Performance Tuning of Scientific Codes with the Roofline Model

8:30am  Welcome/Introductions  all
8:35am  Roofline Introduction  Samuel Williams
9:15am  Roofline on GPU-accelerated Systems  Charlene Yang
10:00am  break  
10:30am  Roofline on CPU-based Systems  Samuel Williams
11:15am  HPC Application Studies  Jack Deslippe
11:55pm  closing remarks / Q&A  all
Introduction to the Roofline Model

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.
Background
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.
Many different components can contribute to kernel run time. Some are application-specific, and some architecture-specific.

<table>
<thead>
<tr>
<th>Component</th>
<th>Measurement</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>
Can’t think about all these terms all the time for every application…

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

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

- LogP

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

- #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

LogGP

Performance Models

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

Performance Models / Simulators

- Historically, many performance models and simulators tracked time 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)

- … 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 TPUs¹, etc...)


https://crd.lbl.gov/departments/computer-science/PAR/research/roofline
Roofline Model:
Arithmetic Intensity and Bandwidth
(DRAM) Roofline

- One could hope to always attain peak performance (Flop/s)
- However, finite reuse and bandwidth limit performance.
- Assuming perfect overlap of communication and computation…

Time = max

\[
\begin{align*}
\text{FP ops / Peak GFlop/s} \\
\text{Bytes / Peak GB/s}
\end{align*}
\]
One could hope to always attain peak performance (Flop/s)
However, finite reuse and bandwidth limit performance.
Assuming perfect overlap of communication and computation...

\[
\frac{\text{Time}}{\#\text{FP ops}} = \max \left\{ \frac{1}{\text{Peak GFlop/s}}, \frac{\#\text{Bytes}}{\#\text{FP ops}} / \text{Peak GB/s} \right\}
\]
One could hope to always attain peak performance (Flop/s)
However, finite reuse and bandwidth limit performance.
Assuming perfect overlap of communication and computation…

$$\frac{\text{#FP ops}}{\text{Time}} = \min \left\{ \frac{\text{Peak GFlop/s}}{\left( \frac{\text{#FP ops}}{\text{#Bytes}} \right) \times \text{Peak GB/s}} \right\}$$
(DRAM) Roofline

- One could hope to always attain peak performance (Flop/s)
- However, finite reuse and bandwidth limit performance.
- Assuming perfect overlap of communication and computation…

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

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:
- Peak Flop/s
- Attainable Flop/s
- DRAM GB/s
- DRAM-bound
- Compute-bound

Arithmetic Intensity (Flop:Byte)
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

```c
#pragma omp parallel for
for (i=0; i<N; i++){
  Z[i] = X[i] + alpha*Y[i];
}
```
Roofline Example #2

- Conversely, 7-point constant coefficient stencil...
  - 7 flops
  - 8 memory references (7 reads, 1 store) per point
  - AI = 0.11 flops per byte (L1)

```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++){
      new[k][j][i] = -6.0*old[k][j][i ]
                    + old[k][j ][i-1]
                    + old[k][j ][i+1]
                    + old[k][j-1][i+1]
                    + old[k][j+1][i ]
                    + old[k-1][j ][i ]
                    + old[k+1][j ][i ];
    }
  }
}
```
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
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

```c
#define omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
    new[k][j][i] = -6.0*old[k][j][i] + old[k][j][i-1] + old[k][j][i+1] + old[k][j-1][i] + old[k][j+1][i] + old[k-1][j][i] + old[k+1][j][i];
}}}
```

$Gflop/s \leq AI \times DRAM\ GB/s$

![Diagram showing the roofline model with TRIAD, 7-point Stencil, and Peak Flop/s calculations.]
Refining Roofline: Memory Hierarchy
Hierarchical Roofline

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

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

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

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

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

- Applications 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 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

\[ \text{Attainable Flop/s} \]
\[ \text{DDR GB/s} \]
\[ \text{MCDRAM cache GB/s} \]
\[ \text{Arithmetic Intensity (Flop:Byte)} \]
\[ \text{Peak Flop/s} \]
Hierarchical Roofline

- Construct superposition of Rooflines...
  - Measure 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

- 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 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 FLOPS/SECOND
  - PEAK FLOPS/SECOND
Hierarchical Roofline

- Construct superposition of Rooflines...
  - Measure 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

![Diagram showing the relationship between attainable Flops/s, Arithmetic Intensity, and DDR GB/s, with MCDRAM bottleneck pulling performance below DDR Roofline.]
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
Refining Roofline: In-core Effects
In-Core Parallelism

- We have assumed one can attain peak flops with high locality.
- In reality, we must …
  - Vectorize loops (16 flops per instruction)
  - Use special instructions (e.g. FMA)
  - Ensure FP instructions dominate the instruction mix
  - Hide FPU latency (unrolling, out-of-order execution)
  - Use all cores & sockets
- Without these, …
  - Peak performance is not attainable
  - Some kernels can transition from memory-bound to compute-bound
Data Parallelism (e.g. SIMD)

- Most processors exploit some form of SIMD or vectors.
  - KNL uses 512b vectors (8x64b)
  - GPUs use 32-thread warps (32x64b)
- In reality, applications are a mix of scalar and vector instructions.
  - Performance is a weighted average between SIMD and no SIMD
Data Parallelism (e.g. SIMD)

- Most processors exploit some form of SIMD or vectors.
  - KNL uses 512b vectors (8x64b)
  - GPUs use 32-thread warps (32x64b)

- In reality, applications are a mix of scalar and vector instructions.
  - Performance is a weighted average between SIMD and no SIMD
  - There is an implicit ceiling based on this weighted average
Return of CISC

- Modern CPUs and GPUs are increasingly reliant on special (fused) instructions that perform multiple operations.
  - FMA (Fused Multiply Add): \( z = a \times x + y \) (\( z, x, y \) are vectors or scalars)
  - WMMA (Tensor Core): \( Z = AB + C \) (\( Z, A, B, C \) are FP16 matrices)

- n.b., this is orthogonal to SIMD where the the same operation(s) is applied to a vector of operands.

- Performance is now a weighted average of scalar, vector, FMA, and WMMA operations.
Return of CISC

- Total lack of FMA reduces performance by 2x on KNL. (4x on Haswell)

- In reality, applications are a mix of FMA, FAdd, and FMul.
  - Performance is a weighted average
  - There is an implicit ceiling based on this weighted average
Return of CISC

- On Volta, Tensor cores can provide 100TFLOPs of FP16 performance (vs. 7.5 TFLOPS for DP FMA)
- Observe, machine balance has now grown to …
  100 TFLOP/s / 800 GB/s
  = 125 FP16 per byte !!
Floating-Point Divides

- Although many processors support a Floating-point divide instruction, most implement divides through a sequence of FP instructions:
  - `rcp` (reciprocal estimate to k bits)
  - Newton-Raphson iterations (mul+FMA) to recover full precision
- All of these instructions can be pipelined and/or executed out of order

- FP Divides increase arithmetic intensity and increase raw Flop rates.
Floating-Point Divides

- #FP operations deduced from source code can be an underestimate…
  - FP Divides require 10+ instructions on KNL and GPUs.
  - These must be included in both AI and Flop/s to affect proper Roofline analysis

- As a result, AI and performance both increase and one can be compute-bound
Superscalar vs. Instruction mix

- Superscalar processors have finite instruction fetch/decode/issue bandwidth (e.g. 4 instructions per cycle)
- Moreover, the number of FP units dictates the FP issue rate required to hit peak (e.g. 2 vector instructions per cycle)

➢ Ratio of these two rates is the FP instruction fraction required to hit peak
Superscalar vs. Instruction mix

- Haswell CPU
  - 4-issue superscalar
  - Only 2 FP data paths
  - Requires 50% of the instructions to be FP to get peak performance
Superscalar vs. Instruction mix

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

- **Conversely, on KNL…**
  - 2-issue superscalar
  - 2 FP data paths
  - Requires 100% of the instructions to be FP to get peak performance
Superscalar vs. Instruction mix

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

- Conversely, on KNL…
  - 2-issue superscalar
  - 2 FP data paths
  - Requires 100% of the instructions to be FP to get peak performance
Superscalar vs. Instruction mix

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

- **Conversely, on KNL…**
  - 2-issue superscalar
  - 2 FP data paths
  - Requires 100% of the instructions to be FP to get peak performance

- Codes that would have been memory-bound are now decode/issue-bound.
Superscalar vs. Instruction mix

- On Volta, each SM is partitioned among 4 warp schedulers
- Each warp scheduler can dispatch 32 threads per cycle
- However, it can only execute 8 DP FP instructions per cycle.
- i.e. there is plenty of excess instruction issue bandwidth available for non-FP instructions.
Refining Roofline: Locality Effects
Naively, we can bound AI using only compulsory cache misses

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

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

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

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

$\text{AI} = \frac{\#\text{Flop's}}{\text{Compulsory Misses + Write Allocates + Capacity Misses}}$

Know the theoretical bounds on your AI.
Overview of Roofline Methodology
Machine Characterization

- “Theoretical Performance” numbers can be highly optimistic…
  - Pin BW vs. sustained bandwidth
  - TurboMode at low concurrency
  - Underclocking for AVX
  - Compiler failing on high-AI loops.

- Take marketing numbers with a grain of salt
Machine Characterization

- To create a Roofline model, we must benchmark...
  - **Sustained Flops**
    - Double/single/half precision
    - With and without FMA (e.g. compiler flag)
    - With and without SIMD (e.g. compiler flag)
  - **Sustained Bandwidth**
    - Measure between each level of memory/cache
    - Iterate on working sets of various sizes and identify plateaus
    - Identify bandwidth asymmetry (read:write ratio)

- Benchmark must run long enough to observe effects of power throttling
Measuring Application AI and Performance

- To characterize execution with Roofline we need...
  - **Time**
  - **Flops** ($\rightarrow$ flop’s / time)
  - **Data movement** between each level of memory ($\rightarrow$ Flop’s / GB’s)

- We can look at the full application...
  - Coarse grained, 30-min average
  - Misses many details and bottlenecks

- or we can look at individual loop nests...
  - Requires auto-instrumentation on a loop by loop basis
  - Moreover, we should probably differentiate data movement or flops on a core-by-core basis.
<table>
<thead>
<tr>
<th>Manual Counting</th>
<th>Perf. Counters</th>
<th>Binary Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go thru each loop nest and count the number of FP operations</td>
<td>Read counter before/after</td>
<td>Automated inspection of assembly at run time</td>
</tr>
<tr>
<td>Works best for deterministic loop bounds</td>
<td>More Accurate</td>
<td>Most Accurate</td>
</tr>
<tr>
<td>or parameterize by the number of iterations (recorded at run time)</td>
<td>Low overhead (&lt;%) == can run full MPI applications</td>
<td>FMA-, VL-, and mask-aware</td>
</tr>
<tr>
<td>Not scalable</td>
<td>Can detect load imbalance</td>
<td>Can count instructions by class/type</td>
</tr>
</tbody>
</table>

- Requires privileged access
- Requires manual instrumentation (+overhead) or full-app characterization
- Broken counters = garbage
- May not differentiate FMADD from FADD
- No insight into special pipelines

- >10x overhead (short runs / reduced concurrency)
# How Do We Measure Data Movement?

## Manual Counting
- Go thru each loop nest and estimate how many bytes will be moved
- Use a mental model of caches
  - Works best for simple loops that stream from DRAM (stencils, FFTs, spare, …)
  - N/A for complex caches
  - Not scalable

## Perf. Counters
- Read counter before/after
  - Applies to full hierarchy (L2, DRAM)
  - Much more Accurate
  - Low overhead (<%) == can run full MPI applications
  - Can detect load imbalance
  - Requires privileged access
  - Requires manual instrumentation (+overhead) or full-app characterization

## Cache Simulation
- Build a full cache simulator driven by memory addresses
  - Applies to full hierarchy and multicore
  - Can detect load imbalance
  - Automated application to multiple loop nests
  - Ignores prefetchers
  - >10x overhead (short runs / reduced concurrency)
Roofline-Driven Optimization
Why is Roofline Useful?

- Imagine a mix of loop nests
- Flop/s alone may not be useful in deciding which to optimize first
Why is Roofline Useful?

- We can sort kernels by AI …
Why is Roofline Useful?

- We can sort kernels by AI ...
- … and compare performance relative to machine capabilities
Why is Roofline Useful?

- Kernels near the roofline are making good use of computational resources
  - kernels can have low performance (Gflop/s), but make good use of a machine
  - kernels can have high performance (Gflop/s), but make poor use of a machine
Tracking Progress Towards Optimality

- One can conduct a Roofline optimization after every optimization (or once per quarter)
  - Tracks progress towards optimality
  - Allows one to quantitatively speak to ultimate performance / KPPs
  - Can be used as a motivator for new algorithms.
Often, one plots performance as a function of thread concurrency

- Carries no insight or analysis
- Provides no actionable information.
Roofline Scaling Trajectories

- Often, one plots performance as a function of thread concurrency
  - Carries no insight or analysis
  - Provides no actionable information.

- Khaled Ibrahim developed a new way of using Roofline to analyze thread (or process) scalability
  - Create a 2D scatter plot of performance as a function of AI and thread concurrency
  - Can identify loss in performance due to increased cache pressure

Roofline Scaling Trajectories

- Observe...
  - AI (data movement) varies with both thread concurrency and problem size.
  - Large problems (green and red) move much more data per thread, and eventually exhaust cache capacity.
  - Resultant fall in AI means they hit the bandwidth ceiling quickly and degrade.
  - Smaller problems see reduced AI, but don’t hit the bandwidth ceiling.
Driving Performance Optimization

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

- Broadly speaking, there are three approaches to improving performance:
  - Maximize in-core performance (e.g. get compiler to vectorize)
Driving Performance Optimization

- 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, unit-stride)
Driving Performance Optimization

- 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, unit stride)
- Minimize data movement (e.g. cache blocking)
Summary
Summary

In this talk, we introduced several concepts…

- Basic terminology (bandwidth, flop/s, arithmetic intensity)
- How to refine the Roofline to account for the memory hierarchy
- How to refine the Roofline to account for complex core architectures
- How to map the 3Cs of Caches onto the Roofline model
- General approaches to constructing a Roofline model for a machine and application
- How to use the Roofline model
Questions?
Backup