Implicit and Explicit Optimizations for Stencil Computations

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

$^1$BeBOP Project, U.C. Berkeley
$^2$Lawrence Berkeley National Laboratory
October 22, 2006

http://bebop.cs.berkeley.edu
kdatta@eecs.berkeley.edu
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”)

![2D Stencil](image1.png) ![3D Stencil](image2.png)
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)

- Our study 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] +
            }
        }
    }
}
```
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
  - Three techniques with varying level of control

- We vary architecture types
  - Significant work (not shown) on low level optimizations
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

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

- N/A
Cache Oblivious Algorithm

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

  \[\text{dx}_0 \quad \text{dx}_1\]

  \[\text{t}_0 \quad \text{t}_1\]

- 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 (x0:x1, t0):

  ![Diagram showing a line of points in space and time]

  - 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 $\geq 2 \times$ 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 showing a trapezoid divided into two regions, Tr1 and Tr2, with time axis and space axis labeled.]

- 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
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</th>
<th>Local Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache (Implicit)</td>
<td>Cache Conscious on Cell</td>
</tr>
<tr>
<td>Cache Conscious</td>
<td>N/A</td>
</tr>
<tr>
<td>Cache Oblivious</td>
<td></td>
</tr>
</tbody>
</table>
Like the cache oblivious algorithm, we have space cuts.

However, cache conscious is NOT recursive and *explicitly* requires cache block dimension \( c \) as a parameter.

Again, trapezoid for a 1D problem above.
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - 3D Animation
Cache Conscious - Optimal Block Size Search

Iteration #4: Mem. Read Traffic (Bytes/Stencil)

<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>256</td>
<td>22.3</td>
<td>21.0</td>
<td>20.8</td>
<td>20.8</td>
<td>20.6</td>
<td>20.2</td>
<td>19.8</td>
</tr>
<tr>
<td>128</td>
<td>10.6</td>
<td>17.5</td>
<td>17.4</td>
<td>17.1</td>
<td>16.6</td>
<td>16.5</td>
<td>16.5</td>
</tr>
<tr>
<td>64</td>
<td>4.7</td>
<td>6.5</td>
<td>15.7</td>
<td>17.1</td>
<td>16.8</td>
<td>16.7</td>
<td>16.6</td>
</tr>
<tr>
<td>32</td>
<td>4.1</td>
<td>2.4</td>
<td>4.8</td>
<td>15.6</td>
<td>17.2</td>
<td>17.1</td>
<td>17.0</td>
</tr>
<tr>
<td>16</td>
<td>4.1</td>
<td>2.0</td>
<td>1.5</td>
<td>5.4</td>
<td>16.7</td>
<td>18.0</td>
<td>17.9</td>
</tr>
<tr>
<td>8</td>
<td>4.0</td>
<td>2.0</td>
<td>1.0</td>
<td>1.5</td>
<td>8.0</td>
<td>19.1</td>
<td>19.7</td>
</tr>
<tr>
<td>4</td>
<td>4.0</td>
<td>2.0</td>
<td>1.0</td>
<td>0.9</td>
<td>4.6</td>
<td>13.8</td>
<td>23.0</td>
</tr>
</tbody>
</table>

Y-Dimension of Cache Block

Legend:
- 0
- 5
- 10
- 15
- 20
- 25
### Cache Conscious - Optimal Block Size Search

**Iteration #4: GFlop Rate**

<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>1.9</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
<td>2.0</td>
<td>1.2</td>
</tr>
<tr>
<td>8</td>
<td>1.9</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
<td>2.0</td>
<td>1.2</td>
</tr>
<tr>
<td>16</td>
<td>1.9</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
<td>2.0</td>
<td>1.2</td>
</tr>
<tr>
<td>32</td>
<td>1.8</td>
<td>2.0</td>
<td>1.9</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>64</td>
<td>1.7</td>
<td>1.8</td>
<td>1.5</td>
<td>1.4</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>128</td>
<td>1.6</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>256</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
<td>1.4</td>
</tr>
</tbody>
</table>

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

- Cache conscious measured with optimal block size on each platform
- Itanium 2 and Opteron both improve
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>Cache Conscious (Explicit)</td>
</tr>
<tr>
<td>Cache Conscious</td>
<td>N/A</td>
</tr>
<tr>
<td>Local Store (Explicit)</td>
<td>Cache Oblivious</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

Stream out planes to target grid

SPE local store
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

![Graph comparing computation rates for different processors and setups.

- **Naive Unaliased**
- **Cache Oblivious**
- **Cache Conscious**

Processor Comparison:
- **Itanium 2**: 1.3 GFlop Rate
- **Opteron**: 0.8 GFlop Rate
- **Power 5**: 1.7 GFlop Rate
- **2.4 GHz Cell**: 5.5 GFlop Rate
- **3.2 GHz Cell**: 7.3 GFlop Rate

The graph illustrates the computation rate for each processor setup, highlighting the performance differences between the Naive Unaliased, Cache Oblivious, and Cache Conscious methods.
Summary - Algorithmic Peak Comparison

![Graph showing comparison of different processors and processors.

- Itanium 2: Naive Unaliased 41, Cache Oblivious 24, Cache Conscious 54
- Opteron: Naive Unaliased 34, Cache Oblivious 37, Cache Conscious 17
- Power 5: Naive Unaliased 45, Cache Oblivious 23, Cache Conscious 24
- 2.4 GHz Cell: Naive Unaliased 87, Cache Oblivious 24, Cache Conscious 88
- 3.2 GHz Cell: Naive Unaliased 88, Cache Oblivious 24, Cache Conscious 88]
Stencil Code Conclusions

- 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