Compiler-Based Autotuning Technology

Lecture 2: Tuning Code with CHiLL

Mary Hall
July, 2011

* This work has been partially sponsored by DOE SciDAC as part of the Performance Engineering Research Institute (PERI), DOE Office of Science, the National Science Foundation, DARPA and Intel Corporation.
CHiLL from a User’s Perspective

• What is it like to tune code with CHiLL?
• Working through a series of examples
• No details on implementation and internal abstractions until tomorrow
• Higher-level abstractions in CUDA-CHiLL on Thursday
Two Different Ways to Use CHiLL

Compiler Developer’s View

Library/Application Developer’s View

ACACES 2011, L2: Tuning code with CHiLL
Outline for Today's Lecture

1. Basics on loop nest transformations
2. CHiLL basics
   a. Statements, loop level
   b. Set of transformations supported
   c. Additional annotations
3. Script examples and results
4. Optimizations for small matrix sizes
5. Optimizations for larger matrix sizes
1. Loop Transformation Basics: Applicability

- Focus is loop nest computations
  - Important to high-end application and library developers
  - Source of data-parallel code

- Mostly, loop nests in the affine domain
  - Array subscripts, loop bounds, control flow tests are linear functions of loop indices

- Generalization
  - Can mix non-affine constructs with care or user intervention
  - May require approximation
1. Loop Transformation Basics: Criteria for Applying Transformations

- Safety
  - After transformation, will the resulting code be “equivalent” to the original code?
- Profitability
  - After transformation, is the resulting code likely to be faster than the original code?

Key observation: With autotuning, we can afford to be very aggressive in predicting profitability and catch erroneous predictions through empirical data. This makes it possible to achieve very high performance with autotuning compilers.
1. Example: Matrix-Matrix Multiply

for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    for(k=0; k<n; k++)
      c[i][j]+=a[i][k]*b[k][j];
A script applies to a single loop nest in a specific procedure in a source code file.

Statements in the loop nest are numbered starting at 0 and are referred to by their number. Statements created by transformations are given new numbers.

Loop level within the loop nest identifies the subloop to which a transformation should be applied, coupled with statement number. Outermost loop is at level 1.

Source code for mxm.c

loop level 1: for(i=0; i<n; i++)
loop level 2: for(j=0; j<n; j++)
loop level 3: for(k=0; k<n; k++)
statement 0: c[i][j]+=a[i][k]*b[k][j];

Example CHiLL script

source: mxm.c
procedure: 0
loop: 0
permute([2,1,3])
unroll(0,3,2)
### 2b. CHiLL Basics: Set of Transformations

<table>
<thead>
<tr>
<th>Transformation and Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>permute ([stmt],[level],order)</td>
<td>Permute optional [stmt] to optional loop [level] according to order. Can omit [stmt] and [level] and entire loop nest is permuted.</td>
</tr>
<tr>
<td>unroll (stmt,level,unrollfactor)</td>
<td>Unroll loop at level for the subloop specified by stmt/level. Unroll by unrollfactor.</td>
</tr>
<tr>
<td>tile (stmt,level,ts,[outerlooplevel])</td>
<td>Tile loop at level for the subloop specified by stmt/level and tile size ts. Place controlling loop at optional [outerlooplevel] or defaults to outermost.</td>
</tr>
<tr>
<td>datacopy (stmt,level,array,[index])</td>
<td>Calculate footprint for all references to array in subloop specified by stmt/level and copy into temporary, replacing original accesses with copy. Optional [index] refers to fastest-changing dimension.</td>
</tr>
<tr>
<td>split(stmt,level,condition)</td>
<td>Split iteration space at subloop specified by stmt/level according to condition and its complement.</td>
</tr>
<tr>
<td>datacopy_privatized (stmt,level,array,[index])</td>
<td>Similar to datacopy, but creates a private copy in parallel thread code.</td>
</tr>
<tr>
<td><strong>Other transformations include:</strong></td>
<td>fuse, distribute, skew, scale, reverse, shift, peel, nonsingular</td>
</tr>
</tbody>
</table>
2c. CHiLL Basics: Annotations

- Two annotations are used to describe data properties
  - known(constraint): establishes additional constraints not derived from source code (e.g., to specialize for ranges of problem sizes)
  - remove_dep(stmt1,stmt2): eliminates dependences across two statements to enable transformations
3. Transformations: Loop Permutation

Permute the order of the loops to modify the traversal order

For row-major order:

\[
\text{for } (i=0; i<3; i++)
\text{ for } (j=0; j<6; j++)
\]

For column-major order:

\[
\text{for } (j=0; j<6; j++)
\text{ for } (i=0; i<3; i++)
\]

New traversal order!

*NOTE: C multi-dimensional arrays are stored in row-major order, Fortran in column major*
3. Permute Loops to New Order

Source code for mxm.c

loop level 1: for(i=0; i<n; i++)

loop level 2: for(j=0; j<n; j++)

loop level 3: for(k=0; k<n; k++)

statement 0: c[i][j]+=a[i][k]*b[k][j];

Resulting code:

for(j=0; j<n; j++)
    for(i=0; i<n; i++)
        for(k=0; k<n; k++)
            c[i][j]+=a[i][k]*b[k][j];

CHiLL script

source: mxm.c
procedure: 0
loop: 0
permute([2,1,3])

ACACES 2011, L2: Tuning code with CHiLL
3. Transformations: Unroll, Unroll-and-Jam

- Unroll simply replicates the statements in a loop, with the number of copies called the unroll factor.
  - As long as the copies don’t go past the iterations in the original loop, it is always safe
    - May require “cleanup” code
- Unroll-and-jam involves unrolling an outer loop and fusing together the copies of the inner loop (not always safe)
- One of the most effective optimizations there is, but there is a danger in unrolling too much

Original:
```c
for (i=0; i<4; i++)
    for (j=0; j<8; j++)
        A[i][j] = B[j+1][i];
```

Unroll j
```c
for (i=0; i<4; i++)
    for (j=0; j<8; j+=2)
        A[i][j] = B[j+1][i];
    A[i][j+1] = B[j+2][i];
```

Unroll-and-jam i
```c
for (i = 0; i<4; i+=2)
    for (j=0; j<8; j++)
        A[i][j] = B[j+1][i];
    A[i+1][j] = B[j+1][i+1];
```
3. Unroll loops at levels 2 and 3

Source code for mxm.c

loop level 1: for(i=0; i<128; i++)
loop level 2: for(j=0; j<128; j++)
loop level 3: for(k=0; k<128; k++)
statement 0: c[i][j]+=a[i][k]*b[k][j];

Resulting code:
for(j=0; j<128; j++) {
  for(i=0; i<128; i+=2) {
    for(k=0; k<128; k+=2) {
      c[i][j]+=a[i][k]*b[k][j];
      c[i][j]+=a[i][k+1]*b[k+1][j];
      c[i+1][j]+=a[i+1][k]*b[k][j];
      c[i+1][j]+=a[i+1][k+1]*b[k+1][j];
    }
  }
}
3. Annotation to specialize for n=10

Source code for mxm.c

loop level 1: for(i=0; i<n; i++)
loop level 2:    for(j=0; j<n; j++)
loop level 3:        for(k=0; k<n; k++)
statement 0:          c[i][j]+=a[i][k]*b[k][j];

Resulting code:
for(j=0; j<10; j++)
    for(i=0; i<10; i+=2)
        for(k=0; k<10; k+=2) {
            c[i][j]+=a[i][k]*b[k][j];
            c[i][j]+=a[i][k+1]*b[k+1][j];
            c[i+1][j]+=a[i+1][k]*b[k][j];
            c[i+1][j]+=a[i+1][k+1]*b[k+1][j];
        }

CHiLL script

source: mxm.c
procedure: 0
loop: 0
known(n=10)
permute([2,1,3])
unroll(0,2,2)
unroll(0,3,2)
4. Optimizations for small matrix sizes

- Previous example comes from optimizing nek5000 (Friday’s lecture)
- Involves optimizing for small matrix sizes
  - Set of expected sizes known and similar for different input data sets
- Specialization and optimizations specific to small matrices leads to very high performance

Example from nek5000
8 input sizes comprise 75% of execution time
4. Optimizations for small matrix sizes

- Optimization opportunities
  - exploit reuse in registers \textit{(unroll-and-jam)}
  - exploit SIMD (in the Opteron SSE) \textit{(permute, unroll)}
  - reduce loop overheads \textit{(unroll, specialize)}
4. Aside: Multimedia Extensions and How to Optimize for Them

- At the core of multimedia extensions
  - SIMD parallelism
  - Variable-sized data fields:
    - Vector length = register width / type size

Scalar: \texttt{add r1,r2,r3}

\begin{center}
\begin{tikzpicture}
  \node at (0,0) {1 \node{r3}};
  \node at (1,0) {2 \node{r2}};
  \node at (2,0) {3 \node{r1}};
  \draw (0.5,0) -- (1.5,0);
  \draw (0.5,-0.5) -- (1.5,-0.5);
  \draw (0.5,-1) -- (1.5,-1);
\end{tikzpicture}
\end{center}

SIMD: \texttt{vadd<sws> v1,v2,v3}

\begin{center}
\begin{tikzpicture}
  \node at (0,0) {1 \node{v3}};
  \node at (1,0) {2 \node{v2}};
  \node at (2,0) {3 \node{v1}};
  \draw (0.5,0) -- (1.5,0);
  \draw (0.5,-0.5) -- (1.5,-0.5);
  \draw (0.5,-1) -- (1.5,-1);
\end{tikzpicture}
\end{center}

\texttt{sws} refers to datatype for instruction-level configurability
4. Aside: Multimedia Extensions and How to Optimize for Them

- Data must be in adjacent memory locations
  - May need to copy to get adjacency (overhead)
- Data must be aligned to superword boundary
  - Unaligned data may produce incorrect results on older platforms
  - Alignment concerns lead to extra control (dynamic alignment)
- Control flow introduces complexity and inefficiency
- Exceptions may be masked
4. Optimizations for small matrix sizes

- Optimization opportunities
  - exploit reuse in registers (unroll-and-jam)
  - exploit SIMD (in the Opteron SSE) (permute, unroll)
  - reduce loop overheads (unroll, specialize)
4. Optimization Parameters and Variants

• For this very simple example, we have several parameters and variants
  - What is the right loop order? (variant)
  - Which loops to unroll? (treat no unrolling as parameter)
  - How much to unroll? (parameter)
4. Heuristics to Prune Search Space

- Focus on loop orders that are best for SSE code generation (3 out of 6):
  - \{123, 213, 231\}
- Unrolling: Limit for I-cache
  - \{U_i, U_j, U_k \leq 2197\} (limit derived empirically)
- Spatial locality for SIMD
  - \{U_i=1 \text{ or } U_j=1 \text{ or } U_k=1\}
- Avoid unrolling cleanup loop to streamline code:
  - \{M \mod U_i=0 \text{ and } N \mod U_j=0 \text{ and } K \mod U_k=0\}

Slide source: Jaewook Shin, ICS ‘10
### 4. Code Variants and Parameters Selected by Autotuning

<table>
<thead>
<tr>
<th>No.</th>
<th>m,k,n</th>
<th>Size</th>
<th>Loop Order</th>
<th>Ui</th>
<th>Uk</th>
<th>Uj</th>
<th>%max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8,10,8</td>
<td>3840</td>
<td>ijk</td>
<td>8</td>
<td>10</td>
<td>4</td>
<td>98.7</td>
</tr>
<tr>
<td>2</td>
<td>10,8,10</td>
<td>4800</td>
<td>ijk</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>10,10,10</td>
<td>6000</td>
<td>jik</td>
<td>1</td>
<td>9</td>
<td>5</td>
<td>99.3</td>
</tr>
<tr>
<td>4</td>
<td>10,8,64</td>
<td>30720</td>
<td>ijk</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8,10,100</td>
<td>48000</td>
<td>ijk</td>
<td>1</td>
<td>10</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>100,8,10</td>
<td>48000</td>
<td>jki</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>10,10,100</td>
<td>60000</td>
<td>jik</td>
<td>1</td>
<td>10</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>100,10,10</td>
<td>60000</td>
<td>jik</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
4. Impact of Using a Different Variant or Parameters

<table>
<thead>
<tr>
<th>No.</th>
<th>m,k,n</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8,10,8</td>
<td>58</td>
<td>27</td>
<td>49</td>
<td>38</td>
<td>58</td>
<td>49</td>
<td>56</td>
<td>54</td>
</tr>
<tr>
<td>2</td>
<td>10,8,10</td>
<td>43</td>
<td>61</td>
<td>58</td>
<td>20</td>
<td>20</td>
<td>51</td>
<td>39</td>
<td>58</td>
</tr>
<tr>
<td>3</td>
<td>10,10,10</td>
<td>39</td>
<td>37</td>
<td>59</td>
<td>31</td>
<td>20</td>
<td>52</td>
<td>44</td>
<td>58</td>
</tr>
<tr>
<td>4</td>
<td>10,8,64</td>
<td>44</td>
<td>20</td>
<td>54</td>
<td>62</td>
<td>61</td>
<td>47</td>
<td>62</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>8,10,100</td>
<td>57</td>
<td>38</td>
<td>57</td>
<td>38</td>
<td>59</td>
<td>50</td>
<td>59</td>
<td>54</td>
</tr>
<tr>
<td>6</td>
<td>100,8,10</td>
<td>27</td>
<td>73</td>
<td>74</td>
<td>19</td>
<td>19</td>
<td>75</td>
<td>58</td>
<td>67</td>
</tr>
<tr>
<td>7</td>
<td>10,10,100</td>
<td>39</td>
<td>37</td>
<td>58</td>
<td>39</td>
<td>61</td>
<td>52</td>
<td>61</td>
<td>57</td>
</tr>
<tr>
<td>8</td>
<td>100,10,10</td>
<td>26</td>
<td>41</td>
<td>71</td>
<td>34</td>
<td>19</td>
<td>62</td>
<td>60</td>
<td>75</td>
</tr>
</tbody>
</table>

(\% of peak)

Slide source: Jaewook Shin,
IWAPT ‘09
Example: loop order ijk, unroll 8-4-1 (Fortran)

FUNCTION M_100_10_8 (A, B, C)
INTEGER M_100_10_8, T4, T6
DOUBLE PRECISION A, B, C
DIMENSION A(8, 10)
DIMENSION B(10, 100)
DIMENSION C(8, 100)
DO 2, T4 = 1, 97, 4
C(1, T4) = 0.0000000000000000000D+00
C(1 + 1, T4) = 0.0000000000000000000D+00
C(1 + 2, T4) = 0.0000000000000000000D+00
C(1 + 3, T4) = 0.0000000000000000000D+00
C(1 + 4, T4) = 0.0000000000000000000D+00
C(1 + 5, T4) = 0.0000000000000000000D+00
C(1 + 6, T4) = 0.0000000000000000000D+00
C(1 + 7, T4) = 0.0000000000000000000D+00
C(1, T4 + 1) = 0.0000000000000000000D+00
C(1 + 1, T4 + 1) = 0.0000000000000000000D+00
C(1 + 2, T4 + 1) = 0.0000000000000000000D+00
C(1 + 3, T4 + 1) = 0.0000000000000000000D+00
C(1 + 4, T4 + 1) = 0.0000000000000000000D+00
C(1 + 5, T4 + 1) = 0.0000000000000000000D+00
C(1 + 6, T4 + 1) = 0.0000000000000000000D+00
C(1 + 7, T4 + 1) = 0.0000000000000000000D+00
C(1, T4 + 2) = 0.0000000000000000000D+00
C(1 + 1, T4 + 2) = 0.0000000000000000000D+00
C(1 + 2, T4 + 2) = 0.0000000000000000000D+00
C(1 + 3, T4 + 2) = 0.0000000000000000000D+00
C(1 + 4, T4 + 2) = 0.0000000000000000000D+00
C(1 + 5, T4 + 2) = 0.0000000000000000000D+00
C(1 + 6, T4 + 2) = 0.0000000000000000000D+00
C(1 + 7, T4 + 2) = 0.0000000000000000000D+00
C(1, T4 + 3) = 0.0000000000000000000D+00
C(1 + 1, T4 + 3) = 0.0000000000000000000D+00
C(1 + 2, T4 + 3) = 0.0000000000000000000D+00
C(1 + 3, T4 + 3) = 0.0000000000000000000D+00
C(1 + 4, T4 + 3) = 0.0000000000000000000D+00
C(1 + 5, T4 + 3) = 0.0000000000000000000D+00
C(1 + 6, T4 + 3) = 0.0000000000000000000D+00
C(1 + 7, T4 + 3) = 0.0000000000000000000D+00
4 CONTINUE
3 CONTINUE
2 CONTINUE
1 CONTINUE
M_100_10_8 = 0
RETURN
END

DO 4, T6 = 1, 10, 1
C(1, T4) = C(1, T4) + A(1, T6) * B(T6, T4)
C(1 + 1, T4) = C(1 + 1, T4) + A(1 + 1, T6) * B(T6, T4)
C(1 + 2, T4) = C(1 + 2, T4) + A(1 + 2, T6) * B(T6, T4)
C(1 + 3, T4) = C(1 + 3, T4) + A(1 + 3, T6) * B(T6, T4)
C(1 + 4, T4) = C(1 + 4, T4) + A(1 + 4, T6) * B(T6, T4)
C(1 + 5, T4) = C(1 + 5, T4) + A(1 + 5, T6) * B(T6, T4)
C(1 + 6, T4) = C(1 + 6, T4) + A(1 + 6, T6) * B(T6, T4)
C(1 + 7, T4) = C(1 + 7, T4) + A(1 + 7, T6) * B(T6, T4)
4 CONTINUE
3 CONTINUE
2 CONTINUE
1 CONTINUE
M_100_10_8 = 0
RETURN
END

ACACES 2011, L2: Tuning code with CHiLL
4. Automatically-Generated Code is Faster than Manually-Tuned Libraries

Target architecture: AMD Phenom, 2.5 GHz, data fits in 64 KB L1, 4 double-precision floating point operations / cycle ➔ 10 GFlops / core peak

ACACES 2011, L2: Tuning code with CHiLL
5. Optimizations for larger matrix sizes

What if data footprint exceeds cache capacity? And there is data reuse?

- exploit locality of reused data in various levels of cache (**tile**)
- reduce conflict misses in cache and simplify addressing (**datacopy**)
- exploit reuse in registers (**unroll-and-jam**)
- exploit SIMD (in the Opteron SSE) (**permute**, **unroll**)
- reduce loop overheads (**unroll**)

ACACES 2011, L2: Tuning code with CHiLL
5. Transformation for larger matrix sizes: Tiling

- Tiling reorders loop nests to bring iterations that reuse data closer together.
- Used to match data footprint to limited-capacity storage (today).
- Also used to divide a computation into parallel threads (Thursday’s parallel code generation).

ACACES 2011, L2: Tuning code with CHiLL
5. Tile Loops to Reduce Data Footprint in Subloop and Exploit Locality

Source code for mxm.c

loop level 1: for(i=0; i<128; i++)
loop level 2: for(j=0; j<128; j++)
loop level 3: for(k=0; k<128; k++)
statement 0: c[i][j]+=a[i][k]*b[k][j];

Resulting code:
for(kk=0; kk <=64; kk+=64)
  for(ii=0; ii<=112; ii+=16)
    for(i=ii; i<=ii+15; i++)
      for(j=0; j<128; j++)
        for(k=kk; k<=kk+63; k++)
          c[i][j]+=a[i][k]*b[k][j];

CHiLL script

source: mxm.c
procedure: 0
loop: 0
permute([1,2,3])
tile(0,1,16)
tile(0,4,64)
5. DataCopy

- Datacopy creates a temporary to be used in a subloop as a substitute for a variable
  - Uses polyhedral scanning to compute footprint of data in subloop
  - Copies variable into temporary in a loop that it creates preceding where the variable is accessed
  - Replaces variable accesses with accesses to temporary
  - May write back values

- Key Uses:
  - Explicit data staging for complex memory hierarchies and software-controlled storage (GPU discussion on Thursday)
  - Eliminate conflict misses and reduce TLB misses by controlling/reducing data footprint (this example)
5. Use DataCopy to Reduce Conflict Misses in Cache

Source code for mxm.c

loop level 1: for(i=0; i<128; i++)
loop level 2: for(j=0; j<128; j++)
loop level 3: for(k=0; k<128; k++)
statement 0: c[i][j]+=a[i][k]*b[k][j];

Resulting code:
for(kk=0; kk <=64; kk+=64)
  for(ii=0; ii<=112; ii+=16) {
    for (i=ii; i<=ii+15; i++)
      for(k=kk; k<=kk+63; k++)
        _P1[i-ii][k-kk] = a[i][k];
    for(i=ii; i<=ii+15; i++)
      for(j=0; j<128; j++)
        for(k=kk; k<=kk+63; k++)
          c[i][j]+=_P1[i-ii][k-kk]*b[k][j];
  }

CHiLL script

source: mxm.c
procedure: 0
loop: 0
permute([1,2,3])
tile(0,1,16)
tile(0,4,64)
datacopy(0,3,a)
5. Optimizations for larger matrix sizes

**code variant I:**
Tile for two levels of cache  
Expose SSE instructions

permute([1,2,3])  
tile(0,2,Tj)  
tile(0,2,Ti)  
tile(0,5,Tk)  
/* a is transposed */  
datacopy(0,3,a,false,1)  
datacopy(0,4,b)  
unroll (0,4,Ui)  
unroll (0,5,Uj)

**code variant II:**
Tile for single level of cache  
Expose SSE instructions

permute([1,2,3])  
tile(0,1,Ti)  
tile(0,4,Tk)  
/* a is transposed */  
datacopy(0,2,a,false,1)  
unroll (0,3,Ui)  
unroll (0,4,Uj)

Ti, Tj, Tk, Ui, Uj are *unbound parameters*
5. Optimizations for larger matrix sizes: Why transpose a?

• By transposing a, matrices b and a can both have adjacent data in their computation, suitable for SSE instructions (warning: this example is in Fortran!)

• We did not do this for small matrices
  - The cost of transpose is prohibitive with modest gain
  - Aggressive unrolling and (implicit) statement reordering can expose data
5. Additional Optimization Parameters and Variants

- Additional parameters and variants
  - What is the right loop order? (variant)
  - Which loops to unroll? (treat no unrolling as parameter)
  - How much to unroll? (parameter)
  - Tile size for each loop (parameter)
  - Whether or not to perform datacopy (variant)
5. Original Code Variant Generation Algorithm

- **Key Insights:**
  - Target data structures to specific levels of the memory hierarchy based on reuse analysis
  - Compose code transformations and determine constraints

For each memory hierarchy level in (Register, L1, L2, ...), use models to:

1. Select the data structure $D$ which has maximum reuse from reuse analysis (if possible, one that has not been considered)
2. Permute the relevant loops and apply tiling (unroll-and-jam for registers) according to newly selected reuse dimension
3. Generate copy variant if copying is beneficial
4. Determine constraints based on $D$ and current memory hierarchy level characteristics, using register/cache/TLB footprint analysis
5. Mark $D$ as considered
5. Mapping Reuse to Memory Hierarchy Levels

- **Register**
  - Loop order: $I,J,K$
  - Unroll & Jam $I,J$
  - **Constraint on UI and UJ** based on register size

- **L1 cache**
  - Loop order: $JJ, KK, I, J, K$
  - Tile $J, K$
  - **Constraint on TJ and TK** based on L1 cache size

- **L2 cache**
  - Loop order: $KK, II, JJ, I, J, K$
  - Tile $I$
  - **Constraint on TK and TI** based on L2 cache size

```
do i=1,n, Ui       // Ui=2
  do j=1,n, Uj      // Uj=2
    do k=1,n
      c(i,j) += a(i,k) * b(k,j)
      c(i,j+1) += a(i,k) * b(k,j+1)
      c(i+1,j) += a(i+1,k) * b(k,j)
      c(i+1,j+1) += a(i+1,k) * b(k,j+1)
    do jj=1,N,Tj
      do kk=1,n,Tk
        do i=1,n, Ui      // Ui=2
          do j=jj, jj+Tk, Uj // Uj=2
            do k=kk,kk+Tk
              c(i,j) += a(i,k) * b(k,j)
              c(i,j+1) += a(i,k) * b(k,j+1)
              c(i+1,j) += a(i+1,k) * b(k,j)
              c(i+1,j+1) += a(i+1,k) * b(k,j+1)
        do ii=1,n, Ti
          do j=jj, jj+Tk, Uj // Uj=2
            do i=ii,ii+Ti, Ui // Ui=2
              do k=kk,kk+Tk
                c(i,j) += a(i,k) * b(k,j)
                c(i,j+1) += a(i,k) * b(k,j+1)
                c(i+1,j) += a(i+1,k) * b(k,j)
                c(i+1,j+1) += a(i+1,k) * b(k,j+1)
```

ACACES 2011, L2: Tuning code with CHiLL
5. Matrix Multiply: Comparison with ATLAS, vendor BLAS and native compiler

matrix multiply on SGI R10K

ACACES 2011, L2: Tuning code with CHiLL
5. Comparison of Search Cost (Matrix Multiply and Jacobi)

<table>
<thead>
<tr>
<th>Code</th>
<th>SGI R10K</th>
<th>Sun US-2e</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM (ATLAS)</td>
<td>35 min</td>
<td>14 min</td>
</tr>
<tr>
<td>MM (ECO)</td>
<td>8 min (60 pts)</td>
<td>6 min (44 pts)</td>
</tr>
<tr>
<td>Jacobi (ECO)</td>
<td>3 min (94 pts)</td>
<td>5 min (148 pts)</td>
</tr>
</tbody>
</table>
Summary of Lecture

- Tuning kernels with CHiLL recipes
- Used primitives today, and will use higher-level commands on Thursday
- Example tuning experiments on linear algebra kernels
- Intuition on when and why to use certain optimizations
The literature contains a very large body of work on loop transformations. Here are a couple comprehensive references.


References on CHiLL scripts and optimization experiments discussed today.

