Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework

Louis-Noël Pouchet¹ Uday Bondhugula² Cédric Bastoul³ Albert Cohen³ J. Ramanujam⁴ P. Sadayappan¹

¹ The Ohio State University
² IBM T.J. Watson Research Center
³ ALCHEMY group, INRIA Saclay / University of Paris-Sud 11, France
⁴ Louisiana State University

November 17, 2010
IEEE 2010 Conference on Supercomputing
New Orleans, LA
Overview

Problem: How to improve program execution time?

- Focus on shared-memory computation
  - OpenMP parallelization
  - SIMD Vectorization
  - Efficient usage of the intra-node memory hierarchy

- Challenges to address:
  - Different machines require different compilation strategies
  - One-size-fits-all scheme hinders optimization opportunities

Question: how to restructure the code for performance?
Objectives for a Successful Optimization

During the program execution, interplay between the hardware resources:

- Thread-centric parallelism
- SIMD-centric parallelism
- Memory layout, inc. caches, prefetch units, buses, interconnects...

→ Tuning the trade-off between these is required

A loop optimizer must be able to transform the program for:

- Thread-level parallelism extraction
- Loop tiling, for data locality
- Vectorization

Our approach: form a tractable search space of possible loop transformations
Running Example

Original code

Example ($tmp = A.B, D = tmp.C$)

```c
for (i1 = 0; i1 < N; ++i1)
    for (j1 = 0; j1 < N; ++j1) {
        R: tmp[i1][j1] = 0;
        for (k1 = 0; k1 < N; ++k1)
            S: tmp[i1][j1] += A[i1][k1] * B[k1][j1];
    }
for (i2 = 0; i2 < N; ++i2)
    for (j2 = 0; j2 < N; ++j2) {
        T: D[i2][j2] = 0;
        for (k2 = 0; k2 < N; ++k2)
            U: D[i2][j2] += tmp[i2][k2] * C[k2][j2];
    }
```

{R, S} fused, {T, U} fused

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Max. fusion</th>
<th>Max. dist</th>
<th>Balanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4 \times$ Xeon 7450 / ICC 11</td>
<td>1×</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$4 \times$ Opteron 8380 / ICC 11</td>
<td>1×</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Running Example

Cost model: maximal fusion, minimal synchronization
[Bondhugula et al., PLDI’08]

Example \( (tmp = A.B, D = tmp.C) \)

\[
\begin{align*}
\text{parfor } (c0 = 0; c0 < N; c0++) & \{ \\
& \text{for } (c1 = 0; c1 < N; c1++) \{ \\
& \text{R: } tmp[c0][c1]=0; \\
& \text{T: } D[c0][c1]=0; \\
& \text{for } (c6 = 0; c6 < N; c6++) \\
& \text{S: } tmp[c0][c1] += A[c0][c6] * B[c6][c1]; \\
& \text{parfor } (c6 = 0; c6 <= c1; c6++) \\
& \text{U: } D[c0][c6] += tmp[c0][c1-c6] * C[c1-c6][c6]; \\
& \} \text{ \{R,S,T,U\} fused} \\
& \text{for } (c1 = N; c1 < 2*N - 1; c1++) \\
& \text{parfor } (c6 = c1-N+1; c6 < N; c6++) \\
& \text{U: } D[c0][c6] += tmp[c0][1-c6] * C[c1-c6][c6]; \\
\}\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Max. fusion</th>
<th>Max. dist</th>
<th>Balanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>4× Xeon 7450 / ICC 11</td>
<td>1×</td>
<td>2.4×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4× Opteron 8380 / ICC 11</td>
<td>1×</td>
<td>2.2×</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Running Example

Maximal distribution: best for Intel Xeon 7450
Poor data reuse, best vectorization

Example ($tmp = A.B, D = tmp.C$)

```
parfor (i1 = 0; i1 < N; ++i1)
    parfor (j1 = 0; j1 < N; ++j1)
        R: tmp[i1][j1] = 0;
    parfor (i1 = 0; i1 < N; ++i1)
        for (k1 = 0; k1 < N; ++k1)
            parfor (j1 = 0; j1 < N; ++j1)
                S: tmp[i1][j1] += A[i1][k1] * B[k1][j1];

{R} and {S} and {T} and {U} distributed
```

```
parfor (i2 = 0; i2 < N; ++i2)
    parfor (j2 = 0; j2 < N; ++j2)
        T: D[i2][j2] = 0;
    parfor (i2 = 0; i2 < N; ++i2)
        for (k2 = 0; k2 < N; ++k2)
            parfor (j2 = 0; j2 < N; ++j2)
                U: D[i2][j2] += tmp[i2][k2] * C[k2][j2];
```

<table>
<thead>
<tr>
<th>Original</th>
<th>Max. fusion</th>
<th>Max. dist</th>
<th>Balanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>4× Xeon 7450 / ICC 11</td>
<td>1×</td>
<td>2.4×</td>
<td>3.9×</td>
</tr>
<tr>
<td>4× Opteron 8380 / ICC 11</td>
<td>1×</td>
<td>2.2×</td>
<td>6.1×</td>
</tr>
</tbody>
</table>
Running Example

Balanced distribution/fusion: best for AMD Opteron 8380
Poor data reuse, best vectorization

Example \((tmp = A.B, D = tmp.C)\)

```
parfor (c1 = 0; c1 < N; c1++)
  parfor (c2 = 0; c2 < N; c2++)
  R: C[c1][c2] = 0;
parfor (c1 = 0; c1 < N; c1++)
  for (c3 = 0; c3 < N;c3++) {
    T: E[c1][c3] = 0;
    parfor (c2 = 0; c2 < N;c2++)
    S: C[c1][c2] += A[c1][c3] * B[c3][c2];
  }
  \{S,T\} fused, \{R\} and \{U\} distributed
parfor (c1 = 0; c1 < N; c1++)
  for (c3 = 0; c3 < N; c3++)
    parfor (c2 = 0; c2 < N; c2++)
    U: E[c1][c2] += C[c1][c3] * D[c3][c2];
```

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Max. fusion</th>
<th>Max. dist</th>
<th>Balanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>4× Xeon 7450 / ICC 11</td>
<td>1×</td>
<td>2.4×</td>
<td>3.9×</td>
<td>3.1×</td>
</tr>
<tr>
<td>4× Opteron 8380 / ICC 11</td>
<td>1×</td>
<td>2.2×</td>
<td>6.1×</td>
<td>8.3×</td>
</tr>
</tbody>
</table>
Running Example

Example ($tmp = A.B, D = tmp.C$)

```matlab
parfor (c1 = 0; c1 < N; c1++)
    parfor (c2 = 0; c2 < N; c2++)
        R: C[c1][c2] = 0;
        parfor (c1 = 0; c1 < N; c1++)
            for (c3 = 0; c3 < N; c3++) {
                T: E[c1][c3] = 0;
            }
            parfor (c2 = 0; c2 < N; c2++)
        } {S,T} fused, {R} and {U} distributed
    parfor (c1 = 0; c1 < N; c1++)
        for (c3 = 0; c3 < N; c3++)
            parfor (c2 = 0; c2 < N; c2++)
        } U: E[c1][c2] += C[c1][c3] * D[c3][c2];
```

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Max. fusion</th>
<th>Max. dist</th>
<th>Balanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>4× Xeon 7450 / ICC 11</td>
<td>1×</td>
<td>2.4×</td>
<td>3.9×</td>
<td>3.1×</td>
</tr>
<tr>
<td>4× Opteron 8380 / ICC 11</td>
<td>1×</td>
<td>2.2×</td>
<td>6.1×</td>
<td>8.3×</td>
</tr>
</tbody>
</table>

The best **fusion/distribution choice** drives the quality of the optimization
Loop Structures

Possible grouping + ordering of statements

- $\{\{R\}, \{S\}, \{T\}, \{U\}\}; \{\{R\}, \{S\}, \{U\}, \{T\}\}; ...$
- $\{\{R, S\}, \{T\}, \{U\}\}; \{\{R\}, \{S\}, \{T, U\}\}; \{\{R\}, \{T, U\}, \{S\}\}; \{\{T, U\}, \{R\}, \{S\}\}; ...$
- $\{\{R, S, T\}, \{U\}\}; \{\{R\}, \{S, T, U\}\}; \{\{S\}, \{R, T, U\}\}; ...$
- $\{\{R, S, T, U\}\}$

Number of possibilities: $>> n!$ (number of total preorders)
Loop Structures

Removing non-semantics preserving ones

- \{\{R\}, \{S\}, \{T\}, \{U\}\}; \{\{R\}, \{S\}, \{U\}, \{T\}\}; ...
- \{\{R,S\}, \{T\}, \{U\}\}; \{\{R\}, \{S\}, \{T,U\}\}; \{\{R\}, \{T,U\}, \{S\}\}; \{\{T,U\}, \{R\}, \{S\}\}; ...
- \{\{R,S,T\}, \{U\}\}; \{\{R\}, \{S,T,U\}\}; \{\{S\}, \{R,T,U\}\}; ...
- \{\{R,S,T,U\}\}

Number of possibilities: 1 to 200 for our test suite
Loop Structures

For each partitioning, many possible loop structures

- \{R\}, \{S\}, \{T\}, \{U\}
- For S: \{i, j, k\}; \{i, k, j\}; \{k, i, j\}; \{k, j, i\}; ...
- However, only \{i, k, j\} has:
  - outer-parallel loop
  - inner-parallel loop
  - lowest striding access (efficient vectorization)
Possible Loop Structures for 2mm

- 4 statements, 75 possible partitionings
- 10 loops, up to 10! possible loop structures for a given partitioning

**Two steps:**
- Remove all partitionings which breaks the semantics: from 75 to 12
- Use static cost models to select the loop structure for a partitioning: from $d!$ to 1

- Final search space: **12 possibilities**
Workflow – Polyhedral Compiler

Original Source Code → Polyhedral source-to-source Compiler → Optimized Source Code → Vendor Compiler → Binary code

- C / C++ / Fortran
- PoCC / Pluto
- ROSE / PolyOpt
- (LLVM / Polly)
- (GCC / Graphite)
- ...

- C code w/
  - OpenMP
  - Vector

- Intel ICC
- GNU GCC
- ...

- Optimized binary

OSU / IBM / INRIA / LSU
Contributions and Overview of the Approach

- Empirical search on possible fusion/distribution schemes
- Each structure drives the success of other optimizations
  - Parallelization
  - Tiling
  - Vectorization
- Use static cost models to compute a complex loop transformation for a specific fusion/distribution scheme
- Iteratively test the different versions, retain the best
  - Best performing loop structure is found
Polyhedral Representation of Programs

Static Control Parts

- Loops have affine control only (over-approximation otherwise)
Polyhedral Representation of Programs

Static Control Parts
- Loops have affine control only (over-approximation otherwise)
- Iteration domain: represented as integer polyhedra

\begin{verbatim}
for (i=1; i<=n; ++i)
  for (j=1; j<=n; ++j)
    if (i<=n-j+2)
      . . . s[i] = ...
\end{verbatim}

\[
D_{S1} = \begin{bmatrix}
1 & 0 & 0 & -1 \\
-1 & 0 & 1 & 0 \\
0 & 1 & 0 & -1 \\
-1 & 0 & 1 & 0 \\
-1 & -1 & 1 & 2
\end{bmatrix}
\begin{bmatrix}
i \\
j \\
n \\
1
\end{bmatrix} \geq \vec{0}
\]
Polyhedral Representation of Programs

Static Control Parts

- Loops have affine control only (over-approximation otherwise)
- Iteration domain: represented as integer polyhedra
- Memory accesses: static references, represented as affine functions of $\vec{x}_S$ and $\vec{p}$

```plaintext
for (i=0; i<n; ++i) {
    s[i] = 0;
    for (j=0; j<n; ++j)
        s[i] = s[i] + a[i][j]*x[j];
}
```

$f_s(\vec{x}_S) = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{pmatrix} \vec{x}_S \\ n \\ 1 \end{pmatrix}$

$f_a(\vec{x}_S) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{pmatrix} \vec{x}_S \\ n \\ 1 \end{pmatrix}$

$f_x(\vec{x}_S) = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{pmatrix} \vec{x}_S \\ n \\ 1 \end{pmatrix}$
Polyhedral Representation of Programs

Static Control Parts

- Loops have affine control only (over-approximation otherwise)
- Iteration domain: represented as integer polyhedra
- Memory accesses: static references, represented as affine functions of $\vec{x}_S$ and $\vec{p}$
- Data dependence between S1 and S2: a subset of the Cartesian product of $D_{S1}$ and $D_{S2}$ (exact analysis)

```
for (i=1; i<=3; ++i) {
  s[i] = 0;
  for (j=1; j<=3; ++j)
    s[i] = s[i] + 1;
}
```

$D_{S1} \Delta D_{S2}$:

\[
\begin{bmatrix}
1 & -1 & 0 & 0 \\
1 & 0 & 0 & -1 \\
-1 & 0 & 0 & 3 \\
0 & 1 & 0 & -1 \\
0 & -1 & 0 & 3 \\
0 & 0 & 1 & -1 \\
0 & 0 & -1 & 3
\end{bmatrix}
\]

\[
\begin{pmatrix}
i_{S1} \\
i_{S2} \\
i_{S2}
\end{pmatrix}
= \begin{pmatrix} 0 \\ \geq 0 \end{pmatrix}
\]

S1 iterations

S2 iterations
Search Space of Loop Structures

- **Partition the set of statements into classes:**
  - This is deciding loop fusion / distribution
  - Statements in the same class will share at least one common loop in the target code
  - Classes are ordered, to reflect code motion

- **Locally on each partition, apply model-driven optimizations**

- **Leverage the polyhedral framework:**
  - Build the smallest yet most expressive space of possible partitionings
    [Pouchet et al., POPL’11]
  - Consider **semantics-preserving partitionings only**: orders of magnitude smaller space
Model-driven Optimizations: Tiling

Two steps: pre-transform to make tiling legal, then tile the loop nest

Tiling in our framework:

- Partition the computation into blocks
- Resulting blocks can be executed with sync-free or pipeline parallelism
- Seamless integration in the polyhedral framework (imperfectly nested loops, parametric tiling)

- Systematic application of the pre-transformation (Tiling Hyperplane method [Bondhugula et al., PLDI’08])
- We tile the transformed loop nest only if:
  - There is at least $O(N)$ reuse
  - the loop depth is $> 1$
Model-driven Optimizations: OpenMP parallelization

- Assume pre-transformation for tiling already done

- By definition, existing parallelism is brought on outer loops
  - Property of the Tiling Hyperplane
  - We drive the optimization to obtain this property on a specific subset of statements

- Simply mark outer parallel loops with `#pragma omp parallel for`
  - First parallel outer tile loop, if any
Model-driven Optimizations: Vectorization

Focus on additional loop transformations, not codegen-related

- Vectorization requires a sync-free parallel inner-most loop
  - Candidate parallel loops can be moved inward
  - Multiple choices!

- To be efficient, favor stride-1 access for the inner-loop
  - The loop iterator appears only in the last dimension of the array
  - Loop permutation changes the stride of memory accesses
  - Use a static cost model [Trifunovic et al., PACT'09]
# Summary of the Optimization Process

<table>
<thead>
<tr>
<th>description</th>
<th>#loops</th>
<th>#stmts</th>
<th>#refs</th>
<th>#deps</th>
<th>#part.</th>
<th>#valid</th>
<th>Variability</th>
<th>Pb. Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2mm</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>75</td>
<td>12</td>
<td>✓</td>
<td>1024x1024</td>
</tr>
<tr>
<td>3mm</td>
<td>9</td>
<td>6</td>
<td>12</td>
<td>19</td>
<td>4683</td>
<td>128</td>
<td>✓</td>
<td>1024x1024</td>
</tr>
<tr>
<td>adi</td>
<td>11</td>
<td>8</td>
<td>36</td>
<td>188</td>
<td>545835</td>
<td>1</td>
<td>1024x1024</td>
<td></td>
</tr>
<tr>
<td>atax</td>
<td>4</td>
<td>4</td>
<td>10</td>
<td>12</td>
<td>75</td>
<td>16</td>
<td>✓</td>
<td>8000x8000</td>
</tr>
<tr>
<td>biccg</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>10</td>
<td>75</td>
<td>26</td>
<td>✓</td>
<td>8000x8000</td>
</tr>
<tr>
<td>correl</td>
<td>5</td>
<td>6</td>
<td>12</td>
<td>14</td>
<td>4683</td>
<td>176</td>
<td>✓</td>
<td>500x500</td>
</tr>
<tr>
<td>covar</td>
<td>7</td>
<td>7</td>
<td>13</td>
<td>26</td>
<td>47293</td>
<td>96</td>
<td>✓</td>
<td>500x500</td>
</tr>
<tr>
<td>doitgen</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>13</td>
<td>4</td>
<td>128x128x128</td>
<td></td>
</tr>
<tr>
<td>gemm</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>1024x1024</td>
<td></td>
</tr>
<tr>
<td>gemver</td>
<td>7</td>
<td>4</td>
<td>19</td>
<td>13</td>
<td>75</td>
<td>8</td>
<td>✓</td>
<td>8000x8000</td>
</tr>
<tr>
<td>gesummv</td>
<td>2</td>
<td>5</td>
<td>15</td>
<td>17</td>
<td>541</td>
<td>44</td>
<td>✓</td>
<td>8000x8000</td>
</tr>
<tr>
<td>gramschmidt</td>
<td>6</td>
<td>7</td>
<td>17</td>
<td>34</td>
<td>47293</td>
<td>1</td>
<td>512x512</td>
<td></td>
</tr>
<tr>
<td>jacobi-2d</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>14</td>
<td>3</td>
<td>1</td>
<td>20x1024x1024</td>
<td></td>
</tr>
<tr>
<td>lu</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>1024x1024</td>
<td></td>
</tr>
<tr>
<td>ludcmp</td>
<td>9</td>
<td>15</td>
<td>40</td>
<td>188</td>
<td>10^{12}</td>
<td>20</td>
<td>✓</td>
<td>1024x1024</td>
</tr>
<tr>
<td>seidel</td>
<td>3</td>
<td>1</td>
<td>10</td>
<td>27</td>
<td>1</td>
<td>1</td>
<td>20x1024x1024</td>
<td></td>
</tr>
</tbody>
</table>

Table: Summary of the optimization process
Experimental Setup

We compare three schemes:

- **maxfuse**: static cost model for fusion (maximal fusion)

- **smartfuse**: static cost model for fusion (fuse only if data reuse)

- **Iterative**: iterative compilation, output the best result
Performance Results - Intel Xeon 7450 - ICC 11

Performance Improvement - Intel Xeon 7450 (24 threads)

Perf. Imp / ICC - fast - parallel

2mm 3mm adi atax bicg correl doitgen gemm gemver gesummv gramschmidt lu ludcmp seidel

OSU / IBM / INRIA / LSU
Performance Results - AMD Opteron 8380 - ICC 11

Performance Improvement - AMD Opteron 8380 (16 threads)

- pocc-maxfuse
- pocc-smartfuse
- iterative

Graph showing performance improvement for various benchmarks.
Performance Results - Intel Atom 330 - GCC 4.3

Performance Improvement - Intel Atom 230 (2 threads)
Assessment from Experimental Results

1. Empirical tuning required for 9 out of 16 benchmarks

2. Strong performance improvements: $2.5 \times - 3 \times$ on average

3. Portability achieved:
   - Automatically adapt to the program and target architecture
   - No assumption made about the target
   - Exhaustive search finds the optimal structure (1-176 variants)

4. Substantial improvements over state-of-the-art (up to $2 \times$)
Frameworks for Polyhedral Compilation

- IBM XL / Poly
- GCC / Graphite (now in mainstream 4.5)
- LLVM / Polly
- R-Stream (Reservoir Labs, Inc.)
- ROSE / Polyopt (DARPA PACE project)

- Numerous affine program fragments in computational applications
- **Our goal: drive programmers to write polyhedral-compliant programs!**
Conclusions

Take-home message:

⇒ Fusion / Distribution / Code motion highly program- and machine-specific

⇒ Minimum empirical tuning + polyhedral framework gives very good performance on several applications

⇒ Complete, end-to-end framework implemented and effectiveness demonstrated

Future work:

▶ Further pruning of the search space (additional static cost models)
▶ Statistical search techniques