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

Center for Domain Specific Computing

UCLA

(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, 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**

- 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)**

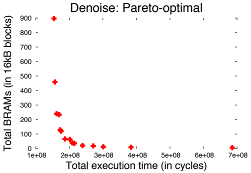
- HLS/ESL tools have made great progresses (ex: AutoESL/Vivado)
- But still extensive manual effort needed for best performance

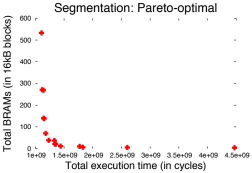
⇒ **Our solution: complete HLS-focused source-to-source compiler**

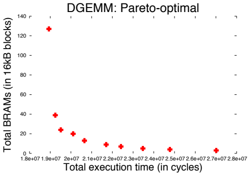
- Numerous previous research work on C-to-FPGA (PICO, DEFACTO, MMAlpha, etc.) and data reuse optimizations
- But (strong) limitations in applicability / transformations supported / performance achieved

⇒ **Our solution: unleash the power of the polyhedral framework (loop transfo., comm. scheduling, etc.)**

Performance Results







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

- 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: 300HMz

Context of This Work

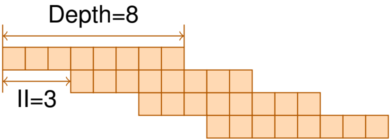
How to get good throughput?

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



- Total cycles: (LoopTripCount - 1) * II + 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)

```
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?

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

Room for Improvement

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

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

```
DependenceSplit:
Input:
    l: Polyhedral loop nest (SCoP)
Output:
    l: in-place modification of l

    D ← getAllDepsBetweenStatementsInLoop(l)
    D ← removeAllLoopIndependentDeps(D, l)
    parts ← {}
    foreach dependence polyhedron 𝒟_{x,y} ∈ D do
        𝒟_y ← getTargetIterSet(𝒟_{x,y}) ∩ 𝒟_l
        if |𝒟_y| < |𝒟_l| then
            parts ← parts ∪ {𝒟_y}
        else
            continue
        end if
    end do
    l' ← split(l, parts)
    if sinkParallelLoops(l') ≠ true
      .or. parentLoop(l) = null then
        l ← l'
        return
    else
        DependenceSplit(parentLoop(l))
    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

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

- 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]`, ... if we have four banks

Definition (Bank conflict)

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

$$x \,\%\, B \quad = \quad y \,\%\, B \tag{1}$$

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

$$\exists k \in \mathbb{Z}, \qquad x - y = B * k$$

Bank Conflict Set

Definition (Bank conflict set)

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

$$\mathcal{C}_{F_A^1,F_A^2} : \left\{ \vec{x}_l \in \mathcal{D}_l \mid \exists k \in \mathbb{Z},\ lin\left(F_A^1\right) - lin\left(F_A^2\right) = k * B \right\}$$

With $lin(x)$ 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

$lst \leftarrow \{\}$
$all \leftarrow \emptyset$
foreach array $A \in l$ **do**
foreach distinct pair of references $F_A^i, F_A^j \in l$ **do**
$C_{F_A^i, F_A^j} \leftarrow$ buildConflictSet($B, sizes(A), F_A^1, F_A^2, \mathcal{D}_l$)
$lst \leftarrow lst \bigcup \{C_{F_A^1, F_A^2}\}$
$all \leftarrow all \cup C_{F_A^1, F_A^2}$
end do
end do
$rem \leftarrow \mathcal{D}_l \setminus all$
$lst \leftarrow \{ lst, rem\}$
$l' \leftarrow$ codegen(lst)
$l \leftarrow$ finalize(l, l')

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

Some Discussions...

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

- 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