A Vision for Integrating Performance Counters into the Roofline model

Samuel Williams\textsuperscript{1,2}

\textit{samw@cs.berkeley.edu}

with Andrew Waterman\textsuperscript{1}, Heidi Pan\textsuperscript{1,3},
David Patterson\textsuperscript{1}, Krste Asanovic\textsuperscript{1}, Jim Demmel\textsuperscript{1}

\textsuperscript{1}University of California, Berkeley
\textsuperscript{2}Lawrence Berkeley National Laboratory
\textsuperscript{3}Massachusetts Institute of Technology
Outline

- Auto-tuning
  - Introduction to Auto-tuning
  - BeBOP’s previous performance counter experience
  - BeBOP’s current tuning efforts

- Roofline Model
  - Motivating Example - SpMV
  - Roofline model
  - Performance counter enhanced Roofline model
Motivation
(Folded into Jim’s Talk)
In economics, the Gini coefficient is a measure of the distribution of wealth within a society. As wealth becomes concentrated, the value of the coefficient increases, and the curve departs from a straight line.

It’s just an assessment of the distribution, not a commentary on what it should be.

http://en.wikipedia.org/wiki/Gini_coefficient
What’s the Gini Coefficient for our Society?

- By **our society**, I mean those working in the performance optimization and analysis world (tuners, profilers, counters)
- Our **wealth**, is knowledge of tools and benefit gained from them.
Why is it so low?

- **Apathy**
  - Performance only matters after correctness
  - Scalability has won out over efficiency
  - Timescale for Moore’s law has been shorter than optimization
- **Ignorance / Lack of Specialized Education**
  - Tools assume broad and deep architectural knowledge
  - Optimization may require detailed application knowledge
- **Significant SysAdmin support**
- **Cryptic tools/presentation**
- **Erroneous data**
- **Frustration**
Certainly unreasonable for every programmer to be cognizant of performance counters

Equally unreasonable for the benefit to be uniform

Making performance tools
- more intuitive
- more robust
- easier to use (always on?)
- essential in a multicore era

will motivate more users to exploit them

Oblivious to programmers, compilers, architectures, and middleware may exploit performance counters to improve performance
auto-tuning & performance counter experience
Introduction to Auto-tuning
Out-of-the-box Code Problem

- 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
Auto-tuning?

- Trade up front loss in productivity cost for continued reuse of automated kernel optimization on other architectures

- Given existing optimizations, Auto-tuning automates the exploration of the optimization and parameter space

- Two components:
  - parameterized code generator (we wrote ours in Perl)
  - Auto-tuning exploration benchmark (combination of heuristics and exhaustive search)

- Auto-tuners that generate C code provide performance portability across the existing breadth of architectures

- Can be extended with ISA specific optimizations (e.g. DMA, SIMD)

(Breadth of Existing Architectures)
BeBOP’s Previous Performance Counter Experience
(2-5 years ago)
Perennially, performance counters have been used
- as a *post-mortem* to validate auto-tuning heuristics
- to bound remaining performance improvement
- to understand unexpectedly poor performance

However, this requires:
- significant kernel and architecture knowledge
- creation of a performance model specific to each kernel
- calibration of the model

Summary: We’ve experienced a progressively lower benefit and confidence in their use due to the variation in the quality and documentation of performance counter implementations
Sparse Matrix Vector Multiplication (SpMV)

- "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply”
  - Applied to older Sparc, Pentium III, Itanium machines
  - Model cache misses (compulsory matrix or compulsory matrix+vector)
  - Count cache misses via PAPI
  - Generally well bounded (but large performance bound)

- "When Cache Blocking Sparse Matrix Vector Multiply Works and Why”
  - Similar architectures
  - Adds a fully associative TLB model (benchmarked TLB miss penalty)
  - Count TLB misses (as well as cache misses)
  - Much better correlation to actual performance trends

- Only modeled and counted the total number of misses (bandwidth only).
- Performance counters didn’t distinguish between ‘slow’ and ‘fast’ misses (i.e. didn’t account for exposed memory latency)
Experience (2)

- MSPc/SIREV papers
  - Stencils (heat equation on a regular grid)
  - Used newer architectures (Opteron, Power5, Itanium2)
  - Attempted to model slow and fast misses (e.g. engaged prefetchers)
  - Modeling generally bounds performance and notes the trends

  - Attempted use performance counters to understand the quirks
  - Opteron and Power5 performance counters didn’t count prefetched data
  - Itanium performance counter trends correlated well with performance
BeBOP’s Current Tuning Efforts (last 2 years)
BeBOP’s Current Tuning Efforts

- Multicore (and distributed) oriented

- Throughput Oriented Kernels on Multicore architectures:
  - Dense Linear Algebra (LU, QR, Cholesky, …)
  - Sparse Linear Algebra (SpMV, Iterative Solvers, …)
  - Structured Grids (LBMHD, stencils, …)
  - FFTs
  - SW/HW co-tuning
  - Collectives (e.g. block transfers)

- Latency Oriented Kernels:
  - Collectives (Barriers, scalar transfers)
(re)design for evolution

- Design auto-tuners for an arbitrary number of threads
- Design auto-tuners to address the limitations of the multicore paradigm
- This will provide **performance portability** across both the existing breadth of multicore architectures as well as their evolution
Roofline Model:
Facilitating Program Analysis and Optimization
Motivating Example:
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)

```c
{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;
}}
```
SpMV Performance
(simple parallelization)

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

Graphs showing SpMV performance on different processors:
- Xeon E5345 (Clovertown)
- Opteron 2356 (Barcelona)
- UltraSparc T2+ T5140 (Victoria Falls)
- QS20 Cell Blade (PPEs)
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): 1.9x using 8 threads
- Opteron 2356 (Barcelona): 2.5x using 8 threads
- UltraSparc T2+ T5140 (Victoria Falls): 43x using 128 threads
- QS20 Cell Blade (PPEs): 3.4x using 4 threads
SpMV Performance
(simple parallelization)

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

Parallelism resulted in better performance, but did it result in good performance?
Auto-tuned SpMV Performance (portable C)

- Fully auto-tuned SpMV performance across the suite of matrices
- Why do some optimizations work better on some architectures?

**Graphs and Charts:**

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

**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?
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 even better performance,
but did it result in good performance?
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?

Should we spend another month optimizing it?
How should the bulk of programmers analyze performance?
<table>
<thead>
<tr>
<th>File</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>file 0</td>
<td>2207.66</td>
<td>2204.21</td>
<td>2204.12</td>
<td>77.44</td>
<td>355.7</td>
<td>59.54</td>
<td>0.64</td>
<td>0.02</td>
<td>2275.55</td>
</tr>
<tr>
<td>file 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>file 8</td>
<td>388.28</td>
<td>68.46</td>
<td>99.94</td>
<td>0.95</td>
<td>0.22</td>
<td>0.14</td>
<td>0.28</td>
<td>0.06</td>
<td>194.44</td>
</tr>
<tr>
<td>file 13</td>
<td>541.99</td>
<td>326.94</td>
<td>26.92</td>
<td>76.74</td>
<td>17.68</td>
<td>5.74</td>
<td>0.09</td>
<td>1.55</td>
<td>269.22</td>
</tr>
</tbody>
</table>

**C1. IPC AND PROCESSOR UTILIZATION**
VTUNE?
Roofline Model

- It would be great if we could always get peak performance

![Graph showing the Roofline Model with a horizontal line at 64 Peak flops]
Roofline Model (2)

- Machines have finite memory bandwidth
- Apply a Bound and Bottleneck Analysis
- Still Unrealistically optimistic model

\[ \text{Gflop/s(Al)} = \min \left\{ \frac{\text{Peak Gflop/s}}{\text{StreamBW} \times \text{AI}} \right\} \]
Naïve Roofline Model
(applied to four architectures)

- Bound and Bottleneck Analysis
- Unrealistically optimistic model
- Hand optimized Stream BW benchmark

Gflop/s(Al) = min \left( \text{Peak Gflop/s StreamBW} \right) \times Al
Roofline model for SpMV

- Delineate performance by architectural paradigm = ‘ceilings’
- In-core optimizations 1..i
- DRAM optimizations 1..j

\[
G\text{Flops}_{i,j}(A) = \min \left\{ \text{InCoreGFlops}_i, \text{StreamBW}_j * A \right\}
\]

- 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

Intel Xeon E5345
(Clovertown)

Opteron 2356
(Barcelona)

Sun T2+ T5140
(Victoria Falls)

IBM QS20
Cell Blade

- Dataset dataset fits in snoop filter
- No SIMD
- No ILP
- No DLP
- No naive SPE implementation
Roofline model for SpMV
(NUMA & SW prefetch)

- compulsory flop:byte ~ 0.166
- utilize all memory channels

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

- No naïve SPE 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/o SIMD
- w/o ILP
- w/o FMA
- mul/add imbalance
- dataset fits in snoop filter

Opteron 2356 (Barcelona)
- peak DP
- w/o SIMD
- w/o ILP
- mul/add imbalance
- dataset fits in snoop filter

Sun T2+ T5140 (Victoria Falls)
- peak DP
- 25% FP
- w/o SW prefetch
- w/o NUMA
- mul/add imbalance
- dataset fits in snoop filter

IBM QS20 Cell Blade
- peak DP
- w/o SIMD
- w/o ILP
- w/o FMA
- mul/add imbalance
- dataset fits in snoop filter

flop:DRAM byte ratio vs. attainable Gflop/s
Roofline model for SpMV
(matrix compression)

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

Performance is bandwidth limited
A Vision for BeBOP’s Future Performance Counter Usage
Deficiencies of the Roofline

- The Roofline and its ceilings are architecture specific
- They are not execution (runtime) specific

- It requires the user to calculate the true arithmetic intensity including cache conflict and capacity misses.

- Although the roofline is extremely visually intuitive, it only says what must be done by some agent (by compilers, by hand, by libraries)
- It does not state in what aspect was the code deficient
In the worst case, without performance counter data performance analysis can be extremely non-intuitive

(delete the ceilings)
- Transition from an architecture specific roofline, to an execution-specific roofline
Performance counters tell us the true memory traffic
Algorithmic Analysis tells us the useful flops
Combined we can calculate the true arithmetic intensity
Given the total memory traffic and total kernel time, we may also calculate the true memory bandwidth.

Must include 3C’s + speculative loads.
- Every idle bus cycle diminishes memory bandwidth
- Use performance counters to bin memory stall cycles

![Architecture-Specific Roofline](image1.png)
![Execution-Specific Roofline](image2.png)
- Measure imbalance between FP add/mul issue rates as well as stalls from lack of ILP and ratio of scalar to SIMD instructions
- Must be modified by the compulsory work
  - e.g. placing a 0 in a SIMD register to execute the _PD form increases the SIMD rate but not the useful execution rate
Relevance to Typical Programmer

- Visually Intuitive
- With performance counter data it’s clear which optimizations should be attempted and what the potential benefit is.
  (must still be familiar with possible optimizations)
Exhaustive search is intractable (search space explosion)

Propose using performance counters to guide tuning:
- Generate an execution-specific roofline to determine which optimization(s) should be attempted next
- From the roofline, it’s clear what doesn’t limit performance
- Select the optimization that provides the largest potential gain
  e.g. bandwidth, arithmetic intensity, in-core performance
- and iterate
Summary
Concluding Remarks

- Existing performance counter tools miss the bulk of programmers

- The Roofline provides a nice (albeit imperfect) approach to performance/architectural visualization

- We believe that performance counters can be used to generate execution-specific rooflines that will facilitate optimizations

- However, real applications will run concurrently with other applications sharing resources. This will complicate performance analysis

- next speaker…
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/QS22 Cell blades
Questions ?
BACKUP SLIDES
What’s a Memory Intensive Kernel?
Arithmetic Intensity in HPC

- True Arithmetic Intensity (AI) ~ Total Flops / Total DRAM Bytes
- Arithmetic intensity is:
  - ultimately limited by compulsory traffic
  - diminished by conflict or capacity misses
Memory Intensive

- A kernel is memory intensive when:
  the kernel’s arithmetic intensity < the machine’s balance (flop:byte)

- If so, then we expect:
  Performance ~ Stream BW * Arithmetic Intensity

- Technology allows peak flops to improve faster than bandwidth.
  ⇒ more and more kernels will be considered memory intensive
The Roofline Model

flop:DRAM byte ratio

attainable Gflop/s

Log scale!

Log scale!
Deficiencies of Auto-tuning

- There has been an explosion in the optimization parameter space.
- Complicates the generation of kernels and their exploration

- Currently we either:
  - Exhaustively search the space (increasingly intractable)
  - Apply very high level heuristics to eliminate much of it

- Need a guided search that is cognizant of both architecture and performance counters.
Deficiencies in usage of Performance Counters

- Only counted the number of cache/TLB misses
- We didn’t count exposed memory stalls (e.g. prefetchers)
- We didn’t count NUMA asymmetry in memory traffic
- We didn’t count coherency traffic
- Tools can be buggy or not portable
- Even worse is just giving a spread sheet filled with numbers and cryptic event names

- In-core events are less interesting as more and more kernels become memory bound