Efficient Matrix Multiplication on FPGAs

Sixiao Zhu, Jiali Jiang, Wenqiang Wang, and Ningyi Xu, MSRA

February 3, 2016

Introduction

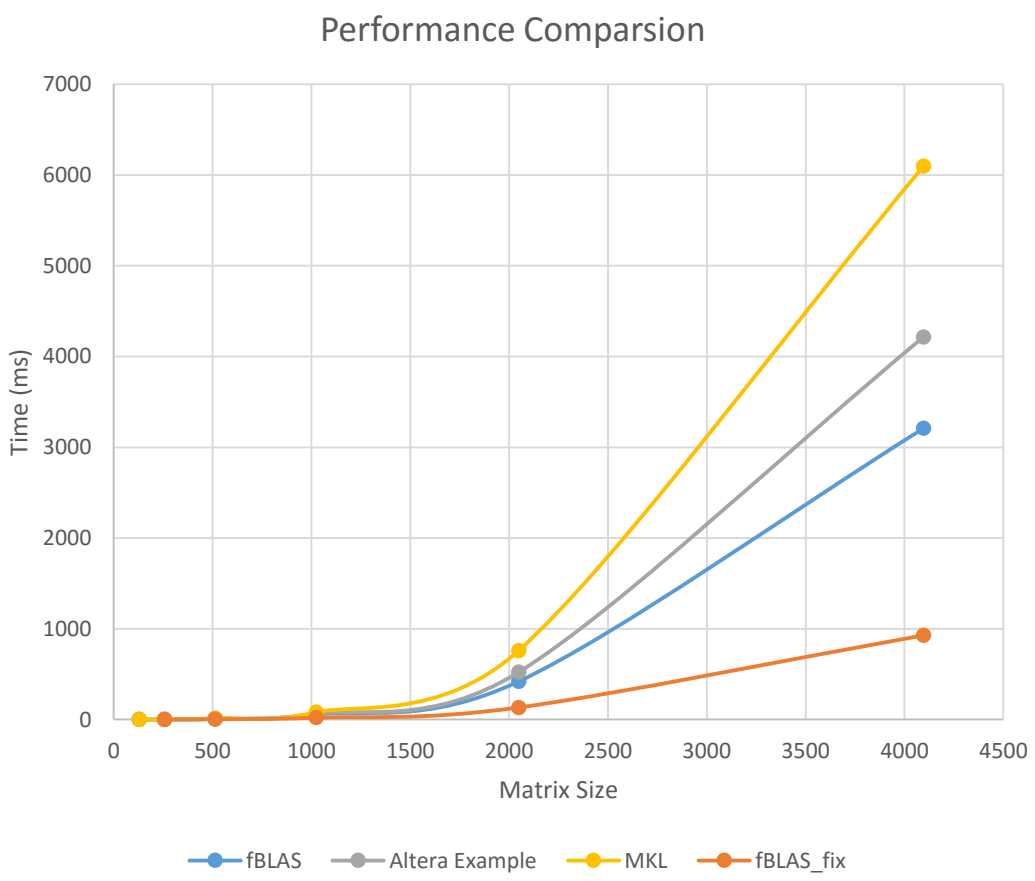
- BLAS (Basic Linear Algebra Subprograms) is fundamental to most computations in scientific applications
  - BLAS Level1: Dot-product, O(N)
  - BLAS Level2: Gaxpy, Matrix*Vector, $O(N^2)$
  - BLAS Level3: GEMM, Matrix*Matrix, $O(N^3)$
- GEMM is important
  - Heavily used in image/speech recognition, and many other machine learning applications
  - Using MM to calculate CNNs has been adopted by many DL libraries (cuDNN, Caffe)
- Goal: a scalable, highly efficient **FPGA BLAS library (fBLAS)** for various users

End User	Programming Language	fBLAS library format
SW programmers who want to use Catapult FPGA	C++/CUDA/OpenCL	cuBLAS-like API and .lib
FPGA programmers	OpenCL	OpenCL source code
RTL designer	Verilog/VHDL/A++	A++ source code/generated RTL

OpenCL-based implementation

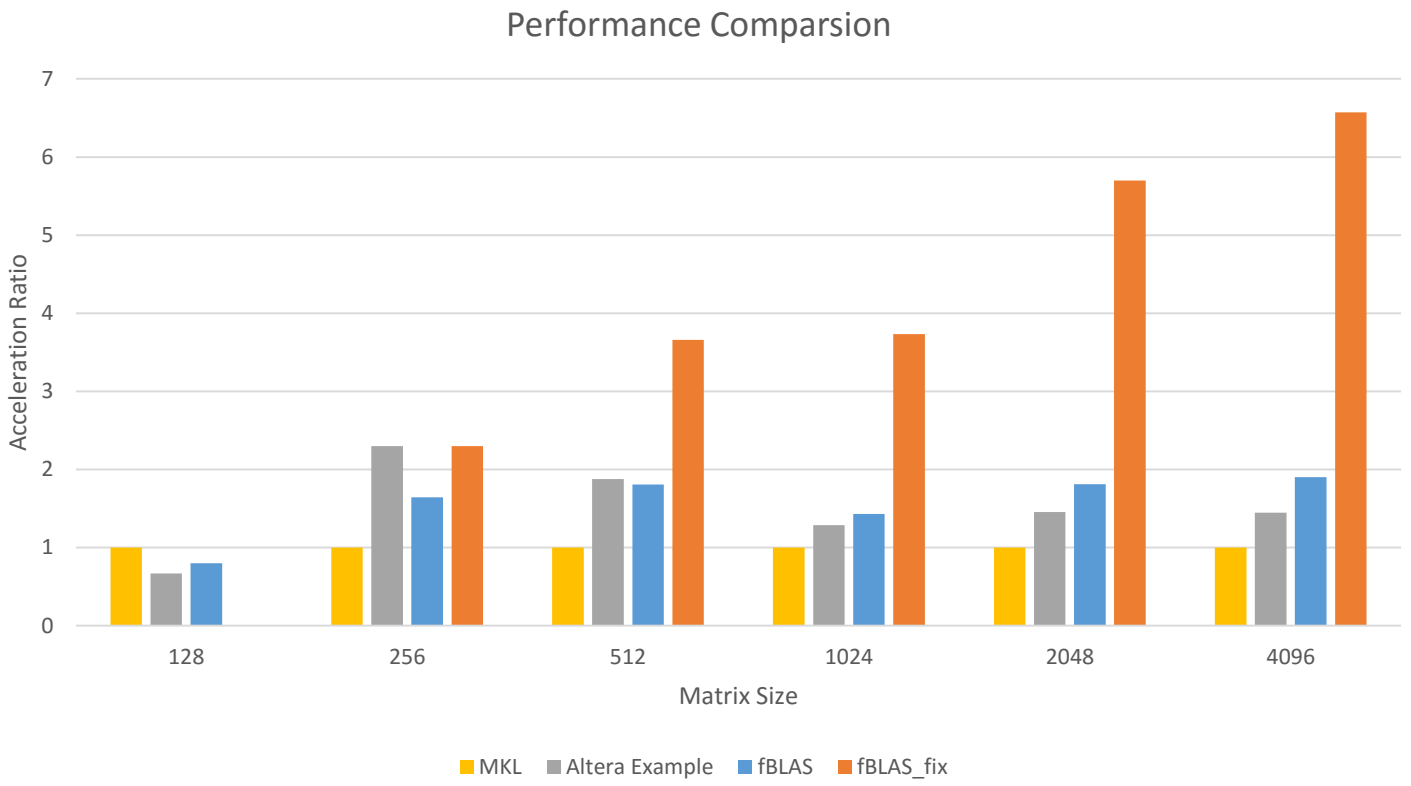
- Approach: single work-item style
- Benefits
  - Easy to use by software programmers (OpenCL APIs)
  - Easy to be integrated to OpenCL-based FPGA projects
  - Scalable (e.g. DSP numbers, matrix size, tile size) and flexible (e.g. floating point -> fixed point)
- Challenge: fine-grained control of generated circuits with OpenCL
- Hardware structure
  - Fully pipelined, tile-based structure
  - Efficient adder tree
  - Ping-pong buffer

OpenCL fBLAS GEMM performance results



- fBLAS/Altera MM
  - Catapult Pikes Peak
  - Altera SDK for OpenCL, 64-Bit Offline Complier, Version 15.1.0 Build 185
  - 42.9 GFLOPS
- MKL
  - Intel Xeon E5620(@2.40GHz) 4 Core 4/8 Threads, Intel compiler version 16.0, MKL version 11.3
  - 22.5 GFLOPS/31.4 GFLOPS

OpenCL fBLAS GEMM performance results



OpenCL implementation bottleneck

	Coding Style	RAM bits (41,246K)	ALMs (172K)	DSPs (1590)	DDR3 BW	Performance (4096x4096)
fBLAS (OpenCL)	Single workitem	11,810K (29 %)	122K (71%)	136 (8.5%)	2GBps (25%)	42.9 GFLOPS
Altera M*M OpenCL	Multiple workitem	10,952K (27%)	166K (96%)	264 (17%)	1.54GBps (18.4%)	33.0 GFLOPS
M*V (by Srinidhi Kestur, John Davis, Eric S. Chung)	Verilog, Xilinx V6-LX240T, 32PE	63%	71%	56%	80%	1.14 GFLOPS

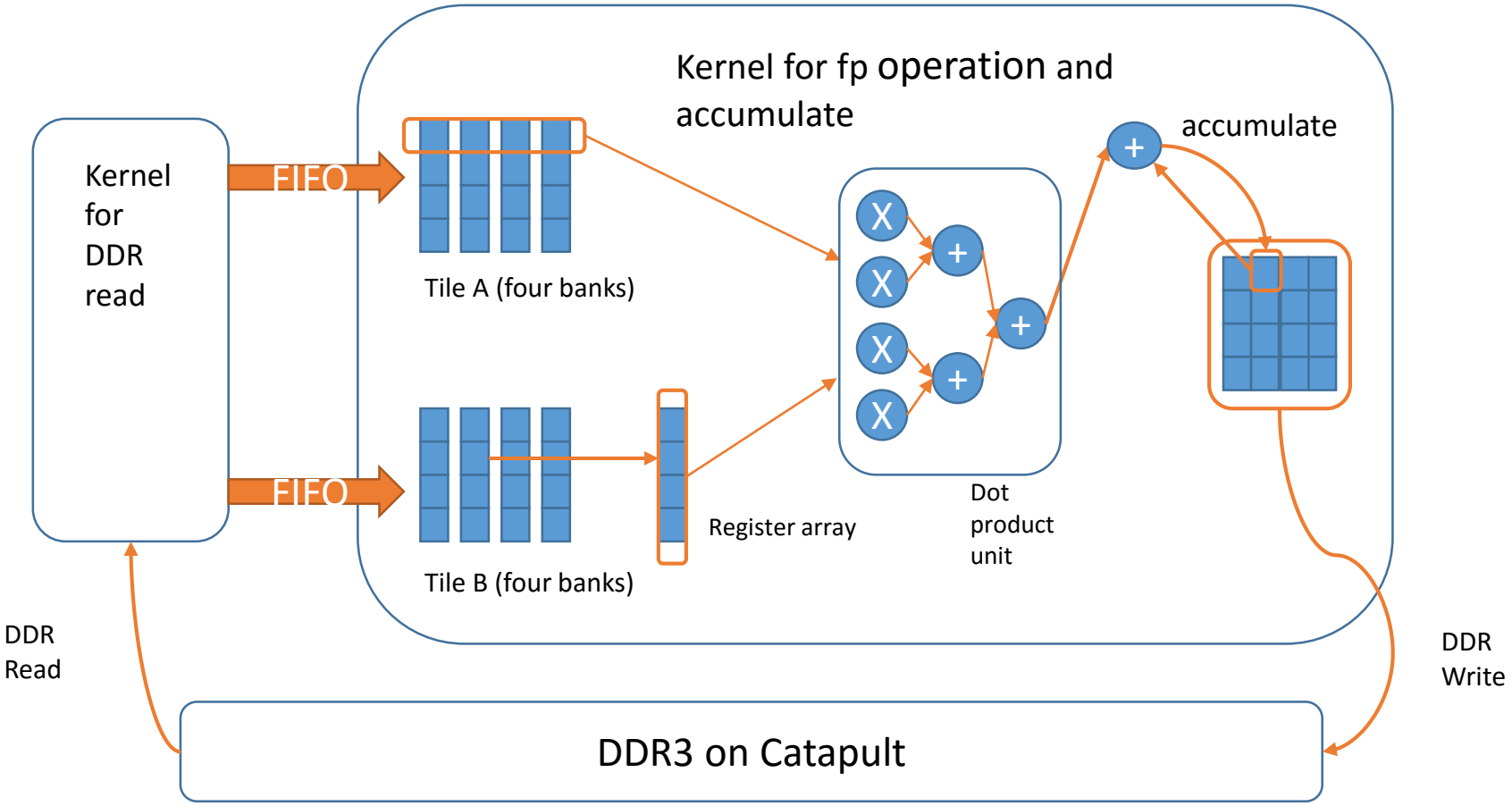
- Bottleneck
  - ALMs: *'reduction operation' in dot-product* (sum up 128 float)
    - Compiler generates human unreadable Verilog code
  - Current version makes use of 128 DSP (total: 1590) because of the ALM limitation, 8.5% DSP uses up 71% logic resource
- Solution(s)
  - OpenCL: more resources (256 DSP version coming soon), optimized reduction operation
  - A++: generates the same Verilog code with AOC
  - Verilog HDL:
    - Utilize all DSPs: 10x in peak GFLOPS, bottleneck will be DDR3 BW
    - Customized floating point unit

Limitations by now: Resource

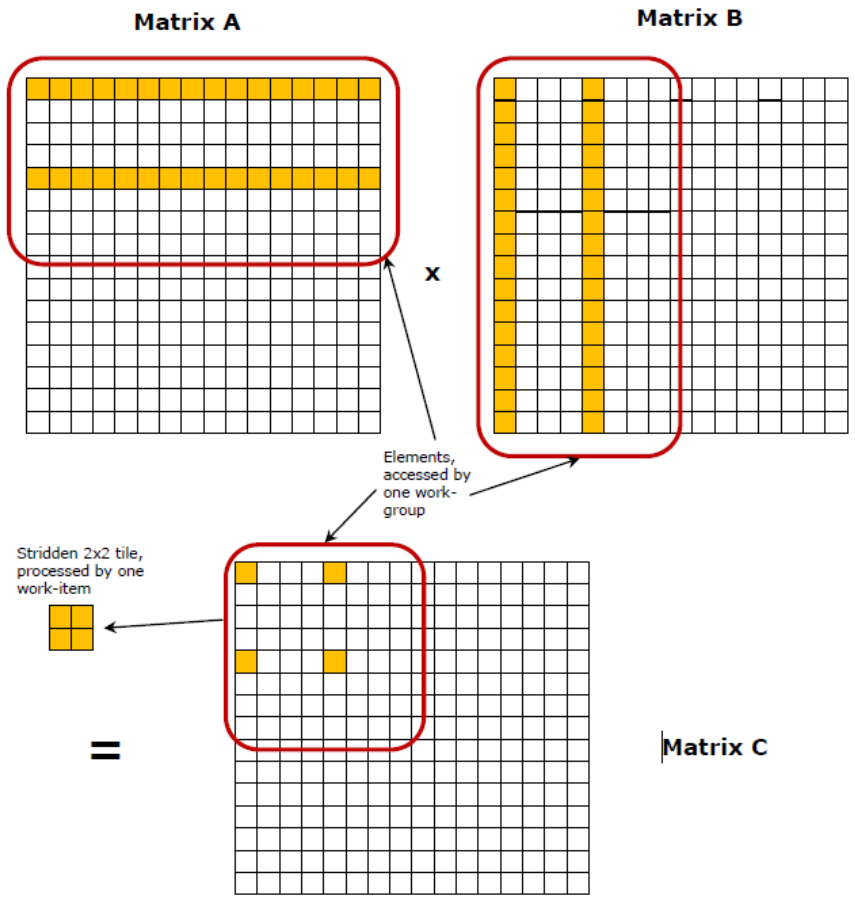
- Resource consumes **heavily** on reducing operation in dot product (128 float points sum up in one cycle).
- Compiler generates human unreadable Verilog code.
- Current version makes use of 128 DSP (totally 1590 on board) because of the ALM limitation, 9% DSP uses up 71% logic resource.

	Coding Style	RAM bits	ALMs	DSPs	Performance (4096x4096)
fBLAS (OpenCL)	Single workitem	11,810K (29 %)	122K (71%)	136 (8.5%)	1.3x
Altera MM OpenCL example	Multiple workitem	10,952K (27%)	166K (96%)	264 (17%)	1

Structure and major techniques



Basic strategy: Do the calculation by tile



Basic strategy: Why tiling?

- The idea of *data locality*.
- In matrix multiplication, data is reused (data is $O(N^2)$, float point operation is $O(N^3)$), thus refetched. We want to fetch data from block RAM rather DDR.
- Suppose we have matrix A: $R \times V$, matrix B: $V \times \mathrm{C}$.
- Without tiling: $N_{DDR\_read} = R \times C \times V$
- With tiling: $N_{DDR\_read} = \frac{R \times C \times V}{SIZE_{tile}}$

Further improve DDR Read with Asynchronous IO

- Decompose IO and computing: kernel 1 for DDR reading and kernel 2 for computing.
- K1 and K2 are connected with a deep FIFO, with push and pop ports width ratio 1:4.
- K1 reads a tile from DDR, pushes it into the deep FIFO with **low bandwidth**.
- K2 pops the tile from the FIFO, with **high bandwidth**, and compute.

- Resulting: K2 is computing current tile, while K1 is pushing into the next tile. In fact a implicitly Ping-Pong.
- Reduces 75% IO time consume.

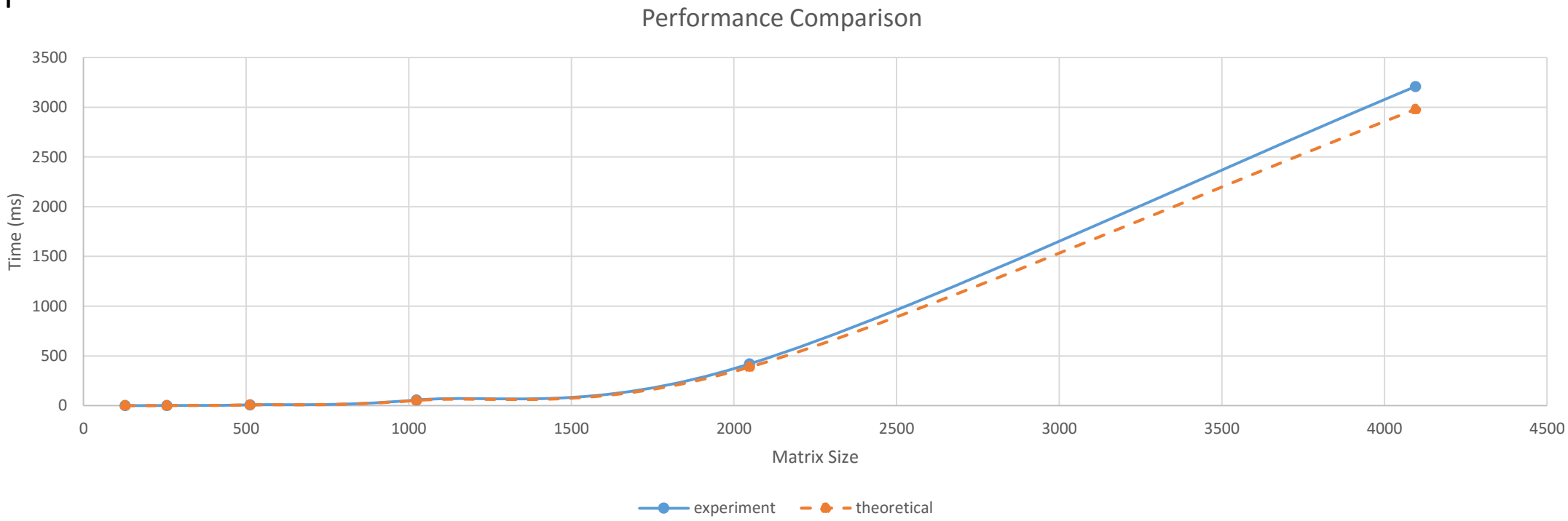
A simple performance analysis

- 128 DSP, tile size 64 float, FIFO port ratio 1:4 (64 float per cycle at computing kernel side)
- $Cycles_{read\_tile} = 2 \times \frac{SIZE_{tile} \times SIZE_{tile}}{SIZE_{tile}} = 64 \times 2$
- $Cycles_{tile\_multiply} = \frac{SIZE_{tile} \times SIZE_{tile} \times SIZE_{tile}}{N_{DSP}} = 64 \times 32$

- IO overhead $1/16$, bottleneck lays on number of computing units, (Or practically, board resource).

A simple performance analysis

- The computing intensive part is fully pipelined, performance is highly predictable.



Optimization detail: Improve local bandwidth

- 128 DSP means on chip storage system must feed 128 + 128 floats to the multipliers.
- How to provide the high local bandwidth? Banking + Dual port reading.
- Banking: One bank for one column, for reading one row per cycle from FIFO to local RAM.

```
#pragma unroll
for (int c = 0; c < TILE_SIZE; ++c)
    tile_b[r][c] = tmp[c];
```

- Dual port reading: Make compiler infer dual port RAM.

```
float2 tmp  = (float2)(tile_b[r][c], tile_b[r + 1][c]);
```

Optimization detail: Asynchronously transpose

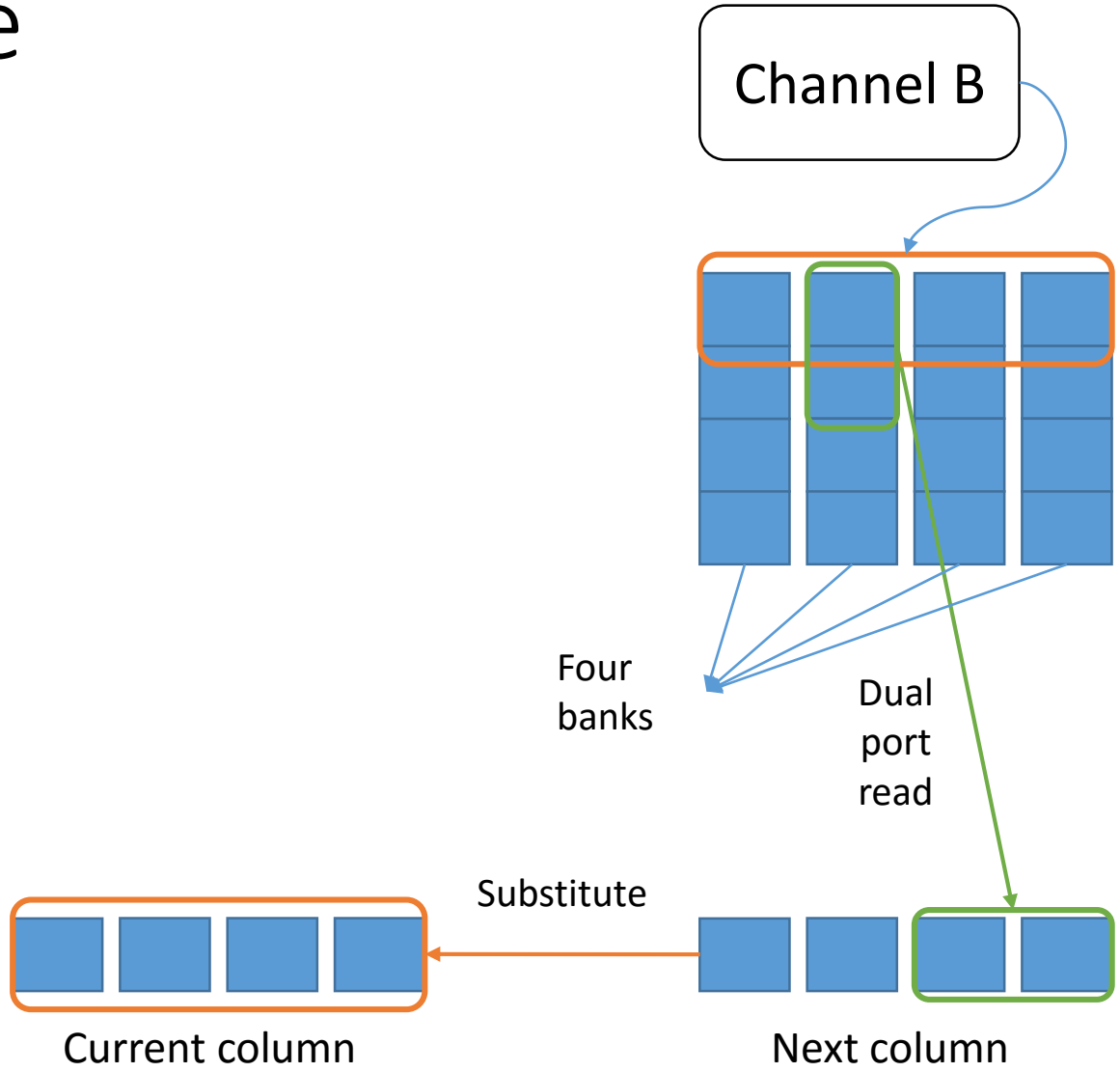
- Matrix A and B are stored *row major*, but in matrix multiplication, rows of A dot product columns of B.
- Tile B need to be transposed, we want O(1) transpose.

- Difficulty: tile B is banked stored, one bank per column, for storing one row per cycle. So we can not fetch an entire column in one cycle.

Optimization detail: Asynchronously transpose

- How to solve this?
- We use this computing order: We fetch out one column of tile B to dot product all rows of tile A.
- So every fetched column of B will be reused for $^{64\times64}/_{128DSP} = 32$ times.
- So, during this 32-cycle computing, we prepare the next column of tile B with a low-bandwidth manner, that is, fetching elements of the next B column in the same bank.

Optimization detail: Asynchronously transpose



Optimization detail: Stall free pipeline at critical region

- Advantage of HLS: Auto pipelining.
- Consider the following code, this is the compute intensive portion of the MXM.

```
for (int i = 0; i < TILE_SIZE; ++i) {
    for (int j = 0; j < TILE_SIZE; ++j) { // one B_column for all A_row
        result_i_j = dot (tile_A_row_j, tile_B_column_i);
        accu[i][j] += result_i_j; // [1] may exists write after read problem
    }
}
```

- At the compiler's perspective, Resign [1] may exists data dependency risk, so it generates hardware that executes serially (stall the pipeline) at this region.
- Yet at the programmer's perspective, there is no such dependency cause we know the logic.

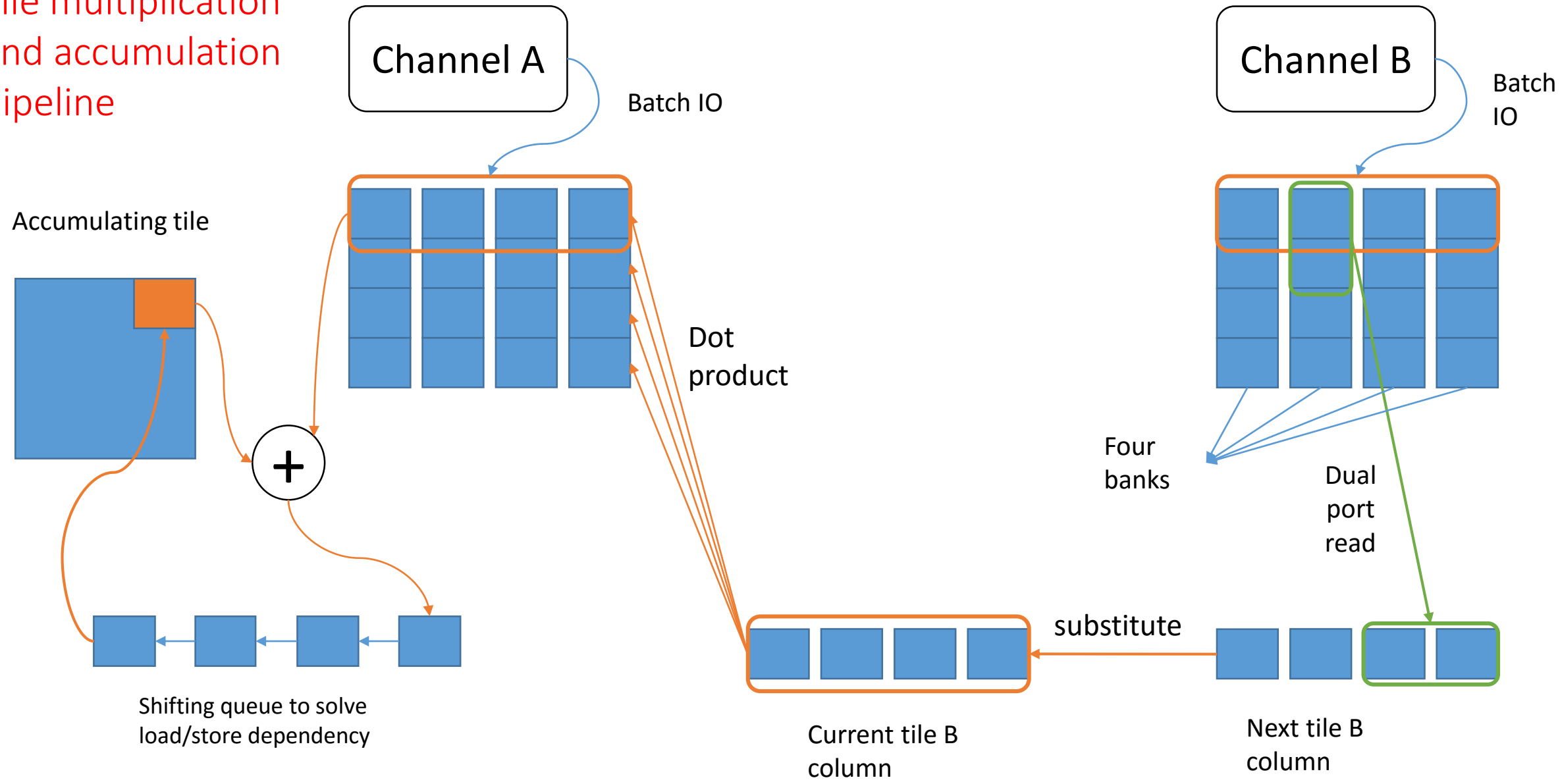
Optimization detail: Stall free pipeline at critical region

- Shift trick:

```
#pragma unroll
for (int i = 0; i < L - 1; ++i) {
    sh[i] = sh[i + 1];
}
accu[ndx - (N_SH - 1)] = sh[0];
sh[N_SH - 1] = accu[ndx] + sum;
```

- We explicitly tell the compiler to shift load and store operation on the same BRAM index by an amount of cycles.
- Making place for the compiler to generate the pipelined hardware, as while meeting our semantics.

Tile multiplication and accumulation pipeline



Altera OpenCL experience

- Altera's OpenCL compiler provides little primitives, design purpose is inferred by coding style. Subtle tricks are needed for generate the design we want.
- We summarized these experiences into a write paper.

Altera OpenCL experience: Example

Inferring a register array

```
reg[SIZE];
#pragma unroll
for (int i = 0; i < SIZE; ++i) {
    reg[i] = 0;
}
```

Inferring a bank interleaved block RAM array

```
banked_bram[SIZE][SIZE]
for(int i = 0; i < SIZE; ++i) {
    #pragma unroll
    for (int j = 0; j < SIZE; ++j) {
        banked_bram[i][j] = 0;
    }
}
```

Future work

- fBLAS
  - BLAS library written in OpenCL/A++
  - Usage
    - APIs to SW
    - Invoked by higher level compilers ()
- Evaluated on Dragontail Peak

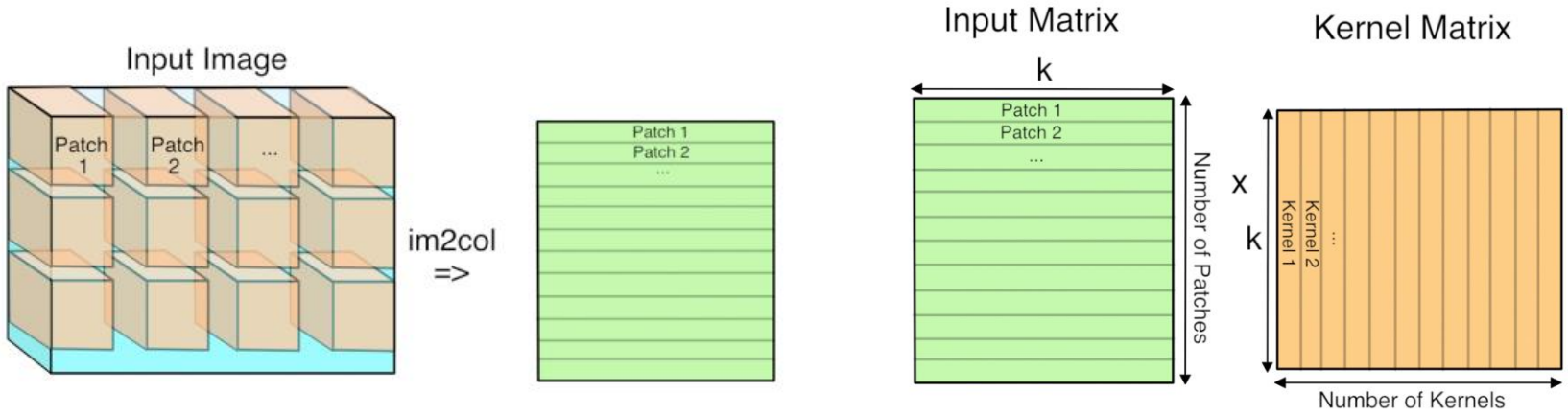
Thanks

3rd-level BLAS (Basic Linear Algebra Subprograms)

- 3rd–level BLAS stands for matrix multiplication:

$$C = \alpha \times A \times B + \beta \times C$$

- Useful in accelerating convolution operation in CNNs
  - Many deep learning library like cuDNN and Caffe have adopted this approach.



A FPGA MM library: fBLAS

- Achieves up to 90% performance advantage over Intel's Math Kernel Library.
- Better than Altera provided example on:
  - Saving resources: 30% performance promote with half DSP usage and about 20% logic resource usage.
  - Single workitem, easier to be integrated into other applications.
  - Performance predictable
  - Fully pipelined, maxing the hardware's computing capacity

Related work

- John David
- FPGA BLAS
- cuBLAS
- MKL