Polyhedral-Based Data Reuse Optimization for Configurable Computing

Louis-Noël Pouchet\textsuperscript{1} Peng Zhang\textsuperscript{1} P. Sadayappan\textsuperscript{2} Jason Cong\textsuperscript{1}

\textsuperscript{1} University of California, Los Angeles  
\textsuperscript{2} The Ohio State University

February 12, 2013  
ACM/SIGDA International Symposium on Field-Programmable Gate Arrays  
Monterey, CA
Overview

The current situation:

▶ Tremendous improvements on FPGA capacity/speed/energy
▶ But off-chip communications remains very costly, on-chip memory is scarce

▶ HLS/ESL tools have made great progresses (ex: AutoESL/Vivado)
▶ But still extensive manual effort needed for best performance

▶ Numerous previous research work on C-to-FPGA (PICO, DEFACTO, MMAAlpha, etc.) and data reuse optimizations
▶ But (strong) limitations in applicability / transformations supported / performance achieved
Overview

The current situation:

▶ Tremendous improvements on FPGA capacity/speed/energy
▶ But off-chip communications remains very costly, on-chip memory is scarce

▶ HLS/ESL tools have made great progresses (ex: AutoESL/Vivado)
▶ But still extensive manual effort needed for best performance

▶ Numerous previous research work on C-to-FPGA (PICO, DEFACTO, MMAlpha, etc.) and data reuse optimizations
▶ But (strong) limitations in applicability / transformations supported / performance achieved
Overview

The current situation:

▶ Tremendous improvements on FPGA capacity/speed/energy
▶ But off-chip communications remains very costly, on-chip memory is scarce

▶ HLS/ESL tools have made great progresses (ex: AutoESL/Vivado)
▶ But still extensive manual effort needed for best performance

▶ Numerous previous research work on C-to-FPGA (PICO, DEFACTO, MMAlpha, etc.) and data reuse optimizations
▶ But (strong) limitations in applicability / transformations supported / performance achieved
Overview

The current situation:

- Tremendous improvements on FPGA capacity/speed/energy
- But off-chip communications remains very costly, on-chip memory is scarce

⇒ Our solution: automatic, resource-aware data reuse optimization framework (combining loop transformations, on-chip buffers, and communication generation)

- HLS/ESL tools have made great progresses (ex: AutoESL/Vivado)
- But still extensive manual effort needed for best performance

- Numerous previous research work on C-to-FPGA (PICO, DEFACTO, MMAlpha, etc.) and data reuse optimizations
- But (strong) limitations in applicability / transformations supported / performance achieved
Overview

The current situation:

▶ Tremendous improvements on FPGA capacity/speed/energy
▶ But off-chip communications remains very costly, on-chip memory is scarce

⇒ Our solution: automatic, resource-aware data reuse optimization framework (combining loop transformations, on-chip buffers, and communication generation)

▶ HLS/ESL tools have made great progresses (ex: AutoESL/Vivado)
▶ But still extensive manual effort needed for best performance

⇒ Our solution: complete HLS-focused source-to-source compiler

▶ Numerous previous research work on C-to-FPGA (PICO, DEFACTO, MMAlpha, etc.) and data reuse optimizations
▶ But (strong) limitations in applicability / transformations supported / performance achieved
Overview

The current situation:

- Tremendous improvements on FPGA capacity/speed/energy
- But off-chip communications remains very costly, on-chip memory is scarce

⇒ **Our solution:** automatic, resource-aware data reuse optimization framework (combining loop transformations, on-chip buffers, and communication generation)

- HLS/ESL tools have made great progresses (ex: AutoESL/Vivado)
- But still extensive manual effort needed for best performance

⇒ **Our solution:** complete HLS-focused source-to-source compiler

- Numerous previous research work on C-to-FPGA (PICO, DEFACTO, MMAlpha, etc.) and data reuse optimizations
- But (strong) limitations in applicability / transformations supported / performance achieved

⇒ **Our solution:** unleash the true power of the polyhedral framework (loop transfo., comm. scheduling, etc.)
The Polyhedral Model in a Nutshell

Affine program regions:

- Loops have affine control only (over-approximation otherwise)
  - Image processing, including medical imaging pipeline (NSF CDSC project)
  - Linear algebra
  - Iterative solvers (PDE, etc.)
The Polyhedral Model in a Nutshell

Affine program regions:

- 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} \cdot \begin{bmatrix} i \\ j \\ n \\ 1 \end{bmatrix} \geq 0
```
The Polyhedral Model in a Nutshell

Affine program regions:

- 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 \( \vec{p} \)

\[
\begin{align*}
\text{for (i=0; i<n; ++i) } & \{ \\
. \text{ s[i] = 0;} \\
. \text{ for (j=0; j<n; ++j) } \\
. \text{ s[i] = s[i]+a[i][j]*x[j]; } \\
\}
\end{align*}
\]

\[
\begin{align*}
\mathbf{f}_s(\vec{x}_{S2}) &= \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_{S2}^x \\ n \\ 1 \end{bmatrix} \\
\mathbf{f}_a(\vec{x}_{S2}) &= \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_{S2}^x \\ n \\ 1 \end{bmatrix} \\
\mathbf{f}_x(\vec{x}_{S2}) &= \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_{S2}^x \\ n \\ 1 \end{bmatrix}
\end{align*}
\]
The Polyhedral Model in a Nutshell

Affine program regions:

- 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 $S_1$ and $S_2$: a subset of the Cartesian product of $D_{S_1}$ and $D_{S_2}$ (exact analysis)

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

$D_{S_1 \delta S_2} = \begin{bmatrix}
1 & -1 & 0 & 0 \\
1 & 0 & 0 & -1 \\
0 & 1 & 0 & -1 \\
0 & 0 & 1 & -1 \\
0 & 0 & -1 & 3
\end{bmatrix}$

\[ \begin{pmatrix}
iS_1 \\ iS_2 \\ jS_2 \\
i\end{pmatrix} \begin{pmatrix}
1 & -1 & 0 & 0 \\
1 & 0 & 0 & -1 \\
0 & 1 & 0 & -1 \\
0 & 0 & 1 & -1 \\
0 & 0 & -1 & 3
\end{pmatrix} \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} \begin{pmatrix}
iS_1 \\ iS_2 \\ jS_2 \\
i\end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \geq 0$

$iS_1$ iterations
$iS_2$ iterations

S1 iterations
S2 iterations
The Polyhedral Model in a Nutshell

Affine program regions:

- 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)

Polyhedral compilation:

- Precise dataflow analysis [Feautrier,88]
- Optimal algorithms for data locality [Bondhugula,08]
- Effective code generation [Bastoul,04]
- Computationally expensive algorithms (ILP/PIP)
Step 1: Scheduling for Better Data Reuse

- **Main idea:** schedule operations accessing the same data as close as possible from each other

- **Tiling is useful, but not all programs are tilable by default!**
  - Need complex sequence of loop transformations to enable tiling
  - The Tiling Hyperplane method automatically finds such sequence
  - Uses an ILP for the optimization problem

- In our software, the first stage is to transform the input code so that:
  1. The number of tilable "loops" is maximized
  2. Temporal data locality is maximized
  3. All tilable loops can be tiled with an arbitrary tile size
Step 2: Reuse Data Using On-Chip Buffers

Key ideas:

- Compute the set of data used at a given loop iteration
- Reuse data between consecutive loop iterations
- The process works for any loop in the program
- Natural complement of tiling: the tile size will determine how much data is read by a non-inner-loop iteration
- The polyhedral framework can be used to easily compute all this information, including what to communicate
Computing the Per-Iteration Data Reuse

// Two-dimensional Jacobi-like stencil
for (t = 0; t < T; ++t)
    for (i = 0; i < N; ++i)
        for (j = 0; j < N; ++j)
            B[i][j] = 0.2*( A[i][j-1]
                            + A[i][j]
                            + A[i][j+1]
                            + A[i-1][j]
                            + A[i+1][j]);
Computing the Per-Iteration Data Reuse

Compute the data space of A, at iteration $\vec{x} = (t, i, j)$

$$DS_A(\vec{x}) = \bigcup_{s \in S} FS_A^s(\vec{x})$$

$F(\vec{x})$ is the image of $\vec{x}$ by the function $F$. 
Computing the Per-Iteration Data Reuse

Compute the data space of A, at iteration \( \vec{y} = (t, i, j - 1) \)

\[
DS_A(\vec{y}) = \bigcup_{s \in S} FS_A^s(\vec{y})
\]
Computing the Per-Iteration Data Reuse

\[ \text{ReuseSet} = \text{DS}_A(\vec{x}) \cap \text{DS}_A(\vec{y}) \]
Computing the Per-Iteration Data Reuse

Per-iteration communication: blue set

$$PerCommSet = DS_B(\vec{x}) - ReuseSet$$
Computing the Per-Iteration Data Reuse

These sets are parametric polyhedral sets

- Use CLooG to scan them
- Work for any value of $t, i, j$

→ an initialization copy is executed before the first iteration of the loop, and communications are done at each iteration
Computing the Per-Iteration Data Reuse

Buffer set: full blue set (data space at \((t, i, j)\))
Quick Overview of the Full Algorithm

1. For each array and each loop, compute:
   - the buffer polyhedron
   - the per-iteration communication polyhedron

2. For a given array, find the loop which minimizes communication volume with a buffer fitting the FPGA resource

3. Make the entire program use on-chip arrays (buffers)
   - Example: \( A[i][j] = A[i][j+1] \) becomes for a buffer \( A_1[bs1][bs2] \):
     \[ A_1[i \mod bs1][j \mod bs2] = A_1[i \mod bs1][(j+1) \mod bs2] \]

4. Insert the codes scanning the polyhedral sets in the program
   - Example of copy-in statement: \( A_1[i \mod bs1][j \mod bs2] = A[i][j] \);
Step 3: HLS-specific Optimizations

For good performance, numerous complementary optimizations needed

- Reduce the II of inner loops by forcing inner-most parallel loops
  - Use polyhedral-based parallelization methods

- Exhibit usable task-level parallelism
  - Use polyhedral-based analysis, and factor the tasks in functions

- Overlap communication and computation
  - Use FIFO communication modules, and scan polyhedral communication sets also in prefetch functions to issue requests

- Find the best tile size / shape for a program
  - Create a machine-specific accurate communication latency model
  - Run AutoESL on a variety of tile sizes, retain the best one
Performance Results

Table 2 summarizes the best version found by our framework, which automatically generated code. On the other hand, the reference manual design uses numerous techniques not implemented in automatically generated code. On the other hand, the reference manual design uses numerous techniques not implemented in automatically generated code. However, the PolyOpt/HLS-E system provides near-optimal performance for this version.

Table 3 reports the performance, in GFlop per second, of a basic manual off-chip-only implementation using numerous techniques not implemented in automatically generated code. We report the performance of a single PE call executing a subset (slice) of the full computation we have been able to place on a single core.

### Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>basic off-chip</th>
<th>PolyOpt</th>
<th>hand-tuned [17]</th>
</tr>
</thead>
<tbody>
<tr>
<td>denoise</td>
<td>3D Jacobi+Seidel-like 7-point stencils</td>
<td>0.02 GF/s</td>
<td>4.58 GF/s</td>
<td>52.0 GF/s</td>
</tr>
<tr>
<td>segmentation</td>
<td>3D Jacobi-like 7-point stencils</td>
<td>0.05 GF/s</td>
<td>24.91 GF/s</td>
<td>23.39 GF/s</td>
</tr>
<tr>
<td>DGEMM</td>
<td>matrix-multiplication</td>
<td>0.04 GF/s</td>
<td>22.72 GF/s</td>
<td>N/A</td>
</tr>
<tr>
<td>GEMVER</td>
<td>sequence of matrix-vector</td>
<td>0.10 GF/s</td>
<td>1.07 GF/s</td>
<td>N/A</td>
</tr>
</tbody>
</table>

- Convey HC-1 (4 Xilinx Virtex-6 FPGAs), total bandwidth up to 80GB/s
- AutoESL version 2011.1, use memory/control interfaces provided by Convey
- Core design frequency: 150MHz, off-chip memory frequency: 300MHz
PolyOpt/HLS

Input full C program

Parser
C-to-AST

PolyParser
AST-to-polyhedra

Unparser
AST-to-C

Outliner
restructure code for HLS

PolyUnparser
PAST-to-AST

Candl
dependence analysis

Pluto
Transfo. for tilability

vectorizer
Transfo. for inner-parallel

CLooG
Polyhedra-to-PAST

LMP
buffer and comm. generation

PIPLib

More at http://www.cs.ucla.edu/~pouchet/software/polyopthls

C code
Sage AST (ROSE)
SCoP (polyhedral rep.)
PAST (Polyhedral AST)

PoCC, the Polyhedral Compiler Collection
PolyOpt, a Polyhedral Optimizer for the ROSE compiler
ROSE compiler infrastructure (LLNL)
Conclusions

Take-home message:

▶ Affine programs are an excellent fit for FPGA/HLS

▶ Recent progresses in HLS tools let compiler researchers target FPGA optimization

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

Future work:

▶ Use analytical models for tile size selection
▶ Improve further the performance with additional optimizations
▶ Support more machines/FPGAs (currently: developed for Convey HC-1)
▶ Improve polyhedral code generation for HLS/FPGAs