Polyhedral Scheduling in the R-Stream Compiler

N. Vasilache, B. Meister, M. Baskaran, R. Lethin
Outline

R-Stream Overview

Balancing Parallelism and Memory

Joint Vectorization and Data Layout Formulations

Results

Conclusions
Benefits of Automatic Parallelization

Optimizations automatically achieved

• Programmer writes at very high level
• Instead of hand coding

Ability to quickly generate code (with these optimizations)

• Substantial coding effort if done manually

New optimizations targeted at future architectures

• Parallelism – locality – other tradeoffs (energy)
• Explicit communication management
• Deep hierarchy (on-going)

Ability to automatically generate various code variants with tunable parameters
Enabling technology is polyhedral abstraction

Uniform Recurrence Equations [Karp et al. 1970]

Loop Transformations and Parallelization [1970-]
- Many: Lamport, Allen/Kennedy, Banerjee, Irigoin, Wolfe/Lam, Pugh, Pingali, etc.
- Vectorization, SMP, locality optimizations
- Dependence summary: direction/distance vectors
- Unimodular transformations
- Mostly linear-algebraic

Systolic Array Mapping: Rajopadhye

Polyhedral Model [1980-]
- Many: Feautrier, Darte, Vivien, Wilde, Rajopadhye, etc.
- Exact dependence analysis: Feautrier
- General affine transformations
- Loop synthesis via polyhedral scanning: Quillere, Bastoul
- New scheduling techniques based on polyhedral representations (Bondhugula)
Polyhedral model – challenges in building a compiler

Mathematical abstraction is not trivial

Scalability of optimizations / representation / code generation

Traditionally confined to dependence preserving transformations

Code can be radically transformed – outputs can look wildly different

Modeling indirections, pointers, non-affine code.

Some of these challenges are solved + other on-going ideas
R-Stream model: polyhedra model

\[ n = f(); \]
\[
\text{for (i=5; i<= n; i+=2) { }
A[i][i] = A[i][i]/B[i];
\text{for (j=0; j<=i; j++) { }
if (j<=10) {
    \ldots A[i+2j+n][i+3]\ldots
}
}\]

\[ \{i, j \in \mathbb{Z}^2 | \exists k \in \mathbb{Z}, 5 \leq i \leq n; 0 \leq j \leq i; j \leq i; i = 2k + 1 \} \]

\[
\begin{bmatrix}
A_0 \\
A_1 \\
1
\end{bmatrix} =
\begin{bmatrix}
1 & 2 & 1 & 0 \\
1 & 0 & 0 & 3 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
i \\
j \\
n
\end{bmatrix}
\]

Affine and non-affine transformations
Order and place of operations and data

Loop code represented (exactly or conservatively) with polyhedrons
→ High-level, mathematical view of a mapping
→ But targets concrete properties: parallelism, locality, memory footprint
R-Stream blueprint

Machine Model → Polyhedral Mapper

Polyhedral Mapper → Raising → Scalar Representation

Polyhedral Mapper → Lowering →Scalar Representation

Scalar Representation → Pretty Printer

EDG C Front End → Scalar Representation
Driving the mapping: the machine model

Target machine characteristics that have an influence on how the mapping should be done

- Local memory / cache sizes
- Communication facilities: DMA, cache(s)
- Synchronization capabilities
- Symmetrical or not
- SIMD width
- Bandwidths

Currently: two-level model (Host and Accelerators)
XML schema and graphical rendering
Machine model example: multi-Tesla

Host

1 thread per GPU

OpenMP morph

XML file

CUDA morph
Mapping process

1- Scheduling:
Parallelism, locality, tilability

2- Task formation:
- Coarse-grain atomic tasks
- Master/slave side operations

3- Placement:
Assign tasks to blocks/threads

- Local / global data layout optimization
- Multi-buffering (explicitly managed)
- Synchronization (barriers)
- Bulk communications
- Thread generation -> master/slave
- CUDA-specific optimizations
Mapper flow

Array expansion
Affine scheduling
Task formation and placement
Memory promotion
Array contraction
Communication generation + Optimization
Multi-buffering
Bulk communication/DMA generation
Synchronization generation
Register tiling
Persistence
Thread generation
Code generation

Significant reuse of modules across targets

Cell
SMP
Tilera
GPU
CSX
RC100

Mapped code
Inside the polyhedral mapper

- GDG representation
- Tactics Module
- Parallelization
  - Locality
  - Optimization
- Tiling
- Placement
- Comm. Generation
- Memory Promotion
- Sync Generation
- Layout Optimization
- Polyhedral Scanning
- Jolylib, …
Inside the polyhedral mapper

Optimization modules engineered to expose “knobs” that could be used by auto-tuner
## Loop transformations (URUK-based representation)

### permutation

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

### reversal

```c
for(i=N-1; i>=0; i--)
  for(j=0; j<N; j++)
    S(j, i);
```

### skewing

```c
for(i=0; i<N; i++)
  for(j=α*i; j<N+α*i; j++)
    S(i, j-α*i);
```

### scaling

```c
for(i=0; i<α*N; i+=α)
  for(j=0; j<N; j++)
    S(i/α, j);
```

### Unimodular

#### permutation

\[
\theta(i, j) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}
\]

#### reversal

\[
\theta(i, j) = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}
\]

#### skewing

\[
\theta(i, j) = \begin{bmatrix} 1 & 0 \\ \alpha & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}
\]

#### scaling

\[
\theta(i, j) = \begin{bmatrix} \alpha & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}
\]
Loop fusion and distribution (URUK-based representation)

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        S1(i, j);

fusion

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        S2(i, j);

distribution

\[
\theta_1(i, j) = \begin{bmatrix}
0 & 0 & 0 \\
1 & 0 & 0 \\
0 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\]

fusion

\[
\theta_1(i, j) = \begin{bmatrix}
0 & 0 & 0 \\
1 & 0 & 0 \\
0 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\]

distribution

\[
\theta_2(i, j) = \begin{bmatrix}
0 & 0 & 0 \\
1 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\]

fusion

\[
\theta_2(i, j) = \begin{bmatrix}
0 & 0 & 0 \\
1 & 0 & 0 \\
0 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]
Loop transformations as scheduling

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

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
Outline

R-Stream Overview

Balancing Parallelism and Memory

Joint Vectorization and Data Layout Formulations

Results

Conclusions
R-Stream: base affine scheduling and fusion

Generalization of Bondhugula's breakthrough algorithm in a new unified formulation
Model based on an *objective function* with several *cost coefficients*:
- slowdown in execution if a loop $p$ is executed sequentially rather than in parallel
- cost in performance if two loops $p$ and $q$ remain unfused rather than fused

\[
\text{minimize} \left( \sum_{l \in \text{loops}} w_l p_l + \sum_{e \in \text{loop edges}} u_e f_e \right)
\]

slowdown in sequential execution  
cost of unfusing two loops

These two cost coefficients address parallelism and locality in a *unified and unbiased manner* (as opposed to traditional compilers).

Fine-grained parallelism, such as SIMD, can also be modeled using similar formulation.
Balancing parallelism quality and memory usage

Enabling technologies:
- Exact dependence analysis and conservative approximations
- Violated dependence analysis
- Ability to reason about temporarily incorrect programs
- Automatic correction of loop transformations
- Polyhedral schedulers

Key ideas:
- Memory budget, autotunable
- Schedule aggressively (and wrongly)
- Correct by expansion (and index-set splitting)
- Need to support tiling (most important program transformation ever)
Algorithm – High-level ideas

Iterative fixed point algorithm: the problem is non-linear
Precisely pinpoint the sources of error (VDA supports tiling)
Expand to correct
If memory budget is exceeded, save the reason why
While there exist errors:
  • Schedule using blackbox scheduler
  • Plug-in saved dependences to constrain the scheduler
  • Fixed-point is reached

Details in the paper
/* Optimization with BLAS */
for loop {
  ...
  BLAS call 1
  Retrieve data Z from disk
  ...
  BLAS call 2
  Store data Z back to disk
  Retrieve data Z from disk !!!
  ...
  BLAS call n
  ...
}

/* Global Optimization*/
doall loop {
  ...
  for loop {
    ...
    [read from Z]
    ...
    [write to Z]
    ...
    [read from Z]
  }
  ...
}

→ Global optimization can expose better parallelism and locality
Parallelism/locality/memory tradeoff example

/*
* Original code:
* Simplified CSLC-LMS
*/
for (k=0; k<400; k++) {
    for (i=0; i<3997; i++) {
        z[i]=0;
        for (j=0; j<4000; j++)
            z[i] = z[i] + B[i][j]*x[k][j];
    }
    for (i=0; i<3997; i++)
        w[i] = w[i] + z[i];
}

→ 2 levels of parallelism, but poor data reuse (on array z_e)

Max. parallelism (no fusion)

Maximum distribution destroys locality

doall (i=0; i<400; i++)
doall (j=0; j<3997; j++)
z_e[j][i]=0
doall (i=0; i<400; i++)
doall (j=0; j<3997; j++)
    for (k=0; k<4000; k++)
        z_e[j][i] = z_e[j][i] + B[j][k]*x[i][k];
doall (i=0; i<3997; i++)
doall (j=0; j<400; j++)
w[i] = w[i] + z_e[i][j];
doall (i=0; i<3997; i++)
z[i] = z_e[i][399];

Array z gets expanded, to introduce another level of parallelism
Parallelism/locality/memory tradeoff example (cont.)

// Original code:
* Simplified CSLC-LMS

for (k=0; k<400; k++) {
  for (i=0; i<3997; i++) {
    z[i]=0;
    for (j=0; j<4000; j++)
      z[i]= z[i]+B[i][j]*x[j][k];
  }
  for (i=0; i<3997; i++)
    w[i]=w[i]+z[i];
}

/*
 * doall (i=0; i<3997; i++)
 * for (j=0; j<4000; j++) {
 *   for (k=0; k<4000; k++)
 *     z[i]=z[i]+B[i][k]*x[j][k];
 *   w[i]=w[i]+z[i];
 * }

Very good data reuse (on array z), but only 1 level of parallelism

Max. fusion

Aggressive loop fusion destroys parallelism (i.e., only 1 degree of parallelism)
/*
 * Original code:
 * Simplified CSLC-LMS
 */
for (k=0; k<400; k++) {
    for (i=0; i<3997; i++) {
        z[i]=0;
        for (j=0; j<4000; j++)
            z[i]= z[i]+B[i][j]*x[k][j];
    }
    for (i=0; i<3997; i++)
        w[i]=w[i]+z[i];
}

/*
 * Doall
 * doall (i=0; i<3997; i++) {
 *     doall (j=0; j<400; j++) {
 *         z_e[i][j]=0;
 *         for (k=0; k<4000; k++)
 *             z_e[i][j]= z_e[i][j]+B[i][k]*x[j][k];
 *     }
 *     for (j=0; j<400; j++)
 *         w[i]=w[i]+z_e[i][j];
 * }
 * doall (i=0; i<3997; i++)
 *     z[i]=z_e[i][399];
 */

Partial fusion doesn't decrease parallelism

2 levels of parallelism with good data reuse (on array z_e)
Interesting facts

Example is a very simplified 2-D from original 4-D problem
Parallelism / locality tradeoff is obtained by changing the cost model

- Coefficients that can be learnt, across programs, across architectures
- Multi-objective linear functions

Base algorithm is enough for good performance:
- Memory budget = infinity
- Minimal amount of expansion for the specified parallelism
- Much smaller than full static expansion (which does not fit in 8GB space)

Other programs are not that friendly:
- Degrade parallelism found by scheduler (set doall bit to 0)
- This produces fewer violations and less expansion
Outline

R-Stream Overview

Balancing Parallelism and Memory

Joint Vectorization and Data Layout Formulations

Results

Conclusions
R-Stream: Joint affine scheduling and fusion

R-Stream uses a heuristic based on an *objective function* with several *cost coefficients*:

- slowdown in execution if a loop $p$ is executed sequentially rather than in parallel
- cost in performance if two loops $p$ and $q$ remain unfused rather than fused

\[
\text{minimize} \left( \sum_{l \in \text{loops}} w_l p_l + \sum_{e \in \text{loop edges}} u_e f_e \right)
\]

slowdown in sequential execution
cost of unfusing two loops

These two cost coefficients address parallelism and locality in a *unified and unbiased manner* (as opposed to traditional compilers).

Fine-grained parallelism, such as SIMD, can also be modeled using similar formulation.
Parallelism + locality + spatial locality + data layout

Hypothesis that auto-tuning should adjust these parameters

\[ \sum_{l \in \text{loops}} w_l p_l + \sum_{e \in \text{loop edges}} u_e f_e \]

- benefits of parallel execution
- benefits of improved locality

New algorithm balances contiguity to enhance coalescing for GPU and SIMDization
Model for scheduling trades 3 objectives jointly

- Fewer Global Memory Accesses
- More Locality
- More Parallelism
- Sufficient Occupancy

Loop Fission:
- More Locality + Loop Fusion
- Loop Fission

Loop Fusion:
- Memory Coalescing + successive thread contiguity
- Better Effective Bandwidth + successive thread contiguity

x Data-Layout Permutations
Additional degree of freedom
Joint affine scheduling and data layout

Enabling technologies:

- Generalization of Bondhugula’s algorithm (Leung)
- Contiguity of a reference (Bastoul)
  - Generalization to any schedule dimension
  - Generalization to any array dimension
- Convex space of all legal multi-dimensional transformations
  - Need to bound the “alpha” variables
- Ability to write $\text{Im } f \leq \text{Im } A$ in a linear formulation
  - Linear when $\text{Im } f = \text{Im } A$ (Leung)
  - Not exact linear when $\text{Im } f < \text{Im } A$
Algorithm – High-level ideas

Start from a multi-dimensional formulation
Incrementally add variables and constraints for more and more general formulations

- Contiguity for innermost schedule and array dimension
- Contiguity for any schedule and array dimension
- Contiguity constraints across all references in a statement
- Contiguity constraints for all statements “that have the same beta prefix”
- Mix with parallelism $\Rightarrow$ simd and vectorization
  - no guarantee on strides in this paper
- Data layout permutations open new doors

Need an invertible solution:

- No magic bullet, depth by depth, heuristic strategies (not permutations)
- On the whole multi-dimensional problem
Joint affine scheduling and data layout

\[ \forall \Delta = \{ T \to S \}, \ \forall k \in [1, \min(d^S, d^T)], \ \forall (i^T, i^S) \in \Delta : \]
\[ \left\{ \begin{array}{l}
\delta^\Delta_k \in \{0, 1\} \\
\sum_{l=1}^{\min(d^S, d^T)} \delta^\Delta_l = 1 \\
\Theta^T_k(i^T) - \Theta^S_k(i^S) \geq -\mathcal{N}_\infty \left( \sum_{i=1}^{\leq k-1} \delta^\Delta_i \right) \cdot (\tilde{n} + 1) + \delta^\Delta_k
\end{array} \right. \]

Figure 1: Convex space of all legal schedules.

+ \text{Im } f \leq \text{Im } a

+ Simd + data layout

\[ \forall S \in \mathcal{G}, \ \forall l \in [1, d^S] \]
\[ \mu - F^r_\tau \cdot \lambda + \mathcal{N}_\infty \cdot (1 - c^F_{r,d}) \geq 0 \]
\[ -\mu + F^r_\tau \cdot \lambda + \mathcal{N}_\infty \cdot (1 - c^F_{r,d}) \geq 0 \]

\[ \begin{array}{l}
\mathcal{K}^S_1 \cdot \Sigma^S_1 \leq \sum_{A \in S} p^S,A_l \\
\forall S \in \mathcal{G}, \ \forall l \in [1, d^S], \ \forall A \in S \\
\mathcal{K}^S_3 \cdot q^S,A_{l,r} \leq \sum_{F \text{ acc. } A} c^F_{l,r}
\end{array} \]
Inner contiguity, innermost array

→ (j, -i+k, i)

no contiguous solution in the positive quadrant
Outer vectorization innermost array dimension

```c
for (i=2; i<=1+N; i++) {
    for (j=2; j<=1+M; j++) {
        for (k=1; k<=L; k++) {
            A[i][j][k]=A[i][j-1][k+1]+A[i-1][j][k+1];
        }
    }
}
```

```
doall (i=5; i<=N+M+L+2; i++) {
    for (j=max(2, i-N-L-1); j<=min(M+1, i-3); j++) {
        for (k=max(2, i-j-L); k<=min(i-j-1, N+1); k++) {
            A[k][j][i-j-k]=A[k][j-1][i-j-k+1]+A[k-1][j][i-j-k+1];
        }
    }
}
```

\[ \rightarrow (i+j+k, j, k) \]
Outline

R-Stream Overview

Balancing Parallelism and Memory

Joint Vectorization and Data Layout Formulations

Results

Conclusions
Radar benchmarks (array expansion)

Beamforming algorithms:

- **MVDR–SER**: Minimum Variance Distortionless Response using Sequential Regression
- **CSLC–LMS**: Coherent Sidelobe Cancellation using Least Mean Square
- **CSLC–RLS**: Coherent Sidelobe Cancellation using Robust Least Square

Expressed in sequential ANSI C

400 radar iterations

Compute 3 radar sidelobes (for CSLC–LMS and CSLC–RLS)

The problem is algorithm selection: which of these 3 algorithms has the most parallelism.
MVDR-SER (outer-sequential (array expansion))

![Graph showing Gflops vs #Channels for different libraries: MKL, R-Stream(GCC), R-Stream(ICC), GCC, and ICC. The graph illustrates performance variation across different channel counts.]
CSLC-RLS (outer-sequential (array expansion))

- MKL
- R-Stream(GCC)
- R-Stream(ICC)
- GCC
- ICC

Gflops vs. #Channels

1K, 2K, 3K, 4K, 5K, 6K, 7K, 8K, 9K, 10K

Reservoir Labs
## Vectorization quality (statistical results)

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Num contiguous</th>
<th>Num simd</th>
<th>Simd depth</th>
<th>Num T/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoObj</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Identity</td>
<td>179</td>
<td>23</td>
<td>27</td>
<td>2</td>
</tr>
<tr>
<td>Permutations</td>
<td>673</td>
<td>75</td>
<td>119</td>
<td>2</td>
</tr>
<tr>
<td>Default</td>
<td>1637</td>
<td>386</td>
<td>586</td>
<td>2</td>
</tr>
<tr>
<td>OuterSimd</td>
<td>2891</td>
<td>348</td>
<td>418</td>
<td>2</td>
</tr>
<tr>
<td>Layout</td>
<td>2107</td>
<td>483</td>
<td>772</td>
<td>2</td>
</tr>
<tr>
<td>OuterSimd + Layout</td>
<td>6999</td>
<td>368</td>
<td>244</td>
<td>3</td>
</tr>
<tr>
<td>AS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

400 kernel benchmarks  
Includes some of the PolyBench  
Includes multiple larger "apps" from radar world  
Scalability limitations many possible formulation improvements  
Applied on whole problem when you would typically apply within a tile
Outline

R-Stream Overview

Balancing Parallelism and Memory

Joint Vectorization and Data Layout Formulations

Results

Conclusions
Conclusion

New formulations that now need to be tuned and scaled up

• UTVPI direction is interesting
  Further opportunities to integrate even more transformations into iterative, fixed-point algorithms

• Contraction, ISS

• But the problem is non-trivial because of placement, synchronizations and communications

• Global problem not yet understood well enough
  Algorithm selection exploration
  Autotuning at every level in the compiler: built but not yet exploited

• Modelization of energy constraints
  Will likely require folding in placement + privatization in scheduling somehow
Conclusion

Still lots of opportunities

- At low-level we compare auto-tuned MKL (human + tools) to fully auto-generated high-level C
- Soon able to model energy consumption

The next frontier is integration with data structures and ADTs

Research collaborations

- More tools to explore and ideas than people at Reservoir
Questions?