Performance Modeling and Analysis

Samuel Williams
Computational Research Division
Lawrence Berkeley National Lab
SWWilliams@lbl.gov
Acknowledgements

- This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231.
- This material is based upon work supported by the DOE RAPIDS SciDAC Institute.
- This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.
- This research used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- Tuomas Koskela for his creation of the Roofline Advisor demo slides
Introduction to Performance Modeling
Why Use Performance Models or Tools?

- Identify performance bottlenecks
- Motivate software optimizations
- **Determine when we’re done optimizing**
  - Assess performance relative to machine capabilities
  - Motivate need for algorithmic changes
- **Predict performance on future machines / architectures**
  - Sets realistic expectations on performance for future procurements
  - Used for HW/SW Co-Design to ensure future architectures are well-suited for the computational needs of today’s applications.
Computational Complexity

- Assume run time is correlated with the number of operations (e.g. FP ops)
- Users define parameterize their algorithms, solvers, kernels
- Count the number of operations as a function of those parameters
- Demonstrate run time is correlated with those parameters
Data Movement Complexity

- Assume run time is correlated with the amount of data accessed (or moved)
- Easy to calculate amount of data accessed… count array accesses
- Data moved is more complex as it requires understanding cache behavior…
  - Compulsory\(^1\) data movement (array sizes) is a good initial guess…
  - … but needs refinement for the effects of finite cache capacities

### Table: Operation Complexity

<table>
<thead>
<tr>
<th>Operation</th>
<th>Flop's</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAXPY</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>DGEMV</td>
<td>O(N^2)</td>
<td>O(N^2)</td>
</tr>
<tr>
<td>DGEMM</td>
<td>O(N^3)</td>
<td>O(N^3)</td>
</tr>
<tr>
<td>FFTs</td>
<td>O(NlogN)</td>
<td>O(N)</td>
</tr>
<tr>
<td>CG</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>MG</td>
<td>O(N^1.33)</td>
<td>O(N^1.33)</td>
</tr>
<tr>
<td>N-body</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

Machine Balance and Arithmetic Intensity

- Data movement and computation can operate at different rates.
- We define machine balance as the ratio of...

\[
\text{Balance} = \frac{\text{Peak DP Flop/s}}{\text{Peak Bandwidth}}
\]

- ...and arithmetic intensity as the ratio of...

\[
\text{AI} = \frac{\text{Flop’s Performed}}{\text{Data Moved}}
\]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Flop’s</th>
<th>Data</th>
<th>AI (ideal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAXPY</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>DGEMV</td>
<td>O(N^2)</td>
<td>O(N^2)</td>
<td>O(1)</td>
</tr>
<tr>
<td>DGEMM</td>
<td>O(N^3)</td>
<td>O(N^2)</td>
<td>O(N)</td>
</tr>
<tr>
<td>FFTs</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(logN)</td>
</tr>
<tr>
<td>CG</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>N-body</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

Kernels with AI greater than machine balance are ultimately compute limited.
### Computational Depth

- Sequential CPUs incur latencies and overheads on memory discontinuities and function calls.
- Parallel machines incur similar overheads on synchronization (shared memory), point-to-point communication, reductions, and broadcasts.
- Thus, we can classify algorithms by **depth** (max depth of the algorithm’s dependency chain).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Flop’s</th>
<th>Data</th>
<th>AI (ideal)</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAXPY</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DGEMV</td>
<td>$O(N^2)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>DGEMM</td>
<td>$O(N^3)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>FFTs</td>
<td>$O(\text{NlogN})$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>CG</td>
<td>$O(N^{1.33})$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>MG</td>
<td>$O(N^{1.33})$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>N-body</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>

**Overheads can dominate at high concurrency or small problems.**
In distributed memory, one communicates by sending messages between processors.

Messaging time can be constrained by several components…
- Overhead (CPU time to send/receive a message)
- Latency (time message is in the network; can be hidden)
- Message throughput (rate at which one can send small messages… messages/second)
- Bandwidth (rate one can send large messages… GBytes/s)

Bandwidths and latencies are further constrained by the interplay of network architecture and contention.

Distributed memory versions of our algorithms can be differently stressed by these components depending on N and P (#processors)
Many different components can contribute to kernel run time. Some are characteristics of the application, some are characteristics of the machine, and some are both (memory access pattern + caches).

<table>
<thead>
<tr>
<th>Component</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>#FP operations</td>
<td>Flop/s</td>
</tr>
<tr>
<td>Cache data movement</td>
<td>Cache GB/s</td>
</tr>
<tr>
<td>DRAM data movement</td>
<td>DRAM GB/s</td>
</tr>
<tr>
<td>PCIe data movement</td>
<td>PCIe bandwidth</td>
</tr>
<tr>
<td>Depth</td>
<td>OMP Overhead</td>
</tr>
<tr>
<td>MPI Message Size</td>
<td>Network Bandwidth</td>
</tr>
<tr>
<td>MPI Send:Wait ratio</td>
<td>Network Gap</td>
</tr>
<tr>
<td>#MPI Wait’s</td>
<td>Network Latency</td>
</tr>
</tbody>
</table>
Performance Models

- Can’t think about all these terms all the time for every application…

\[
\begin{align*}
\text{Computational Complexity} & \quad \text{#FP operations} & \quad \text{Flop/s} \\
\text{Cache data movement} & \quad \text{Cache GB/s} \\
\text{DRAM data movement} & \quad \text{DRAM GB/s} \\
\text{PCIe data movement} & \quad \text{PCIe bandwidth} \\
\text{Depth} & \quad \text{OMP Overhead} \\
\text{MPI Message Size} & \quad \text{Network Bandwidth} \\
\text{MPI Send:Wait ratio} & \quad \text{Network Gap} \\
\text{#MPI Wait’s} & \quad \text{Network Latency}
\end{align*}
\]
Performance Models

- Because there are so many components, performance models often conceptualize the system as being dominated by one or more of these components.

![Roofline Model Diagram]

- #FP operations
- Cache data movement
- DRAM data movement
- PCIe data movement
- Depth
- MPI Message Size
- MPI Send:Wait ratio
- #MPI Wait’s
- Flop/s
- Cache GB/s
- DRAM GB/s
- PCIe bandwidth
- OMP Overhead
- Network Bandwidth
- Network Gap
- Network Latency

Performance Models

- Because there are so many components, performance models often conceptualize the system as being dominated by one or more of these components.

#FP operations    Flop/s
Cache data movement    Cache GB/s
DRAM data movement    DRAM GB/s
PCIe data movement    PCIe bandwidth
Depth    OMP Overhead
MPI Message Size    Network Bandwidth
MPI Send:Wait ratio    Network Gap
#MPI Wait’s    Network Latency

Performance Models

- Because there are so many components, performance models often conceptualize the system as being dominated by one or more of these components.

<table>
<thead>
<tr>
<th>#FP operations</th>
<th>Flop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache data movement</td>
<td>Cache GB/s</td>
</tr>
<tr>
<td>DRAM data movement</td>
<td>DRAM GB/s</td>
</tr>
<tr>
<td>PCIe data movement</td>
<td>PCIe bandwidth</td>
</tr>
<tr>
<td>Depth</td>
<td>OMP Overhead</td>
</tr>
</tbody>
</table>

- MPI Message Size | Network Bandwidth |
- MPI Send:Wait ratio | Network Gap |
- #MPI Wait’s | Network Latency

Because there are so many components, performance models often conceptualize the system as being dominated by one or more of these components.

<table>
<thead>
<tr>
<th>#FP operations</th>
<th>Flop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache data movement</td>
<td>Cache GB/s</td>
</tr>
<tr>
<td>DRAM data movement</td>
<td>DRAM GB/s</td>
</tr>
<tr>
<td>PCIe data movement</td>
<td>PCIe bandwidth</td>
</tr>
<tr>
<td>Depth</td>
<td>OMP C</td>
</tr>
<tr>
<td>MPI Message Size</td>
<td>Network Bandwidth</td>
</tr>
<tr>
<td>MPI Send:Wait ratio</td>
<td>Network Gap</td>
</tr>
<tr>
<td>#MPI Wait’s</td>
<td>Network Latency</td>
</tr>
</tbody>
</table>

LogCA

Think about which model to use

Introduction to the Roofline Model
Performance Models / Simulators

- Historically, many performance models and simulators tracked latencies to predict performance (i.e. counting cycles)

- The last two decades saw a number of latency-hiding techniques…
  - Out-of-order execution (hardware discovers parallelism to hide latency)
  - HW stream prefetching (hardware speculatively loads data)
  - Massive thread parallelism (independent threads satisfy the latency-bandwidth product)

- Effective latency hiding has resulted in a shift from a latency-limited computing regime to a **throughput-limited computing regime**
Roofline Model

- **Roofline Model** is a throughput-oriented performance model...
  - Tracks rates not times
  - Augmented with Little’s Law
    (concurrency = latency*bandwidth)
  - Independent of ISA and architecture (applies to CPUs, GPUs, Google TPUs\(^1\), etc...)

(DRAM) Roofline

- One could hope to always attain peak performance (Flop/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
  - Idealized processor/caches
  - Cold start (data in DRAM)

\[
\text{Time} = \max \left\{ \frac{\#\text{FP ops}}{\text{Peak GFlop/s}}, \frac{\#\text{Bytes}}{\text{Peak GB/s}} \right\}
\]
One could hope to always attain peak performance (Flop/s)

However, finite locality (reuse) and bandwidth limit performance.

Assume:
- Idealized processor/caches
- Cold start (data in DRAM)

\[
\text{Time} = \max \left\{ \frac{1}{\text{Peak GFlop/s}}, \frac{\#\text{Bytes}}{\#\text{FP ops}} / \text{Peak GB/s} \right\}
\]
One could hope to always attain peak performance (Flop/s)

However, finite locality (reuse) and bandwidth limit performance.

Assume:
- Idealized processor/caches
- Cold start (data in DRAM)

\[
\frac{\text{#FP ops}}{\text{Time}} = \min \left\{ \frac{\text{Peak GFlop/s}}{\left(\frac{\text{#FP ops}}{\text{#Bytes}}\right) \times \text{Peak GB/s}} \right\}
\]
One could hope to always attain peak performance (Flop/s)

However, finite locality (reuse) and bandwidth limit performance.

Assume:
- Idealized processor/caches
- Cold start (data in DRAM)

\[
\text{GFlop/s} = \min \left\{ \text{Peak GFlop/s}, \text{AI} \times \text{Peak GB/s} \right\}
\]

Note, Arithmetic Intensity (AI) = Flops / Bytes (as presented to DRAM)
(DRAM) Roofline

- Plot Roofline bound using Arithmetic Intensity as the x-axis
- **Log-log scale** makes it easy to doodle, extrapolate performance along Moore’s Law, etc…
- Kernels with AI less than machine balance are ultimately DRAM bound (we’ll refine this later…)

![Diagram of DRAM Roofline](image)
Roofline Example #1

- Typical machine balance is 5-10 flops per byte...
  - 40-80 flops per double to exploit compute capability
  - Artifact of technology and money
  - Unlikely to improve

- Consider STREAM Triad...
  - 2 flops per iteration
  - Transfer 24 bytes per iteration (read \(X[i]\), \(Y[i]\), write \(Z[i]\))
  - \(AI = 0.083\) flops per byte == Memory bound
Roofline Example #2

- Conversely, 7-point constant coefficient stencil...
  - 7 flops
  - 8 memory references (7 reads, 1 store) per point
  - Cache can filter all but 1 read and 1 write per point
  - $AI = 0.44$ flops per byte == memory bound, but 5x the flop rate

```
#pragma omp parallel for
for(k=1;k<dim+1;k++){  
    for(j=1;j<dim+1;j++){  
        for(i=1;i<dim+1;i++){  
            int ijk = i + j*jStride + k*kStride;  
            new[ijk] = -6.0*old[ijk]  
                + old[ijk-1]  
                + old[ijk+1]  
                + old[ijk-jStride]  
                + old[ijk+jStride]  
                + old[ijk-kStride]  
                + old[ijk+kStride];  
        }}}
```

Arithmetic Intensity (Flop:Byte)
Hierarchical Roofline

- Real processors have multiple levels of memory
  - Registers
  - L1, L2, L3 cache
  - MCDRAM/HBM (KNL/GPU device memory)
  - DDR (main memory)
  - NVRAM (non-volatile memory)

- Applications can have locality in each level
  - Unique data movements imply unique AI’s
  - Moreover, each level will have a unique bandwidth
Hierarchical Roofline

- Construct superposition of Rooflines…
  - Measure a bandwidth
  - Measure AI for each level of memory
    - Although an loop nest may have multiple AI’s and multiple bounds (flops, L1, L2, … DRAM)…
    - … performance is bound by the minimum

![Hierarchical Roofline Diagram](image-url)
Hierarchical Roofline

- Construct superposition of Rooflines…
  - Measure a bandwidth
  - Measure AI for each level of memory
    - Although an loop nest may have multiple AI’s and multiple bounds (flops, L1, L2, … DRAM)…
    - … performance is bound by the minimum

- Attainable Flop/s
- MCDRAM cache GB/s
- DDR GB/s
- DDR bottleneck pulls performance below MCDRAM Roofline

- Peak Flop/s
Hierarchical Roofline

- Construct superposition of Rooflines…
  - Measure a bandwidth
  - Measure AI for each level of memory
    - Although an loop nest may have multiple AI’s and multiple bounds (flops, L1, L2, … DRAM)…
    - … performance is bound by the minimum
Hierarchical Roofline

- Construct superposition of Rooflines...
  - Measure a bandwidth
  - Measure AI for each level of memory
    - Although an loop nest may have multiple AI’s and multiple bounds (flops, L1, L2, … DRAM)…
    - … performance is bound by the minimum

Arithmetic Intensity (Flop:byte) vs Attainable Flop/s

Peak Flop/s

MCDRAM bottleneck pulls performance below DDR Roofline

L2 GB/s

MCDRAM cache GB/s

DDR GB/s
**NUMA Effects**

- Cori’s Haswell nodes are built from 2 Xeon processors (sockets)
  - Memory attached to each socket (fast)
  - Interconnect that allows remote memory access (slow == NUMA)
  - Improper memory allocation can result in more than a 2x performance penalty

![Diagram showing CPU0 and CPU1 with DRAM and Flop/Byte relation](image)
Modeling In-Core Performance Effects
Data, Instruction, Thread-Level Parallelism...

Modern CPUs use several techniques to increase per core Flop/s

**Fused Multiply Add**
- $w = x \cdot y + z$ is a common idiom in linear algebra
- Rather than having separate multiply and add instructions, processors can use a fused multiply add (FMA)
- The FPU chains the multiply and add in a single pipeline so that it can complete FMA/cycle

**Vector Instructions**
- Many HPC codes apply the same operation to a vector of elements
- Vendors provide vector instructions that apply the same operation to 2, 4, 8, 16 elements…
  \[ x[0:7] \cdot y[0:7] + z[0:7] \]
- Vector FPUs complete 8 vector operations/cycle

**Deep pipelines**
- The hardware for a FMA is substantial.
- Breaking a single FMA up into several smaller operations and pipelining them allows vendors to increase GHz
- Little’s Law applies… need $FP\_Latency \cdot FP\_bandwidth$ independent instructions
If every instruction were an ADD (instead of FMA), performance would drop by 2x on KNL or 4x on Haswell.

Similarly, if one had no vector instructions, performance would drop by another 8x on KNL and 4x on Haswell.

FP Divides can be even worse.

Lack of threading will reduce performance by 64x on KNL.
Superscalar vs. instruction mix

- Define in-core ceilings based on instruction mix…
- e.g. Haswell
  - 4-issue superscalar
  - Only 2 FP data paths
  - Requires 50% of the instructions to be FP to get peak performance
Superscalar vs. instruction mix

- Define in-core ceilings based on instruction mix…

  e.g. Haswell
  - 4-issue superscalar
  - Only 2 FP data paths
  - Requires 50% of the instructions to be FP to get peak performance

  e.g. KNL
  - 2-issue superscalar
  - 2 FP data paths
  - Requires 100% of the instructions to be FP to get peak performance
Superscalar vs. instruction mix

- Define in-core ceilings based on instruction mix...
  - e.g. Haswell
    - 4-issue superscalar
    - Only 2 FP data paths
    - Requires 50% of the instructions to be FP to get peak performance
  - e.g. KNL
    - 2-issue superscalar
    - 2 FP data paths
    - Requires 100% of the instructions to be FP to get peak performance

Diagram:
- Arithmetic Intensity
- DDR GB/s
- Attainable Flop/s
- Peak Flop/s
- 100% FP
- 50% FP
- 25% FP

Cloud:
- non-FP instructions can sap instruction issue bandwidth and pull performance below Roofline
Modeling Cache Effects
Locality Walls

- Naively, we can bound AI using only compulsory cache misses

\[
\text{AI} = \frac{\text{#Flop's}}{\text{Compulsory Misses}}
\]
Locality Walls

- Naively, we can bound AI using only compulsory cache misses
- However, write allocate caches can lower AI

\[
AI = \frac{\text{#Flop's}}{\text{Compulsory Misses} + \text{Write Allocates}}
\]
**Locality Walls**

- Naively, we can bound AI using only compulsory cache misses.
- However, write allocate caches can lower AI.
- Cache capacity misses can have a huge penalty.

\[ AI = \frac{\text{Flop's}}{\text{Compulsory Misses} + \text{Write Allocates} + \text{Capacity Misses}} \]
Locality Walls

- Naively, we can bound AI using only compulsory cache misses
- However, write allocate caches can lower AI
- Cache capacity misses can have a huge penalty

Compute bound became memory bound

Know the theoretical bounds on your AI.

\[
AI = \frac{\text{#Flop}'}{\text{Compulsory Misses} + \text{Write Allocates} + \text{Capacity Misses}}
\]
General Strategy Guide

- Broadly speaking, there are three approaches to improving performance:
Broadly speaking, there are three approaches to improving performance:

- **Maximize in-core performance (e.g. get compiler to vectorize)**
General Strategy Guide

- Broadly speaking, there are three approaches to improving performance:
  - Maximize in-core performance (e.g. get compiler to vectorize)
  - Maximize memory bandwidth (e.g. NUMA-aware allocation)
General Strategy Guide

- Broadly speaking, there are three approaches to improving performance:
  - Maximize in-core performance (e.g. get compiler to vectorize)
  - Maximize memory bandwidth (e.g. NUMA-aware allocation)
  - Minimize data movement (increase AI)
Constructing a Roofline Model requires answering some questions…
Questions can overwhelm users…

Properties of the target machine (Benchmarking)

- What is my machine’s peak flop/s?
- How much data did my kernel actually move?
- How many flop’s did my kernel actually do?
- How much did that divide hurt?
- What is my machine’s DDR GB/s?
- L2 GB/s?
- How important is vectorization or FMA on my machine?

Properties of an application’s execution (Instrumentation)

- What is my machine’s peak flop/s?
- What is my machine’s DDR GB/s?
- How important is vectorization or FMA on my machine?
- How much data did my kernel actually move?
- How many flop’s did my kernel actually do?
- How much did that divide hurt?

Fundamental properties of the kernel constrained by hardware (Theory)

- What is my kernel’s compulsory AI?
- (communication lower bounds)
- Can my kernel ever be vectorized?
To answer these questions, we need tools...
Node Characterization?

- "Marketing Numbers" can be deceptive...
  - Pin BW vs. real bandwidth
  - TurboMode / Underclock for AVX
  - compiler failings on high-AI loops.

- LBL developed the Empirical Roofline Toolkit (ERT)...
  - Characterize CPU/GPU systems
  - Peak Flop rates
  - Bandwidths for each level of memory
  - MPI+OpenMP/CUDA == multiple GPUs

https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/
Instrumentation with Performance Counters?

- **Characterizing applications with performance counters can be problematic...**
  - Flop Counters can be broken/missing in production processors
  - Vectorization/Masking can complicate counting Flop’s
  - Counting Loads and Stores doesn’t capture cache reuse while counting cache misses doesn’t account for prefetchers.
  - DRAM counters (Uncore PMU) might be accurate, but...
    - are privileged and thus nominally inaccessible in user mode
    - may need vendor (e.g. Cray) and center (e.g. NERSC) approved OS/kernel changes
Forced to Cobble Together Tools...

- Use tools known/observed to work on NERSC’s Cori (KNL, HSW)...
  - Used **Intel SDE** (Pin binary instrumentation + emulation) to create software Flop counters
  - Used **Intel VTune** performance tool (NERSC/Cray approved) to access uncore counters
- Accurate measurement of Flop’s (HSW) and DRAM data movement (HSW and KNL)
- Used by NESAP (NERSC KNL application readiness project) to characterize apps on Cori...

http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/
Initial Roofline Analysis of NESAP Codes

**MFDn**

**EMGeo**

**PICSAR**

**DRAM-only Roofline was insufficient for PICSAR**
Evaluation of LIKWID

- LIKWID provides easy to use wrappers for measuring performance counters…
  - Works on NERSC production systems
  - Minimal overhead (<1%)
  - Scalable in distributed memory (MPI-friendly)
  - Fast, high-level characterization
  - No detailed timing breakdown or optimization advice
  - Limited by quality of hardware performance counter implementation (garbage in/garbage out)

- Useful tool that complements other tools

https://github.com/RRZE-HPC/likwid
Need an integrated solution...

- Having to compose VTune, SDE, and plotting tools...
  - ✓ worked correctly and benefited NESAP’s application readiness
  - x forced users to learn/run multiple tools and manually parse/graph the output
  - x forced users to instrument routines of interest in their application
  - x lacked integration with compiler/debugger/disassembly

- LIKWID was...
  - ✓ fast and easy to use
  - x Suffered from the same limitations as VTune/SDE

- ERT...
  - ✓ Characterized flops, and bandwidths (cache, DRAM)
  - ✓ Interoperable with MPI, OpenMP, and CUDA
  - x Required users to manually parse/incorporate the output
Intel Advisor

- Includes Roofline Automation…
  - Automatically instruments applications (one dot per loop nest/function)
  - Computes FLOPS and AI for each function (**CARM**)
  - AVX-512 support that incorporates masks
  - **Integrated Cache Simulator**¹ (hierarchical roofline / multiple AI’s)
  - Automatically benchmarks target system (calculates ceilings)
  - Full integration with existing Advisor capabilities

http://www.nersc.gov/users/training/events/roofline-training-1182017-1192017

¹Technology Preview, not in official product roadmap so far.
Hierarchical Roofline vs. Cache-Aware Roofline

...understanding different Roofline formulations in Advisor
There are two Major Roofline Formulations:

- **Hierarchical Roofline (original Roofline w/ DRAM, L3, L2, …)**
  - Chapter 4 of “Auto-tuning Performance on Multicore Computers”, 2008
  - Defines multiple bandwidth ceilings and multiple Al’s per kernel
  - Performance bound is the minimum of flops and the memory intercepts (superposition of original, single-metric Rooflines)

- **Cache-Aware Roofline**
  - Defines multiple bandwidth ceilings, but uses a single AI (flop:L1 bytes)
  - As one looses cache locality (capacity, conflict, …) performance falls from one BW ceiling to a lower one at constant AI

**Why Does this matter?**

- Some tools use the Hierarchical Roofline, some use cache-aware == Users need to understand the differences
- Cache-Aware Roofline model was integrated into production Intel Advisor
- Evaluation version of Hierarchical Roofline¹ (cache simulator) has also been integrated into Intel Advisor

¹Technology Preview, not in official product roadmap so far.
Hierarchical Roofline

- Captures cache effects
- AI is Flop:Bytes after being filtered by lower cache levels
- Multiple Arithmetic Intensities (one per level of memory)
- AI dependent on problem size (capacity misses reduce AI)
- Memory/Cache/Locality effects are observed as decreased AI
- Requires performance counters or cache simulator to correctly measure AI

Cache-Aware Roofline

- Captures cache effects
- AI is Flop:Bytes as presented to the L1 cache (plus non-temporal stores)
- Single Arithmetic Intensity
- AI independent of problem size
- Memory/Cache/Locality effects are observed as decreased performance
- Requires static analysis or binary instrumentation to measure AI
Example: STREAM

- **L1 AI…**
  - 2 flops
  - 2 x 8B load (old)
  - 1 x 8B store (new)
  - = 0.08 flops per byte

- **No cache reuse…**
  - Iteration i doesn’t touch any data associated with iteration i+delta for any delta.

- … leads to a DRAM AI equal to the L1 AI

```c
#pragma omp parallel for
for(i=0;i<N;i++){
    Z[i] = X[i] + alpha*Y[i];
}
```
Example: STREAM

Hierarchical Roofline

- Performance is bound to the minimum of the two intercepts:
  - $\text{AI}_{L1} \times \text{L1 GB/s}$
  - $\text{AI}_{\text{DRAM}} \times \text{DRAM GB/s}$

- Multiple AI’s:
  1. Flop:DRAM bytes
  2. Flop:L1 bytes (same)

Cache-Aware Roofline

- Observed performance is correlated with DRAM bandwidth
- Single AI based on flop:L1 bytes

Arithmetic Intensity (Flop:Byte)
Example: 7-point Stencil (Small Problem)

- **L1 AI…**
  - 7 flops
  - 7 x 8B load (old)
  - 1 x 8B store (new)
  - = 0.11 flops per byte
  - some compilers may do register shuffles to reduce the number of loads.

- **Moderate cache reuse…**
  - old[ijk] is reused on subsequent iterations of i,j,k
  - old[ijk-1] is reused on subsequent iterations of i.
  - old[ijk-jStride] is reused on subsequent iterations of j.
  - old[ijk-kStride] is reused on subsequent iterations of k.

- … leads to DRAM AI larger than the L1 AI

```c
#pragma omp parallel for
for(k=1;k<dim+1;k++){
    for(j=1;j<dim+1;j++){
        for(i=1;i<dim+1;i++){
            int ijk = i + j*jStride + k*kStride;
            new[ijk] = -6.0*old[ijk]
                       + old[ijk-1]
                       + old[ijk+1]
                       + old[ijk-jStride]
                       + old[ijk+jStride]
                       + old[ijk-kStride]
                       + old[ijk+kStride];
        }
    }
}
```
Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

<table>
<thead>
<tr>
<th>DRAM GB/s</th>
<th>Attainable Flop/s</th>
<th>Arithmetic Intensity (Flop:Byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.11</td>
<td>0.44</td>
</tr>
</tbody>
</table>

Performance bound is the minimum of the two

Multiple AI’s….

1) flop:DRAM ~ 0.44
2) flop:L1 ~ 0.11

Cache-Aware Roofline

<table>
<thead>
<tr>
<th>DRAM GB/s</th>
<th>Attainable Flop/s</th>
<th>Arithmetic Intensity (Flop:Byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.11</td>
<td>0.44</td>
</tr>
</tbody>
</table>

Peak Flop/s
Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

Cache-Aware Roofline

Performance bound is the minimum of the two

Multiple AI’s:
1) flop:DRAM ~ 0.44
2) flop:L1 ~ 0.11

Observed performance is between L1 and DRAM lines (== some cache locality)

Single AI based on flop:L1 bytes
Example: 7-point Stencil (Large Problem)

Hierarchical Roofline

- **Peak Flop/s**
- **Attainable Flop/s**
- **Arithmetic Intensity (Flop:Byte)**
- **L1 GB/s**
- **DRAM GB/s**

Capacity misses reduce DRAM AI and performance

Multiple AI’s:
1) flop:DRAM ~ 0.20
2) flop:L1 ~ 0.11

Cache-Aware Roofline

- **Peak Flop/s**
- **Attainable Flop/s**
- **Arithmetic Intensity (Flop:Byte)**
- **L1 GB/s**
- **DRAM GB/s**

Observed performance is closer to DRAM line (== less cache locality)

Single AI based on flop:L1 bytes
Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline

- Peak Flop/s
- Attainable Flop/s
- Arithmetic Intensity (Flop:Byte)

Actual observed performance is tied to the bottlenecked resource and can be well below a cache Roofline (e.g. L1).

Cache-Aware Roofline

- Peak Flop/s
- Attainable Flop/s
- Arithmetic Intensity (Flop:Byte)

Observed performance is closer to DRAM line (== less cache locality)

Single AI based on flop:L1 bytes
Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline

- Actual observed performance is tied to the bottlenecked resource and can be well below a cache Roofline (e.g. L1).

Cache-Aware Roofline

- Observed performance is closer to DRAM line (== less cache locality)

Arithmetic Intensity (Flop:Byte)

- Single AI based on flop:L1 bytes
Example of Roofline in Practice
Sparse Matrix Vector Multiplication

• **What’s a Sparse Matrix?**
  - Most entries are 0.0
  - Performance advantage in only storing/operating on the nonzeros
  - Requires significant meta data to reconstruct the matrix structure

• **What’s SpMV?**
  - Evaluate $y=Ax$ where $A$ is a sparse matrix, $x$ & $y$ are dense vectors

• **Challenges**
  - Very low arithmetic intensity (often <0.166 flops/byte)
  - Difficult to exploit ILP (bad for pipelined or superscalar),
  - Difficult to exploit DLP (bad for SIMD)

![Sparse Matrix Vector Multiplication](image-url)
Roofline model for SpMV

- Double precision roofline models
- In-core optimizations 1..i
- DRAM optimizations 1..j

\[ \text{GFlops}_{i,j}(\text{AI}) = \min \left\{ \frac{\text{InCoreGFlops}_i}{\text{StreamBW}_j} \times \text{AI} \right\} \]

- FMA is inherent in SpMV (place at bottom)
Roofline model for SpMV
(overlay arithmetic intensity)

- Two unit stride streams
- Inherent FMA
- No ILP
- No DLP
- FP is 12-25%
- Naïve compulsory flop:byte < 0.166
Roofline model for SpMV
(out-of-the-box parallel)

- Two unit stride streams
- Inherent FMA
- No ILP
- No DLP
- FP is 12-25%
- Naïve compulsory flop:byte < 0.166
- For simplicity: dense matrix in sparse format
Roofline model for SpMV
(NUMA & SW prefetch)

- compulsory flop:byte ~ 0.166
- utilize all memory channels
Roofline model for SpMV
(matrix compression)

- Inherent FMA
- Register blocking improves ILP, DLP, flop:byte ratio, and FP% of instructions
In 2010, we began to use Roofline to compare CPU and GPU performance for a variety of double-precision kernels

- Flop/s were theoretical book values;
- Memory bandwidth from STREAM or SHOC;
- AI was based on compulsory data movement;
- Optimized kernel performance was well-correlated with Roofline for both platforms.
- Some irregular applications (PIC) underperformed and motivated further study.
Questions?
Backup
*DRAM Roofline and OS/X Advisor GUI:* These are preview features that may or may not be included in mainline product releases. They may not be stable as they are prototypes incorporating very new functionality. Intel provides preview features in order to collect user feedback and plans further development and productization steps based on the feedback.

http://www.nersc.gov/users/training/events/roofline-training-1182017-1192017
Intel Advisor

- **Integrated Performance Analysis Tool**
  - Performance information including timings, flops, and trip counts
  - Vectorization Tips
  - Memory footprint analysis
  - **Originally used the Cache-Aware Roofline Model**
  - All connected back to source code

- **CRD/NERSC began a collaboration with Intel**
  - Ensure Advisor runs on Cori in user-mode
  - Push for **Hierarchical (Integrated) Roofline**
  - Make it functional/scalable to many MPI processes across multiple nodes
  - Validate results on NESAP, SciDAC, and ECP codes

---

NESAP is NERSC's KNL application readiness project
SciDAC is the DOE Office of Science's Scientific Discovery thru Advanced Computing program
ECP is the DOE's Exascale Computing Project
Intel Advisor (Useful Links)

Background
- https://www.youtube.com/watch?v=h2QEM1HpFgg

Running Advisor on NERSC Systems
Using Intel Advisor at NERSC

- Compile...
  use ‘-g’ when compiling

- Submit Job...
  % salloc -perf=vtune <<< interactive sessions; --perf only needed for DRAM Roofline
  -or-
  #SBATCH -perf=vtune <<< batch submissions; --perf only needed for DRAM Roofline

Benchmark...
  % module load advisor
  % export ADVIXE_EXPERIMENTAL=roofline_ex <<< only needed for DRAM Roofline
  % srun [args] advixe-cl -collect survey -no-stack-stitching -project-dir $DIR -- ./a.out [args]

- Use Advisor GUI...
  % module load advisor
  % export ADVIXE_EXPERIMENTAL=roofline_ex <<< only needed for DRAM Roofline
  % advixe-gui $DIR
Welcome to Intel Advisor 2017
Vectorization Optimization and Thread Prototyping

Current project: advi.stencil.aug2.16

- Show My Result
- Configure Project...
- New Project...
- Open Project...
- Open Result

Recent Projects:
- advi.dram.stencil.aug2.16
### Function Call Sites and Loops

<table>
<thead>
<tr>
<th>Loop in bench_stencil_ver4...</th>
<th>Vector Issues</th>
<th>Self Time</th>
<th>Total Time</th>
<th>Type</th>
<th>FLOPS</th>
<th>AI</th>
</tr>
</thead>
<tbody>
<tr>
<td>[loop in bench_stencil_ver4...](loop in bench_stencil_ver4...)</td>
<td>1 Ineffective peeled/...</td>
<td>159.95s</td>
<td>159.95s</td>
<td>Vectorized (Body)</td>
<td>23.08</td>
<td>0.117</td>
</tr>
<tr>
<td>[loop in bench_stencil_ver3...](loop in bench_stencil_ver3...)</td>
<td>1 Ineffective peeled/...</td>
<td>160.03s</td>
<td>160.03s</td>
<td>Vectorized (Body: Peel...)</td>
<td>15.66</td>
<td>0.117</td>
</tr>
<tr>
<td>[loop in bench_stencil_ver2...](loop in bench_stencil_ver2...)</td>
<td>1 Ineffective peeled/...</td>
<td>159.37s</td>
<td>159.37s</td>
<td>Vectorized (Body: Peel...)</td>
<td>10.21</td>
<td>0.117</td>
</tr>
<tr>
<td>[loop in bench_stencil_ver1...](loop in bench_stencil_ver1...)</td>
<td>1 Potential under...</td>
<td>157.99s</td>
<td>157.99s</td>
<td>Scalar</td>
<td>9.01</td>
<td>0.117</td>
</tr>
</tbody>
</table>

### Source

Loop in bench_stencil_ver0$omp$parallel_for...

at stencil_v2.c:29

---

### Average Trip Counts

512

### Instruction Mix

- Memory: 5
- Compute: 9
- Mixed: 4
- Other: 4
- Number of Vector Registers: 9

### GFLOPS

9.00873

### Code Optimizations

- Compiler: Intel(R) C Intel(R) 64
- Compiler for applications running on Intel(R) 64,
- Version: 17.0.2.174 Build 20170213
Intel Advisor:
Stencil Roofline Demo*

*DRAM Roofline and OS/X Advisor GUI: These are preview features that may or may not be included in mainline product releases. They may not be stable as they are prototypes incorporating very new functionality. Intel provides preview features in order to collect user feedback and plans further development and productization steps based on the feedback.
7-point, Constant-Coefficient Stencil

- Apply to a $512^3$ domain on a single NUMA node (single HSW socket)
- Create 5 code variants to highlight effects (as seen in advisor)

ver0. Baseline: thread over outer loop (k), but prevent vectorization

```c
#pragma novector
int ijk = i*iStride + j*jStride + k*kStride;  // variable iStride to confuse the compiler
```

ver1. Enable vectorization

```c
int ijk = i + j*jStride + k*kStride;  // unit-stride inner loop
```

ver2. Eliminate capacity misses

*2D tiling of j-k iteration space*  // working set had been $O(6MB)$ per thread

ver3. Improve vectorization

*Provide aligned pointers and strides*

ver4. Force vectorization / cache bypass

```c
__assume(jStride%8 == 0);  // stride by variable is still aligned
#pragma omp simd, vector nontemportal  // force simd; force cache bypass
```
Function Call Sites and Loops

```
[loop in bench_stencil_ver4...]
Self Time: 159.595s
```

```
[loop in bench_stencil_ver3...]
Self Time: 159.595s
```

```
[loop in bench_stencil_ver2...]
Self Time: 160.035s
```

```
[loop in bench_stencil_ver1...]
Self Time: 159.307s
```

```
[loop in bench_stencil_ver0...]
Self Time: 115.7994s
```

File: stencil_v2.c:29 bench_stencil_ver0$omp$parallel_for@25

```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
    for(j=1;j<dim+1;j++){
        #pragma novector
        for(i=1;i<dim+1;i++){
            int ijk = i*iStride + j*jStride + k*kStride;
            new[ijk] = -6.0*old[ijk] + old[ijk-iStride];
        }
    }
}
```

Total Time: 9.890s
Loop/Function Time: 102.403s

Selected (Total Time): 102.403s
Cache-Aware Roofline
Cache-Aware Roofline
Cache-Aware Roofline
Cache-Aware Roofline
Cache-Aware Roofline
DRAM Roofline*

![Image of DRAM Roofline](image.png)
DRAM Roofline*
DRAM Roofline*
DRAM Roofline*
DRAM Roofline*

[Diagram of DRAM Roofline analysis tool]

Summary of performance metrics:
- Elapsed time: 50.40s
- Total Elapsed Time: 9.956s
- Total Performance: 18.98 GFLOPS
- Total L1 Arithmetic Intensity: 0.41 FLOP/Byte

[Code analysis details]
- File: stencil_v2.c:193
- Total Time: 8.004ms
- Loop/Function Time: 160161.000ms

[Recommendations for vectorization]
Little’s Law

Applied to Memory
- Consider a CPU with 100GB/s of bandwidth and 100ns memory latency.
- Little’s law states that we must express 10KB of concurrency (independent memory operations) to the memory subsystem to attain peak performance.
- Solution #1: use multiple cores or threads to satisfy the requisite MLP.
- Solution #2: express the memory access pattern in a streaming fashion in order to engage the prefetchers.

Applied to FPUs
- Consider a CPU with 2 FPU’s each with a 4-cycle latency.
- Little’s law states that we must express 8-way ILP to fully utilize the machine.
- Solution #1: rely on OOO to find parallelism across loop iterations.
- Solution #2: unroll/jam the code to express 8 independent FP operations.
- Note, simply unrolling dependent operations (e.g. reduction) does not increase ILP. It simply amortizes loop overhead.