

Optimizing SDRAM Bandwidth for Custom FPGA Loop Accelerators

Samuel Bayliss and George A. Constantinides

February 24, 2012

Introduction

- ▶ Many algorithms suitable for FPGA acceleration require off-chip memory access
- ▶ Desirable to consider the cost of off-chip communication up-front in the design process
- ▶ Using the Polyhedral Model, we can:
 - ▶ Analyse communication cost at compile time
 - ▶ Synthesize an efficient SDRAM memory controller
 - ▶ Exploit **Data Reuse** and Transaction Reordering

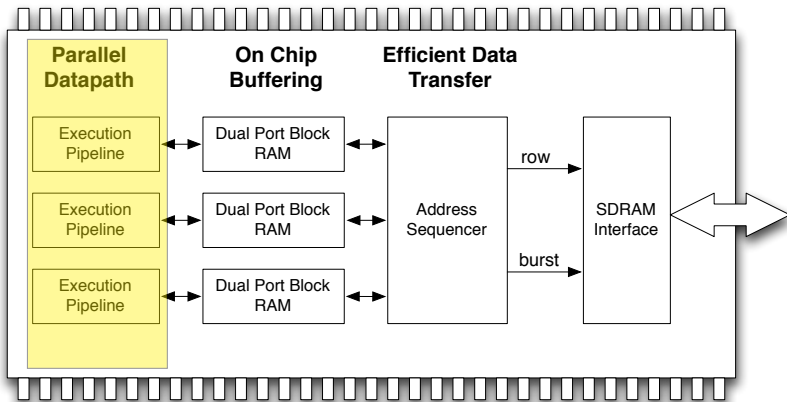


Introduction

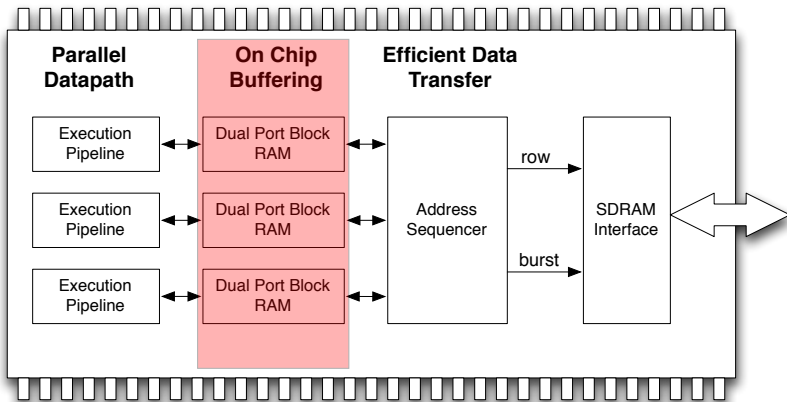
- ▶ Many algorithms suitable for FPGA acceleration require off-chip memory access
- ▶ Desirable to consider the cost of off-chip communication up-front in the design process
- ▶ Using the Polyhedral Model, we can:
 - ▶ Analyse communication cost at compile time
 - ▶ Synthesize an efficient SDRAM memory controller
 - ▶ Exploit **Data Reuse** and **Transaction Reordering**



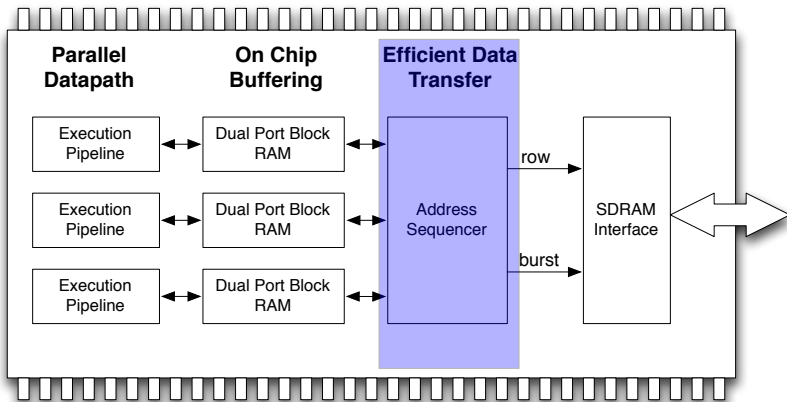
Proposed Computation Platform



Proposed Computation Platform



Proposed Computation Platform



SDRAM Memory Characteristics

- ▶ **Advantage** : SDRAM is **cheap**, high capacity, commodity memory
- ▶ **Disadvantage** : Internal device architecture means high latency (20-30 clock cycles in FPGA)
- ▶ Physical device structure imposes timing constraints
 - ▶ Explicit 'activation' of a row before data is read from it
 - ▶ Explicit 'precharge' of a row before another row is 'activated'
- ▶ Significant bandwidth difference improvements possible when reordering external memory transactions
 - ▶ Typically $> 5\times$ bandwidth difference between optimal and worst-case performance



Idea

- ▶ With predictable patterns of memory access
 - ▶ Can **Prefetch** data
 - ▶ SDRAM latency needn't impact bandwidth
 - ▶ Can **Reuse** data in on-chip memory
 - ▶ Reduce number of external memory transactions
 - ▶ Can **Reorder** external memory transactions
 - ▶ Reduce the number of SDRAM row-swaps: increase bandwidth

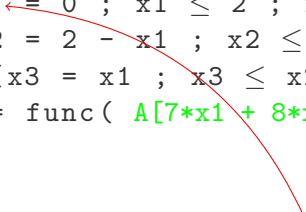
Requirement

- ▶ Need a mathematical framework to analyze memory patterns



Nested Loops in the Polyhedral Model

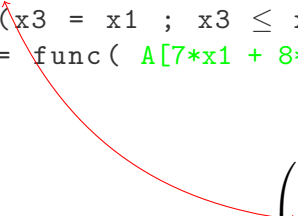
```
int A[56]; int y;  
for (x1 = 0 ; x1 ≤ 2 ; x1++ )  
  for (x2 = 2 - x1 ; x2 ≤ 2 ; x2++ )  
    for (x3 = x1 ; x3 ≤ x2 ; x3++ )  
      y = func( A[7*x1 + 8*x2 + 9*x3] ) ;
```


$$\begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix}$$



Nested Loops in the Polyhedral Model

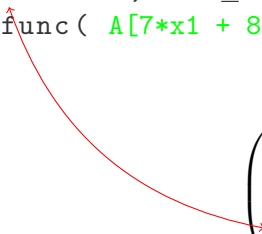
```
int A[56]; int y;  
for (x1 = 0 ; x1 ≤ 2 ; x1++ )  
  for (x2 = 2 - x1 ; x2 ≤ 2 ; x2++ )  
    for (x3 = x1 ; x3 ≤ x2 ; x3++ )  
      y = func( A[7*x1 + 8*x2 + 9*x3] ) ;
```


$$\begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix}$$



Nested Loops in the Polyhedral Model

```
int A[56]; int y;  
for (x1 = 0 ; x1 ≤ 2 ; x1++ )  
  for (x2 = 2 - x1 ; x2 ≤ 2 ; x2++ )  
    for (x3 = x1 ; x3 ≤ x2 ; x3++ )  
      y = func( A[7*x1 + 8*x2 + 9*x3] ) ;
```


$$\begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix}$$



Nested Loops in the Polyhedral Model

```
int A[56]; int y;  
for (x1 = 0 ; x1 ≤ 2 ; x1++ )  
  for (x2 = 2 - x1 ; x2 ≤ 2 ; x2++ )  
    for (x3 = x1 ; x3 ≤ x2 ; x3++ )  
      y = func( A[7*x1 + 8*x2 + 9*x3] ) ;
```

$$\begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ -1 & -1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 2 \\ -2 \\ 2 \\ 0 \\ 0 \end{pmatrix}$$



Nested Loops in the Polyhedral Model

```
int A[56]; int y; int buf [5] []; param t=1
for (x1 = 0 ; x1 ≤ 2 ; x1++ ) ← fill(&buf);
    for (x2 = 2 - x1 ; x2 ≤ 2 ; x2++ )
        for (x3 = x1 ; x3 ≤ x2 ; x3++ )
            y = func( buf[...][...] ) ;
```

$$\begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ -1 & -1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 2 \\ -2 \\ 2 \\ 0 \\ 0 \end{pmatrix}$$



Nested Loops in the Polyhedral Model

```
int A[56]; int y; int buf [3] [];  
for (x1 = 0 ; x1 ≤ 2 ; x1++ )  
    for (x2 = 2 - x1 ; x2 ≤ 2 ; x2++ )  
        for (x3 = x1 ; x3 ≤ x2 ; x3++ )  
            y = func( buf[...] [...] ) ;
```

param t=2

fill(&buf);

$$\begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ -1 & -1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 2 \\ -2 \\ 2 \\ 0 \\ 0 \end{pmatrix}$$



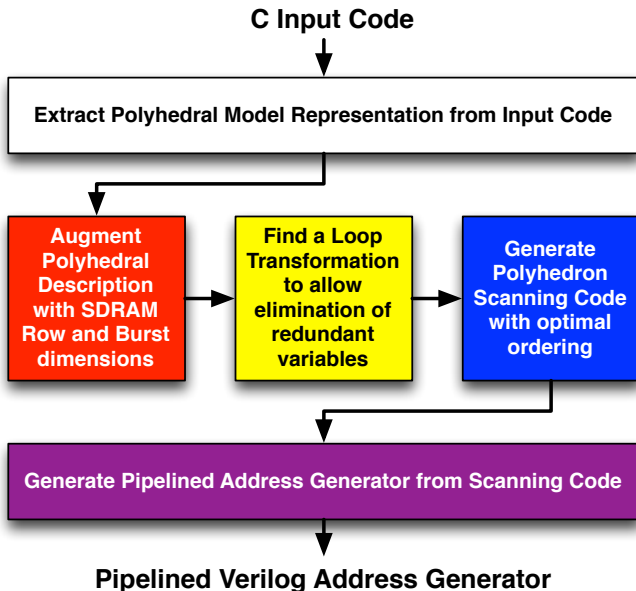
Nested Loops in the Polyhedral Model

```
int A[56]; int y; int buf [3] [];  
for (x1 = 0 ; x1 ≤ 2 ; x1++ )  
  for (x2 = 2 - x1 ; x2 ≤ 2 ; x2++ )  
    for (x3 = x1 ; x3 ≤ x2 ; x3++ ) ← fill(&buf);  
      y = func( buf[...] ] [...] ) ;
```

param t=3

$$\begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ -1 & -1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 2 \\ -2 \\ 2 \\ 0 \\ 0 \end{pmatrix}$$





Representing SDRAM Parameters Explicitly

- ▶ Memory references are affine functions
 - ▶ e.g. $\mathbf{f}\mathbf{x} + c = 7 \cdot x_1 + 8 \cdot x_2 + 9 \cdot x_3$
 - ▶ Array offset represented by constant (c)
- ▶ Can represent SDRAM rows (r) and bursts (u) exactly
 - ▶ R is num/rows in a SDRAM bank, B is num/bursts in a SDRAM row

$$r = \left\lfloor \frac{\mathbf{f}\mathbf{x} + c}{R} \right\rfloor \quad u = \left\lfloor \frac{\mathbf{f}\mathbf{x} + c - Rr}{B} \right\rfloor$$

$R = 16$ $B = 4$

- ▶ *or* using linear inequalities

$$\mathbf{f}\mathbf{x} + c + R - 1 \leq Rr \leq \mathbf{f}\mathbf{x} + c$$

$$\mathbf{f}\mathbf{x} + c - rR - B + 1 \leq Bu \leq \mathbf{f}\mathbf{x} + c - rR$$



- Augmented matrix captures each loop iteration and its associated SDRAM row (r) and burst (u)

$$\begin{pmatrix} -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 7 & 8 & 9 & -R & 0 \\ -7 & -8 & -9 & R & 0 \\ 7 & 8 & 9 & -R & -B \\ -7 & -8 & -9 & R & B \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \\ r \\ u \end{pmatrix} \leq \begin{pmatrix} 0 \\ 2 \\ -2 \\ 2 \\ 0 \\ 0 \\ 15 \\ 0 \\ 3 \\ 0 \end{pmatrix}$$



Optimization of Array Pre-fetching Code

- ▶ We can reconstruct code in the form of a loop nest which visits every point in the polytope
 - ▶ Use Cloog¹ to create code scanning rows at the outer-most loop level
 - ▶ Create code which 'scans' through rows at the outermost loop level
 - ▶ Generated code has **minimal** number of row-swaps
- ▶ Code still contains redundant accesses
 - ▶ e.g. in example $\mathbf{x} = (0 \ 2 \ 1)$ and $\mathbf{x} = (1 \ 1 \ 1)$ both access row 1, burst 2.
- ▶ Wish to remove those redundant accesses by elimination of variables.

¹*Code Generation in the Polyhedral Model is easier than you think*

Cedric Bastoul et al 2004



Elimination of Variables

- ▶ Use existing results from the operations research field²
 - ▶ Safe conditions for eliminating a variable from an **integer** set
- ▶ New two stage approach:
 - ▶ Find a loop transformation to *maximize* the number of eliminated variables
 - ▶ Eliminate all variables which meet the conditions for safe elimination
- ▶ Corresponds to reduction of the area of generated address sequencer

²*The Elimination of Integer Variables* H.P. Williams 1992



← 5 Variables →

$$\begin{pmatrix} -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 7 & 8 & 9 & -R & 0 \\ -7 & -8 & -9 & R & 0 \\ 7 & 8 & 9 & -R & -B \\ -7 & -8 & -9 & R & B \end{pmatrix}$$

← 3 Variables →

$$\begin{pmatrix} -29 & 12 & 48 \\ 2 & -4 & -1 \\ -1 & 2 & 0 \\ -11 & 16 & 4 \\ -1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & -2 & 0 \end{pmatrix}$$

Using loop
Transformation

$$\begin{pmatrix} -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 1 & 1 & -1 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ -1 & -1 & 9 & -16 & 0 \\ 1 & 1 & -9 & 16 & 0 \\ -1 & -1 & 9 & -16 & -4 \\ 1 & 1 & -9 & 16 & 4 \end{pmatrix}$$

Using safe
variable
elimination
conditions

Code Generation

- ▶ We can take the reduced polyhedron and use existing techniques to produce code to **visit each integer point** within the (reduced) polytope
- ▶ Define an order which generates nested loops with 'row' variable as the outermost loop
 - ▶ Taking the AST produced by Cloog, we generate a fully pipelined address sequencer
 - ▶ Sequencer gives one SDRAM address (row and burst) per cycle
 - ▶ Auto-pipelining allows high frequency operation

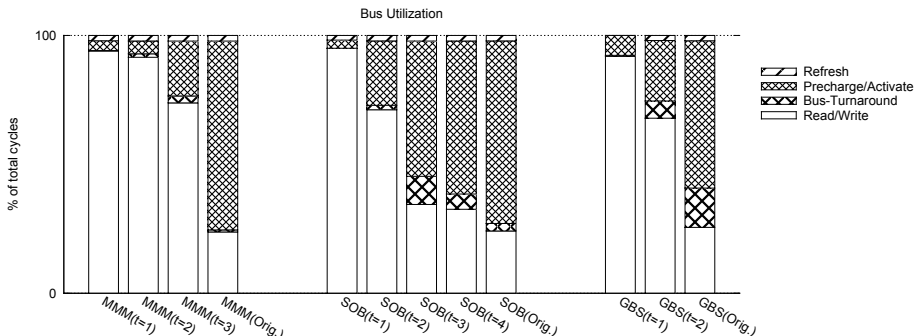


How well do we do?

Benchmark	Level	Read/Write Cycles	Total Cycles
MMM	t=1	5012 (94.00%)	5332
MMM	t=2	66804 (91.54%)	72978
MMM	t=3	590000 (73.86%)	798804
MMM	Orig.	2000004 (23.78%)	8410260
SOB	t=1	8920 (94.93%)	9396
SOB	t=2	77328 (71.18%)	108632
SOB	t=3	180300 (32.43%)	523646
SOB	t=4	442932 (32.58%)	1359474
SOB	Orig.	839236 (24.18%)	3471284
GBS	t=1	1832 (91.88%)	1994
GBS	t=2	13888 (67.90%)	20454
GBS	Orig.	61348 (25.55%)	240106



What was the impact on bandwidth efficiency?



How big were the generated address sequencers?

Benchmark	Level	Req. on-chip mem. words	ALUTs	Regs	Frequency
MMM	t=1	61200	575	764	296 MHz
MMM	t=2	21200	1050	1666	174 MHz
MMM	t=3	416	1346	2098	179 MHz
MMM	Orig.	0	1003	2740	184 MHz
SOB	t=1	11411	592	717	300 MHz
SOB	t=2	579	1551	2251	182 MHz
SOB	t=3	19	1355	1907	144 MHz
SOB	t=4	7	1200	2566	153 MHz
SOB	Orig.	0	1107	3607	148 MHz
GBS	t=1	2772	833	1156	242 MHz
GBS	t=2	288	952	1366	211 MHz
GBS	Orig.	0	804	2263	186 MHz



What is the big picture?

- ▶ We demonstrate up to $4\times$ improvement in bandwidth efficiency through **transaction reordering**
- ▶ Results show a methodology for **automatically** exploring at compile-time, the trade-off between the amount of on-chip memory used to buffer data and the number of off-chip memory transactions issued
- ▶ Tool automatically generates pipelined address sequencers operate at frequencies which can **saturate** the memory interface and a cost of up to $1.4\times$ increase in the LUTs dedicated to address generation



Where do we go from here?

- ▶ Can we match data-path and memory system throughput?
 - ▶ Use the exact knowledge we have on memory access latency to improve datapath resource sharing
- ▶ Can we deliver tight WCET bounds for real-time applications?
 - ▶ Use new mathematical results on integer point counting to tightly bound worst-case execution time for code kernels using external SDRAM



Where do we go from here?

- ▶ Can we match data-path and memory system throughput?
 - ▶ Use the exact knowledge we have on memory access latency to improve datapath resource sharing
- ▶ **Can we deliver tight WCET bounds for real-time applications?**
 - ▶ Use new mathematical results on integer point counting to tightly bound worst-case execution time for code kernels using external SDRAM



Final remarks

Tool Availability

Tool will be available at website (<http://cas.ee.ic.ac.uk/AddrGen>)

Offline Questions

Can email me with any questions s.bayliss08@imperial.ac.uk

Acknowledgments

Many thanks to Sven Verdooleage for his Integer Set Library

