Title
The roofline model: A pedagogical tool for program analysis and optimization

Permalink
https://escholarship.org/uc/item/3qf383m0

ISBN
9781467388719

Authors
Williams, S
Patterson, D
Oliker, L
et al.

Publication Date
2016-05-20

DOI
10.1109/HOTCHIPS.2008.7476531

Peer reviewed
The Roofline Model:
A pedagogical tool for program analysis and optimization

Samuel Williams$^{1,2}$, David Patterson$^1$, Leonid Oliker$^{1,2}$, John Shalf$^2$, Katherine Yelick$^{1,2}$

$^1$University of California, Berkeley
$^2$Lawrence Berkeley National Laboratory

samw@cs.berkeley.edu
Outline

- Motivation, Goals, Audience, etc…
- Survey of multicore architectures
- Description of the Roofline model
- Introduction to Auto-tuning
- Application of the roofline to auto-tuned kernels
  - Example #1 - SpMV
  - Example #2 - LBMHD
- Conclusions
Motivation

- Multicore guarantees neither good scalability nor good (attained) performance
- Performance and scalability can be extremely non-intuitive even to computer scientists

- Success of the multicore paradigm seems to be premised upon their programmability
- To that end, one must understand the limits to both scalability and efficiency.

- How can we empower programmers?
Primary Focus

- Throughput-oriented kernels (rather than time)
- Our performance metrics are: Gflop/s and % of peak (efficiency)

for purposes of this talk, I will focus on memory-intensive 64b floating-point SPMD kernels.

- Not focused on algorithmic innovation or computational complexity
Goals & Audience

❖ Goals for Roofline:
   ▪ Provide everyone (especially undergrads) with a graphical aid that provides: **realistic expectations of performance and productivity**
   ▪ Show inherent hardware limitations for a given kernel
   ▪ Show potential benefit and priority of optimizations

❖ Who’s not the audience for the Roofline:
   ▪ Not for those interested in fine tuning (+10%)
   ▪ Not for those challenged by parallel kernel correctness
Multicore SMPs of Interest

(used throughout the rest of the talk)
Multicore SMPs Used

Intel Xeon E5345 (Clovertown)

- 4 Coherency Hubs
- 2x128b controllers
- 667MHz FBDIMMs
- 4MB shared L2
- FSB 10.66 GB/s
- Chipset (4x64b controllers)
- 21.33 GB/s (read)
- 10.66 GB/s (write)

AMD Opteron 2356 (Barcelona)

- 2x64b memory controllers
- 667MHz DDR2 DIMMs
- 512KB victim
- 4MB Shared L2 (16 way) (64b interleaved)
- 2MB Shared quasi-victim (32 way)
- HyperTransport
- SRI / crossbar
- 4GB/s (each direction)

Sun T2+ T5140 (Victoria Falls)

- 4MB shared L2 (16 way) (64b interleaved)
- 4 Coherency Hubs
- 2x128b controllers
- 667MHz FBDIMMs
- 179 GB/s
- 20 GB/s
- 10.66 GB/s

IBM QS20 Cell Blade

- 512K L2
- Crossbar
- EIB (ring network)
- XDR memory controllers
- 25.6 GB/s
- 512MB XDR DRAM
- 512K L2
Multicore SMPs Used

Intel Xeon E5345 (Clovertown)

- 8 cores
- 4MB shared L2 cache per core
- 10.66 GB/s (write)
- 21.33 GB/s (read)
- 667MHz FBDIMMs

AMD Opteron 2356 (Barcelona)

- 8 cores
- 512KB victim
- 2MB Shared quasi-victim (32 way)
- SRI / crossbar
- 667MHz DDR2 DIMMs

Sun T2+ 75140 (Victoria Falls)

- 8 cores
- 4MB shared L2 cache (64b interleaved)
- 2x128b controllers
- 21.33 GB/s (read)
- 10.66 GB/s (write)
- 667MHz FBDIMMs

IBM QS20 Cell Blade

- 8 cores
- 512K L2 cache
- 25.6 GB/s (each direction)
- 667MHz DDR2 DIMMs

Conventional Cache-based Memory Hierarchy

Disjoint Local Store Memory Hierarchy
Multicore SMPs Used

Intel Xeon E5345 (Clovertown)

Core | Core | Core | Core
-----|------|------|------
4MB shared L2 | 4MB shared L2 | 4MB shared L2 | 4MB shared L2

FSB 10.66 GB/s
Chipset (4x64b controllers)
21.33 GB/s (read) 10.66 GB/s (write)
667MHz FBDIMMs

AMD Opteron 2356 (Barcelona)

Option | Option | Option | Option
-------|-------|-------|-------
512KB victim | 512KB victim | 512KB victim | 512KB victim
2MB Shared quasi-victim (32 way)
SRI / crossbar

HyperTransport

Option | Option | Option | Option
-------|-------|-------|-------
512KB victim | 512KB victim | 512KB victim | 512KB victim
2MB Shared quasi-victim (32 way)
SRI / crossbar

HyperTransport

2x64b memory controllers
667MHz DDR2 DIMMs
667MHz DDR2 DIMMs

Sun T2+ T5140 (Victoria Falls)

4MB Shared L2 (1/16 way) (64b interleaved)
4 Coherence Hubs
2x128b controllers

179 GB/s (read) 190 GB/s (write)
667MHz FBDIMMs

IBM QS20 Cell Blade

VMT PPE

MT SPARC MT SPARC MT SPARC MT SPARC
MT SPARC MT SPARC MT SPARC MT SPARC
Crossbar

179 GB/s (read) 190 GB/s (write)
667MHz FBDIMMs

VMT PPE

MT SPARC MT SPARC MT SPARC MT SPARC
MT SPARC MT SPARC MT SPARC MT SPARC
Crossbar

25.6 GB/s XDR memory controllers
512MB XDR DRAM
Multicore SMPs Used
(peak double precision flops)

Intel Xeon E5345 (Clovertown)

- 75 GFlop/s

AMD Opteron 2356 (Barcelona)

- 74 Gflop/s

Sun T2+ T5140 (Victoria Falls)

- 19 GFlop/s

IBM QS20 Cell Blade

- 29* GFlop/s

*SPEs only
Multicore SMPs Used
(total DRAM bandwidth)

Intel Xeon E5345 (Clovertown)
- 21 GB/s (read)
- 10 GB/s (write)

AMD Opteron 2356 (Barcelona)
- 21 GB/s

Sun T2+ T5140 (Victoria Falls)
- 42 GB/s (read)
- 21 GB/s (write)

IBM QS20 Cell Blade
- 51* GB/s

*SPEs only
Roofline models for multicore SMPs

(for memory-intensive double precision floating-point kernels)
Arithmetic Intensity in HPC

- True Arithmetic Intensity (AI) \( \sim \) Total Flops / Total DRAM Bytes
  - constant with respect to problem size for many problems of interest
  - ultimately limited by compulsory traffic
  - diminished by conflict or capacity misses.

O(1) O(log(N)) O(N)
Naïve Roofline Model

- Based on **Bound and Bottleneck analysis**\(^1\)

- Performance is upper bounded by both the peak flop rate, and the product of streaming bandwidth and the flop:byte ratio
- (well understood in the performance oriented communities)

\[
\text{Gflop/s} (\text{AI}) = \min \left\{ \begin{array}{c}
\text{Peak Gflop/s} \\
\text{AI} \times \text{StreamBW}
\end{array} \right. 
\]

\[where\ \text{AI}\ \text{is\ the\ actual\ arithmetic\ intensity}\]

- Assumptions:
  - Bandwidth is independent on arithmetic intensity
  - Bandwidth is independent of optimization or access pattern
  - Computation is independent of optimization
  - Complete overlap of either communication or computation

\(^1\)D. Lazowska, J. Zahorjan, G. Graham, K. Sevcik, “Quantitative System Performance”
Naïve Roofline Model

- Unrealistically optimistic model
- Hand optimized Stream BW benchmark

- Intel Xeon E5345 (Clovertown)
- Opteron 2356 (Barcelona)
- Sun T2+ T5140 (Victoria Falls)
- IBM QS20 Cell Blade

Graphs showing the relationship between flop:DRAM byte ratio and attainable Gflop/s for different processors.
Naïve Roofline Model

- Unrealistically optimistic model
- Hand optimized Stream BW benchmark

This level of performance is only attainable with extensive optimizations.
Naïve Roofline Model

- Unrealistically optimistic model
- Hand optimized Stream BW benchmark

How sensitive is each architecture to removing those optimizations?
Better Roofline

- Collect \( \text{StreamBW}_j \) with progressively fewer optimizations
- Estimate \( \text{InCoreGFlops}_i \) with progressively fewer optimizations

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

\( \text{is the attainable performance with:} \)

- memory optimizations\(_{1\ldots i}\) - and -
- in-core optimizations\(_{1\ldots j}\)

- These denote a series of ceilings below the roofline

- Assumptions:
  - Bandwidth is independent on arithmetic intensity
  - Complete overlap of either communication or computation
Roofline models
(dram bandwidth)

- What happens as bandwidth optimizations are stripped out?
- Form a series of bandwidth ceilings below the roofline
- Small problems fit in the snoop filter in Clovertown’s MCH
- Most architectures see NUMA and prefetch variations

Graphs showing the relationship between attainable Gflop/s and flop:DRAM byte ratio for different processors:

- Intel Xeon E5345 (Clovertown)
- Opteron 2356 (Barcelona)
- Sun T2+ T5140 (Victoria Falls)
- IBM QS20 Cell Blade
Roofline models
(dram bandwidth)

- What happens as bandwidth optimizations are stripped out?
- Form a series of bandwidth ceilings below the roofline
- Small problems fit in the snoop filter in Clovertown’s MCH
- Most architectures see NUMA and prefetch variations
Roofline models
(dram bandwidth)

- What happens as bandwidth optimizations are stripped out?
- Form a series of bandwidth ceilings below the roofline
- Small problems fit in the snoop filter in Clovertown’s MCH
- most architectures see NUMA and prefetch variations
Roofline models
(dram bandwidth)

- What happens as bandwidth optimizations are stripped out?
- Form a series of bandwidth ceilings below the roofline
- Small problems fit in the snoop filter in Clovertown’s MCH
- Most architectures see NUMA and prefetch variations

Graphs showing:
- Intel Xeon E5345 (Clovertown): dataset fits in snoop filter, hand optimized Stream BW, peak DP
- Opteron 2356 (Barcelona): hand optimized Stream BW, peak DP
- Sun T2+ T5140 (Victoria Falls): hand optimized Stream BW, peak DP
- IBM QS20 Cell Blade: hand optimized Stream BW, peak DP

Flop:DRAM byte ratio:
- Intel Xeon E5345: 1/16 to 1/2
- Opteron 2356: 1/16 to 1/2
- Sun T2+ T5140: 1/16 to 1/2
- IBM QS20 Cell Blade: 1/16 to 1/2

Attainable Gflop/s:
- Intel Xeon E5345: 1 to 128
- Opteron 2356: 1 to 128
- Sun T2+ T5140: 1 to 128
- IBM QS20 Cell Blade: 1 to 128

Pick: flop:DRAM byte ratio, attainable Gflop/s.
Roofline models
(dram bandwidth)

What happens as bandwidth optimizations are stripped out?

Form a series of bandwidth ceilings below the roofline

Small problems fit in the snoop filter in Clovertown’s MCH

most architectures see NUMA and prefetch variations
Roofline models
(dram bandwidth)

- What happens as bandwidth optimizations are stripped out?
- Form a series of bandwidth ceilings below the roofline
- Small problems fit in the snoop filter in Clovertown’s MCH
- Most architectures see NUMA and prefetch variations

**Graphs:**
- Intel Xeon E5345 (Clovertown)
- Opteron 2356 (Barcelona)
- Sun T2+ T5140 (Victoria Falls)
- IBM QS20 Cell Blade
In-core Performance

- Define a similar set of ceilings for in-core performance

- In-core performance can be limited by (among other things):
  - Not satisfying all forms of in-core parallelism:
    - Instruction-level parallelism (multi-issue, pipeline, …)
    - Data-level parallelism (SIMD)
    - Functional unit parallelism (adders + multipliers + …)
  - Non-FP instructions can consume instruction issue bandwidth
    - As the FP fraction decrease, how sensitive is attainable performance?

- One or the other is usually more difficult to satisfy on a given architecture/kernel
  = Architecture’s Achilles’ Heel
Roofline models
(in-core performance = in-core parallelism?)

- Covering the breadth of in-core parallelism is the preeminent challenge on most architectures
- Form a series of parallelism ceilings below the roofline
- On Niagara machines, instruction latencies are easily hidden with 8-way multithreading
Roofline models
(in-core performance = instruction mix?)

- All machines have a limited instruction issue bandwidth.
- Non-FP instructions sap instruction issue bandwidth needed by FP instructions.
- As the FP fraction of the dynamic instruction mix decreases, so might performance.
- On Cell, double precision instructions stall subsequent issues for 7 cycles.
Roofline models
(Achilles' Heel)

- It's clear that in-core parallelism is more important on the superscalars.
- Instruction mix is more important on Niagara2.
- Each architecture has its own Achilles' Heel when it comes to in-core performance.
Roofline models
(ceilings constrain performance)

The ceilings act to constrain performance to a much smaller region.
Roofline models
(ceilings constrain performance)

The ceilings act to constrain performance to a much smaller region.
Roofline models

(thickness)

The ceilings act to constrain performance to a much smaller region.

Thickness of the roofline is indicative of requisite compiler or SW complexity.
Three Categories of Software Optimization
Optimization Categorization

Maximizing In-core Performance

- Exploit in-core parallelism (ILP, DLP, etc…)
- Good (enough) floating-point balance

Maximizing Memory Bandwidth

- Exploit NUMA
- Hide memory latency
- Satisfy Little’s Law

Minimizing Memory Traffic

- Eliminate:
  - Capacity misses
  - Conflict misses
  - Compulsory misses
  - Write allocate misses

- Exploit NUMA
- Hide memory latency
- Satisfy Little’s Law
Maximizing Attained in-core Performance

- Compilers may not have as much knowledge as the programmer
- Express more in-core parallelism and amortize non-FP instructions
- Software optimizations:
  - Explicit SIMDization
  - Loop unrolling
  - Unroll and jam
  - Reordering
  - Predication
- Punch through ceilings
Maximizing Attained Memory Bandwidth

- Compilers won’t give great out-of-the-box bandwidth
- Optimizations:
  - long unit stride accesses
  - NUMA aware allocation and parallelization
  - SW prefetching
  - Maximize MLP
- Punch through bandwidth ceilings

Graph showing AMD Opteron 2356 (Barcelona) benchmark results with various optimizations and bandwidth limitations.
Minimizing Total Memory Traffic

- Use performance counters to measure flop:byte ratio (AI)
- Out-of-the-box code may have an AI ratio much less than the compulsory ratio
- Optimizations:
  - Array padding: conflict
  - Cache blocking: capacity
  - Cache bypass: compulsory
- Push arithmetic intensity to the compulsory limit
Optimization Categorization

Maximizing In-core Performance

- Exploit in-core parallelism (ILP, DLP, etc...)
- Good (enough) floating-point balance

Maximizing Memory Bandwidth

- Exploit NUMA
- Hide memory latency
- Satisfy Little’s Law

Minimizing Memory Traffic

- Eliminate:
  - Capacity misses
  - Conflict misses
  - Compulsory misses
  - Write allocate behavior

Each optimization has a large parameter space. What are the optimal parameters?
Introduction to Auto-tuning
Out-of-the-box Code

- Out-of-the-box code has (unintentional) assumptions on:
  - cache sizes (>10MB)
  - functional unit latencies (~1 cycle)
  - etc...

- These assumptions may result in poor performance when they exceed the machine characteristics
What is auto-tuning?

- Goal: provide **performance portability** across the existing breadth and evolution of microprocessors
- At the expense of a one time up front productivity cost that’s amortized by the number of machines it’s used on

- Auto-tuning does not invent new optimizations
- **Auto-tuning automates the exploration of the optimization and parameter space**
- Two components:
  1. parameterized code generator (we wrote ours in Perl)
  2. Auto-tuning exploration benchmark
     (combination of heuristics and exhaustive search)
- Can be extended with ISA specific optimizations (e.g. DMA, SIMD)
Distinguishing the Roofline and Auto-tuning

- Roofline specifies what's deficient, but not how to fix it.

- Auto-tuning attempts to fix it by searching the parameter space for the existing body of optimization work.
Application of the Roofline Model to sample Kernels

*Does the roofline model provide insight into the limitations of architecture, implementation, and algorithm?*
Things to watch for:

1. do performance graphs alone provide insight into the limitations of kernel or architecture?

2. does the roofline show the ultimate performance limitations of kernel and architecture?

3. does the roofline show which optimizations will be necessary?
Example #1: Auto-tuning Sparse Matrix-Vector Multiplication (SpMV)

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 $y = Ax$
  - $A$ 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 superscalar),
  - Difficult to exploit DLP (bad for SIMD)

(a) algebra conceptualization  
(b) CSR data structure  
(c) CSR reference code

```
for (r=0; r<A.rows; r++) {
    double y0 = 0.0;
    for (i=A.rowStart[r]; i<A.rowStart[r+1]; i++){
        y0 += A.val[i] * x[A.col[i]];
    }
    y[r] = y0;
}
```
The Dataset (matrices)

- Unlike dense BLAS, performance is dictated by sparsity
- Suite of 14 matrices
- All bigger than the caches of our SMPs
- We’ll also include a median performance number

2K x 2K Dense matrix stored in sparse format

Well Structured (sorted by nonzeros/row)
- Protein
- FEM / Spheres
- FEM / Cantilever
- Wind Tunnel
- FEM / Harbor
- QCD
- FEM / Ship
- Economics
- Epidemiology

Poorly Structured hodgepodge
- FEM / Accelerator
- Circuit
- webbase

Extreme Aspect Ratio (linear programming)
- LP
SpMV Performance
(simple parallelization)

- Out-of-the-box SpMV performance on a suite of 14 matrices
- Scalability isn’t great
- Is this performance good?

**Xeon E5345 (Clovertown)**

- GFLOP/s
- Parallel
- Naïve

**Opteron 2356 (Barcelona)**

- GFLOP/s
- Parallel
- Naïve

**UltraSparc T2+ T5140 (Victoria Falls)**

- GFLOP/s
- Parallel
- Naïve

**QS20 Cell Blade (PPEs)**

- GFLOP/s
- Parallel
- Naïve
Auto-tuned SpMV Performance
(architecture specific optimizations)

- Fully auto-tuned SpMV performance across the suite of matrices
- Included SPE/local store optimized version
- Why do some optimizations work better on some architectures?
- Performance is better, but is performance good?

Graphs showing performance of different architectures:

- Xeon E5345 (Clovertown)
- Opteron 2356 (Barcelona)
- UltraSparc T2+ T5140 (Victoria Falls)
- QS20 Cell Blade (SPEs)

Legend:

- +Cache/LS/TLB Blocking
- +Matrix Compression
- +SW Prefetching
- +NUMA/Affinity
- Naïve Pthreads
- Naïve
Auto-tuned SpMV Performance (architecture specific optimizations)

- Fully auto-tuned SpMV performance across the suite of matrices
- Included SPE/local store optimized version
- Why do some optimizations work better on some architectures?
- Performance is better, but is performance good?

Auto-tuning resulted in better performance, but did it result in good performance?
Roofline model for SpMV

- Double precision roofline models
- FMA is inherent in SpMV (place at bottom)

<table>
<thead>
<tr>
<th>Processor</th>
<th>Available Gflop/s</th>
<th>flop:DRAM byte ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon E5345</td>
<td>128</td>
<td>1</td>
</tr>
<tr>
<td>Opteron 2356</td>
<td>128</td>
<td>1</td>
</tr>
<tr>
<td>Sun T2+ T5140</td>
<td>128</td>
<td>1</td>
</tr>
<tr>
<td>IBM QS20 Cell Blade</td>
<td>128</td>
<td>1</td>
</tr>
</tbody>
</table>

- peak DP
- w/out SIMD
- w/out ILP
- w/out FMA
- mul/add imbalance
- dataset fits in snoop filter
- w/out SW prefetch
- w/out NUMA

- 25% FP
- 12% FP

- IBM QS20 Cell Blade
- Intel Xeon E5345 (Clovertown)
- Opteron 2356 (Barcelona)
- Sun T2+ T5140 (Victoria Falls)
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

Intel Xeon E5345 (Clovertown)
- peak DP
- w/out SIMD
- w/out ILP
- mull/add imbalance

Opteron 2356 (Barcelona)
- peak DP
- w/out SIMD
- w/out ILP

Sun T2+ T5140 (Victoria Falls)
- peak DP
- 25% FP
- 12% FP
- w/out SW prefetch
- w/out NUMA

IBM QS20 Cell Blade
- peak DP
- w/out SIMD
- w/out ILP
- w/out FMA

No naïve Cell implementation
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

Intel Xeon E5345 (Clovertown)
- peak DP
- w/out SIMD
- w/out ILP
- w/out NUMA
- w/out unaligned DMA

Opteron 2356 (Barcelona)
- peak DP
- w/out SIMD
- w/out ILP
- w/out FMA

Sun T2+ T5140 (Victoria Falls)
- peak DP
- 25% FP
- 12% FP
- w/out SIMD
- w/out ILP
- w/out NUMA

IBM QS20 Cell Blade
- peak DP
- w/out SIMD
- w/out ILP
- w/out FMA

No naïve Cell implementation
Roofline model for SpMV (matrix compression)

- Inherent FMA
- Register blocking improves ILP, DLP, flop:byte ratio, and FP% of instructions

**Intel Xeon E5345 (Clovertown)**
- peak DP
- w/out SIMD
- w/out ILP
- mull/add imbalance

**Opteron 2356 (Barcelona)**
- peak DP
- w/out SIMD
- w/out ILP
- mull/add imbalance

**Sun T2+ T5140 (Victoria Falls)**
- peak DP
- 25% FP
- w/out SIMD
- w/out ILP
- w/out SW prefetch
- w/out NUMA

**IBM QS20 Cell Blade**
- peak DP
- w/out SIMD
- w/out ILP
- w/out FMA
- 25% FP
Roofline model for SpMV
(matrix compression)

- Inherent FMA
- Register blocking improves ILP, DLP, flop:byte ratio,
  and FP% of instructions

All machines are on the bandwidth roofline!
Example #2: Auto-tuning Lattice-Boltzmann Magneto-Hydrodynamics (LBMHD)


Best Paper, Application Track
Plasma turbulence simulation via Lattice Boltzmann Method

- Two distributions:
  - momentum distribution (27 scalar components)
  - magnetic distribution (15 vector components)

- Three macroscopic quantities:
  - Density
  - Momentum (vector)
  - Magnetic Field (vector)

- Arithmetic Intensity:
  - Must read 73 doubles, and update 79 doubles per lattice update (1216 bytes)
  - Requires about 1300 floating point operations per lattice update
    - Just over 1.0 flops/byte (ideal)

- Out-of-the-box, no unit stride memory access patterns
Initial LBMHD Performance

- Generally, scalability looks good
- but is performance good?

```
collision() only
```
Auto-tuned LBMHD Performance
(architecture specific optimizations)

- Auto-tuning avoids cache conflict and TLB capacity misses
- Additionally, it exploits SIMD where the compiler doesn’t
- Include a SPE/Local Store optimized version

- +small pages
- +Explicit SIMDization
- +SW Prefetching
- +Unrolling
- +Vectorization
- +Padding
- Naïve+NUMA

*collision() only*
Roofline model for LBMHD

- Far more adds than multiplies (imbalance)
- Huge data sets

Graphs showing performance for:
- Intel Xeon E5345 (Clovertown)
- Opteron 2356 (Barcelona)
- Sun T2+ T5140 (Victoria Falls)
- IBM QS20 Cell Blade

Performance metrics include:
- Peak DP
- Mul/add imbalance
- Without NUMA
- Without SW prefetch
- Without ILP
- Without SIMD
- Dataset fits in snoop filter

Graphs illustrating flop:DRAM byte ratio and attainable Gflop/s for various configurations and platforms.
Roofline model for LBMHD
(overlay arithmetic intensity)

- Far more adds than multiplies (imbalance)
- Essentially random access to memory
- Flop:byte ratio ~0.7
- NUMA allocation/access
- Little ILP
- No DLP
- High conflict misses

**Intel Xeon E5345** (Clovertown)
- Peak DP
- Mul/add imbalance
- W/out SIMD
- W/out ILP
- Dataset fits in snoop filter

**Opteron 2356** (Barcelona)
- Peak DP
- Mul/add imbalance
- W/out SIMD
- W/out ILP

**Sun T2+ T5140** (Victoria Falls)
- Peak DP
- 25% FP
- 12% FP
- W/out NUMA
- W/out ILP

**IBM QS20 Cell Blade**
- No naïve Cell implementation
- W/out SIMD
- W/out ILP
Roofline model for LBMHD
(out-of-the-box parallel performance)

- Far more adds than multiplies (imbalance)
- Essentially random access to memory
- Flop:byte ratio ~0.7
- NUMA allocation/access
- Little ILP
- No DLP
- High conflict misses

Peak VF performance with 64 threads (out of 128) - high conflict misses

No naïve Cell implementation
Roofline model for LBMHD
(Padding, Vectorization, Unrolling, Reordering, …)

- Vectorize the code to eliminate TLB capacity misses
- Ensures unit stride access (bottom bandwidth ceiling)
- Tune for optimal VL
- Clovertown pinned to lower BW ceiling

**Intel Xeon E5345 (Clovertown)**

- Peak DP
- Mul/add imbalance
- W/out SIMD
- W/out ILP
- Dataset fits in snoop filter

**Opteron 2356 (Barcelona)**

- Peak DP
- Mul/add imbalance
- W/out SIMD
- W/out ILP

**Sun T2+ T5140 (Victoria Falls)**

- 25% FP
- 12% FP
- W/out SIMD
- W/out ILP

**IBM QS20 Cell Blade**

- Peak DP
- Mul/add imbalance
- W/out SIMD
- W/out ILP

No naïve Cell implementation
Roofline model for LBMHD
(SIMDization + cache bypass)

- Make SIMDization explicit
- Technically, this swaps ILP and SIMD ceilings
- Use cache bypass instruction: *movntpd*
- Increases flop:byte ratio to ~1.0 on x86/Cell
Roofline model for LBMHD
(SIMDization + cache bypass)

- Make SIMDization explicit
- Technically, this swaps ILP and SIMD ceilings
- Use cache bypass instruction: `movntpd`
- Increases flop:byte ratio to ~1.0 on x86/Cell

3 out of 4 machines hit the Roofline
Conclusions
Summary

- The Roofline model is a visually intuitive figure for kernel analysis and optimization
- We believe undergraduates will find it useful in assessing performance and scalability limitations

- It is easily extended to other architectural paradigms
- We believe it is easily extendable to other metrics:
  - performance (sort, graphics, crypto…)
  - bandwidth (L2, PCIe, …)

- We believe that a performance counters could be used to generate a **runtime-specific** roofline that would greatly aide the optimization
As architectures are presented over the next two days, we invite you to create a roofline model for each. Estimate the ceilings.

Then contemplate performance and productivity among them.
Acknowledgements

- Research supported by:
  - Microsoft and Intel funding (Award #20080469)
  - DOE Office of Science under contract number DE-AC02-05CH11231
  - NSF contract CNS-0325873
  - Sun Microsystems - Niagara2 / Victoria Falls machines
  - AMD - access to Quad-core Opteron (barcelona) access
  - Forschungszentrum Jülich - access to QS20 Cell blades
  - IBM - virtual loaner program to QS20 Cell blades
Questions?


Best Paper, Application Track