BUILDING A BETTER MACHINE LEARNING HARDWARE ACCELERATOR WITH HARP

An Undergraduate Research Scholars Thesis

by

JACOB SACCO

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Sunil Khatri

May 2019

Major: Computer Engineering, Electrical Engineering Track

TABLE OF CONTENTS

ABSTRACT

Building a Better Machine Learning Hardware Accelerator with HARP

Jacob Sacco
Department of Electrical and Computer Engineering
Texas A&M University

Research Advisor: Dr. Sunil Khatri
Department of Electrical and Computer Engineering
Texas A&M University

As machine learning is applied to ever more ambitious tasks, higher performance is required to be able to train and evaluate neural nets in reasonable amounts of time. To this end, many hardware accelerators for machine learning have been made, ranging from ASICs to CUDA code that runs on a conventional GPU. GPU and FPGA based accelerators have seen more success than ASICS due to the ease with which the design can be tweaked or revised, but still suffer from latency resulting from the interface between the processor and the accelerator (generally PCIe). The purpose of this paper is to build a hardware accelerator on Intel's Heterogeneous Architecture Research Platform, which includes a Xeon processor and Arria 10 FPGA on the same mainboard, which share access to common memory. This should significantly reduce latency and increase throughput. This accelerator is expected to at least match the performance of a typical machine learning library implementation on a GPU, and will hopefully significantly exceed it.

NOMENCLATURE

CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HARP	Heterogeneous Architecture Research Platform
ML	Machine Learning
MAC	Multiply-Accumulate
PCIe	Peripheral Component Interconnect express
CUDA	Compute Unified Device Architecture
ASIC	Application Specific Integrated Circuit
CCI	Core-Cache Interface
QPI	Intel QuickPath Interconnect
PE	Processing Element
PB	Processing Block
API	Application Programming Interface
CSR	Control-Status Register
IP	Intellectual Property
FIFO	First In, First Out

LIST OF FIGURES

Page

CHAPTER I

INTRODUCTION

In the modern world, machine learning is becoming a more and more important part of daily life. It is showing up in everything from self-driving cars recognizing objects around them, to companies trying to determine what to recommend to a given consumer to get the highest sales. As the size and complexity of machine learning applications and frameworks increases, so too does the amount of resources required to train and utilize these neural nets. As a result, there is wide interest in hardware acceleration for machine learning. Borne of this interest are a wide variety of products and frameworks for hardware acceleration, ranging from Google's Tensor processing unit, which is a custom integrated circuit for machine learning, to PyTorch, which is a Python machine learning library that is capable of using the user's GPU (Graphics Processing Unit) for acceleration. While the potential speedup resulting from creating an ASIC (Application Specific Integrated Circuit) -based accelerator is greater, the ease with which the design can be tweaked or restructured has caused GPU-based (and to a lesser extent FPGA-based) accelerators to be more popular than ASIC-based solutions [1]. GPUs also suffer from high latency from the interface method, and have reduced peak performance and power efficiency due to the fact that GPUs are designed for graphics applications and not machine learning. FPGAs (Field Programmable Gate Arrays), on the other hand, can be tailored to their application to a much greater degree, as they provide a method to place digital logic directly onto the hardware in a reconfigurable manner, and much work has been done in optimizing matrix multiplication on FPGAs [2], as well as in things like extremely fast approximate multipliers such as SiMul [3]. Finally, Intel has created the Heterogeneous Architecture Research Program, or HARP, which

combines a Xeon CPU and Altera Stratix V FPGA on the same board, which share access to memory resources. The purpose of this proposal is to leverage past work into matrix multiplication and FPGA machine learning accelerators to implement a hardware accelerator for the HARP platform, with the aim of combining the speed of an FPGA based solution with the low latency and high throughput of directly accessible shared memory. This will allow for neural nets to be trained and used faster than was previously possible with a single machine.

CHAPTER II

METHODS

The first step was to gain access to and learn to use one of the HARP prototypes. Now that that has been accomplished, the core of the system, a matrix multiplication engine is designed and implemented. Different methods of structuring the engine based on past research [2, 3] will be trialed in order to determine which structure works best. Accompanying this will be a series of control modules and a driver for making calls to the engine. The performance of the design will be compared against what the Xeon processor that the HARP system contains can do without the assistance of a hardware accelerator. The pipeline will then be optimized as much as possible, leveraging past research in this as well [4], in order to maximize the speed that the accelerator can reach. Once this has been accomplished, final data will be collected on the system to determine its overall performance. This will be analyzed to determine the suitability of heterogeneous architecture systems for machine learning applications.

The hardware accelerator is accessed by user level software on the system CPU, written in either C or C++. The matrices are stored in column-major order, and their cacheline address is sent to the hardware accelerator via its CSRs (Control Status Registers). The accelerator is then given the signal to start, and the result is awaited in a pre-allocated shared memory buffer.

CHAPTER III

STRUCTURE AND OPERATING PRINCIPLE

The design of this hardware accelerator relies on the heavy parallelization of the individual multiplication operations involved in a matrix multiplication to achieve a speedup over a processor-based implementation. To accomplish this parallelization, the accelerator includes many small processing elements that are designed to be tiled, which are at the bottom of a hierarchy of progressively larger and coarser-grained levels of control. This hierarchy starts with a high-level controller at the top level. The controller interfaces with user programs to receive requests, report status information, and return results, and also is responsible for dispatching chunks of matrices to a processing block. A processing block is a coarse structure that is dispatched a chunk of the input matrices, and computes a partial output matrix. Each processing block contains sixteen processing elements, each of which can compute the dot-product of a single row/column pair. During the multiplication of a matrix, the processing block is directed to multiply sixteen rows of the first operand with the entirety of the second operand. The processing block assigns row/column pairs to its processing elements, progressively working down the columns of the second operand until it has completed the partial multiplication. The high-level controller will continue to dispatch sixteen-row matrix chunks to processing blocks as they become available until all the rows of the first operand have been multiplied by all the columns of the second operand. Each bunch of rows produces a chunk of the result matrix which is written out to memory wherever the final result needs to be placed. A coarse block diagram of the overall system is shown in Figure 1.

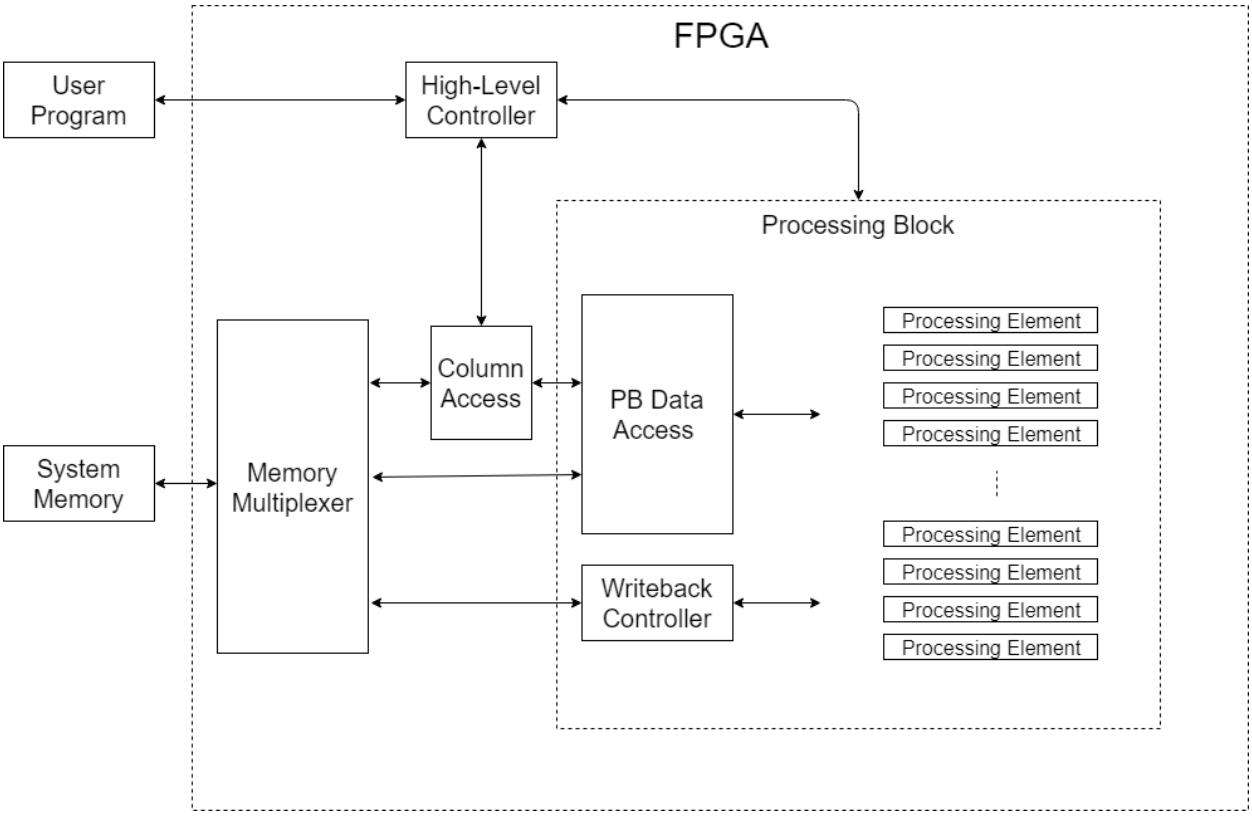

Figure 1. The overall system block diagram

The bottleneck for this scheme is the speed with which it can read matrix data out of memory to be processed, so having a highly efficient memory access model is paramount. The HARP system provides a bus for interaction with the system memory called the Core-Cache Interface (CCI). This interface is an abstraction over the actual connections between the FPGA and processor, which take up two PCIe (Peripheral Component Interconnect) x8 lanes and a QPI (QuickPath Interconnect) bus. This interface allows for up to approximately 100 memory transactions to be in flight at once, and bandwidth of up to approximately 25 GB/s, when the CCI bus is clocked at its full speed of 400 MHz. This interface is fairly sophisticated and only presents one unified bus, so there is a need to abstract its interactions away from individual processing elements and also multiplex access to it.

The multiplexing is accomplished with a very basic memory multiplexer module which has a simple interface for the owner of a port to request access to the CCI bus for some number of cycles. The multiplexer services these requests in a round-robin fashion, and has a reconfigurable number of ports so that an arbitrary number of processing blocks can be connected to it. Each processing block contains three memory access modules with different behaviors: the column access module, the row access module, and the writeback module. The column access module reads down the columns of the second operand one cache line (64 bytes) at a time. The row access module has to read data against the 'grain' of the first operand as the inputs are required to be in column-major order, so its operation is somewhat more complicated. It fires off large bursts of cacheline requests and fills in data as it comes in, with each cacheline of data containing one index for each of the sixteen processing elements. Once sixteen contiguous read requests have come back, the module signals the processing elements that their data is ready. Upon this signal, each processing element ingests one of the sixteen row chunks and the cacheline from the current column, and performs a multiply-accumulate on it. While this is occurring, the row access and column access modules are sending any necessary requests and waiting for the next batch of data to come in. Each PE sends the results of its multiply-accumulate operation along with metadata to the writeback module, which accumulates partial results from each input cacheline until it has a full cacheline's worth of result indices. The writeback module then requests access to the CCI bus and writes that result out.

CHAPTER IV

IMPLEMENTATION

The various modules of this project were built one by one, and then tested against the others. As functionality was fleshed out and bugs were removed, each module was iterated on until it reached its final form. Every module has a well-defined interface with its neighbors, and a scheme for ensuring that data gets across the boundaries safely and as rapidly as possible. Each module also includes one or more state machines responsible for maintaining those interfaces and handling internal operations, and some include large memory caches. A detailed description and discussion of each module follows.

The Memory Multiplexer

The purpose of the memory multiplexer, or memory muxer, is to allow any module in the design that requires access to system memory to transmit requests and receive results. To this end it provides a standard interface that modules may use to request access to the CCI-P bus, and to send memory requests and get responses. The interface consists of an eight-bit bus where the module places how many cycles it would like to access the CCI-P bus for, and a want-tx signal. These come alongside copies of the standard CCI-P bus, and a tx-ready line that runs from the muxer to the module. The muxer maintains a state machine that services requests for access in a round robin fashion. The muxer's high level interface is shown in Figure 2, and the muxer port interface is shown in Figure 3.

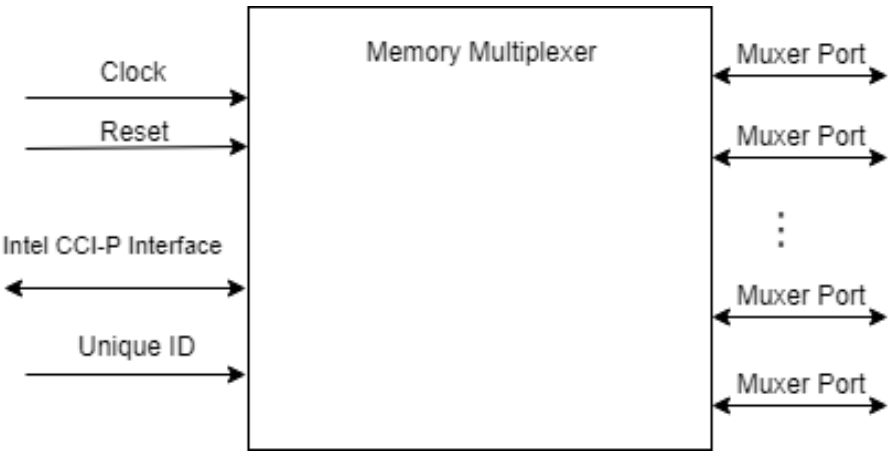

Figure 2: The high-level interface for the memory multiplexer

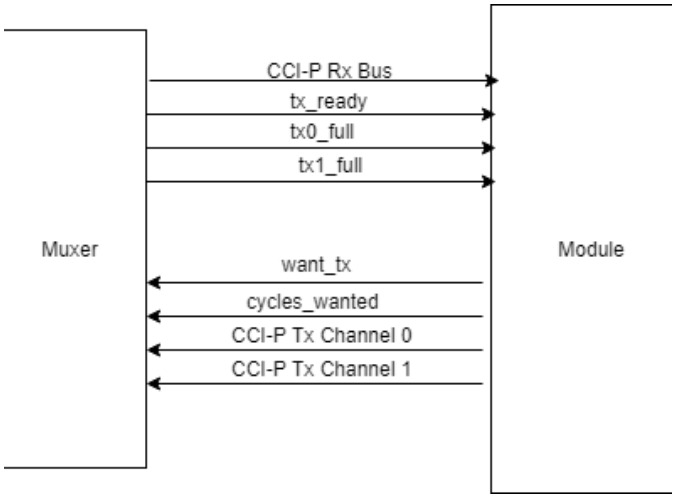

Figure 3: A close-up of the muxer port interface

When a module needs to transmit a memory request, it asserts want_tx, and places how many cycles for which it needs the interface on cycles_wanted. When the muxer reaches that module, it asserts tx_ready and relays the module-side copies of both of the TX channels to the external CCI-P interface for as many cycles the module requested before de-asserting tx_ready, and moving to the next module. The CCI-P bus provides the tx0_full and tx1_full signals, which indicates that the respective channel is nearly at capacity for the number of requests that can be in flight at once. These are fed straight through to the modules, and it is expected that modules will

heed these signals and not attempt to send requests when the bus is full. Finally, the CCI-P Rx bus, which contains all read responses, is relayed to all connected modules at all times without multiplexing. This allows for a module to only hold onto the interface for as long as it takes to send its requests, and then can release it while the module waits for responses. It is expected that modules will be able to correctly detect whether or not a response belongs to that module based on the attached metadata.

The Memory Multiplexer State Machine

The operations of the memory multiplexer are controlled by a single state machine, shown in Figure 4. The state machine consists of four states: Relaying, Stopping, Stopping 2, and Changing. The state machine starts out in the Relaying state, where it latches both TX channels from the zeroth module onto the external CCI-P bus for as long as that module is requesting it. Every cycle spent in relaying, the muxer decrements a counter. When the counter hits zero, the muxer transitions to the Stopping state. In this state, the muxer de-asserts the current module's tx_ready signal, but still relays any requests on that cycle. The muxer transitions directly to Stopping 2 on the next cycle. In Stopping 2, tx_ready remains de-asserted, but the muxer still relays any requests that come through. This is so that heavily pipelined modules will not have their requests thrown away by the muxer because tx_ready did not propagate in time, or because the requests themselves were pipelined. From Stopping 2, the muxer transitions directly to Changing. In this state, the muxer changes which module it is servicing. It cycles through one module every cycle until it finds one that has asserted its want_tx line. When one is found, the muxer then asserts that module's tx_ready line, sets its timer to the module's cycles_wanted value, and transitions to the Relaying state, where the process starts again.

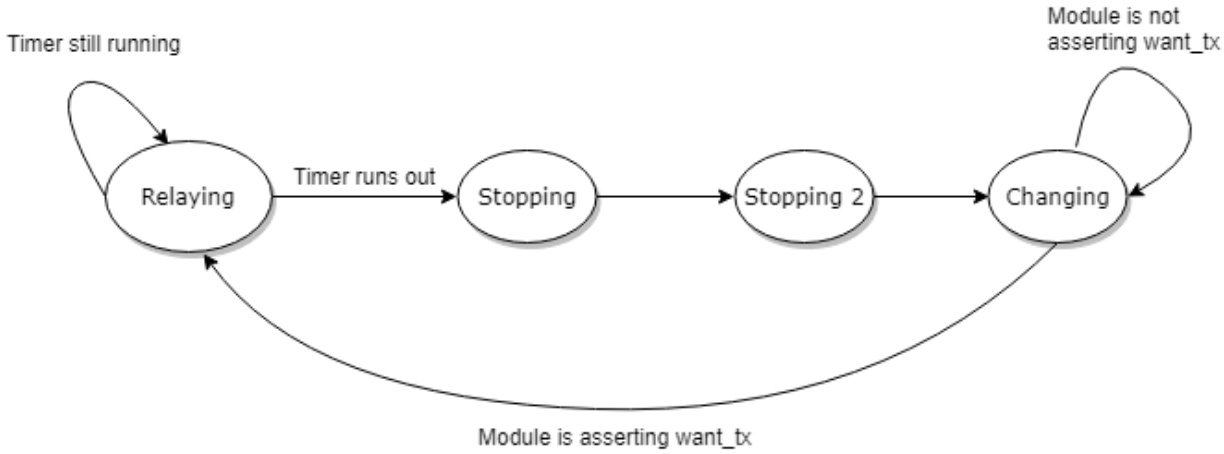

Figure 4: The Memory Multiplexer state machine

The Column Access Module

The purpose of the column access module is to provide data from the second operand matrix to the processing blocks and elements in column-major order. To this end it makes requests to system memory for the second operand matrix's data, caches the responses in the Pipelined Memory submodule, and maintains an interface with each data recipient that ensures both that the data gets across the boundary, and that the recipients all remain in lockstep. The operations of the column access module are triggered by the arguments_valid signal, and the module automatically resets itself to the beginning of the second operand matrix when all column data has been latched out. The overall interface for the column access module is shown below in Figure 5, while a close-up view of the interface that the column access module shares with processing blocks is shown in Figure 6.

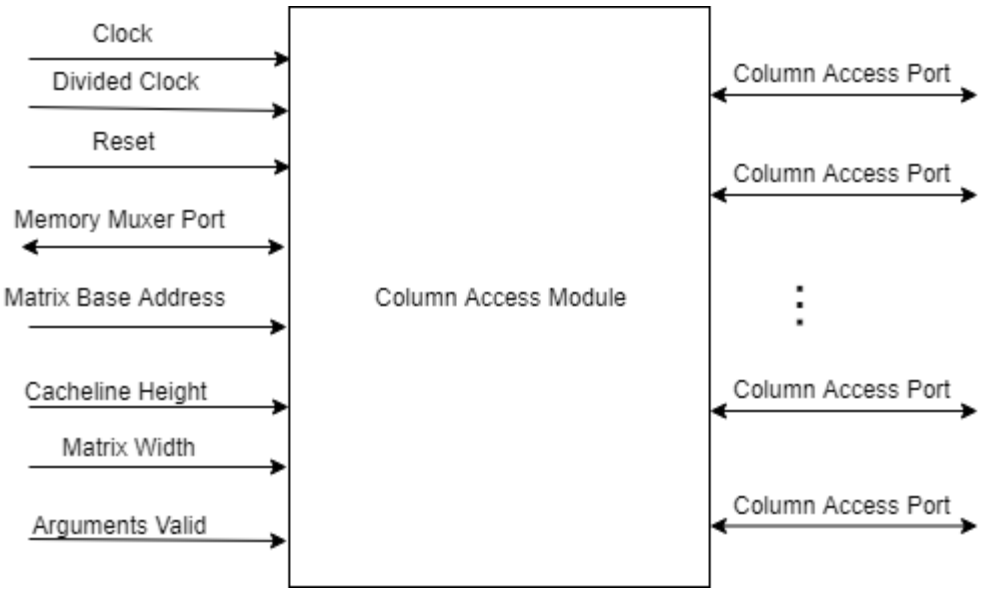

Figure 5: The column access module's interface

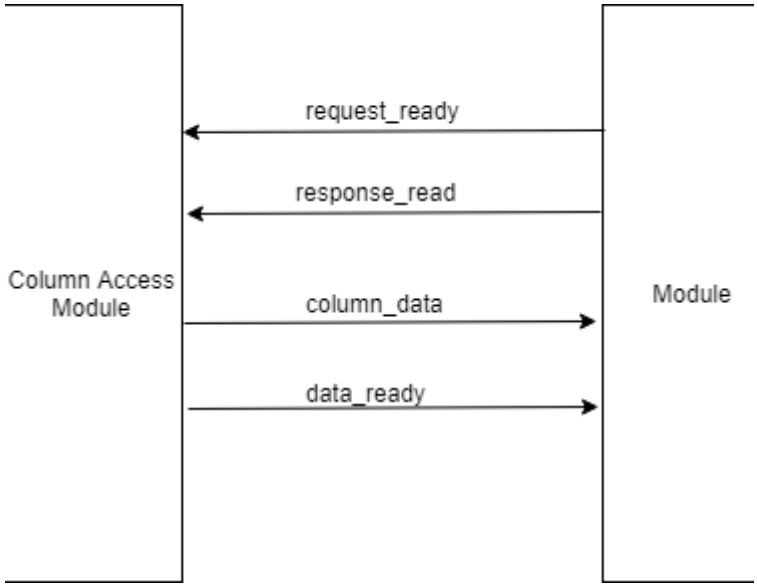

Figure 6: A close up view of each column access port

The column access module can be broken up into four main components: the read-request logic, the read-receipt logic, the column-data cache, and finally the latch logic. The read-request and latch-logic both have dedicated state machines, while the receipt logic consists of some

simple sequential logic that controls writes to the column-data cache, which is its own module. Each of these components is summarized below.

The Read-Request State Machine

The read-request state machine is a simple device that maintains the CCI-P interface via the memory multiplexer, and has only three states: Idle, Prep Request, and Send Request. The state machine starts out in the Idle state. When the module receives the arguments_valid signal, the machine latches in the base address and dimensions of the operand matrix, asserts the want_tx line on the module's memory muxer port, and transitions to the Prep Request state. In this state, the module prepares a memory request header with the current address as well as various tags and flags. Namely among these, the header will be tagged with an identifier marking the request as coming from the column access module, and will have a flag instructing the CCI-P interconnect not to bother caching the data, as it will only ever be read once. On the next cycle, the state machine transitions directly to the Send Request state. The send request state holds onto the request header until it gets all the necessary ready-signals. In particular, the memory muxer must assert the module's tx_ready line indicating that it is now relaying requests from the column access module, and the tx0_full line (which indicates whether there is room for another request on the CCI-P bus) must be de-asserted. Having to wait to send the request is a fairly common occurrence, as there are typically many other modules needing to make their own requests that the muxer may service first, and the system on the whole works to keep the bus as full as possible to maintain maximum throughput. When both signals are in their proper states, the state machine strobes the ready signal on the first channel of its TX bus. This causes the request to be sent along to the memory muxer, which in turn relays it to the main CCI-P bus. Once this happens, the state machine increments the current address to point to the next cacheline, and leaves the Send

Request state. If the sent request was for the last cacheline of the second operand matrix, the transition will be to the Idle state, where the state machine will remain until it needs to load a new matrix. If not, the transition will be to the Prep Request state, where a new header will be prepared with the new address, and the process will repeat. Figure 7 depicts the state machine implementing this functionality below.

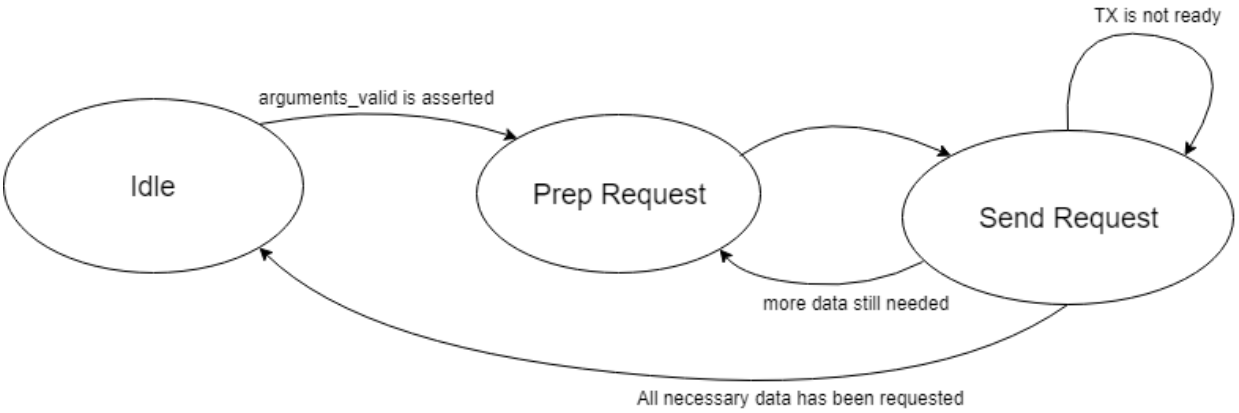

Figure 7: The read-request state machine

This state machine has a few unique aspects and peculiarities to it. Most prominent among these is the fact that there are two different states that alternate to produce and send memory requests. The purpose of this design decision is two-fold. The first and most important reason is a limitation of the CCI-P memory interface: a request can only be sent over it only every other cycle. Having two states that alternate will throttle the maximum frequency of request transmission down to once every other cycle without any further fuss. The second reason is that when building the RTL for this system for FPGA, the fitter tends to have a significant amount of trouble getting all necessary signals to meet timing constraints in the column access module. This is likely due to the fact that the data busses passing through the column access module are very large and require many registers and on-die wires to implement. This cannot be avoided, so the solution is to split operations over two cycles everywhere possible, and also to pipeline all low-

speed signals. Exactly which signals were pipelined and in what manner will not be discussed here, but can be seen in the column access module code attached in the appendix.

The Read Receipt Logic and Memory Cache

When data requested by the read request state machine eventually arrives, its arrival must be detected, and it must be stored in such a way that it can be retrieved when needed, potentially multiple times. Facilitating this is the pipelined memory cache module, and the various pieces of logic that control and interface with it. This logic requires no state machine, but does have some state. The memory muxer feeds all data received by the system to every module that maintains a memory muxer port, so the first step is to determine whether the data received originated from one of the column access module's requests. This is done by a partial Huffman coding. If the first bit of the metadata tag is zero, then the request came from the column access module, and if it is a one, then it came from one of the processing blocks' ports, and the preceding bits will identify which one. If the tags match, then the logic will strobe the data_arrived line on the input to the memory cache, which will in turn latch in the data sitting on its input (which is registered from the memory muxer port's receive-data bus). When this occurs, a counter indicating the number of cachelines received is incremented. The total number of cachelines in the entire operand matrix is precomputed, and once the number of cachelines received reaches that number, the logic asserts the ready_for_reading input line to the memory cache, which uses that signal to switch to reading mode.

The memory cache starts out in write mode, receiving data as it comes in, and writing it to its internal block RAMs. On the assertion of the ready_for_reading line, the module switches to reading mode, and makes the first cacheline available on its read_data output line. When the latch state machine is ready to get the next cacheline, it strobes the cache_ack_read signal, which

causes the memory cache to increment its internal address and fetch the next cacheline. When all data from the second operand matrix has been latched out or the module's soft_reset line is asserted, the module will jump back to the first cacheline and resume latching out cachelines sequentially as requested. The memory cache is its own module, and so will be discussed in more detail in the next module subsection.

The Column Data Latch State Machine

The column data latch state machine is responsible for maintaining the interface between the processing blocks and the column access module's internal memory cache module. This is a low speed interface, as the elements of a given cacheline of column data are consumed one at a time by each processing element in the system. Since each cacheline contains sixteen column elements, the column data latch need only work at the rate of one cacheline per sixteen cycles in order for the processing elements to still be able to work at full throughput. In addition, due to the design complexity associated with having one block RAM read port for every processing block, the state machine also latches the same cacheline out to each processing block in parallel. This requires that all processing blocks be in lockstep with one another, but since all processing blocks trigger at once and operate at full throughput this has little to no effect on latency. As a result of the low required speed and required synchronicity between multiple recipients, the column data latch uses a handshake-style method of data exchange as opposed to the streaming-style used in most of the rest of the system. In this method, the column data latch essentially treats the processing blocks collectively as a single data recipient. As far as the column data latch is concerned, the processing blocks are only requesting data when *all* processing blocks have their request_ready line asserted, and the processing blocks have only acknowledged receipt of data when *all* processing blocks have their response_read line asserted. This serves to achieve the

desired synchronicity without introducing significant design complexity. The state machine implementing this functionality is shown below in Figure 8.

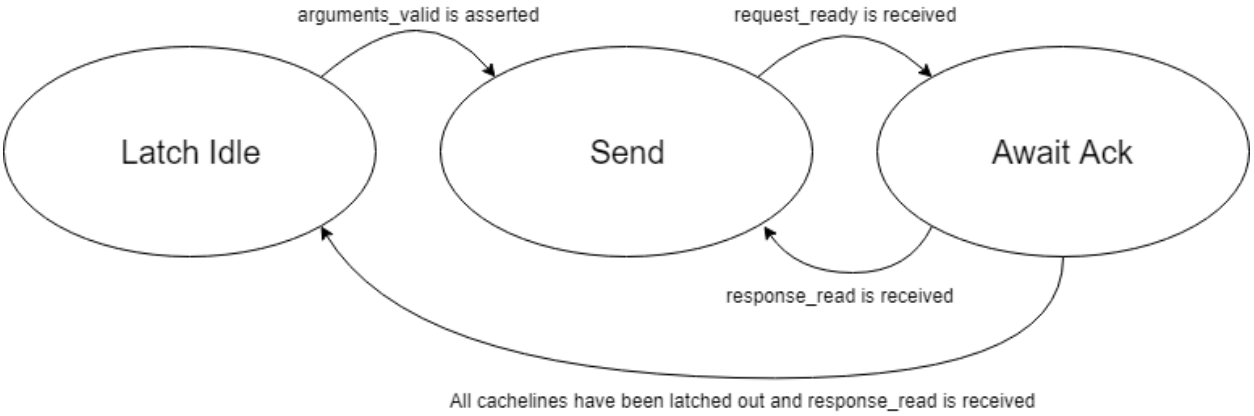

Figure 8: The column data latch state machine

The column data latch state machine has three states: Latch Idle, Send, and Await Ack. It starts out in the Latch Idle state. When the module's arguments_valid line is asserted, it transitions to the Send state. It will wait in the Send state until the column data has been received and the memory cache module has asserted its data_ready output line, and also until the processing blocks have all asserted their request_ready lines. In the system's final configuration, the memory cache module will not assert data_ready until all the data for the second operand matrix has been received, so once data latching begins, it need never be considered again. When both data_ready and the request_ready lines have been asserted, the state machine asserts its own data_ready output line to the processing blocks and transitions to the Await Ack state. It will wait in this state until every processing block has de-asserted its request_ready line, and asserted its response_read line. This ensures that the state machine does not move on until every processing block has read and acknowledged the new data. Once the Ack signals have been received, the state machine will de-assert its data_ready signal and leave the Await Ack state. If the last cacheline was just latched out, then the state machine will transition to the Latch Idle state where

it will await another arguments_valid signal before starting again (quite possibly on the same data). If not, then the state machine will transition back to the Send state where it will repeat the process.

As with the other components to the column access module, the column data latch is fairly heavily pipelined in spite of the relatively light combinational logic involved in its operations. Incoming handshake signals from the processing blocks are registered, and outgoing data and handshake signals are also registered, as well as some internal signals to the state machine. This is so when the system is being built for FPGA, the fitter will spend relatively little time on the latch state machine where having a single cycle of latency is just as good as having ten cycles, and instead focus on more involved modules. An overall summary of the column access module depicting both its interface with other modules and the interfaces between individual components is shown below in figure 9. In addition the "Memory Cache" seen in the figure is discussed in detail in the following subsection "The Pipelined Memory Module".

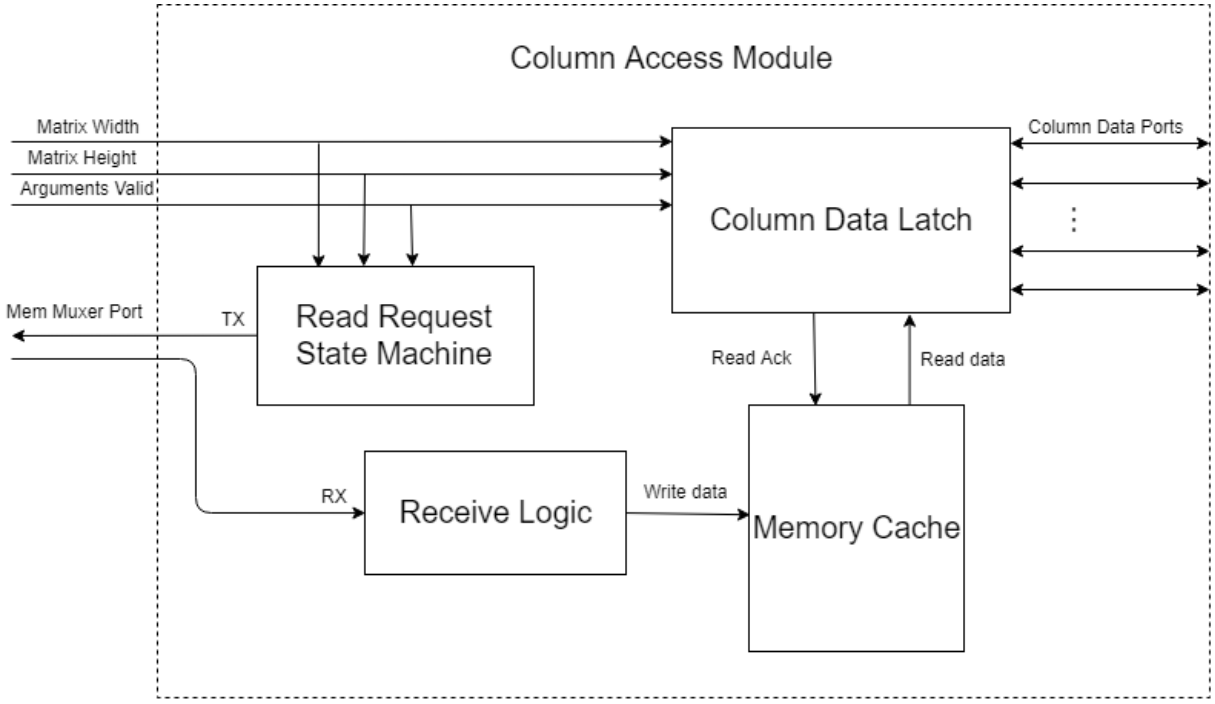

Figure 9: A more detailed view of the components of the column access module

The Pipelined Memory Module

The purpose of the pipelined memory module is to provide a means of storing the entirety of the second operand matrix such that it can be stored and retrieved in a reasonably efficient manner. In particular, the memory must be able to write a cacheline on every cycle, and must be able to read cachelines at the rate of at least one cacheline per sixteen cycles. The majority of iterations of the column access module design held the second operand matrix data in a large array defined directly in the Verilog RTL. This was functional, and can work at up to 200 MHz, but cannot reach the target clock speed of 400 MHz As the complexity and degree of pipelining of the memory increased, eventually it was necessary to break it out into a separate module. In its final form, the pipelined memory module is heavily pipelined and dual-clocked. The block RAM IPs are in a divided clock domain, and clock-crossing FIFOs are used to allow reads and writes to get across the border without any metastability issues. A summary view of the pipelined memory module is shown in Figure 10.

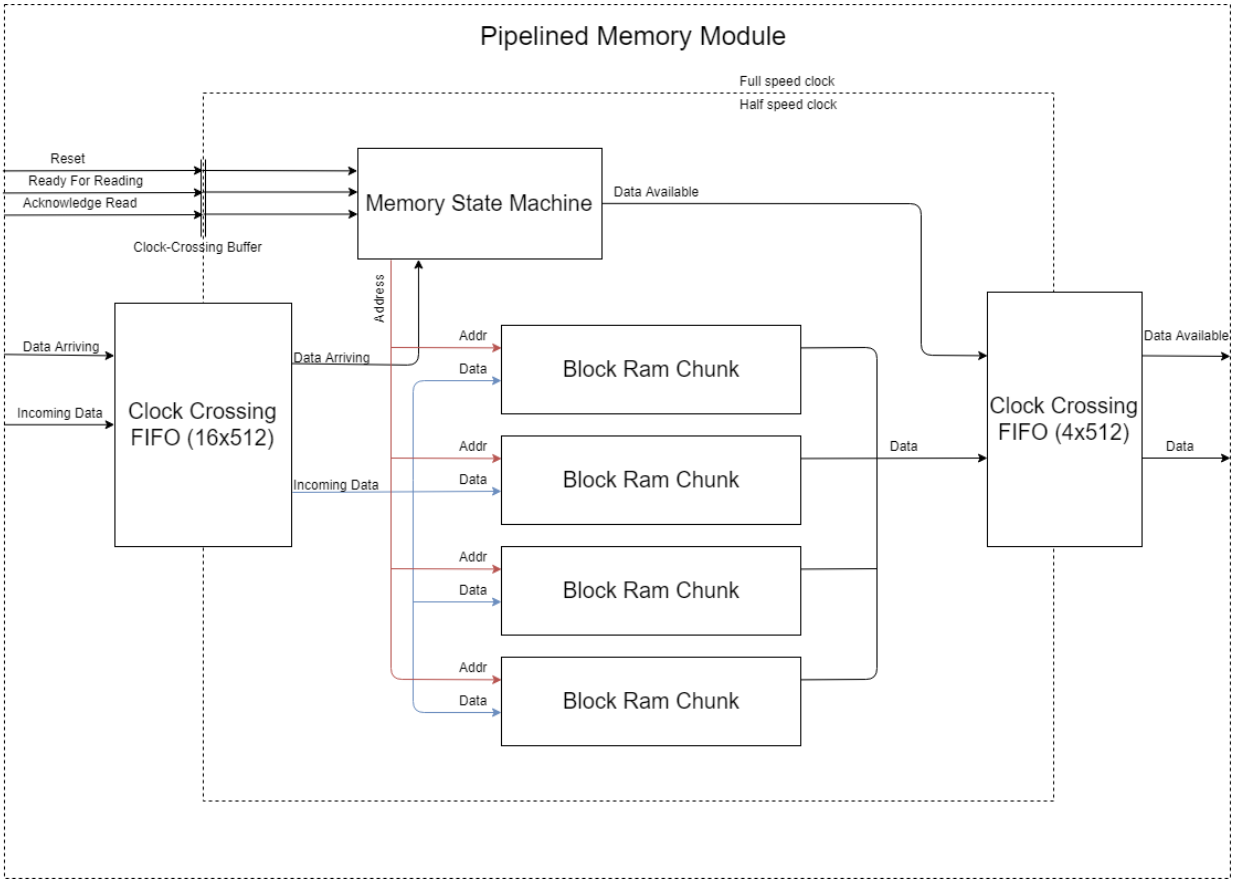

Figure 10: The pipelined memory module. The outer dotted line box shows the borders of the module itself, and the inner dotted box shows the borders of the divided clock domain.

A natural consequence of clocking the memory at half the design clock speed is that writes to the memory can only occur every other cycle (from the perspective of the CCI-P bus). Since the CCI interface allows for read responses to arrive on consecutive clock cycles, this presents a problem that must be rectified. Fortunately, since memory requests can only be sent every other cycle, responses must on average arrive no faster than every other cycle. Thus the divided clocked memory will never fall behind, provided the read data is sufficiently buffered. The column access module is configured to request sixty-four cycles of access to the main CCI-P bus from the memory multiplexer, so the column access module may send at most thirty-two read requests in a burst before getting preempted. Thus to account for the unlikely event that those thirty-two read

responses arrive on thirty-two consecutive clock cycles, the write-side clock-crossing FIFO is set to be one cacheline wide (512 bit), and sixteen cachelines deep. (Over the thirty-two cycles, the memory can consume sixteen responses, so only the second sixteen need be buffered).

A peculiar feature of the pipelined memory module is that instead of having one single block RAM IP, it has four, each one containing one fourth of the column data. This is a measure to relieve timing pressure for the fitter. Not shown in the diagram are multiple stages of pipelining that ensures that even though the (quite large) memory IPs may be physically far apart from each other on the die, each one is small enough to propagate signals in a single cycle, and input and output signals will still reach them. In particular, data exiting the write-side clock-crossing FIFO is split into the four chunks that will go into each block RAM, and then registered before entering the IP. Similarly, the address line exiting the memory state machine is split into four identical signals, and then registered individually. This divides the fanout of the address signal by four, and also allows the fitter to place the address register entering into each IP physically close to that IP on the die, instead of having a single address bus having to traverse long distances to get to some or all of the memory IPs. In a similar manner, read data leaving the memory IPs is registered immediately, and only then combined to get a full cacheline of data before it is again registered in the read-side clock-crossing FIFO.

The Pipelined Memory Module State Machine

The pipelined memory module state machine's purpose is to control when the module is reading versus writing, and also to manage ready-signals. It, like the block RAM IPs and all the pipelining stages, is clocked at half the design clock speed. Like many of the modules in this design, the state machine is relatively simple with only three states: Writing, Reading, and Read (note that "Read" is past-tense here). Figure 11 below depicts this state machine. The machine

starts out in the Writing state, with a zeroed address. On every cycle where data is available in the write-side FIFO, that data is fed through as described above. The data-available line is buffered by a single cycle before being fed into the write-enable line of the block RAM IPs to ensure that it arrives at the same time as the data. The unbuffered data-available line is fed into the state machine, which increments the address immediately. Since the address signal is pipelined in the same manner as the data and enable lines, this has the effect of the address signals going into the memory IPs incrementing immediately after the data arrives at them. This allows data to be read from the write-side FIFO on every clock cycle, which is necessary to guarantee that the memory module never falls behind due to its slower clock.

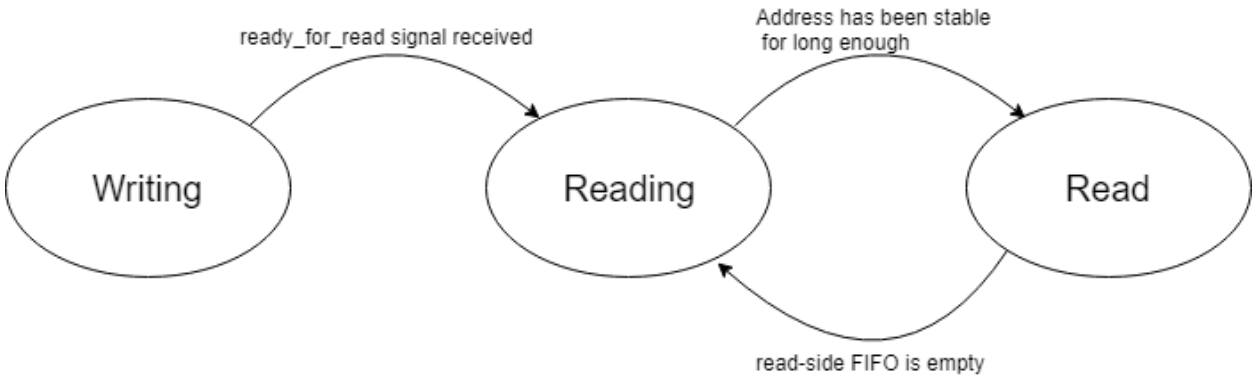

Figure 11: The pipelined memory module state machine

When the module receives the ready_for_reading signal from outside, the state machine transitions from the Writing state into the Reading state. Because the column access module is configured to wait until it has received all data from the second operand matrix before beginning to latch out data, the pipelined memory module will never need to alternate between writing and reading. Once in the reading state, the address signal going into the block RAM IPs is reset to zero, and latency-enforcing logic kicks in. A separate sequential logic circuit detects changes in

the address signal and resets a counter. The latency between changing the address signal at the state machine level and the data from that address appearing at the inputs to the read-side FIFO is three cycles. The circuit increments the counter for every cycle that the address signal is stable. When the counter hits the necessary three cycles, the write-enable line going into the read-side FIFO is strobed, and the state machine transitions into the Read state. An empty-signal on the divided clock-side of the read-side FIFO will then report that the FIFO holds data. Meanwhile a similar signal on the output side of the FIFO will let the column access module know that data is available. When the column access module latches in that data and acknowledges it, the read-side FIFO will once again report as empty. Once the empty signal is asserted, the state machine will transition from the Read state back into the Reading state, incrementing the address signal by one. This in turn triggers a reset of the latency counter, and the process repeats.

The Processing Block Module

The processing block module is the coarsest level of subdivision for the processing functionality of the system. It is intended to be tiled, as many as will fit on the FPGA die. They can be dynamically assigned data and then set to work, and work independently of each other, only requiring access to memory muxer ports, and to column access data ports. Each processing block is assigned a ‘ribbon’ of sixteen rows of the first operand matrix, and computes the dot product of each of the sixteen rows with each column of the second operand matrix. It writes back the accumulated results as it finishes them, and reports back to the top-level controller once done, where it may be assigned another ribbon if there are still more rows to compute. The processing block directly contains no calculation logic, and instead serves as a container of other interconnected modules, namely the Processing Block Memory Controller, the Writeback module, and sixteen Processing Element instances. The processing block at the top level also has no state,

bar some inputs that are registered for pipelining purposes. It uses two memory muxer ports, and a column data port as well as various parameter and signal lines as part of its interface, and has the longest signature of any of the modules. Figure 12 depicts a summary of this interface, as well as the interfaces between the various submodules of each processing block. The three types of module seen in the diagram are discussed in detail in the following three subsections.

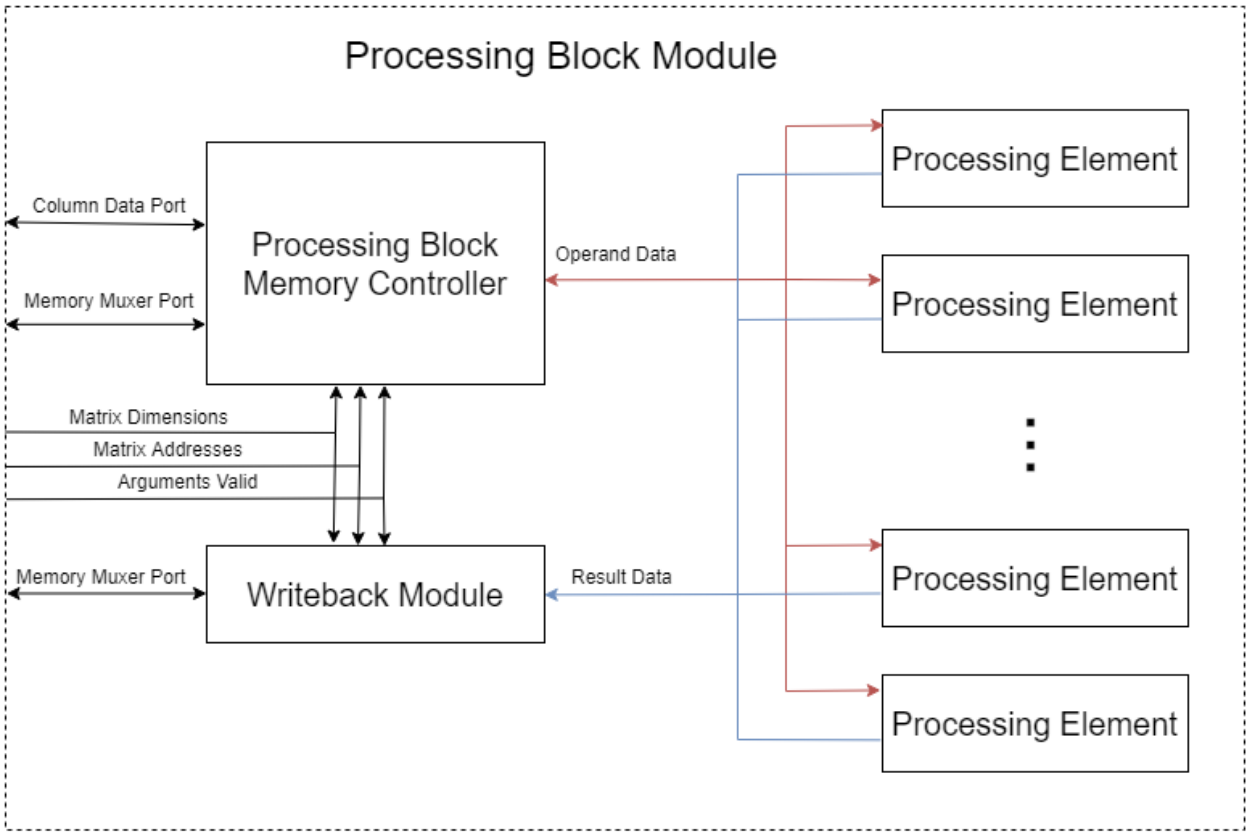

Figure 12: A summary block diagram of the Processing Block Module

The Processing Block Memory Controller

The purpose of the processing block memory controller is to request, receive, and store data from the first operand matrix, to retrieve data from the second operand matrix, and to latch both data to the processing elements with full throughput (i.e., one element per clock cycle, sustained). To implement this functionality, the PB memory controller houses three state machines, one submodule, and various pieces of combinational and sequential logic linking the

four sub-components together. The first state machine is the memory request state machine, which is very similar to the memory request state machine present in the column access module. The second is the column data latch state machine, which is responsible for maintaining the interface with the column access module and retrieving column data before it is needed. The sub-module is the PB memory cache module, which is a somewhat pipelined series of block RAM IPs responsible for storing and retrieving row data as the memory requests sent by the first state machine arrive. Finally, the processing block data latch state machine is responsible for reading row data out of the PB memory cache, and getting column data from the column data latch, and sending the proper data to each of the sixteen processing elements via the processing data ports. A summary view of this module is shown in Figure 13 below.

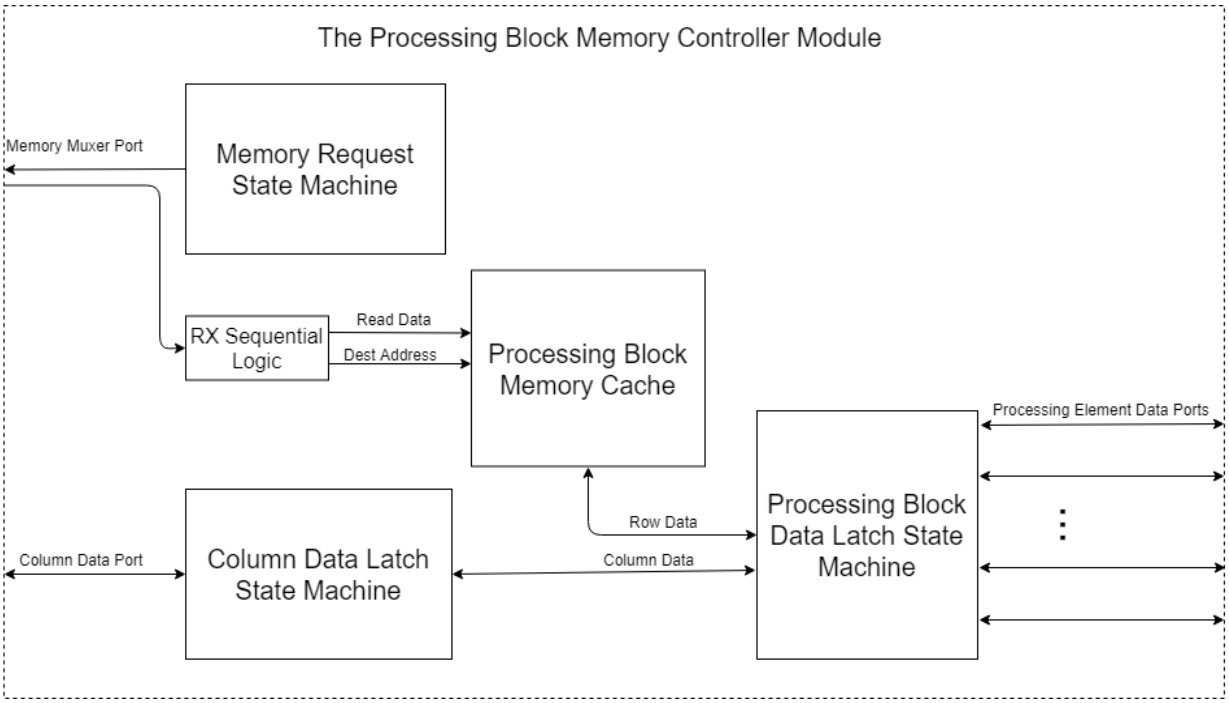

Figure 13: A summary view of the processing block memory controller module

The Column Data Latch State Machine

The column data latch state machine is responsible for making data from the second operand matrix available to the processing block data latch state machine so that it can continuously latch out matrix elements to the processing elements. To do this, it keeps track of both the current cacheline of column data and the next upcoming cacheline of column data. The PB data latch state machine provides a latching_last_thing signal that indicates that the data latch is currently latching out the last element of the current cacheline. When this signal is received, assuming that the next cacheline is available, it will be 'demoted' to the current cacheline on that cycle, allowing the PB data latch to keep latching out data with no pauses in between cachelines. To implement this functionality, the column data latch state machine has seven states: Idle, Wait For Rows, Get Current, Await Ack-Current, Get Next, Await Ack-Next, and Wait. These are all shown in Figure 14 below.

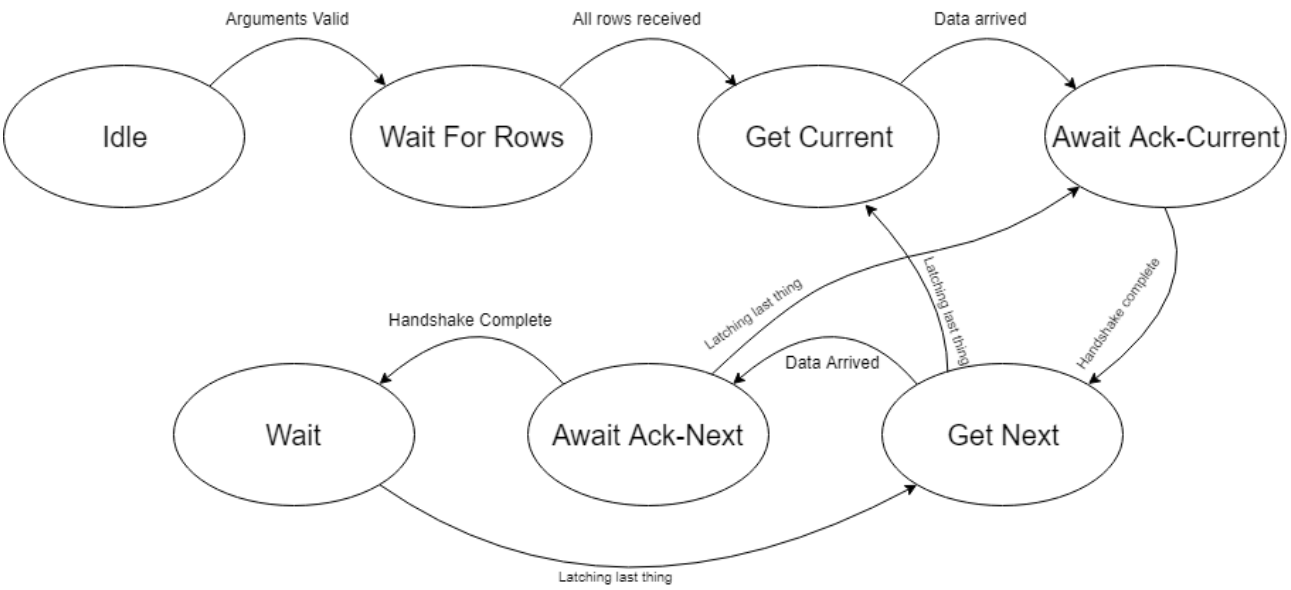

Figure 14: The column data latch state machine

The state machine starts out in the Idle state. When the arguments_valid signal is received, it transitions to the Wait For Rows state, while in this state, the machine effectively continues

idling, but is waiting for the sequential logic responsible for tracking read request responses to signal that all row data has been collected. This ensures that data latching does not begin before all the necessary data is available. Once the signal has been received, the state machine transitions to the Get Current State. In this state it asserts the request_ready line on the modules column data port, and waits for the column access module to assert the data_ready line. Once this occurs, the state machine latches in the data on the column data port, marks the current cacheline data as being ready, de-asserts its request_ready line, asserts its response_read line, and transitions to the Await Ack-Current state. In this state the machine waits for the column access module to de-assert the data_ready line, which indicates that it is now safe to request the next cacheline. Once this has happened, the machine transitions to the Get Next state where it again requests a cacheline just like in the Get Current state. When the next cacheline arrives, the machine transitions to the Await Ack-Next state, where it again awaits the acknowledgement of the column access module. Once this has been received, the machine transitions into the Wait state, where it waits for the latching_last_thing signal to go high.

The reason that there are separate pairs of states for requesting the current and next cacheline is so the process of demotion works properly. The column data latch state machine needs to have the next cacheline available before the data latch state machine needs it to ensure full throughput. To accomplish this, if the state machine is in the Wait state, then both the current and next cachelines are ready to go. When latching_last_thing is received, the next current cacheline is thrown away, the next cacheline is demoted to the current cacheline, and the state machine transitions to the Get Next state, so that it can get the new next cacheline. If the state machine is in the Await Ack-Next state when latching_last_thing is received, then as before, the next cacheline is demoted to be the current cacheline, but the state machine transitions to Await

Ack-Current, because the cacheline whose acknowledgement is being awaited is now the current one. This way, when the acknowledgement is received, the machine will continue on to Get Next as normal, and not before the column access module is ready. Likewise, if the machine is in the Get Next state, it will transition to the Get Current state upon receipt of the latching_last_thing signal. In this case however, the next cacheline is not yet available, so nothing is put into the current cacheline, and it is marked as unready. This is considered an error condition, as it means that the latching of data has to stop in the middle of a column. The final configuration of this design is set so that it is impossible for this particular error condition to occur, but more error tolerance is never a bad thing.

The Data Latch State Machine

The Data Latch State Machine is responsible for combining the column data received by the column data latch state machine with row data read from the memory controller's row-data cache, and latching both sets of data out to the processing elements in sync. In order for the processing block to have reasonable performance, this machine must be able to send a pair of elements to each processing element on every clock cycle, for a rate of sixteen multiplies per cycle. To accomplish this, it first waits for both the row data cache and the column latch state machine to have data available. It then lets the row data cache know to start latching out row data, and tracks the index of each element as it comes through. The element index is used to identify which element of the current column cacheline to send along. Row data comes at the rate of one cacheline per clock cycle, and each cacheline is split into sixteen elements which are fed into the "A" busses of the sixteen processing elements. The column data comes at the rate of one element per clock cycle, and all processing elements get the same column data element fed into their "B" bus on a given clock cycle. Combinational logic generates the latching_last_thing signal that the

column data latch state machine discussed above uses. When the element index is close to the end of the current cacheline, the latching_last_thing signal will automatically strobe, letting the column data latch state machine know that it needs to replace the current cacheline with the next one. The state machine also responds to back-pressure from the processing elements. It will not start latching until all the processing elements have asserted their ready-signal. The state machine that implements all this functionality is shown below in Figure 15.

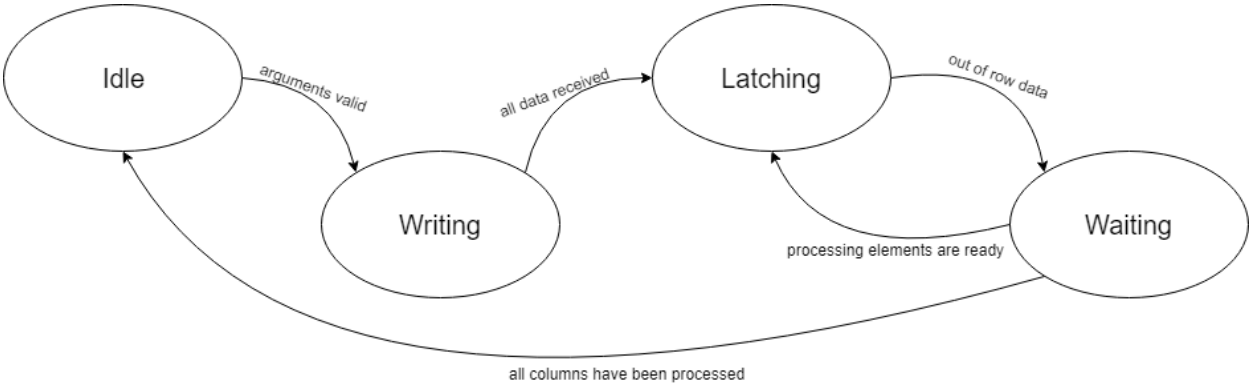

Figure 15: A diagram of the data latch state machine

The state machine starts out in the Idle state, and transitions to the Writing state when the arguments_valid signal is received. In this state the machine is listening for responses to read requests. When one arrives over the memory muxer port, it is fed to the address in the row data cache corresponding to that response's metadata tag, and a counter is incremented. When the counter shows that every response has been received and the processing elements all have their ready-lines asserted, and the column data latch state machine has data available, the state machine transitions to the Latching state. In this state, it signals the row data cache to start sending out data, and maintains the element index as mentioned above. Data starts flowing through the machine at the rate of sixteen pairs of matrix elements every cycle. This continues until the machine has moved the entirety of the sixteen rows assigned to it, and one column of the second operand matrix through to the processing elements. Every sixteen cycles, the latching_last_thing

signal strobes, causing the column data latch state machine to replace the column data that this state machine is reading from with the next cacheline of column data that it in turn previously requested from the column access module. When the entirety of the row/column pairs have been processed, the machine transitions to the Waiting state, where it waits for the processing elements to send the write request containing the sixteen elements of the result matrix that have just been computed. Once all processing elements once again have their ready-line asserted, the state machine may do one of two things. If there are still more columns to iterate through, then the machine will transition back to the Latching state, starting back at the beginning of the row data, and now iterating over the next column of the second operand matrix. If there are no more columns to iterate over, then the machine will transition back to the Idle state and await a new assignment of rows.

The Processing Element Module

The processing element module is the computational core of this design. All the other modules are, in the end, a means to get data to the processing elements, or a means to get results out of the processing elements. Each processing element instance houses a simple state machine and a single multiply-accumulator IP. The state machine maintains the interfaces that the processing element has with the processing block memory controller on the incoming-data side, and with the writeback module on the outgoing-data side. The multiply-accumulator will compute and accumulate the dot product of a single row of the first operand matrix, and a single column of the second operand matrix. When this is complete, the total is sent to the writeback module, and then the processing element prepares to compute the next row/column pair. Figure 16 depicts the structure of the processing element module in a simple block diagram.

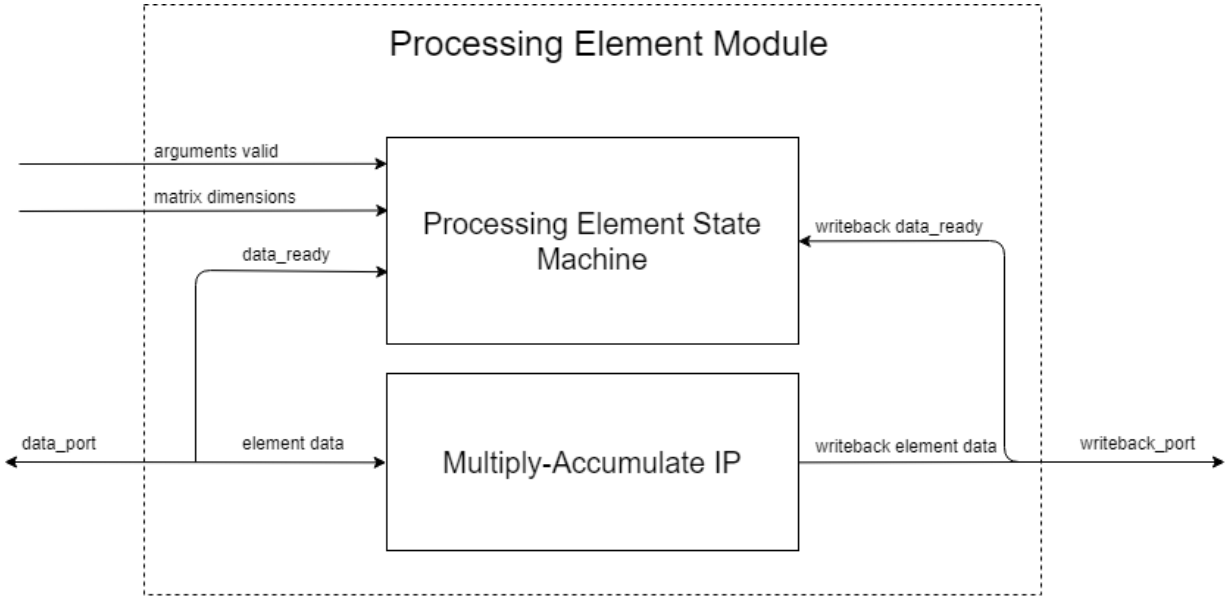

Figure 16: A block diagram of the processing element module

Zoomed-in views on the data_port and writeback_port interfaces seen in Figure 16 are shown below in Figures 17 and 18 respectively. Finally, Figure 19 shows the state machine diagram for the processing element state machine.

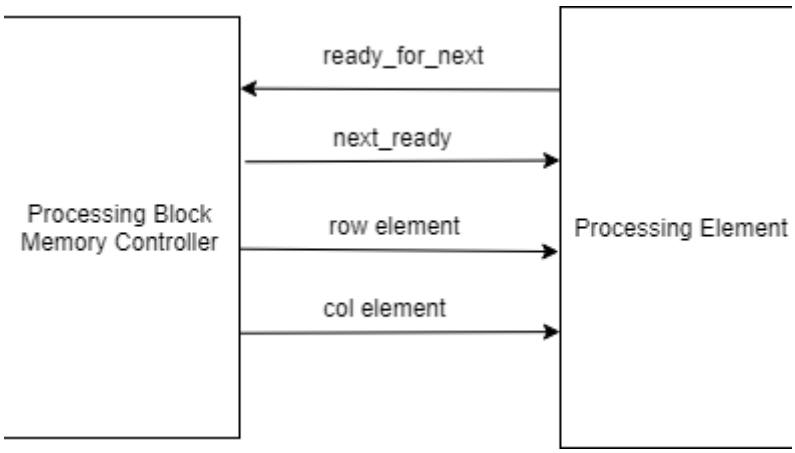

Figure 17: A close-up view of the data_port interface that each processing element uses to communicate with the processing block memory controller

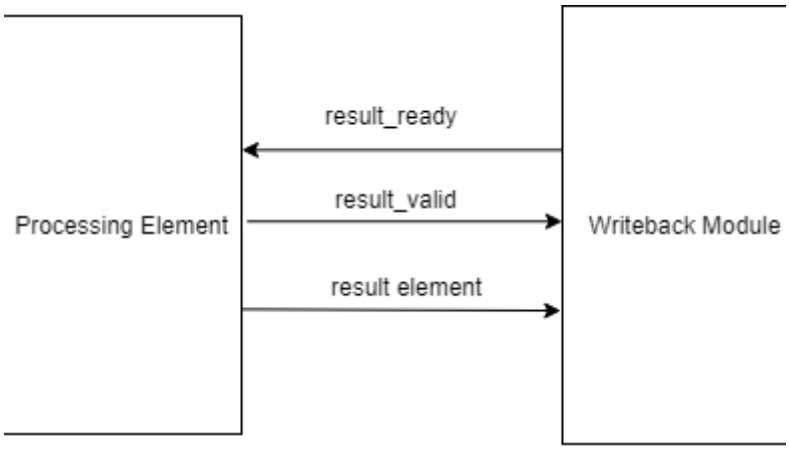

Figure 18: A close-up view of the writeback_port interface that each processing element uses to communicate with the writeback module

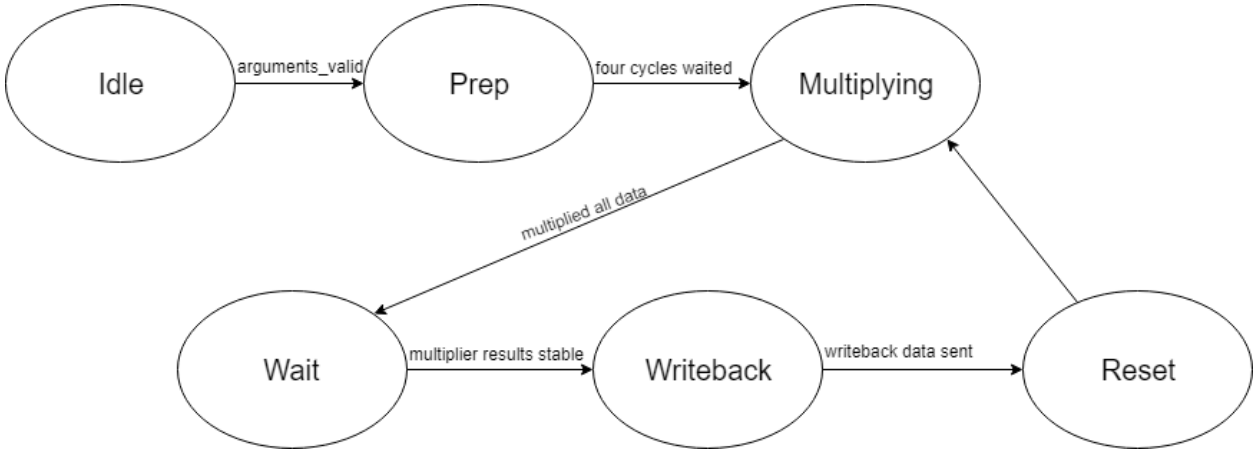

Figure 19: The processing element state machine

The processing element state machine starts out in Idle. When the arguments_valid signal is received, the machine transitions to the Prep state. In this state, the multiplier IP is enabled, and allowed to sit for a total of four clock cycles before the state machine transitions to the multiplying state. In the multiplying state, the ready_for_next line is asserted, and the processing element waits for data to start arriving. Once it does start to arrive (signaled by the next_ready line getting asserted), the processing element independently counts how many elements have

arrived. As the last element approaches, the processing element will de-assert its ready_for_next line. After the last pair of elements has passed through, the machine transitions to the Wait state. The multiply-accumulator IP used has a latency of four clock cycles, so data fed into its inputs does not reflect in the output until four cycles afterwards, so to get correct outputs, the processing element must wait four cycles before continuing. Once this wait period is complete, the machine transitions to the Writeback state, where the output of the multiply-accumulator is placed on the result element bus of the writeback port, and result_ready is asserted . Once result_read is asserted by the writeback module, the processing element transitions to the Reset state, which triggers a reset of the multiply-accumulator, and then it transitions back to the Multiplying state, where the process repeats.

CHAPTER V

PERFORMANCE

Upon completion of the design, benchmarking was done to determine how it performed. This was done with the use of the driver C++ program (which is included in the appendix), and several helper scripts. In particular, the driver program builds two square matrices of a given dimension, and writes them into a buffer that is shared with the hardware accelerator. It then starts a timer, gives the hardware accelerator the signal to start multiplying by performing a write to the accelerator's fifth CSR (Control Status Register), and waits for the signal that the operation is complete by monitoring the status of the zeroth CSR, and finally reports back results. The benchmarking scripts build a picture of the system's overall performance by repeating this process many times on matrices of different sizes.

When the design was first fully debugged and working, performance was relatively poor. This is shown in Figure 20. This is the result of two major factors: a relatively low level of parallelism, and low clock speeds. The initial implementation of the design was intended to be able to multiply matrices of up to 1024 elements to a side. In order to accomplish this, the memory caches in the accelerator needed to be able to store up to as much data as a 1024 matrix needs. This turned out to be very difficult to achieve, especially for the column access module, which needed four mebibytes of on-die block rams to hold all the data it needed. Four megabytes of memory is over half of what the Arria 10 FPGA has in total, leaving little left over for processing and controller overhead, and for the processing blocks' memory caches. In response to this, the maximum matrix size was reduced to 512 elements, which caused the memory requirements of the column access module to shrink by a factor of four, down to one mebibyte.

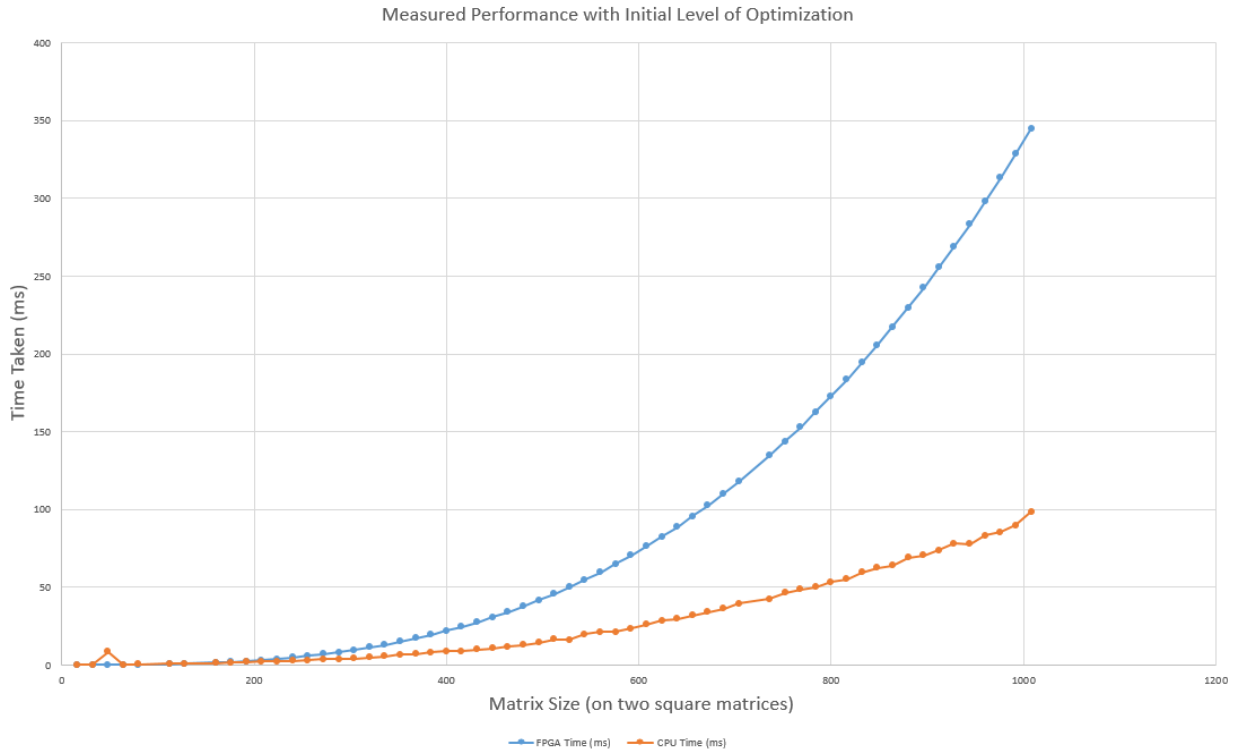

Figure 20: The initial performance seen by the hardware accelerator. The time that the accelerator spends is shown in blue, while the time spent doing the operation purely on the processor is shown in orange.

After the initial shrink of maximum matrix size, some minor tweaks were made to get higher efficiency in internal transactions. In particular, the amount of time that the processing elements spend waiting for resets and multiplier outputs to stabilize was reduced to the minimum possible values (a difference of four clock cycles per column processed). Next, the number of ports that the memory multiplexer provides was reduced so that it tightly fits the number required by the number of processing blocks in the current build of the design. This was made based on the observation when analyzing the code that even when no other modules are requesting access to the CCI-P bus, a writeback module may need to wait tens of cycles to obtain access. This was a result of the fact that the multiplexer cycles through its ports at the rate of one per cycle when looking for one requesting access to the memory bus. Consequently, if there are thirty-two ports,

the writeback module may need to wait at most thirty-two cycles between requesting access to the bus and receiving it, just to send a single write request. By reducing the number of ports, the worst-case wait was improved to eight cycles, which is imperfect but still a large improvement. After these incremental improvements, the accelerator managed to match, but not beat what the server-grade processor used on the HARP system could do on its own. Figure 21 shows the performance seen after these changes.

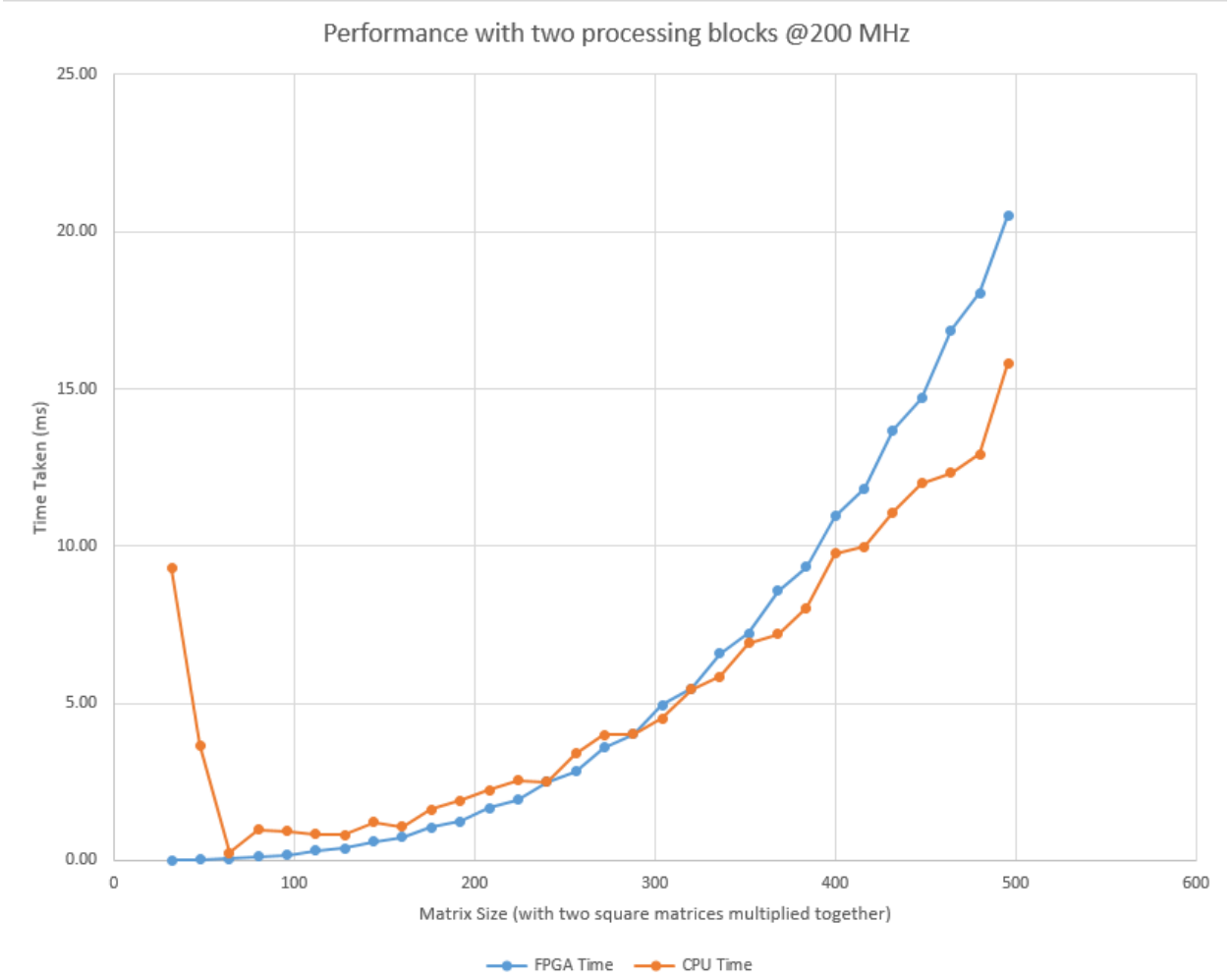

Figure 21: The performance of the hardware accelerator after some incremental improvements. Again, the time taken by the accelerator is shown in blue, and the time taken with a CPU multiply is shown in orange.

The next primary candidates to obtain larger performance improvements were to increase the design clock speed, and to increase the number of processing blocks tiled in the design. The clock speed was an especially attractive option, as memory bandwidth on the CCI-P bus is directly proportional to the clock speed, and so are the number of multiplies per second. The initial performance measurements for this system were obtained at a speed of 200 MHz, but the maximum speed on the CCI-P bus is 400 MHz. It is very challenging to get a design with any complexity up to this speed, but based on slack measurements when builds were attempted, it was deemed possible ("slack" is a measure of how close a design is to meeting timing. When a path between two registers on the FPGA die is too long, a slack of, for example, -0.123 indicates that a signal passing between the two will arrive 123 picoseconds too late). A significant amount of development time was devoted to reworking the implementation to be pipelined in an incredibly aggressive manner, so that it could reach the desired speed. This included introducing pipeline stages in between individual or pairs of combinational logic blocks, breaking up any indexing of arrays into multiple stages, and registering both incoming and outgoing signals on most of the modules. This can be seen especially clearly in the column access module, where there are almost no unregistered signals or purely combinational assignments. Exactly which signals are registered, and in what manner is too tedious a topic to expect any reader to follow, so refer to the Verilog code describing these modules in the appendix for more detailed information on the pipelining.

Eventually the system was made to work at the desired 400 MHz, and an immediate performance improvement was seen. Because both the memory bandwidth and the number of multiplies per second doubled, the speed of multiplication for matrices almost doubled. This is seen in Figure 22.

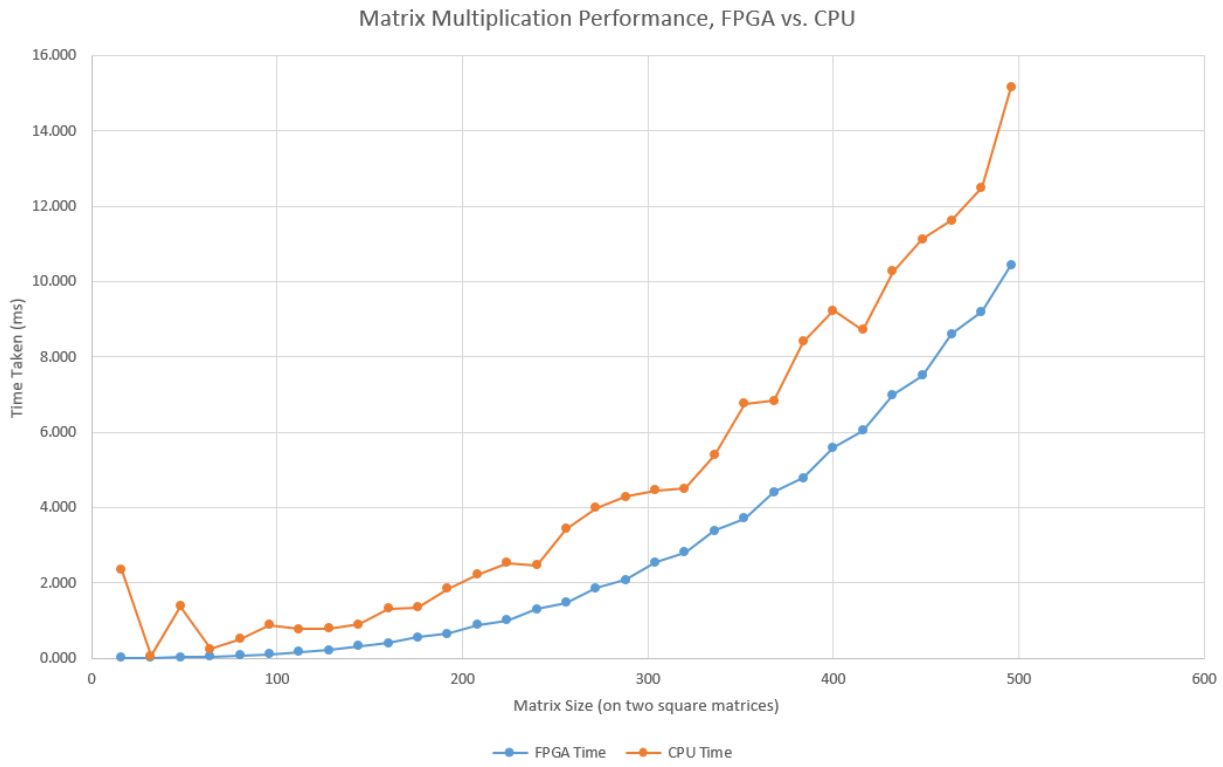

Figure 22: The final performance of the hardware accelerator, with two processing blocks clocked at 400 MHz. As before, the time that the hardware accelerator takes is in blue, while the time that the system processor takes to do the same operation is in orange.

In addition to timing measurements, data is also collected on when modules start and stop working, and in the case of the processing elements, how many cycles are spent on each task. Figure 23 depicts one such measurement on a processing block when working on a single ribbon of rows of a 512 element product matrix. The figure shows that the processing block spends only a very small amount of time waiting on data from the memory, and spends the vast majority of its time latching the data out to processing elements. This is as expected, because even though the time spent waiting on row data is relatively long, on the order of thousands of cycles, that data has to be iterated over not once, but once for every column in the second operand matrix, which in this case means 512 iterations. This says little about the efficiency of the actual computations, but

does indicate that memory accesses by the processing block are not taking more time or resources than they should.

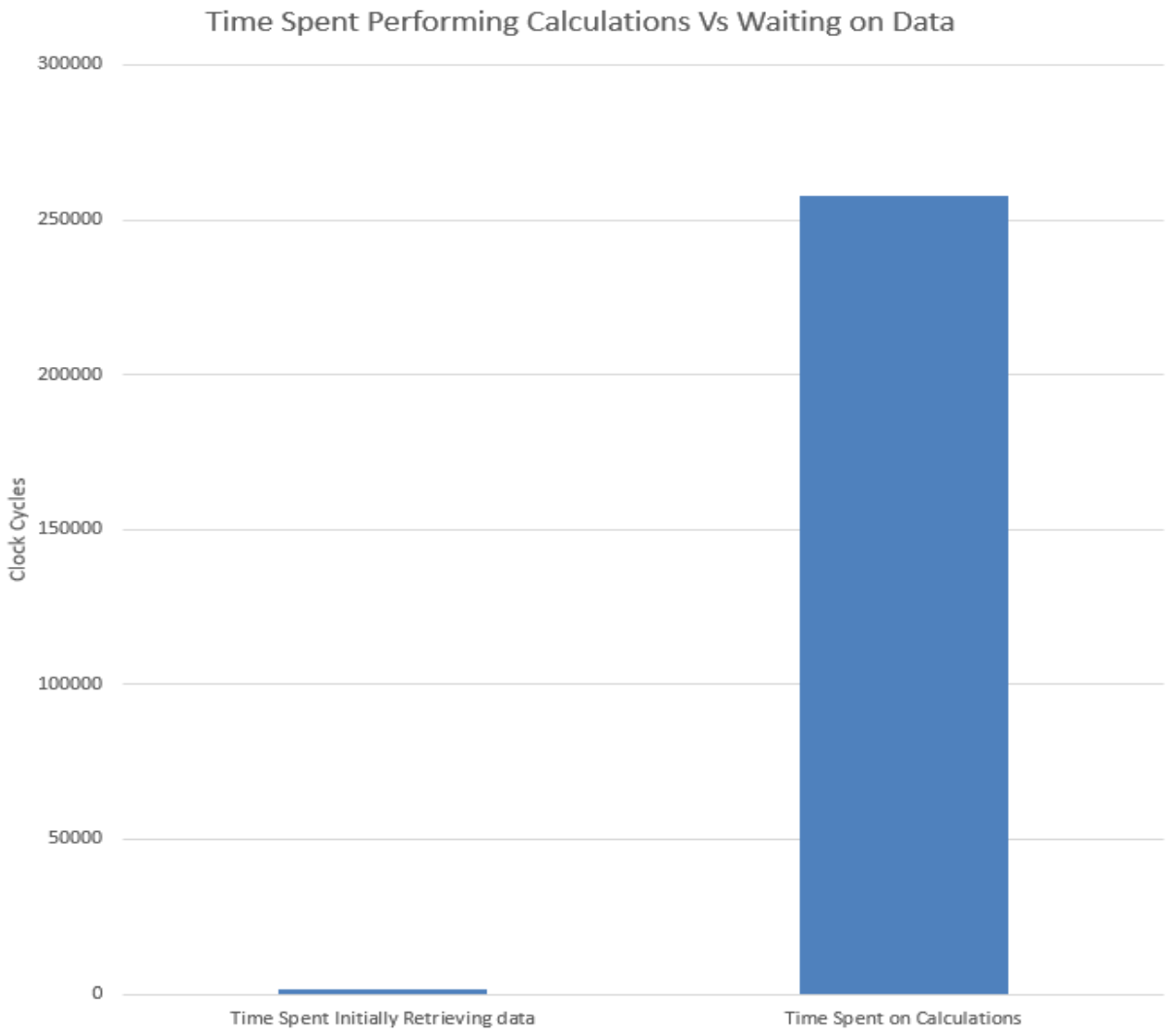

Figure 23: The number of cycles a processing block spends on retrieving data, and on latching it to processing elements when working on a ribbon of rows of a 512 element product matrix.

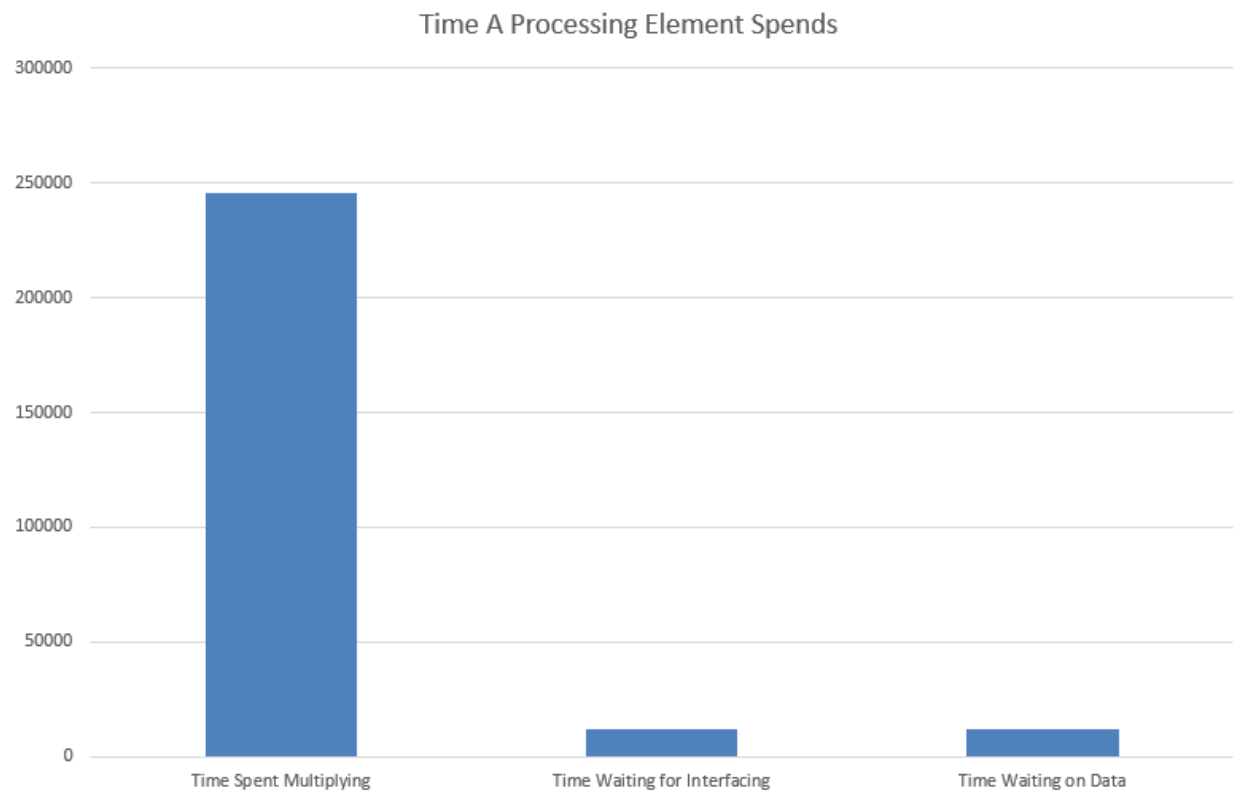

Figure 24: How many cycles a processing element spends on various tasks when operating on 512 element matrices.

To obtain a more detailed picture on what happens at the processing element-level, Figure 24 depicts how many cycles a processing element spends on each subtask as it is working on 512 element matrices. The first bar is the number of cycles spent multiplying. In particular, it means that the cycle in question was spent multiplying a single pair of elements together and adding the result to the accumulator. The next bar indicates time waiting for interfacing, such as when the processing element is waiting for the writeback module to acknowledge its result, or for the reset line to be held high on the multiply-accumulator for long enough. Over the course of a 512-element multiply, this time adds up to be approximately eleven-thousand cycles. Since the multiply-writeback-reset cycle happens 512 times, this corresponds to an average latency of twenty cycles for every 512 cycles spent multiplying. In addition to delays associated with

waiting for resets and writebacks, there are also delays resulting from waiting for the processing block memory controller to be ready to start sending data, which is shown in the third bar of Figure 24. The bulk of this delay is accrued as the processing block is waiting to receive row data, and it also amounts to approximately eleven thousand clock cycles on the maximum matrix size. Overall the delays that the processing element sees causes it to have an efficiency of approximately 92% (time spent multiplying divided by time spent waiting). This is imperfect, but due to Amdahl's law even a 100% efficiency rating would only result in a modest improvement for a potentially large amount of work.

CHAPTER VI

FUTURE WORK

The approach to matrix multiplication discussed in chapters 3 and 4 is not the only way to multiply two matrices together. That method (which I will call the standard approach) essentially focuses on doing the naïve method of matrix multiplication, where each row of the first operand matrix is multiplied with each column of the second operand matrix to produce each element of the product matrix. While this method is easily parallelizable, it has one major downside: it has a computational complexity of $O(n^3)$, which is not the best achievable complexity. This causes large penalties to be imposed when operating on very large (n of several hundred) matrices.

One method of matrix multiplication that offers a better time complexity than the standard method is called Strassen's algorithm. Strassen's algorithm operates by dividing the operand matrices into eight block matrices, each one being one-quarter of an operand matrix. Seven intermediate product matrices are then formed through the multiplication of various combinations of block matrices, and then the product matrix is formed from sums of different intermediate matrices. The standard approach, if applied to the block matrices in the same way, would require eight multiplies between matrices of size $\frac{n}{2}$, whereas Strassen's algorithm requires only seven. This means a roughly fifteen percent decrease in the number of element-multiplication operations required, at the cost of many additional addition operations being required. However, since multiplication has a complexity of $O(n^2)$ while addition has linear complexity, the method still offers a theoretical performance boost. When applied recursively, Strassen's algorithm achieves an overall computational complexity of approximately $O(n^{2.8})$, which for very large matrices (n of

at least several hundred), means having to do significantly fewer multiplies. Overall, performance gains still tend to be small though, as the added time required from having to do many addition operations, from cache misses due to nonsequential memory access, and from controller-level overhead tend to reduce performance far below what the algorithm is theoretically capable of.

Another method of matrix multiplication achieves performance gains by not reducing computational complexity, but by eliminating repeated reads of the same data from memory. This method was devised by Scott and Khatri, and will be referred to here as Scatter multiplication [2]. Scatter multiplication operates by reversing the data order of the operand matrices as it is read in. In other words, the method reads in a column of the first operand matrix, and a row of the second operand, and multiplies those together (as opposed to a row of the first operand and a column of the second). When this is done, every unique pair of elements from the two vectors produces a partial product of a different element of the product matrix. Once the product of every unique pair of elements from the two vectors has been computed, the data from both vectors has made a contribution to every part of the product matrix, and thus will not be needed again. Since the naïve approach typically requires that one of the operands be read in many times, and memory access is a major performance bottleneck in modern computing, Scatter multiplication can potentially offer large performance benefits. The main downside to scatter multiplication is that it requires that individual elements of the product matrix be capable of accumulating partial results piecemeal. This mandates a minimum amount of logic for each element of the product matrix. Consequently, for matrices with n greater than a few hundred elements, even if each element of the product matrix is assumed to require only a single logic block (an unrealistic assumption), the majority of FPGA fabric space will be taken up by accumulators, and only accumulators. For matrices below a few hundred elements however, Scatter multiplication can potentially offer a

large performance increase, which makes it a valuable target for future work on the HARP system.

Finally, there is still work to be done on the standard approach discussed in this paper. While clock speed eventually reached the maximum possible on HARP, more parallelism is still possible. The final implementation of the design only utilizes two processing blocks, for a total of thirty-two multiplies per clock cycle. The processing block scheme is designed to be arbitrarily scalable, as processing blocks do not depend on each other, so increasing the number of processing blocks will increase the number of multiplies per clock cycle without incurring any significant delays. In addition, fusing an implementation of Strassen's method with this setup is an attractive prospect for expanding its capabilities to be able to operate on arbitrarily large matrices with better computational efficiency than either the standard approach or scatter multiplication could do on their own. Increasing the processing blocks and fusing this design with Strassen's method are outside the scope of what can be done by a single researcher in a year, but are very promising for further research into, or commercial implementation of a machine learning hardware accelerator.

CHAPTER VII

CONCLUSION

Over the course of this project, a matrix multiplication engine was implemented on Intel's HARP system. It was optimized and pipelined to a high degree, with the target of being able to multiply large matrices together much faster than what could be done purely on a CPU. To this end, the processing capabilities were divided into progressively smaller units of granularity. First are the processing blocks which are responsible for forming the inner-product of bunches of rows at once. The processing blocks are tileable so that the design can be adapted to fit any amount of available logic or memory. Each processing block contains sixteen processing elements, each one of which can do one inner product of one row and column pair at a time. The processing elements all work in parallel, and the processing blocks above all work in parallel as well, allowing for potentially very many operations to be in flight simultaneously.

The final form of this scheme does outperform what can be done with the on-system processor, but not by a large margin. The performance difference ranges from more than a doubling of speed for smaller matrix sizes (n less than 250), to as little as a 25% improvement (for n of 512). This is due to a variety of factors. Foremost among them is the fact that not all the logic available on the FPGA die was taken by the design. The amount of time required to expand the parallelization without sacrificing clock speed due to increasingly long paths required to connect modules, means that only two processing blocks could be tiled without dropping down the clock speed, which would incur unacceptable performance penalties. However, only approximately 27% of available memory on the FPGA die is taken by the design, and approximately 11% percent of the logic elements are taken up. A large proportion of those figures results from

constant overhead such as the memory multiplexer and column access modules. Every additional processing block only increases the memory and logic usage by about 1%. This suggests that far more parallelism is feasible, if it could not be achieved by a single undergraduate researcher. In addition, incremental improvements in parallelism between memory accesses and operations are possible, which would give further performance improvements.

In conclusion, this system can multiply matrices faster than what can be done in a purely CPU-based implementation. The current implementation does not on its own outperform the processor to enough of a degree to be commercially viable, but its performance combined with the relatively light logic usage of the FPGA suggests that a more skilled implementation could far outperform the CPU. As a result, HARP is a prime candidate both for future research into the subject of machine learning hardware accelerators, and for commercial applications of hardware acceleration.

REFERENCES

1. Leopold, George. “FPGAs, ASICs Seen Driving Machine Learning.” EnterpriseTech, 18 Dec. 2017, www.enterprisetech.com/2017/12/18/fpgas-asics-seen-driving-machine-learning/.

2. Campbell, Scott J., and Sunil P. Khatri. “Resource and Delay Efficient Matrix Multiplication Using Newer FPGA Devices - Semantic Scholar.” Semanticscholar.org, 2006, www.semanticscholar.org/paper/Resource-and-delay-efficient-matrix-multiplication-Campbell-Khatri/89b1848e6810a598444996bc3825cc7ec6a7a53f.

3. Liu, Zhenhong, et al. “SiMul: An Algorithm-Driven Approximate Multiplier Design for Machine Learning.” IEEE Micro, vol. 383, no. 4, July 2018, pp. 50–59.

4. Nguyen, Ngoc Hung, et al. “A High-Performance, Resource-Efficient, Reconfigurable Parallel-Pipelined FFT Processor for FPGA Platforms.” ScienceDirect, 18 Apr. 2018, doi:https://doi-org.ezproxy.library.tamu.edu/10.1016/j.micpro.2018.04.003.

APPENDIX

The Column Access Module Verilog Code

```
`ifndef COL_ACCESS
`define COL_ACCESS

`include "cci_mpf_app_conf.vh"
`include "generated/mem_muxer.sv"
`include "generated/pipelined_mem.sv"

`define NUM_COL_BLOCK_PORTS 16
`define MAX_BLOCK_PORT_INDEX 15


interface col_block_port;
logic request_ready;
logic response_read;

logic [511:0] data;
logic data_ready;
endinterface

module col_port_stub(
    col_block_port col_port
);
assign col_port.request_ready = 1;
assign col_port.response_read = 1;
endmodule

module col_access(
    input logic clk,
    input logic clk_div2,
    input logic reset_in,
    input logic soft_reset_in,
    input logic [31:0] cycles_in,
    mem_muxer_port port,
    col_block_port col_ports[`NUM_COL_BLOCK_PORTS],
    input t_cci_clAddr base_addr,
    input logic [7:0] stride_len,
    input logic [9:0] num_cols_in,
    input logic [7:0] num_blocks_in,
    input logic args_valid_in,
    input logic [3:0] unique_id,
    output logic [31:0] time_of_start,
    output logic [31:0] time_of_latch,
    output logic [31:0] time_of_end,
    output logic error
);

logic [31:0] cycles;
logic [9:0] num_cols;
logic reset;
logic reset2;
logic reset3;
logic soft_reset;
logic args_valid;

always @(posedge clk) begin
    reset <= reset_in;
    reset2 <= reset_in;
    reset3 <= reset_in;
    soft_reset <= soft_reset_in;
    args_valid <= args_valid_in;
    cycles <= cycles_in;
```

```
    num_cols <= num_cols_in;
end

assign port.cycles_wanted = 64;
assign port.tx1.valid = 0;
t_cci_mpf_c0_ReqMemHdr rd_hdr;
t_cci_mpf_ReqMemHdrParams rd_hdr_params;

always_comb
begin
    rd_hdr_params = cci_mpf_defaultReqHdrParams(1);
end


logic read_valid;
logic [511:0] out_data;

logic data_arrived;
logic [511:0] incoming_data;

always @(posedge clk) begin
    data_arrived <= cci_c0Rx_isReadRsp(port.rx) && !port.rx.hdr.mdata[15];
    incoming_data <= port.rx.data;
end

logic [15:0] recieving_addr;
logic [15:0] reading_addr;
logic [15:0] next_reading_addr;
logic [7:0] num_blocks;
logic [15:0] total_blocks;
always @(posedge clk) begin
    next_reading_addr <= reading_addr + 1;
    num_blocks <= num_blocks_in;
end

logic cache_ack_read;
logic cache_read_ready;
logic ready_for_reading;
logic ready_for_reading_int;
logic [511:0] read_data_in;
logic sending;

pipelined_mem column_cache(
    clk,
    clk_div2,
    reset,
    soft_reset,
    ready_for_reading,
    data_arrived,
    incoming_data,
    cache_ack_read,
    unique_id,
    cache_read_ready,
    read_data_in,
    error
);

assign cache_ack_read = sending;

//logic [15:0] total_blocks_recieved;
logic [15:0] total_blocks_sent;
logic [15:0] total_blocks_requested;

always @(posedge clk) begin
    if(reset2) begin
        ready_for_reading_int <= 0;
        ready_for_reading <= 0;
        recieving_addr <= 0;
    end
    else begin
        if(data_arrived) begin
```

```
            recieving_addr <= recieving_addr + 1;
        end
        if(error) $display("ERROR column cache overflowed!");

        ready_for_reading_int <= (recieving_addr == total_blocks) && (total_blocks != 0);
        ready_for_reading <= ready_for_reading_int;

        if(ready_for_reading_int)
            time_of_latch <= cycles;
    end
end

always @(posedge clk) begin
    if(cache_read_ready) begin
        out_data <= read_data_in;
    end
end

logic [17:0] mult_out;
int_mult mult0(
    num_blocks,
    num_cols,
    clk,
    reset,
    mult_out
);

always @(posedge clk) begin
    total_blocks <= mult_out;
end

logic [`NUM_COL_BLOCK_PORTS-1:0] request_ready;
logic [`NUM_COL_BLOCK_PORTS-1:0] response_read;

logic data_ready;
logic [9:0] cols_sent;
logic [9:0] next_cols_sent;

logic [7:0] blocks_sent;
logic [7:0] next_blocks_sent;

logic [15:0] next_total_blocks_sent;

always @(posedge clk) begin
    next_total_blocks_sent <= total_blocks_sent + 1;
end

always @(posedge clk) begin
    %%repeat i 0 `MAX_BLOCK_PORT_INDEX
    request_ready[%%i] <= col_ports[%%i].request_ready;
    response_read[%%i] <= col_ports[%%i].response_read;
    col_ports[%%i].data_ready <= data_ready;
    col_ports[%%i].data <= out_data;
    %%repeat
end

typedef enum logic [2:0] {
    LATCH_IDLE,
    SEND,
    AWAIT_ACK
} col_latch_state_t;

col_latch_state_t latch_state;

logic [15:0] counter;
assign sending = (latch_state == SEND) && request_ready == 16'hffff && cache_read_ready;

//LATCH LOGIC
always @(posedge clk) begin
    if(reset3 || soft_reset) begin
        data_ready <= 0;
```

```
        blocks_sent <= 0;
        cols_sent <= 0;
        if(reset) begin
            latch_state <= LATCH_IDLE;
        end
        else begin
            latch_state <= SEND;
        end
        counter <= 0;
        reading_addr <= 0;
    end
    else begin
        case(latch_state)
        LATCH_IDLE: begin
            data_ready <= 0;
            blocks_sent <= 0;
            cols_sent <= 0;
            reading_addr <= 0;
            total_blocks_sent <= 0;
            if(args_valid) begin
                time_of_start <= cycles;
                latch_state <= SEND;
            end
        end
        SEND: begin
            if(request_ready == 16'hffff && cache_read_ready) begin
                data_ready <= 1;
                latch_state <= AWAIT_ACK;
            end
        end
        AWAIT_ACK: begin
            if(response_read == 16'hffff) begin
                if(next_total_blocks_sent == total_blocks) begin
                    latch_state <= LATCH_IDLE;
                end
                else begin
                    latch_state <= SEND;
                    reading_addr <= next_reading_addr;
                    total_blocks_sent <= next_total_blocks_sent;
                end
                data_ready <= 0;
            end
        end
        endcase
    end
end

%%track_state_machine(col_latch_state_t, latch_state);


logic [7:0] blocks_requested;
logic [11:0] cols_requested;
logic [7:0] next_block_requested;
assign next_block_requested = (blocks_requested + 1 == num_blocks)? 0: (blocks_requested + 1);
logic [11:0] next_col_requested;
assign next_col_requested = cols_requested + 1;
t_cci_clAddr data_addr;

logic [15:0] next_total_blocks_requested;
assign next_total_blocks_requested = total_blocks_requested + 1;

typedef enum logic [1:0] {
    IDLE,
    PREP_REQUEST,
    SEND_REQUEST
} col_request_state_t;

col_request_state_t req_state;

logic sending_now;
assign port.tx0 = cci_mpf_genC0TxReadReq(rd_hdr, (req_state == SEND_REQUEST && sending_now));
```

```
always @(posedge clk) begin
    sending_now <= !port.tx0full && port.tx_ready;
end

logic [15:0] metadata;
assign metadata[15] = 0;
assign metadata[14:0] = total_blocks_requested[14:0];

//REQUEST LOGIC
always @(posedge clk) begin
    if(reset3) begin
        data_addr <= 0;
        req_state <= IDLE;
        blocks_requested <= 0;
        cols_requested <= 0;
        port.want_tx <= 0;
    end
    else begin
        case(req_state)
        IDLE: begin
            if(args_valid) begin
                total_blocks_requested <= 0;
                blocks_requested <= 0;
                cols_requested <= 0;
                data_addr <= base_addr;
                req_state <= PREP_REQUEST;
                port.want_tx <= 1;
            end
        end
        PREP_REQUEST: begin
            rd_hdr <= cci_mpf_c0_genReqHdr(eREQ_RDLINE_I,
                                           data_addr,
                                           t_cci_mdata'(unique_id),
                                           rd_hdr_params);
            req_state <= SEND_REQUEST;
        end
        SEND_REQUEST: begin
            if(sending_now) begin
                blocks_requested <= next_block_requested;
                data_addr <= data_addr + 1;
                total_blocks_requested <= next_total_blocks_requested;
                if(next_total_blocks_requested == total_blocks) begin
                    req_state <= IDLE;
                    port.want_tx <= 0;
                end
                else begin
                    req_state <= PREP_REQUEST;
                end
            end
        end
        endcase
    end
end

%%track_state_machine(col_request_state_t, req_state, 0);

endmodule

`endif
```

The Processing Block Module Verilog Code

```
`ifndef PROCESSING_BLOCK
`define PROCESSING_BLOCK

`include "cci_mpf_if.vh"
`include "csr_mgr.vh"
```

```
`include "afu_json_info.vh"
`include "cci_mpf_app_conf.vh"

`include "generated/mem_muxer.sv"
`include "generated/pb_memory_controller.sv"
`include "generated/processing_element.sv"

module processing_block(
    input logic clk,
    input logic reset,
    input logic [31:0] cycles_in,
    mem_muxer_port writeback_ext_port,
    mem_muxer_port mem_controller_ext_port,
    col_block_port col_port,
    input logic [9:0] width,
    input logic [7:0] stride_len,
    input t_ccip_clAddr base_addr,
    input t_ccip_clAddr res_addr,
    input logic args_valid_in,
    input logic run_in_idle_in,
    input logic [3:0] unique_id,
    input logic spokesman,
    output logic done,
    output logic [31:0] pbmem_start,
    output logic [31:0] pbmem_latch,
    output logic [31:0] pbmem_end,
    output logic [31:0] petotal,
    output logic [31:0] pemult,
    output logic [31:0] pewait
);

writeback_port write_ports[16]();
data_port data_ports[16]();

logic [31:0] cycles;

logic reset_reg0;
logic reset_reg1;
logic reset_reg2;
logic reset_reg3;
logic args_valid0;
logic args_valid1;
logic args_valid2;
logic run_in_idle;
always @(posedge clk) begin
    reset_reg0 <= reset;
    reset_reg1 <= reset;
    reset_reg2 <= reset;
    reset_reg3 <= reset;
    args_valid0 <= args_valid_in;
    args_valid1 <= args_valid_in;
    args_valid2 <= args_valid_in;
    run_in_idle <= run_in_idle_in;

    cycles <= cycles_in;
end

writeback write0(
    clk,
    reset_reg0,
    writeback_ext_port,
    write_ports,
    res_addr,
    stride_len,
    width,
    args_valid0,
    spokesman,
    done,
    unique_id
);
```

```
pb_mem_controller mem0(
    clk,
    reset_reg1,
    cycles,
    mem_controller_ext_port,
    col_port,
    data_ports,
    base_addr,
    stride_len,
    width,
    args_valid1,
    spokesman,
    run_in_idle,
    unique_id+1,
    pbmem_start_in,
    pbmem_latch_in,
    pbmem_end_in
);

logic [7:0] pe_states [15:0];

logic [31:0] petotals [15:0];
logic [31:0] pemults [15:0];
logic [31:0] pewaits [15:0];

processing_element pe[16](
    clk,
    reset_reg2,
    data_ports,
    write_ports,
    width,
    args_valid2,
    spokesman? 16'b0000000000000001: 16'b0,
    8'b0,
    petotals,
    pemults,
    pewaits
);

always @(posedge clk) begin
    petotal <= petotals[0];
    pemult <= pemults[0];
    pewait <= pewaits[0];

    pbmem_start <= pbmem_start_in;
    pbmem_latch <= pbmem_latch_in;
    pbmem_end <= pbmem_end_in;
end

endmodule

`endif
```

The Top Level Module Verilog Code

```
`include "cci_mpf_if.vh"
`include "csr_mgr.vh"
`include "afu_json_info.vh"
`include "cci_mpf_app_conf.vh"

`include "generated/mem_muxer.sv"
`include "generated/pb_memory_controller.sv"
`include "generated/processing_element.sv"
`include "generated/writeback.sv"
`include "generated/processing_block.sv"

`define NUM_PBS         2
`define MAX_PB_INDEX     1
```

```
//`define SLOW_CLK
//`define _SIM

module app_afu
   (
    input  logic clk_in,
    input logic clk_div2_in,
    input logic clk_div4_in,

    // Connection toward the host.  Reset comes in here.
    cci_mpf_if.to_fiu fiu,

    // CSR connections
    app_csrs.app csrs,

    // MPF tracks outstanding requests.  These will be true as long as
    // reads or unacknowledged writes are still in flight.
    input  logic c0NotEmpty,
    input  logic c1NotEmpty
);

logic clk;
logic clk_div2;

`ifdef _SIM
    assign clk = clk_in;

    always @(posedge clk) begin
        if(fiu.reset)
            clk_div2 <= 0;
        else begin
            clk_div2 <= ~clk_div2;
        end
    end
`else
    `ifndef SLOW_CLK
        assign clk = clk_in;
        assign clk_div2 = clk_div2_in;
    `else
        assign clk = clk_div2_in;
        assign clk_div2 = clk_div4_in;
    `endif
`endif


logic [7:0] unique_id;
assign unique_id = %%assign_unique_id;

logic reset;
assign reset = fiu.reset;

localparam CL_BYTE_IDX_BITS = 6;
typedef logic [$bits(t_cci_clAddr) + CL_BYTE_IDX_BITS - 1 : 0] t_byteAddr;

function automatic t_cci_clAddr byteAddrToClAddr(t_byteAddr addr);
    return addr[CL_BYTE_IDX_BITS +: $bits(t_cci_clAddr)];
endfunction

function automatic t_byteAddr clAddrToByteAddr(t_cci_clAddr addr);
    return {addr, CL_BYTE_IDX_BITS'(0)};
endfunction

t_ccip_clAddr result_addr;

logic start_operation;
t_ccip_clAddr first_mat_addr;
t_ccip_clAddr second_mat_addr;

logic [9:0] mat_height;
logic [9:0] mat_width;
```

```
logic error;
logic error_out;

always @(posedge clk) begin
    if(reset) begin
        error_out <= 0;
    end
    else begin
        if(error) begin
            error_out <= 1;
        end
    end
end

logic [31:0] cycles;

always @(posedge clk) begin
    if(reset) begin
        cycles <= 0;
    end
    else begin
        cycles <= cycles + 1;
    end
end

logic [31:0] petotals[`NUM_PBS-1:0];
logic [31:0] pemults[`NUM_PBS-1:0];
logic [31:0] pewaits[`NUM_PBS-1:0];
logic [31:0] pbmem_starts[`NUM_PBS-1:0];
logic [31:0] pbmem_latchs[`NUM_PBS-1:0];
logic [31:0] pbmem_ends[`NUM_PBS-1:0];
logic [31:0] colstart;
logic [31:0] collatch;
logic [31:0] colend;

logic done;

always_comb
begin
    // The AFU ID is a unique ID for a given program.  Here we generated
    // one with the "uuidgen" program and stored it in the AFU's JSON file.
    // ASE and synthesis setup scripts automatically invoke afu_json_mgr
    // to extract the UUID into afu_json_info.vh.
    csrs.afu_id = `AFU_ACCEL_UUID;

    // Default
    for (int i = 7; i < NUM_APP_CSRS; i = i + 1)
    begin
        csrs.cpu_rd_csrs[i].data = 64'(0);
    end

    csrs.cpu_rd_csrs[0].data = 64'(done);

    csrs.cpu_rd_csrs[1].data = {petotals[0], pemults[0]};
    csrs.cpu_rd_csrs[2].data = pewaits[0];
    csrs.cpu_rd_csrs[3].data = {pbmem_starts[0], pbmem_latchs[0]};
    csrs.cpu_rd_csrs[4].data = pbmem_ends[0];
    csrs.cpu_rd_csrs[5].data = {colstart, collatch};
    csrs.cpu_rd_csrs[6].data = {colend, 32'(error)};
end

always_ff @(posedge clk)
begin
    if(reset) begin
        result_addr <= 0;
        first_mat_addr <= 0;
        second_mat_addr <= 0;
        start_operation <= 0;
        mat_height <= 0;
        mat_width <= 0;
```

```
    end
    else begin
        if (csrs.cpu_wr_csrs[0].en)
        begin
            result_addr <= byteAddrToClAddr(csrs.cpu_wr_csrs[0].data);
            $display("result addr: %d", byteAddrToClAddr(csrs.cpu_wr_csrs[0].data));
        end

        if (csrs.cpu_wr_csrs[1].en)
        begin
            first_mat_addr <= byteAddrToClAddr(csrs.cpu_wr_csrs[1].data);
            $display("first addr: %d", byteAddrToClAddr(csrs.cpu_wr_csrs[1].data));
        end

        if (csrs.cpu_wr_csrs[2].en)
        begin
            second_mat_addr <= byteAddrToClAddr(csrs.cpu_wr_csrs[2].data);
            $display("second addr: %d", byteAddrToClAddr(csrs.cpu_wr_csrs[2].data));
        end

        if (csrs.cpu_wr_csrs[3].en)
        begin
            mat_height <= csrs.cpu_wr_csrs[3].data;
            $display("height: %d", csrs.cpu_wr_csrs[3].data);
        end

        if (csrs.cpu_wr_csrs[4].en)
        begin
            mat_width <= csrs.cpu_wr_csrs[4].data;
            $display("width: %d", csrs.cpu_wr_csrs[4].data);
        end

        start_operation <= csrs.cpu_wr_csrs[5].en;
    end
end



logic yup_we_good;
assign yup_we_good = 1;/*(mat_height != 0) &&
                      (mat_width != 0) &&
                      (result_addr != 0) &&
                      (first_mat_addr != 0) &&
                      (second_mat_addr != 0);*/

logic [3:0] pb_index;
logic [9:0] rows_covered;
logic [3:0] cycles_waited;
logic [3:0] next_cycles_waited;
logic [7:0] address_offset;
assign next_cycles_waited = cycles_waited + 1;

t_ccip_clAddr curr_input_addr;
t_ccip_clAddr curr_res_addr;
logic pb_start;

//these will be in an array eventually
t_ccip_clAddr input_addr[`NUM_PBS-1:0];
t_ccip_clAddr input_res_addr[`NUM_PBS-1:0];
logic [`NUM_PBS-1:0] pb_enable;
logic [`NUM_PBS-1:0] pb_done;

logic all_rows_covered;
logic addressed_all_pbs;
//assign all_rows_covered = rows_covered[9:3] == mat_width[9:3];

always @(posedge clk) begin
    all_rows_covered <= rows_covered[9:3] == mat_width[9:3];
    addressed_all_pbs <= pb_index == `NUM_PBS;
end
```

```
typedef enum logic [2:0] {
    IDLE,
    DISPATCH,
    DISPATCH_SET,
    START,
    WAIT,
    RESET_PB,
    RESET_COL
} controller_state_t;

controller_state_t state;

always @(posedge clk) begin
    if(reset) begin
        state <= IDLE;
        rows_covered <= 0;
        cycles_waited <= 0;
        //curr_input_addr <= 0;
        //curr_res_addr <= 0;
        pb_index <= 0;
        done <= 0;
        address_offset <= 0;
        %%repeat i 0 `MAX_PB_INDEX
        input_addr[%%i] <= 0;
        input_res_addr[%%i] <= 0;
        pb_enable[%%i] <= 0;
        %%repeat
    end
    else begin
        case(state)
        IDLE: begin
            if(start_operation && yup_we_good) begin
                state <= DISPATCH;
                //curr_input_addr <= first_mat_addr;
                //curr_res_addr <= result_addr;
                address_offset <= 0;
                pb_index <= 0;
                rows_covered <= 0;
            end
            if(start_operation)
                done <= 0;
        end
        DISPATCH: begin
            if(addressed_all_pbs) begin
                state <= START;
            end
            else begin
                if(!all_rows_covered) begin
                    input_addr[pb_index] <= first_mat_addr + address_offset;
                    input_res_addr[pb_index] <= result_addr + address_offset;
                    pb_enable[pb_index] <= 1;
                    rows_covered <= rows_covered + 16;
                    address_offset <= address_offset + 1;
                end
                else begin
                    pb_enable[pb_index] <= 0;
                end
                state <= DISPATCH_SET;
            end
            pb_index <= pb_index + 1;
        end
        DISPATCH_SET: begin
            state <= DISPATCH;
        end
        START: begin
            state <= WAIT;
        end
        WAIT: begin
            if(pb_done|~pb_enable == {`NUM_PBS{1'b1}}) begin
                state <= RESET_PB;
```

```
                cycles_waited <= 0;
            end
            //if(pb_done != 0) begin
            //    $display("ERROR: PB IS REPORTING AS DONE WTF");
            //end
        end
        RESET_PB: begin
            cycles_waited <= cycles_waited + 1;
            if(cycles_waited == 8) begin
                if(all_rows_covered) begin
                    done <= 1;
                    state <= RESET_COL;
                    cycles_waited <= 0;
                end
                else begin
                    pb_index <= 0;
                    state <= DISPATCH;
                end
            end
        end
        RESET_COL: begin
            cycles_waited <= next_cycles_waited;
            if(cycles_waited == 8) begin
                state <= IDLE;
            end
        end
        endcase
    end
end

%%track_state_machine(controller_state_t, state);

mem_muxer_port mem_muxer_ports[`NUM_MEM_MUXER_PORTS]();
col_block_port col_ports[16]();

logic [3:0] unique_ids[`NUM_PBS-1:0];
assign unique_ids = %%assign_unique_ids(`NUM_PBS, 2);

mem_muxer mux0(
    clk,
    reset,
    cycles,
    fiu,
    mem_muxer_ports,
    0
);

col_access col0(
    clk,
    clk_div2,
    reset || (state == RESET_COL),
    (state == RESET_PB),    //soft reset
    cycles,
    mem_muxer_ports[2*`NUM_PBS],
    col_ports,
    second_mat_addr,
    mat_height >> 4,
    mat_width,
    mat_height >> 4,
    start_operation && yup_we_good,
    0,
    colstart,
    collatch,
    colend,
    error
);

genvar i;
for(i=0; i<`NUM_PBS; i+=1) begin : make_pbs
    processing_block pb0(
        clk,
```

```
        reset || (state == RESET_PB),
        cycles,
        mem_muxer_ports[2*i],
        mem_muxer_ports[2*i + 1],
        col_ports[i],
        mat_width,
        mat_height >> 4,
        input_addr[i],
        input_res_addr[i],
        pb_enable[i] && (state == START),
        !pb_enable[i] && ((state == WAIT) || (state == START)),
        unique_ids[i],
        (i == 0), //spokesman
        pb_done[i],
        pbmem_starts[i],
        pbmem_latchs[i],
        pbmem_ends[i],
        petotals[i],
        pemults[i],
        pewaits[i]
    );
end

assign top_level_states = 32'(0);

for(i=`NUM_PBS; i<16; i+=1) begin
    assign col_ports[i].request_ready = 1;
    assign col_ports[i].response_read = 1;
end

for(i=2*`NUM_PBS+1; i<`NUM_MEM_MUXER_PORTS; i+=1) begin
    assign mem_muxer_ports[i].want_tx = 0;
end

endmodule
```

The Memory Multiplexer Module Code

```
`ifndef MEM_MUXER
`define MEM_MUXER

`include "cci_mpf_if.vh"
`include "csr_mgr.vh"
`include "afu_json_info.vh"
`include "cci_mpf_app_conf.vh"

`define NUM_MEM_MUXER_PORTS    5
`define MUXER_PORTS_MAX_INDEX  4
`define MEM_MUXER_PORT_BITS    4

interface mem_muxer_port;
    t_if_cci_c0_Rx rx;
    logic tx_ready;
    logic tx0full;
    logic tx1full;

    logic want_tx;
    logic [7:0] cycles_wanted;
    t_if_cci_mpf_c0_Tx tx0;
    t_if_cci_mpf_c1_Tx tx1;
endinterface

module mem_muxer_port_stub(
    mem_muxer_port port
);
assign want_tx = 0;
endmodule
```

```
module mem_muxer(
    input logic clk,
    input logic reset,
    input logic [31:0] cycles_in,
    cci_mpf_if.to_fiu fiu,

    mem_muxer_port ports[`NUM_MEM_MUXER_PORTS],
    input logic [7:0] unique_id
);

logic [31:0] cycles;

always @(posedge clk) begin
    cycles <= cycles_in;
end

logic [`MEM_MUXER_PORT_BITS - 1:0] port;
logic [`MEM_MUXER_PORT_BITS - 1:0] next_port;
//assign next_port = (port + 1)%`NUM_MEM_MUXER_PORTS;
assign next_port = (port == `MUXER_PORTS_MAX_INDEX)? 0: (port+1);
logic [7:0] cycles_left;

assign fiu.c2Tx.mmioRdValid = 1'b0;

//t_if_cci_mpf_c0_Tx selected_tx0;
//t_if_cci_mpf_c1_Tx selected_tx1;

always @(posedge clk) begin
    case(port)
    %%repeat i 0 `MUXER_PORTS_MAX_INDEX
    %%i: begin
        fiu.c0Tx <= ports[%%i].tx0;
        fiu.c1Tx <= ports[%%i].tx1;
    end
    %%repeat
    endcase

    //if(fiu.c0TxAlmFull) $display("TX ALM FULL");
    //if(fiu.c0Tx.valid) $display("VALID");
end

genvar i;
for(i=0; i<`NUM_MEM_MUXER_PORTS; i+=1) begin
    assign ports[i].tx0full = fiu.c0TxAlmFull;
    assign ports[i].tx1full = fiu.c1TxAlmFull;
    //assign ports[i].rx = fiu.c0Rx;
end

always @(posedge clk) begin
    %%repeat i 0 `MUXER_PORTS_MAX_INDEX
    ports[%%i].rx <= fiu.c0Rx;
    %%repeat
end

typedef enum logic [2:0] {
    RELAYING,
    STOPPING,
    STOPPING2,
    CHANGING
} mem_muxer_states;

mem_muxer_states state;

logic [7:0] counter;

always @(posedge clk) begin
    if(reset) begin
        counter <= 0;

        state <= CHANGING;
        port <= 0;
```

```
        %%repeat i 0 `MUXER_PORTS_MAX_INDEX
        ports[%%i].tx_ready <= 0;
        %%repeat
        //fiu.c0Tx.valid <= 0;
        //fiu.c1Tx.valid <= 0;
    end
    else begin
        counter <= counter + 1;
        case(state)
        RELAYING: begin
            case(port)
            %%repeat i 0 `MUXER_PORTS_MAX_INDEX
            %%i: begin
                ports[%%i].tx_ready <= 1;
            end
            %%repeat
            endcase
            cycles_left <= cycles_left - 1;
            if(cycles_left == 0) begin
                state <= STOPPING;
            end
        end
        STOPPING: begin
            case(port)
            %%repeat i 0 `MUXER_PORTS_MAX_INDEX
            %%i: begin
                ports[%%i].tx_ready <= 0;
            end
            %%repeat
            endcase
            state <= STOPPING2;
        end
        STOPPING2: begin
            state <= CHANGING;
        end
        CHANGING: begin
            //fiu.c0Tx.valid <= 0;
            //fiu.c1Tx.valid <= 0;
            case(next_port)
            %%repeat i 0 `MUXER_PORTS_MAX_INDEX
            %%i: begin
                if(ports[%%i].want_tx) begin
                    cycles_left <= ports[%%i].cycles_wanted;
                    ports[%%i].tx_ready <= 1;
                    state <= RELAYING;
                end
            end
            %%repeat
            endcase
            port <= next_port;
        end
        endcase
    end
end


%%track_state_machine(mem_muxer_states, state, 0);


endmodule

`endif
```

The Processing Block Memory Controller Code

```
`ifndef PB_MEMORY_CACHE
`define PB_MEMORY_CACHE
```

```
`include "cci_mpf_if.vh"
`include "csr_mgr.vh"
`include "afu_json_info.vh"
`include "cci_mpf_app_conf.vh"

module pb_memory_cache(
    input logic clk,
    input logic reset,
    input logic [511:0] write_data_in,
    input logic write_ready_in,
    input logic [8:0] write_addr_in,
    input logic ready_for_latch,
    output logic [511:0] read_data_out,
    output logic read_valid_out,
    output logic [9:0] addr,
    input logic [7:0] unique_id
);

logic [9:0] next_addr;
logic [9:0] written;
logic last_addr;
logic read_valid_in;
logic read_valid;
logic [511:0] write_data;
//logic [511:0] read_data;

%%repeat i 0 7
logic [9:0] addr%%i;
logic write_ready%%i;
%%repeat

assign next_addr = addr0 + 1;

always @(posedge clk) begin
    write_data <= write_data_in;
    //read_data_out <= read_data;
    read_valid_out <= read_valid;
    read_valid <= read_valid_in;
    %%repeat i 0 7
    write_ready%%i <= write_ready_in;
    %%repeat
end

%%repeat i 0 7
logic [63:0] read_data%%i;
logic [63:0] read_data_out%%i;

always @(posedge clk) begin
    read_data_out%%i <= read_data%%i;
end
%%repeat

assign read_data_out = {
    read_data_out7,
    read_data_out6,
    read_data_out5,
    read_data_out4,
    read_data_out3,
    read_data_out2,
    read_data_out1,
    read_data_out0
};

%%repeat i 0 7
pb_block_ramtry2 pb_ram%%i(
    write_data[64*(%%i+1)-1:64*%%i],
    addr%%i[8:0],
    write_ready%%i,
    clk,
    read_data%%i
```

```
);
%%repeat

typedef enum logic [3:0] {
    WRITING,
    READING,
    WAITING
} pb_mem_cache_state_t;

pb_mem_cache_state_t pb_mem_state;

always @(posedge clk) begin
    if(reset) begin
        written <= 0;
        pb_mem_state <= WRITING;
        read_valid_in <= 0;
        %%repeat i 0 7
        addr%%i <= 0;
        %%repeat
    end
    else begin
        case(pb_mem_state)
        WRITING: begin
            addr <= written;
            if(write_ready_in) begin
                written <= written + 1;
                %%repeat i 0 7
                addr%%i <= write_addr_in;
                %%repeat
                $display("PB memory cache latching in %h", write_data);
            end
            else if(ready_for_latch) begin
                if(written[3:0] != 0)
                    $display("ERROR: pb mem cache did not read a multiple of 16 rows in! (%d)",
written);
                pb_mem_state <= READING;
                %%repeat i 0 7
                addr%%i <= 0;
                %%repeat
                read_valid_in <= 1;
                //$display("ROW CACHE HAS %d ROWS WRITTEN", written);
            end
        end
        READING: begin
            %%repeat i 0 7
            addr%%i <= next_addr;
            %%repeat
            addr <= addr0;
            //$display("Data: %h", read_data_out);
            if(next_addr != written) begin
                read_valid_in <= 1;
            end
            else begin
                read_valid_in <= 0;
                pb_mem_state <= WAITING;
                //$display("ROW CACHE DONE");
            end
        end
        WAITING: begin
            %%repeat i 0 7
            addr%%i <= 0;
            %%repeat
            if(ready_for_latch) begin
                pb_mem_state <= READING;
                read_valid_in <= 1;
            end
        end
        endcase
    end
end
```

```
%%track_state_machine(pb_mem_cache_state_t, pb_mem_state, 1);

endmodule

`endif
```

The Pipelined Memory Code

```
`ifndef PIPELINED_MEM
`define PIPELINED_MEM

`include "cci_mpf_if.vh"
`include "csr_mgr.vh"
`include "afu_json_info.vh"
`include "cci_mpf_app_conf.vh"

module pipelined_mem(
    input logic clk,
    input logic clk_div2,
    input logic reset,
    input logic soft_reset_in,
    input logic ready_for_reading_in,
    input logic write_ready_in,
    input logic [511:0] write_data_in,
    input logic ack_read,
    input logic [7:0] unique_id,
    output logic read_ready,
    output logic [511:0] read_data_out,
    output logic error_overflow
);

logic reset_div2;
logic reset_div2_0;
logic reset_div2_1;
logic reset_div2_2;
logic reset_div2_3;
logic soft_reset;
always @(posedge clk_div2) reset_div2 <= reset;
always @(posedge clk_div2) reset_div2_0 <= reset;
always @(posedge clk_div2) reset_div2_1 <= reset;
always @(posedge clk_div2) reset_div2_2 <= reset;
always @(posedge clk_div2) reset_div2_3 <= reset;
always @(posedge clk_div2) soft_reset <= soft_reset_in;

logic no_write_data;    //on clk_div2
logic wrfull;
assign error_overflow = wrfull && write_ready_in;
logic write_ready;
assign write_ready = !no_write_data;
logic [511:0] write_data_int;
logic [511:0] read_data;

logic ready_for_reading_buf;
logic ready_for_reading_buf2;
logic ready_for_reading_buf3;
logic ready_for_reading;

always @(posedge clk) begin
    ready_for_reading_buf <= ready_for_reading_in;
end

always @(posedge clk_div2) begin
    ready_for_reading_buf2 <= ready_for_reading_buf;
    ready_for_reading_buf3 <= ready_for_reading_buf2;
    ready_for_reading <= ready_for_reading_buf3;
end
```

```
bigfifo writefifo(
    write_data_in,
    write_ready_in,
    write_ready,
    clk,
    clk_div2,
    reset,
    write_data_int,
    no_write_data,
    wrfull
);

logic write_out_ready;
logic rdempty;
logic out_fifo_empty;

logic rdempty_dummy0;
logic rdempty_dummy1;
logic rdempty_dummy2;
logic out_fifo_empty_dummy0;
logic out_fifo_empty_dummy1;
logic out_fifo_empty_dummy2;

logic rdempty_int;
logic [127:0] read_data_out_int0;
logic [127:0] read_data_out_int1;
logic [127:0] read_data_out_int2;
logic [127:0] read_data_out_int3;
logic [127:0] read_data_out0;
logic [127:0] read_data_out1;
logic [127:0] read_data_out2;
logic [127:0] read_data_out3;


readfifotry3 outfifo0(
    read_data[127:0],
    write_out_ready,
    ack_read,
    clk_div2,
    clk,
    reset_div2_0,
    read_data_out_int0,
    rdempty_dummy0,
    out_fifo_empty_dummy0
);

readfifotry3 outfifo1(
    read_data[255:128],
    write_out_ready,
    ack_read,
    clk_div2,
    clk,
    reset_div2_1,
    read_data_out_int1,
    rdempty_dummy1,
    out_fifo_empty_dummy1
);

readfifotry3 outfifo2(
    read_data[383:256],
    write_out_ready,
    ack_read,
    clk_div2,
    clk,
    reset_div2_2,
    read_data_out_int2,
    rdempty_dummy2,
    out_fifo_empty_dummy2
);
```

```
readfifotry3 outfifo3(
    read_data[511:384],
    write_out_ready,
    ack_read,
    clk_div2,
    clk,
    reset_div2_3,
    read_data_out_int3,
    rdempty_int,
    out_fifo_empty
);

always @(posedge clk) begin
    rdempty <= rdempty_int;
    read_data_out0 <= read_data_out_int0;
    read_data_out1 <= read_data_out_int1;
    read_data_out2 <= read_data_out_int2;
    read_data_out3 <= read_data_out_int3;
end

assign read_data_out = {
    read_data_out3,
    read_data_out2,
    read_data_out1,
    read_data_out0
};

assign read_ready = !rdempty;

typedef enum logic [2:0] {
    WRITING,
    READING,
    READ
} pipelined_mem_states_t;

pipelined_mem_states_t memstate;
logic [13:0] address;
logic [13:0] naive_next_address;
logic [13:0] next_address;
logic [13:0] written;
assign naive_next_address = address + 1;
assign next_address = (naive_next_address == written)? 0: naive_next_address;

always @(posedge clk_div2) begin
    if(reset_div2) begin
        memstate <= WRITING;
        address <= 0;
        written <= 0;
        //$display("RESET DIV2");
    end
    else if(soft_reset) begin
        address <= 0;
        memstate <= READING;
    end
    else begin
        case(memstate)
        WRITING: begin
            if(write_ready) begin
                address <= naive_next_address;
            end
            else if(ready_for_reading) begin
                memstate <= READING;
                written <= address;
                if(address[3:0] != 0) begin
                    $display("ERROR: col cache did not read a multiple of 16 cl's in!");
                end
                address <= 0;
            end
        end
        READING: begin
            //$display("READING");
```

```
            if(write_out_ready) begin
                memstate <= READ;
            end
        end
        READ: begin
            //$display("READ");
            if(out_fifo_empty) begin
                address <= next_address;
                //$display("READING FROM ADDR %d COMPLETE", address);
                memstate <= READING;
            end
        end
        endcase
    end
end

%%track_state_machine(pipelined_mem_states_t, memstate)

logic [15:0] wr_enable0;
logic [15:0] wr_enable1;
logic [15:0] wr_enable2;
logic [15:0] wr_enable3;

logic [512:0] read_out [15:0];
//assign read_data = read_out[address[13:10]];

logic [9:0] addr [15:0];

logic [511:0] write_data;
always @(posedge clk_div2) begin
    write_data <= write_data_int;
    %%repeat i 0 15
    addr[%%i] <= address[9:0];
    %%repeat
    wr_enable0 <= write_ready << address[13:10];
    wr_enable1 <= write_ready << address[13:10];
    wr_enable2 <= write_ready << address[13:10];
    wr_enable3 <= write_ready << address[13:10];

    read_data <= read_out[address[13:10]];
end

genvar i;
//addr, data, wren, and data output are all registered
for(i=0; i<16; i+=1) begin : gen_blocks0
    blockramtry5div16 blocks0(
        write_data[127:0],
        addr[i][9:0],
        wr_enable0[i],
        clk_div2,
        read_out[i][127:0]
    );
end

for(i=0; i<16; i+=1) begin : gen_blocks1
    blockramtry5div16 blocks1(
        write_data[255:128],
        addr[i][9:0],
        wr_enable1[i],
        clk_div2,
        read_out[i][255:128]
    );
end

for(i=0; i<16; i+=1) begin : gen_blocks2
    blockramtry5div16 blocks2(
        write_data[383:256],
        addr[i][9:0],
        wr_enable2[i],
        clk_div2,
        read_out[i][383:256]
```

```
    );
end

for(i=0; i<16; i+=1) begin : gen_blocks3
    blockramtry5div16 blocks3(
        write_data[511:384],
        addr[i][9:0],
        wr_enable3[i],
        clk_div2,
        read_out[i][511:384]
    );
end

logic [3:0] input_change_counter;
logic [13:0] old_addr;

`define READ_LATENCY 3

always @(posedge clk) begin
    write_out_ready <= (memstate == READING) && (input_change_counter == `READ_LATENCY);

    //if(write_out_ready) begin
    //    $display("WRITE OUT READY");
    //end
end

always @(posedge clk_div2) begin
    if(reset_div2) begin
        old_addr <= 0;
        input_change_counter <= 0;
    end
    else if(soft_reset) begin
        input_change_counter <= 0;
    end
    else begin
        if(memstate == READING && address != old_addr) begin
            input_change_counter <= 0;
        end
        else begin
            if(input_change_counter <= `READ_LATENCY) begin
                input_change_counter <= input_change_counter + 1;
            end
        end
        old_addr <= address;
    end
end

endmodule

`endif
```

The Processing Element Code

```
`ifndef PROCESSING_ELEMENT
`define PROCESSING_ELEMENT

`include "cci_mpf_if.vh"
`include "csr_mgr.vh"
`include "afu_json_info.vh"
`include "cci_mpf_app_conf.vh"

`include "generated/pb_memory_controller.sv"
`include "generated/writeback.sv"

module processing_element(
    input logic clk,
    input logic reset,
```

```
    data_port port,
    writeback_port writebackport,
    input logic [9:0] num_cols_in,
    input logic args_valid,
    input logic spokesman,
    input logic [7:0] unique_id,
    output logic [31:0] total_cycles,
    output logic [31:0] cycles_latching,
    output logic [31:0] cycles_waiting
);

logic am_ready;
assign port.ready_for_next = am_ready;

logic latching;
logic am_ready_delayed;
assign latching = am_ready_delayed && port.next_ready;

logic [9:0] num_cols;
always @(posedge clk) begin
    num_cols <= num_cols_in;
    am_ready_delayed <= am_ready;
end

logic mult_enabled;
logic reset_mult;

logic [31:0] mult_output;
logic [31:0] result;
assign writebackport.data = result;

logic mult_latch;

logic [31:0] a;
logic [31:0] b;
logic [31:0] a_reg;
logic [31:0] b_reg;

//assign a = mult_latch? a_reg: 0;
//assign b = mult_latch? b_reg: 0;

logic mult_enabled_reg;
logic reset_mult_reg;

always @(posedge clk) begin
    //a_reg <= port.a;
    //b_reg <= port.b;
    mult_enabled_reg <= mult_enabled;
    reset_mult_reg <= reset_mult;
    mult_latch <= latching;

    a <= mult_latch? port.a: 0;
    b <= mult_latch? port.b: 0;
end

multiply_acc mult0(
    a,
    mult_enabled_reg,
    reset_mult_reg,
    b,
    clk,
    mult_output
);

typedef enum logic [2:0] {
    IDLE,
    PREP,
    DUMMYWAIT,
    MULTIPLYING,
    WAIT,
    WRITEBACK,
```

```
    RESET
} t_state;

t_state state;

logic [9:0] indices_added;
logic [9:0] next_indices_added;
assign next_indices_added = indices_added + 1;
logic [9:0] next_next_indices_added;
assign next_next_indices_added = indices_added + 2;
logic [3:0] cycles_waited;
logic [3:0] next_cycles_waited;
assign next_cycles_waited = cycles_waited + 1;

assign reset_mult = (reset ||  (state == RESET) || (state == IDLE));

always @(posedge clk) begin
    am_ready <= ((state == MULTIPLYING) && (next_next_indices_added < num_cols));
end
//assign am_ready = ((state == MULTIPLYING) && (next_indices_added < num_cols));


always @(posedge clk) begin
    if(reset) begin
        state <= IDLE;
        indices_added <= 0;
        writebackport.result_valid <= 0;
        result <= 0;
        mult_enabled <= 0;
    end
    else begin
    case(state)
        IDLE: begin
            if(args_valid) begin
                state <= PREP;
                indices_added <= 0;
                cycles_waited <= 0;
                mult_enabled <= 1;
            end
            else begin
                mult_enabled <= 0;
            end
            writebackport.result_valid <= 0;
        end
        PREP: begin
            cycles_waited <= next_cycles_waited;
            if(cycles_waited == 4) begin
                state <= MULTIPLYING;
                cycles_waited <= 0;
            end
        end
        MULTIPLYING: begin
            if(mult_latch)
                if(spokesman) $display("%d   %f * %f = RES: %f %h",
                                       indices_added,
                                       $bitstoshortreal(a),
                                       $bitstoshortreal(b),
                                       $bitstoshortreal(mult_output),
                                       mult_output
                                      );
            if(latching) begin
                indices_added <= next_indices_added;
                if(next_indices_added == num_cols) begin
                    state <= WAIT;
                    cycles_waited <= 0;
                end
            end
        end
        WAIT: begin
            cycles_waited <= next_cycles_waited;
            if(cycles_waited == 5) begin
```

```
                writebackport.result_valid <= 1;
                if(spokesman) $display("Set for writeback! %h", mult_output);
                state <= WRITEBACK;
                result <= mult_output;
            end
        end
        WRITEBACK: begin
            if(writebackport.result_read) begin
                writebackport.result_valid <= 0;
                state <= RESET;
                mult_enabled <= 0;
            end
        end
        RESET: begin
            state <= MULTIPLYING;
            indices_added <= 0;
            mult_enabled <= 1;
        end
    endcase
    end
end


%%track_state_machine(t_state, state, spokesman);


endmodule

`endif
```

The Writeback Module Code

```
`ifndef WRITEBACK
`define WRITEBACK

`include "cci_mpf_if.vh"
`include "csr_mgr.vh"
`include "afu_json_info.vh"
`include "cci_mpf_app_conf.vh"

`include "generated/mem_muxer.sv"

interface writeback_port;
logic result_valid;
logic [31:0] data;
logic result_read;
endinterface

module writeback(
    input logic clk,
    input logic reset,
    mem_muxer_port port,
    writeback_port pe_ports[16],
    input t_cci_clAddr base_addr,
    input logic [7:0] stride_len,
    input logic [9:0] row_width,
    input logic args_valid,
    input logic spokesman,
    output logic done,
    input logic [3:0] unique_id
);

typedef enum logic [1:0] {
    IDLE,
    WAIT,
    PREP_REQUEST,
```

```
    SEND_REQUEST
} writeback_state_t;

t_cci_mpf_c1_ReqMemHdr wr_hdr;

writeback_state_t state;
t_cci_clAddr curr_addr;
logic [15:0] results_ready;
logic result_read;
logic [511:0] data;
logic [9:0] cols_written;
logic [9:0] next_col;
logic all_results_ready;

assign port.cycles_wanted = 8;
assign port.tx0.valid = 0;
assign next_col = cols_written + 1;
assign port.tx1.valid = ((state == SEND_REQUEST) && port.tx_ready && !port.tx1full);
assign port.tx1.hdr = wr_hdr;

genvar i;
for(i=0; i<16; i+=1) begin
    //assign results_ready[i] = pe_ports[i].result_valid;
    assign pe_ports[i].result_read = result_read;
end

always @(posedge clk) begin
    %%repeat i 0 15
    results_ready[%%i] <= pe_ports[%%i].result_valid;
    data[32*(%%i+1)-1:32*%%i] <= pe_ports[%%i].data;
    %%repeat

    all_results_ready <= results_ready == 16'hffff;
end

always @(posedge clk) begin
    if(reset) begin
        state <= IDLE;
        curr_addr <= 0;
        port.want_tx <= 0;
        result_read <= 0;
        cols_written <= 0;
        done <= 0;
    end
    else begin
        case(state)
        IDLE: begin
            result_read <= 0;
            port.want_tx <= 0;
            if(args_valid) begin
                state <= WAIT;
                curr_addr <= base_addr;
                cols_written <= 0;
                done <= 0;
            end
        end
        WAIT: begin
            result_read <= 0;
            if(all_results_ready) begin
                state <= PREP_REQUEST;
                port.want_tx <= 1;
            end
        end
        PREP_REQUEST: begin
            $display("Writeback module #%d is writing back column %d",
                     unique_id, cols_written);
            $display("\tData: %h", data);
            wr_hdr <= cci_mpf_c1_genReqHdr(eREQ_WRLINE_I,
                                           t_cci_clAddr'(curr_addr),
                                           t_cci_mdata'({12'hfff, unique_id}),
                                           cci_mpf_defaultReqHdrParams(1));
```

```
            port.tx1.data <= t_ccip_clData'(data);
            result_read <= 1;
            state <= SEND_REQUEST;
        end
        SEND_REQUEST: begin
            if(port.tx_ready && !port.tx1full) begin
                cols_written <= next_col;
                curr_addr <= curr_addr + stride_len;
                port.want_tx <= 0;
                if(next_col == row_width) begin
                    state <= IDLE;
                    done <= 1;
                end
                else begin
                    state <= WAIT;
                end
            end
        end
        endcase
    end
end


%%track_state_machine(writeback_state_t, state, spokesman);

assign states = 0;

endmodule

`endif
```

The Synthesis Script Built

```
#!/bin/bash

./generate
if [[ $? -ne "0" ]]; then
    echo generation failed!
    exit 1
fi

if [ -z "$1" ]; then
    BUILD_DIR=build_fpga
else
    BUILD_DIR="$1"
fi

if [ ! -d "../$BUILD_DIR" ]; then
    cd ..
    afu_synth_setup -s rtl/sources.txt $BUILD_DIR
    if [[ $? -ne "0" ]]; then
        echo Creation of $BUILD_DIR directory failed!
        exit
    fi
    cd rtl
fi

set -o pipefail

cd ../$BUILD_DIR

qsub-synth

sleep 2

tail -f build.log | python ../colorize.py
```

```
cd ../rtl
```

The Script Used For Colorizing Simulation and Build Logs

```
import re
import sys

if len(sys.argv) == 2:
    outfile = open(sys.argv[1], 'w')
else:
    outfile = None

useless_file = open("../useless_messages.txt", 'r')

useless_lines = useless_file.readlines()
useless_lines = [line.strip() for line in useless_lines]

def get_code(codes):
    if type(codes) == int:
        codes = [codes]
    code_str = ";".join(map(str, codes))
    return "\033[{}m".format(code_str)

def insert(s, to_insert, index):
    return s[:index] + to_insert + s[index:]

def convert(text, mapping):
    for key, codes in mapping.items():
        for match in list(re.finditer(key, text, re.IGNORECASE))[::-1]:
            code = get_code(codes)
            reset = get_code(RETURN)
            text = insert(text, reset, match.start()+len(key))
            text = insert(text, code, match.start())
    lines = [line+"\n" for line in text.split("\n")]
    for useless_line in useless_lines:
        lines = [line for line in lines if useless_line not in line]
    text = "".join(lines)
    return text


BOLD = 1
FAINT = 2
ITALIC = 3
RED_BG = 41
RED_FG = 31
WHITE_BG = 47
WHITE_FG = 37
YELLOW_BG = 43
YELLOW_FG = 33
BLACK_BG = 40
BLACK_FG = 30
RETURN = 0

mapping = {
    "error": [RED_BG, BOLD],
    "warning": [YELLOW_BG, BLACK_FG]
}

try:
    while True:
        text = raw_input()
        converted = convert(text, mapping)
        if converted.strip():
            if outfile:
                outfile.write(converted)
                outfile.flush()
            print converted,
```

```
except EOFError:
    exit()
except:
    try:
        while True:
            text = raw_input()
            if outfile:
                outfile.write(text)
                outfile.flush()
            print(text)
    except:
        exit()
```

The Script Used for Expansion of Repetitive Templates

```
import sys
import os
import re

files = [file for file in os.listdir('.') if file.endswith('.sv')]

file_header = """
//------------------------------------------------------------------------------------------
//                                    {}
//------------------------------------------------------------------------------------------


"""

state_tracking_header = """
//This state machine tracker was generated by {1}
`define DEBUG_LEVEL_NEEDED 1
{0} old_{0};
always @(posedge clk) begin
    if(`DEBUG_LEVEL >= `DEBUG_LEVEL_NEEDED) begin
        old_{0} <= {2};
        if({2} != old_{0}) begin
            case({2})"""

state_tracking_case = """
            {0}:
                $display("{1: <10}  #%d  {2} is now state {0}", unique_id);"""

alt_state_tracking_case = """
            {0}:
                if({3}) $display("{4}{1: <20}{5}  #%d  {2} is now state {0}", unique_id);"""

state_tracking_footer = """
            endcase
        end
    end
end
`undef DEBUG_LEVEL_NEEDED



"""

file_color_map = {
    "col_access.sv": 'red',
    "mat_mult_afu_template.sv": 'yellow',
    "pb_memory_controller.sv": 'green',
    "processing_element.sv": 'blue',
    "writeback.sv": 'magenta',
    "pb_memory_cache": 'cyan'
}

color_code_map = {
```

```
    'red': "\033[97;41m",
    'yellow': "\033[97;43m",
    'green': "\033[97;42m",
    'blue': "\033[97;44m",
    'magenta': "\033[97;45m",
    'cyan': "\033[97;46m",
    'reset': "\033[37;40m"
}

header_file = open("states_header.h", 'w')
header_file.write("//This header was generated by {}\n\n".format(sys.argv[0]))
header_file.write("#ifndef STATES_HEADER\n")
header_file.write("#define STATES_HEADER\n\n")

header_array_start = "const char* {}[] = {{\n"
header_array_index = '\t"{}",\n'
header_array_end = "\n};\n\n\n"


enum_pattern = re.compile(r"typedef enum logic \[[^:]:[^:]\]\s*{\s*\r?\n?")

unique_id = 0

for infilename in files:
    if "-v" in sys.argv or "--verbose" in sys.argv:
        print("expanding {}".format(infilename))

    if infilename in file_color_map:
        file_color = file_color_map[infilename]
    else:
        file_color = 'reset'
    file_color = color_code_map[file_color]
    reset = color_code_map['reset']

    if infilename.endswith("_template.sv"):
        filename = infilename[:-12]
    else:
        filename = infilename[:-3]
    outfilename = "generated/.tmp/{}.sv".format(filename)
    with open(infilename, 'r') as infile:
        lines = infile.readlines()

    module_name = filename.replace("_", " ")

    reading_a_repeat = False
    repeat_was_found = True

    defines = dict()

    for index, line in enumerate(lines):
        if line.strip().startswith("`define"):
            parts = [part.strip() for part in line.split(' ') if part.strip()]
            if len(parts) == 3:
                defines.update({'`' + parts[1]: parts[2]})
        if "%%assign_unique_ids" in line:
            numstr = line.split('(')[1].split(')')[0]
            nums = []
            for num in numstr.split(","):
                if num in defines:
                    num = defines[num]
                nums.append(int(num))
            num_to_generate = nums[0]
            increment = 1
            if len(nums) > 1:
                for num in nums[1:]:
                    increment *= num
            to_replace = "%%assign_unique_ids(" + numstr + ")"
            generated = '{'
            for i in range(num_to_generate):
                generated += "4'd{}, \n".format(unique_id)
                unique_id += increment
```

```
            generated = generated[:-3] + "\n}"
            lines[index] = line.replace(to_replace, generated)

        elif "%%assign_unique_id" in line:
            lines[index] = line.replace("%%assign_unique_id", "4'd{}"
                                        .format(unique_id))
            unique_id += 1

    if unique_id > 15:
        print("ERROR: tried to assign too many unique ids: {}".format(unique_id))
        exit(1)

    while repeat_was_found:
        for index, line in enumerate(lines):
            if line.strip().startswith("`define"):
                parts = [part.strip() for part in line.split(' ') if part.strip()]
                if len(parts) == 3:
                    defines.update({'`' + parts[1]: parts[2]})
            if line.strip().startswith("%%repeat"):
                if not reading_a_repeat:
                    comm, var, start, end = line.strip().split(' ')
                    beginning_index = index
                    reading_a_repeat = True
                    if end in defines:
                        end = defines[end]
                else:
                    ending_index = index
                    reading_a_repeat = False
                    template = lines[beginning_index + 1:ending_index]
                    reps = []
                    i = int(start)
                    while i <= int(end):
                        new_rep = [s for s in template]
                        for index, l in enumerate(new_rep):
                            new_rep[index] = l.replace("%%{}".format(var), str(i))
                        reps += new_rep
                        i += 1
                    lines = lines[0:beginning_index] + reps + lines[ending_index+1:]
                    repeat_was_found = True
                    break
        else:
            repeat_was_found = False

    enums = dict()
    starting_lines = []

    reading_an_enum = False
    curr_enum = None
    for i, line in enumerate(lines):
        if reading_an_enum:
            curr_enum.append(line)

            if ';' in line:
                reading_an_enum = False

                states = []

                for line in curr_enum:
                    line = line.strip()
                    if line.endswith(';'):
                        type_name = line[1:-1].strip()
                        break
                    else:
                        if line[-1] == ',':
                            line = line[:-1]
                        states.append(line)
                else:
                    print("File {}, line {}: Didn't find a type name for an enum!"
                          .format(infilename, starting_lines[i]))
                    exit(1)
```

```
                enums.update({type_name: states})

            if enum_pattern.match(line):
                curr_enum = []
                starting_lines.append(i)
                reading_an_enum = True

        tracked_state_machine = True
        while tracked_state_machine:
            for i, line in enumerate(lines):
                line = line.strip()
                if "%%track_state_machine" in line:
                    args = line.split("(")[1].split(")")[0]
                    args = [arg.strip() for arg in args.split(',')]
                    if len(args) == 2:
                        args += ['1']
                    type_name, state_name, spokesman = args
                    tracker = []
                    tracker.append(state_tracking_header.format(
                                        type_name,
                                        sys.argv[0],
                                        state_name
                                    ))
                    for state in enums[type_name]:
                        tracker.append(alt_state_tracking_case.format(
                                            state,
                                            module_name,
                                            state_name,
                                            spokesman,
                                            file_color,
                                            reset
                                            )
                                        )
                    tracker.append(state_tracking_footer)

                    header = header_array_start.format(type_name)
                    for state in enums[type_name]:
                        header += header_array_index.format(state)
                    header = header[:-2]
                    header += header_array_end
                    header_file.write(header)

                    lines = lines[:i] + tracker + lines[i+1:]
                    break
            else:
                tracked_state_machine = False

        with open(outfilename, 'w') as outfile:
            outfile.write("//This file was modified by {}\n".format(sys.argv[0]))
            for line in lines:
                outfile.write(line)

header_file.write("#endif\n")
header_file.close()
```