Verified Validation for Affine Polyhedral Scheduling

Xuyang Li, Hongjin Liang, Xinyu Feng

Nanjing University

2023.11.18
Outline

Background
  Motivation
  Loop optimization and Polyhedral Compilation
  Affine Polyhedral Transformation

Abstraction of Instruction Language

The Verified Validator

Case Study
  the verified compiler, CompCert
  the polyhedral compiler, Pluto

Discussion
Motivation

- Verified compilation is useful, like CompCert [Leroy, 2009b, Leroy, 2009a].
- Optimizing compilation is useful, like loop optimization.

Status Quo

- Verified compilation is not enough optimizing. [Monniaux and Six, 2022, Gourdin et al., 2023]
- Loop optimization is error prone. [Livinskii et al., 2023]

Efforts are needed to bridge the gap.
Some loop optimizations do structural transformation on nested loop, seeking for better memory locality (and parallelizability). Sample:\(^1:\)

\(^1\)https://github.com/verif-scop/speed-up
Some loop optimizations do structural transformation on *nested loop*, seeking for better memory locality (and parallelizability).

Sample\(^1\):

---

Original code \(\pi'_{cov}\) for covariance matrix calculation, 1.84s

```c
for (j1 = 1; j1 <= M; j1++) {
    for (j2 = j1; j2 <= M; j2++) {
        for (i = 1; i <= N; i++) {
            symmat[j1][j2] += data[i][j1] * data[i][j2];
        }
        symmat[j2][j1] = symmat[j1][j2];
    }
}
```

---

\(^1\)https://github.com/verif-scop/speed-up
Loop Optimization...

Some loop optimizations do structural transformation on nested loop, seeking for better memory locality (and parallelizability).

Sample\(^1\):

Optimized code \(\pi_{cov}'\) for covariance matrix calculation, 0.43s, with loop distribution and loop interchange

```c
for (i = 1; i <= N; i++) {
    for (j1 = 1; j1 <= M; j1++) {
        for (j2 = j1; j2 <= M; j2++) {
            symmat[j1][j2] += data[i][j1] * data[i][j2];
        }
    }
}
for (j1 = 1; j1 <= M; j1++) {
    for (j2 = j1; j2 <= M; j2++) {
        symmat[j2][j1] = symmat[j1][j2];
    }
}
```

\(^1\)https://github.com/verif-scop/speed-up
Such optimizations can be done with polyhedral compilation techniques, the famous one is Pluto [Bondhugula et al., 2008]. In polyhedral model, we are able to analyze at a finer grain, i.e., each dynamic instance of every instruction, with the support of integer programming. Though, it impose the restrictions that all loop bounds, conditions, array subscripts should be affine expressions w.r.t. constant variables and iterators.
... and Polyhedral Compilation

Original code $\pi'_{cov}$ for covariance matrix calculation, 1.84s

```c
for (j1 = 1; j1 <= M; j1++) {
    for (j2 = j1; j2 <= M; j2++) {
        for (i = 1; i <= N; i++) {
            I_0: symmat[j1][j2] += data[i][j1] * data[i][j2];
        }
        I_1: symmat[j2][j1] = symmat[j1][j2];
    }
}
```

We use refer each instance of an instruction with an iteration vector, values of its surrounding constant variables (i.e., $[M, N]$, denoted with $\mathcal{E}_\star$) and iterators.

$I_0(M,N;j_1,j_2,i) = \text{symmat}[j_1][j_2] += \text{data}[i][j_1] * \text{data}[i][j_2]$;
$I_1(M,N;j_1,j_2) = \text{symmat}[j_2][j_1] = \text{symmat}[j_1][j_2]$;
Original code $\pi'_{\text{cov}}$ for covariance matrix calculation, 1.84s

```
for (j1 = 1; j1 <= M; j1++) {
    for (j2 = j1; j2 <= M; j2++) {
        for (i = 1; i <= N; i++) {
            I_0: symmat[j1][j2] += data[i][j1] * data[i][j2];
        }
        I_1: symmat[j2][j1] = symmat[j1][j2];
    }
}
```

We record the domain (set) of an instruction’s instances with a polyhedron, i.e., a list of integral and linear constraints.

$$
\mathcal{D}_0 = \{(M, N; j_1, j_2, i) \mid 1 \leq j_1 \leq M \land j_1 \leq j_2 \leq M \land 1 \leq i \leq N\}
$$

$$
\mathcal{D}_1 = \{(M, N; j_1, j_2) \mid 1 \leq j_1 \leq M \land j_1 \leq j_2 \leq M\}
$$
and Polyhedral Compilation

Original code $\pi'_{cov}$ for covariance matrix calculation, 1.84s

```c
for (j1 = 1; j1 <= M; j1++) {
    for (j2 = j1; j2 <= M; j2++) {
        for (i = 1; i <= N; i++) {
            I_0: symmat[j1][j2] += data[i][j1] * data[i][j2];
        }
        I_1: symmat[j2][j1] = symmat[j1][j2];
    }
}
```

We additionally record a timestamp for each instance, representing its execution order, with an affine function called `schedule`.

$$
\theta_0 = (M, N; j1, j2, i) \mapsto (j1, j2, i) \\
\theta_1 = (M, N; j1, j2) \mapsto (j1, j2)
$$
... and Polyhedral Compilation

Original code $\pi'_{cov}$ for covariance matrix calculation, 1.84s

```c
for (j1 = 1; j1 <= M; j1++) {
    for (j2 = j1; j2 <= M; j2++) {
        for (i = 1; i <= N; i++) {
            I_0: symmat[j1][j2] += data[i][j1] * data[i][j2];
        }
        I_1: symmat[j2][j1] = symmat[j1][j2];
    }
}
```

To do optimizations, we need write and read access functions for instruction’s memory access patterns, which should be consistent with instruction’s semantics.

\[
\begin{align*}
\mathcal{W}_0 &= (M, N; j1, j2, i) \mapsto \{ (\text{symmat}, [j1, j2]) \} \\
\mathcal{R}_0 &= (M, N; j1, j2, i) \mapsto \{ (\text{symmat}, [j1, j2]), (\text{data}, [i, j1]), (\text{data}, [i, j2]) \} \\
\mathcal{W}_1 &= (M, N; j1, j2) \mapsto \{ (\text{symmat}, [j2, j1]) \} \\
\mathcal{R}_1 &= (M, N; j1, j2) \mapsto \{ (\text{symmat}, [j1, j2]) \}
\end{align*}
\]
To sum up, any polyhedral model $\mathcal{P}$, contains a list of polyhedral instructions $\mathcal{I}$, its constant variables $E_\star$, and its typing information $\Upsilon$ (like, `symmat` is of type `INT[Y][Z]`). Each polyhedral instruction $I$ is a five-tuple $= (I, D, \theta, \mathcal{W}, \mathcal{R})$, as pre-mentioned. Its semantic judgement: $\models \mathcal{P}, \sigma \Rightarrow \sigma'$. 
Affine Polyhedral Transformation

We focus on reordering transformations only, i.e., changing each schedule function $\theta$ into another $\theta'$.

Optimized code $\pi'_{cov}$ for covariance matrix calculation, 0.43s, with loop distribution and loop interchange

```
for (i = 1; i <= N; i++) {
    for (j1 = 1; j1 <= M; j1++) {
        for (j2 = j1; j2 <= M; j2++) {
            symmat[j1][j2] += data[i][j1] * data[i][j2];
        }
    }
}
```

```
for (j1 = 1; j1 <= M; j1++) {
    for (j2 = j1; j2 <= M; j2++) {
        symmat[j2][j1] = symmat[j1][j2];
    }
}
```

$\theta'_0 = (M, N; j1, j2, i) \mapsto (0, i, j1, j2)$

$\theta'_1 = (M, N; j1, j2) \mapsto (1, j1, j2)$
Affine Polyhedral Transformation

The correctness criterion: Bernstein’s conditions [Feautrier, 2011]. It says, preservation of all WAW/RAW/WAR dependences implies semantics preservation.
Polyhedral compilation does high-level structural transformations and only impose a few properties of the underlying instruction language (called $\mathcal{I}$). The validation function given in this work is parameterized by $\mathcal{I}$.

$\mathcal{I}$ allows user define the syntax, types, state, semantics of the language, how it initializes, and its the non-alias proposition. It demands user to provide a verified Checker function to validate the consistency between the read and write access function and an instruction’s semantics, and prove that any two instances that satisfy Bernstein’s conditions are permutable.
Module Type $\mathbb{I} \triangleq$

\[
\begin{align*}
T, I, S: & \text{ Type} \\
\models & : I \rightarrow \text{List}(\mathbb{Z}) \rightarrow \text{Memory Cells} \\
& \rightarrow \text{Memory Cells} \rightarrow S \rightarrow S \rightarrow \text{Prop} \\
\text{Compat}: & \text{List(Identifier)} \rightarrow S \rightarrow \text{Prop} \\
\text{Consistent}: & \text{List(Identifier} \times T) \rightarrow \text{List(\mathbb{Z})} \rightarrow S \rightarrow \text{Prop} \\
\text{NonAlias}: & S \rightarrow \text{Prop} \\
\text{NonAliasPsrv}: & \forall I, \sigma, \sigma'. \text{NonAlias}(\sigma) \land p \models I, \sigma \rightarrow \sigma' \Rightarrow \text{NonAlias}(\sigma'). \\
\text{Checker}: & I \rightarrow \text{Access Functions} \\
& \rightarrow \text{Access Functions} \rightarrow \text{Bool} \\
\text{Correct(Checker)}: & \forall I, W, R. \text{Checker}(I, W, R) = \text{true} \Rightarrow (\forall \sigma, \sigma', p, \Delta_r, \Delta_w. p \models I, \sigma \xrightarrow{\Delta_r, \Delta_w} \sigma' \Rightarrow \Delta_r \subseteq R(p) \land \Delta_w \subseteq W(p)). \\
\text{BCPermut}: & \forall I_1, I_2, p_1, p_2, \sigma, \sigma', \sigma'', \Delta_r, \Delta_w, \Delta_r', \Delta_w'. \\
(p_1 \models I_1, \sigma \xrightarrow{\Delta_r, \Delta_w} \sigma' \land p_2 \models I_2, \sigma' \xrightarrow{\Delta_r', \Delta_w'} \sigma'' \\
\land \Delta_r \cap \Delta_r' = \emptyset \land \Delta_w \cap \Delta_w' = \emptyset \land \Delta_r \cap \Delta_w \cap \Delta_r' \cap \Delta_w' = \emptyset) \\
\Rightarrow \exists \sigma^*. p_2 \models I_2, \sigma \xrightarrow{\Delta_r', \Delta_w'} \sigma^* \land p_1 \models I_1, \sigma^* \xrightarrow{\Delta_r, \Delta_w} \sigma''.
\end{align*}
\]

Figure 1: Definition of Instruction Language Module $\mathbb{I}$
Our work: Verified Validation for Affine Polyhedral Scheduling

We define a validation function *Validate* that checks Bernstein’s condition within polyhedral model, and machanized it with Coq proof assistent. Here we give our proof goal. The proof basically deals with permutation.

Definition (correctness of the validator)

\[
\text{Correct}(\text{Validate}) \triangleq \forall P_s, P_t. \text{Validate}(P_s, P_t) = \text{true} \implies P_t \subseteq P_s.
\]

\[
P_t \subseteq P_s \triangleq \forall \sigma, \sigma'.
\]

\[
| = P_t, \sigma \Rightarrow \sigma' \implies | = P_s, \sigma \Rightarrow \sigma'.
\]
Case study: Towards verified polyhedral compilation for CompCert

We instantiate the validation function with CompCert C's type, state and subset of its instruction language, and implement Checker with symbolic execution. All verified. Only differences are, affine expression is evaluated in $\mathbb{Z}$ (no overflow), and multi-dimensional array access is sugarized.

\[
\begin{align*}
\text{(Base Type)} & \quad \tau_{\bot} ::= \text{int32s} \\
\text{(Type)} & \quad \tau \in \text{Base Type} \times \text{List}(\mathbb{Z}) \\
\text{(Value)} & \quad v ::= \text{I32}(n) | \ldots \\
\text{(Iterator)} & \quad i \in \mathbb{N} \\
\text{(Unop)} & \quad op_1 ::= - | \ldots \\
\text{(Binop)} & \quad op_2 ::= + | * | \ldots \\
\text{(May Affine Expression)} & \quad \varepsilon ::= z | i | op_1 \varepsilon | \varepsilon_1 \ op_2 \varepsilon_2 \\
\text{(Access Expression)} & \quad \epsilon \in \text{Identifier} \times \text{Base Type} \times \text{List}(\text{May Affine Expression}) \\
\text{(Expression)} & \quad e ::= v | i | \epsilon | op_1 \epsilon | e_1 \ op_2 \epsilon_2 \\
\text{(Base Instruction)} & \quad I ::= \text{skip} | \epsilon:=e
\end{align*}
\]
Case study: Validating Pluto compiler’s core algorithm

We adapt the validation function to real world polyhedral compiler, the Pluto compiler (optioned to affine scheduling only). We use OpenScop format for file exchange. We *assume* Pluto infers correct access functions, and language for input language conforms to $\mathbb{I}$. This is because Pluto’s frontend does not provide typing information, supports complex expressions and possible different languages. All 62 available tests in Pluto’s testsuit are proved by our validation function with reasonable time. This shows our work’s practicality (being complete enough).
Case study: Validating Pluto compiler’s core algorithm

<table>
<thead>
<tr>
<th>Test</th>
<th>Time of Pluto (ms)</th>
<th>Time of Validation (ms,ms)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>covcol</td>
<td>3.5</td>
<td>434.6, 320.7</td>
<td>EQ</td>
</tr>
<tr>
<td>dsyr2k</td>
<td>2.6</td>
<td>106.0, 83.4</td>
<td>EQ</td>
</tr>
<tr>
<td>fdtd-2d</td>
<td>46.4</td>
<td>1615.5, 1296.3</td>
<td>EQ</td>
</tr>
<tr>
<td>gemver</td>
<td>7.0</td>
<td>247.9, 240.4</td>
<td>EQ</td>
</tr>
<tr>
<td>lu</td>
<td>6.1</td>
<td>410.6, 331.2</td>
<td>EQ</td>
</tr>
<tr>
<td>mvt</td>
<td>2.2</td>
<td>70.2, 56.3</td>
<td>EQ</td>
</tr>
<tr>
<td>ssymm</td>
<td>40.7</td>
<td>726.0, 551.2</td>
<td>EQ</td>
</tr>
<tr>
<td>tce</td>
<td>568.6</td>
<td>4442.0, 4422.5</td>
<td>EQ</td>
</tr>
<tr>
<td>adi</td>
<td>77.5</td>
<td>2531.7, 2377.8</td>
<td>EQ</td>
</tr>
<tr>
<td>corcol</td>
<td>5.5</td>
<td>442.5, 362.1</td>
<td>EQ</td>
</tr>
<tr>
<td>dct</td>
<td>21.8</td>
<td>879.4, 739.4</td>
<td>EQ</td>
</tr>
<tr>
<td>dsyrk</td>
<td>1.8</td>
<td>96.8, 78.9</td>
<td>EQ</td>
</tr>
<tr>
<td>floyd</td>
<td>12.1</td>
<td>502.6, 421.7</td>
<td>EQ</td>
</tr>
<tr>
<td>jacobi-1d-imper</td>
<td>3.8</td>
<td>184.0, 167.8</td>
<td>EQ</td>
</tr>
<tr>
<td>matmul-init</td>
<td>2.9</td>
<td>257.8, 192.4</td>
<td>EQ</td>
</tr>
<tr>
<td>pca</td>
<td>202.5</td>
<td>2923.6, 2679.5</td>
<td>EQ</td>
</tr>
<tr>
<td>strmm</td>
<td>1.9</td>
<td>141.4, 110.8</td>
<td>EQ</td>
</tr>
<tr>
<td>tmm</td>
<td>1.6</td>
<td>109.7, 89.6</td>
<td>EQ</td>
</tr>
<tr>
<td>advect3d</td>
<td>1023.1</td>
<td>579.1, 498.1</td>
<td>EQ</td>
</tr>
<tr>
<td>corcol3</td>
<td>13.6</td>
<td>851.3, 733.4</td>
<td>EQ</td>
</tr>
<tr>
<td>doitgen</td>
<td>10.4</td>
<td>1069.2, 837.4</td>
<td>EQ</td>
</tr>
<tr>
<td>fdtd-1d</td>
<td>6.0</td>
<td>268.7, 229.9</td>
<td>EQ</td>
</tr>
<tr>
<td>jacobi-2d-imper</td>
<td>17.7</td>
<td>619.5, 543.5</td>
<td>EQ</td>
</tr>
<tr>
<td>matmul</td>
<td>3.2</td>
<td>157.1, 125.5</td>
<td>EQ</td>
</tr>
<tr>
<td>seidel</td>
<td>24.5</td>
<td>818.1, 725.5</td>
<td>EQ</td>
</tr>
<tr>
<td>strsm</td>
<td>6.4</td>
<td>209.3, 161.2</td>
<td>EQ</td>
</tr>
<tr>
<td>trisolv</td>
<td>5.1</td>
<td>338.9, 248.8</td>
<td>EQ</td>
</tr>
<tr>
<td>1dloop-invar</td>
<td>0.3</td>
<td>6.7, 6.0</td>
<td>EQ</td>
</tr>
<tr>
<td>costfunc</td>
<td>0.8</td>
<td>47.4, 35.0</td>
<td>EQ</td>
</tr>
<tr>
<td>fusion1</td>
<td>0.9</td>
<td>15.3, 13.9</td>
<td>EQ</td>
</tr>
</tbody>
</table>

Table 1: Evaluation results on Pluto’s test suits
Discussion

- Around 18000 lines of Coq, 1000 lines of OCaml. Open source at https://github.com/verif-scop/PolCert.

- Our work bases on Verified Polytope Library [Boulmé et al., 2018], and is compatible to PolyGen [Courant and Leroy, 2021] in POPL’21. However, we found polyhedral model’s semantics of PolyGen flawsome ², so we cannot prove the semantics equivalence (we tried indeed!). Left as future work.

- s2sloop [Pilkiewicz, 2013] shows similar idea (i.e., checking Bernstein’s condition), though this work is incomplete.

²https://github.com/Ekdohibs/PolyGen/issues/1
Discussion

- To achieve complete polyhedral extension to CompCert, we still need a verified extractor. Engineering efforts are also needed in CompCert’s frontend, supporting pragma and sugarization.

- Optimistic approach [Doerfert et al., 2017] seems promising in dealing with polyhedral model’s heavy assumptions, like integer overflow [Cuervo Parrino et al., 2012].
References I


References II


Formally verifying optimizations with block simulations. 
*Proc. ACM Program. Lang.*, 7(OOPSLA2).

Formal verification of a realistic compiler. 

A formally verified compiler back-end. 

Fuzzing loop optimizations in compilers for c++ and data-parallel languages. 
*Proc. ACM Program. Lang.*, 7(PLDI).

https://github.com/pilki/s2sLoop.