Efficient GEMM on FPGA with HLS

Sixiao Zhu, Jiali Jiang, Wenqiang Wang, Ningyi Xu, *Microsoft Research Asia*

1 Overview

- A introduce to GEMM and why it is important.
- A FPGA GEMM library that achieves up to 190% performance advantage over Intel's Math Kernel Library.
- A summary on our develop experience onf high level synthesis tool.

2 Introduce to GEMM

GEMM stands for General Matrix multiplication. It performs the following operation:

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

Where , B and C are dense float point matrices and α and β are float point scalar coefficients.

GEMM can be useful in accelerating convolution operation in convolutional neural network. The basic idea is to lower the fliter tensor into a matrix, and duplicating the original input data into another matrix, batch convolution operation could be done in a single multiplication this way. Many deep learning library like cuDNN[1] and Caffe[2] have adopted this approach.

3 Single Work Item GEMM Design

3.1 Environment specification

FPGA Accelerating Card	Microsoft Catapult FPGA accelerating card
Board DRAM	A single memory channel to the on-board 4GB of DDR3 memory (512M x 72b + ECC, 800MHz bus), exposing an Avalon bus interface operating at 200 MHz
ALMs	172600
On chip Memory Bits	41246720
DSP Blocks	1590

Table 1. Environment specification

3.2 Function

We provide a HLS based, signal work item, float point GEMM library on FPGA, it is compile time reconfigurable on tile size and compute unit usage, for users to trade off in terms of FPGA area consumption and performance. It can be runtime parameterized on Matrix size for general use.

3.3 Performance

Performance of our GEMM library is well predictable.

Suppose we have matrix A with size $R \times V$, matrix B with size $V \times C$, we use tile size of $TILESIZE$, we have N_{DSP} float point multiplication units (this parameter must be one or two times of $TILESIZE$), the execution time could be predicted by the following formula:

$$N_{tile} = \frac{R}{TILESIZE} \times \frac{C}{TILESIZE}$$

$$Cycles_{tile\_multiply} = \frac{TILESIZE \times TILESIZE \times TILESIZE}{N_{DSP}}$$

$$Cycles_{read\_tile} = TILESIZE$$

$$Cycles_{tile\_accumulation} = \frac{V}{TILESIZE} \times (Cycles_{tile\_multiply} + Cycles_{read\_tile} \times 2)$$

$$Cycles_{accu\_init} = \frac{TILESIZE \times TILESIZE}{2}$$

$$Cycles_{writeback\_tile} = TILESIZE \times TILESIZE$$

$$Cycles_{total} = N_{tile} \times (Cycles_{writeback\_tile} + Cycles_{accu\_init} + Cycles_{tile\_accumulation})$$

$$Execution\ time = Cycles_{total} \times \frac{1}{Frequency}$$

The bottleneck of our library lays on the computing capability. The DRAM reading overhead takes $4/TILESIZE$ proportion of execution time. 64 is a typical tile size, where the overhead is only 1/16.

With $TILESIZE$ set to 64, N_{DSP} set to 128 and $Frequency$ set to 200MHz (Both our library and the HLS Vendor Example), we get the following theoretical and experiment results, these values are nearly the same:

Matrix Size (size A = size B)	Theoretical (ms)	Experiment (ms)	HLS Vendor Example (ms)	MKL (ms)
128X128	0.2	0.5	0.6	0.4
256X256	1.2	1.4	1	2.3
512X512	7.5	8.3	8	15.0
1024X1024	52.4	56.6	63	81.0
2048X2048	388.0	419.7	522	760.0
4096X4096	2978.0	3208.4	4215	6097.0

Table 2 Performance

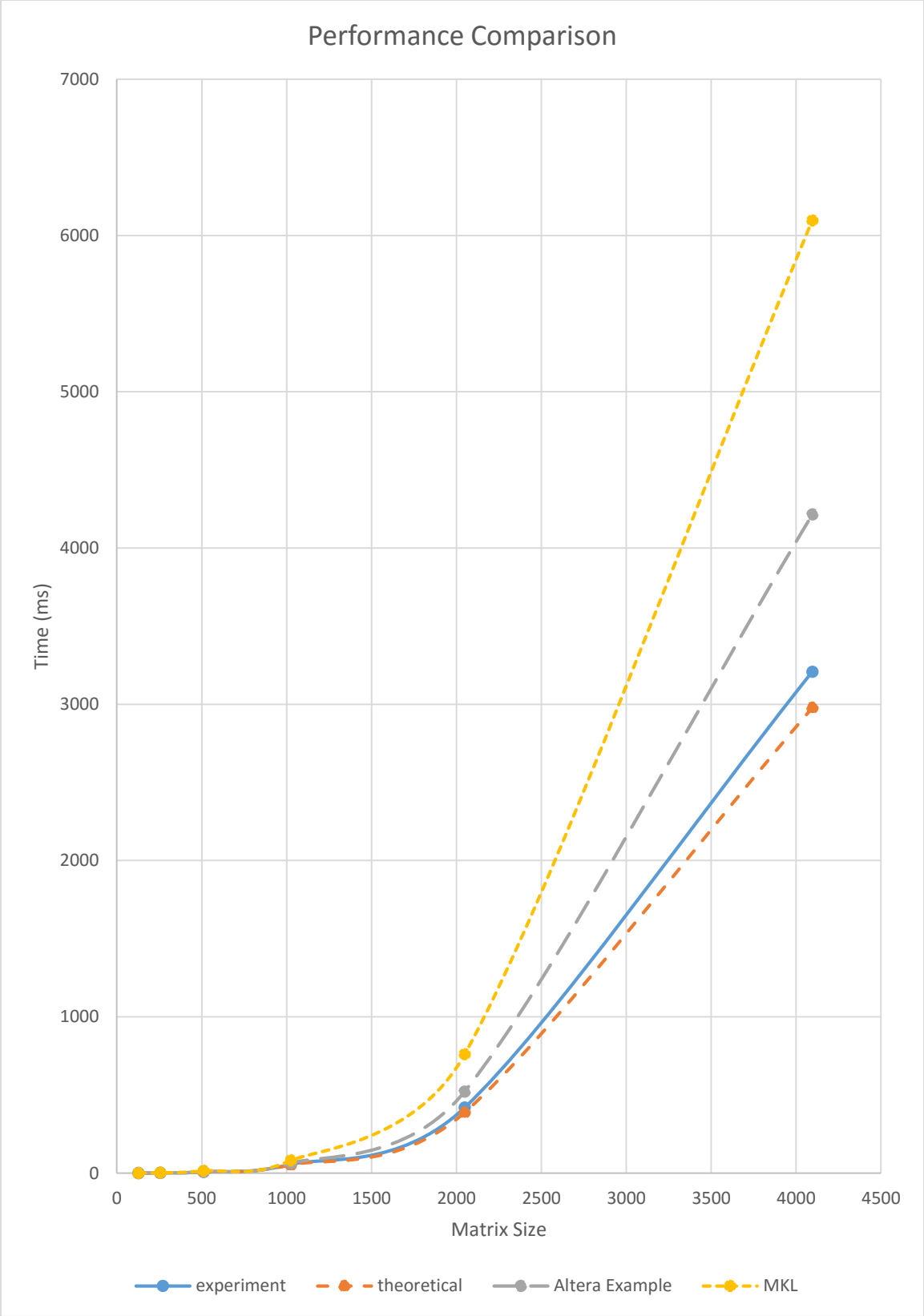

Figure 1 Performance curve

HLS Vendor gives matrix multiplication example on for its HLS compiler, it uses multiple work-item design and SIMD feature. Our library achieves 30% advance over HLS Vendor Example's performance with half DSP block usage, and saves about 20 % - 30 % logic resource.

3.4 Strategy

Intuitively, the time needed to finish a matrix multiplication can be estimated by: $number\ of\ total\ fp\ operations\ needed / number\ of\ fp\ operations\ done\ in\ unit\ time$, the $number\ of\ total\ fp\ operation$ is predefined by the matrix size, so we put our effort in increasing the number fp operation in per cycle. To achieve this, we need to use as much as possible computing unit (DSP) on the FPGA board, as well make the system to supply these computing units a steadily and high bandwidth input data flow.

There are big resource overhead for HLS Vendor HLS compiler to implement float point multiplication and addition, this overhead is beyond the programmer's control. On our board, 256 DSP multiplier is the limit that we could use in our application.

Due to the limitation of DSP usage, we focus more on elaborating our system to bring high bandwidth data supply to the computation units.

3.4.1 Tiling

A matrix element will be reused multiple times during the whole multiplication. To make sure the efficiency of re-fetching a particular data, we must adopt a data storage strategy that has good *locality*, that is, we want mostly fetch data from local block ram instead of DDR3 controller. A common manner to increasing locality in matrix multiplication is *tiling*.

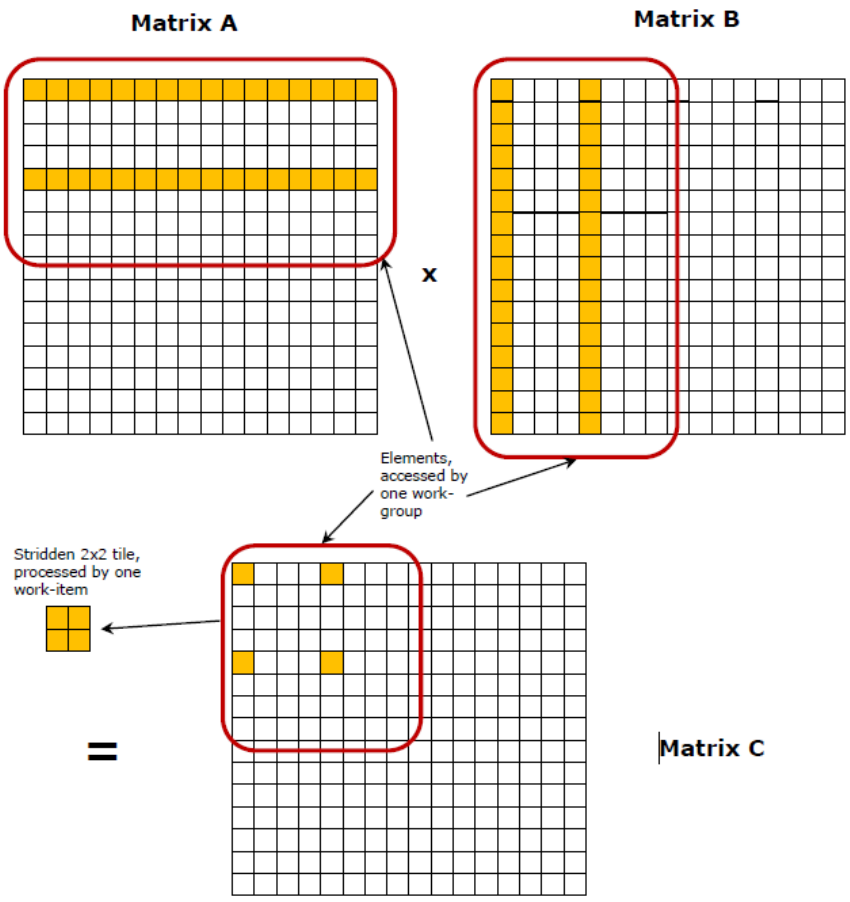

Figure 2 Tiling

Let R stands for matrix A's number of rows, C stands for matrix B's number of columns, V stands both for A's number of columns and B's number of rows, and L stands for tile size, we have:

$$number\ of\ tiles = 2 \times \left(\frac{R}{L}\right) \times \left(\frac{C}{L}\right) \times \left(\frac{V}{L}\right)$$

$$DDR\ time\ for\ single\ tile \propto (L \times L)$$

$$DDR\ time \propto (number\ of\ tiles \times DDR\ time\ for\ single\ tile) = \frac{R \times C \times V}{L}$$

As we can see from above, despite tilling does not increase computing capability, it reduces DDR IOs needed.

3.4.2 Buffering

We further reduce the DDR time by using an implicitly ping-pong operation.

We use a separate *ddr_in* kernel for DDR read, this kernel reads a tile from DDR controller and sends it to the calculation kernel through a deep fifo (*channel* in HLS Vendor HLS's term). This fifo's interface could be set much wider than the DDR bus, it enables the calculation kernel to pop data on a high bandwidth, and the *ddr_in* kernel to push data on a low bandwidth. This

strategy produces this scenario: While the calculation kernel is doing current tile's computing job, the ddr_in kernel is pushing next tile into the fifo.

This strategy saves data IO time by the proportion of the popping port width and the pushing port width in our case, DDRs have limited bandwidth, DDR burst read 512 bits per cycle, the calculation kernel pops 2048 bits per cycle, the accelerating rate is 400%.

3.4.3 High local memory bandwidth technics

After reading a tile to the local memory, we are faced with the problem how to feed simultaneously multiple float numbers to their corresponding multipliers. We achieve high local memory bandwidth with two tricks:

- Bank interleaving on one dimension. Suppose one physical block ram has 32 bit IO port (one float point), we could use multiple banks to store the whole tile, one block ram for one bank, and one bank for one column. We can batch read one row of the tile by this manner.
- Dual port reading on the other dimension. HLS Vendor HLS could configure its block ram as dual port ram, two reads or writes can be done in single cycle. Taking advantage of this feature, we could fetch two float point number in one cycle from a single bank, thus doubling the data fetching rate.

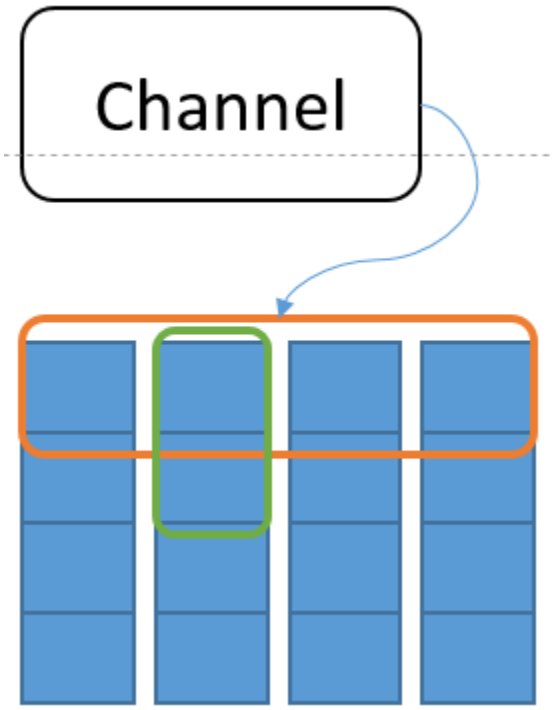

Figure 3 Bank interleaving and dual port reading, one vertical connected array stands for a bank, the horizontal circle stands for batch IO using bank interleaving, and the vertical circle for dual port IO.

The following code snippet shows how to generate the memory distribution above:

```
// banking
```

```
// unroll on the second dimension to force compiler to generate
banked ram
// distribution that allow batch IO in single cycle
#pragma unroll
for (int c = 0; c < TILE_SIZE; ++c) {
    tile_b[r][c] = tmp[c];
}
// Dual port IO
// two read in one cycle at a single bank, to instruct the
compiler to generate
// dual port ram for each bank.
float2 tmp  = (float2)(tile_b[r][c], tile_b[r + 1][c]);
```

Code Snippet 1

3.4.4 Efficient transposing

In our design, we assume matrix A and matrix B are both row major stored. This homogeneous storage persists to local block rams, in other words, tile for A and tile for B are bank interleaved on the same dimension. But matrix multiplication rule requires rows of A to multiply columns of B, we need to transpose tile B. All transpose method faces the same issue: The bank interleaving strategy could only ensure batch IO on one dimension of the tile. Using registers as tile storage meets the high bandwidth requirement on both dimensions, but the mass register interconnect produces unbearable FPGA resource consumption.

We propose "runtime transpose" to solve the above problem. Consider tile A multiplies tile B, in every step, we fetch out a single column of B from block rams, to multiply every row of A, so every column of B is reused *number of rows of A* cycles, during these cycles, we fetch the next column of B from its corresponding bank, with one (or two for dual port read) float point number a cycle. After the current column of tile B has done dot product with all rows of tile A, the next column of B is ready, then we do a substitution and keep the dot product going.

3.4.5 Fully pipelining in performance critical part

From Figure 2, we know that we need to accumulate multiple tiles' compute result to get a tile's final result and write back to matrix C. So after we have got a dot product result of one tile A row and one tile B column, we need to add this result to the corresponding position in the accumulating array. The pseudo-code below shows this process.

```
for (int i = 0; I < TILE_SIZE; ++i) {
    for (int j = 0; j < TILE_SIZE; ++j) {
        result_i_j = dot (tile_A_row_j, tile_B_column_i);
        accu[i][j] += result_i_j;
    }
}
```

Code Snippet 2

Here a technical problem arises: this code generates *data dependency* in HLS Vendor's HLS compiler. "+=" operator indicates the semantic to load value from a specific block ram named

accu, add to it a local result, and write back to the same location. The float point add take about 7 cycle to finish, so the inner loop must wait 7 cycles for each iteration to ensure the program's semantic, yet the we know the loop could launch one time per cycle, because there are no "read after write" hazard in this situation.

We used a "shift register" hack to solve this problem. We fetch value from accu block ram, do the addition and put the result to a shifting-forward queue's tail, after a constant amount of cycles (length of the shifting queue), the previously put value is shifted to the head of the queue, we write it back. This process is show bellow.

```
// shifting
#pragma unroll
for (int i = 0; i < L - 1; ++i) {
    sh[i] = sh[i + 1];
}
// put current adding result to the tail
sh[N_SH - 1] = accu[ndx] + sum;
// write back previously put value
accu[ndx - (N_SH - 1)] = sh[0];
```

Code Snippet 3

Ths hack works for the follow reason: From the compiler's perspective, there is no direct dependency between the load and store operation happens per cycle, as opposed to fp additon dependency in the naive version.

With this hack, we ensure the fully pipelined execution at the critical part, the DSP multipliers and tree adders' result flows out every cycle, all the computing capability are exhausted.

3.4.6 Put it all together

Following is a summary on the details mentioned above.

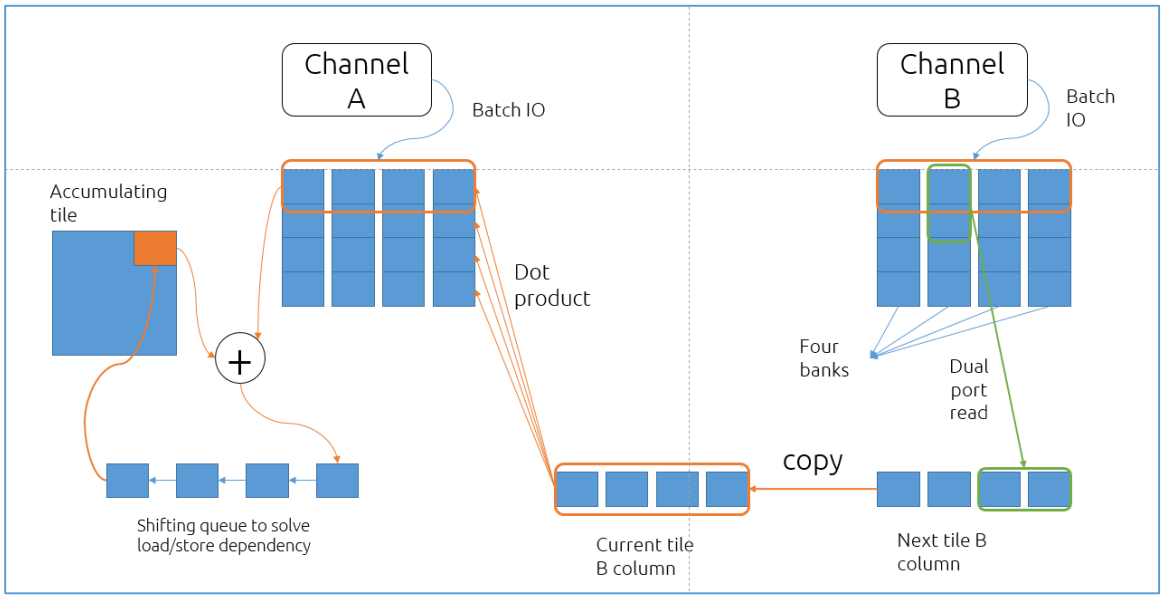

Figure 4 Whole flow

4 HLS High Level Synthesis coding styles

High level synthesis tool helps design FPGA applications without RTL level trivial implementation. But current HLS tools are far from mature, special tricks are needed to implement design intent, we would like to share our experience on using HLS Vendor's HLS tool here.

4.1 Control the inference of block RAM and register

Generally, dynamic indexing to an array makes it inferred as block ram, for the address could only be determined at run time and it is the ram's nature to random access. The following code shows how to initialize a block ram.

```
bram[SIZE];
for (int i = 0; i < SIZE; ++i) {
    bram[i] = 0;
}
```

Code Snippet 4 Inferring a bram

Unroll the above for loop causes a register inferring. Multiple hardware are generated, each writing a single register.

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

Code Snippet 5 Inferring a register array

If the unrolling happens on only one dimension of a two dimension array, the array will be inferred as multiple block ram bank to meet the timing requirement.

```
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;
    }
}
```

Code Snippet 6 Inferring a bank interleaving bram array

4.2 Adapting between different speed ports

Developing high performance applications, a common requirement is to adapt low bandwidth port and high bandwidth port. Consider this case: Transferring data from a 32bit-port ram to a

32bit-register array. Direct for loop style may cause reg array to be wrongly inferred as bram, as shown below.

```
float reg[SIZE];
float bram[SIZE];
for(int i = 0; i < SIZE; ++i) {
    reg[i] = bram[i];
}
// following batch read from reg array
```

Code Snippet 7 Naive copying

To make the reg array correctly inferred, use a shift style, or statically indexing style

```
float reg[SIZE];
float bram[SIZE];
for (int ndx = 0; ndx < SIZE; ++ndx) {
    #pragma unroll
    for (int j = 0; j < SIZE; ++j) {
        if (j == ndx) {
            reg[j] = bram[ndx];
        }
    }
}
// following batch read from reg array
```

Code Snippet 8 Statically indexing the reg array

In the above code snippet, #pragma unroll indicates multiple copying hardware generated, each having compile time fixed index j, so reg array is statically indexed. The ndx value is dynamical and the bram array is addressed at rumtime.

```
float reg[SIZE];
float bram[SIZE];
for(int ndx = 0; ndx < SIZE; ++ndx) {
    #pragma unroll
    for (int j = 0; j < SIZE - 1; ++j) {
        reg[j] = reg[j + 1];
    }
    reg[SIZE - 1] = bram[ndx];
}
// following batch read from reg array
```

Code Snippet 9 Shifting style

In the above code snippet, bram is dynamically addressed, reg is statically addressed and shifting forward. After SIZE cycles, the content of bram are all copied to reg array.

4.3 Inconsistence between emulation and hardware behavior

There are actually another way to solve load/store dependency mentioned in section 3.4.5. Since the load/store dependency on same array lays inside one tile's calculation, we could do a ping-pong operation when calculating successive tiles. When calculating even id tiles, we add dot product results and accumulating array 1 to accumulating array 2, vice versa.

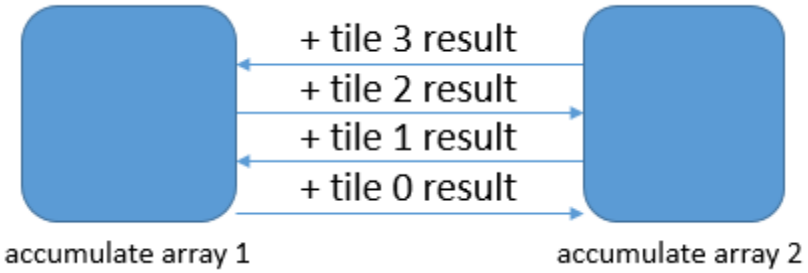

Figure 5 Ping-Pong accumulate

The above strategy sounds reasonable, the compilation report shows the dependency solved and the emulator behaves as expected, only that all our versions using this strategy receives hardware runtime error. It seems some tiles' calculation are skipped due to the wrong target code, we could not determine the exact cause of this inconsistence. The following code snap shows core operation of this ping-pong strategy.

```
if (ndx_tile & 1) {
    accu1[c - 1][r] = accu2[c - 1][r] + dot_result;
}
else {
    accu2[c - 1][r] = accu1[c - 1][r] + dot_result;
}
```

Code Snippet 10 Wrongly behaved code

4.4 State machine coding style

Some nested loop structure may cause slightly performance loss.

```
// loop 1
for (int i = 0; i < 16; ++i) {
    for (int j = 0; j < var; ++j) {
        // do something
    }
}
// loop 2
for (int i = 0; i < var; ++i) {
    for (int j = 0; j < 16; ++j) {
        // do something
    }
}
// loop 3
```

```
for (int i = 0; i < 16; ++i) {
    for (int j = 0; j < 16; ++j) {
        // do something
    }
}
// loop 4
for (int i = 0; i < var; ++i) {
    for (int j = 0; j < var; ++j) {
        // do something
    }
}
```

Code Snippet 11 Different loop styles

Compilation report shows loop 1 and loop 2 fully pipelined, loop 3 and loop 4's inner loop fully pipelined, but loop 3 and loop 4's out loop "Successive iterations launched every 2 cycles due to: Pipeline structure".

This is because the iteration time of loop 3 and loop 4's inner loop are variable, the inner loop exit information needs an additional cycle to be generated (j comparing to var), and the outer loop must wait at this additional cycle. Through this will not hurt inner loop's pipeline structure, it will slow down the total performance by $1/_{out\ loop\ round}$. The less the inner loop iterates, the bigger this overhead will be.

We recommend to organize the code structure as a big while loop, using state machine to implement logic. No inner loop inside the while loop unless it is unrolled. This coding style brings two benefits:

- Timing to be expectable. Execution time is only determined by the while loop's reported dependency and total loop rounds.
- Dependencies between different levels of loops avoided. These kinds of dependencies are hard to locate and solve, as shown below.

```
for (...) {
    for (...) {
        // store operation
        for (...) {
            // load operation
        }
    }
}
```

Code Snippet 12 Nested loop dependency

If we construct the code as a state machine as below, compiler generates a pipeline for each condition case, and automatically padding shift registers to align the conflicting load/store operation in difference case, thus resolves many potential dependencies.

```
int bram[SIZE];
int state = 0;
while (1) {
  switch (state) {
    case 0:{
              int a = 0;
              a += 1;                 // @ cycle 1
              a += 1;                 // @ cycle 2
              a += 1;                 // @ cycle 3
              // store operation
              bram[runtime_ndx] = a; // @ cycle 4

              // ...

              // runtime determined state transfer
              // ...
              break;
            }
    case 1:{
              // load operation
              int a = bram[runtime_ndx]; // @ cycle 1
              a += 1;                    // @ cycle 2
              a += 1;                    // @ cycle 3
              a += 1;                    // @ cycle 4

              // ...

              // runtime determined state transfer
              // ...
              break;
            }
  }
  if (state == EXIT_STATE) break;
}
```

Code Snippet 13 State machine coding style

Following is naive pipeline structure of code above, new data flows in every three cycles to meet the code semantic and to avoid rw or wr data hazards.

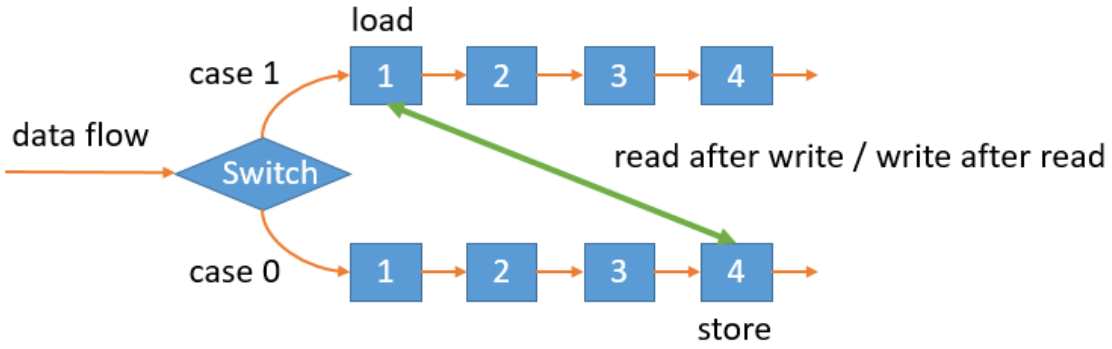

Figure 6 Naive pipeline structure

High level synthesis tool optimized pipeline structure, increase pipeline depth to resolver data dependency, delivering one data per cycle throughput.

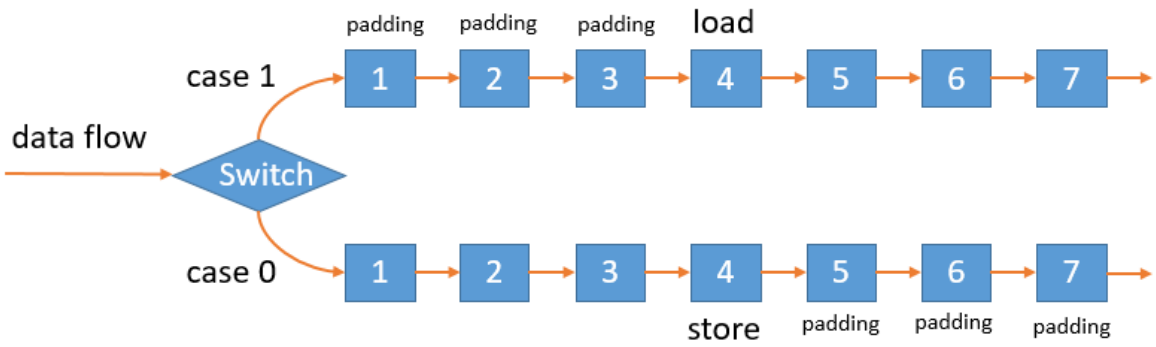

Figure 7 Optimized state machine pipeline structure

5 Summary

We developed a reconfigurable efficient GEMM library for accelerating research and engineering applications, we also shared our experience on using the emerging high level synthesis tools, and proved the feasibility of using high level synthesis tools to develop on heterogeneous systems.

6 References

[1] Chetlur, Sharan, et al. "cudnn: Efficient primitives for deep learning." arXiv preprint arXiv:1410.0759 (2014).

[2] Abuzaid, Firas, et al. "Caffe con Troll: Shallow Ideas to Speed Up Deep Learning." arXiv preprint arXiv:1504.04343 (2015).