Scalable Machine Learning Using FPGAs

Authors: Mark Gorelik, Theodor Johansson, Jared Rodriguez: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—A system capable of training many machine learning models quickly and at a lower price point than modern GPUs.**

***Index Terms*—Machine Learning, Distributed Systems, FPGA, GPU**

1 INTRODUCTION

Currently, machine learning research involves solving the dual problems of hyper-parameter optimization and feature selection, and solutions to these problems always involve expensive guess-and-check procedures. These guess-and-check procedures always have the same form: a set of models with different hyper-parameters is trained on various transformations of a data set, and some procedure assesses their quality when deciding which models to train next.

The current norm is to train these models in parallel on a GPU because CPU computing is much slower. Machine Learning researchers tend not to have funds for their own GPUs, so many departments use a communal cluster, which results in crowded infrastructure and long wait times.

The goal of this project is to develop a system that can train a large number of machine learning models in a short amount of time at a lower price point than a GPU. We measure this with a metric that we call "model throughput": The number of models trained from start to finish divided by the amount of time taken. We verify our system with a written benchmark suite of models to be trained on a nonlinear function defined in our benchmark file, using a suite of models that are representative of the size of models used in embedded systems.

2 DESIGN REQUIREMENTS

In order to exploit data locality, we must be training more than one model on the same input sample on the same FPGA at a single point in time. Our design consists of a Data Source Machine, a Worker Board, and a bus between the two. Since hyper-parameter optimization and feature selection are memory-hard problems, our main constraint is the throughput on the SDRAM bus on the Worker Board. Our implementation exploits data locality by training multiple models on one board: the largest model in our benchmark suite requires 3.5MB for weights, gradients, and intermediate space.

From the user's point of view, our system needs to satisfy the following requirements:

- User can assign a set of models to a cluster of Worker Boards
- User can stream input data points to the cluster, and the relevant models on relevant board will train on those points
- User can retrieve metrics such as loss and accuracy regarding models on any board at any time
- User can retrieve model weights from any model on any board at any time

Our overall metrics are model throughput and model throughput per dollar:

Let N be the number of models, T be the amount of time taken to train every model, and C be the cost of the system that trains the models.

$$\text{Throughput} = \frac{N}{T} \tag{1}$$

$$\text{Throughput per dollar} = \frac{N}{T * C} \tag{2}$$

After the move to online learning, we developed a contingency plan to calculate these values using simulated cycle counts. The equation we use to calculate model throughput is described as:

T_a	Time taken to assign all models to a board
T_{t1}	Time taken to train all models on a board on one input
T_N	Time taken to train all models on a board on the entire dataset
L_B	Bus latency for sending a model
L_D	Data Pipeline Router latency for assigning a model
C_1	Clock cycles taken to train the slowest model on a board on one input, given that the weights and input are already in memory
N_M	Number of models on a board
M_1	Model Throughput for one board
M_N	Model throughput for all boards

$T_a = N_m * (L_B) + (L_D)$

$T_{t1} = (C_1) * (50M \text{ clock cycles per second})$

$T_N = N * T_{t1}$

$M_1 = N_M / (T_a + T_N)$

$M_N = \text{sum of } M_1 \text{ for each board in use}$

Due to our difficulty in building an integrated system, all of the hardware metrics (and therefore, overall metrics) that we present in this paper are calculated using simulated cycle counts.

3 ARCHITECTURE OVERVIEW

3.1 Important Terminology

3.1.1 Data Pipeline

A Data Pipeline is a function that is applied to an input and extracts features which are then used as input to a neural network. Our system will have two data pipelines: the identity transformation and the grayscale transformation. We chose these two pipelines to demonstrate that our system can train suites of models that process different features. In the end product, the user will be able to specify their own data pipelines and assign their own models to them.

3.1.2 Worker Board

A "Worker Board" is simply an abstract term for a computer (or group of computers) that connects to Ethernet and communicates as a worker using our Transport Layer Protocol [1]. In our implementation, a Worker Board is a Raspberry Pi with a GPIO connection to a DE0-Nano board.

3.1.3 Memory Handle

An efficient compute system should avoid copying data and simply pass pointers to data wherever possible. In our implementation, these pointers are called "memory handles", since they also contain control signals for memory units. In code, a memory handle is:

```
interface mem_handle;
  logic [`ADDR_SIZE-1:0] region_begin,
                         region_end,
                         ptr;
  logic                  w_en, r_en;
  logic                  avail, done;
  logic                  write_through, read_through;
  logic [`DATA_SIZE-1:0] data_store, data_load;
endinterface
```

The memory handle struct contains information about a region of memory that is not to be shared with any other memory handle. What this means is that no model will ever read to or write from another model's weight, gradient, or intermediate space. This assumption allows us to implement write-back caches, which reduce traffic on shared memory buses and increase model throughput.

3.2 System Design

Figure 1 in Appendix A shows our Top-Level Block Diagram. Our architecture is divided into three components: Software, Bus, and Hardware, which were chosen so that work could be efficiently divided between all three group members.

3.2.1 Software

The software side includes the point of interaction between the user and the system. This is where the user will define the datasets to be used, the transformations to be applied to the inputs, and the models that will be trained. This information is sent via a Python API to a Workload Manager, which assigns models and data pipelines using Algorithms 1 and 2 below.

We chose to use a Python API for interaction between the user and the system because an API would be easy for us to write and will explicitly satisfy all of the user-level requirements listed in section (2).

The Workload Manager will facilitate interactions with the Worker Boards using the Transport Layer Protocol we wrote and described in [1]. The Workload Manager is in charge of assigning data pipelines and models to boards, using Algorithms 1 and 2 below.

Algorithm 1: Allocation of data pipelines to boards to ensure equitable distribution

```
Result: Data pipelines will be assigned to boards
        in an equitable fashion.
for dp in data_pipelines do
    calculate proportion of models assigned to dp;
end
for board in available_boards do
    calculate proportion of model managers assigned to board;
end
for dp in data_pipelines do
    while dp.proportion_of_managers
     dp.proportion_of_models do
        Assign dp to the first available board;

        available_boards = available_boards[1:];

        dp.proportion_of_managers +=
        board.proportion_of_managers;
    end
end
```

The algorithm above assigns data pipelines to boards in an equitable manner, meaning that data pipelines with more work that needs to be done will have more computing power assigned to them. This necessity manifests in the attribute that a data pipeline with P_m percent of models that need to be trained is assigned to boards whose model managers sum up to around P_m percent of model managers in the system. This design decision was made because it will maximize model throughput: in order to do so, work must be divided equitably in the system.

Algorithm 2: Allocation of models to boards to minimize unused resources

```
Result: Models will be assigned to boards in a
        way that will reduce the amount of idle
        resources.
for dp in data_pipelines do
    untrained_models = dp.models;
    sort untrained_models in decreasing order by size;
    while untrained_models not empty do
        for model in untrained_models do
            if board has space for model then
                assign model to board;
            end
        end
    end
end
```

In accomplishing the same goal of maximizing model throughput, we wrote the Algorithm 2 to pack as many models as possible on a single board at once. In order to maximize model throughput, we must allow as few model managers to be idle as possible at any point in time, and thus we must maximize utilization of the model managers on a board. Since we are constrained by both model memory requirements and the number of model managers on a given board, our algorithm will simply assign models to a board until either (a) the board runs out of model managers or (b) all of the models remaining require more memory than is available on the given board. Unfortunately, we were not able to implement these algorithms in time. As a result, the current code alternates assignment between the available boards, making sure all boards have an even amount of models.

3.2.2 Bus

The bus manages the physical connection and data transfer between the data source machine and the Worker Boards. Packets are routed via TCP/IP over Ethernet connected to an unmanaged switch organized in a star topology.

For high throughput, the Worker Boards use a gigabit Ethernet connection. The current boards do not have an Ethernet PHY chip, instead connecting to a Raspberry Pi via SPI bus operating at the maximum clock speed available for reliable transfer. The SPI bus acts as the main bottleneck, not reaching the potential throughput of the Ethernet connection, but is still fast enough to support necessary data transfer.

3.2.3 Hardware

Originally, we planned to support convolutional and other supporting layers (i.e. MaxPool, Flatten) in our system. After the move to online learning, we realized that the amount of work that would entail, and we decided to move convolutional layers out of scope and focus on building the architecture before implementing complex convolution operations. Our final system does not support the convolutional and supporting layers.

The core component of the Hardware System is the FPGA board that performs computation. The definition of "Worker Board" includes a Raspberry Pi that interfaces with the Ethernet bus and IP network, but the Raspberry Pi is included in the Bus component of the project because the Hardware component is very complex and we need to distribute work equally between group members. The FPGA board interacts with the Raspberry Pi over GPIO, which is to be implemented as part of the Bus component of the project.

The Hardware System itself has four core components that are synthesized onto the FPGA chip. There is a Data Pipeline Router, which facilitates interaction with the Software System over the Bus System. The Data Pipeline Router interacts with a bank of Model Managers, which individually control the training process for a single model. The Model Managers and Data Pipeline Router both have connectivity to the Memory Management Unit, which facilitates reads and writes to both the M9K blocks and the SDRAM chip in a universal address space. Each Model Manager will expose its connections to the MMU to a port on the FPU Bank, which will control the matrix computations involved in training a model and will drive signals going to the MMU.

In our ideal solution, there is more than one model per data pipeline assigned to a given board. Thus, it makes the most sense to have the Data Pipeline Router write input samples to on-chip memory and simply expose memory handles to the relevant Model Managers, thereby eliminating redundant writes, saving memory, and ultimately increasing model throughput.

There are too many memory handles in the system to have each connected to SDRAM or M9K control signals. Therefore, it makes sense to have the MMU contain SDRAM and M9K controllers that iterates round-robin over memory handles and service requests in serial. This is a necessary feature of the system, and there is no reasonable workaround. The SDRAM chips on the DE0-Nano can store 32MB, whereas our largest model takes approximately 3.5MB for weights, gradients, and intermediate storage. Storing multiple models on one board will not be an issue.

Our FPU system should be agnostic to the number of FPUs that can be synthesized onto the hardware. Since we want to have the model managers idling for as little time as possible, our design contains one FPU Job Manager per Model Manager on the board.

4 DESIGN TRADE STUDIES

4.1 Board Selection

Our system needs an FPGA chip with headers for GPIO. There are many such available chips, but since we care about the price of our setup, we aimed to buy the

simplest boards that we could. The DE0-Nano turned out to be one of the better options because it was cheap ($69), contains hardware that we are familiar with (Cyclone IV chip), and are stocked by the ECE department. The DE10-Nano board was another good option because it remains low-price ($110) and has a built-in network stack, but the uncertainty of learning to use this physical layer posed a risk. At the end, we decided that it was safer to write a GPIO transport protocol between a Raspberry Pi and a DE0-Nano board.

4.2 Transfer Protocol

The system requires bidirectional data transfer between the Worker Boards and the host software. Our first consideration was to use a CAN bus, however the low data throughput led us to reconsider and settle for Ethernet. Currently each board requires an Ethernet controller for each Worker Board, raising the cost of individual workers in exchange for the large boost in throughput.

Between the Raspberry Pi and the workers, a simpler protocol was necessary. The initial protocol featured a 16-bit wide, asynchronous bus for data transfer to the worker, with an 8-bit wide bus for data from the worker. Early tests and library research revealed that the Raspberry Pi had lower expected throughput due to the time required to toggle individual pins. Further considerations involved libraries native to the Raspberry Pi, namely I2C and SPI. SPI has the higher throughput of the two, and therefore is present in the current design.

4.3 Software

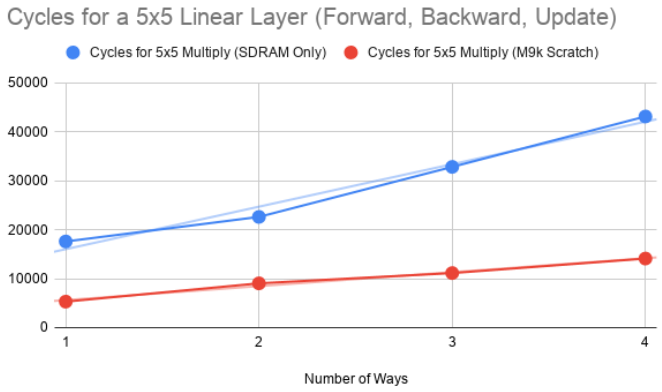
One of the main design choices for the Software portion of the project was deciding what language to write our code in. We agreed on using Python for several reasons, primarily due to familiarity with coding languages and level of complexity. Python and C were the immediate front runners since all three of us had equal familiarity in both those languages as opposed to Java or other high level programming languages. Although C programs execute faster than Python programs, it is much easier to write code in Python due to syntax and memory management, which is an important factor as as user will also be expected to write some code in order to add their ML models to their respective pipelines. Additionally, Python is much easier to debug and error check as opposed to C, making it easier for both ourselves and our users to validate the correctness of the code.

4.4 Matrix serialization

Initially, our plan was to implement convolution operations to support Convolutional, MaxPool, and Flatten Layers that enable image processing. In order to use the fastest memory access patterns, we needed to access memory in straightforward incremental walks from one address to another. In the convolution forward, backward, and gradient operations, these could be made faster if the next memory location to be accessed (i.e. the next output channel) was at the current index in memory plus one word. For 4-dimensional filter tensors which are indexed by [output channels][input channels][height][width], this meant storing the tensor in column-major order. We made this decision uniform across the system, and all tensors (regardless of dimension) are stored in column-major order.

4.5 Scratch Memory

For any forward/backward/update pass, temporary memory regions must be allocated for the output of each layer, gradients of outputs, and weight and bias gradients for applicable layers. Our initial implementation placed these values in SDRAM, but after seeing our final model throughput, we decided to try using M9K for these scratch spaces. Below is a graph of the cycle counts for SDRAM and M9K scratch spaces, showing different counts based on the number of models present on the board.



Using M9K memory for scratch space will always use fewer cycles than SDRAM, and the cycle counts increase with a smaller slope than does the SDRAM-only implementation. At the beginning, we opted against this approach so that we would have on-chip memory to spare for the packet buffer, but it has become clear that further utilizing on-chip memory is necessary to increase model throughput.

5 SYSTEM DESCRIPTION

Our system is complex, so it is easiest to describe how the system components work together throughout the process of model training, which begins when the end user starts making calls to our Python API as described in Figure 3.

5.1 Software

First, the user defines the following:

- Data Sets
- Data pipelines
- Models and their relationships to pipelines

- An information retrieval process

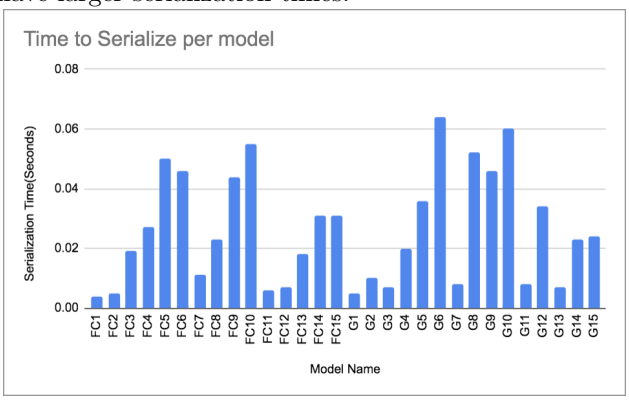
Before the Workload Manager can distribute any data pipelines or models to Worker Boards, it needs information on how many boards are in the network and how to address them. The Worker Finder will expose a database of active Worker Boards and their IP addresses to the Workload Manager. It will find active Worker Boards by broadcasting UDP packets to the IP network on which the Worker Boards are listening and record any responses it receives. This occurs once, before training time.

Before the training loop executes, the Workload Manager component of the software system will assign each data pipeline to available Worker Boards, which it discovers by interacting with the Worker Finder component of the software system. Using the Transport Layer Protocol defined in [1], the Workload Manager will assign a set of models to each Worker Board. During the execution of the training loop, the Workload Manager will stream data points in batches to relevant workers, where they will be queued in software on a Raspberry Pi before being sent over GPIO to the Data Pipeline Router on the FPGA. After training completes, the user's code will call API routes that retrieve model weights and metrics from the Worker Boards as shown in the code in Figure 3.

5.2 Software Metrics

5.2.1 Network Model Serialization Latency

This latency describes the amount of time it takes for the software side to parse through a model and serialize it to the format that we had specified in the Transport Layer Protocol. Below are the timings for each of the thirty models we defined for our test bench. The average serialization time was 0.026 seconds with a standard deviation of 0.018 seconds. Models that have more or larger linear layers tend to have larger serialization times.



5.3 Bus

The Worker Finder broadcasts discovery messages across the bus, listening for responses to tally available Worker Boards. Each Worker Board responds with the number of available Model Managers it hosts. The Worker Finder then contributes the received data to the Worker Manager to divide the workloads. Once training begins, the bus facilitates all communication between the host and Worker Boards.

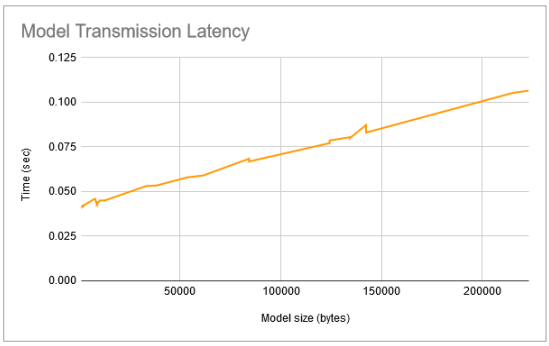
5.4 Bus Metrics

5.4.1 Throughput

With a single Worker Board, the Raspberry Pi and the hardware SPI client cooperate to translate the message from the speed of the gigabit connection to the speed of the FPGA internal clock. Due to limitations of the GPIO headers on the Raspberry Pi, the SPI operates at a maximum rate of 15.6 MHz. Furthermore, additional overhead in computation limit the effective transfer rate to 12.8 Mbps, or 1.6 MB/s.

5.4.2 Transmission Latency

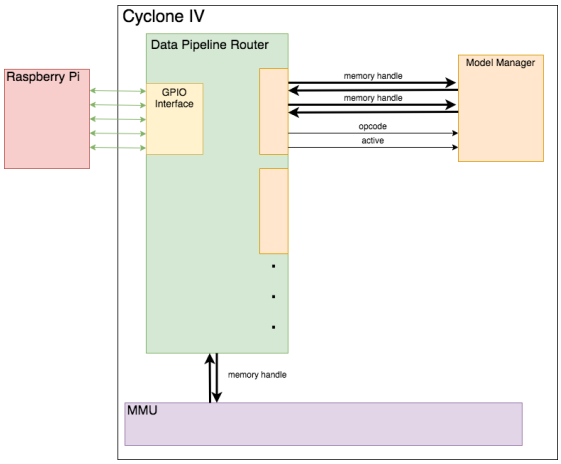
This latency describes the amount of time it takes to send a serialized model over the bus and store it in memory on the Worker Board. Larger models were sent more efficiently due to overhead on the Raspberry Pi per model.



5.5 Hardware

There are four core components of the Hardware Subsystem: The Data Pipeline Router, the Model Manager, the Memory Management Unit, and the FPU Bank.

5.5.1 Data Pipeline Router



The connectivity of the Data Pipeline Router.

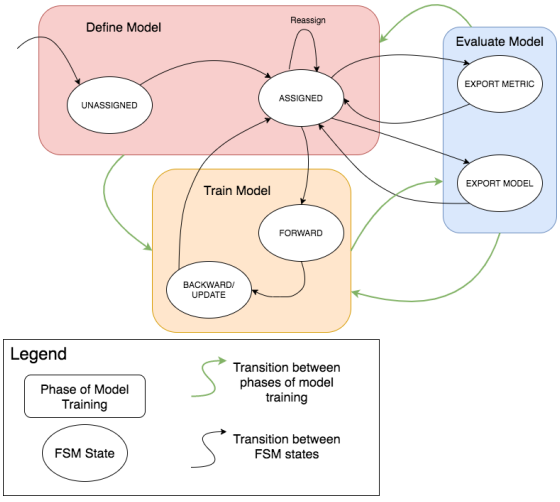
There are three core interactions between the Worker Board and the Data Source machine: model and pipeline assignment, input data streaming, and metric/weight retrieval. These use cases will be fulfilled by the Data Pipeline Router, which facilitates interactions between the Worker Board and any other components in the system.

Before training, a set of data pipelines are assigned to each Worker Board. It is possible to train models from different pipelines on the same board, but extra bus bandwidth would have to be used to stream input data from two sources to the board, so this is a less efficient decision than to simply train many models from the same Data Pipeline on one Worker Board.

After being assigned a Data Pipeline, the Data Pipeline Router will have a set of models assigned to it. Each of these models will have initial weights sent to it by the Workload Manager – this decision was made to allow for custom weight initialization methods and to avoid having to implement pseudo-random number generators in hardware. Upon receiving a model on the bus, the Data Pipeline Router will write the initial weights to off-chip SDRAM and expose a memory handle to a free Model Manager. The Model Manager is then assigned to the given model, and will be sent input data samples for the given model as they arrive.

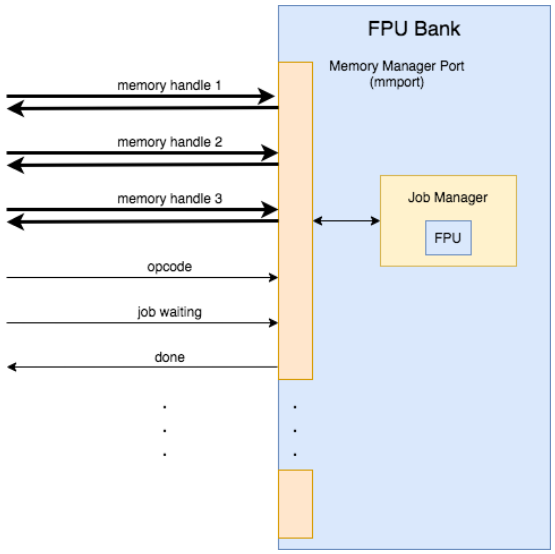
Once models have been assigned, the Worker will start to receive training samples from the Workload Manager. The Data Pipeline ID is sent in the same packet as its corresponding sample, so the Data Pipeline Router will know which models need to be trained on a given input. Upon receiving the sample, the Data Pipeline Router will write it to on-chip M9K memory (for single-cycle read), and will pass a memory handle pointing to it to each Model Manager that is training on the given Data Pipeline. The relevant Model Managers will then perform a forward/backward/update step using the given input.

5.5.2 Model Manager



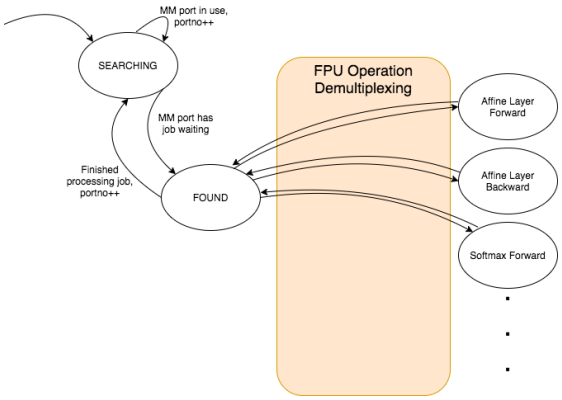
The Model Manager FSM.

Upon receiving an input sample, a Model Manager will train on it according to the Stochastic Gradient Descent algorithm. Specifically, the model will perform a forward pass (calculating intermediate layers and model outputs), a backward pass (calculating weight gradients), and an update pass (updating weights based on their gradients). In doing so, the Model Manager will set control signals to its port in the FPU Bank, which will perform matrix calculations.



Connectivity between a Model Manager and a port on the FPU Bank.

5.5.3 FPU Bank



The FPU Job Manager FSM. The FPU Bank itself is not a state machine – it is merely a shell around a bank of FPU Job Managers.

The FPU Bank has one port for each model manager and services requests that come in through these ports using a bank of FPU Job Managers, which each have an assigned Model Manager. Every Job Manager will be capable of performing any FPU operation that the Model Managers can request. In doing so, the Job Manager will drive the memory control signals in the memory handles exposed to it in order to read and write from necessary regions in memory.

The following FPU operations are supported:

$$z = Wx + b \text{ (linear forward)} \tag{3}$$

$$\frac{\partial L}{\partial x} = W * \frac{\partial L}{\partial z} \text{ (linear backward)} \quad (4)$$

$$\frac{\partial L}{\partial W} = x * \frac{\partial L}{\partial z} \text{ (linear weight gradient)} \quad (5)$$

$$W = W + \lambda \frac{\partial L}{\partial W} \text{ (linear weight update)} \quad (6)$$

$$\frac{\partial L}{\partial b} = 1 \text{ (linear bias gradient)} \quad (7)$$

$$b = b + \lambda \frac{\partial L}{\partial b} \text{ (linear bias update)} \quad (8)$$

$$z = ReLU(x) \text{ (ReLU forward)} \quad (9)$$

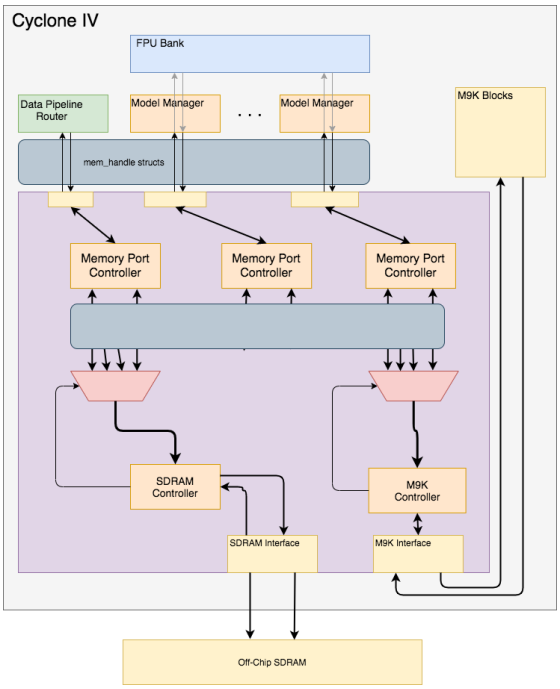
$$\frac{\partial L}{\partial X} = ReLU\_bw(\frac{\partial L}{\partial z}) \text{ (ReLU backward)} \quad (10)$$

$$L = \sum_i (y_i - \hat{y}_i)^2 \text{ (MSE forward)} \quad (11)$$

$$\frac{\partial L}{\partial \hat{y}} = -2(\sum_i (y_i - \hat{y}_i)) \text{ (MSE backward)} \quad (12)$$

This list has changed since our Design Review Document in that we have removed convolutional, max pool, and flatten layers, but also in that we have switched Cross Entropy and Softmax for Mean Squared Error, which is much easier to implement in hardware.

5.5.4 MMU



The Hardware System Memory Hierarchy. The MMU has no control FSM, and is simply a shell around the Memory Port Managers and the SDRAM/M9K controllers.

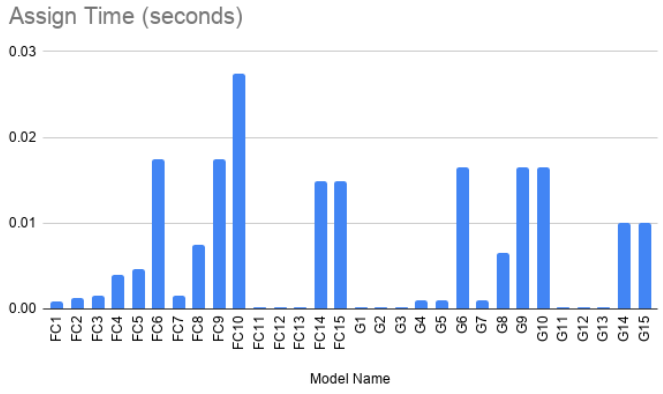
Finally, the Memory Management Unit (MMU) services write and read requests to and from both on-chip M9K memory blocks and off-chip SDRAM. Every memory handle that exists in the Data Pipeline Router and Model Managers will have a memory handle port, which interfaces with the MMU. Each memory handle port has a memory port controller, which maintains a small memory cache. If the address coming in points to a place within the cache, then the read or write will be made from or to the cache in the memory port handler. Since there are no regions in memory that will have two components writing to them, it is safe to implement a lazy cache write policy, where the entire cache is written only once a request comes in that misses the cache. This way, we hope to reduce bus traffic on the SDRAM and speed up matrix computation.

The MMU will have two controllers that interface with actual memory: the M9K controller and the SDRAM controller. Because of the actual constraints defined by the specifications for M9k and SDRAM components, there can only be one controller for each of these, so they will iterate round-robin over ports and service requests in an atomic and serial manner.

5.6 Hardware Metrics

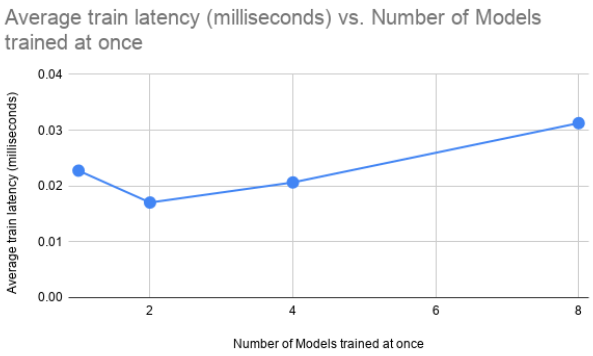
5.6.1 Model Assignment Latency

Model Assignment Latency is measured as the time delta between an ASN_MODEL packet begin presented to the Data Pipeline Router and a model manager reaching the ASSIGNED state after having the model assigned to it. This is a one-time cost, so it has very little effect on the model throughput compared to recurring costs like Model Train Latency. For our model suite, assignment latency varies from 0.1 to 27 milliseconds with an average of 6 milliseconds.



5.6.2 Model Training Latency

Model Training Latency is the largest cost in our system because it will be encountered once for every batch to be trained, or 12,500 times in our benchmark suite. Model Training Latency varies depending on the number of layers in a model, the size of each model, and the number of models being trained in parallel. The minimum latency found was 1.4 milliseconds (Only one small model being trained on a board) and 1,860 milliseconds (Eight models being trained in parallel, with the largest model determining latency for the entire group).

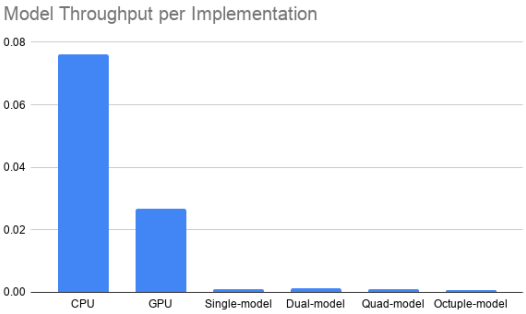


Model Training Latency is the main determiner of Model Throughput for the entire system. We found that our Model Train Latency is minimized when only two models were being trained on a board, meaning that the SDRAM sees a lot more usage than we expected. Our caching system helps decrease the amount of traffic on the SDRAM bus, but not as much as we hoped.

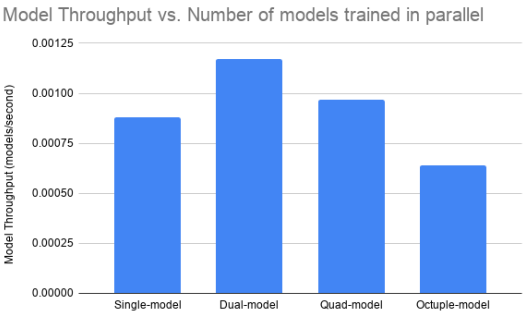
5.7 Overall System Metrics

5.7.1 Model Throughput

In the end, we found that our CPU and GPU benchmarks had significantly higher model throughputs than our FPGA system.



Our model throughput peaked when we were training two models simultaneously.



We were hoping that the model throughput would peak at a higher value, and when more models were being trained on a board at once. The bottleneck in the final system is the throughput on the SDRAM bus, which causes our FPU operations to stall. We expected that the SDRAM Controller spend much more time stalling than was actually observed, and thus would allow increasing model managers to increase the model throughput linearly. In reality, the SDRAM Bus reached its peak utilization when only two model managers were operating on a board at once. We can reduce congestion on the bus by using M9K memory as scratch space as described in subsection 5 of the Design Trade Studies section, but getting the most out of the M9K would require more optimizations than we have time for. We have produced preliminary cycle counts for a 5x5 Linear Layer that uses M9K memory for all scratch space, but the speedup is not enough to make our system's model throughput comparable to those of the CPU and GPU. Since M9K blocks are divisible in 8192-Kb blocks which are each dual-ported, the M9K memory has much more potential than what we are able to utilize by changing scratch space addresses to the M9K region of memory.

6 PROJECT MANAGEMENT

6.1 Schedule

Our Gantt chart is located in Figure 2. Our schedule changed from the design review in that wehad to lengthen many tasks, for example, the FPU Bank development. Many things turned out to be harder than we expected, and we had to accomodate by cutting into our planned integration time.

6.2 Team Member Responsibilities

The project responsibilities are divided into three primary groups: Software, Bus Protocol, and Hardware. Mark Gorelik is in charge of the Software portion, and focuses on the Bus as a secondary responsibility. Jared Rodriguez is the lead engineer on the Bus Protocol, and has Hardware as a secondary responsibility. Theodor Johansson took point on the Hardware portion of the project, and is focusing on the Software aspect as a secondary task. As the semester went on, Jared had increasing difficulty debugging the SPI bus due to lack of equipment, and getting this bus working became the sole target of his effort. Along the way, Theodor realized that he could not implement all operations necessary for Convolutional Layers alone, which were ambitious to begin with. Thus, we restricted the FPU operations list to include only what was necessary to implement linear feedforward neural networks.

6.3 Budget

A full breakdown of the prices of each part can be found in Table 1 in the Appendix, but there are several notes about these values. The total cost of the system is assuming that no parts are able to be acquired from academic or other types of discounts. This is an important value to have because it is what it would cost a researcher to replicate our system. On that note, we were fortunate enough to be able to use two DEO-Nanos provided by the university for free as well as purchase the remaining two FPGAs with the academic listed price ($61), reducing our total cost to

Item	Cost	Quantity	Total Cost
DE0-Nano Development and Education Board	$79	2	$188
Raspberry Pi 2B	$35	2	$70
NETGEAR 5-Port Gigabit Switch	$33	1	$33
8-Pack 5-foot Cat5e Ethernet Cable	$16	1	$16
		Total	$XXXX

Table 1: Costs of hardware

$311. We believe that we will use a certain portion of the remaining budget on AWS credits in order to train our test set of models on different GPUs as a form of benchmarking and validation against our FPGA implementation. Additionally, because our project is designed to allow a variation of the number of boards connected to the Network Switch, the price can be reduced significantly by opting to work with fewer boards.

6.4 Risk Management

After spring break, it was made clear that we would have no more physical work sessions in the HH1307 where we have easy access to resources to build the circuits needed for the SPI bus implementation. As a result, we produced a contingency plan that we could use to calculate model throughput given simulated cycle counts needed for model assignment time and model train time.

T_a	Time taken to assign all models to a board
T_{t1}	Time taken to train all models on a board on one input
T_N	Time taken to train all models on a board on the entire dataset
L_B	Bus latency for sending a model
L_D	Data Pipeline Router latency for assigning a model
C_1	Clock cycles taken to train the slowest model on a board on one input, given that the weights and input are already in memory
N_M	Number of models on a board
M_1	Model Throughput for one board
M_N	Model throughput for all boards

$T_a = N_m * (L_B) + (L_D)$

$T_{t1} = (C_1) * (50M \text{ clock cycles per second})$

$T_N = N * T_{t1}$

$M_1 = N_M / (T_a + T_N)$

$M_N = \text{sum of } M_1 \text{ for each board in use}$

In the end, we did not get the SPI bus working in time for us to completely integrate the system. Thus, all of the hardware metrics in this paper are from simulated testbenches.

7 RELATED WORK

There are a couple of hardware solutions such as Google's Tensor Processing Unit (TPU) [6] and the Intel Neural Compute Stick [5], and these fill similar niches to that of our project. However the TPU is built for deploying rather than training models. The Intel Neural Compute Stick is expensive at $75, and we believed we could achieve higher model throughput than this solution.

8 SUMMARY

While our software and bus components met their requirements in preparing data for use by the hardware component, the SDRAM bottleneck reduced our model throughput below the benchmarks we set early in the project. We could increase the model throughput of our system by utilizing M9K memory more, but we did not have time to do so.

8.1 Lessons Learned

An FPGA-based solution to hardware computation should be dead simple (i.e. implement the bare minimum number of operations) and should essentially represent a fast memory architecture for making vectorized computation.
When using extra hardware like the Raspberry Pi, some specifications do not represent real capability. The maximum speed for the SPI clock was much higher than that capable of sending data.

9 Bibliography

References

[1] Mark Gorelik, Theodor Johansson, and Jared Rodriguez. *Transport Layer Protocol.* https://docs.google.com/document/d/1I2FRMwITUbSbkqIw_w6eQ5OyxKJneer853 xGVAFtx5I/edit

[2] Mark Gorelik. *Software Code.* https://github.com/CMU-18-500-TeamTensor/Benchmarks

[3] Jared Rodriguez. *Bus Code.* https://github.com/CMU-18-500-TeamTensor/SPI-Bus

[4] Theodor Johansson. *Hardware Code.* https://github.com/CMU-18-500-TeamTensor/HardwareWorker

[5] Intel, Inc. *Intel Neural Compute Stick.* https://software.intel.com/en-us/neural-compute-stick

[6] Google, Inc. *Cloud Tensor Processing Units (TPUs).* https://cloud.google.com/tpu/docs/tpus

10 Appendix

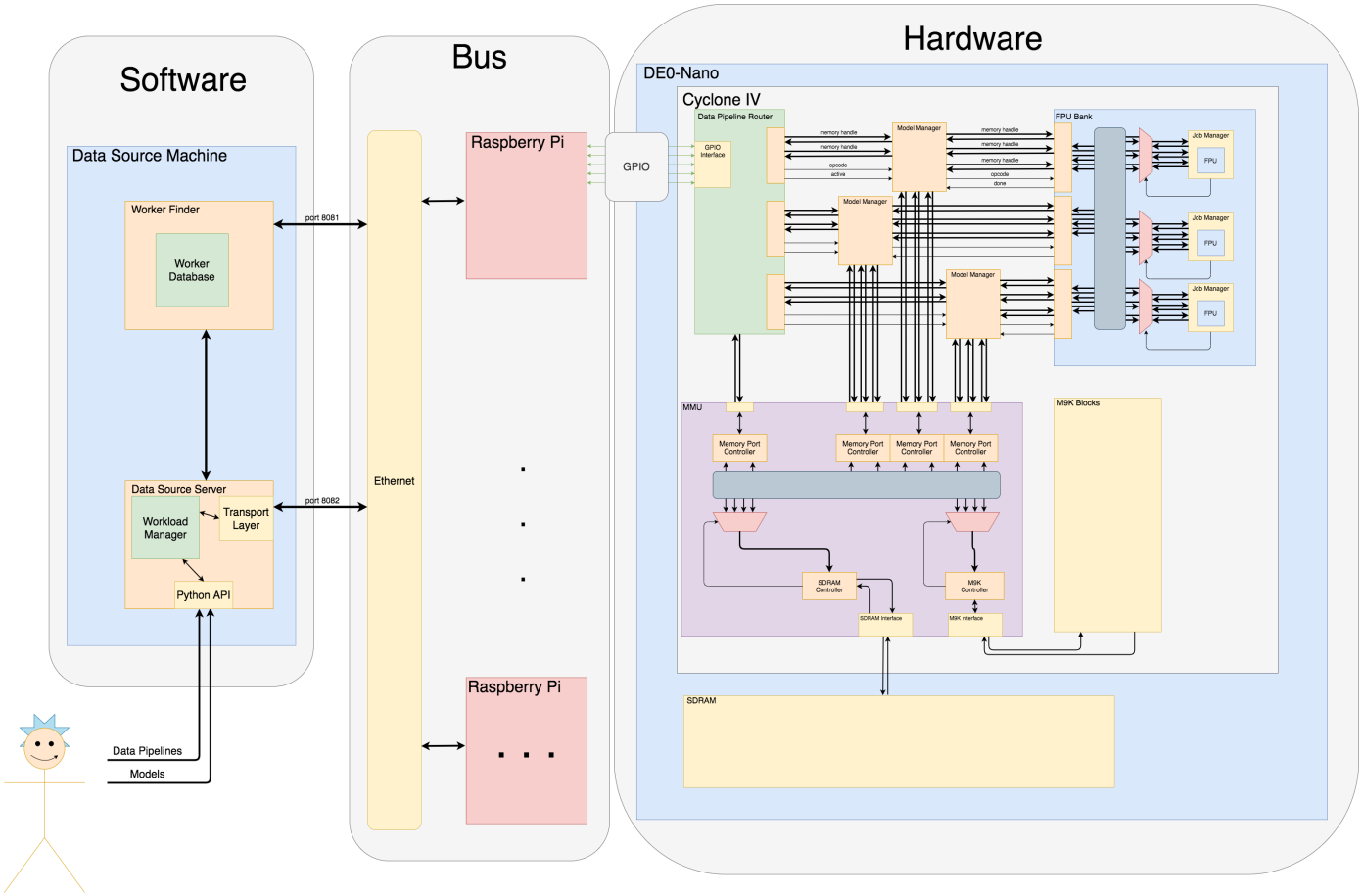


Figure 1: The Top-Level Block Diagram for the System.

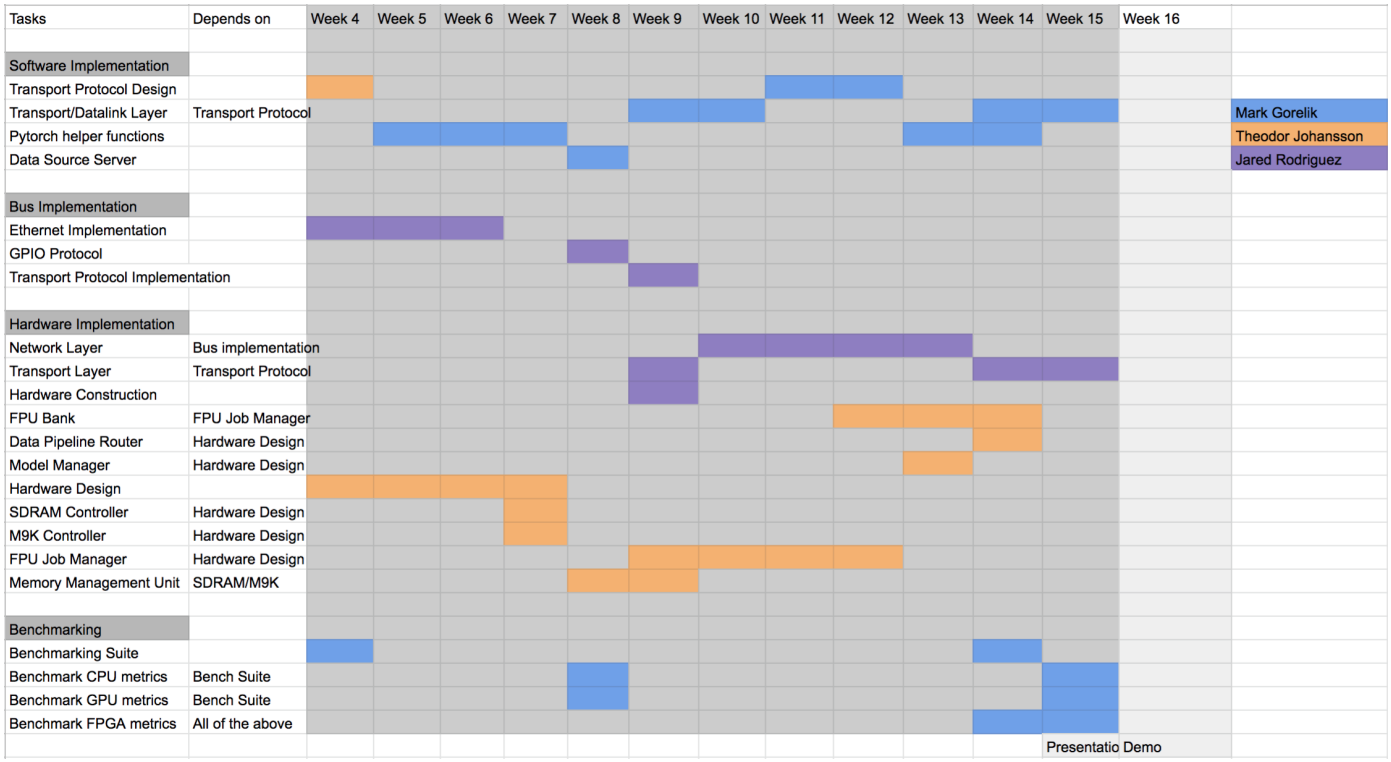


Figure 2: Gantt Chart

```python
def main():
    dpm = DataPipelineManager(1)

    # add_pipeline() returns the int ID of the data pipeline, as seen by the
    # FPGA
    # buffer size is the number of samples can be stored in the Data Source Server
    # buffer.

    #user function to pull dataset
    trainloader = get_CIFAR10_dataset() # iterable over tuples (x, y)


    #grab the whole model list, in this scenario just one model
    model_list = bench_suite

    pipeline_fn, fc_models = model_list["full_color"]
    pipeline_fn2, gc_models = model_list["grayscale"]
    dp_id1 = dpm.add_pipeline(pipeline_fn, "full_color", 10)
    dp_id2 = dpm.add_pipeline(pipeline_fn, "grayscale", 10)


    #User specificed which model goes to which pipeline
    #In this scenario, we are adding all full color models to the full color pipeline
    #and all grayscale models to the grayscale pipeline
    for i in range(15):
        fc_model = fc_models[i]
        gc_model = gc_models[i]
        dpm.add_model(fc_model, dp_id1)
        dpm.add_model(gc_model, dp_id2)


    #Once all the models have been added to their respective pipelines and the data
    #has been pulled, we begin training the models
    #Pass in data pipeline manager and trainloader
    train_models(dpm, trainloader)
```

Figure 3: Example code detailing exactly how the user will interact with the Python API.