Fall 2022

A RISC-V Matrix Multiplier Using Systolic Arrays

Miao Wang
San Jose State University

A RISC-V MATRIX MULTIPLIER USING SYSTOLIC ARRAYS

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Miao Wang

December 2022

The Designated Thesis Committee Approves the Thesis Titled

A RISC-V MATRIX MULTIPLIER USING SYSTOLIC ARRAYS

by

Miao Wang

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

SAN JOSÉ STATE UNIVERSITY

December 2022

Nima Karimian, Ph.D.	Department of Computer Engineering
Younghee Park, Ph.D.	Department of Computer Engineering
Kaikai Liu, Ph.D.	Department of Computer Engineering

ABSTRACT

A RISC-V MATRIX MULTIPLIER USING SYSTOLIC ARRAYS

by Miao Wang

Many modern day applications can be solved with the usage of machine learning, which involves training a computer to learn on large amounts of data without direct programmer guidance. Conventional computers typically use normal general purpose central processing units, though more specialized tasks may take advantage of more parallel hardware such as graphics processing units. In the pursuit of increased performance to facilitate increasingly more complex machine learning models, researchers in both academia and industry look towards field-programmable gate arrays and application specific integrated circuits for their needs. Various implementations, both theoretical and practical, exist across a wide variety of designs. A custom design, using systolic arrays and built on the existing RISC-V Instruction Set Architecture, will be used to accelerate matrix calculations, with example performance on the MNIST dataset measured.

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

Machine learning is a burgeoning field in modern computing, concerned with drawing patterns and relationships out of huge quantities of data. As technology progresses, larger and larger amounts of data must be collected, with corresponding increases in computational complexity. While general purpose computing hardware such as central processing units (CPUs) have some degree of parallelization, the mathematical operations typical of machine learning involve even larger amounts of parallel operations. Graphics processing units (GPUs) are commonly used for machine learning for increased efficiency, however in extreme cases it may be desirable to pursue more dedicated custom hardware. This follows a general trend of increased specialization when it comes to high performance computing, as mentioned by Patterson in his discussion of domain specific architectures (DSAs) [1]. The application of DSAs when applied to machine learning requires that custom machine learning hardware is either mapped to a field-programmable gate array (FPGA), or even more ideally a custom-built application specific integrated circuit (ASIC). Generally speaking, machine learning performance increases as specialization increases, with the slowest being CPUs, followed by GPUs, FPGAs, and ASICs.

Custom machine learning accelerators are commonly implemented in either FPGAs or ASICs, depending on the goals of the of researchers. Typical examples of physically implemented ASIC designs commonly come from corporate researchers with access to appropriate resources, such as from E. Nurvitadhi et al. at Intel [2], [3] or N. P. Jouppi et al. from Google [4]. On the other hand, a great deal of research from academics will typically be limited to designs mapped onto FPGAs, due to the relative affordability of an individual FPGA as opposed to the entire process of fabricating a custom ASIC chip. While much of this research, from either public or private sector, focuses on accelerating machine learning algorithms of various types, the vast majority of machine learning designs reduce to optimizing the performance of highly parallel mathematical computation in the form of matrix multiplication.

This paper will discuss a custom implementation of a systolic array used for accelerating matrix multiplication calculations in hardware. The design consists of a standard 5 stage pipeline implementing a subset of the RISC-V (Reduced Instruction Set Computer V) Instruction Set Architecture (ISA), along with additional instructions that can be sent to a systolic array coprocessor used for matrix calculations. This system will implemented on a standard Terasic De-10 FPGA and used to conduct neural network inferences on a standard MNIST image set, with comparisons between speed and accuracy between various numerical formats. Due to limitations in hardware, the results may not necessarily be particularly impressive, but interesting comparisons can still be drawn between this design and more common options such as simple CPUs.

2 BACKGROUND

2.1 Machine Learning Basics

Machine learning is a broad field consisting of many different mathematical, numerical, or statistical approaches to interpreting data. While more commonly known methods are grounded in statistics, such as linear regression or various forms of statistical classification, one of the areas of most intense research involves neural networks [5]. Neural networks attempt on some level to imitate the operation of the human brain, with numerous small functional units called “neurons” which take in potentially multiple inputs and produce an output based on weights and biases generated during a process of training. Each of these individual neurons potentially represents multiple simple mathematical operations, typically consisting of multiplications and additions. Combining thousands, millions, or even billions of these neurons together into a “layer” of neurons, and then chaining these layers together, produces a neural network. These neural networks, when properly “trained” through a process of passing data along with labels (correct “answers” to the data), can produce surprisingly accurate results. Fundamentally, usage of such large neural networks boils down to large matrix operations, repeated millions of times with large sets of data. As researchers attempt to increase the accuracy of models built on neural networks, increasingly large amounts of neurons and training data are required to produce better results. For something as simple as image classification, an increase of accuracy from 93.3% to 96.4%, in other words a mere 3% increase in accuracy, results in a neural network increasing in size from 22 layers to 152 layers [5]. If the cost of increasing accuracy by 3% requires almost an order of magnitude more computational resources, it becomes clear that an obvious roadblock to improving machine learning models is the potential performance of the model itself. While it may be possible to design more efficient machine learning models that produce comparable results, the field is still quite immature and this may not be a feasible solution in the short to medium term. The only other solution would be to find a way to increase the computational resources available for running these complex machine learning models.

2.2 Computational Platforms

The typical workload for a computer CPU is concerned with very general performance across a number of domains, as the average computer will be a consumer product that must deal with a variety of tasks. For a normal person using their laptop or other computing device, they may need to spend time on work related issues such as word processing, or they may browse the internet for topical news and information, or they may stream high-definition video. It is possible that a person would do all three things simultaneously, or at least may switch between them frequently and without pattern. As a result, the computer architecture used must be able to handle a wide variety of workloads with different computational requirements and differing memory access patterns. The generalized use case will also likely be very branch heavy, due both to the workloads themselves and due to handling so much context switching. These varied workloads will inherently be less parallelizable than other more regular workloads. Due to this typical use case, the general trend in computing has been to design simpler, more modular instruction sets which can break down typical tasks into smaller and easy to execute blocks [1]. However, this is not necessarily the case for more specialized, high-performance environments such as servers, or, more relevant to this paper, machine learning. The typical machine learning task must process an extremely large quantity of data, sometimes repeatedly, in a very wide and parallelizable fashion. For these types of situations, a less generalized and more optimized approach is more appropriate. GPUs were originally designed to be used for graphics rendering, which is primarily concerned with generating a viewable image from a computer screen. This involves a great deal of geometric projection in the form of matrix computations. However, it turns out that machine learning also maps very well to GPUs, as most machine learning calculations are equivalent to common matrix operations. Thus, performing machine learning calculations on a GPU is significantly faster than on that of a generalized CPU. For situations where even more performance is needed, users must then resort to creating custom hardware.

Field-programmable gate arrays (FPGAs) are reconfigurable hardware that consist of reprogrammable logic blocks surrounded by configurable interconnects. These devices allow an end user to configure the logic blocks to do whatever specific tasks they want and configure the surrounding interconnect to connect the custom logic blocks in any way needed. Essentially, this allows a user to create hardware customized to their needs on a hardware level. By mapping specific computer architecture designs that are optimized for machine learning to an FPGA, even greater performance can be achieved than CPUs or even their more parallel cousins GPUs. However, this reconfigurability comes at a cost, as, once again, the mere ability to reconfigure something means there is generality that must be paid in a cost of performance. For the most extreme version of custom hardware, the only remaining option becomes application specific integrated circuits, or ASICs. ASICs are custom hardware that are designed from the ground up to perform a specific task, and they can only do that specific task. Once fabricated, they can no longer be changed or reprogrammed for the end user. However, because these ICs are designed specifically for a singular purpose, they are extremely efficient at executing the workloads they are designed for. The only drawback to this performance is, of course, the lack of reconfigurability, and potentially cost. This cost limits the economic viability of most designs, however, for large companies or cloud providers, they can offer incredible results [4] [6].

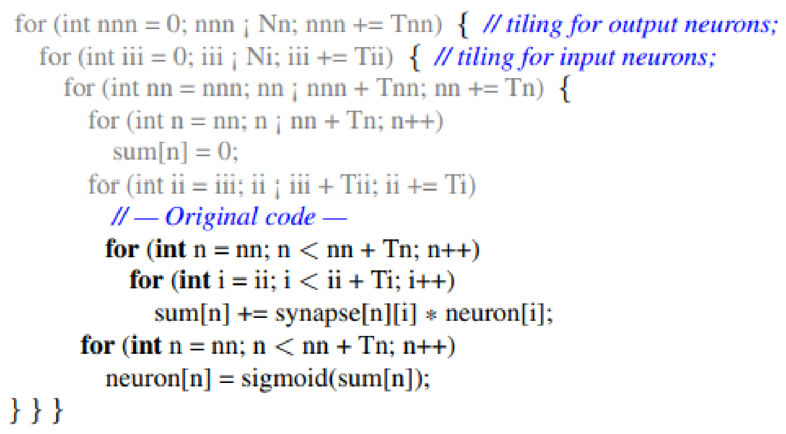
Combining knowledge about machine learning algorithms and computer architecture allows for novel designs that may be optimized for a particular type of machine learning execution pattern. Many potential hardware accelerators have been proposed by academics, such as DianNao [7] or Q-Learning designs [8]. Typically, designs done in an academic setting will either be tested through simulation in software (typically with a language such as MATLAB or C++), or they may be mapped to an FPGA for a more realistic use case. Due to the cost of design, verification, and fabrication of a custom chip, most novel designs from academia will be tested on this purely theoretical basis. However, custom machine learning hardware is also an area of interest to industry, with many companies such as Google [4] or Amazon [6] providing access to these ASICs through cloud services. Other companies such as Qualcomm [6] may offer such custom hardware to original equipment manufacturers (OEMs) which may create consumer devices that have specialized needs. Going even further, some proposed designs [4] [5] [6] [7] may be even more unconventional, taking advantage of the fault tolerance of some machine learning algorithms to integrate analog components or even theoretical memristors, to achieve performance that is potentially impossible in a purely digital realm.

3 RELATED WORK

3.1 DianNao

One of the most common use cases in machine learning is the use of Neural Networks (NNs), specifically Convolutional Neural Networks (CNNs) and Deep Neural Networks (DNNs), to solve a wide variety of problems. CNNs and DNNs consist of countless neurons organized first into horizontal layers, and then stacked together as the layers pass information forward and backwards from layer to layer. For CNNs specifically, these layers can be broadly broken down into two different types, convolutional layers, and pooling layers, each of which has slightly different memory access patterns and computational behavior. Convolutional layers essentially involve sliding an N by N convolutional filter across a memory array, performing a matrix multiplication for each iteration. On the other hand, Pooling consists of sliding a M by M filter across an array, choosing the largest, smallest, or average value within the filter. The final layers of a CNN consist of a "classifier layer", which is typically a "multi-layer perceptron", used to convert numerical data from a previous layer into a classified result [7]. For all of these types of layers, a non-linear function is applied to the output of each neuron, which allows the neural network to solve a wide variety of problems.

A major portion of optimizing neural network performance, and the focus of the DianNao paper, consists of optimizing memory accesses to improve neural network performance. As the neural network must operate on large amounts of data, carefully structuring the design to take advantage of parallel memory accesses leads to some of the greatest gains in performance. Input and Output neurons are primarily concerned with introducing novel information to the network, and since there may be a large number of input neurons, it is possible that all input data cannot fit in the L1 cache. The authors decide use memory tiling to optimize memory accesses and minimize cache usage [7]. Internal nodes, called synapses, will generally fit in an L2 cache, though that needs to be optimized as well. The optimized source code will essentially break loops down into smaller loops represented by a tile factor Tii, which essentially ensures that subloops maximize cache usage. An example of the tiling pseudocode used in DianNao can be found in Fig. 1.



```
for (int nnn = 0; nnn ¡ Nn; nnn += Tnn) { // tiling for output neurons;
  for (int iii = 0; iii ¡ Ni; iii += Tii) { // tiling for input neurons;
    for (int nn = nnn; nn ¡ nnn + Tnn; nn += Tn) {
      for (int n = nn; n ¡ nn + Tn; n++)
        sum[n] = 0;
      for (int ii = iii; ii ¡ iii + Tii; ii += Ti)
        // — Original code —
         for (int n = nn; n < nn + Tn; n++)
           for (int i = ii; i < ii + Ti; i++)
             sum[n] += synapse[n][i] * neuron[i];
       for (int n = nn; n < nn + Tn; n++)
         neuron[n] = sigmoid(sum[n]);
} } }
```

Fig. 1. DianNao Memory tiling pseudocode [7] ©2014 IEEE.

The general architecture consists of memory buffers that feed into Neural Functional units (NFUs), which "reflect the decomposition of a layer into computational blocks of Ti inputs/synapses and Tn output neurons" [7]. Multiple NFUs are arranged in sequence, taking advantage of pipelining to maximize throughput and parallelism. The input and output memory buffers are optimized to minimize cache usage and maximize performance. The general block diagram is found on Fig. 2.

As with any processor, the authors defined control logic and control instructions that would be translated into hardware operations. The control instructions could be used "in order to explore different implementations of layers, and to provide machine-learning researchers with the flexibility to try out different layer implementations" [7]. While the low-level details are beyond the scope of this literature review, the instruction set used was optimized for performance as well.

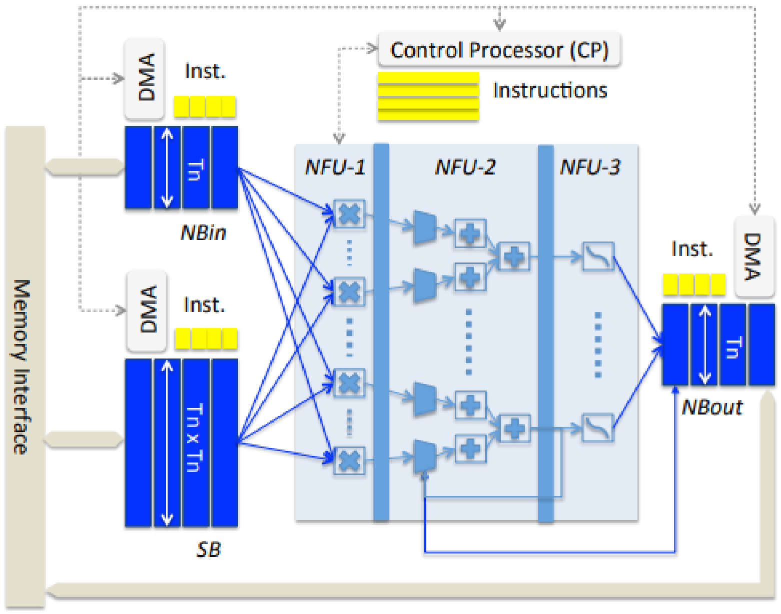


Fig. 2. DianNao Accelerator diagram [7] ©2014 IEEE.

The design was synthesized and placed and routed using Synopsys tools, with power estimated using PrimeTime PX [7]. Compared to the baseline of a GEM5+McPAT implementation, the DianNao was both faster (1000x) and used less power (20%), showing that CNN ASICs can be both faster and more energy efficient than alternative methods.

3.2 Hardware Implementation of Q-Learning

According to Spano, et. al., "Reinforcement Learning (RL) is a Machine Learning (ML) approach used to train an entity, called agent, to accomplish a certain task" [8]. Reinforcement learning consists of developing an "agent" who learns how to do things in an environment, through taking actions that give a response. Fundamentally, the Agent exists in an environment, and when it performs an action, it receives a reward. The most common reinforcement learning algorithm is Q-learning, where the data structure called a Q-Matrix is used to track the rewards of specific actions taken in specific states. The

Q-Matrix is of size S by A, where S represents the number of unique states, and A represents the number of actions that can be taken [8]. Training this agent consists of taking an action, seeing what the reward is, and then updating the Q matrix for that specific State-Action combination. By iterating until convergence, the Q-Learning algorithm will eventually converge into a result that can accomplish the desired task. One common operation done in the Q-learning algorithm is to take the maximum value of all potential rewards given a specific state. For extremely large action spaces, this is an expensive operation, and thus must be optimized. While it may not be realistic for typical general purpose hardware such as a CPU or GPU to perform this operation quickly, the proposed custom design uses a streamlined sequence of multipliers in order to make calculating the maximum value as quickly as possible. The general layout of the hardware components can be found in Fig. 3. The design uses a "Xilinx Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit featuring the XCZU7EV 2FFVC1156 FPGA" [8], with measurements taken with "Vivado 2019.1 EDA tool", and focused on optimizing various parameters in their custom design with existing basic blocks in the FPGA, such as "Look Up Tables (LUTs), Flip-Flops (FF), and Digital Signal Processing Slices (DSP)" [8]. Given a wide variety of parameters for Q-Matrix Size, unique states, and unique numbers of actions, they found that their custom implementation was significantly more power efficient than an existing design in previous literature [8]. While Q-learning is currently a less important topic of machine learning than CNNs, this paper shows that there is much promise in this avenue of research, should reinforcement learning become more important in the future.

3.3 Binarized Neural Networks

As neural networks increase dramatically in size, one to way to simplify and streamline behavior is to use binary values for weights, instead of the typical floating point values [2]. Typical neural networks use floating point values to track weights on a

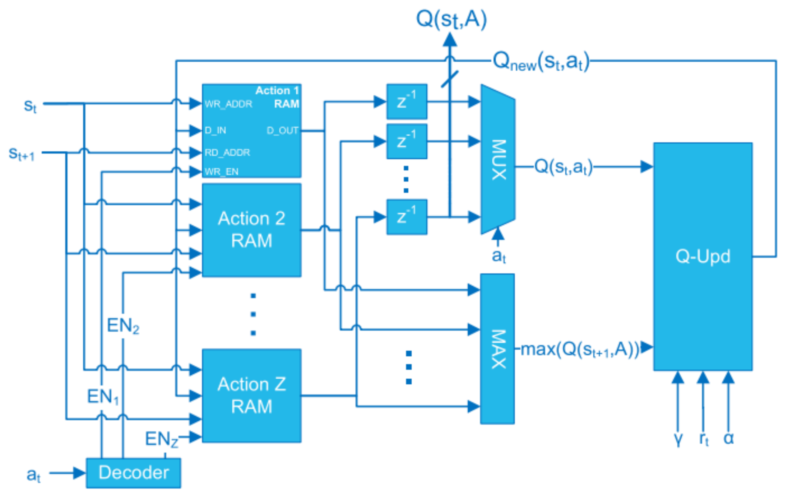


Fig. 3. Q-Learning Hardware Block Diagram [8] ©2019 IEEE.

per neuron basis, which offers a great degree of precision in the resulting classification. However, floating point operations are significantly more expensive in hardware than even a simple integer operation. However, for certain machine learning tasks, such as image classification, neural networks can be remarkably fault tolerant. By taking a conventional neural network and replacing weights with binary values representing +1 and -1, neural network hardware can be simplified, since this converts conventional addition and multiplication into logical operations of XOR and XNOR, respectively [2]. While neural networks must be trained first, before being used for classification, the paper focuses on the classification phase, as that is the typical use case for the average user. Training is generally considered a one time cost that can be amortized across many devices using the same trained weight values, and as such is a minor portion of all computational time [2]. The hardware must first convert input data from floating point to binary form, and then all subsequent operations are conducted purely on binary values. The general structure of the accelerator is found in Fig. 4.

Weights are read from memory and input into a simple multiply-add-accumulate block. This is done in parallel for a variety of layers, and the same hardware can be used both for

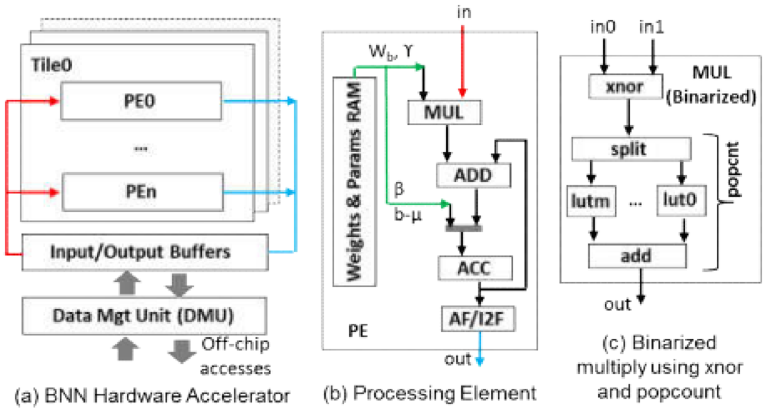


Fig. 4. BNN Accelerator Block Diagram [2] ©2016 IEEE.

simple weight calculations and also other operations such as activation functions and batch normalization. These blocks make up each individual Processing Elements (PEs), and the design can be parameterized for wider or narrower data with more or less PEs.

The design was tested on multiple platforms, with simulated code equivalents on CPU (Intel XEON E5-2699v3) and GPU (Nvidia Titan X), and actual hardware implementations on an Altera Aria 10 FPGA and a custom ASIC design using Intel technology [2]. The results displayed in Fig. 5 show that the CPU baseline performs the worst, with GPU about 5x faster than CPU, FPGA about 50x faster than the CPU, and ASIC 400x faster than the CPU. Binarization of the neural network also contributed significantly to the increased performance, as a nonbinarized version of the same network performed achieved only 20% the performance as the binarized model across all platforms.

3.4 Recurrent Neural Network Design

Recurrent Neural Networks (RNNs) are a type of neural network used to learn from data that consists of sequences where ordering matters. One common use case for RNNs is natural language processing, where word meaning and significance can change dramatically based on the order of appearance [3]. Gated Recurrent Units (GRUs) are a

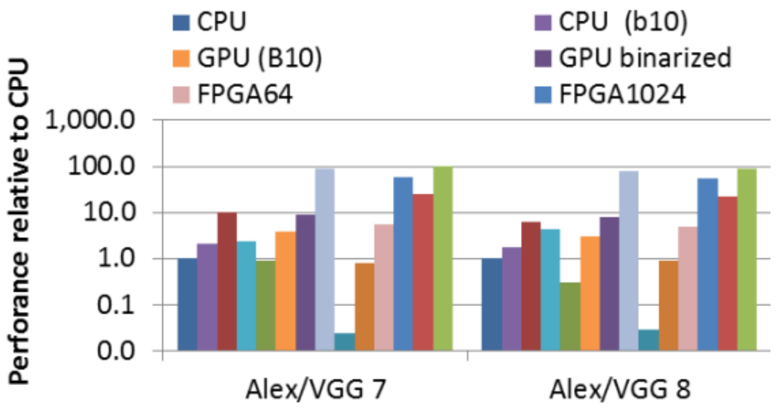


Fig. 5. BNN results relative to other platforms [2] ©2016 IEEE.

more complex variant of RNN that has increased performance in terms of learning ordering dependent patterns. GRUs consist of multiple gates that combine and selectively ignore past and present information in order to learn useful patterns. Past information is essentially fed back into RNNs continuously in a loop in order to maintain knowledge of previous results. Similar to the binarized neural network implementation, this design is primarily concerned with inferences, which is when information is fed into the neural network in an attempt to classify inputs. Because the pattern of inferences is mainly concerned with feeding new information forward, certain weight multiplications can be precalculated and stored in memory [3]. While the authors note that this results in a noticeable increase in memory usage (approximately 3x), the performance benefits of avoiding multiply-accumulate operations are significant enough to justify the memory usage.

The custom RNN design is divided into columns, shown in Fig. 6, where “each floating-point multiply-accumulate processes a row in a column block” [3]. These results are stored into on-chip memory. The same algorithm was implemented on an Intel Xeon CPU, Nvidia Titan GPU, both StratixV and Aria10 FPGAs, and ASIC. The results showed that generally, GPU performed about 4x better than CPU, and ASIC performed

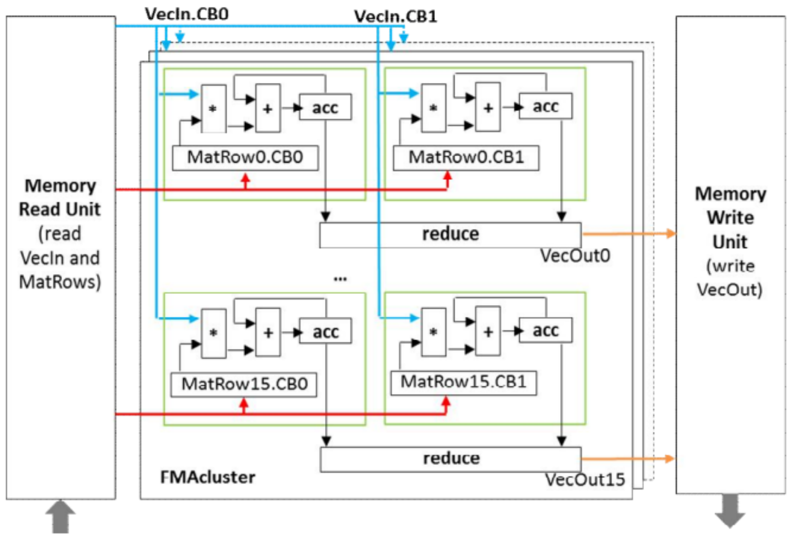


Fig. 6. RNN Accelerator Design [3] ©2016 IEEE.

about 7x better than FPGA, once again confirming that machine learning ASICs can achieve very high performance [3].

3.5 Google TPU Using Systolic Arrays

A wide variety of neural networks exist to perform several types of work, but in reality, the vast majority of neural network implementations fall under only a few categories. Google offers cloud services that provide access to machines that can perform neural network computations, and through their data collection researchers determined that 95% of neural network inference workloads consisted of either "Multilayer Perceptrons (MLP), Convolutional Neural Networks (CNN), or Recurrent Neural Networks (RNN)" [4]. With this in mind, they designed a PCIe coprocessor called a Tensor Processing Unit (TPU) that could perform NN calculations with much higher performance than the Nvidia K80 GPUs they were originally using. Focus was placed on designing the coprocessor such that it could run almost autonomously with as little input from a host CPU as possible, which means that the design consists of long instructions that perform many operations in a row. While this flies in the face of the trend towards simpler instructions in general purpose computing, it turns out that for more specialized uses such as machine learning ASICs, certain computer architecture concepts that were once deemed obsolete or impractical are actually quite useful. Specifically, “Ideas that didn’t fly for general-purpose computing may be ideal for domain-specific architectures” [4]. While the usage of decoupled-access/execute, which decouples memory accesses from actual execution, and CISC instructions (complex instruction set computers), which uses long instructions of varying length to direct operations, both were useful in TPU design, the most significant portion of the design comes from the use of a systolic array.

A systolic array is a large array consisting of multiply-add units, where vectors of data are fed into the array from two sides, and the data flows through the array, iteration by iteration, to eventually calculate a complete result [4]. The primary use of a systolic array is to calculate large matrix operations quickly and efficiently, relative to conventional computer architecture designs. While systolic arrays were originally just a historic curiosity from the 1980s, it turns out that they are very effective and performing neural network operations.

The matrix multiply unit that performs all of the mathematical calculations in the TPU is essentially one large systolic array. Feeding data into that are multiple memory buffers that fetch from memory data and weights and feed them into the systolic array. Activation functions, pooling, and normalization are done after the matrix multiply unit. All of this is controlled by the control unit, which receives instructions across a PCIe bus from the host CPU [4]. Compared to typical general-purpose processors, where a nontrivial portion of chip area consists of control logic, the Google TPU control logic only occupies 2% of the die [4]. This means that a larger portion of silicon is dedicated towards completing work, which leads to increased efficiency.

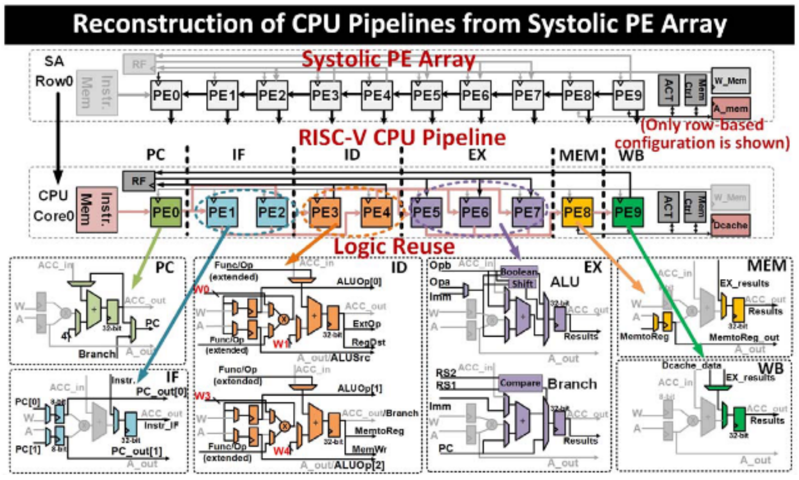
Performance using the Google TPU is significantly than a CPU baseline. Power efficiency is approximately 30x that of a baseline CPU, and performance is also about 30x faster than the CPU, along with 15x faster than the K80 GPU [4]. It is clear from this implementation that given the resources of a large technology company, machine learning ASICs are significantly more efficient than general purpose CPUs and GPUs.

3.6 RISC-V Designs using Systolic Arrays

RISC-V is a relatively new instruction set, though it is an area of active research. While the majority of machine learning accelerators covered so far are custom designs with custom instruction sets, there do exist designs based on the RISC-V ISA. According to Ju, Yuhao and Gu, Jie, conventional deep neural network (DNN) accelerators are designed with a conventional CPU core and a separate accelerator, with communication between the two systems managed through a direct memory access engine (DMA) [9]. The separation between these two components means that large amounts of latency exist when moving data from component to component. While many methods have been explored to mitigate these issues, Ju and Gu propose “a systolic neural CPU (SNCPU)” which merges the two together [9]. The architecture contains components that can be reconfigured to work as either a RISC-V CPU or a systolic convolutional neural network (CNN) accelerator, which leads to increased resource usage at a minimal overhead of about 10% [9].

The systolic array consists of a 10x10 array of 8 bit integer units, which are dynamically reconfigured to do appropriate functions in a standard RISC-V pipeline. The general design can be seen in Fig. 7. Considering that approximately 64-80% of the systolic array processing engines (PEs) can be reused for the RISC-V pipeline, this is a fairly interesting design.

A. Gonzalez et al. propose a design that integrates a variety of existing open source solutions, including a Berkeley Out-of-Order Machine (BOOM) along with a Gemmini systolic array DNN accelerator [10]. Their design contains a 16x16 systolic array of 8-bit integers, with promising results. On the other hand, V.N. Chander and K. Varghese



Reconstruction of CPU Pipelines from Systolic PE Array
Systolic PE Array
RISC-V CPU Pipeline
(Only row-based configuration is shown)
Logic Reuse
SA Row0
CPU Core0
PC
IF
ID
EX
MEM
WB
ALU
Branch

Fig. 7. RNN Accelerator Design [9] ©2022 IEEE.

designed a RISC-V vector processor that operates on 8 unit wide vectors of 32-bits integers [11]. While their results are also promising, it is interesting to note that the majority of existing research focuses on integer vector operations.

4 PROJECT DESIGN

4.1 High Level Overview

The custom design implemented in this paper consists of a simple RISC-V CPU implementing basic 32-bit integer instructions, with support for standard arithmetic, memory, and branch/jump instructions. Specific CPU behavior is designed to match descriptions provided by the official RISC-V specification edited by Andrew Waterman and Krste Asanovi´c [12]. The design is based on a standard 5 stage RISC pipeline, with Instruction Fetch, Decode, Execute, Memory, and Register Write-back stages. A general outline of the design is shown in Fig. 8 below.

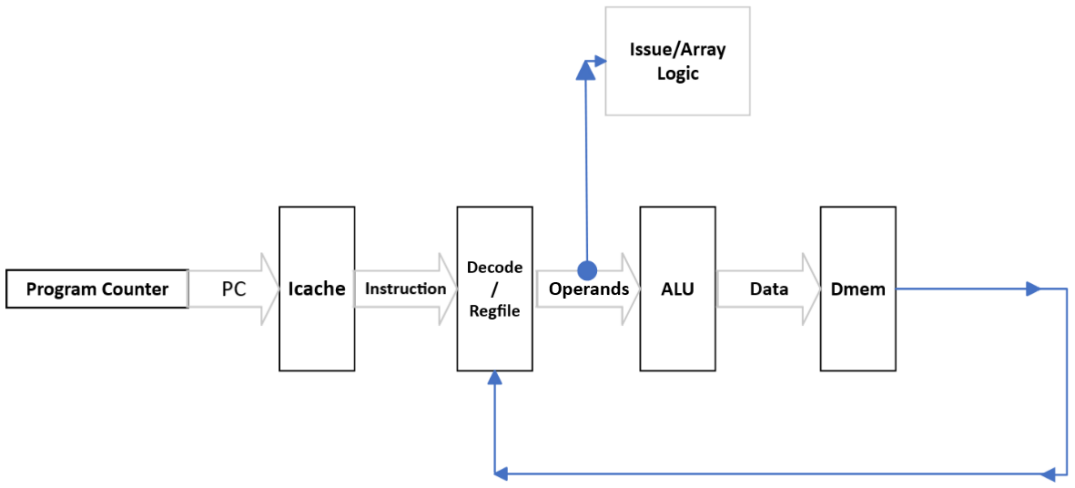


Fig. 8. High level block diagram of a RISC-V CPU.

The program counter tracks which particular instruction is being executed, and is used as in index into the instruction cache (Icache), where the relevant instruction is fetched. This instruction is deciphered in the Decode stage, where the appropriate signals are generated to control the rest of CPU execution, along with fetching the appropriate data from the register file. These control signals and register operands are passed further down to the arithmetic logic unit (ALU), where the appropriate calculations are made with the input data. If necessary, memory accesses are then handled through the data memory

(Dmem), and the result is written back to the register file when appropriate. If appropriate, the decode stage detects if the instruction is actually a special instruction be passed into the systolic array, and if so the instruction is instead routed to the issue logic of the systolic array, instead of the rest of the RISC-V pipeline.

4.2 Fixed Point Arithmetic

While typical CPUs, including the majority of existing work covered previously in this thesis, generally operate on integer numbers for simplicity, the custom design implemented here uses a 16-bit fixed-point number format instead, consisting of a sign bit and a 15 bit number. Instead of the typical two's complement number representation used in the vast majority of computing, this fixed point representation tracks magnitude only, with the sign bit determining whether the number is positive or negative instead. While this makes operations with negative numbers slightly less elegant, it does mean that multiplication is very straightforward, as there is no need to convert both operands to positive to avoid issues with multiplying negative numbers. The general breakdown of the 16-bit fixed-point format used is outlined in Fig. 9.

$$b_{sign}b_4b_3b_2b_1b_0.b_{-1}b_{-2}b_{-3}b_{-4}b_{-5}b_{-6}b_{-7}b_{-8}b_{-9}b_{-10}$$

$$b_4b_3b_2b_1b_0.b_{-1}b_{-2}b_{-3}b_{-4}b_{-5}b_{-6}b_{-7}b_{-8}b_{-9}b_{-10} * b_4b_3b_2b_1b_0.b_{-1}b_{-2}b_{-3}b_{-4}b_{-5}b_{-6}b_{-7}b_{-8}b_{-9}b_{-10}$$

$$\underbrace{b_9b_8b_7b_6b_5}_{\text{Overflow}}\underbrace{b_4b_3b_2b_1b_0.b_{-1}b_{-2}b_{-3}b_{-4}b_{-5}b_{-6}b_{-7}b_{-8}b_{-9}b_{-10}}_{\text{Result}}\underbrace{b_{-11}b_{-12}b_{-13}b_{-14}b_{-15}b_{-16}b_{-17}b_{-18}b_{-19}b_{-20}}_{\text{Discarded}}$$

Fig. 9. Fixed-Point bit arithmetic.

When using fixed point numbers, a point of contention is where to place the decimal point of the number, as that determines both the maximum possible range and the minimal precision supported by the number. As the intention of this project is to test results of this custom CPU against a deep neural network (DNN) data set, the Modified National Institute of Standards and Technology database (MNIST) was used to train a

DNN. The resulting weights and biases were extracted from the DNN, and inspected using python code to check the typical numerical range of neural network training. From an overview of results on the MNIST data set, it became clear that the vast majority of weights, biases, intermediate results, and final results typically fell within the range of -32 to 32. As a result, the fixed point was chosen to be after bit 10, meaning that in the 16 bit number, there is a sign bit, 5 bits to the left of the decimal place, and 10 bits to the right of the decimal place. This gives a maximum magnitude of approximately 32 (with all 15 bits 1), and a maximum precision of about 0.001 (1/1024).

Matrix multiplication is fairly simple, as two numbers can simply have their magnitudes multiplied together. The sign bit is a simple exclusive-or of the input signs, as two negatives cancel when multiplied, and any situation where only one number is negative means that the product must necessarily be negative. Given a 15 bit by 15 bit multiplication, the resulting 30 bit product can be broken down into an overflow of the first 5 bits, a 15 bit result, and the lower 10 bits are discarded as they fall under the minimum precision of 10 bits below the decimal point.

For matrix adds, logic is slightly more complicated as sign bits do not roll over in the way that they would for two's complement arithmetic. If both numbers are the same sign, then the magnitudes can be added and the sign of the result is the same. If the numbers have differing signs, then the magnitude of both numbers A and B must be compared, and the number with the larger magnitude has its sign bit propagated. Similarly, the difference between both numbers, A-B and B-A are both calculated, and the number with the larger magnitude determines which difference is chosen (if A is larger, then the result is A-B, vice versa for B and B-A).

4.3 Systolic Array

4.3.1 Systolic Cell

A systolic array is, as the name implies, an array of systolic cells. A single systolic cell is responsible for calculating a single multiply-accumulate within a computation. A systolic cell consists of two numerical inputs named A and B, two control inputs to start and stop computation, and a clock input. Inside the cell, there are registers to latch the inputs A, B, start, and stop, along with a multiplier to find the product A*B, and an adder to add this product to an accumulator register. There is also a result register, that stores completed results when they are finished. When the systolic cell sees the start signal go positive, it stores A*B directly into the accumulator without adding. Conversely, when the systolic cell detects the stop signal, then the result of the accumulator is stored in the result register. All data, including A, B, start, stop, and the result, is propagated out of the cell for use by other cells. The general structure of a systolic cell can be seen in Fig. 10.

4.3.2 Array of Cells

A systolic array is a group of 4 by 4 systolic cells, with the outputs of one cell daisy chained into the inputs of the next cell. Input vectors consist of four 16-bit fixed-point numbers, grouped into a 64 bit vector register. These vectors are shifted into the array cycle by cycle in a staggered format, with a start signal asserted at the beginning of a new calculation, and a stop signal asserted at the end. Result vectors come out after a delay of 8 cycles, which compensates for shifting an entire input vector in and then back out. Because of the design of the array, calculations can be pipelined so that multiple calculations are in flight at one time.

It is important to note that while one set of inputs (the left side of the matrix calculation, input A) is shifted in element by element, the second set of inputs (input B) are shifted in vertically in a staggered fashion. This matches the expected behavior of a typical matrix multiply, where the rows of the first matrix are multiplied against the

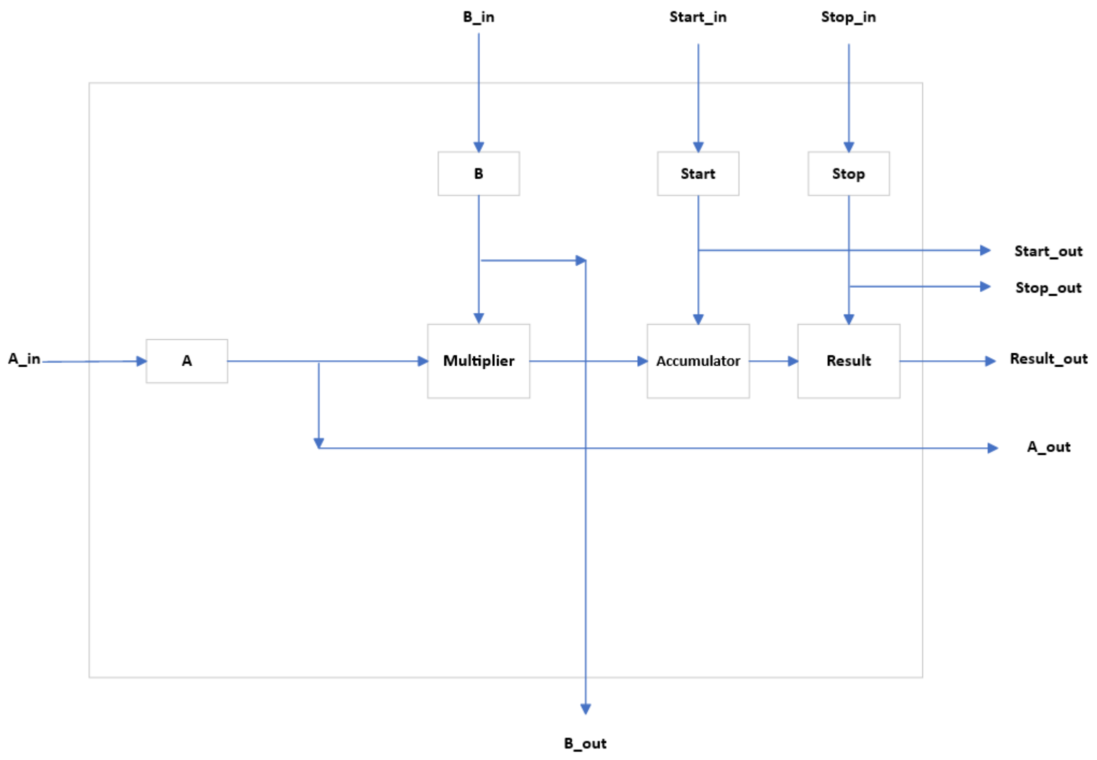


Fig. 10. Systolic Cell block diagram.

columns of the second matrix. As the vectors are shifted in cycle by cycle, each cell generates a sum of products that matches the result of a matrix multiplication, where each nth element of a row is multiplied with the nth element of a column, and then added together. The pipeline registers used to stagger input shifts, along with the array itself, can be seen in Fig. 11. An example calculation can be seen in Fig. 12.

4.4 Issue Logic

The systolic array unit is designed to handle a total of 5 unique instructions, which covers all necessary functions to do matrix multiply operations in a typical register-register architecture. The specific instructions supported are vector add, vector multiply, vector load, vector store, and vector relu, all of which are shown in Fig. 13.

The issue unit contains 128 64-bit registers, with each 64-bit register containing 4 16-bit fixed point numbers. The 128 registers are organized into 4 registers batches,

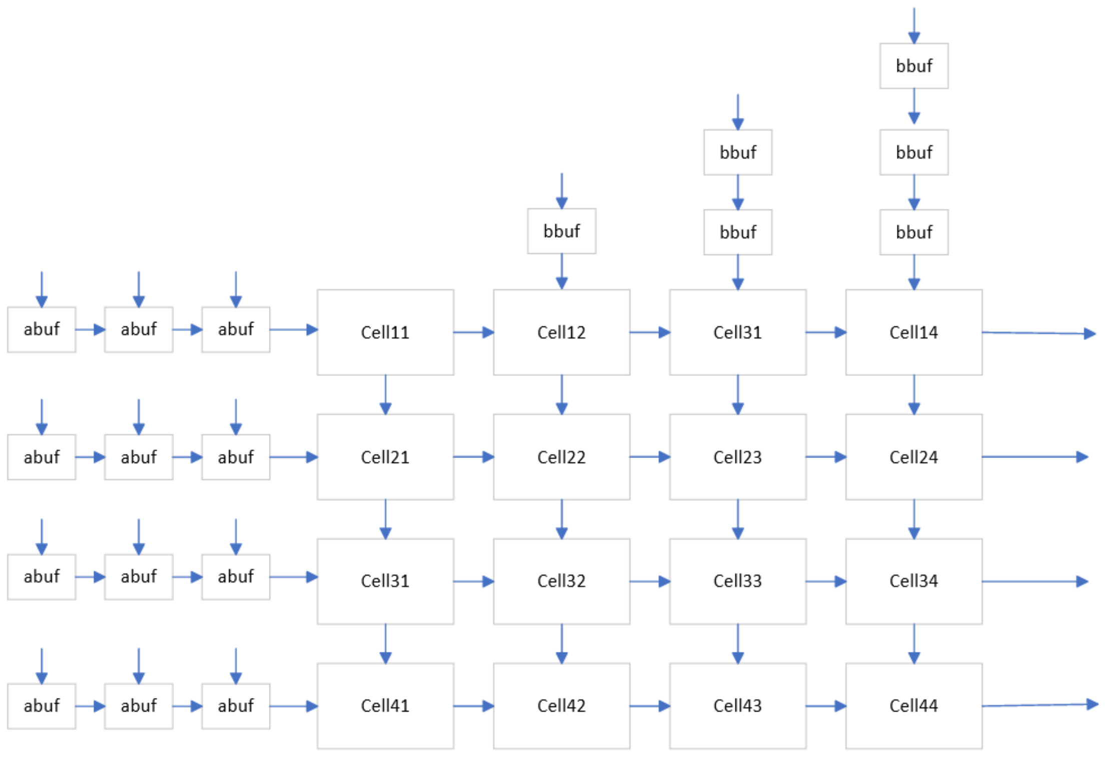
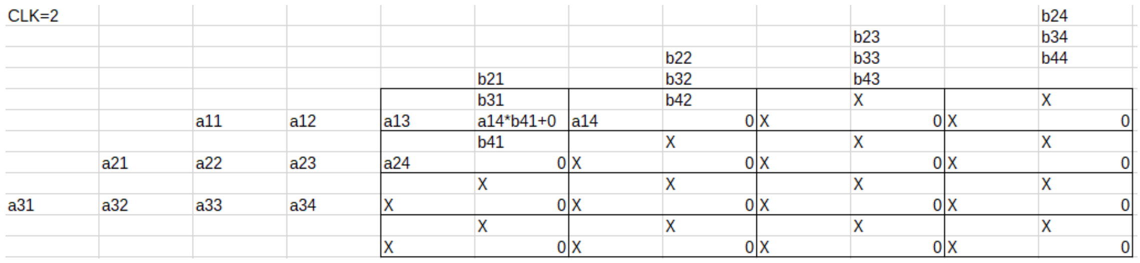


Fig. 11. Systolic Array block diagram.



CLK=2											b24
									b23		b34
							b22		b33		b44
					b21		b32		b43		
					b31		b42		X		X
		a11	a12	a13	a14*b41+0	a14	0	X	0	X	0
					b41		X		X		X
	a21	a22	a23	a24	0	X	0	X	0	X	0
					X		X		X		X
a31	a32	a33	a34	X	0	X	0	X	0	X	0
					X		X		X		X
				X	0	X	0	X	0	X	0

Fig. 12. Matrix Multiplication in flight.

```
Systolic Array Instructions
         31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
opcode  |        funct7       |      rs2       |      rs1       | funct3 |      rd      |      opcode        |
VFIXADD | 0  0  0  0  0  0  0|      vs2       |      vs1       | 0  0  0|      vd      | 1  1  1  1  0  0  0|
VFIXMULT| 0  0  0  0  0  0  0|      vs2       |      vs1       | 0  0  0|      vd      | 1  1  1  1  0  0  1|

VFIXLOAD|          stride[11:0]              |      base      | 0  0  0|      vd      | 1  1  1  1  0  1  0|
VFIXSTOR|          stride[11:0]              |      base      | 0  0  0|      vs1     | 1  1  1  1  0  1  1|

VFIXRELU|            xxxxxxxxx               |      vs1       | 0  0  0|      vd      | 1  1  1  1  1  0  0|
```

Fig. 13. Systolic Array custom vector instructions.

meaning that the register file contains 32 groups of 4 registers each. All vector instructions operate on 4 registers at a time, with each register handled on consecutive clock cycles.

The Instruction buffer takes input instructions from the top level RISC-V CPU, and holds them until the array unit is ready to accept new instructions. If the buffer is full, it will indicate to the RISC-V CPU to stall until array instructions finish. When the array is available to execute on new instructions, the instruction buffer is released to the control unit, which contains a state machine that tracks instruction execution. The state machine determines which registers are being operated on, and which execution unit (Adder, Systolic Array, Relu, or Data cache) needs to do work. The adder is simply a vector adder that adds 4 separate numbers and out puts the result, the systolic array is the multiply unit, the Relu unit applies the Relu activation function on the register inputs, and the Data cache (Dcache) deals with memory accesses of both reads and writes. Relu is particularly simple to implement in hardware for sign-bit numbers, as a simple check of the sign bit determines whether the magnitude is preserved (for positive numbers) or zeroed out (for negative numbers). Results are calculated in 4 consecutive clock cycles, with reads and writes overlapping during the 4 clock cycles where a grouping of 4 vectors is executed. The general dataflow pattern is shown in Fig. 14.

4.5 Memory Accesses

Matrices are typically stored in memory as large contiguous arrays of numbers, with each number one after the other. For a matrix of arbitrary dimension M by N of M rows and N columns, loading the correct sub-matrix is necessary when breaking down a matrix multiply into appropriately sized sub-products. Vector memory instructions take a base address from the host RISC-V CPU, along with a stride value indicated by the immediate field of the instruction. Memory accesses will first operate on the base address, followed by the base address+stride, then +stride*2, then +stride*3. This means that the stride value should be varied to access the desired 4x4 sub matrix, where the stride value is

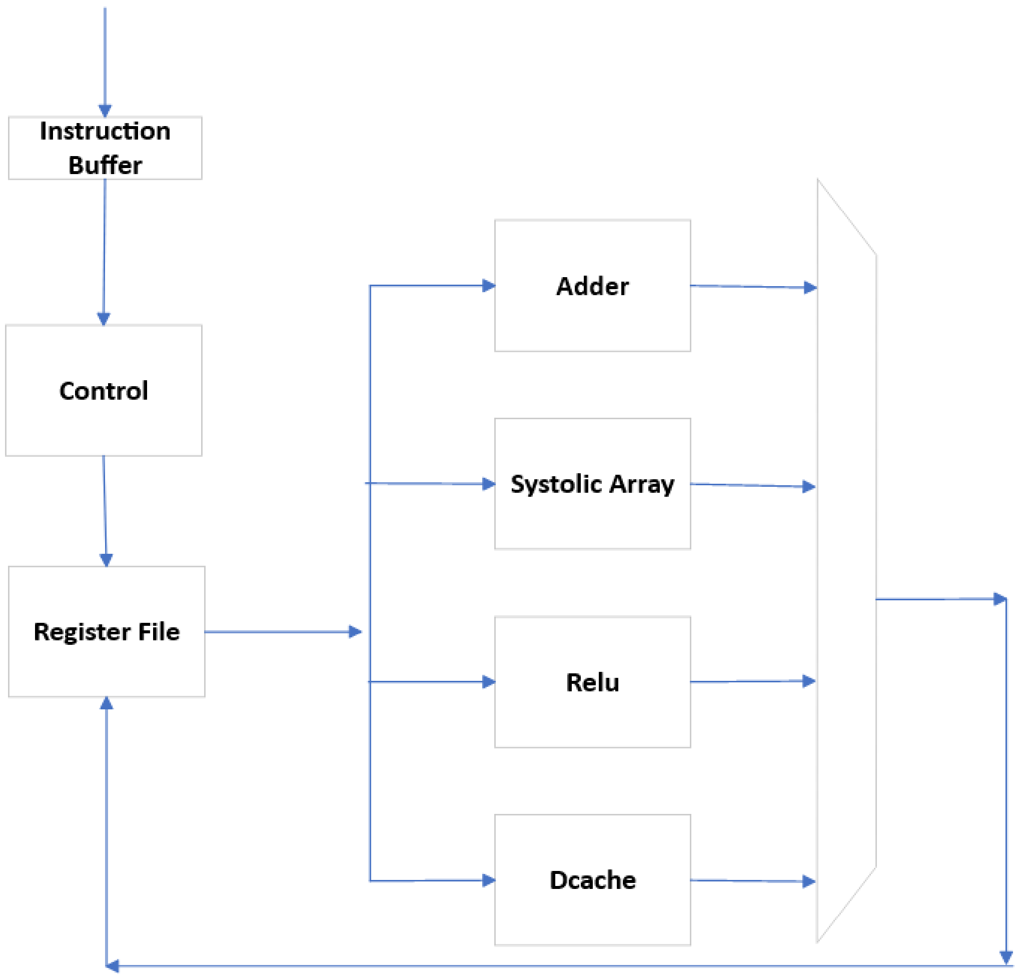


Fig. 14. Systolic Array Issue block diagram.

essentially the width N divided by 4. An example of the stride pattern applied to a 64 element array can be found in Fig. 15.

4.6 TensorFlow Neural Network

The MNIST data set consists of 60000 handwritten number images in a training set, along with 10000 more number images as a test set. Each image is a 28x28 black and white image, where each pixel is represented by a single byte. The images are converted into a single 1x784 array of numbers, which are then fed into a simple neural network containing two fully connected layers, the first of which has 16 nodes with relu activiation

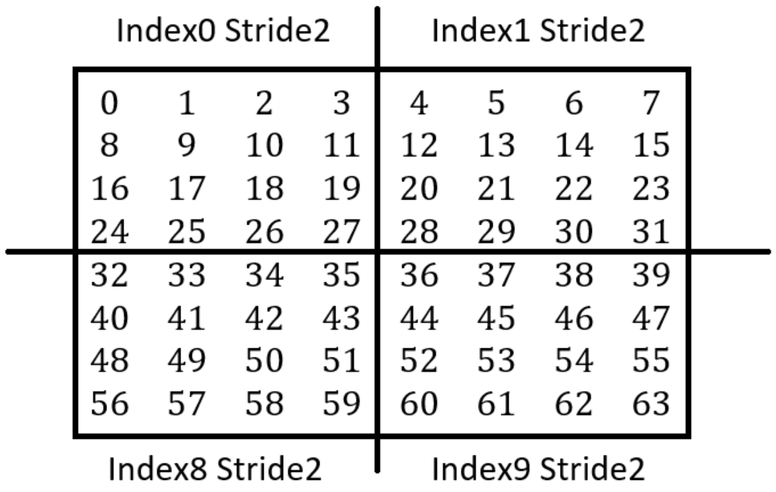
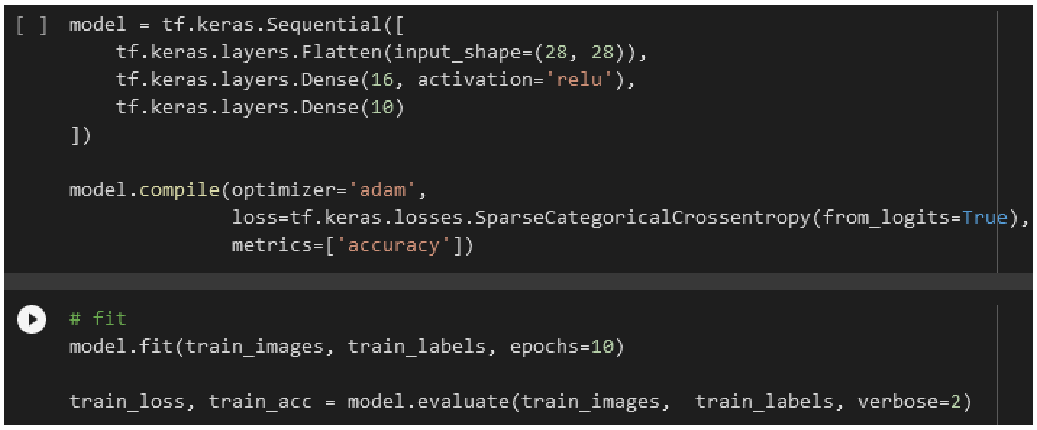


Fig. 15. Memory Access stride organization.

function, and the second output classification layer having 10 nodes. The general structure of the TensorFlow code is shown in figure Fig. 16.



```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(10)
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
# fit
model.fit(train_images, train_labels, epochs=10)

train_loss, train_acc = model.evaluate(train_images,  train_labels, verbose=2)
```

Fig. 16. Simple MNIST classifier in TensorFlow.

The neural network was trained on Google Tensorflow using adam optimizer and SparseCategoricalCrossentropy loss, which is used for classification scenarios such as distinguishing between handwritten numbers. The resulting trained neural network could then have the weights and biases extracted. The first layer takes an input of 1x784, and has an output of 16x1. This means that the weight matrix for the first layer has dimensions of 784x16. The bias matrix of the first layer is simply 16x1, followed by a Relu activation function. The next layer is an output layer of 10 nodes, meaning that the 1x16 input must generate an output of 10 numbers. The weight array of the second layer is 16x10, followed by a bias array of 10x1. These weights and biases are padded with zeros so that the height and width dimensions are a multiple of 4, then converted into 16 bit fixed-point numbers and then packed into memory. The approximate structure of the neural network can be seen in Fig. 17, though the first layer is compressed to only 28 nodes for readability.

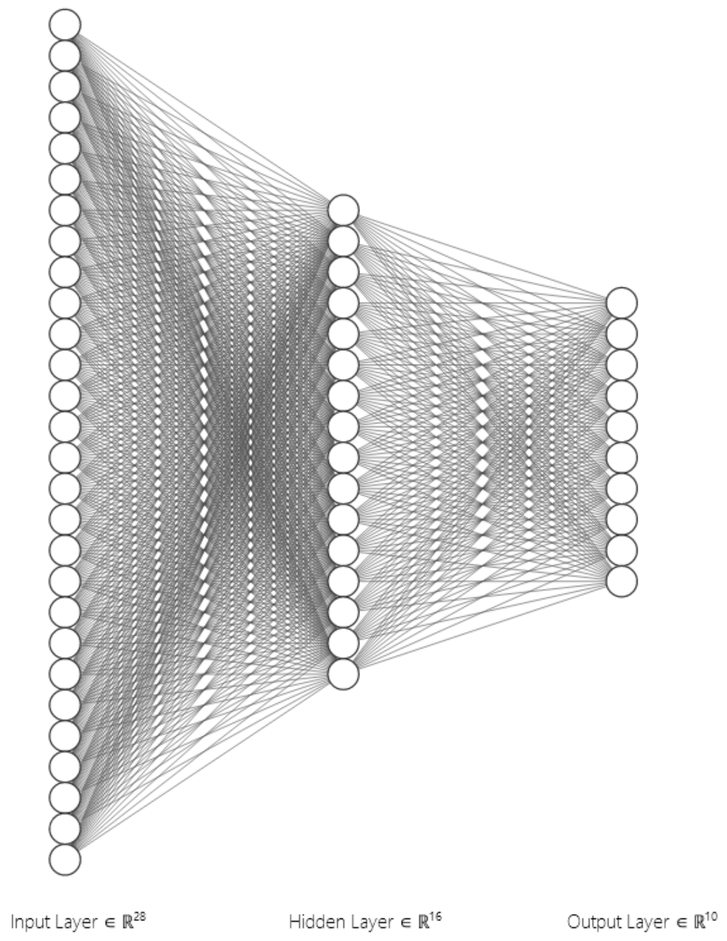
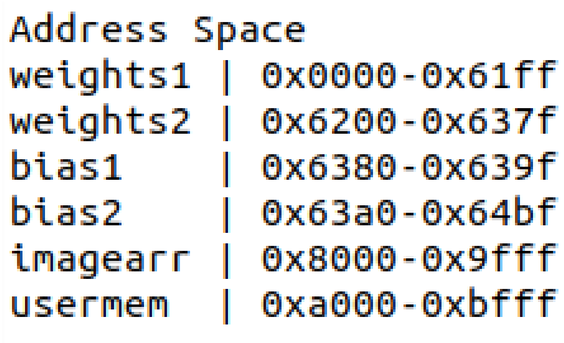

Input Layer $\in \mathbb{R}^{28}$
Hidden Layer $\in \mathbb{R}^{16}$
Output Layer $\in \mathbb{R}^{10}$

Fig. 17. Neural Network Diagram, notice that the input is only 28 instead of 784 for scale.

4.7 Memory and Data Organization

The neural network parameters consist of the first 784x16 weight vector followed by the second padded 16x12 weight vector, and two bias vectors that are reshaped into 4x4 vectors each. These vectors are implemented as lookup tables for simplicity, as storing them in a memory and then designing a cache around it was judged to be outside of the scope of the project. Similarly, source images of 1x784 bytes could also be converted into fixed point values and packed into a lookup table. User memory could be implemented as a register array is both readable and writable. For the purposes of this project, these memory regions are explicitly delineated with memory ranges, as seen in Fig. 18.

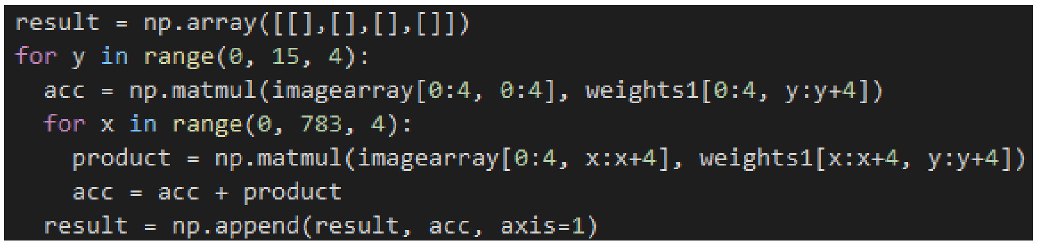


```
Address Space
weights1 | 0x0000-0x61ff
weights2 | 0x6200-0x637f
bias1    | 0x6380-0x639f
bias2    | 0x63a0-0x64bf
imagearr | 0x8000-0x9fff
usermem  | 0xa000-0xbfff
```

Fig. 18. Table of data memory ranges.

4.8 Matrix Multiplication Assembly Code

The matrix multiplication algorithm was implemented manually in custom RISC-V assembly, as modifying a compiler for these purposes was deemed outside of the scope of the project. Using known memory addresses for weights and image data previously discussed, along with available branch and arithmetic instructions, approximately 100 lines of assembly code were written to break down the multiple matrix multiplications into loops of 4x4 sub-matrix products followed by adds. The general structure of the code can be seen in Fig. 19, as a python simulation.



```
result = np.array([[],[],[],[]])
for y in range(0, 15, 4):
  acc = np.matmul(imagearray[0:4, 0:4], weights1[0:4, y:y+4])
  for x in range(0, 783, 4):
    product = np.matmul(imagearray[0:4, x:x+4], weights1[x:x+4, y:y+4])
    acc = acc + product
  result = np.append(result, acc, axis=1)
```

Fig. 19. Python code simulating matrix multiplication.

5 RESULTS AND CONCLUSIONS

5.1 Results

The results of inferences using the custom matrix multiply unit are noticeably different from inferences conducting using TensorFlow, however the margin of error for a fault tolerant process like image recognition means that results are still reasonably accurate compared to desired results.

The resulting vector from an example inference is shown in Fig. 20, and is 8556 a13e 05a5 a4fd 1063 8e49 9470 0979 8c53 0690, which corresponds to decimal values of -1.33, -8.31, 1.41, -9.24, 4.10, -3.57, -5.11, 2.37, -3.08, 1.64. While the numbers vary slightly from the more precise floating point calculation, numbers are generally of the same magnitude, and critically, the largest number of the systolic array result matches the expected result from the TensorFlow inference in Fig. 21, corresponding to number 4.

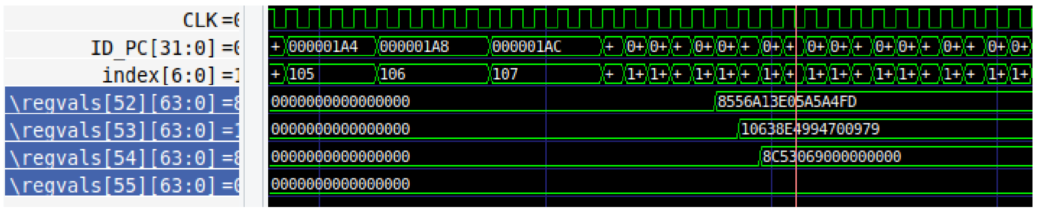
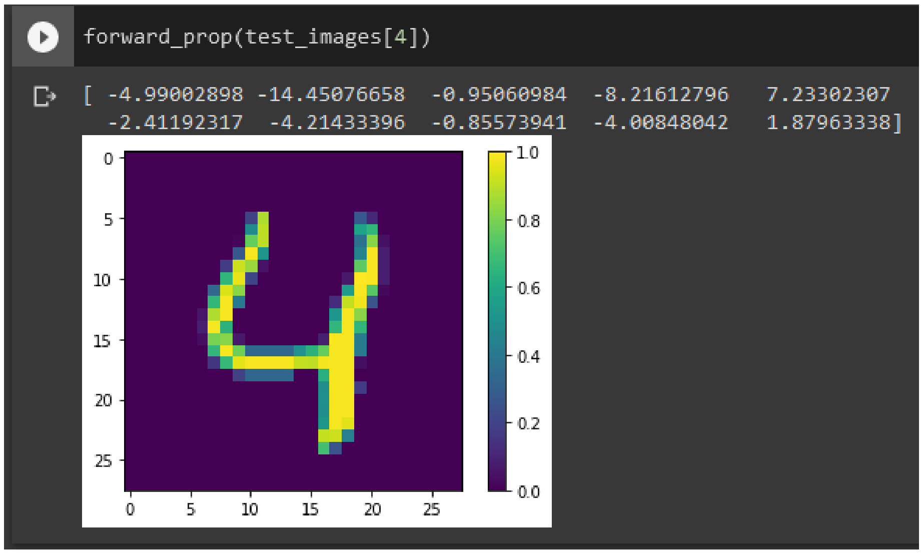


Fig. 20. Waveform of Systolic Array calculation.

An approximate analysis of clock cycles required to execute this on the systolic array is approximately 5000 clock cycles amortized over parallel executions, which could be increased significantly with even more systolic arrays added. However, a naive linear implementation using a standard RISC-V pipeline without a systolic array would take approximately 784x16x3=37600 clock cycles, leading to an approximate speedup of about 9.4x. While this is assuming ideal memory behavior for both cases, you can generally assume that since both scenarios are operating on the same data, the number of memory accesses across both scenarios would be comparable.



```
forward_prop(test_images[4])
```

```
[ -4.99002898 -14.45076658  -0.95060984  -8.21612796   7.23302307
  -2.41192317  -4.21433396  -0.85573941  -4.00848042   1.87963338]
```

Fig. 21. TensorFlow results and source image.

5.2 Conclusion

Machine learning computations in general, and more specifically neural network inferences, are easily reducible to large matrix multiplication operations. One way to to effectively parallelize these matrix computations is through the use of a systolic array of many individual cells that conduct multiply-add operations. Though most research typically focuses on integer operations for simplicity and efficiency, fixed-point arithmetic allows a level of additional decimal precision without too much additional logic complexity. This custom RISC-V systolic array design is able to conduct neural network inferences with reasonable accuracy when compared with TensorFlow inferences using floating point, while also being noticeably faster than a naive design using a simple RISC-V arithmetic unit. Though this design is still fairly simple in the grand scheme of things, hopefully this thesis demonstrates that there may be potential in using fixed point arithmetic as a compromise between typical integer and the more expensive floating point calculations.

5.3 Future Work

While this thesis successfully demonstrated that a RISC-V CPU with a fixed-point systolic array can conduct neural network inference with reasonable accuracy, a great deal of simplifying assumptions were made that can be elaborated on for future work. Increasing the parallelism of the design with multiple systolic arrays operating in parallel can be done fairly trivially, with a per-array memory offset setting to allow the multiple arrays to work on different data in parallel. This would require implementing some basic synchronization primitives, but would result in much larger increases in speed for parallel tasks. Furthermore, the current design assumes idealized memory behavior in the form of lookup tables for all weights and input data. A more realistic design would need to implement a complex caching system that could load all manner of weights, biases, and input data in parallel while maintaining coherence. While optimizing cache behavior could justify a thesis on its own, adding and optimizing a cache would be a natural extension of the existing work. Furthermore, as more functionality is added to this custom CPU, writing assembly programs by hand would become more and more unsustainable. It would be practical to write a custom compiler to handle generating machine code for this systolic array accelerator as well, though that was judged to be outside of the focus of the project for now. All in all, while the current design is promising, many extensions can naturally be made to add functionality and increase performance.

Literature Cited

[1] D. Patterson, “50 years of computer architecture: From the mainframe cpu to the domain-specific tpu and the open risc-v instruction set,” in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pp. 27–31, 2018.

[2] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, “Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic,” in *2016 International Conference on Field-Programmable Technology (FPT)*, pp. 77–84, 2016.

[3] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, “Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic,” in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, 2016.

[4] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017. testnote.

[5] S. Shim. (2021). Lecture 5: CNN Part 2 [PowerPoint slides].

[6] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, “Ai accelerator survey and trends,” in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–9, 2021.

[7] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, 2014.

[8] S. Spano, G. C. Cardarilli, L. Di Nunzio, R. Fazzolari, D. Giardino, M. Matta, A. Nannarelli, and M. Re, “An efficient hardware implementation of reinforcement learning: The q-learning algorithm,” *IEEE Access*, vol. 7, pp. 186340–186351, 2019.

[9] Y. Ju and J. Gu, “A 65nm systolic neural cpu processor for combined deep learning and general-purpose computing with 95% pe utilization, high data locality and enhanced end-to-end performance,” in *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 65, pp. 1–3, 2022.

[10] A. Gonzalez, J. Zhao, B. Korpan, H. Genc, C. Schmidt, J. Wright, A. Biswas, A. Amid, F. Sheikh, A. Sorokin, *et al.*, “A 16mm 2 106.1 gops/w heterogeneous risc-v multi-core multi-accelerator soc in low-power 22nm finfet,” in *ESSCIRC 2021-IEEE 47th European Solid State Circuits Conference (ESSCIRC)*, pp. 259–262, IEEE, 2021.

[11] V. N. Chander and K. Varghese, “A soft risc-v vector processor for edge-ai,” in *2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID)*, pp. 263–268, 2022.

[12] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*, December 2019. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.