Parameterized Tiling

Previously
- Code generation with Fourier-Motzkin elimination

Today
- Code generation for tiling
- Code generation with parameterized tile sizes

Logistics
- HW10 due Friday
- Final report for project due next Wednesday, 5/2/12
- Quiz 4 due next Friday, 5/4/12
- Poster session Thursday May 10 from 2-4pm

Tiling transformation (slide from Daegon Kim)

```
for i1 = 1 to 6
  for i2 = 1 to 6
    S1;
```
Tiling transformation (slide from Daegon Kim)

for $i_1 = 1$ to $6$
for $i_2 = 1$ to $6$
    $S1$;

Tile shape:
- hyper-rectangle

for $t_1 = 1$ to $6$ by $3$
for $t_2 = 1$ to $6$ by $3$
    for $i_1 = \max(1, t_1)$ to $\min(6, t_1 + 2)$
    for $i_2 = \max(1, t_2)$ to $\min(6, t_2 + 2)$
        $S1$;

Tile origins

Tiling transformation (slide from Daegon Kim)

for $i_1 = 1$ to $6$
for $i_2 = 1$ to $6$
    $S1$;

Tile shape:
- hyper-rectangle

for $t_1 = 1$ to $6$ by $3$
for $t_2 = 1$ to $6$ by $3$
    for $i_1 = \max(1, t_1)$ to $\min(6, t_1 + 2)$
    for $i_2 = \max(1, t_2)$ to $\min(6, t_2 + 2)$
        $S1$;

Tile-loops  Point-loops
Tiling

A non-unimodular transformation that ...
- groups iteration points into tiles that are executed atomically
- can improve spatial and temporal data locality
- can expose larger granularities of parallelism

Implementing tiling
- how can we specify tiling?
- when is tiling legal?
- how do we generate tiled code?

```
  do ii = 1, 6, by 2
  do jj = 1, 5, by 2
    do i = ii, ii+2-1
      do j = jj, min(jj+2-1,5)
        A(i,j) = ...
```

Specifying Tiling

Rectangular tiling
- tile size vector \((s_1, s_2, \ldots, s_d)\)
- tile offset, \((o_1, o_2, \ldots, o_d)\)

Possible Transformation Mappings
- creating a tile space
  \[
  \{[i,j] \rightarrow [ti, tj, i, j] \mid ti = \text{floor}((i - o_1)/s_1) \\
  \quad \land tj = \text{floor}((j - o_2)/s_2)\}\]
- keeping tile iterators in original iteration space
  \[
  \{[i,j] \rightarrow [ii, jj, i, j] \mid ii = s_1 \text{floor}((i - o_1)/s_1) + o_1 \\
  \quad \land jj = s_2 \text{floor}((j - o_2)/s_2) + o_2\}\]
**Legality of Tiling**

A legal rectangular tiling
- each tile executed atomically
- no dependence cycles between tiles
- Check legality by verifying that transformed data dependences are lexicographically positive

**Fully permutable loops**
- rectangular tiling is legal on fully permutable loops

**Code Generation for Tiling**

**Fixed-size Tiles**
- Omega library
- Cloog
- for rectangular space and tiles, straight-forward
  
  ```
  do ii = 1, 6, by 2 
  do jj = 1, 5, by 2 
  do i = ii, ii+2-1 
  do j = jj, min(jj+2-1,5) 
  A(i,j) = ...
  ```

**Parameterized tile sizes**
- Parameterized tiled loops for free, PLDI 2007
- HiTLOG - A Tiled Loop Generator that is part of AlphaZ

**Overview of decoupled approach**
- find polyhedron that may contain any tile origins
- generate code that traverses that polyhedron
- post process the code to start a tile origins and step by tile size
- generate loops over points in tile to stay within original iteration space and within tile
Adaptive Tiling

“By no longer requiring the effect of an optimization to persist indefinitely, we can allow executables adapt to changes in their usage and environment. [...] this view helps us to regain the original promise of software — that it is flexible and easy to change.”

Overcoming the challenges to feedback directed optimization
Michael Smith, Keynote at DYNAMO ‘00

Loop tiling is an important optimization

Goal: Generate adaptive tiled code
**Loop Tiling**

Tiling partitions the iterations into groups called tiles.

Tiling is widely used to:
- expose coarse grained parallelism
- exploit cache locality
- exploit ILP and register locality

**Fixed Tiled Loop Generation**

4x4 tiling

`for (iT = 0; iT < 12; iT += 4)`
`for (jT = 0; jT < 20; jT += 4)`
`for (i = iT; i < iT+4; i++)`
`for (j = jT; j < jT+4; j++)`
`s(i,j)`

`tile-loops` & `point-loops`
Parameterized Tiled Loop Generation

\[
\begin{align*}
\text{for } (i = 0; i < 12; i++) \\
\text{for } (j = 0; j < 20; j++) \\
S(i,j)
\end{align*}
\]

\[
\begin{align*}
\text{for } (iT = 0; iT < 12; iT += s_i) \\
\text{for } (jT = 0; jT < 20; jT += s_j) \\
\text{for } (i = iT; i < \min(iT + s_i, 12); i++) \\
\text{for } (j = jT; j < \min(jT + s_j, 20); j++) \\
S(i,j)
\end{align*}
\]

\[s_i \times s_j \text{ tiling}\]

loop bounds and step sizes have symbolic tile sizes

Generalizing to arbitrary polyhedra

\[
\begin{align*}
\text{for } (k = 1; k <= N_k; k++) \\
\text{for } (i = k+1; i <= k+N_i; i++) \\
S1(k,i);
\end{align*}
\]

\[kTLB = -Sk+2; \quad kTLB = [kTLB/Sk]*Sk;\]
\[\text{for}(kT = kTLB; kT <= N_k; kT += Sk)\]
\[iTLB = kT-Si+2; \quad iTLB = [iTLB/Si]*Si;\]
\[\text{for}(iT = iTLB; iT <= kT+Ni+Sk-1; iT += Si)\]
\[\text{for}(k = \max(kT,1); k<\min(kT+Sk-1,Nk); k++)\]
\[\text{for}(i = \max(iT,k+1); i<\min(iT+Si-1,k+Ni); i++)\]
\[S1(k,i);\]

parameterized \(s_i \times s_j\) tiling

parameterized tiled loops for arbitrary polyhedra have complicated bounds

Efficient generation of efficient parameterized tiled loops
Contributions

Parameterized tiled loop generation algorithm
– handles arbitrary polyhedral iteration spaces

Two new polyhedral sets called outset & inset
– reuse of loop generation tools (Omega/Cloog)

Parameterization for free
– study of cost of parameterization (code quality)
– comparison of code generation time

Open source tool

Tile Origins

(empty tile)
(empty tile origin)
(full tile)
(full tile origin)
(partial tile)
(partial tile origin)
Outset

Set of non-empty tile origins
(red & green dots)

Bounding box contains lots of empty
tile (blue) origins

Outset is a tight
approximation of the set
of non-empty tile origins

Outset Formal Definition

Original Iteration
Space

\[ P_{iter} = \{ \bar{x} \mid Q\bar{x} \geq (q + B\bar{p}) \} \]

Outset

\[ P_{out} = \{ \bar{x} \mid Q\bar{x} \geq (q + B\bar{p}) - Q^+ s^t \} \]

\[ Q^+_{ij} = \begin{cases} Q_{ij}, & \text{if } Q_{ij} \geq 0 \\ 0, & \text{if } Q_{ij} < 0 \end{cases} \]

Outset is a parameterized polyhedra

Note that the tile sizes are parameters

Can be constructed in linear time/space
Outset Properties

- Tight approximation of the set of non-empty tile origins
- Linear time/space construction
- Tile sizes are parameters
- Parameterized polyhedron
  - enables use of standard loop generation tools

Generating tile-loops with Outset

- Construct outset
- Generate loops that scan all points in the outset
- Skip non-tile origins
  - Shift lower bounds
  - Stride iterations
Outset based Tiled Loop Generation Method

Method outline

- Gen. tile-loops using outset
  - using CLOOG
- Post-process to add
  - lower bound shifts
  - strides (step sizes)
- Generate point-loops
  - our generator + CLOOG
- Insert point-loops inside tile-loops.

The TLoG Tool

Tiled Loop Generator

Implements parameterized tiling
- Outset based (parameterized / fixed /mixed)
- Classic method
- Inset based separation of full and partial tiles
- Internally uses CLOOG (can also use Omega)

Used (beta tested 😊) in graduate level course at Colorado State University
Evaluation

Goal is to compare
- generation efficiency
- quality of generated code

Benchmarks
- stencils, LUD, SSYRK, STRMM

Methods compared
- Outset and Bounding box for parameterized
- Outset and classic for fixed tile sizes

Cost of Parameterization (1)

Experiments run on Intel coreDuo @ 1.86 GHz with 2 MB L2 cache

Benchmark: Triangular matrix product from BLAS 3 - tetrahedral iteration space

Parameterization overhead is insignificant
**Cost of Parameterization (2)**

*Experiments run on Intel coreDuo @ 1.86 GHz with 2 MB L2 cache*

**Benchmark:** Symmetric rank k update from BLAS 3 - tetrahedral iteration space

Parameterization overhead is insignificant

Code Generation Time

*Tiled loop generation times in milliseconds*

<table>
<thead>
<tr>
<th>Fixed Classic</th>
<th>LUD</th>
<th>SSYRK</th>
<th>STRMM</th>
<th>3D Stencil</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Decom</td>
<td>55.2</td>
<td>51.0</td>
<td>50.4</td>
<td>45.0</td>
</tr>
<tr>
<td>Param Outset</td>
<td>52.0</td>
<td>53.8</td>
<td>52.1</td>
<td>54.1</td>
</tr>
<tr>
<td>Param Bbox*</td>
<td>53.5</td>
<td>53.2</td>
<td>51.2</td>
<td>54.0</td>
</tr>
</tbody>
</table>

Fixed classic method does not scale

Parameterized tiled code can be generated as fast as fixed tiled code

*Bounding box computation time not included*
Summary of Results

Two fundamental polyhedra: Outset and Inset

- Useful for generating a variety of tiled loops

Parameterization for free

Single unified technique

- fixed, parameterized or mixed tiled loop generation

Ideal for compilers

Open source tool available

Inset

(Exact) Set of full-tile origins
Linear time/space construction
Parameterized tile sizes
Higher dimensional polyhedron

standard tools can be used to scan
Separating Partial and Full Tiles

Full tiles have simpler loop bounds
  – increase applicability of optimizations

```
tile-loops:
  (enumerate tile origins)

if (tile origin is in inset)
  point-loops: tile-box-bounds
else
  point-loops: tile-box and IS bounds
```

Concepts

**Code generation for fixed tiling**
  – Using omega to generate tiled code
  – Tile loops
  – Point loops
  – Tile origin

**Code generation for parameterized tiling**
  – outset
  – inset
  – Generating code using the outset
Next Time

Lecture
- Answer questions about FM and Farkas
- Review of all material in class
- Review expectations for final report and poster

Schedule
- HW10 due Friday
- Final report for project due next Wednesday, 5/2/12
- Quiz 4 due next Friday, 5/4/12
- Poster session Thursday May 10 from 2-4pm
\[ [i, j] \rightarrow [ii, jj, i, j] \mid ii = s_1 \cdot \lfloor \frac{(i-1)}{s_1} \rfloor + o_1 \]
\[ \land jj = s_2 \cdot \lfloor \frac{(j-1)}{s_2} \rfloor + o_2 \]

Recall some identities:

\[ x = \lfloor mx \rfloor \quad \text{if} \quad x \leq m < x + 1 \]

\[ a \div b = c \quad \text{iff} \quad \exists r \quad c \cdot b + r = a \]
\[ \land 0 \leq r < b \]

\[ ii = s_1 \alpha + o_1, \quad \alpha = \lfloor \frac{(i-1)}{s_1} \rfloor + 1 \]

Since \( \beta \) must be \( \lfloor \frac{(i-1)}{s_1} \rfloor \), \( \alpha + 1 \)

\[ \beta = \lfloor \frac{(i-1)}{s_1} \rfloor \]

\[ \exists \gamma \quad \beta \cdot s_1 + \gamma = i - 1 \]
\[ 0 \leq \gamma < s_1 \]

\[ jj = s_2 \alpha + o_2, \quad \alpha \leq \frac{(j-1)}{s_2} \]

\[ \exists \gamma \quad jj = s_2 \alpha + o_2, \quad \alpha s_2 + \gamma = j - 0 \]
\[ 0 \leq \gamma < s_2 \]
\[ \mathbf{v}_{\text{iter}} = \delta \bar{z} \]  
\[ \sum \mathbf{z} \geq (\mathbf{g} + \mathbf{b}) \]  
\[ \text{dim}(\bar{z}) = d \]

Tile size vector  
\[ \bar{s} = (s_1, s_2, \ldots, s_d) \]

where  
\[ \bar{s}' = \delta - \bar{s} \]

\[ \text{tile}(x) = \{ \bar{z} \mid x \leq \bar{z} \leq x + \bar{s}' \} \]

points in tile w/tile origin \( x \)

True outset, polyhedron that includes all and only tile origins where at least one point in the iteration space is in the tile

\[ P_{\text{out}} = \{ x \mid (\text{tile}(x) \cap P_{\text{iter}}) \neq \emptyset \} \]

Good approximation

\[ \hat{P}_{\text{out}} = \{ \bar{x} \mid \mathbf{Q} \bar{x} \geq (\mathbf{g} + \mathbf{b}) - \mathbf{Q}^+ \delta \} \]

\[ \mathbf{Q}^+ = \{ Q_{ij} \mid Q_{ij} \geq 0 \} \]

\[ Q_{ij} = 0 \quad \text{if} Q_{ij} < 0 \]
Main idea:
- Shift all lower bounds along their normal.
- Normal that points out of iteration space.
- Shift by largest negative projection of file vectors on normal.