Automatic Performance Tuning

Samuel Williams¹, Kaushik Datta¹,², Shoaib Kamil¹,²  
Vasily Volkov¹,², Cy Chan³, Archana Ganapathi², Leonid Oliker¹,²,  
John Shalf¹, Jonathan Carter¹, Katherine Yelick¹,²

¹Lawrence Berkeley National Laboratory  
²University of California, Berkeley  
³Massachusetts Institute of Technology

SWWilliams@lbl.gov  
kdatta@eecs.berkeley.edu  
skamil@eecs.berkeley.edu
Motivation

- There are many kernels/methods whose performance is key to good application performance.
- By optimizing performance we may achieve the same throughput for substantially reduced cost (time, hardware, energy)

- The fastest **code** may vary with architecture or input.
- The fastest **data structure** may vary with architecture or input.
- The fastest **algorithm** may vary with architecture or input.
- Architectures continue to rapidly evolve and diversify (hand optimizing for one machine solves yesterday’s challenge)

- We believe a solution that provides **performance portability** is well worth an up front productivity cost.
Auto-tuning is built on the premise that if one can test every possible implementation of a program, one can find the fastest.

Key steps:
- Enumeration of an optimization space
- Generation of test cases from that space
- Exploration of those test cases (results in database/history)

Additional Components:
- Interpolation of results to select best implementation for runtime problem
- Assessment of the quality of results.
- Packaging the auto-tuner with an interface the end programmer can productively exploit and/or modify
Development Cycle

- Production Quality Auto-tuners are built after substantial initial development work

**initial domain research**
- domain/kernel-specific
- invention and enumeration of optimizations
- initial benchmarking

**prototype auto-tuner**
- proof of concept
- designed for experts
- restricted functionality
- inefficient search/history
- limited intelligence

**production auto-tuner construction**
- designed for end users
- improved search/history/interpolation capabilities
- flexible implementation allows for application to broad range of data/kernels

- Although this talk focuses on the initial research and construction of prototype auto-tuners for one particular domain, we’ve conducted research into other domains (noted later)
This talk is composed of four mini talks (by three speakers) discussing results to date as well as future research directions.

For conciseness, we’ll focus on:
- one particular domain: finite difference methods for PDEs.
- only cache-based computers.

- Auto-tuning the Laplacian
  - Sam Williams

- Roofline Model
  - Sam Williams

- Application of Machine Learning
  - Kaushik Datta

- Portability to other kernels
  - Shoaib Kamil
Auto-tuning the Laplacian Operator

Quality of Results

Resource-, Time-, and Knowledge-efficient Tuning

Portability to other kernels

Auto-tuning the Laplacian

Roofline Model

Application of Machine Learning

Productive Auto-tuner Construction

Sam Williams

Sam Williams

Kaushik Datta

Shoaib Kamil
The Laplacian Operator
(Finite Difference Method)
The Heat Equation Stencil

- Consider Poisson’s Equation (common PDE)
  - $\nabla^2 u(x,y,z) = f(x,y,z)$ $\nabla^2$ is the Laplacian differential operator
  - Solving this equation via the finite difference method (GS, GSRB, Jacobi) devolves into performing iterative stencil sweeps over a 3D regular volume.
  - Stencils are a linear combination of a point’s nearest neighbors.
  - Many possible derivations with different finite difference representations.

- We will apply auto-tuning to improve the performance of the heat equation: $\nabla^2 u(x,y,z,t) = \frac{\partial}{\partial t} u(x,y,z,t)$

- We will restrict ourselves to
  - optimizing the per sweep performance on a $256^3$ problem
  - single node performance (MPI extension is trivial)
  - Jacobi’s (iterative) method (even/odd grids for even/odd time steps)
  - Employ a computational collective model (all threads collective process one problem)
Simplest derivation of the Laplacian operator results in a 7-point stencil:

\[ u(x, y, z, t+1) = \alpha u(x, y, z, t) + \beta (u(x, y, z-1, t) + u(x, y-1, z, t) + u(x-1, y, z, t) + u(x+1, y, z, t) + u(x, y+1, z, t) + u(x, y, z+1, t)) \]

Clearly each stencil performs:
- 8 floating-point operations
- 8 memory references

*all but 2 should be filtered by an ideal cache*
Multicore SMPs of Interest

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

- Intel Xeon X5550 (Nehalem)
- Intel Xeon X5355 (Clovertown)
- Blue Gene/P
- Sun T2+ T5140 (Victoria Falls)
Multicore SMPs Used
(peak performance)

- Intel Xeon X5550 (Nehalem)
  - 85.33 Gflop/s
  - SIMD + mul/add

- Intel Xeon X5355 (Clovertown)
  - 85.33 Gflop/s
  - SIMD + mul/add

- Blue Gene/P
  - 13.6 Gflop/s
  - FMA + SIMD

- Sun T2+ T5140 (Victoria Falls)
  - 18.66 Gflop/s
Multicore SMPs Used
(pin bandwidth)

Intel Xeon X5550 (Nehalem)

- 51.2 GB/s (NUMA)

Intel Xeon X5355 (Clovertown)

- 21.33 GB/s (aggregate FSB BW)

Blue Gene/P

- 13.6 GB/s

Sun T2+ T5140 (Victoria Falls)

- 42.66 GB/s (read)

- 21.33 GB/s (write) (NUMA)
Multicore SMPs Used
(in-order cores = sw/compiler must pick up the slack)

Intel Xeon X5550 (Nehalem)

- 16 cores
- 32 GB of memory
- 4x4MB L2 cache
- 8x667MHz DDR3 DIMMs

Intel Xeon X5355 (Clovertown)

- 4 cores
- 8 GB of memory
- 4x256KB L1 cache
- 8x1066MHz DDR3 DIMMs

Blue Gene/P

- PowerPC 450d
- 4GB of memory
- 2x128b controllers
- 13.6 GB/s

Sun T2+ T5140 (Victoria Falls)

- SPARC T2+ processors
- 179 GB/s
- 4MB Shared L2 (16 way)
- 4x128b controllers
- 21.33 GB/s
Multicore SMPs Used
(thread-level parallelism)

Intel Xeon X5550 (Nehalem)

- 16 threads
- 8 threads (SMP mode)

Intel Xeon E5345 (Clovertown)

- 8 threads

Blue Gene/P

- 4 threads
- 128 threads!!

Sun T2+ T5140 (Victoria Falls)
Auto-tuning the Heat Equation
Stencil Performance
(reference code)

- NOTE (for 7-point):
  1 GStencil/s = 8 Gflop/s

- No scalability
- Poor performance
- VF performance peaks using ½ of one socket

<table>
<thead>
<tr>
<th>Hardware</th>
<th>GStencils/s</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon X5550</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Nehalem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xeon X5355</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Clovertown)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BlueGene/P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UltraSparc T2+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5140 (Victoria Falls)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reference Implementation
Greedy Search

- Resulting Optimization space will be so large that exhaustive search (even knowing the problem size) is intractable.

- Greedy search orders the optimizations (with expert architecture knowledge) and then searches them sequentially.

- Essentially transforms complexity from an $O(N^D)$ into a $O(N \times D)$ problem

- Fast, but not guaranteed to be optimal
- NUMA SMPs are common.
- It is essential the auto-tuner control data allocation to ensure maximum bandwidth.
- One `malloc()` with parallel initialization (exploit first touch).

- Caches have limited associativity.
- Also requires auto-tuner to control data structure allocation.
- Avoids cache conflict misses.
- Explore paddings up to 32 in unit stride.
Naïve parallelization gives each thread Z/P
Cache blocking gives each thread a number of consecutive (CX \times CY \times CZ) blocks.
- avoids LLC capacity misses
- Explore all possible power of 2 blockings

Register Blocking attempts to create locality in the register file/L1 cache
- Many possible blockings (RX \times RY \times RZ)
- avoids RF/L1 capacity misses
- Explore all possible power of 2 blockings
Prefetching initiates now the loads of points that will eventually be needed.

- Explore a number of prefetch distances
- *hides memory latency*

Thread blocking adds a second level of parallelization for SMT architectures.

- **inter-thread locality via shared L1’s**

Explicit SIMDization attempts to compensate for a compiler’s inability to generate good SIMD code.

- Cache bypass exploits instructions designed to eliminate write allocate traffic
Auto-tuned Stencil Performance
(full tuning, greedy search)

- Dramatically better performance and scalability
- Clearly cache blocking (capacity misses) were critical to performance despite these machines having 8-16MB of cache
- XLC utterly failed to register block (unroll&jam)
- Array padding was critical on VF (conflict misses)

<table>
<thead>
<tr>
<th>Xeon X5550 (Nehalem)</th>
<th>Xeon X5355 (Clovertown)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>GStencils/s</td>
</tr>
<tr>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>0.4</td>
</tr>
<tr>
<td>4</td>
<td>0.8</td>
</tr>
<tr>
<td>8</td>
<td>2.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BlueGene/P</th>
<th>GStencils/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.05</td>
</tr>
<tr>
<td>2</td>
<td>0.15</td>
</tr>
<tr>
<td>4</td>
<td>0.35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UltraSparc T2+ T5140 (Victoria Falls)</th>
<th>GStencils/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.05</td>
</tr>
<tr>
<td>2</td>
<td>0.2</td>
</tr>
<tr>
<td>4</td>
<td>0.7</td>
</tr>
<tr>
<td>8</td>
<td>1.0</td>
</tr>
<tr>
<td>16</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- +2nd pass thru greedy search
- +Cache bypass
- +Explicit SIMDization
- +Thread Blocking
- +SW Prefetching
- +Register Blocking
- +Cache Blocking
- +Padding
- +NUMA
- Reference Implementation
Explore alternate finite difference method derivations.

Subtly performance (stencils/s) can go down (more flops per stencil) but both performance (gflop/s) and time to solution can improve (fewer sweeps to converge)

Two approaches:
- 27-point stencil (~30 flops per stencil)
- 27-point stencil with inter-stencil common subexpression elimination (~20+ flops/stencil)
Auto-tuned 7-point Stencil
(full tuning, greedy search)

- When embedded in an iterative solver, 27-point should require fewer iterations to converge.
- However, performance (stencils/second) is lower, so one must tune for the right balance of performance per iteration and number of iterations.
- Observe Clovertown performance didn’t change (bandwidth is so poor it’s the bottleneck in either case).

![Graphs showing performance comparison between Xeon X5550 (Nehalem), Xeon X5355 (Clovertown), BlueGene/P, and UltraSparc T2+ T5140 (Victoria Falls).]

- +Common Subexpression Elimination
- +Cache bypass
Auto-tuned 27-point Stencil
(full tuning, greedy search)

- When embedded in an iterative solver, 27-point should require fewer iterations to converge.
- However, performance (stencils/second) is lower, so one must tune for the right balance of performance per iteration and number of iterations.
- Observe Clovertown performance didn’t change (bandwidth is so poor it’s the bottleneck in either case).

![Graphs showing performance on Xeon X5550 (Nehalem), Xeon X5355 (Clovertown), BlueGene/P, and UltraSparc T2+ T5140 (Victoria Falls).]
We’ve shown that auto-tuning can provide performance portability across a wide range of cache-based processor architectures

- **6x** on the most advanced architecture (Nehalem)
- **8x** on the most parallel architecture (Victoria Falls)

Clearly, optimizations that control data structure, modify code, and change algorithms (different stencils) are required to attain the best performance.

This work was submitted as:


Additional Auto-tuning Work
Cell and GPU’s

- All the stencil work was also implemented via an auto-tuner on a QS22 CBE, and a direct optimized implementation on a GTX280.
- Clearly, the new (DDR2) Cell is bandwidth-starved.
We’ve also implemented a sparse matrix-vector multiply auto-tuner on multicore and Cell.
LBMHD Auto-tuner

- and another for a key kernel extracted from a lattice-boltzmann magneto-hydrodynamics code (LBMHD)
- Essentially a variant on stencils/structured grids
Additionally, we extended auto-tuning to the distributed implementation of LBMHD.

Implemented an auto-tuned hybrid version where we tune for the optimal balance between threads and processes.

+600 Gflop/s performance boost at 128 nodes.

Note each of the last 3 bars may have unique MPI decompositions as well as VL/unroll/DLP.

Observe that for this large problem, auto-tuning flat MPI delivered significant boosts (2.5x).

However, expanding auto-tuning to include the domain decomposition and balance between threads and processes provided an extra 17%.

2 processes with 2 threads was best.
As Kamesh discussed yesterday, we’ve optimized a particle-in-cell (PIC), called GTC for multicore and are considering when/where/how auto-tuning is needed.

Similarly, we’re (led by Aparna Chandramowlishwaran) conducting the initial research into optimizing the Fast Multipole Method (FMM) for multicore.
Limitations of Auto-tuning
Limitations

- Although we showed auto-tuning dramatically improved performance there are some critical limitations to our approach:
  - There are currently no absolutes when it comes to performance (only relative statements of better)
  - The designer must possess PhD-level architectural- and domain-knowledge to both construct the code generator and search components.
  - Even with a greedy search algorithm the tuning time/resources/foreknowledge can be prohibitively high.
  - The auto-tuner is specific to one kernel. Any changes to that kernel (changing 7pt to 27pt) requires constructing an entirely new auto-tuner.

- Our latest research is focused on these problems and the results to date will be discussed parts II, III, and IV.
Quantitative Assessment of Results

- Auto-tuning the Laplacian
  - Sam Williams

- Roofline Model
  - Sam Williams

- Application of Machine Learning
  - Kaushik Datta

- Resource-, Time-, and Knowledge-efficient Tuning
  - Sam Williams

- Portability to other kernels
  - Shoaib Kamil
Motivation for Performance Assessment

- We want to say more than simply performance improved by 147%.
- We want to be able to say that no matter how much additional work is poured into the tuner, performance can only increase by 5%.

- With this information:
  - we can abort tuning early (when we reach a certain level of quality)
  - in conjunction with performance counters, dynamically select the optimization order of the greedy search.
  - pinpoint performance bottlenecks to motivate new optimizations
  - pinpoint performance bottlenecks so as to drive future procurements
  - provide feedback to architects and computational scientists as to how to design the next generation of architectures or algorithms.

- We believe that this information can be easily visualized with a model we call the Roofline Model.
Key Concepts...
Arithmetic Intensity in HPC

- **True Arithmetic Intensity (AI)** ~ Total Flops / Total DRAM Bytes
- Some HPC kernels have an arithmetic intensity that scales with problem size (increased temporal locality), but remains constant on others
- Arithmetic intensity is ultimately limited by compulsory traffic
- Arithmetic intensity is diminished by conflict or capacity misses.
Categorization of Software Optimizations
Optimization Categorization

<table>
<thead>
<tr>
<th>Maximizing <em>(attained)</em></th>
<th>Maximizing <em>(attained)</em></th>
<th>Minimizing <em>(total)</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>In-core Performance</td>
<td>Memory Bandwidth</td>
<td>Memory Traffic</td>
</tr>
<tr>
<td>Maximizing In-core Performance</td>
<td>Maximizing Memory Bandwidth</td>
<td>Minimizing Memory Traffic</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>• Exploit in-core parallelism (ILP, DLP, etc…)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Good (enough) floating-point balance</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Optimization Categorization

Maximizing In-core Performance

• Exploit in-core parallelism (ILP, DLP, etc…)

• Good (enough) floating-point balance

Maximizing Memory Bandwidth

Minimizing Memory Traffic

- unroll & jam
- reorder
- explicit SIMD
- eliminate branches
- eliminate branches
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
- SW prefetch
- DMA lists
- TLB blocking
- Unit-stride streams
- Memory affinity

Diagram:
- Unroll & jam
- Reorder
- Eliminate branches
- Explicit SIMD
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

- Unroll & jam
- Reorder
- Explicit SIMD
- Eliminate branches

- Unit-stride streams
- Memory affinity
- DMA lists
- SW prefetch

- Array padding
- Cache blocking
- Compress data
- Streaming stores
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

- unroll & jam
- reorder
- explicit SIMD
- eliminate branches
- unit-stride streams
- memory affinity
- DMA lists
- SW prefetch
- TLB blocking
- array padding
- cache blocking
- compress data
- streaming stores
Introduction to the Roofline Model
Roofline Model
Basic Concept

- Synthesize communication, computation, and locality into a single visually-intuitive performance figure using bound and bottleneck analysis.

\[ \text{Attainable Performance}_{ij} = \min \left\{ \text{FLOP/s with Optimizations}_1^1, \text{AI} \times \text{Bandwidth with Optimizations}_1^1 \right\} \]

- where optimization \( i \) can be SIMDize, or unroll, or SW prefetch, …
- Given a kernel’s arithmetic intensity (based on DRAM traffic after being filtered by the cache), programmers can inspect the figure, and bound performance.

- Moreover, provides insights as to which optimizations will potentially be beneficial.
Plot on log-log scale

- Given AI, we can easily bound performance
- But architectures are much more complicated
- We will bound performance as we eliminate specific forms of in-core parallelism
- Opterons have dedicated multipliers and adders.
- If the code is dominated by adds, then attainable performance is half of peak.
- We call these **Ceilings**
- They act like constraints on performance
Opterons have 128-bit datapaths.

If instructions aren’t SIMDized, attainable performance will be halved.
On Opterons, floating-point instructions have a 4 cycle latency.

If we don’t express 4-way ILP, performance will drop by as much as 4x.
We can perform a similar exercise taking away parallelism from the memory subsystem.
Explicit software prefetch instructions are required to achieve peak bandwidth.
Opterons are NUMA.

As such memory traffic must be correctly balanced among the two sockets to achieve good Stream bandwidth.

We could continue this by examining strided or random memory access patterns.
We may bound performance based on the combination of expressed in-core parallelism and attained bandwidth.
Roofline Model
locality walls

- Remember, memory traffic includes more than just compulsory misses.
- As such, actual arithmetic intensity may be substantially lower.
- Walls are unique to the architecture-kernel combination

\[ AI = \frac{\text{FLOPs}}{\text{Compulsory Misses}} \]
Remember, memory traffic includes more than just compulsory misses.

As such, actual arithmetic intensity may be substantially lower.

Walls are unique to the architecture-kernel combination.

\[
AI = \frac{\text{FLOPs}}{\text{Allocations} + \text{Compulsory Misses}}
\]
Remember, memory traffic includes more than just compulsory misses.

As such, actual arithmetic intensity may be substantially lower.

Walls are unique to the architecture-kernel combination.
Remember, memory traffic includes more than just compulsory misses.

As such, actual arithmetic intensity may be substantially lower.

Walls are unique to the architecture-kernel combination.
Ceiling-Optimization Interplay

- Ceilings act to constrain performance coordinates
  - ceilings limit performance
  - walls limit AI
- Optimizations target specific ceilings and remove them as (potential) constraints to performance.
Optimizations remove these walls and ceilings which act to constrain performance.
Optimizations remove these walls and ceilings which act to constrain performance.
Optimizations remove these walls and ceilings which act to constrain performance.
Optimizations remove these walls and ceilings which act to constrain performance.
Application of the Roofline Model to the 7-point Stencil
Roofline Model

- DRAM-FP roofline models for the architectures of interest.
- NOTE: as VF is an inorder dual-issue CMT architecture, its ceilings are floating-point mix.
Roofline Model

- Overlay of range in arithmetic intensity and corresponding performance bound (in Gflop/s)
- Clearly, performance will be heavily memory bound on some architectures, but will require varying degrees of in-core optimizations.
- Arithmetic intensity is the ideal compulsory limit for either case.
- Capacity misses, conflict misses, padding, prefetching will further decrease it.

- NOTE:
  - red = write allocate
  - green = cache bypass
Roofline Model

- Reference performance.
- Note, the x-coordinate is not well defined without accurate performance counters and is shown based on an ideal write-allocate cache.
Auto-tuned performance is shown in green.

- x-coordinate should be much more accurate but still an upper bound.

- Cleary, performance is close to the roofline limit.
Roofline clearly enumerates the performance bound.

We observe that after auto-tuning, performance is very close to the bound.

The roofline model could be extended to the 27-point stencil, but one should implement a hierarchical model as cache bandwidth can also be a bottleneck.

Unfortunately the biggest limitation of the model is its reliance on accurately knowing/measuring arithmetic intensity.

Without accurate performance counters, this is extremely difficult.
III

Accelerating Tuning via Machine Learning

Auto-tuning the Laplacian

Sam Williams

Roofline Model

Sam Williams

Application of Machine Learning

Kaushik Datta

Quality of Results

Resource-, Time-, and Knowledge-efficient Tuning

Portability to other kernels

Productive Auto-tuner Construction

Shoaib Kamil
Motivation

- For the 7-point stencil:

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Parameters</th>
<th>Total Configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Count</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Domain Decomposition</td>
<td>4</td>
<td>36</td>
</tr>
<tr>
<td>Software Prefetching</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>Padding</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>Inner Loop</td>
<td>8</td>
<td>480</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>16</strong></td>
<td><strong>4x10^7</strong></td>
</tr>
</tbody>
</table>

- There are more than 40 million different configurations for this simple stencil auto-tuner.
- This problem continues to get worse when:
  - More cores are added to a die
  - More kernels are combined to form an application
- Our previous method for traversing the search space required significant expert knowledge.
- **How can non-experts search this space efficiently and effectively?**
Solution: Machine Learning

- We propose that machine learning:
  - Efficiently traverses auto-tuning’s vast parameter space
  - Produces faster search to high-quality solution
  - Requires little domain knowledge
  - Allows us to handle problems with even larger search spaces

Image courtesy of: www.onlinelearningbooks.com
Platforms

AMD Barcelona

- Cache-based, x86 Architectures
- 2 sockets x 4 cores/socket x 1 HW thread/core
- gcc on Barcelona, icc on Clovertown
- PAPI for performance counter data

Intel Clovertown

- Cache-based, x86 Architectures
- 2 sockets x 4 cores/socket x 1 HW thread/core
- gcc on Barcelona, icc on Clovertown
- PAPI for performance counter data
Experimental Methodology

**Goal**: Find best value for configuration parameters to optimize performance

1. Sample 1500 data points from configuration space
2. Run stencil with chosen configurations
3. Identify relationship between optimization configurations and performance
4. Manipulate this relationship to find the best performing configuration
Finding Correlations

Raw Data Space

Configuration features
Performance Features

Raw Data Space

Project onto dimensions of maximum correlation

Projected Data Space

$K_X^*A$

Same Neighbors

$K_Y^*B$
3 Challenges in using ML

1. How to represent configuration parameters and performance metrics as feature vectors
2. How to compare similarity of two feature vectors
3. How to leverage existing ML techniques to find optimal configuration
## Configuration Parameters

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Parameter Type</th>
<th>Parameter Name</th>
<th>Parameter Range</th>
<th>Number of Configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Count</td>
<td>Number of Threads</td>
<td>( Nthreads )</td>
<td>( {2^0 \ldots 2^3} )</td>
<td>4</td>
</tr>
<tr>
<td>Domain Decomposition</td>
<td>Block Size</td>
<td>( CX )</td>
<td>( {2^7 \ldots NX} )</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( CY )</td>
<td>( {2^1 \ldots NY} )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( CZ )</td>
<td>( NZ )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Chunk Size</td>
<td>( CX \times CY \times CZ \times NThreads )</td>
<td>( {\ldots} )</td>
<td></td>
</tr>
<tr>
<td>Software Prefetching</td>
<td>Prefetching Type</td>
<td></td>
<td>{register block, plane, pencil}</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Prefetching Distance</td>
<td></td>
<td>( {0, 2^0 \ldots 2^9} )</td>
<td>6</td>
</tr>
<tr>
<td>Padding</td>
<td>Padding Size</td>
<td></td>
<td>( {0 \ldots 31} )</td>
<td>32</td>
</tr>
<tr>
<td>Inner Loop Optimizations</td>
<td>Register Block Size</td>
<td>( RX )</td>
<td>( {2^0 \ldots 2^1} )</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( RY )</td>
<td>( {2^0 \ldots 2^1} )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( RZ )</td>
<td>( {2^0 \ldots 2^3} )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Statement Type</td>
<td></td>
<td>{complete, individual}</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Read From Type</td>
<td></td>
<td>{array, variable}</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Pointer Type</td>
<td></td>
<td>{fixed, moving}</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Neighbor Index Type</td>
<td></td>
<td>{register block, plane, pencil}</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>FMA-like Instructions</td>
<td></td>
<td>{yes, no}</td>
<td>2</td>
</tr>
</tbody>
</table>

### Feature Vector

<table>
<thead>
<tr>
<th>Threads</th>
<th>Block Size ( CX )</th>
<th>Block Size ( CY )</th>
<th>Block Size ( CZ )</th>
<th>Padding Size</th>
<th>Prefetch type</th>
<th>Prefetch distance</th>
<th>Statement Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>32</td>
<td>128</td>
<td>256</td>
<td>32</td>
<td>Plane</td>
<td>64</td>
<td>individual</td>
</tr>
</tbody>
</table>
## Performance Metrics

### Feature Vector \( Y \)

<table>
<thead>
<tr>
<th>Counter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_TOT_CYC</td>
<td>Cycles per thread per job</td>
</tr>
<tr>
<td>PAPI_L1_DCM</td>
<td>L1 data cache misses per thread</td>
</tr>
<tr>
<td>PAPI_L2_DCM</td>
<td>L2 data cache misses per thread</td>
</tr>
<tr>
<td>PAPI_TLB_DCM</td>
<td>TLB misses per thread</td>
</tr>
<tr>
<td>PAPI_CA_SHR</td>
<td>Accesses to shared cache lines</td>
</tr>
<tr>
<td>PAPI_CA_CLN</td>
<td>Accesses to clean cache lines</td>
</tr>
<tr>
<td>PAPI_CA_ITV</td>
<td>Cache interventions</td>
</tr>
</tbody>
</table>

Power meter: Watts consumed

### Total Cycles

<table>
<thead>
<tr>
<th>Counter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Cycles</td>
<td>1.9E7</td>
</tr>
<tr>
<td>L1_DCM</td>
<td>2.4E5</td>
</tr>
<tr>
<td>L2_DCM</td>
<td>1.5E5</td>
</tr>
<tr>
<td>TLB_DCM</td>
<td>1.2E4</td>
</tr>
<tr>
<td>CA_SHR</td>
<td>1.2E5</td>
</tr>
<tr>
<td>CA_CLN</td>
<td>1.4E4</td>
</tr>
<tr>
<td>CA_ITV</td>
<td>1.2E3</td>
</tr>
<tr>
<td>Energy Efficiency</td>
<td>2.3E4</td>
</tr>
</tbody>
</table>

\[
\text{Feature Vector } Y = \frac{\text{Total cycles} \times \# \text{ of flops}}{\text{clk rate} \times \# \text{ of watts}}
\]
3 Challenges in using ML

1. How to represent configuration parameters and performance metrics as feature vectors

2. How to compare similarity of two feature vectors

3. How to leverage existing ML techniques to find optimal configuration
Kernel functions

Are X1 and X2 more similar than X2 and X3?

- Euclidian distance does not suffice
  - Non-numeric data

- Custom measure of similarity
  - Numeric Columns: Gaussian Kernel
    \[ K(x_i, x_j) = \exp(-||x_i-x_j||^2 / \tau) \]
  - Non-numeric Columns:
    \[ K(x_i, x_j) = \begin{cases} 1 & \text{if } x_i = x_j, \\ 0 & \text{if } x_i \neq x_j \end{cases} \]

- Similarity(X1, X2) = average(K(X1i, X2i))

<table>
<thead>
<tr>
<th>Threads</th>
<th>Block Size CX</th>
<th>Block Size CY</th>
<th>Block Size CZ</th>
<th>Padding Size</th>
<th>Prefetch type</th>
<th>Prefetch distance</th>
<th>Statement Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>32</td>
<td>128</td>
<td>256</td>
<td>32</td>
<td>Plane</td>
<td>64</td>
<td>Individual X1</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>32</td>
<td>Plane</td>
<td>64</td>
<td>Individual X2</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>32</td>
<td>Pencil</td>
<td>64</td>
<td>complete X3</td>
</tr>
</tbody>
</table>
3 Challenges in using ML

1. How to represent configuration parameters and performance metrics as feature vectors
2. How to compare similarity of two feature vectors
3. How to leverage existing ML techniques to find optimal configuration
KCCA

\[ \begin{bmatrix} 0 & K_X K_Y \\ K_Y K_X & 0 \end{bmatrix} [A] = \lambda \begin{bmatrix} K_X K_X & 0 \\ 0 & K_Y K_Y \end{bmatrix} [A] \]

KX*A

KY*B

Stencil Code

Optimization Configurations

Platform

Performance
Finding Optimal Configurations

Configuration features

Performance Metrics

Raw Data Space

KCCA

KCCA Data Space

K_X^A

K_Y^B

Nearest Neighbors
Finding Optimal Configurations

Configuration features

Raw Data Space

Genetic Algorithm (Permute Optimizations)

Optimization Configurations

Stencil Code

Platform

Performance
Evaluation Benchmarks

- "No Optimization": running naïve code
- "Expert Optimized": ordered the optimizations, applied them consecutively, this was explained previously

- "Random Raw Data": best performing point in raw data
- "Genetic on Raw Data": permute configs for top three best performing points in raw data
Barcelona Performance

- No Optimization
- Expert Optimized
- Random Raw Data
- Genetic on Raw Data
- SML Optimized

GFlops/second

7 point stencil
27 point stencil
Intel Clovertown

Clovertown Performance

- No Optimization
- Expert Optimized
- Random Raw Data
- Genetic on Raw Data
- SML Optimized

GFlops/second

7 point stencil

27 point stencil
Auto-tuning Time

- **Exhaustive Search:**
  - $4 \times 10^7$ configs x 0.08 s/trial x 5 trials > 180 days

- **Expert Optimized:**
  - Ordering optimizations: *Cannot be quantified*
  - Applying optimizations consecutively: $570$ configs x 0.08 s/trial x 5 trials = 3.8 min

- **Our Technique:**
  - Training data: 1500 configs x 0.08 s/trial x 5 trials = 10 minutes
  - Training time using KCCA: 90 minutes
  - Genetic algorithm: 243 configs x 0.08 s/trial x 5 trials = 1.6 min
  - Total Time < 2 hrs

- Our performance results are up to 18% faster than expert-optimized!
- Reduce the model training time (currently performed on a laptop)
- Try other architectures
- Try other motifs (e.g. sparse matrices)
- Expand the search space:
  - Tuning optimization parameters AND compiler flags
  - Tuning for multiple metrics of merit
  - Tuning for composition of multiple kernels
- ML:
  - KCCA + kernel regression + gradient descent to find optimal configurations
IV

Productive Construction of Domain-Specific Auto-tuners

- Auto-tuning the Laplacian: Sam Williams
- Roofline Model: Sam Williams
- Application of Machine Learning: Kaushik Datta
- Portability to other kernels: Shoaib Kamil

Resource-, Time-, and Knowledge-efficient Tuning

Quality of Results

Productive Auto-tuner Construction
Problem

- So far have built *kernel-specific* auto-tuners
- For each kernel, requires PhD-level architectural- and domain-knowledge
- For each kernel, requires *months or years* of development effort

- Can we productively build auto-tuners for a *class of kernels*: provide performance portability across architectures *and* kernels
Why not a BLAS for Stencils?

- Unlike dense linear algebra, many different variants of stencils
  - variants in data structure
  - variants in grid topology
  - etc

- Better solution: represent stencils in an abstract representation
  - based on the application’s implementation
  - basis for code generation and transformation
  - express optimizations as code transformations
Built a proof-of-concept auto-tuner for stencil kernels

Given a Fortran application with stencil kernels:

1. Programmer annotates stencil loops
2. Auto-tuning system automatically
   - converts stencils into internal representation
   - generate candidate versions of stencil & test harness
   - discover best implementation of each stencil
   - produce library of best implementations
3. Programmer updates application to call optimized library
**Benchmark Stencils**

**Laplacian**

```
read_array[ ]
```

```
x dimension
```

```
y x product
```

```
write_array[ ]
```

```

**Divergence**

```
read_array[ ][ ]
```

```
x dimension
```

```
y x product
```

```
write_array[ ]
```

```
u
```

```
x y z
```

```

```
Studied Kernels

**Gradient**

```
read_array[]
```

```
x dimension
```

```
write_array[][ ]
```

```
xy product
```

---

**Bilateral Filter**

```
read_array[]
```

```
x dimension
```

```
filter_array[]
```

```
lookup
```

```
write_array[]
```

```
u'
```

---
MRI images need edge-preserving smoothing to remove artifacts from instruments.

Normal Gaussian filters smooth images, but destroy sharp edges.

This kernel performs anistropic filtering thus preserving edges.

For each neighboring voxel, must lookup filter value based on photometric difference between center point and neighbor.
Studied Kernels
Studied Kernels
**Results Key**

- **serial reference**
  - Original code in Fortran

- **Auto-parallelization**
  - Auto-parallelized using the stencil framework (no tuning)

- **Auto-NUMA**
  - Auto-parallelized plus NUMA optimization

- **Auto-tuning**
  - Auto-tuned and auto-parallelized using the stencil framework

- **STREAM Predicted**
  - Memory-bound performance predicted using OpenMP STREAM benchmark

- **OpenMP Comparison**
  - Performance of a NUMA-aware auto-parallelized with OpenMP version of the original code
Auto-parallelization by itself does not scale well on CPUs
- requires NUMA-aware alloc to get decent performance
- our auto-parallelizer gets equal or better performance than OpenMP
- Overall speedups of up to 22x on Nehalem (vs. serial reference)
Divergence Results

- Less benefit from auto-tuning on cache-based architectures here
  - As we expect based on arithmetic intensity
- Overall speedups of up to 13x on Victoria Falls
Gradient Results

- Heavily memory-bound, so architectures with high memory BW get higher performance
- Overall speedups of up to 8.1x on Nehalem
Bilateral Filter Results (r=3)

- Heavily compute-bound, plus lookup for filter weights
  - Most of auto-tuning benefit comes from better innermost-loop
- Overall speedups of 14.8x for Barcelona, 20.7x for Nehalem
- Near linear speedup as cores increase
Productivity Results

- Building this auto-tuning system
  - similar in terms of time & size (LoC) to single-kernel tuners
  - but embeds knowledge gained from single-kernel tuners into an auto-tuning system

- Running the auto-tuning system
  - few seconds to generate candidate versions
  - few hours to find best implementation
    - with better search (as covered in Kaushik’s talk) this could be reduced dramatically
  - simple enough for application developers to use: no specialized understanding of the architecture required
Summary

- Proof-of-concept shows we can build an auto-tuner for a class of stencil kernels
- Performance is excellent: equal to single-kernel auto-tuning with the same optimizations
- Research goal: now that proof-of-concept shows it is possible
  - build a usable auto-tuning system for stencils
  - add in optimizations from single-kernel auto-tuners
  - make sure components can be re-used for other auto-tuning systems or compilers
- Huge step forward for auto-tuning: users no longer need highly-specialized knowledge to auto-tune their stencil kernels
Auto-tuning Summary
Summary

- Auto-tuning has been shown to benefit a large number of scientific and high performance codes across a large number of domains.
- On-going research is allowing us to expand into other domains and apply auto-tuning to novel architectures (e.g. Cell).

- Recent research progress has allowed us to:
  - quantify our auto-tuning success
  - accelerate/simplify the tuning search process
  - provide a means of productive auto-tuner construction for the structured grid domain

- Requested Feedback: future research directions?
  - outside scientific computing what are the kernels/problems of interest?
Acknowledgements

- Research supported by:
  - DOE Office of Science under contract number DE-AC02-05CH11231
  - Microsoft and Intel funding (Award #20080469)
  - NSF contract CNS-0325873
  - Sun Microsystems - Niagara2 / Victoria Falls donations
  - Georgia Tech - access to QS22 Cell blades
  - Nvidia – GTX280 donations
Questions ?