R-Stream: A Parametric High Level Compiler

Eric Schweitz, Richard Lethin, Allen Leung, Benoit Meister
Reservoir Labs, Inc.

This work was sponsored by DARPA IPTO in the Polymorphous Computing Architectures program with Dr. William Harrod as Program Manager; contract number F30602-03-C-0033.
Outline

• Introduction
  – Morphware Forum and PCA
  – Applications
  – Challenges

• R-Stream: High-Level Design

• Polyhedral Model

• Preliminary Results on IBM Cell

• Conclusion
Morphware: Phased Compilation

C program
for (i = 0; i < N; i++)
for (j = 0; j < M; j++)
A[i][j] = f(B[i], C[j]);

PCA Machine Model
R-Stream High-Level Compiler
Morphware MSI
PCA SVM
Smart Mem LLC
MONARCH LLC
TRIPS LLC
RAW LLC
Morphware Stable Interface

- **MSI exposes an open layer between compile stages**
  - defines a computational model
  - provides a productivity layer
- **Productivity layer**
  - enables separate development of HLCs and LLCs
  - eases expansion to new targets
  - enables development of implementation libraries for use with LLCs
Exploiting Locality and Parallelism

- High performance at low(er) power – FLOPS/W
Applications: GMTI

Doppler Filtering
DFT 8 Mflop

STAP weights
LQQR 60 Kflop

Space-time
Adaptive Processing
Matrix Multiply 6 Mflop

Target Detection
CFAR 3D Grouping
300 Kflop

Target Parameter
Estimation MLE, Spline Interpolation
85 Kflop
Compiler Challenges

• Automatic management of small on-chip scratchpad memories

• High performance requires locality of reference

• Exploiting parallelism from the source code
  – Traditional approach: exploit the parallelism in loops
    • good source of data-parallelism
  – Large body of research to draw upon
    • including the polyhedral model; still, implementation lags research

• Locality and parallelism
  – a.k.a., Stream Processing
  – Pipelining data between different tasks across the PEs on chip
R-Stream: High-Level Design
R-Stream Design Goals

- **Static mapping**
- **Optimizations:**
  - Parallelism extraction
  - Loop transformations
  - Locality optimizations
  - Data layout transformations
  - DMA generation
  - Communication optimizations
  - Data distribution and processor mapping
- **Combining optimizations**
- **Flexible platform for experimentation**
High-Level Compiler

R-Stream 3.0

Polyhedral Mapper

ISO C Front End

Compiler Infrastructure

Raising

Lowering

Code Gen/Back End

Low-Level Compilers

SVM

...
Compiler Infrastructure: Raising

- Very much like a traditional compiler in structure
- Goal is different: want to raise abstraction for mapping

ISO C (EDG) → IL (JNI) → JIL → IR → IR → SSA IR → Abstraction Recovery → GDG

- polyhedral mapper
- structured loops
- array objects
- affine expressions
- memory disambiguation

- SSA
- Conditional Constant Propagation (CCP)
- Algebraic Simplification
- Dead Code Elimination (DCE)
- Operator Strength Reduction (OSR)
Polyhedral Mapper

- **Affine Partitioning**
  - finding the parallelism

- **Loop Fusion**
  - coarse grouping of statements

- **Tiling**
  - refining statement groups into tasks

- **Placement**
  - scheduling tiles to physical processors

- **Local Memory Compaction**
  - reducing memory footprint

- **Communication Generation**
  - inter-task communications

- **DMA Optimization**
  - create DMA operations
  - insert barriers, etc.

- **Synchronization Generation**
  - generate new code from polyhedral abstraction

- **Polyhedral Scanning**
  - insert barriers, etc.
Compiler Infrastructure: Lowering

- Again, similar to traditional compiler in structure
- Goal: lower the mapped code for output to the LLC

Diagram:
- GDG
- Target API
- IR normalization
- Scalar optimizations
- de-SSA
- Syntactic sugaring
- Output restructured C/SVM

- Per target machine information
- Making output more readable
The Polyhedral Model
Polyhedral Model

- semantics preserving optimizations
- affine transformations
- dependence preserving optimizations

- classical
- basic polyhedral model
Why the Polyhedral Model?

• A natural representation for static control programs

• All our optimizations can live in the same mathematical space.
  – Make it easier to combine optimizations

• More powerful modeling and computation techniques

• Parametric analysis

• Clean mathematical model to reason about loops

• Extensible
Classical Loop Transformations

Abstract syntax tree

```
for(j=0; j<N; j++)
  for(i=0; i<N; i++)
    S(j,i);
```

apply transformation (tiling)

```
for(i=0; i<N/M; i++)
  for(j=0; j<N/M; j++)
    for(ii=0; ii<M; ii++)
      for(jj=0; jj<M; jj++)
        S(j+jj, i+ii);
```

Often limited to:
- Coarse dependence summary: e.g., direction and distance vectors
- Single statement transformations
- Perfectly nested loops
- Unimodular transformations
- Specific ordering of phases

Program representation tied to syntax
Often divided into phases:
1. modeling
2. analysis
3. code generation
# Polyhedral Representation in a Nutshell

```plaintext
for (i=2; i<=M; i++) {
    for (j=0; j<=N; j+=2)
        A[i,N-j] = C[i-2,4*i+j/2];
    for (j=i; j<=N; j++)
        B[i,N-j] = A[i,j+1];
}
```

**Variables and access functions**

- **B**
  - \[
    \begin{bmatrix}
      1 & 0 & 0 & 0 & 0 \\
      0 & -1 & 0 & 1 & 0 \\
      0 & 0 & 0 & 0 & 1 \\
    \end{bmatrix}
  \]

- **A**
  - \[
    \begin{bmatrix}
      1 & 0 & 0 & 0 & 0 \\
      0 & 1 & 0 & 0 & 1 \\
      0 & 0 & 0 & 1 & 1 \\
    \end{bmatrix}
  \]

**Iteration domains as polyhedra**

\[ \{(i, j) \mid 2 \leq i \leq M, i \leq j \leq N \} \]

**Affine schedules** determine the execution order

\[
\begin{bmatrix}
  0 & 0 & 0 & 0 & 0 \\
  1 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 1 & 1 \\
  0 & 1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

**Dependence relations** tie these components together

---

**reservoir labs°**

**HPEC**

September 20, 2006
Transformations in the Polyhedral Model

- Optimizations are mathematical transformations
- Can use exact dependence
- Find schedules for multiple statements
- Not limited perfectly nested loops
- Not limited to unimodular transformations
- Not limited to linear-algebraic techniques
- Stay within a single mathematical representation between optimizations
- Compositional
- Parametric
- Loop synthesis at the very end to generate code

Abstract syntax tree

```
for(j=0; j<N; j++)
    for(i=0; i<N; i++)
        S(i,j);
```

```
for(i=4-4*N; i<=0; i++)
    for(j=max((1-i-N)/3, 0);
        j<=min((-i/3), N-1);  j++)
        S(j,-i-3*j);
```
### Subsumes Classic Loop Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Code</th>
<th>Unimodular Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Permutation</strong></td>
<td>for(i=0; i&lt;N; i++) for(j=0; j&lt;N; j++) s(i,j);</td>
<td>( \theta(i, j) = \begin{bmatrix} 0 &amp; 1 \ 1 &amp; 0 \end{bmatrix} )</td>
</tr>
<tr>
<td><strong>Reversal</strong></td>
<td>for(i=N-1; i&gt;=0; i--) for(j=0; j&lt;N; j++) s(j,i);</td>
<td>( \theta(i, j) = \begin{bmatrix} -1 &amp; 0 \ 0 &amp; 1 \end{bmatrix} )</td>
</tr>
<tr>
<td><strong>Skewing</strong></td>
<td>for(i=0; i&lt;N; i++) for(j=( \alpha \cdot i ); j&lt;N+( \alpha \cdot i ); j++) s(i,j-( \alpha \cdot i ));</td>
<td>( \theta(i, j) = \begin{bmatrix} 1 &amp; 0 \ \alpha &amp; 1 \end{bmatrix} )</td>
</tr>
<tr>
<td><strong>Scaling</strong></td>
<td>for(i=0; i&lt;( \alpha \cdot N ); i+=( \alpha )) for(j=0; j&lt;N; j++) s(i/( \alpha ),j);</td>
<td>( \theta(i, j) = \begin{bmatrix} \alpha &amp; 0 \ 0 &amp; 1 \end{bmatrix} )</td>
</tr>
</tbody>
</table>
Loop Transformations as Scheduling

 iteration space of a statement $S(i,j)$

\[ \theta : \mathbb{Z}^2 \rightarrow \mathbb{Z}^2 \]

Schedule $\theta$ maps iterations to **multi-dimensional** time

A feasible schedule *must* preserve dependencies

Loop transformations/synthesis mean generating code to execution iterations of a loop in the **lexicographical** order of time
Polyhedral Challenges

• **Algorithms for solving the problems**
  - may be very expensive
  - may not scale (can be super-exponential in time and space)
  - may not exist (specific problem not studied in the literature)

• **Power is a double-edged sword**
  - can find “all” parallelism
  - can find infinite families of legal schedules
  - must know what parallelism is best suited for target architecture

• **Application of transformations is easy**
  - knowing when, where and why (or why not) to apply them can be much more difficult to decide
  - must decide based on the target architecture’s features and limitations
Specific Reservoir Innovations

- **Parallelism extraction**
  - extensions to work of Lim and Lam, and others
  - extensions deal with locality and communication minimization

- **Local memory compaction**
  - rearranges data layout in local memory to improve memory usage
  - extensions to the algorithms of Schreiber and Cronquist

- **DMA optimization**
  - compute min-cost sets of DMA transfers
  - extension of algorithms for generating efficient message passing by Paek, et.al.

- **Parametric tiling**
  - generality: operates on collections of imperfect loop nests
  - find a best tiling, defined in terms of polynomial objective functions and constraints, over a space of possible tile sizes
  - considers re-use, memory footprint, etc.
R-Stream on Cell
Kernel Codes

```c
for (int i = 0; i < N; i++) {
}

for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        C[i][j] = 0;
        for (int k = 0; j < N; j++) {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
```

N=8*1024*1024

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        C[i][j] = 0;
        for (int k = 0; j < N; j++) {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
```

N=1024

Single precision floating point only
Vector Sum of Squares Results

```c
float (* restrict A_local) __attribute__((aligned(128))) = ...;
float (* restrict B_local) __attribute__((aligned(128))) = ...;
float (* restrict C_local) __attribute__((aligned(128))) = ...;
for (int i = 0; i < N; i++) {
    C_local[i] = A_local[i] * A_local[i] + B_local[i] * B_local[i];
}
```

<table>
<thead>
<tr>
<th>Trials</th>
<th>SIMDized</th>
<th>Pipelined</th>
<th>N</th>
<th>Time</th>
<th>GFlops/SPU/s</th>
<th>Bandwith (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>no</td>
<td>no</td>
<td>2048</td>
<td>9.25s</td>
<td>0.32</td>
<td>10.38</td>
</tr>
<tr>
<td>1024</td>
<td>no</td>
<td>yes</td>
<td>2048</td>
<td>5.62s</td>
<td>0.53</td>
<td>17.08</td>
</tr>
<tr>
<td>1024</td>
<td>yes</td>
<td>no</td>
<td>2048</td>
<td>5.29s</td>
<td>0.57</td>
<td>18.15</td>
</tr>
<tr>
<td>1024</td>
<td>yes</td>
<td>yes</td>
<td>2048</td>
<td>4.83s</td>
<td>0.62</td>
<td>19.88</td>
</tr>
<tr>
<td>4096</td>
<td>yes</td>
<td>yes</td>
<td>2048</td>
<td>18.21s</td>
<td>0.66</td>
<td>21.09</td>
</tr>
</tbody>
</table>

Memory bandwidth: 25.6GB/s peak, ~21GB/s sustainable
Matrix Multiply Results

- Not bandwidth bound; plenty of parallelism
- **Key component to performance:**
  - excellent SIMDization on the SPUs
- **Working on closing the gap between the HLC and LLC**
  - working with IBM; new version of their compiler “in the mail”
  - anticipate results will achieve closer to 50% of peak

<table>
<thead>
<tr>
<th>Trials</th>
<th>SIMDimized</th>
<th>Pipelined</th>
<th>Time</th>
<th>GFlops/SPU/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>no</td>
<td>no</td>
<td>79.4</td>
<td>0.202</td>
</tr>
<tr>
<td>64</td>
<td>no</td>
<td>yes</td>
<td>78.5</td>
<td>0.204</td>
</tr>
<tr>
<td>64</td>
<td>yes</td>
<td>no</td>
<td>13.36</td>
<td>1.20</td>
</tr>
<tr>
<td>64</td>
<td>yes</td>
<td>yes</td>
<td>12.55</td>
<td>1.27</td>
</tr>
</tbody>
</table>

25.6 GFlops/SPU/s peak
Research Challenges

• **Some architectures just now realizing silicon**
  – Much more experimentation needed
  – What optimizations are needed? Which work best?
• **Scalability: how big a problem can we map?**
• **What are the implications for parallel programming languages?**
• **How can we guide the programmer constructively?**
• **More work on optimization for the memory hierarchy**
• **How much performance do we sacrifice with layered compilation?**
  – Need strategies to ameliorate this potential gap
Conclusion

- Investigation and experimentation with phased compilation
  - PCA, Morphware Forum, SVM
- **Compiler that can automatically**
  - find parallelism
  - manage small scratchpad memories
  - generate communications
- **Advancement of polyhedral model**
  - improving published algorithms
  - new algorithms and optimizations
  - mapper uses polyhedral algorithms upon polyhedral representation
- **Some initial results of applying polyhedral mapper on Cell**
  - working on closing the gaps between HLC and LLC
- **Still more work to be done...**