Tuning 3D Stencil Codes

By Kaushik Datta\textsuperscript{1}, Shoaib Kamil\textsuperscript{1,2}, Samuel Williams\textsuperscript{1,2}, Leonid Oliker\textsuperscript{2}, John Shalf\textsuperscript{2} and Katherine A. Yelick\textsuperscript{1,2}

\textsuperscript{1}BeBOP Project, U.C. Berkeley
\textsuperscript{2}Lawrence Berkeley National Laboratory

Presented by Kaushik Datta
kdatta@cs.berkeley.edu
CScADS Autotuning Workshop
July 11, 2007
Outline

- Introduction to Stencil Codes
- Hardware/Software Optimizations
- Inner Loop Tuning
- Open Design Questions
What are stencil codes?

- For a given point, a *stencil* is a pre-determined set of nearest neighbors (possibly including itself).
- A *stencil code* updates every point in a regular grid with a weighted subset of its neighbors (“applying a stencil”).
Stencils are critical to many scientific applications:
- Diffusion, Electromagnetics, Computational Fluid Dynamics
- Both explicit and implicit iterative methods (e.g. Multigrid)
- Both uniform and adaptive block-structured meshes

Many type of stencils
- 1D, 2D, 3D meshes
- Number of neighbors (5-pt, 7-pt, 9-pt, 27-pt, ...)
- Gauss-Seidel (update in place) vs Jacobi iterations (2 meshes)

Our study focuses on 3D, 7-point, Jacobi iteration
void stencil3d(double A[], double B[], int nx, int ny, int nz) {
    for all grid indices in x-dim {
        for all grid indices in y-dim {
            for all grid indices in z-dim {
                B[center] = S0* A[center] +
            }
        }
    }
}
Stencil Codes Achieve Low % of Machine Peak

% of Machine Peak for One Iteration of Naive Stencil Code

- Itanium2
- Opteron
- Power5

Cubic Grid Dimension
Potential Optimizations

- Performance is limited by memory bandwidth and latency
  - Re-use is limited to the number of neighbors in a stencil
  - For large meshes (e.g., $512^3$), cache blocking helps
  - For smaller meshes, stencil time is roughly the time to read the mesh once from main memory
  - Tradeoff of blocking: reduces cache misses (bandwidth), but increases prefetch misses (latency)
  - See previous paper for details [Kamil et al, MSP ’05]
- We look at merging across iterations to improve reuse
Outline

- Introduction to Stencil Codes
- Hardware/Software Optimizations
- Inner Loop Tuning
- Open Design Questions
Optimization Strategies

- Two software techniques
  - *Cache oblivious* algorithm recursively subdivides
  - *Cache conscious* has an explicit block size
- Two hardware techniques
  - Fast memory (*cache*) is managed by hardware
  - Fast memory (*local store*) is managed by application software

If hardware forces control, software cannot be oblivious.
Opt. Strategy #1: Cache Oblivious

- Two software techniques
  - *Cache oblivious* algorithm recursively subdivides
    - Elegant Solution
    - No explicit block size
    - No need to tune block size
  - *Cache conscious* has an explicit block size
- Two hardware techniques
  - Cache managed by hw
    - Less programmer effort
  - Local store managed by sw
Cache Oblivious Algorithm

- Developed by Frigo and Strumpen
- Recursive algorithm consists of *space cuts*, *time cuts*, and a base case
- Operates on well-defined trapezoid \((x_0, dx_0, x_1, dx_1, t_0, t_1)\):

  ![Diagram of Cache Oblivious Algorithm](image)

  - Trapezoid for 1D problem; our experiments are for 3D (shrinking cube)
Cache Oblivious Algorithm - Base Case

- If the height=1, then we have a line of points \((x_0:x_1, t_0)\):

- At this point, we stop the recursion and perform the stencil on this set of points
- Order does not matter since there are no inter-dependencies
Cache Oblivious Algorithm - Space Cut

- If trapezoid width >= 2*height, cut with slope=-1 through the center:

- Since no point in Tr1 depends on Tr2, execute Tr1 first and then Tr2
- In multiple dimensions, we try space cuts in each dimension before proceeding
Cache Oblivious Algorithm - Time Cut

- Otherwise, cut the trapezoid in half in the time dimension:

  ![Diagram](image)

- Again, since no point in Tr1 depends on Tr2, execute Tr1 first and then Tr2
Poor Itanium 2 Cache Oblivious Performance

- Fewer cache misses BUT longer running time
Poor Cache Oblivious Performance

- Much slower on Opteron and Power5 too

Opteron Cycle Comparison

Power5 Cycle Comparison
Improving Cache Oblivious Performance

- Fewer cache misses did NOT translate to better performance:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra function calls</td>
<td>Inlined kernel</td>
</tr>
<tr>
<td>Poor prefetch behavior</td>
<td>No cuts in unit-stride dimension</td>
</tr>
<tr>
<td>Recursion stack overhead</td>
<td>Maintain explicit stack</td>
</tr>
<tr>
<td>Modulo Operator</td>
<td>Pre-computed lookup array</td>
</tr>
<tr>
<td>Recursion even after block fits in cache</td>
<td>Early cut off of recursion</td>
</tr>
</tbody>
</table>
Cache Oblivious Performance

- Only Opteron shows any benefit
Opt. Strategy #2: Cache Conscious

- Two software techniques
  - *Cache oblivious* algorithm recursively subdivides
  - *Cache conscious* has an explicit block size
    - Easier to visualize
    - Tunable block size
    - No recursion stack overhead
- Two hardware techniques
  - Cache managed by hw
    - Less programmer effort
  - Local store managed by sw

<table>
<thead>
<tr>
<th>Hardware Cache (Implicit)</th>
<th>Local Store (Explicit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Conscious</td>
<td>N/A</td>
</tr>
<tr>
<td>Cache Oblivious</td>
<td></td>
</tr>
<tr>
<td>Local Store</td>
<td></td>
</tr>
<tr>
<td>Cache Conscious on Cell</td>
<td></td>
</tr>
</tbody>
</table>
Like the cache oblivious algorithm, we have space cuts.
However, cache conscious is NOT recursive and \textit{explicitly} requires cache block dimension $c$ as a parameter.

Again, trapezoid for a 1D problem above.
Cache Blocking with Time Skewing Animation
Cache Conscious - Optimal Block Size Search

<table>
<thead>
<tr>
<th>X-Dimension of Cache Block</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.2</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>8</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>16</td>
<td>1.5</td>
<td>1.6</td>
<td>1.5</td>
<td>1.4</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>32</td>
<td>1.6</td>
<td>1.7</td>
<td>1.6</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>64</td>
<td>1.6</td>
<td>1.7</td>
<td>1.7</td>
<td>1.5</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>128</td>
<td>1.6</td>
<td>1.7</td>
<td>1.7</td>
<td>1.6</td>
<td>1.3</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>256</td>
<td>1.6</td>
<td>1.7</td>
<td>1.7</td>
<td>1.7</td>
<td>1.3</td>
<td>0.9</td>
<td>0.8</td>
</tr>
</tbody>
</table>

4 Iter: GFlop Rate

GOOD

Legend:
- Light Blue: Good
- Dark Blue: Better
- Green: Best
Cache Conscious - Optimal Block Size Search

- Reduced memory traffic does correlate well to higher GFlop rates
Cache Conscious Performance

- Cache conscious measured with optimal block size on each platform
- Itanium 2 and Opteron both improve
Create a Performance Model!

**GOAL**: Find optimal cache block size without exhaustive search

- Most important factors: *memory traffic* and *prefetching*
- First count the number of cache misses
  - Inputs: cache size, cache line size, and grid size
  - Model then classifies block sizes into 5 cases
  - Misses are classified as either “fast” or “slow”
- Then predict memory performance by factoring in prefetching
  - STriad microbenchmark determines cost of “fast” and “slow” misses
  - Combine with cache miss model to compute running time
- If memory time is less than compute time, use compute time
  - Tells us we are compute-bound for that iteration
Memory Read Traffic Model

Memory Read Traffic (Bytes/Stencil) for 4 Iter of a $256^3$ Problem [Itanium 2]
Memory Read Traffic Model

- Diagonal is where problem is falling out of cache
- We underpredict since we can’t use entire cache
Performance Model

Time/Iteration (sec.) for 4 Iter of a 256^3 Problem [Itanium 2]

- Actual
- Predicted
- Processor Predicted

G O O D
Performance Model

4 Iter: Percent Error in GFlop Rate

Y-Dimension of Cache Block

X-Dimension of Cache Block

Overpredict

Underpredict

Overpredict

Underpredict

[Grid showing percent error with colors indicating overprediction and underprediction]
Performance Model Benefits

- Avoids exhaustive search
- Identifies performance bottlenecks
  - Allows us to tune appropriately
- Eliminates poor block sizes
  - But, does not choose best block size (lack of accuracy)
  - Still need to do search over pruned parameter space
### Opt. Strategy #3: Cache Conscious on Cell

- **Two software techniques**
  - *Cache oblivious* algorithm recursively subdivides
  - *Cache conscious* has an explicit block size
    - Easier to visualize
    - Tunable block size
    - No recursion stack overhead
- **Two hardware techniques**
  - Cache managed by hw
  - Local store managed by sw
    - Eliminate extraneous reads/writes

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache (Implicit)</td>
<td>Oblivious (Implicit)</td>
</tr>
<tr>
<td>Cache Conscious on Cell</td>
<td>Cache Conscious (Explicit)</td>
</tr>
<tr>
<td>Cache Oblivous</td>
<td>Local Store (Explicit)</td>
</tr>
<tr>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>
Cell Processor

• PowerPC core that controls 8 simple SIMD cores ("SPE’s")
• Memory hierarchy consists of:
  – Registers
  – Local memory
  – External DRAM
• Application *explicitly* controls memory:
  – Explicit DMA operations required to move data from DRAM to each SPE’s local memory
  – Effective for predictable data access patterns
• Cell code contains more low-level intrinsics than prior code
Cell Local Store Blocking

Stream in planes from source grid

SPE local store

Stream out planes to target grid
Excellent Cell Processor Performance

- Double-Precision (DP) Performance: 7.3 GFlops/s
- DP performance still relatively weak
  - Only 1 floating point instruction every 7 cycles
  - Problem becomes computation-bound when cache-blocked
- Single-Precision (SP) Performance: 65.8 GFlops/s!
  - Problem now memory-bound even when cache-blocked
- If Cell had better DP performance or ran in SP, could take further advantage of cache blocking
Summary - Computation Rate Comparison

[Bar chart showing computation rates for different processors with labels: Itanium 2, Opteron, Power 5, 2.4 GHz Cell, and 3.2 GHz Cell.]
Summary - Algorithmic Peak Comparison

![Bar chart comparing algorithmic peak performance.](chart.png)
Observations

- Cache-blocking performs better when explicit
  - But need to choose right cache block size for architecture
- Software-controlled memory boosts stencil performance
  - Caters memory accesses to given algorithm
  - Works especially well due to predictable data access patterns
- Low-level code gets closer to algorithmic peak
  - Eradicates compiler code generation issues
  - Application knowledge allows for better use of functional units
Outline

- Introduction to Stencil Codes
- Hardware/Software Optimizations
- Inner Loop Tuning
- Open Design Questions
Inner Loop Optimizations

**GOAL**: Find best combination of inner loop optimizations

- Many possible optimizations, like:
  - Loop Unrolling: How often? Which dimension?
  - Explicit SW Prefetching: How far ahead? For reads *and* writes?
  - Common subexpression elimination (like NAS MG Benchmark)
  - Choosing appropriate compiler flags, pragmas, etc.

- These optimizations are usually *not* independent
  - Optimizations affect generated instructions in complex ways
  - Interactions are often difficult to model
Exploring Inner Loop Optimizations

All results in GFlops/sec.
Choosing Inner Loop Optimizations

- Every optimization helped (at least a little) in this case
- This will not likely be general case
- Harder to model- likely search instead
Outline

- Introduction to Stencil Codes
- Hardware/Software Optimizations
- Inner Loop Tuning
- Open Design Questions
Open Design Questions

- How to design the user API (e.g. how to specify a stencil/weights)?
- What tuning parameters should be chosen?
- How do we traverse parameter space?
- Where do we search or use performance model?
- What code generator language should be used?
- What intermediate forms should be represented?
- What should be done offline or at runtime?
- Should C or Fortran be generated?
Fortran vs. C on Opteron
Backup Slides
Memory System Performance

- Data based on simple model that only counts compulsory misses
- BW performance is inferred from computation performance