VISUALIZING AND FINDING OPTIMIZATION OPPORTUNITIES WITH INTEL® ADVISOR ROOFLINE FEATURE

AGENDA

• Vectorization is becoming more and more important

• What is the theoretical roofline model?

• How is it implemented in Advisor?

• Some examples
Evolution of SIMD for Intel Processors

**Goal:**
8x peak FLOPs (FMA) over 4 generations!

**2nd Generation Intel® Core™ Processors**
- Intel® AVX (256 bit):
  - 2x FP Throughput
  - 2x Load Throughput

**3rd Generation Intel® Core™ Processors**
- Half-float support
- Random Numbers

**4th Generation Intel® Core™ Processors**
- Intel® AVX2 (256 bit):
  - 2x FMA peak
  - Gather Instructions

**Present & Future:**
- Intel® MIC Architecture
- Intel® AVX-512:
  - 512 bit Vectors
  - 2x FP/Load/FMA

Since 1999:
- 128 bit Vectors
AGENDA

• Vectorization is becoming more and more important

• What is the theoretical roofline model?

• How is it implemented in Advisor?

• Some examples
What is the Roofline Model?
Do you know how fast you should run?

• Comes from Berkeley
• Performance is limited by equations/implementation & code generation/hardware
• 2 hardware limitations
  • PEAK Flops
  • PEAK Bandwidth
• The application performance is bounded by hardware specifications

\[
\text{Gflop/s} = \min \left\{ \frac{\text{Platform PEAK}}{\text{Platform BW} \times AI} \right\}
\]

Arithmetic Intensity (Flops/Bytes)
Platform Peak Flops

How many floating point operations per second

\[ \text{Gflop/s} = \min \left\{ \frac{\text{Platform PEAK}}{\text{Platform BW} \times \text{AI}} \right\} \]

- Theoretical value can be computed by specification
  Example with 2 sockets Intel® Xeon® Processor E5-2697 v2
  \[ \text{PEAK FLOP} = 2 \times 2.7 \times 12 \times 8 \times 2 = 1036.8 \text{ Gflop/s} \]

  - Number of sockets
  - Number of cores
  - Core Frequency
  - Number of single precision element in a SIMD register
  - 1 port for addition, 1 for multiplication

- More realistic value can be obtained by running Linpack
  \(~ 930 \text{ Gflop/s} \text{ on a 2 sockets Intel® Xeon® Processor E5-2697 v2} \)
PLATFORM PEAK BANDWIDTH

How many bytes can be transferred per second

\[ \text{Gflop/s} = \min \left\{ \frac{\text{Platform PEAK}}{\text{Platform BW} \times AI} \right\} \]

- Theoretical value can be computed by specification
  Example with 2 sockets Intel® Xeon® Processor E5-2697 v2
  \[ \text{PEAK BW} = 2 \times 1.866 \times 8 \times 4 = 119 \text{ GB/s} \]

- More realistic value can be obtained by running Stream
  \(~= 100 \text{ GB/s} \) on a 2 sockets Intel® Xeon® Processor E5-2697 v2
DRAWING THE ROOFLINE
Defining the speed of light

\[
Gflop/s = \min \left\{ \frac{\text{Platform PEAK}}{\text{Platform BW} \times AI} \right\}
\]

2 sockets Intel® Xeon® Processor E5-2697 v2
Peak Flop = 1036 Gflop/s
Peak BW = 119 GB/s

1036 Gflops/s
8.7 AI [Flop/B]
Drawing the Roofline

Defining the speed of light

\[
Gflop/s = \min \left\{ \frac{\text{Platform PEAK}}{\text{Platform BW} \times AI} \right\}
\]

- **2 sockets Intel® Xeon® Processor E5-2697 v2**
  - Peak Flop = 1036 Gflop/s
  - Peak BW = 119 GB/s
WHAT IS THE PERFORMANCE BOUNDARY?

Manual way to do it

• Manual counting on matrix/matrix multiplication

  \[
  \text{for}(i=0; i<N; i++)
  \]
  \[
  \quad \text{for}(j=0; j<N; j++)
  \]
  \[
  \quad \quad \text{for}(k=0; k<N; k++)
  \]
  \[
  c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]
  \]

• \# add = \(N \times N \times N\)  \quad \# \text{Read} = 3 \times N \times N \times 4 \text{ bytes}

• \# mul = \(N \times N \times N\)  \quad \# \text{Write} = N \times N \times 4 \text{ bytes}

• \(AI = \frac{2N^3}{16N^2} = \frac{1}{8}N\)
Compute the maximum performance

BW * Arithmetic Intensity

\[ \text{Gflop/s} = \min \left\{ \frac{\text{Platform PEAK}}{\text{Platform BW} \times AI} \right\} \]

- 2 sockets Intel® Xeon® Processor E5-2697 v2
  - Peak Flop = 1036 Gflop/s
  - Peak BW = 119 GB/s

If \( N = 8 \), sgemm should not be able to perform better than 119 GFlop/s on a 2 sockets Ivy Bridge

For sgemm
- \( AI = \frac{1}{8} N \)
- If \( N = 8 \), \( AI = 1 \)
How to get better performance?

- Vectorization + threading
- Optimize memory access

1036
119

Gflops/s

1 8.7
AGENDA

• Vectorization is becoming more and more important

• What is the theoretical roofline model?

• How is it implemented in Advisor?

• Some examples
ROOFLINE IN INTEL® ADVISOR

The cache aware roofline model

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
- Bytes and Flops -> Instrumentation
- Time -> Sampling

Typically AI_CARM < AI_DRAM < AI_TRAM

CARM: Cache aware Roofline Model
DRAM: DRAM aware Roofline Model
TRAM: Theoretical Roofline Model
UNDERSTANDING THE ROOFLINE IN INTEL® ADVISOR

Better optimized – smaller potential (gap)

Purely Cache/DRAM-bound

Big optimization gap

Purely compute-bound
AGENDA

• Vectorization is becoming more and more important

• What is the theoretical roofline model?

• How is it implemented in Advisor?

• Some examples
ROOFLINE MODEL AND COMPILER OPTIMIZATIONS
ROOFLINE MODEL AND OPTIMIZATIONS

• Matrix/matrix addition

```c
void addition(float* a, float* b, float* c, int size){
    int i, j;
    for(j=0; j<size; j++){
        for(i=0; i<size; i++){
            c[i*size + j] = a[i*size + j]+b[i*size + j];
        }
    }
}
```

• Let’s have a look at the roofline model
ROOFLINE MODEL AND OPTIMIZATIONS

• Compilation with –O1

Very poor performance, far from the DRAM roofline!
ROOFLINE MODEL AND OPTIMIZATIONS

• Lets look at the Memory Access Pattern Analysis

Constant stride found !!! Looks like loops should be reversed
ROOFLINE MODEL AND OPTIMIZATIONS

• Compilation with –O3
VECTORIZATION OF LOOP CARRIED DEPENDENCY
VECTORIZATION OF LOOP CARRIED DEPENDENCY

• Loop carried dependency

```c
void addition(float* a, float* b, float* c, int size) {
    int i, j;
    for(i=0; i<size; i++) {
        for(j=pad; j<size; j++) {
            c[i*size + j] = a[i*size + j] + c[i*size + j-pad];
        }
    }
}
```
VECTORIZATION OF LOOP CARRIED DEPENDENCY
VECTORIZATION OF LOOP CARRIED DEPENDENCY

• Loop carried dependency

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

Confirm dependency is real

There is no confirmation that a real (proven) dependency is present in the loop. To confirm: Run a Dependencies analysis.
VECTORIZATION OF LOOP CARRIED DEPENDENCY

• Loop carried dependency

```c
void addition(float* a, float* b, float* c, int size){
    int i, j;
    for(i=0; i<size; i++){
        #pragma omp simd safelen(4)
        for(j=pad; j<size; j++){
            c[i*size + j] = a[i*size + j]+c[i*size + j-pad];
        }
    }
}
```

In this case, we assume that pad >=4
VECTORIZATION OF LOOP CARRIED DEPENDENCY
### Vectorization of Loop Carried Dependency

**Safelen was 4**

<table>
<thead>
<tr>
<th>Function Call Sites and Loops</th>
<th>Performance Issues</th>
<th>Self Time</th>
<th>Total Time</th>
<th>Type</th>
<th>W. N.</th>
<th>Vectorized Loops</th>
<th>Vector...</th>
<th>Efficiency</th>
<th>Gain F...</th>
<th>VL (Ve...</th>
<th>Self GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>[loop in addition at main3.c:67]</td>
<td></td>
<td>0.398s</td>
<td>0.398s</td>
<td>Vectorized (Body: ...)</td>
<td>AVX</td>
<td>97%</td>
<td>3.89x</td>
<td>4</td>
<td>1,003</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in addition at main3.c:67]</td>
<td></td>
<td>0.398s</td>
<td>0.398s</td>
<td>Vectorized (Body)</td>
<td>AVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in addition at main3.c:67]</td>
<td></td>
<td>0.000s</td>
<td>0.000s</td>
<td>Vectorized (Remain...)</td>
<td>AVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in initialize at main3.c:57]</td>
<td>3 Data type conv...</td>
<td>0.030s</td>
<td>0.050s</td>
<td>Inside vectorized</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in initialize at main3.c:57]</td>
<td>2 Data type conv...</td>
<td>0.020s</td>
<td>0.060s</td>
<td>Inside vectorized</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f [Import thunk rand]</td>
<td></td>
<td>0.012s</td>
<td>0.012s</td>
<td>Function</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in initialize at main3.c:57]</td>
<td>2 Data type conv...</td>
<td>0.012s</td>
<td>0.052s</td>
<td>Inside vectorized</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f -start</td>
<td></td>
<td>0.000s</td>
<td>0.560s</td>
<td>Function</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f initialize</td>
<td></td>
<td>0.000s</td>
<td>0.050s</td>
<td>Inlined Function</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
VECTORIZATION OF FUNCTION CALL
VECTORIZATION OF A FUNCTION CALL WITH OMP

• Function call inside of a loop can kill the vectorization

```c
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
        single_line_addition(a, c, i * SIZE + j);
    }
}

// function is defined in another compilation unit
void single_line_addition(float* a, float* c, int ind) {
    c[ind] = a[ind] + c[ind];
}
```
VECTORIZATION OF A FUNCTION CALL WITH OMP

Performance (GFLOPS)

100
10
1
0.1
0.01
0.1
1
10

L1 Bandwidth: 43.29 GB/sec
L2 Bandwidth: 131.55 GB/sec
L3 Bandwidth: 52.64 GB/sec
DRAM Bandwidth: 19.09 GB/sec

SP Vector FMA Peak: 138.92 GFLOPS
DP Vector FMA Peak: 70.31 GFLOPS
SP Vector Add Peak: 34.04 GFLOPS
DP Vector Add Peak: 16.76 GFLOPS
Scalar Add Peak: 4.59 GFLOPS

Arithmetic Intensity (FLOP/Byte)
VECTORIZATION OF A FUNCTION CALL WITH OMP

Advisor tells you that this pattern can be a problem and proposes a solution.
VECTORIZATION OF A FUNCTION CALL WITH OMP

• Omp declare simd

```c
for(int i=0; i<SIZE; i++){
    #pragma omp simd
    for(int j=0; j<SIZE; j++){
        single_line_addition(a, c, i*SIZE + j);
    }
}
```

```c
#pragma omp declare simd uniform(a, c) linear(ind)
void single_line_addition(float* a, float* c, int ind);
```
### Vectorization of a Function Call with OMP

#### Function Call Sites and Loops

<table>
<thead>
<tr>
<th>Vectorized Loops</th>
<th>FLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE</td>
<td>1,515</td>
</tr>
</tbody>
</table>

#### Performance Issues

<table>
<thead>
<tr>
<th>Source</th>
<th>Top Down Code Analytics</th>
<th>Assembly</th>
<th>Recommendations</th>
<th>Why No Vectorization?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorized</td>
<td></td>
<td></td>
<td></td>
<td>Vectorized (Body)</td>
</tr>
<tr>
<td>2 Data type conversion</td>
<td>0.202s</td>
<td>0.000s</td>
<td>Inside vectorized</td>
<td></td>
</tr>
<tr>
<td>2 Data type conversion</td>
<td>0.012s</td>
<td>0.012s</td>
<td>Function</td>
<td></td>
</tr>
<tr>
<td>2 Data type conversion</td>
<td>0.012s</td>
<td>0.050s</td>
<td>Inside vectorized</td>
<td></td>
</tr>
<tr>
<td>3 Data type conversion</td>
<td>0.008s</td>
<td>0.052s</td>
<td>Inside vectorized</td>
<td></td>
</tr>
<tr>
<td>start</td>
<td>0.000s</td>
<td>3.290s</td>
<td>Function</td>
<td></td>
</tr>
<tr>
<td>initial</td>
<td>0.000s</td>
<td>0.050s</td>
<td>Inlined Function</td>
<td></td>
</tr>
</tbody>
</table>

© 2017 Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others. For more complete information about compiler optimizations, see our Optimization Notice.*
VECTORIZATION OF A FUNCTION CALL WITH OMP

Before

After
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.

**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