Throughput Optimization for High-Level Synthesis Using Resource Constraints

Peng Li$^{1,2}$  Louis-Noël Pouchet$^{3,2}$  Jason Cong$^{3,1,2}$

1 Center for Energy-efficient Computing and Application, Peking University
2 PKU/UCLA Joint Research Institution for Science and Engineering
3 University of California, Los Angeles

January 20, 2014
Fourth International Workshop on Polyhedral Compilation Techniques
Vienna, Austria
(Very) High Level Picture

1. FPGAs: Field-Programmable Gate Arrays
2. HLS: High-Level Synthesis (from C to RTL)
3. Synthesis: “from RTL to FPGA”
4. => A toolchain from C to hardware! (ex: Xilinx Vivado ISE)

- Our job: C to FPGA, using source-to-source C transfo.
- We focus on affine C programs :-/
A Previous Work: PolyOpt/HLS

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
A Previous Work: PolyOpt/HLS

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
A Previous Work: PolyOpt/HLS

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
A Previous Work: PolyOpt/HLS

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
A Previous Work: PolyOpt/HLS

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
A Previous Work: PolyOpt/HLS

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 power of the polyhedral framework (loop transfo., comm. scheduling, etc.)
Performance Results

<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
Context of This Work

How to get good throughput?

1. Good management of off-chip communications, and on-chip data reuse
2. Effective on-chip computation module

- Previous work focused on tiling, comm. optimization, localization, and “coarse-grain” parallelism exposure
- This work: focus on improving computation module (assume data is on-chip)
  - Question: are previous techniques enough?
  - Question: can we design techniques to improve pipelining efficiency?
Loop Pipelining [1/3]

- Depth: number of cycles needed to complete one iteration
- Initiation Interval (II): number of cycles to wait before the next iteration can start

![Diagram showing depth and initiation interval]

- Total cycles: \((\text{LoopTripCount} - 1) \times \text{II} + \text{Depth}\)
- Reasons for II > 1
  - Data dependence (typically loop-carried dependence)
  - Resource constraints (typically the resource needed is still in use)
Loop Pipelining [2/3]

Example (dgemm)

```c
for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
        #pragma AP pipeline II=1
        for (k = 0; k < nk; ++k)
            C[i][j] += alpha * A[i][k] * B[k][j];
```

This code has:

- inner loop marked for pipelining, target is 1
- but a loop-carried dependence
- Vivado finally uses II=6
Loop Pipelining [2/3]

Example (dgemm)

```
for (i = 0; i < ni; i++)
    for (k = 0; k < nk; k++)
        #pragma AP pipeline II=1
        for (j = 0; j < nj; ++j)
            C[i][j] += alpha * A[i][k] * B[k][j];
```

This code has:
- inner loop marked for pipelining, target is 1
- no loop-carried dependence
- Vivado finally uses II=1, a **6x speedup**
Loop Pipelining [3/3]

Loop pipelining in our work:

- Critical performance impact on loop-dominated codes
- We focus on pipelining inner loops only
  - Each inner loop is marked for pipelining
- Our goal: reach II=1 through loop transformations
  - Parallelization (affine scheduling and ISS)
  - Split loops with resource conflicts into multiple loops
Reminder: Tiling + Parallelization

First scheme: “Pluto” plus vectorization-like transfo.

1. Schedule/transform the code for maximal locality + tilability
2. Move one of the parallel dimension inner-most
   ▶ integrated in pluto
   ▶ complemented by a post-pass to perform loop permutation
3. Implemented in PolyOpt/HLS [FPGA’13]

What’s special for FPGAs?

▶ inner loop parallelization is NOT vectorization (simpler problem)
▶ trade-off latency vs. resource
   ▶ Tile size drives the (scarce!) on-chip BRAM usage
   ▶ Resource sharing happens when statements are fused
   ▶ Conservative scheduling: a single slow iteration slows the whole loop
### How Good is This Approach?

<table>
<thead>
<tr>
<th>Bmk.</th>
<th>Description</th>
<th>Version</th>
<th>II</th>
<th>Cycles</th>
<th>CP(ns)</th>
<th>LUT</th>
<th>FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>2mm</td>
<td>Matrix-multiply $D = \alpha A B C + \beta D$</td>
<td>Orig</td>
<td>5</td>
<td>21512194</td>
<td>7.981</td>
<td>1612</td>
<td>1410</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>1</td>
<td>8335874</td>
<td>7.612</td>
<td>1782</td>
<td>1510</td>
</tr>
<tr>
<td>3mm</td>
<td>Matrix-multiply $G = (A B) (C D)$</td>
<td>Orig</td>
<td>5</td>
<td>31948803</td>
<td>8.174</td>
<td>1600</td>
<td>1552</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>1</td>
<td>636371</td>
<td>8.908</td>
<td>2580</td>
<td>2371</td>
</tr>
<tr>
<td>atax</td>
<td>Matrix Transpose and Vector Mult</td>
<td>Orig</td>
<td>5</td>
<td>1511502</td>
<td>8.257</td>
<td>1385</td>
<td>1093</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>1</td>
<td>531852</td>
<td>7.726</td>
<td>1488</td>
<td>1174</td>
</tr>
<tr>
<td>bictg</td>
<td>Kernel of BiCGStab Linear Solver</td>
<td>Orig</td>
<td>5</td>
<td>1255502</td>
<td>8.176</td>
<td>1438</td>
<td>1158</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>1</td>
<td>53185</td>
<td>7.763</td>
<td>1606</td>
<td>1428</td>
</tr>
<tr>
<td>doitgen</td>
<td>Multiresolution Analysis Kernel</td>
<td>Orig</td>
<td>5</td>
<td>5607425</td>
<td>7.828</td>
<td>1126</td>
<td>1024</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>1</td>
<td>1114331</td>
<td>7.659</td>
<td>1769</td>
<td>1776</td>
</tr>
<tr>
<td>gemm</td>
<td>Matrix-multiply $C = \alpha A B + \beta C$</td>
<td>Orig</td>
<td>6</td>
<td>12582925</td>
<td>7.701</td>
<td>1225</td>
<td>1089</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>1</td>
<td>2124418</td>
<td>8.062</td>
<td>1783</td>
<td>1753</td>
</tr>
<tr>
<td>gemver</td>
<td>Vector Mult. and Matrix Addition</td>
<td>Orig</td>
<td>5</td>
<td>3250551</td>
<td>7.902</td>
<td>2778</td>
<td>2427</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>1</td>
<td>555991</td>
<td>7.791</td>
<td>3733</td>
<td>3656</td>
</tr>
<tr>
<td>gesummv</td>
<td>Scalar, Vector and Matrix Mult</td>
<td>Orig</td>
<td>5</td>
<td>1260501</td>
<td>7.705</td>
<td>1652</td>
<td>1541</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>1</td>
<td>532737</td>
<td>7.705</td>
<td>1652</td>
<td>1541</td>
</tr>
<tr>
<td>mvt</td>
<td>Matrix Vector Product and Transpose</td>
<td>Orig</td>
<td>6</td>
<td>3000016</td>
<td>7.496</td>
<td>1371</td>
<td>1108</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>1</td>
<td>265361</td>
<td>7.573</td>
<td>1897</td>
<td>1890</td>
</tr>
<tr>
<td>syrk</td>
<td>Symmetric rank-k operations</td>
<td>Orig</td>
<td>6</td>
<td>12599316</td>
<td>7.808</td>
<td>1397</td>
<td>1217</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>1</td>
<td>2124418</td>
<td>8.028</td>
<td>1784</td>
<td>1793</td>
</tr>
<tr>
<td>syr2k</td>
<td>Symmetric rank-2k operations</td>
<td>Orig</td>
<td>10</td>
<td>20987924</td>
<td>8.123</td>
<td>1675</td>
<td>1415</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>1</td>
<td>2126978</td>
<td>7.982</td>
<td>3055</td>
<td>3069</td>
</tr>
</tbody>
</table>
## Room for Improvement

<table>
<thead>
<tr>
<th>Bmk.</th>
<th>Description</th>
<th>Version</th>
<th>II</th>
<th>Cycles</th>
<th>CP(ns)</th>
<th>LUT</th>
<th>FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>floyd-walshall</td>
<td>Finding Shortest Paths in a Graph</td>
<td>Orig</td>
<td>8</td>
<td>16777218</td>
<td>5.827</td>
<td>1085</td>
<td>791</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>8</td>
<td>16980993</td>
<td>5.889</td>
<td>1182</td>
<td>852</td>
</tr>
<tr>
<td>trmm</td>
<td>Triangular matrix-multiply</td>
<td>Orig</td>
<td>5</td>
<td>5642753</td>
<td>7.398</td>
<td>1387</td>
<td>1229</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>5</td>
<td>3913057</td>
<td>7.418</td>
<td>2160</td>
<td>1964</td>
</tr>
<tr>
<td>trisolv</td>
<td>Triangular Solver</td>
<td>Orig</td>
<td>5</td>
<td>637001</td>
<td>9.091</td>
<td>4418</td>
<td>2962</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>2</td>
<td>266002</td>
<td>9.035</td>
<td>4445</td>
<td>2992</td>
</tr>
</tbody>
</table>
A Detour to Vivado HLS

- Vivado HLS is a compiler :-)  
  - Very powerful, but fragile  
  - Limited support for high-level optimizations  
  - Conservative dependence/resource analysis  
  - Excellent report on optimizations attempted

- Our goal: transform the code to eliminate the reason for failing to meet II=1, and pass information to Vivado  
  - Pragma for pipelining, with target II  
  - Pragma for lack of data dependence  
  - Pragma for Array Partitioning  
  - But no pragma for lack of resource conflict!
Exposing Inner Parallel Loops

Fact: for many affine benchmarks, we can expose one parallel inner loop with affine scheduling

Fact: for some benchmarks partial and non-uniform dependences make our tool fail

Proposed solution:
  Goal: expose parallel inner loops for pipelining
  => develop a customized algorithm using scheduling+ISS
  Make our life “simple” by focusing only the problems observed
Proposed Algorithm

\textbf{DependenceSplit:}
\textbf{Input:}
\hspace{1cm} l: Polyhedral loop nest (SCoP)
\textbf{Output:}
\hspace{1cm} l: in-place modification of l

1. \( D \leftarrow \text{getAllDepsBetweenStatementsInLoop}(l) \)
2. \( D \leftarrow \text{removeAllLoopIndependentDeps}(D, \ l) \)
3. \( parts \leftarrow \{\} \)
4. \textbf{foreach} dependence polyhedron \( D_{x,y} \in D \) \textbf{do}
5. \hspace{1cm} \( D_y \leftarrow \text{getTargetIterSet}(D_{x,y}) \cap D_l \)
6. \hspace{1cm} \textbf{if} \( |D_y| < |D_l| \) \textbf{then}
7. \hspace{2cm} \( parts \leftarrow parts \cup \{D_y\} \)
8. \hspace{1cm} \textbf{else}
9. \hspace{2cm} \textbf{continue}
10. \hspace{1cm} \textbf{end if}
11. \textbf{end do}
12. \( l' \leftarrow \text{split}(l, \ parts) \)
13. \textbf{if} \( \text{sinkParallelLoops}(l') \neq \text{true} \)
14. \hspace{1cm} \text{or} \( \text{parentLoop}(l) = \text{null} \) \textbf{then}
15. \hspace{2cm} \( l \leftarrow l' \)
16. \hspace{2cm} \textbf{return}
17. \textbf{else}
18. \hspace{1cm} \text{DependenceSplit}(\text{parentLoop}(l))
19. \textbf{end if}

- Works from inner-most to outer-most level
- Always legal (split does not change exec. order)
- Split can re-merge loops
# Some Results and Comments

<table>
<thead>
<tr>
<th>Bmk.</th>
<th>Description</th>
<th>Version</th>
<th>II</th>
<th>Cycles</th>
<th>CP(ns)</th>
<th>LUT</th>
<th>FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>floyd-walshall</td>
<td>Finding Shortest Paths in a Graph</td>
<td>Orig</td>
<td>8</td>
<td>16777218</td>
<td>5.827</td>
<td>1085</td>
<td>791</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>8</td>
<td>16980993</td>
<td>5.889</td>
<td>1182</td>
<td>852</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ISS-Dep</td>
<td>2</td>
<td>4407041</td>
<td>5.645</td>
<td>1435</td>
<td>1481</td>
</tr>
<tr>
<td>trmm</td>
<td>Triangular matrix-multiply</td>
<td>Orig</td>
<td>5</td>
<td>5642753</td>
<td>7.398</td>
<td>1387</td>
<td>1229</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>5</td>
<td>3913057</td>
<td>7.418</td>
<td>2160</td>
<td>1964</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ISS-Dep</td>
<td>2</td>
<td>2101106</td>
<td>7.696</td>
<td>1374</td>
<td>1500</td>
</tr>
</tbody>
</table>

- Useful for only two cases in our experiments
- Severe trade-off in resource usage (split increases resource)
- ISS should be used with caution, only when needed
- Parallelism exposure is needed for next stage
Where Is My II=1?

- For 4 benchmarks, still II=2
- Reason (as per Vivado): memory port conflict
  - Two accesses to the same array/bank in the same cycle
  - Must wait 2 cycles before starting the next loop iteration

- A careful manual analysis showed:
  - not all loop iterations have a conflict, only some
  - it is often possible to split the iterations in two sets: one “conflict-free” and another for the rest
Memory Port Conflict

- Rationale: memory port conflicts usually do not occur between each loop iteration, but only between a subset of them
  - when accessing the same banks: \( A[i], A[i+4], A[i+8], \ldots \) if we have four banks

**Definition (Bank conflict)**

Given two memory addresses \( x \) and \( y \) (assuming cyclic mapping of addresses to banks using the \( \% \) function). They access the same bank iff:

\[
x \% B = y \% B
\]

with \( B \) the number of banks. It can be equivalently written:

\[
\exists k \in \mathbb{Z}, \quad x - y = B \times k
\]
Bank Conflict Set

Definition (Bank conflict set)

Given an inner-most loop \( l \), whose iteration domain is \( \mathcal{D}_l \), and two references \( F^1_A \) and \( F^2_A \) accessing the same array \( A \). The bank conflict set \( C_{F^1_A,F^2_A} \subseteq \mathcal{D}_l \) is:

\[
C_{F^1_A,F^2_A} : \{ \bar{x}_l \in \mathcal{D}_l | \exists k \in \mathbb{Z}, \text{lin} (F^1_A) - \text{lin} (F^2_A) = k \cdot B \}
\]

With \( \text{lin}(x) \) the linearized form of \( x \).
**Proposed Algorithm**

```
ResourceSplit:
Input:
  l: inner-most parallel affine loop
  sz: size of arrays in l
  B: number of banks available
Output:
  l: in-place modification of l

1. lst ← ∅
2. all ← Ø
3. foreach array A ∈ l do
   4. foreach distinct pair of references F^i_A, F^j_A ∈ l do
      5. C^i,j_A ← buildConflictsSet(B, sizes(A), F^i_A, F^j_A)
      6. lst ← lst ∪ {C^i,j_A}
      7. all ← all ∪ C^i,j_A
   end do
4. rem ← D_l \ all
5. lst ← {lst, rem}
6. l' ← codegen(lst)
7. l ← finalize(l, l')
```

- Works only on parallel inner loops (always legal)
- Codegen is ISL codegen
- Finalize can re-merge loops
Some Discussions...

<table>
<thead>
<tr>
<th>Bmk.</th>
<th>Description</th>
<th>Version</th>
<th>II</th>
<th>Cycles</th>
<th>CP(ns)</th>
<th>LUT</th>
<th>FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>floyd-walshall</td>
<td>Finding Shortest Paths in a Graph</td>
<td>Orig</td>
<td>8</td>
<td>16777218</td>
<td>5.827</td>
<td>1085</td>
<td>791</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>8</td>
<td>16980993</td>
<td>5.889</td>
<td>1182</td>
<td>852</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ISS-Dep</td>
<td>2</td>
<td>4407041</td>
<td>5.645</td>
<td>1435</td>
<td>1481</td>
</tr>
<tr>
<td>trmm</td>
<td>Triangular matrix-multiply</td>
<td>Orig</td>
<td>5</td>
<td>5642753</td>
<td>7.398</td>
<td>1387</td>
<td>1229</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>5</td>
<td>3913057</td>
<td>7.418</td>
<td>2160</td>
<td>1964</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ISS-Dep</td>
<td>2</td>
<td>2101106</td>
<td>7.696</td>
<td>1374</td>
<td>1500</td>
</tr>
<tr>
<td>trisolv</td>
<td>Triangular Solver</td>
<td>Orig</td>
<td>5</td>
<td>637001</td>
<td>9.091</td>
<td>4418</td>
<td>2962</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Affine</td>
<td>2</td>
<td>266002</td>
<td>9.035</td>
<td>4445</td>
<td>2992</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ISS-Res</td>
<td>1.5</td>
<td>219002</td>
<td>8.799</td>
<td>5360</td>
<td>3575</td>
</tr>
</tbody>
</table>

- ISS (dep or res) useful for three benchmarks
- Big resource increase! But good latency improv.
- Many open questions left, comparison missing
- Interesting “simple” approach: separate out problematic iterations
Conclusions and Future Work

Take-home message:

- Vivado HLS is fragile, lots of room for improvement
- Index-Set Splitting can be very useful also for HLS
- Memory port conflict may be solved with simple splitting
- Trade-off latency vs. resource needs to be considered
- Better / more integrated solution should be designed
- Useful only in special cases (but really useful!)

Future work:

- Extensive comparison with other approaches (array partitioning, ...)
- Remove restrictions of the algorithms (legality)
- Single unified problem for throughput optimization