An Auto-Tuning Framework for Parallel Multicore Stencil Computations
Stencils
What is a Stencil Computation?

- Nearest Neighbor Computations
  - E.g. finite difference between data points

- Sweeps over a structured Grid
  - Like a n-dimensional Array
  - Iterative: $i \rightarrow i+1 \rightarrow i+2$

Left Two: http://iopscience.iop.org/1749-4699/2/1/015005/fulltext
Middle: http://en.wikipedia.org/wiki/Stencil_(numerical_analysis)
Right: http://en.wikipedia.org/wiki/Five-point_stencil
Example: 2D 5-Points-Stencil

//Stencil-loop
do k=2, xLength-1, 1
    do i=2, yLength-1, 1
        writeArray[k][i] = useStencil(k,i)
    enddo
enddo

//Stencil-function
function useStencil(k,i)
    int result = readArray[k][i]
    + readArray[k+1][i]
    + readArray[k-1][i]
    + readArray[k][i+1]
    + readArray[k][i-1]
    result = result/5
    return result
endfunction
Example

(readArray)

(2+1+3+3+8)/5 = 3
## Example

**readArray**

<table>
<thead>
<tr>
<th>5</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>8</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>77</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

\[(3+3+3+7+7)/5 = 4\]

**writeArray**

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

(readArray)

<table>
<thead>
<tr>
<th>5</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>8</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>77</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

(writeArray)

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\frac{(1+3+7+3+6)}{5} = 4$
Example from the paper: Gradient \n
\[
\begin{align*}
\text{do } & k=2, nz-1, 1 \\
\text{do } & j=2, ny-1, 1 \\
\text{do } & i=2, nx-1, 1 \\
\end{align*}
\]

\[
x(i,j,k)=\alpha \ast (u(i+1,j,k)-u(i-1,j,k)) \\
y(i,j,k)=\beta \ast (u(i,j+1,k)-u(i,j-1,k)) \\
z(i,j,k)=\gamma \ast (u(i,j,k+1)-u(i,j,k-1))
\]

enddo
enddo
enddo
Why?

- Solving Partial Differential Equations
  - Used by many branches of Science
    - Heat Equations
    - Wave Equations
    - “Automatic beam path analysis of laser wakefield particle acceleration data”
    - ...

Images: http://www.math.uwaterloo.ca/~fpoulin/Files_html/fpcmresearch.html
Characteristics of stencil computations

- High memory traffic

- Low arithmetic intensity
  - CPUs can handle it

Computations are memory bound
  - Auto-tuning for better memory access management

```c
//Stencil-function
function useStencil(k,i)
    int result = readArray[k][i]
    + readArray[k+1][i]
    + readArray[k-1][i]
    + readArray[k][i+1]
    + readArray[k][i-1]
    result = result/5
    return result
endfunction
```
The Framework
Overview

- Not the first auto-tuning framework for stencils
  - But other work about static/single kernel instantiations

- Proof-of-Concept
  - Supports broad range of stencil kernels
    - Fully generalized framework
  - Auto-parallelisation
  - Multiple back-end architectures
    - Even a GPU
Framework flow

```
Reference Implementation
Parse as AST
Transformation Engine
Strategy Engine
  GPU  Parallel x86  Serial x86
Code Generators
  FORTRAN
  C with pthreads
  CUDA
Search Engines
  .c  .f95  .cu
Myriad of equivalent, optimized implementations
In context of Specific Problem
Best performing implementation and configuration parameters
```

Inspired by a picture of the paper
Strategy Engine

- Parameter Space is massive
  - Combined serial and parallel optimizations

- Decides on a appropriate subset of parameter combinations (strategies)
  - Based on the underlying architecture

- Knows about correlation of different optimizations
  - Chooses only legal combinations
Transformation Engine

- Transforms the AST
  - First applies auto-parallelization
  - Then uses auto-tuning

- Has domain knowledge
  - Can do transformations a compiler can not
Auto-parallelization

- Basically dividing the problem space into blocks
  - Core blocks, thread blocks and register blocks
  - Creates new loops for every block

- Non-Uniform Memory Access (NUMA)-Aware

- Separate stencil for the border cases

Image: http://www.1024cores.net/home/parallel-computing/cache-oblivious-algorithms
Auto-parallelization

Decomposition of a Node Block into a Chunk of Core Blocks

Decomposition into Thread Blocks

Decomposition into Register Blocks
Auto-tuning

- Loop unrolling and register blocking
  - Improves innermost loop efficiency

- Cache blocking
  - Exposes temporal locality and increases cache reuse

- Arithmetic simplifications

- Many more possible
  - It is a prove-of-concept

Example for cache blocking: http://techpubs.sgi.com/library/dynaweb_docs/0640/SGI_Developer/books/OrOn2_PfTune/sgi_html/ch06.html
Search Engine

- Runs all the different tuned versions of the stencil kernel
  - $256^3$ grids (16,777,216 Elements) initialized with random values

- User can replace the original kernel with the fastest one
Limitations

- Only 2D or 3D
- Only Arrays
  - No sophisticated Data structures
- Only arithmetic stencils
- They want to change that in future work
Code Generator

- Creates code from the modified ASTs
  - For the CPUs: pthreads
  - For the GPU: CUDA thread blocks
  - Serial fortran and c code also possible
Tested

Stencils and Architectures
Used Stencils

Laplacian Stencil

Divergence Stencil

Gradient Stencil

\[
\begin{align*}
\text{uNext}(i,j,k) &= \\
& \alpha u(i,j,k) + \\
& \beta(u(i+1,j,k)+u(i-1,j,k)+u(i,j+1,k)+u(i,j-1,k)+u(i,j,k+1)+u(i,j,k-1))
\end{align*}
\]

\[
\begin{align*}
\text{u}(i,j,k) &= \\
& \alpha(x(i+1,j,k)-x(i-1,j,k)) + \\
& \beta(y(i,j+1,k)-y(i,j-1,k)) + \\
& \gamma(z(i,j,k+1)-z(i,j,k-1))
\end{align*}
\]

\[
\begin{align*}
\text{x}(i,j,k) &= \alpha u(i+1,j,k)-u(i-1,j,k) \\
\text{y}(i,j,k) &= \beta u(i,j+1,k)-u(i,j-1,k) \\
\text{z}(i,j,k) &= \gamma u(i,j,k+1)-u(i,j,k-1)
\end{align*}
\]
# Used Architectures

<table>
<thead>
<tr>
<th>Core Architecture</th>
<th>AMD Barcelona</th>
<th>Intel Nehalem</th>
<th>Sun Niagara2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>superscalar out of order</td>
<td>superscalar out of order</td>
<td>HW multithread dual issue</td>
</tr>
<tr>
<td>Clock (GHz)</td>
<td>2.30</td>
<td>2.66</td>
<td>1.16</td>
</tr>
<tr>
<td>DP GFlop/s</td>
<td>9.2</td>
<td>10.7</td>
<td>1.16</td>
</tr>
<tr>
<td>Local-Store</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64KB</td>
<td>32KB</td>
<td>8KB</td>
</tr>
<tr>
<td>private L2 cache</td>
<td>512KB</td>
<td>256KB</td>
<td>—</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System Architecture</th>
<th>Opteron 2356 (Barcelona)</th>
<th>Xeon X5550 (Gainestown)</th>
<th>UltraSparc T5140 (Victoria Falls)</th>
</tr>
</thead>
<tbody>
<tr>
<td># Sockets</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Cores per Socket</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Threads per Socket‡</td>
<td>4</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>primary memory parallelism paradigm</td>
<td>HW prefetch</td>
<td>HW prefetch</td>
<td>Multithreading</td>
</tr>
<tr>
<td>shared L3 cache</td>
<td>2×2MB (shared by 4 cores)</td>
<td>2×8MB (shared by 4 cores)</td>
<td>2×4MB (shared by 8 cores)</td>
</tr>
<tr>
<td>DRAM Capacity</td>
<td>16GB</td>
<td>12GB</td>
<td>32GB</td>
</tr>
<tr>
<td>DRAM Pin Bandwidth (GB/s)</td>
<td>21.33</td>
<td>51.2</td>
<td>42.66(read) 21.33(write)</td>
</tr>
<tr>
<td>DP GFlop/s</td>
<td>73.6</td>
<td>85.3</td>
<td>18.7</td>
</tr>
<tr>
<td>DP Flop:Byte Ratio</td>
<td>3.45</td>
<td>1.66</td>
<td>0.29</td>
</tr>
<tr>
<td>Threading</td>
<td>Pthreads</td>
<td>Pthreads</td>
<td>Pthreads</td>
</tr>
<tr>
<td>Compiler</td>
<td>gcc 4.1.2</td>
<td>gcc 4.3.2</td>
<td>gcc 4.2.0</td>
</tr>
</tbody>
</table>
Results
One Result

Laplacian

![Graph showing performance of different configurations on a Nehalem system with varying threads. The x-axis represents the number of threads, and the y-axis represents GFlop/s. The configurations include baseline, auto-parallel, +NUMA, +auto-tuning, and OpenMP.]
Results
Conclusion

- **Pro**
  - It does work. Concept is proven
    - Fully general
  - Performance comparable to hand-optimized code
  - “Programmer Production Benefits”
    - Few minutes to annotate code

- **Contra**
  - OpenMP works good, too
  - New architecture means new coding
  - Peak not yet reached

*Quote from Paper*
End of Presentation