Hardware Acceleration of Matrix Multiplication on a Xilinx FPGA

Nirav Dave, Kermin Fleming, Myron King, Michael Pellauer, Muralidaran Vijayaraghavan
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
Email: {ndave, kfleming, mdk, pellauer, vmurali}@csail.mit.edu

1 Introduction

This year, the first MEMOCODE hardware/software co-design contest [2] posed the following problem: optimize matrix-matrix multiplication in such a way that it is split between the FPGA and PowerPC on a Xilinx Virtex IIPro 30. In this paper we discuss our solution, which we implemented on a Xilinx XUP development board with 256 MB of DRAM. The design was done by the five authors over a span of approximately 3 weeks, though of the 15 possible man-weeks, about 9 were actually spent working on this problem. All hardware design was done using Bluespec SystemVerilog (BSV) [1], with the exception of an imported Verilog multiplication unit, necessary only due to the limitations of the Xilinx FPGA toolflow optimizations.

For this exercise, all data was stored as complex numbers with 2.14 signed fixed-point representations of both the real and complex components. The three matrices (two multiplicands and the product) were specified by their size and starting DRAM address in code running on the PowerPC. Matrices were assumed to be square with sides of length ranging from 64 to 1024 in powers of two. The given test framework featured a software-only implementation and our task was to use the FPGA as an accelerator to decrease execution time.

1.1 Design Principles

Early on, we realized the strict contest deadline meant that the main resource to conserve was designer time. With such a limited design timeframe, any small delay could mean failure, so as a result, we decided to decouple the various aspects of our design so that work on different modules could progress independently. This decision led us to a latency-insensitive modular design with FIFO channels as the only form of communication.

Keeping all datatypes in sync between different modules was another issue of concern. In the hardware specification, it was relatively straightforward to use Bluespec's algebraic types to avoid dealing directly with bits. Maintaining consistency with the software, however, was a more complicated task. To solve this, we wrote parameterized functions to keep track of the exact bit packing, taking care that the implementations matched the default bit packing of the Bluespec compiler.

The last major issue was testing. With so many modules, so many developers, and so much fluctuation in the system, it was important that we be able to quickly isolate and fix bugs. The approach we used was to maintain a verified reference implementation for each module which we could either plug into the larger system simulator or run in a simple testbench. These golden models would then be used to verify each change in our hardware implementation. The ease with which BSV modules interface made this process completely natural, allowing us to swap out reference and experimental module implementations with ease.

2 Design Infrastructure and Architecture

A major design decision was whether to choose matrices or vectors as the primitive datatype for the Functional Unit. Since our algorithms make use of sub-matrix blocking, we decided that choosing $n \times n$ sub-matrices as our primitive would give the hardware more flexibility in its operation, possibly resulting in better circuit-level design.

We partitioned our architecture into several blocks as shown in Figure 1, each with a specific task listed below:

PowerPC: Code running on the PowerPC orchestrates the computation. Communication with the Feeder takes place through two memory-mapped FIFOs.

Feeder: The Feeder is responsible for passing instructions from the PowerPC to the hardware controller as well as signaling to the PowerPC when an instruction is complete. For debugging purposes, it also sends periodic heartbeats to the PowerPC.

Controller: The controller decodes the ISA instructions into various control commands which are then forwarded in an asynchronous manner to the appropriate system modules.

PLB Master: The PLB Master communicates directly with the Processor-Local Bus (PLB), handling all memory manipulation directly.

Memory Switch: The memory switch routes memory traf-

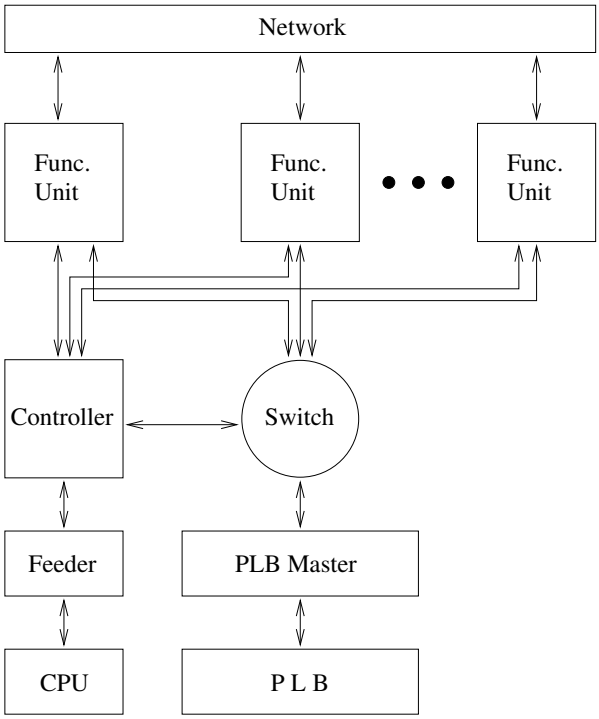

Figure 1. Block Diagram of Matrix Design

fic between the functional units and the PLB Master.

Functional Unit: The functional unit handles all matrix computation, containing three logical matrix registers (A, B, and C).

Functional Unit Network: The functional unit network allows data to be passed from functional unit to functional directly without going through memory.

2.1 ISA

The hardware accelerator implements a simple ISA consisting of 3 types of Operations: Memory, Compute, and Forward. Memory ops move data between main memory and the logical registers of Functional Units. Forwarding ops move data between functional units, while Compute ops move data between registers within a functional unit. The compute opcodes are accumulate ($C = C \pm A$), multiply ($C = A \times B$), multiply-accumulate ($C = C \pm A \times B$), and zero ($C = 0$).

3 PLB Master

Given the size of the matrices, efficient communication with the system memory on the XUP board was of utmost importance. We considered three possible solutions: the PowerPC could be used to orchestrate the communication, our hardware could connect to the system memory bus and communicate with the memory controller directly, or we could implement our own DRAM controller, and communicate directly with the DRAM. While the first option is attractive since it is easy to implement and the majority of the design effort is in software, it is also the slowest, with a maximum bandwidth of 150 MB/s. The second option adds some hardware complexity, as a bus protocol must be implemented in hardware, but offers a significant bandwidth improvement of up to 800 MB/s. Though the final options provides the most memory bandwidth, the it requires a complex control state machine — a significant increase in complexity over the other two approaches. We were unsure whether we could complete such an ambitious design within the deadline so we chose the second option, since it offered a good combination of bandwidth potential and design complexity.

The Processor Local Bus (PLB) [3] is an IBM-specified bus used on the Xilinx XUP board to connect the DRAM memory controller to the processor. We implemented a bus master interface capable of loading and storing matrices from memory, parameterized to support different matrix sizes. Our PLB master transfers data in bursts of sixteen 32-bit words, thereby amortizing the cost of the bus protocol and resulting in a typical transfer rate of just over one word every two cycles. Extending the controller to support 64-bit transfers would increase the memory bandwidth, and though this is fully supported by the PLB Master, no other part of the system was designed to use this feature.

3.1 PLB Master Timing Issues

When synthesized independently, our PLB master implementation had a 7 ns critical path, but when connected to the PLB, the synthesis tool reported a critical path of nearly 12 ns. This large change occurred because the PLB Master had a combinational path that ran through the PLB arbitration logic. Because the arbitration logic is a black-box, we had to heavily modify our PLB Master to ensure its critical path was short enough to make timing.

Fortunately, most of the offending logic could be retimed, a procedure which pushes logic across state elements. This transformation preserves the observable behavior of the module, but can reduce the critical path of the system. By retiming the logic that touched PLB control wires, we were able to remove almost 3ns from our critical paths, and achieve our timing goal of 10ns.

4 Functional Units

The functional unit is responsible for doing the block-level computations and consists of a command FIFO holding operations to be executed, a pair of FIFOs interfacing to the memory switch, and another pair for the functional unit network. In addition it contains three local "memories" corresponding to three logical registers A, B and C used as operands in its instruction set, supporting the commands zero, load, store, multiply, multiply-accumulate, and forward (from one FU to another through the network).

Our original design consisted of a FIFO of outstanding commands, and a series of FSMs, generated via Bluespec's StmtFSM embedded language, each responsible for executing one of the opcodes (i.e. load, store, mul, etc.). These

FSMs contained most of the address generation logic and were required to run mutually exclusively, thereby exposing all memory transfer latency.

We refined the design by first extracting out the memory address lookup logic for each operation and merging them into a single FSM module handling various read and write orders for each of the three registers. This caused three important benefits. First, this simplified all of the operation FSMs to the point that they need only call the appropriate "startRead" and "startWrite" methods on the memory modules with the appropriate patterns and do their simple per-step operation until the "doneRead" and "doneWrite" signals are set, making the FSMs simple rules. Second, since all the accesses were all localized into these new register modules, it was trivial to replace the combinational memories with more efficient single-cycle BRAM models (important for efficient FPGA synthesis). Last, with all the information of where reads and writes are localized, allowing operations to run two operations concurrently in a streaming fashion (e.g. load A happening in tandem with multiply) needed only a little additional logic to prevent reads and writes to the same register from getting out of order. This allows the functional units to almost completely hide memory latency.

As a final optimization we extended the blocking from one complex multiply per cycle to n per cycle. This required changing the memories to have n-word blocks and slightly complicated the logic designed to keep things in order. By parameterizing our functional units by the number of parallel multiplies, we can easily change the profile of the design until we achieve the most effect use of implementation resources.

4.1 Sub-block Multiplication Algorithms

In our original design we used naive N^3 multiplication, where each value of C is computed one element at a time. Assuming n complex multiplications per cycle, the computation looked like:

```
for(i=0; i < N; i++)
  for(j=0; j < N; j++)
    for(k = 0; k = N; k+=n)//cycle
      for(z = 0; z < n; z++)
        c[i][j]+=a[i][k+z]*b[k+z][j];
```

To improve memory access we needed to store the B matrix in column major order, a somewhat awkward task. Additionally, each step of computation requires us to sum the n products from each round. This means we have a $log_2(n)$ depth adder-tree, which greatly limits the size of n that we could generate. Instead the following algorithm was proposed:

```
for(i=0; i < N; i++)
  for(j=0; j < N; j++)
    for(k = 0; k = N; k+=n) // cycle
      for(z = 0; z < n; z++)
         c[k+z][j]+=a[i][j]*b[j][k+z];
```

While we have not fundamentally changed the cycle-level performance of the design the logic required to implement this algorithm is much faster. Now instead of a single adder-tree, we have n parallel additions, resulting in a shorter critical path. It also allows us to keep the B matrix in row-major simplifying the loading and storing logic. Of course now we must write to the C matrix many times, but since the logical registers in the Functional Units synthesize to fast BRAMs this is a negligible penalty.

4.2 Design Variation

We generated two designs in parallel, both of which did essentially the same work. One used hardware to calculate when two operations could be executed simultaneously, while the other used a slight variant of the ISA to explicitly merge the high-level instructions. While the hardware-pipelined design implemented generalized instruction chaining, the software-pipelined design was far less general purpose and subsumed the memory switch. In addition, the design which implemented hardware instruction chaining also included a higher degree of parameterization and could (for example) be parameterized by the degree of add parallelization. The fully parameterized design was faster, since we found 64×64 blocks hid more of the memory latency.

5 Algorithm

At first we thought that the various clever algorithms, such as Strassen's algorithm [4] would yield better performance by replacing expensive multiplies with less expensive additions. While relatively limited on an FPGA, multipliers are quite cheap when compared to the routing costs of complicated muxing logic needed to implement these algorithms. We therefore focused exclusively on simple n^3 algorithms. Our initial algorithm, made use of only one functional unit:

```
for(i = 0; i < NumBlocks; i++){
  for(j = 0; j < NumBlocks; j++) {
    zero();
    for(k = 0; k < NumBlocks; k++){
      loadIntoFU(B, k, j);
      loadIntoFU(A, i, k);
      mulAccumulate();
    }
    store(C, i, j);
}}
```

It loads the in the data to calculate sub-matrix $C_{(i,j)}$ and then stores the result in the appropriate memory location. One slight optimization we made was to load the B matrix before the corresponding A matrix. This completely hides half the total load latency since the functional unit's access of A naturally matches the order in which it is loaded, allowing the computation to begin as soon as the first element of A returns from memory.

Our second algorithm was made use of multiple functional units, though the per-block computation remained identical. Data loads were spread across the n functional blocks to allow n multiplies to happen concurrently.

Our last implementation attempted to reduce the number of sub-matrix loads. We observed that the last sub-matrix in

SW Pipelined System - 16 muls, 16×16 block	
size	time (μs)
64^2	578
128^2	6,274
256^2	48,374
512^2	383,598
1024^2	3,043,532
HW Pipelined System - 8 muls, 64×64 block	
64^2	799
128^2	5,122
256^2	45,318
512^2	332,011
1024^2	2,711,073
Pentium 4, Reference Algorithm	
64^2	11,000
128^2	75,000
256^2	608,000
512^2	12,805,000
1024^2	139,000,000

Figure 2. Performance Results

a column of B could be reused instead of reloading it from memory. To accomplish this reuse, we changed the write pattern of sub-blocks of C to column major. The input matrices are accessed in a snaking fashion, first accessing a row (or column) in one direction and then the other for the next row (or column). This algorithm gives an approximately 1% improvement in performance:

```
forward = True;
loadIntoFU(B, k_index, i)
for(i = 0; i < NumBlocks; i++){
  for(j = 0; j < NumBlocks; j++){
    zero();
    for(k = 0; k < NumBlocks; k++){
      k_idx = (forward)? k : MaxBlocks-k;
      if(k > 0)
        loadIntoFU(B, k_index, i);
        loadIntoFU(A, j, k_index);
        mulAccumulate();
    }
    storefromFU(C, j, i);
    forward = !forward;
}}
```

6 Results

Figure 2 shows the time each design took to multiply matrices of various sizes. While the hardware-pipelined design is generally faster, the fact that the software-pipelined design beats it for the smallest size indicates there could be room for further optimization.

7 Further Work

Due to the time constraints imposed by the contest, many shortcuts were taken. There are many aspects of this design left to be explored and a number of obvious points still left to be optimized. While we successfully parameterized the both the number of functional units, as well the number of multipliers in each functional unit, not enough exploration was done to find the optimal points for these parameters.

Our final design used a single clock, but due to our communication format, partitioning the design into multiple clock domains would have been a natural optimization to try.

To do a single complex multiply we used the simple algorithm requiring four fixed-point multipliers:

```
function complex_mult(a,b);
  return CMult{
     i: a.i*b.i - a.q*b.q,
     q: a.i*b.q + b.i*a.q
   };
endfunction
```

However, we can save one of the multipliers (a limited resource on the FPGA) with the following technique:

```
function complex_mult(a,b);
  let ii   = a.i*b.i
  let qq   = a.q*b.q
  let iqiq = (a.i+a.q)*(b.i*b.q);
  return CMult{
     i: ii-qq,
     q: iqiq - (ii+qq)
   };
endfunction
```

It is possible that this saving of multipliers would be worth the additional logic coupling.

Finally, not enough work was done to optimize the software and block-level algorithmic aspects. Simple optimizations could easily improve performance, the most notable of which would be to alter the block data layout. In the original scheme, the matrix is stored in row-major order, causing each read and write to the memory to "jump" addresses on each line, which in turn leads to an increased number of RAS misses per block load. Taking the n^2 cost to reorder the matrix that blocks are stored in a contiguous area of memory would greatly improve runtime costs. This, along with modifications to the PLB master could result in full memory bandwidth saturation.

Acknowledgments

The authors would like to thank Chris Batten, Joel Emer, and Mieszko Lis for their advice and insight in algorithmic organization.

References

[1] Bluespec Inc. `http://www.bluespec.com`.

[2] Forrest Brewer and James C. Hoe. The First MEMOCODE HW/SW Co-design Contest. `https://memocode07.ece.cmu.edu/contest.html`, March 2007.

[3] IBM, Inc. *The CoreConnect (TM) Bus Architecture*, 1999.

[4] V. Strassen. Gaussian elimination is not optimal. In *Numerische Mathematik 13*, pages 354–356, 1969.