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

Louis-Noël Pouchet[1] Uday Bondhugula[2] Cédric Bastoul[3] Albert Cohen[3]
J. Ramanujam[4] P. Sadayappan[1]

[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 ressources:

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

$\longrightarrow$ **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 ($tmp = A.B$, $D = tmp.C$)

```
  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

	Original	Max. fusion	Max. dist	Balanced
4× Xeon 7450 / ICC 11	$1\times$			
4× Opteron 8380 / ICC 11	$1\times$			

Running Example

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

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

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

	Original	Max. fusion	Max. dist	Balanced
4× Xeon 7450 / ICC 11	1×	2.4×		
4× Opteron 8380 / ICC 11	1×	2.2×		

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];
```

	Original	Max. fusion	Max. dist	Balanced
4× Xeon 7450 / ICC 11	1×	2.4×	3.9×	
4× Opteron 8380 / ICC 11	1×	2.2×	6.1×	

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];
```

	Original	Max. fusion	Max. dist	Balanced
4× Xeon 7450 / ICC 11	$1\times$	$2.4\times$	$3.9\times$	$3.1\times$
4× Opteron 8380 / ICC 11	$1\times$	$2.2\times$	$6.1\times$	$8.3\times$

Running Example

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];
```

	Original	Max. fusion	Max. dist	Balanced
4× Xeon 7450 / ICC 11	$1\times$	$2.4\times$	$3.9\times$	$3.1\times$
4× Opteron 8380 / ICC 11	$1\times$	$2.2\times$	$6.1\times$	$8.3\times$

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)

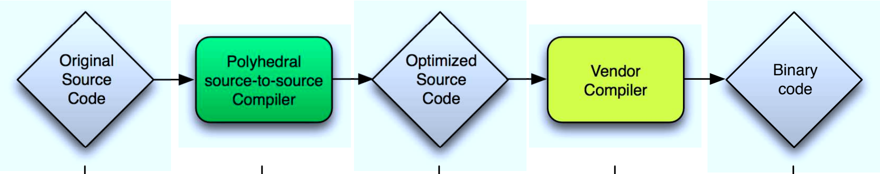
Possible Loop Structures for 2mm

- 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 possibilites**

Workflow – Polyhedral Compiler



C / C++ / Fortran

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

C code w/
- OpenMP
- Vector

- Intel ICC
- GNU GCC
- ...

Optimized binary

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)

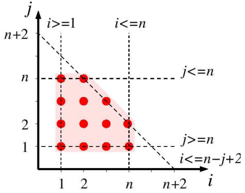
Polyhedral Representation of Programs

Static Control Parts

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

```
for (i=1; i<=n; ++i)
. for (j=1; j<=n; ++j)
. . if (i<=n-j+2)
. . . s[i] = ...
```

$$\mathcal{D}_{S1} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ -1 & -1 & 1 & 2 \end{bmatrix} . \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \geq \vec{0}$$



Iteration domain of S_1

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 $\vec{p}$

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

$$f_{\mathbf{s}}(\vec{x_{S2}}) = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} . \begin{pmatrix} \vec{x_{S2}} \\ n \\ 1 \end{pmatrix}$$

$$f_{\mathbf{a}}(\vec{x_{S2}}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} . \begin{pmatrix} \vec{x_{S2}} \\ n \\ 1 \end{pmatrix}$$

$$f_{\mathbf{x}}(\vec{x_{S2}}) = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} . \begin{pmatrix} \vec{x_{S2}} \\ n \\ 1 \end{pmatrix}$$

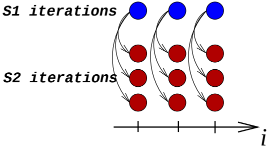
Polyhedral Representation of Programs

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 $\mathcal{D}_{S1}$ and $\mathcal{D}_{S2}$ (**exact analysis**)

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

$$\mathcal{D}_{S1\delta S2}: \left[\begin{array}{cccc} 1 & -1 & 0 & 0 \\ \hline 1 & 0 & 0 & -1 \\ -1 & 0 & 0 & 3 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 3 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 3 \end{array}\right] \cdot \begin{pmatrix} i_{S1} \\ i_{S2} \\ j_{S2} \\ 1 \end{pmatrix} \begin{array}{c} = 0 \\ \geq \vec{0} \end{array}$$



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

	description	#loops	#stmts	#refs	#deps	#part.	#valid	Variability	Pb. Size
2mm	Linear algebra (BLAS3)	6	4	8	12	75	12	✓	1024x1024
3mm	Linear algebra (BLAS3)	9	6	12	19	4683	128	✓	1024x1024
adi	Stencil (2D)	11	8	36	188	545835	1		1024x1024
atax	Linear algebra (BLAS2)	4	4	10	12	75	16	✓	8000x8000
bicg	Linear algebra (BLAS2)	3	4	10	10	75	26	✓	8000x8000
correl	Correlation (PCA: StatLib)	5	6	12	14	4683	176	✓	500x500
covar	Covariance (PCA: StatLib)	7	7	13	26	47293	96	✓	500x500
doitgen	Linear algebra	5	3	7	8	13	4		128x128x128
gemm	Linear algebra (BLAS3)	3	2	6	6	3	2		1024x1024
gemver	Linear algebra (BLAS2)	7	4	19	13	75	8	✓	8000x8000
gesummv	Linear algebra (BLAS2)	2	5	15	17	541	44	✓	8000x8000
gramschmidt	Matrix normalization	6	7	17	34	47293	1		512x512
jacobi-2d	Stencil (2D)	5	2	8	14	3	1		20x1024x1024
lu	Matrix decomposition	4	2	7	10	3	1		1024x1024
ludcmp	Solver	9	15	40	188	10^{12}	20	✓	1024x1024
seidel	Stencil (2D)	3	1	10	27	1	1		20x1024x1024

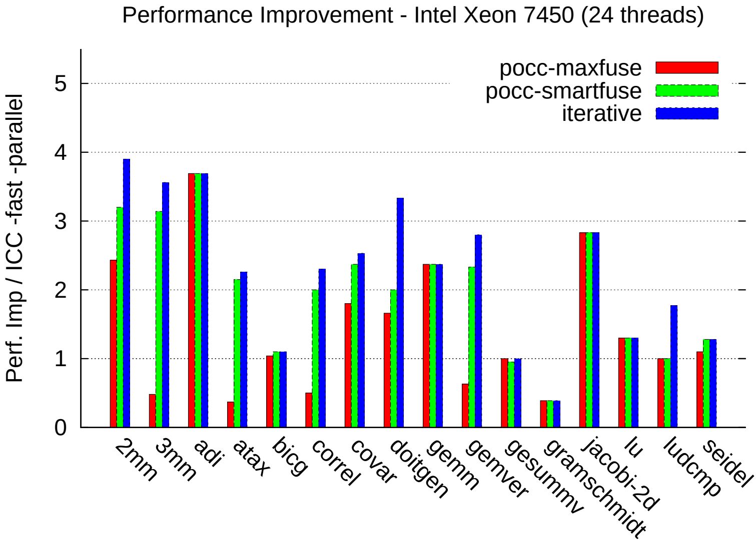
Table: Summary of the optimization process

Experimental Setup

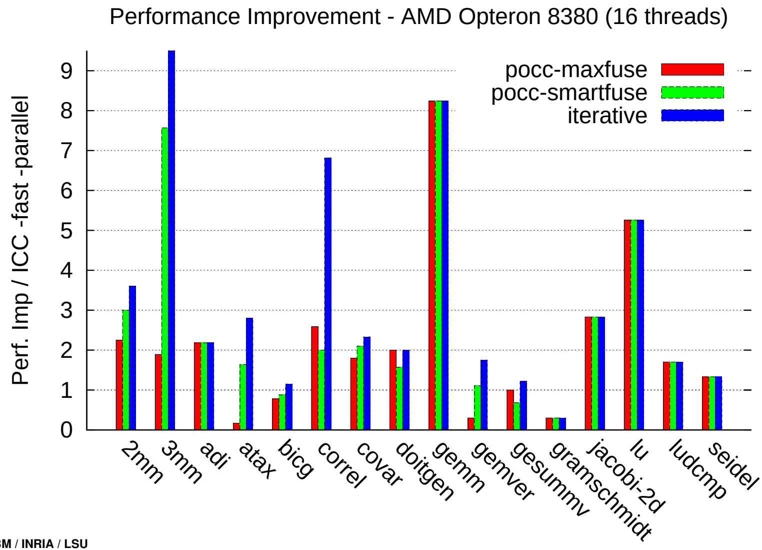
We compare three schemes:

- **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

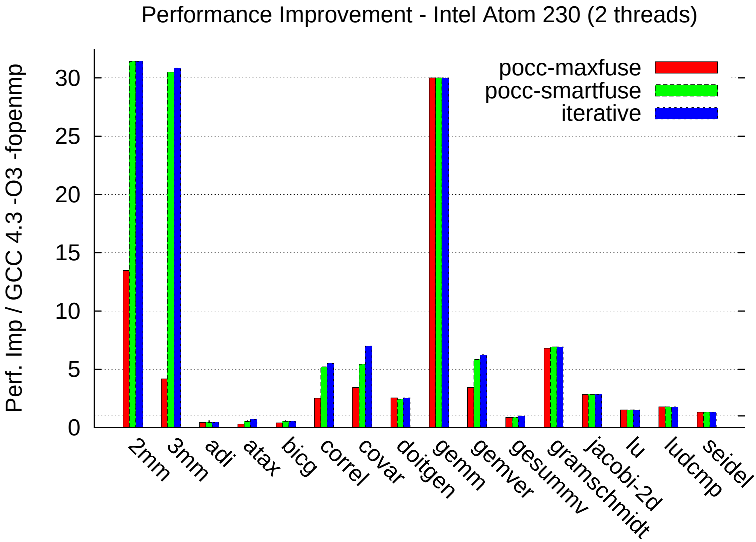
Performance Results - Intel Xeon 7450 - ICC 11



Performance Results - AMD Opteron 8380 - ICC 11



Performance Results - Intel Atom 330 - GCC 4.3



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:

$\Rightarrow$ **Fusion / Distribution / Code motion highly program- and machine-specific**

$\Rightarrow$ **Minimum empirical tuning + polyhedral framework gives very good performance on several applications**

$\Rightarrow$ **Complete, end-to-end framework implemented and effectiveness demonstrated**

Future work:

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