Bandwidth Avoiding Stencil Computations

By **Kaushik Datta**, Sam Williams, Kathy Yelick, and Jim Demmel, and others

**Berkeley Benchmarking and Optimization Group**
UC Berkeley
March 13, 2008

http://bebop.cs.berkeley.edu
kdatta@cs.berkeley.edu
Outline

- Stencil Introduction
- Grid Traversal Algorithms
- Serial Performance Results
- Parallel Performance Results
- Conclusion
Outline

- Stencil Introduction
- Grid Traversal Algorithms
- Serial Performance Results
- Parallel Performance Results
- Conclusion
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 constant weighted subset of its neighbors ("applying a stencil").

2D Stencil

3D Stencil
Stencil Applications

- 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)

- This talk focuses on 3D, 7-point, Jacobi iteration
Naïve Stencil Pseudocode (One iteration)

```c
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] +
            }
        }
    }
}
```
**2D Poisson Stencil - Specific Form of SpMV**

Stencil uses an *implicit* matrix
- No indirect array accesses!
- Stores a single value for each diagonal

3D stencil is analogous (but with 7 nonzero diagonals)

\[
T = \begin{pmatrix}
4 & -1 &   &   &   &   &   \\
-1 & 4 & -1 &   &   &   &   \\
   & -1 & 4 & -1 &   &   &   \\
   &   & -1 & 4 & -1 &   &   \\
   &   &   & -1 & 4 & -1 &   \\
   &   &   &   & -1 & 4 & -1 \\
   &   &   &   &   & -1 & 4
\end{pmatrix}
\]

Graph and “stencil”
Reduce Memory Traffic!

- Stencil performance usually limited by memory bandwidth
- **Goal**: Increase performance by minimizing memory traffic
  - Even more important for multicore!
- Concentrate on getting reuse both:
  - within an iteration
  - across iterations \((Ax, A^2x, ..., A^kx)\)
- Only interested in final result
Outline

• Stencil Introduction
• Grid Traversal Algorithms
• Serial Performance Results
• Parallel Performance Results
• Conclusion
Grid Traversal Algorithms

- One common technique
  - *Cache blocking* guarantees reuse within an iteration
- Two novel techniques
  - *Time Skewing* and *Circular Queue* also exploit reuse across iterations

* Under certain circumstances
## Grid Traversal Algorithms

- **One common technique**
  - *Cache blocking* guarantees reuse within an iteration
- **Two novel techniques**
  - *Time Skewing* and *Circular Queue* also exploit reuse across iterations

<table>
<thead>
<tr>
<th></th>
<th>Inter-iteration Reuse</th>
<th>Intra-iteration Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Naive</strong></td>
<td>No*</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Cache Blocking</strong></td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Time Skewing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Circular Queue</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Under certain circumstances
Naïve Algorithm

- Traverse the 3D grid in the usual way
  - No exploitation of locality
  - Grids that don’t fit in cache will suffer
Grid Traversal Algorithms

- One common technique
  - *Cache blocking* guarantees reuse within an iteration
- Two novel techniques
  - *Time Skewing* and *Circular Queue* also exploit reuse across iterations

<table>
<thead>
<tr>
<th>Cache Blocking</th>
<th>Inter-iteration Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Naive</th>
<th>Inter-iteration Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time Skewing</th>
<th>Inter-iteration Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circular Queue</td>
<td>Yes</td>
</tr>
</tbody>
</table>

* Under certain circumstances
Cache Blocking- Single Iteration At a Time

- Guarantees reuse within an iteration
  - “Shrinks” each plane so that three source planes fit into cache
  - However, no reuse across iterations

- In 3D, there is tradeoff between cache blocking and prefetching
  - Cache blocking reduces memory traffic by reusing data
  - However, short stanza lengths do not allow prefetching to hide memory latency

- **Conclusion**: When cache blocking, don’t cut in unit-stride dimension!
Grid Traversal Algorithms

- One common technique
  - *Cache blocking* guarantees reuse within an iteration
- Two novel techniques
  - *Time Skewing* and *Circular Queue* also exploit reuse across iterations

<table>
<thead>
<tr>
<th>Inter-iteration Reuse</th>
<th>Intra-iteration Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>No*</td>
<td>Yes</td>
</tr>
<tr>
<td>Naive</td>
<td>N/A</td>
</tr>
<tr>
<td>Time Skewing</td>
<td></td>
</tr>
<tr>
<td>Circular Queue</td>
<td></td>
</tr>
</tbody>
</table>

* Under certain circumstances
Time Skewing- Multiple Iterations At a Time

- Now we allow reuse across iterations
- Cache blocking now becomes trickier
  - Need to shift block after each iteration to respect dependencies
  - Requires cache block dimension $c$ as a parameter (or else cache oblivious)
  - We call this "Time Skewing" [Wonnacott '00]

- Simple 3-point 1D stencil with 4 cache blocks shown above
2-D Time Skewing Animation

- Since these are Jacobi iterations, we alternate writes between the two arrays after each iteration

- Cache Block #1
  - Cache Block #2
  - Cache Block #3
  - Cache Block #4

- y (unit-stride)
Time Skewing Analysis

- **Positives**
  - Exploits reuse across iterations
  - No redundant computation
  - No extra data structures

- **Negatives**
  - Inherently sequential
  - Need to find optimal cache block size
    - Can use exhaustive search, performance model, or heuristic
  - As number of iterations increases:
    - Cache blocks can “fall” off the grid
    - Work between cache blocks becomes more imbalanced
Time Skewing - Optimal Block Size Search
Time Skewing- Optimal Block Size Search

- Reduced memory traffic does correlate to higher GFlop rates
Grid Traversal Algorithms

- One common technique
  - *Cache blocking* guarantees reuse within an iteration
- Two novel techniques
  - *Time Skewing* and *Circular Queue* also exploit reuse across iterations

<table>
<thead>
<tr>
<th></th>
<th>Inter-iteration Reuse</th>
<th>Intra-iteration Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>No*</td>
<td>No*</td>
</tr>
<tr>
<td>Cache Blocking</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>Time Skewing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Circular Queue</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Under certain circumstances
2-D Circular Queue Animation

Read array  First iteration  Second iteration  Write array
Parallelizing Circular Queue

- Each processor receives a colored block
- Redundant computation when performing multiple iterations

Stream in planes from source grid

Stream out planes to target grid
Circular Queue Analysis

**Positives**
- Exploits reuse across iterations
- Easily parallelizable
- No need to alternate the source and target grids after each iteration

**Negatives**
- Redundant computation
  - Gets worse with more iterations
- Need to find optimal cache block size
  - Can use exhaustive search, performance model, or heuristic
- Extra data structure needed
  - However, minimal memory overhead
Algorithm Spacetime Diagrams

**Naive**

<table>
<thead>
<tr>
<th>1st Block</th>
<th>2nd Block</th>
<th>3rd Block</th>
<th>4th Block</th>
</tr>
</thead>
</table>

**Cache Blocking**

<table>
<thead>
<tr>
<th>1st Block</th>
<th>2nd Block</th>
<th>3rd Block</th>
<th>4th Block</th>
</tr>
</thead>
</table>

**Time Skewing**

<table>
<thead>
<tr>
<th>1st Block</th>
<th>2nd Block</th>
<th>3rd Block</th>
<th>4th Block</th>
</tr>
</thead>
</table>

**Circular Queue**

<table>
<thead>
<tr>
<th>1st Block</th>
<th>2nd Block</th>
<th>3rd Block</th>
<th>4th Block</th>
</tr>
</thead>
</table>
Outline

- Stencil Introduction
- Grid Traversal Algorithms
- **Serial Performance Results**
- Parallel Performance Results
- Conclusion
Serial Performance

- Single core of 1 socket x 4 core Intel Xeon (Kentsfield)
- Single core of 1 socket x 2 core AMD Opteron
Outline

• Stencil Introduction
• Grid Traversal Algorithms
• Serial Performance Results
• Parallel Performance Results
• Conclusion
Multicore Performance

1 iteration of $256^3$ Problem

- **Left side:**
  - Intel Xeon (Clovertown)
  - 2 sockets x 4 cores
  - Machine peak DP: 85.3 GFlops/s

- **Right side:**
  - AMD Opteron (Rev. F)
  - 2 sockets x 2 cores
  - Machine peak DP: 17.6 GFlops/s
Outline

- Stencil Introduction
- Grid Traversal Algorithms
- Serial Performance Results
- Parallel Performance Results
- Conclusion
Stencil Code Conclusions

• Need to autotune!
  – Choosing appropriate algorithm AND block sizes for each architecture is not obvious
  – Can be used with performance model
  – My thesis work :)
• Appropriate blocking and streaming stores most important for x86 multicore
• Getting good performance out of x86 multicore chips is hard!
  – Applied 6 different optimizations, all of which helped at some point
Backup Slides
Poisson’s Equation in 1D

Discretize:
\[ \frac{d^2u}{dx^2} = f(x) \]
on regular mesh:
\[ u_i = u(i*h) \]
to get:
\[ \frac{[u_{i+1} - 2u_i + u_{i-1}]}{h^2} = f(x) \]
Write as solving:
\[ Tu = -h^2 * f \]
for \( u \) where

\[ T = \begin{pmatrix}
2 & -1 \\
-1 & 2 & -1 \\
-1 & 2 & -1 \\
-1 & 2 & -1 \\
\end{pmatrix} \]
Cache Blocking with Time Skewing Animation
Cache Conscious Performance

- Cache conscious measured with optimal block size on each platform
- Itanium 2 and Opteron both improve
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
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
Summary - Algorithmic Peak Comparison

![Bar chart showing percentage of algorithmic peak across different processors and cell types.]

- Naive Unaliased
- Cache Oblivious
- Cache Conscious

<table>
<thead>
<tr>
<th>Processor/Cell Type</th>
<th>Naive Unaliased</th>
<th>Cache Oblivious</th>
<th>Cache Conscious</th>
</tr>
</thead>
<tbody>
<tr>
<td>Itanium 2</td>
<td>41</td>
<td>24</td>
<td>54</td>
</tr>
<tr>
<td>Opteron</td>
<td>34</td>
<td>37</td>
<td>34</td>
</tr>
<tr>
<td>Power 5</td>
<td>23</td>
<td>24</td>
<td>45</td>
</tr>
<tr>
<td>2.4 GHz Cell</td>
<td>87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2 GHz Cell</td>
<td>88</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outline

• Stencil Introduction
• Grid Traversal Algorithms
• Serial Performance Results
• Parallel Performance Results
• Conclusion