** Roofline profile?*
(using Advisor from USB stick/goto link)

*We will **NOT** do this during hands-on*

As simple as:

```
$ source advixe-vars.sh

1st method. **Not compatible** with MPI applications:

$ advixe-cl -collect roofline --project-dir ./your_project -- <your-executable-with-parameters>

2nd method (compatible with MPI, more flexible):

$ advixe-cl -collect survey --project-dir ./your_project -- <your-executable-with-parameters>

$ advixe-cl -collect tripcounts -flop --project-dir ./your_project -- <your-executable-with-parameters>

(optional) copy data to your UI desktop system

$ advixe-gui ./your_project

$ advixe-cl -report roofline --project-dir ./your_project > roofline.html
```
Exporting Integer and Integrated Roofline as HTML

**GUI:** Use Export as HTML button

**Command line:**
- Set ADVIXE_EXPERIMENTAL=int_roofline
- advixe-cl –report roofline
- -data-type=float
- -memory-level=L2
- -memory-operation-type=load
- -project-dir /path/to/project/dir

Possible data types: float, int, mixed
memory levels: L1, L2, L3, DRAM
memory operation types: load, store, all

- Export Roofline from command line does not need GUI sub-system on clusters
- Useful for rooflines quick exchange
Python API

- Fairly new Advisor Extensibility/customization mechanism. Actively used internally in Intel
- Up to 500 metrics for each loop/function. Really easy to use:
  - `<advisor_install_dir>/pythonapi/examples`
  - `$python survey_bottomup.py <project_dir>`
  - Generate your own customized roofline charts

```
import sys
try:
    import advisor
project = advisor.open_project(sys.argv[1])
survey = project.load(advisor.SURVEY)
for row in survey.bottomup:
    # row as string
    print(row)
    # row as iterator
    for key in row:
        # row as dictionary
        print('{}: {}'.format(key, row[key]))
```

From ISC’18 paper (cudos Tuomas)
Legal Disclaimer & Optimization Notice

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 © 2018, Intel Corporation. All rights reserved. Intel, Xeon, Core, VTune, 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