Recent Results, Insights and Lessons from Autotuning
Three Motifs

UC Berkeley ParLab
Berkeley Benchmarking and Optimization Group (BeBOP)
Shoaib Kamil (skamil@cs.berkeley.edu)
Overview

- Other autotuning at Berkeley

- Recent work autotuning three parallel kernels
  - Sparse Matrix Vector Multiply (SpMV)
  - Lattice Boltzmann MHD
  - Stencils (Heat Equation)

- Integrating SpMV Advances into OSKI

- Towards a framework for building autotuners
  - What is the role of the compiler?

- Open questions
Other Autotuning Work

- Using Delta Debugging to automatically “fix” precision errors
  - attempt to use extra precision only when necessary
  - preliminary implementation by Kamil with Kaushik Sen

- Autotuning PGAS Collectives
  - Using run-time and install-time tuning to speed up collective operations such as barrier, reductions, etc.
  - Tuning space involves trees for collective operations
  - Current work by Rajesh Nishtala
Other Autotuning Work

- Using Delta Debugging to automatically “fix” precision errors
  - attempt to use extra precision only when necessary
  - preliminary implementation by Kamil with Kaushik Sen

- Autotuning PGAS Collectives

![Performance Advantages of Looser Synchronization](chart.png)
Recent autotuning results from these kernels

<table>
<thead>
<tr>
<th></th>
<th>Regular Entries</th>
<th>Irregular Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Regular Sparsity Pattern</strong></td>
<td><strong>Stencil, LBMHD, Climate</strong></td>
<td><strong>Image Segmentation</strong></td>
</tr>
<tr>
<td><strong>Irregular Sparsity Pattern</strong></td>
<td><strong>Laplacian of a Graph</strong></td>
<td><strong>SpMV</strong></td>
</tr>
</tbody>
</table>
SpMV Overview

- A sparse matrix has few non-zeros
  - Performance advantage in only storing/operating on non-zeros
  - Requires significant metadata
- Our sparse matrix-vector multiply ($Ax = y$) operation represents:
  - $A$ as a sparse matrix in Compressed Sparse Row (CSR) format
  - $x$ and $y$ as dense contiguous vectors
- SpMV has indirect and irregular memory access patterns

![Diagram](image)

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

```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;
}
```
Heat Equation Stencil Overview

- A *stencil code* updates every point in a regular grid with a constant weighted subset of its neighbors.
- Typically derive from finite-difference techniques for solving PDE’s.
- Stencils have direct and regular memory access patterns.
- We examine an out-of-place 3D 7-point stencil with constant coefficients.

\[
\text{Next}[x,y,z] = \begin{align*}
  c_0 \times & \text{Current}[x,y,z] + \\
  c_1 \times ( & \text{Current}[x+1,y,z] + \\
  & \text{Current}[x-1,y,z] + \\
  & \text{Current}[x,y+1,z] + \\
  & \text{Current}[x,y-1,z] + \\
  & \text{Current}[x,y,z+1] + \\
  & \text{Current}[x,y,z-1] ) \\
\end{align*}
\]
LBMHD Overview
(Lattice Boltzmann Magneto-Hydrodynamics)

- Plasma turbulence simulation (structured grid code with time steps)
- Two distributions:
  - Momentum distribution (27 scalar velocities)
  - Magnetic distribution (15 vector velocities)
- Three macroscopic quantities:
  - Density
  - Momentum (vector)
  - Magnetic field (vector)
- Distribution functions used to reconstruct macroscopic quantities
- For each spatial point in $128^3$ grid:
  - 73 doubles read and 79 doubles written (min 1200 bytes)
  - Approximately 1300 flops performed
Arithmetic Intensity
(Ratio of flops to DRAM bytes)

- AI is a rough indicator of whether kernel is memory or compute-bound.
- Counting *only* compulsory misses:

  - The range in AI values results from:
    - SpMV: the amount register blocking reduces redundant column indices
    - Stencil and LBMHD: whether the architecture is write-allocate
  - Actual AI values are typically lower (due to other types of cache misses)
Autotuning provides a **portable** and effective method for tuning

All three kernels used Perl scripts to generate code variants

Implemented by Kaushik Datta and Sam Williams

**SpMV: Heuristic search**
- Chose best parameter values via heuristics
- Example: Selected best matrix compression parameters by finding minimum matrix size

**LBMHD: Full exhaustive search**
- Relatively small search space allowed full exhaustive search using power-of-two values

**Stencil: “Greedy” exhaustive search**
- Large search space, so applied each optimization individually
- Performed exhaustive search using power-of-two values within each optimization’s parameter space
- Chose the best value before proceeding
## Summary of Optimizations

<table>
<thead>
<tr>
<th>No.</th>
<th>Optimization</th>
<th>OSKI 1.0.1h</th>
<th>Williams et al.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Register Blocking</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2.</td>
<td>Cache Blocking</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.</td>
<td>Software Prefetching</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>4.</td>
<td>Matrix Compression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Serial Benchmarking</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Data Decomposition within Shared Memory</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>7.</td>
<td>Parallel Benchmarking</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>Data Decomposition across Distributed Memory</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>9.</td>
<td>SIMDization</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>10.</td>
<td>Software Pipelining</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>11.</td>
<td>TLB Blocking</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Architectures
(Architectural Features)

Core
4MB
shared L2

Core
4MB
shared L2

Core
4MB
shared L2

Intel Clovertown

AMD Barcelona

Cache-based
Superscalar

Cache-based
Multithreaded

Local Store-based
SIMD (SPEs)

Sun Niagara2 (Victoria Falls)

IBM Cell Blade
Architectures
(Double Precision Peak Flops)

Intel Clovertown
- 75 GFlops/s
- 10.66 GB/s (write)
- 21.33 GB/s (read)
- 667MHz FBDIMMs

AMD Barcelona
- 74 GFlops/s
- 10.6 GB/s
- 667MHz DDR2 DIMMs

Sun Niagara2 (Victoria Falls)
- 18.7 GFlops/s
- 4MB shared L2 (16 way)
- 667MHz FBDIMMs

IBM Cell Blade
- 29 GFlops/s (SPEs only)
- 512MB XDR DRAM

75 GFlops/s
74 GFlops/s
18.7 GFlops/s
29 GFlops/s (SPEs only)
Architectures (Raw DRAM Bandwidth)

Intel Clovertown

- 21.33 GB/s (read)
- 10.66 GB/s (write)

AMD Barcelona

- 21.33 GB/s

Sun Niagara2 (Victoria Falls)

- 42.66 GB/s (read)
- 21.33 GB/s (write)

IBM Cell Blade

- 51.2 GB/s
- <20 GB/s (each direction)
Kernel Performance

- Naïve portable C code
- Little programmer effort
- Performance quality still unknown

SpMV number is average over suite of matrices
Kernel Performance

- Autotuned portable C code
- Significant programmer effort, but applicable across architectures
- Performance noticeably improved on all machines
Kernel Performance

- Autotuned platform-specific C code
- Significant programmer effort for each architecture
- Performance dramatically improved on Cell (now using SPEs)
Note that different optimizations have different impacts per arch!
Overview

- Other autotuning at Berkeley

- Recent work autotuning three parallel kernels
  - Sparse Matrix Vector Multiply (SpMV)
  - Lattice Boltzmann MHD
  - Stencils (Heat Equation)

- Integrating SpMV Advances into OSKI

- Towards a framework for building autotuners
  - What is the role of the compiler?

- Open questions
OSKI Overview

- Optimized Sparse Kernel Interface (OSKI) by Rich Vuduc et al
  - autotuned library for sparse linear algebra (SpMV, TrSV, etc)
  - incorporated into PETSc
- Collection of low-level primitives that provides automatically tuned computational kernels on sparse matrices, for use by solver libraries and applications.
- Current implementation targets cache-based superscalar uniprocessor machines.
- Non-trivial run-time tuning cost for SpMV: up to ~40 mat-vecs
  - Dominated by conversion time
- Design point: user calls “tune” routine explicitly
  - Exposes cost
  - Tuning time limited using estimated workload
    - Provided by user or inferred by library
- The work presented here only optimizes SpMV execution.
Goal: A system that works on single core, multicore, multisocket, CluMPs.
Optimizations Added to OSKI

- **Matrix Specific Allocation**
  - Each matrix can have an associated allocator/deallocator and any auxiliary information
  - A parallel layer on top can provide NUMA-Aware allocator/deallocator and store thread information in the ‘info’ field
  - Important to preserve locality

- **Prefetching**
  - Prefetch Column Index and Values Array into cache
  - Prefetch distance currently fixed at 256 bytes ahead for Values Array and 128 bytes ahead for Column Index Array
  - Future version will tune over prefetch distance
Pthread Parallelization of OSKI by Ankit Jain

Goal: Provide the optimizations presented in Williams, et al. in a distributable library for others to use.

Data Decomposition

- Exhaustive search over all combinations of data decompositions with available software threads (Limited to 32 for this study)
- Threads load balanced by number of nonzeros
- E.g: 4x2 Decomposition of a Matrix

pBench

- Run multiple instances of SpMV for a dense matrix stored in sparse format for all block sizes in parallel
- Benchmarked for all powers of 2 threads up to 64
## Summary of Optimizations

<table>
<thead>
<tr>
<th>No.</th>
<th>Optimization</th>
<th>OSKI 1.0.1h</th>
<th>OSKI 1.1</th>
<th>pOSKI</th>
<th>Williams et al.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Register Blocking</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>2.</td>
<td>Cache Blocking</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>3.</td>
<td>Software Prefetching</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>4.</td>
<td>Matrix Compression</td>
<td>(✔)</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>5.</td>
<td>Serial Benchmarking</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Data Decomposition within Shared Memory</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>Parallel Benchmarking</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>Data Decomposition across Distributed Memory</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>SIMDization</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>Software Pipelining</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>TLB Blocking</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Architecture Studied

AMD Opteron X2
Preliminary Results

Matrix Name

- bibd_22_8.pua
- dense2.pua
- ex11.rua
- marca_tcomm.rua
- mc2depi.rua
- raefsky4.rua
- rail4284s.pua
- scircuit.rua
- webbase-1M.rua

MFlop/s

- naïve csr
Preliminary Results

The diagram shows the performance of various matrix names using different algorithms. The x-axis represents the matrix names, and the y-axis represents MFlop/s. The diagram compares two algorithms:

- `oski 1.0.1h`
- `naïve csr`

The matrix names are:

- `bibd_22_8.pua`
- `dense2.pua`
- `ex11.rua`
- `marca_tcomm.rua`
- `mc2depi.rua`
- `raefsky4.rua`
- `rail4284s.pua`
- `scircuit.rua`
- `webbase-1M.rua`
Preliminary Results

Matrix Name

- bibd_22_8.pua
- dense2.pua
- ex11.rua
- marca_tcomm.rua
- mc2depi.rua
- raefsky4.rua
- rail4284s.pua
- scircuit.rua
- webbase-1M.rua

MFlop/s

- oski 1.1
- oski 1.0.1h
- naïve csr
“Untuned pOSKI” uses tuned OSKI, but no tuning at the pOSKI layer
Preliminary Results

Matrix Name

MFlop/s

Matrix Name

MFlop/s

poski + numa
untuned poski
oski 1.1
oski 1.0.1h
naïve csr
Preliminary Results

Matrix Name

poski + numa + pbench
poski + numa
untuned poski
oski 1.1
oski 1.0.1h
naïve csr

MFlop/s

Matrix Name

bibd_22_8.pua
dense2.pua
ex11.rua
marca_tcomm.rua
mc2depi.rua
raefsky4.rua
rail4284s.pua
scircuit.rua
webbase-1M.rua
Coming Soon…

- Search over Prefetch Distances
  - During serial tuning step
  - Optimal distance has been found to be matrix specific
- Add Matrix Compression to OSKI since it is the next most effective optimization
- Collect Data on other systems
  - Currently building pOSKI on Intel Xeon(Clovertown) and Sun Victoria Falls(Maramba)
- A release of pOSKI
Overview

- Other autotuning at Berkeley

- Recent work autotuning three parallel kernels
  - Sparse Matrix Vector Multiply (SpMV)
  - Lattice Boltzmann MHD
  - Stencils (Heat Equation)

- Integrating SpMV Advances into OSKI

- Towards a framework for building autotuners
  - What is the role of the compiler?

- Open questions
Common Optimizations

- Threading and Parallelization
  - Thread blocking
- Maximizing in-core performance
  - Loop unrolling/reordering
  - SIMDization
- Maximizing memory bandwidth
  - Limiting number of memory streams
  - NUMA-Aware (collocating data with processing threads)
  - Software prefetching
- Minimizing memory traffic (Addressing 3 C’s model)
  - Padding (Conflict misses)
  - Cache blocking (Capacity misses)
  - Cache bypass (via intrinsic for x86) (Compulsory misses)
Common Optimizations

Reinventing the Wheel?

- Threading and Parallelization
  - Thread blocking

- Maximizing in-core performance
  - Loop unrolling/reordering
  - SIMDization

- Maximizing memory bandwidth
  - Limiting number of memory streams
  - NUMA-Aware (collocating data with processing threads)
  - Software prefetching

- Minimizing memory traffic (Addressing 3 C’s model)
  - Padding (conflict misses)
  - Cache blocking (capacity misses)
  - Cache bypass (via intrinsic for x86)
Common Optimizations

- Threading and Parallelization
  - Thread blocking
- Maximizing in-core performance
  - Loop unrolling/reordering
  - SIMDization
- Maximizing memory bandwidth
  - Limiting number of memory streams
  - NUMA-Aware (collocating data with processing threads)
  - Software prefetching
- Minimizing memory traffic (Addressing 3 C’s model)
  - Padding (conflict misses)
  - Cache blocking (capacity misses)
  - Cache bypass (via intrinsic for x86)
Why Are We Autotuning Then?

- Domain-specific knowledge gives more information than is available from just the source code
  - e.g. guaranteed no aliasing, ATLAS anecdotes, SpMV unique indirection
- “The focus on specialized tuning systems is too narrow, and so only compilers, which apply most broadly, are the most sensible investment.”
  - Yes and No.
- “What can we do to build common tool bases for compiler-based autotuning and for construction of self-tuning or autotuning libraries?”
  - Leverage compiler infrastructure
  - Compiler & autotuning community must cooperate
- “Self-tuned libraries will always outperform compiler-generated code.”
  - cooperatively-tuned libraries/code will perform better than “self-tuning” alone or “compiler-generated” alone
What I Want

An Expert Interface to the Compiler

- Use existing compiler infrastructure + domain knowledge
- Parsing, AST transformations, SIMDization, code generation
  - I don’t want to replicate these
  - I want to add my own transformations or SIMDization logic
- Are pragmas enough?

- For my autotuning project, I need all of these capabilities!
What issues are we as a community ignoring?

- Different metrics for success
  - Power.

- Composition/scheduling
  - how can all these autotuned libraries + autotuners + compilers work together if each piece of code is tuned independently?

- Usability!
  - Code availability & licensing, platform independence
  - Does it actually compile
  - Is the pain worth the performance improvement