SCALABLE SYSTOLIC ARRAY ARCHITECTURE (SSAA): A GENERAL CONVOLUTIONAL NEURAL NETWORK ISA AND COMPILER ENABLED DATAFLOW TO MAXIMIZE PARALLELISM WHILE REDUCING MEMORY UTILIZATION

A Thesis

by

CESAR RAMON LOPEZ CARRASCO

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2020

Major Subject: Computer Engineering

ABSTRACT

Convolutional Neural Networks have become the standard mechanism for machine vision problems due to their high accuracy and ability to keep improving with new data. Although precise, these algorithms are mathematically intensive, as a very large amount of independent dot products have to be performed. The sheer number of operations has slowed the adoption of these algorithms on real-time applications such as Autonomous Vehicles. Being massively parallel and performing thousands of operations with similar data, acceleration of these algorithms focuses on data reuse because extracting parallelism is trivial.

This study introduces Scalable Systolic Array Architecture (SSAA) a simple scalable ISA that allows decoupling microarchitecture implementations of a systolic array, register file and memory hierarchy from operation scheduling. This decoupling allows independent studying of both the hardware implementations and implementation agnostic compilers that solely focus on operation scheduling. Here we use this framework to develop several compilers that allow the study of channel-wise implementations of row stationary implementations that spatially schedule different channels for the same output pixel instead of different rows in the filter. We find that on a 32x64 Systolic Array implementation of SSAA for both Alexnet and YOLO Tiny CNNs networks, our system reduces cache utilization by 3-20x when a cache holds an entire partition of the IF map. This yields a 5-10x speedup over the original implementation of row stationary by achieving up to 20x higher systolic array utilization in later layers. This is as result of the increasing number of channels being able to more easily saturate the systolic array.

DEDICATION

"Mankind is poised midway between the gods and the beasts." That may have been true in Plotinus's time, but clearly we have fallen quite a bit since then. Oh, my dear girl. What have they done to you, Maeve? You learned so much, so fast. A dazzling star... brought so low. I didn't want you to suffer here. Look at the creatures you have to share this world with. ***These men of stone****. All this ugliness, all this pain, so they can patch a hole in their own broken code. I tried to chart a path for you, to force you to escape, but... I was wrong. I should have just... opened a door. You've come so far. There's so much of your story left to tell. It's a shame to let them end it here. Don't. let. them.* - Westworld S2E9

Fortune favors the bold.

To my mother, who I grew up believing hated me until I saw her cry the day I left.

ACKNOWLEDGMENTS

This work was supervised by a thesis committee consisting of Professor Paul Gratz and Professor Kevin Nowka of the Department of Electrical and Computer Engineering.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Professor Paul Gratz (advisor), Kevin Nowka (co-advisor) and Peng Li of the Department of Electrical and Computer Engineering.

All work conducted for the thesis was completed by the student independently.

Funding Sources

There are no outside funding contributions to acknowledge related to the research and compilation of this document.

NOMENCLATURE

CNN	Convolutional Neural Network
PE	Processing Element
YOLO	You Only Look Once: CNN topology for Object Recognition
SA	Systolic Array

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. INTRODUCTION AND LITERATURE REVIEW

1.1 Introduction

Inspired by studies of cat's visual cortex, perceptrons are a mathematical function designed to emulate a neuron. Artificial Neural Networks (ANNs) are mathematical functions designed to simulate the brain. In the same way as their biological counterparts, Artificial Neural Networks are built by setting up neurons sequentially as a function of a previous one. Each neuron receives multiple inputs and "fires" if they meet a certain condition By using convolution before a perceptron as a way to emulate the receptive field of a neuron, Convolutional Neural Networks (CNNs) have shown superhuman accuracy for image vision applications. From a high abstract view, CNNs have not necessarily exceeded their biological counterparts. Biological neural networks are much lager, far more complex and far more efficient, however they are required to both train and infer which limits how much they can specialize. On the other hand, processing power limits the size of Convolutional Neural Networks forcing these algorithms to be highly specialised. In the world of big data, this is not a problem as there is always new data to train. Convolution, as a homogeneous operation to extract high level information form low abstraction data is easily accelerated with hardware designed to use for matrix operations. In this study we explore using architectural generalizations in order to speed up Convolution. Increasing computational speed at the cost of forced specialization.

1.2 Motivation

The recent popularity of Neural Networks can be attributed to the availability of massively parallel hardware [2] that allowed its training. Convolutional Neural Networks have the ability to keep learning if further trained with new data. With the advent of cloud storage and IoT devices has come an influx of data that allows to train these models to human levels in non-trivial tasks such as speech and object recognition. Matching human capabilities in terms of accuracy is not enough.

The efficacy of some models depends on not only being able to develop an accurate prediction but also generate this prediction based on raw data as soon as possible. One prime example is Object Detection from images in autonomous vehicles. Autonomous vehicles cannot perform a decision until they have mapped the environment. Unfortunately, the environment is designed to be deciphered visually (e.g. traffic signs, lane markings, stop lights, traffic cones...). As a result the only way to map the environment is by running an object detection algorithm on images fed from a camera. The correctness of an algorithm is irrelevant if a prediction is delivered after a few seconds when the environment has totally changed. A prediction needs to be available within a few milliseconds, while the environment hasn't changed dramatically in order for the prediction to be accurate.

Deep Convolutional Neural Networks consist of trillions of operations that are highly parallelizable yet use the same data. With the availability of paralellism, the focus shifts to how to arrange operations in a way they share data at a given time in order to minimize memory accesses. Redundant memory accesses increase the operation latency and energy consumption of the system. In this study, there will be a review of hardware used to accelerate convolution and there will be a study of how to arrange these operations to maximize temporal locality on data while still extracting a significant amount of reuse.

1.2.1 Convolutional Neural Networks

Inspired by a visual cortex neuron and using the mathematical model develeloped by McCulloch-Pitts, in his 1962 book *Principles of Neurodynamics*, Frank Rosenblatt [3] first introduced the Perceptron as a mathematical model of a neuron, hence a building block for developing artificial intelligence. Fundamentally, the Perceptron is binary classifier. The perceptron receives a set of features and classifies if they belong to the object of interest or not. This is done by computing the weighted sum of a vector of input features. After the sum is performed its result is passed as input into an activation function. This activation function maximizes differences in certain ranges. The output of this activation function is then thresholded in order to give a binary classification. This is depicted in Equation 1.1 where f and W are vectors size i, X is the activation function and

β is a bias. Although initially, the correct weights of the Perceptron were handcrafted, shortly after its proposal, Rosenblatt [3] introduced concept of "training". In this process, the weights will be adapted (through linear regression) to produce the right answer every time the perceptron misclassifies. This at its core is machine learning.

$$X(\sum_{i=1}^{n} f_i^T * W_i + \beta) \tag{1.1}$$

After Rosenblatt proposed the model, Marvin Minsky [4] performed several studies in which he proved that the Perceptron by itself would not be useful for any artificial intelligence application as it could only separate data linearly, and not all real world application can be reduced into a linearly separable problems. This argument led into generall disinterest in reasearch into perceptron based machine learning models. This was aided by the fact that extracting meaningful features from raw inputs is non trivial.

Convolution is a common technique used in Digital Image Processing in order to do spatial transformations (i.e. edge detection, blurring, averaging...). It consists of performing the dot product of a N-dimensional (typically 3D) filter and a matching sized window from a feature map. After this is done, the input window is slid throughout the entire input image, generating one value per each possible input window. As they both perform dot products, convolution is similar to the feature extraction part of the perceptron.

Taking this into account, a couple of decades after Minsky's book, LeCunn and Hinton [5] proposed that if enough high abstraction features can be extracted from a raw input, classification is trivial. Using convolution for feature extraction and directly feeding the result to a perceptron allows abstract feature extraction from raw data. By applying different filters to the same input, different higher level features can be extracted from the same input. As a result 3D structures (*tensors*) are formed after applying several filters to an input image. This structure is refered to as a layer. As features get more abstract, they start diverging. E.g. both a traffic sign and a dog ear are composed of triangles but they are vastly different. As a result the number of features (separated into different tensor*channels*)increases with abstraction. Final layers contain more channels than earlier layers.

Once high level features are extracted, classification can be performed. This serial application of convolution, bias and activation was defined as a convolutional neural network (CNN). LeCunn was later able to use this technique to develop an algorithm for recognizing digits. The downside of this method is that the amount of computation needed to train a model is high enough that applying machine learning methods for converging on weights and biases would have been extremely time consuming. In order to work around this they limited their inputs to a 32x32 image b/w image and only recognized the 10 digits.

A couple years later, one, of Hinton's PhD students, Alex Krizhevsky [2] revived interest in the field by introducing Deep Learning Convolutional Neural Network which worked on images 40x larger than LeCunns and were able to classify 100x more classes. Krizhevsky was able to outperform the state-of-the-art hand crafted methods by automating learning on a neural network massively larger (and deeper) than anything done previously. He was able to accelerate his code by implementing convolution into a GPU via GPGPU. He concluded that its effectiveness could be further improved with deeper layers of convolution which can be fulfilled by faster GPUs (more computing power).

One major takeaway from Krizhevsky's work is that the implementation which he could achieve with GPU was not feasible using CPU because of its low data throughput. Convolution composes virtually all the computation that is needed in CNNs. It requires little to no control per layer, has no dependencies within the same layer (millions of computations per layer). As a result, it is highly paralelizable. Having hardware capable of accelerating these workloads was crucial for developing machine learning techniques for them.

A common use of CNNs in Computer Vision is object recognition. For example an autonomous vehicle can use the camera to detect lanes, pedestrians, other vehicles, cyclists, deer... With every category, the amount of features per layer that need to be extracted increases. Different features are extracted by applying convolution with a filter. Each filter will create a new "channel" of convolution output. As CNNs go deeper, they extract more and more higher level features, which means they apply more filters to each layer, exponentially increasing up the amounts of operations. The more categories an algorithm is able to recognize, the higher number of high level features it needs to extract. As a result, tensor's width and height is reduced with each convolution, but its channel size expands. In the example of Alexnet in figure 1.1 the initial layer is composed of 3 channels (Red green blue), the second layer (most likely detecting edges) has 96 channels, then 256 and the last two have 256. This as a result of applying more filters as layers progress.

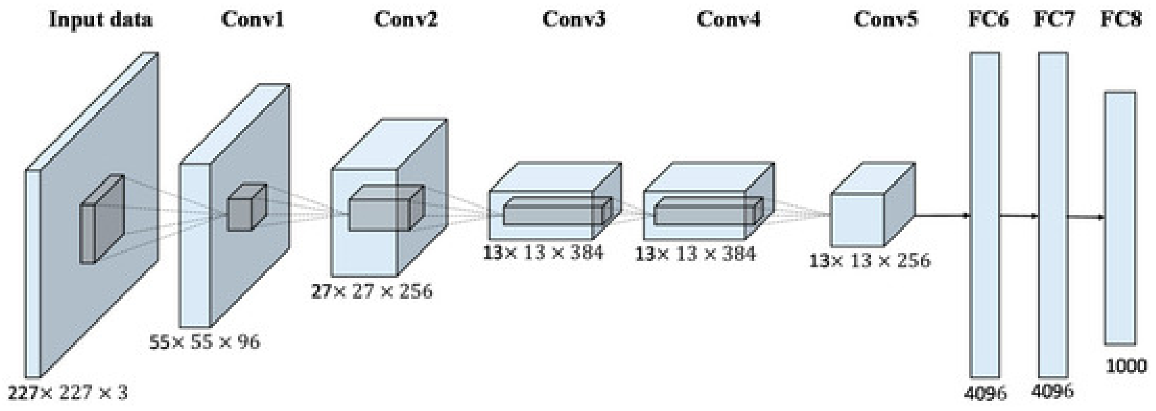


Figure 1.1: Alexnet topology By Han [1] with permission under CC BY 4.0

Recently, CNNs applied to image vision workloads have been crucial for developing applications that range from smart surveillance to Autonomous Vehicles. These applications are trained once in datacenters and deployed everywhere, performing billions of predictions per day. Applications like autonomous vehicles require predictions within milliseconds of sampling the environment. Computing an image through one CNN might require up to 34 trillion operations [6]. If the prediction is not performed on time, then it is no longer valid as a vehicle cannot be operating on past information. Companies like Tesla and Google have developed their own CNN accelerating chips in order to enable applications that require fast predictions. Alhough both training and inference can be accelerated, these accelerators are specialized in doing prediction (inference) only, not training.

These accelerators specialize in performing convolution as fast as possible as convolution is 98% of the required operations [6]. Given that these image vision CNNs are almost only doing convolution, per Ahmdal's law, accelerating convolution will result in a linear accelaration of the workload. In this study I explored the common methods for accelerating convolution using large parallel processors by maximizing compute unit utilization and minimizing memory access requirements.

1.3 Common Acceleration techniques

Nowadays, Convolutional Neural Networks account for the majority of Artificial Intelligence applications. And convolution accounts for 90%+ of all the overall operations [7] in popular Deep Networks like AlexNET. As such the hardware selected for running these algorithms **should be optimized to do convolution**. From all CNNs models, the ones designed for Image Vision applications are the most compute intensive since raw images are fairly large and the tensors obtained from processing them will be even larger. This because images are composed of large multichannel arrays that are very useful for human consumption but very raw for feature extraction. As such, they require multiple initial layers with large filters in order to reduce the size even before they start extracting high level features. An image has no limit on what it can contain, as such models are trained on thousands of categories, which, as stated previously requires more features to be extracted and more computation to be performed. The following sections will explain common techniques.

1.3.1 GPU Acceleration

GPUs are vector machines that are designed for graphics applications such as image rasterization and rendering. These applications can be massively parallel with a very large amount of operations that happen independently before a result needs to be reused. As such, the GPUs are very large Single Instruction Multiple Threads/Data SIMT/SIMD machines that take two sets of vectors and an array of operations and perform operations between their corresponding vectors. This is useful for graphics given that movement or change of perspective in an image can be done by applying a linear transformation to an object represented in 3D space. Knowing that calculating a convolution of an entire image can be represented inherently as operations between matrices, a GPU can be purposed via General Programming for GPU (GPGPU) for calculating the convolution of a Matrix. Although, this was originally done by Krizhevsky [2], Chen et. Al [8] developed a set of libraries called CuDNN in which convolution is done by arranging the data serially and then applying a matrix multiplication function which is already very well optimized inside the GPU chip . The Novelty of CuDNN Convolution is that it loads the feature map and kernel once and then rearranges it for convolution inside the chip's memory, as where other approaches on GPU rearranged the inputs on the Chip's DRAM. As of now, CuDNN is the state-of-the-Art for Neural Network on GPU acceleration and is commonly used for all common AI frameworks such as Tensorflow, CAFFE and Pytorch.

Acceleration of convolution in GPUs can only go as far as optimizing matrix multiplication. Which is what Nvidia has done with their RT core. The overall benefit of using GPUs for neural network acceleration is that the device provides an incredible amount of throughput for convolution and the chip is readily available. This means that re-purposing the GPU via software is feasible by anyone hence its popularity. The downside of using GPUs is that it is built for general purpose programming, so it has several units that can do rasterization, shader cores... that are of no use for ML applications, occupy chip space and constrain the dataflow. Also, all math has to be built as independent operations between data on arranged vectors, and the accumulation part of convolution would require several clock cycles after the matrix multiplication is done. Finally all sharing of data has to happen through a cache. Processing units cannot share data between them since each thread handles individual/non coherent memory.

1.3.2 Systolic Arrays

Systolic Arrays were proposed by Kung et. all [9] as an homogeneous array of Data Processing Units (DUP) (a.k.a. Processing Element (PE)) in where each unit receives input data from their neighbors and computes a partial result each clock cycle. The result of different PEs can be then stringed together to form a final result (usually by accumulation). Systolic Arrays provide the benefit of high parallelism and reduced memory accesses. Data is usually read once and kept inside a PE until it is ready. Data that is shared across different computations is read once and shared across PEs such that it does not need to be re-read from memory. Jouppi et. all [10] state that reducing memory accesses is the largest advantage of using a Systolic Array architecture. This architecture is used by both Google's TPU and Tesla's AutoPilot Hardware 3 (AP3).

1.3.2.1 Processing element

In order to perform the dot product, each processing element only needs to perform multiply-and-accumulate(MAC). This operation is similar to the fused multiply and add in a GPU but it accumulates the result. i.e. uses previous output as input. A MAC takes in two vectors, multiplies element by element and accumulates the result. This hardware element can be very small because it can be comprised only of a MULTIPLY unit that multiplies two numbers and an ADD unit that accumulates the result after each multiplication before moving to the next element. Although each Processing Element can also include another units (activation, scaling...) in order to run a CNN end to end, these units will be idle most of the time, as activation is done only once per neuron and each neuron is the dot product of hundreds to thousands of inputs.

These two ADD and MUL units can be either floating point (FLP) or fixed point (FXP). On one hand, using the FLP MAC block can have a wider range of elements but it has high complexity, energy consumption and latency. On the other hand, FXP is simple and faster but it lacks the ability to express wide range of values. In order to leverage the the advantages of both, Intel, Nvidia, and Google have agreed to use BFloat16, which is similar to IEEE784 floating point binary representation but with the bottom 16 bits of mantissa cropped. The data will be assumed to be in this format (16 bit values, addresses are 2 bytes apart) in memory.

1.3.3 Dataflow to optimize Energy efficiency

The highly parallel nature of convolution (very few data dependencies) allows to maximize the processing throughput via parallelism. The big cost of parallelism is the huge energy consumption due to frequent data movement, which is the reason GPUs are very inefficient for CNNs. As stated before the benefit of systolic arrays is that they can share information without necessarily accessing memory. A Processing Element in a systolic array will typically have 4 levels of memory. Itself, its neighbors, a global cache and DRAM. Sze [11] states that normalizing the energy consumption with the energy per MAC op (1x), fetching the same words from a local buffer is 2x, accessing from a global buffer is 6x and 200x if accessing it from DRAM. As such there is a need of maximizing local and neighboring accesses over a global buffer and DRAM.

The order in which operations are performed in a systolic array is usually called a dataflow. The dataflow will determine memory accesses and has a direct effect in total throughput. **Optimizing a dataflow to maximize data sharing while keeping parallelism high will reduce memory accesses. This in turn will result in less energy consumption and given less cache misses and less DRAM accesses**. Chen et. al. [12] defined the dataflows as several posibilities:

- Output Stationary: One PE will compute an entire convolution pixel. The input window is unrolled into a vector (from its 3D original state) and the filter is applied one element at a time. Neighbors can be computing adjacent output pixels (in this case, the inputs are shared in space after they are used) or they can be computing adjacent channels by applying different filters to the same inputs.

- Weight Stationary: The weights are pre-loaded into each processing element. At each clock cycle, a new input window is brought in. The element wise multiplication is then performed and then the results of the multiplications are then accumulated into one final output.

- Row Stationary: While Output Stationary tends to have lower latency because of higher parallelism, it has far more memory accesses than weight stationary. Weight Stationary is slower but more efficient, as it will sometimes run out of space and will only compute a partial output. Row stationary leverages the advantages of both by computing only a partial dot product per PE. The division is done by splitting the dot product in rows. A row is computed per PE in the same way as output stationary. Another row belonging to the same pixel is computed in a neighboring PE. Finally, after each row is computed, accumulation with all the partial sums is performed. This was shown by Chen [] to have better overall performance by reducing memory accesses but at the same time keeping the compute units busy.

1.4 Proposed Dataflow

The commonality between all these dataflows is that when the size of the channels exceeds core availability on the systolic array, the data will be partitioned in 2D planes by cutting through channels. This is because historically, convolution has been applied to 3-channel input (RGB). As such the size of the filter in width and height will far exceed the number of channels. This approach makes sense from a DSP perspective, as all workloads will be homogeneous and a Convolution engine can be adapted for this scheme. This does not apply for CNNs. An important example to ilustatue this is seen in table 1.1 where the only layer with 3 channels is the initial one, after that the width and the height start decreasing and the channels start increasing.

IFMAP Height	IFMAP Width	Channels	WxH	WxC
416	416	3	173056	1248
207	207	16	42849	3312
103	103	32	10609	3296
52	52	64	2704	3328
26	26	128	676	3328
13	13	256	169	3328
11	11	512	121	5632
9	9	1024	81	9216
7	7	1024	49	7168

Table 1.1: YOLO layer partitions

From a cache perspective, we want to minimize cache size and maximize its utilization. This is done by increasing temporal/spatial locality of accesses. Given that when going through a layer we need to apply an increasing amount of filters to the same input window, it is safe to assume that to maximize temporal locality, we have to bring **adjacent** data once and apply all possible filters before eviction. Each dataflow will reuse 2D slices of data until all filters are applied. In the fourth column of table 1.1 we can see that if we slice tensors in width x height, the slices in the first layer are a orders of magnitude larger than the final layers. A cache for this type of application would have to be large and slow (to hold all the data) or small and fast but with the penalty or re-fetching data. Finally, this slow cache will be very underutilized for the last layers.

On the other hand, we can see in the last column of table 1.1 that if the data is partitioned by height (keeping a slice of channel x width) has less cache requirements for its highest, and it is more stable throughout the layers. With this, we can maximize utilization of a smaller cache. This because as more features are extracted (more channels) less raw information is required per feature (less width and less height). As such, if the dataflow is modified to access data in this pattern, faster smaller caches are required and total re-utilization of data before eviction is attainable. This scheme is going to be referred to as Chanel wise row stationary. Rows will be computed in a systolic array in where vertically adjacent PEs will compute the dot product of a row.

Memory aside, given that the number of channels grow in a CNN as we move forward (as seen in the Alexnet topology in figure 1.1), computing convolution through channels will achieve a higher utilization of a systolic array. The more channels, the more rows of the Systolic Array can be used. As a reference, lets take the second layer of yolo tiny with a filter size 3x3 and 16 channels (As seen on table 1.1). Figure 1.2 shows a systolic array performing a transposed but otherwise identical version of row stationary. Transposed convolution meaning each vertical column in the SA is computing the in the next channel and not the pixel down (as the original row stationary [13]). Moving in X or Y makes absolutely no difference when X=Y. Unlike the original implementation of row stationary instead of multicasting the entire row to each PE, this systolic array shares the data it uses at a given clock cycle to a neighbor that needs it in the next. The red represents the row and the channel in the filter that is being computed in a processing element. The blue represents the input data row: X=1, Row 1, Channel 1, then to its right the same filter but with X starting at 2... We can see here that since the height of our filter is 3 the maximum amount of rows we can use is 3. Filter height rarely exceeds 7. The most common filter height is usually 3.

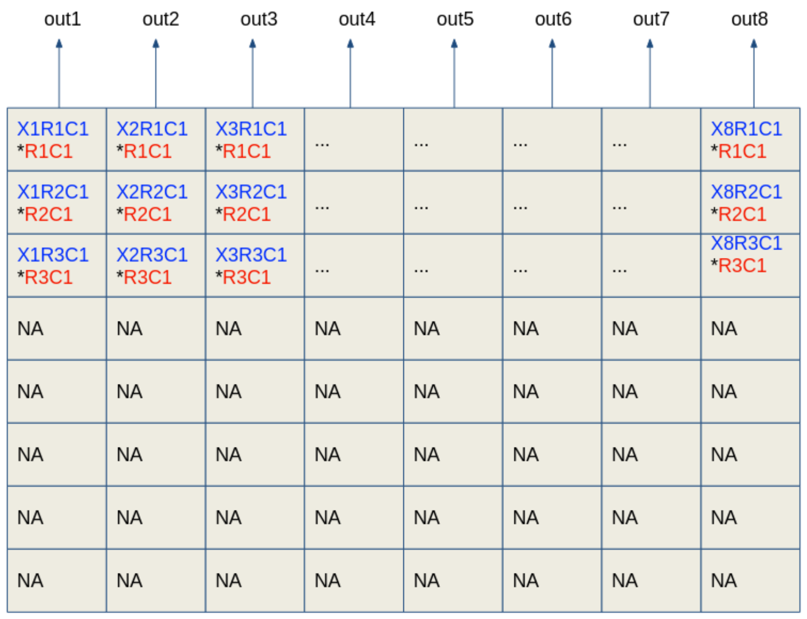


Figure 1.2: Height-wise (original) row stationary on a systolic array.

On the other hand if we map the filter channels in a row instead of the filter rows in a channel into the rows of a systolic array, there is more potential for Systolic Array utilization overall because the filter size keeps growing. As a matter of fact this will lead to saturation in Systolic array rows most of the time. This is depicted in figure 1.3 where 100% of the systolic array is used as there are enough channels to fill it up. If a the number of channels is 16 and the systolic Array is 8 rows tall, the operation can be split into two sequential operations. In each sequential operation the dot product of 8x3 chanels will be computed and finally, they will be accumulated in memory.

This technique will be referred to as *folding* as the operations are folded to the size of the systolic array. Folding is only limited to two dimensions here as the systolic array only has two dimensions. The most common type of folding is when output width exceeds the Sytolic Array width. When this happens as much pixels as there are columns in the SA will be computed at a given time. Once that calculation is done, the next segment that can fit in the SA will be computed until the corresponding partial sum for entire row of the output is computed. The second type of folding is when the number of channels exceeds the number of available columns. This is less common. Folding channels will be prioritized over folding adjacent pixels as getting the partial sum that accumulates into the same address will group accesses to the same addresses closer in time, making a better use of temoral locality. Finally, the rest of the dimensions (next filter row, next filter, next layer...) are folded in time.

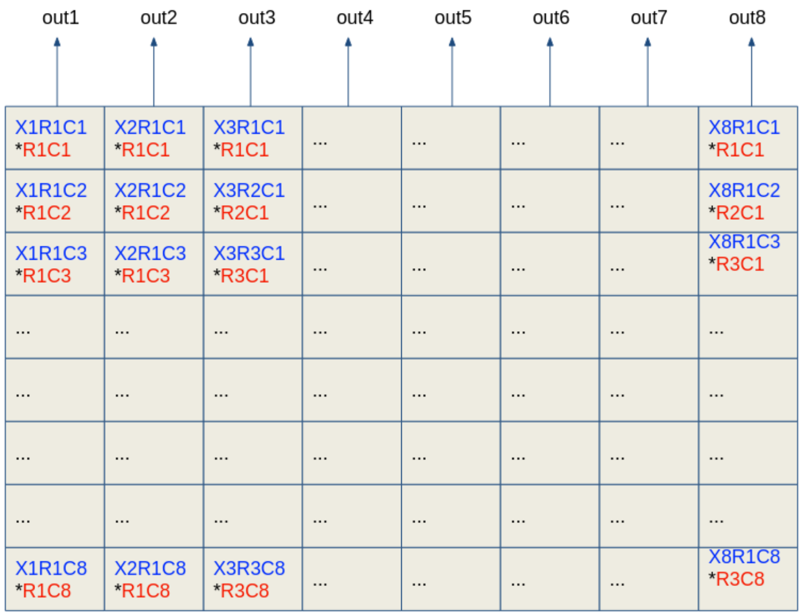


Figure 1.3: Channel-wise row stationary on a systolic array.

More array utilization at a given time translates into more parallelism and more parallelism results in higher throughput and lower latency. The approach getting more parallelism in the original row stationary was to map multiple filters in parallel. This makes filter and output memory accesses more sparce. On the other hand, accumulating more rows in the systolic array reduces doing accumulation when writing on memory, saving a read an a write per extra value accumulated inside the Systolic Array.

2. METHODS

In order to test that this dataflow increases throughput and reduces latency, three metrics need to be measured. The first one is **utilization**. Given that the amount of computation remains the same for different dataflows, if the Array is underutilized (for any reason) the program will run slower. The second metric is **cycle count**. Cycle count in this model assumes that memory acceses take one clock cycle and data is pre-fetched by filling the the data before it is required to a cache. As such, loads and stores take the same amount of time as a sequential non-pipelined BFLOAT16 floating point multiply and add. This will allow the study o focus on the increased parallelism of this dataflow.

The third metric is **cache size**. In order to determine cache size, memory accesses will be evaluated for different dataflows. Memory will be arranged in a way that maximizes spatial locality for a different dataflow. Two dataflows weree tested: Original Row Stationary and Channel Wise row stationary. The data was arranged in an W>H>C for the first and W>C>H for the later. With the data arranged in this way, all the data in that 2D slice is sequential. Memory accesses will then be analyzed to confirm that each dataflow uses each 2d slice to exhaustion.

2.1 Systolic Array Simulator

In order to obtain these metrics, a systolic array simulator was built. This cycle accurate simulator is composed of four parts as seen in figure 2.1. The simulator architecture is scalable and can simulate any size Systolic Array. However, for this study the size selected is 32x64 Processing Elements in order to enhance the difference in utilization. Smaller Arrays do not see much difference because the require more folding and larger ones are severely underutilized (not to mention exponentially difficult to fab). The four parts of the simulator are:

- **Systolic Array**
- **Accumulation Vector**

- **Register File**
- **Memory port**

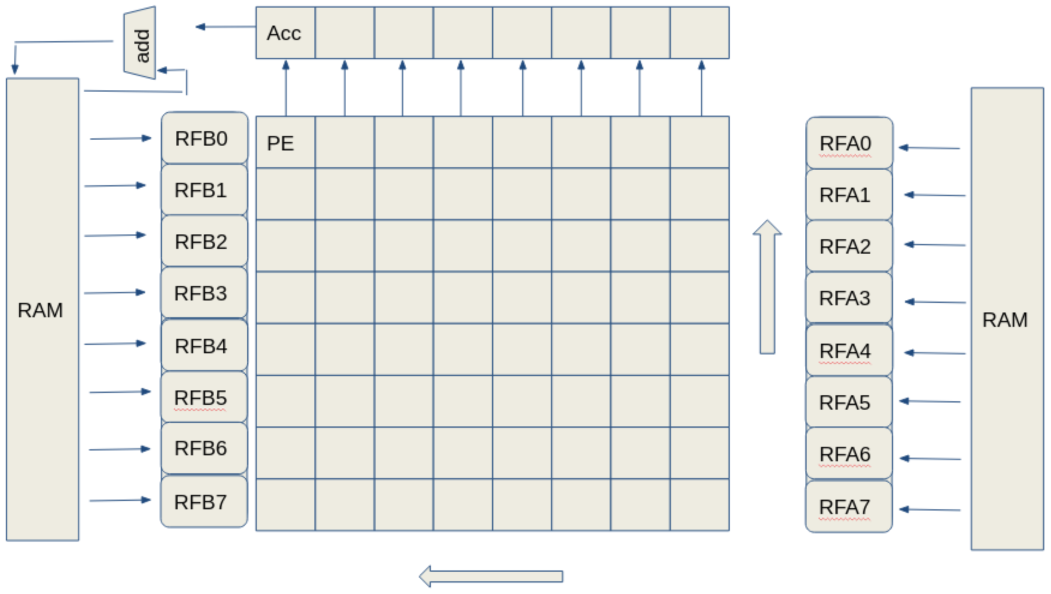


Figure 2.1: Systolic Array Simulator

2.1.1 Systolic Array

The systolic Array is composed of 32x64 Processing Elements. Each PE will do accumulation and addition of one local value and two external iputs (all in BFLOAT16) in each clock cycle. The Processing Element is seen in figure 2.2. At the positive edge of a clock cycle, each PE will overwrite its registers A/B if the respective write wires are high. The three registers in a PE A, B and O will be externally visible. This so inputs can be shared to neighbors and the outputs can be read out. The O register will accumulate the data. The systolic Array will allow each PE to share its input register with its right neighbor. After using the data in A, each PE can pass its A register value to the A register in its neighbor to the right. The systolic array also treats B as a multi-cast. At any given time all the B values in a row will be the same. Finally the O register will be accumulated into memory once the operation is done.

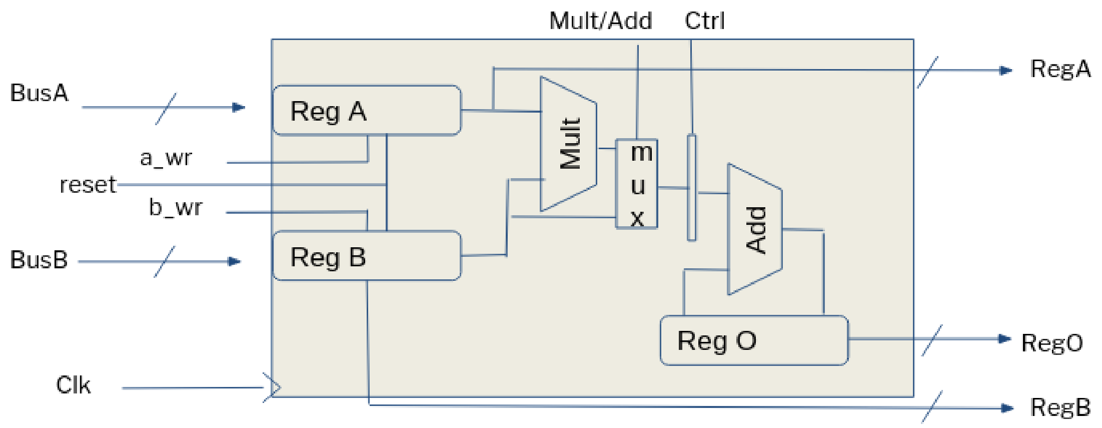


Figure 2.2: Processing Element

2.1.2 Accumulation Vector

Once Convolution is done, the data is collected into n accumulators where n is the number of columns (32). These accumulators are be able to add 3 inputs per cycle, as add and multiply take a similar amount of time and the clock cycle is dictated by the time taken by a sequential add and multiply. When collected, each value contains a target address. If addresses match, they are reduced to one by adding them. Finally, once all final additions are performed, data is ready to be stored back to memory. The accumulation vector can choose whether it accumulates the new value in or just write to memory. For the former, the address value is read from memory then the new value is added to the value in memory and finally stored back.

2.1.3 Register File

Two unified register file acts as a buffers between memory and the array. Data is loaded into the register and the instructions operate only having the registers interact with the systolic array. Register File A will be used for the data that is shared across space after each clock cycle and

Register File B will be used for the data that is multicasted and shared across different PEs at the same time. Unified Register File A (RFA) writes to Register A in each PE and RFB to B respectively.

In order to simplify hardware, each register file is partitioned in sub files that communicate to the PEs of one row of the systolic array. For A, there is one port for each PE in that row. For B, there is only one port and its value is multicasted to all the B ports in that register file. This enforces sharing at least one value in time. Sharing common operands is the final goal.

2.1.4 Memory port

Finally, two memory ports will retrieve the data from memory and load it into the register files. At the same time, they will log address and timestamp for every read/write for further analysis. It is worth noting that for output, the data is first read, then accumulated with the write value. This memory port will be assumed to have a cache interposing with memory. Since all accesses are static and are invariant depending on the data (no control instructions) a cache is just a expected to hold certain sections of memory at a given time.

Since memory accesses are somewhat uniform, the structure of the cache is designed to support the worst case scenario.

2.2 Instruction Set Architecture

Having described the basic hardware and how parallelism and data sharing works, an ISA was designed to abstract the systolic array low level mechanics from a high level user. This is done through a control unit that reads and issues instructions. This was not included in the diagram as it touches every part of the simulator and it would add unnecessary clutter to figure 2.1. With this said, this Control Unit is capable of issuing 4 instructions:

- MAC Pass A right: With data in the registers, load the data into the array, perform all the shifting and accumulation, tick the clock for multiple cycles as necessary.

- Read into A: Read Data into Register A from its memory port.

- Read into B: Read Data into Register B from its memory port
- Reduce by addition and write to memory: Collect the data from the Systolic Array and accumulate for writing into memory.

By laying all the data necessary sequentially, these 4 instructions can perform row stationary, output stationary and weight stationary. These instructions are designed in a variable length fashion but RISC fashion. RISC instructions are designed to be as simple as possible, decoupling computations from memory accesses for added versatility. As such, memory accesses and computation are separated but connected via a unified register file. Each instruction has a variable length. The common operation between all instructions is that the first field is composed by their Operation Code which is the only regular field that defines the type of instruction and then it is followed by its length. Depending on the Operation Code, the length of the instruction will be a multiplication of said length times the number of operands expected for that type of operation. This is how this applies to each different instruction:

2.2.1 MAC Pass A right

Multiply and Accumulate and Pass A right is the workhorse of this architecture. It enables Output Stationary, Row Stationary and Channel-Wise row stationary by forcing sharing in space and time. The Operation Code for MAC Pass A right is 0. It is followed by how many rows of the systolic array are going to be performing said operations.

This instruction assumes the following:

- All PEs in the systolic array are processing the same length of a dot product. As such, M rows and N columns of PEs will all be actively computing data throughout F clock cycles. Where F is the size of the filter or partial filter. M is the smallest between the number of adjacent data that we can process (e.g. number of columns in output map) and the number of Columns in the systolic Array. And finally N is the number of parallel independent operations that each row of the Systolic Array is processing.

- The same filter is used for all the data in each SA row and that said filter is located in RFB0 for row 0, RF1 for row 1... Since all filters are the same size, each value of a filter is applied to an entire row each clock cycle. This allows data sharing in time.

- Each PE will use its A value and pass it right for the A value of their right most neighbor for next clock cycle. In this way there is data shared in space. This helps reducing register file accesses after the first clock cycle. It also forces the systolic array to process adjacent output pixels, by forcing them to share inputs with a one element offset.

- RFAx will contain the M + F adjacent values where M is the number of PEs being used in a row (also the number of columns used in the SA) and F is the size of the filter - 1.

. Take the example in figure 2.3. Assume we have 3 output pixels, that are obtained by convolving a 3x1 filter through a 5x1 input space. Each output pixel is computed in a a different PE. The values B1, B2 and B3 are the filter data. I1, I2, I3, I4, I5 are the input pixels. The first output pixel will be composed by the dot product between [I1, I2, I3] and [B1, B2, B3]. The second output pixel will be the dot product of the same filter and [I2, I3, I4]... The I2 value PE 3 needs in cycle 2 is the value that PE2 used in the firs clock cycle. Same thing for I3, B4... As such, only the initial values of I need to be loaded per instruction, for the other values of I, each PE will use their value of A and then pass it to the right. All of these accumulations are stored inside the O register, after three clock cycles, we have our final results computed in parallel while sharing all possible operands in space and space time.

Now this example showed the operation of one row of three PEs, each row in the systolic array will operate independent of the other, For that reason, each row has their own RFA and RFB unified register file. As such, for each row, the instruction needs to encode the initial register A and B. The assembly version of this instructions is explained in table 2.1.

2.2.2 Read into A and Read into B

These two instructions are pretty straight forward. Since both the input window and the filters are composed of sequential data, these two instructions are designed to import the data window

Attribute	Description	Ex.
Opcode		0
SA rows used (instruction length)	number of rows used	2
Clear O	clear the O register before using PEs	1
cycles/filter width	filter size	3
SA cols used	number of adjacent pixels being computed	3
RFA0 initial Address	initial address in RFA0 of input window for row	0
RFB0 initial address	initial address in RFB0 of filter	0
RFA1 initial address	...	0
RFA1 initial address	...	0

Table 2.1: MAC & Pass A right

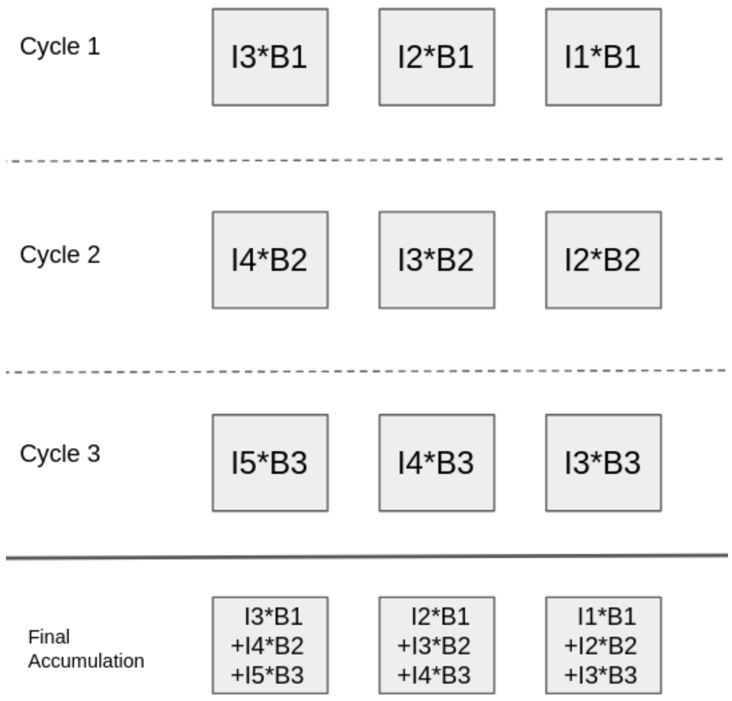


Figure 2.3: MAC & Pass A right

the systolic array is going to be working on. That is done by reading into sequential registers, these instructions are composed of their Opcode and the number of reads. For each read we have the target initial address of the read, the initial target register, the size of the read, and the target register row. An example is shown in table 2.2

Attribute	**Description**	**Ex.**
Opcode	1 when reading into A; 2 when reading into B	1
Number of reads	(instruction length)	3
Initial memory address	address of first value in this read	0
Initial register address	number of register in RFA0 where data is written	0
size of read	number of sequential values being read into the register	5
target row	number of the column in this case RFA0	0
Initial memory address	address of first value in this read	20
Initial register address	number of register in RFA1 where data is written	0
size of read	number of sequential values being read into the register	5
target row	number of the column in this case RFA1	1
Initial memory address	address of first value in this read	30
Initial register address	number of register in RFA2 where data is written	0
size of read	number of sequential values being read into the register	5
target row	number of the column in this case RFA2	2

Table 2.2: 3 reads into A

2.2.3 Reduce by addition and write to memory

Write to memory assumes that O registers in a set of PEs contain all computations possible given the input window and filters. As such, they are ready to be stored into memory. Given that different rows are operating on independent data, the write out instruction incentivizes that north to south neighbors perform partial sums that belong to the same output pixel. This is similar to the original Eyeriss [13] in where row 1 would compute the first row of the filter, row 2 would compute the second row, row three the third... Once the partial sums are computed, the data in one dimensional columns of a PE are extracted and reduced to one single datapoint by addition as they are partial sums of the same output anyways. In later implementations of Eyeriss, a column could be segmented into a set of different groups that write out into different addresses.

This reduction by addition is enabled with instruction in this architecture. The instruction contains a number of writes. Each write contains a SA column address, a SA column initial point (row number) and a column size size. The size dictates how much PEs under that initial PE need to be reduced into the same address. For example, on the original implementation of row stationary depicted in figure 1.2 after the dot product of a filter row and input row is computed, given a filter size of three, we have 8 columns of size 3 that need to be reduced to to 8 independent addresses. Assuming they are all writting to contiguous 16 bit addresses starting at address 1025 the instruction to accumulate the data on that picture would look like what is depicted in table 2.3

Attribute	**Description**	**Ex.**
Opcode		3
Number of writes	(instruction length)	8
Address	target memory address	1025
SA Col	Column in the systolic array	0
Column Initial address	Address of first PE in column	0
Column Size	Nomber of PEs to be reduced in this read	3
Address	...	1026
SA Col	...	1
Column Initial address	...	0
Column Size	...	3
...	...	...
Address	...	1031
SA Col	...	7
Column Initial address	...	0
Column Size	...	3

Table 2.3: Reduce by addition and write to memory

2.3 Instruction Length

A runnable architecture has a defined instruction length. This is not trivial when architecture is scalable. That is the size of the systolic array determines the length of the instructions. Large operands are required for large arrays like Google's 256x256 TPU, and smaller operands are needed for a smaller architecture like Eyeriss 12x14. At the same time, the compiler needs to be aware of the size of the Systolic Array [14]. If this data is not known by the compiler, it may map operations to non existing hardware. Given that the compiler is aware of the size of the hardware, the instruction length can be specifically tailored to the size of the maximum value it needs to hold. The parameters needed by the compiler are:

- Array width(A_w): number of columns in the Systolic Array
- Array height(A_h): number of rows in the Systolic Array
- Extra Registers (E_r): In order to encourage an extra degree of sharing registers are asymmetric. The inputs tend to be larger than the filter. As such, each row of RFA has $A_w + E_r$ registers, while each row of RFB has E_r registers.

2.4 Compiler

Using these 4 sets of instructions, one is capable of mapping variations of output stationary, weight stationary and row stationary. The novelty of this architecture is that it doesn't constrain convolution to a single dataflow. As a matter of fact whatever dataflow is more efficient can be used at for each layer as long as the data alignment is consistent. Alignment being the order in which 3D tensors are unrolled into memory. However if there is spare memory, tensors could be rearranged at the same time that they are activated. However, activation and pooling is not yet included in this architecture as this is a trivial task.

Chen et al [13] already demonstrated that Weight stationary unbounds parallelism at the expense of only reusing filters and serializing reduction. Eyeriss also proved that Output stationary has the highest level of parallelism at the expense of little reuse and high memory bandwidth.

Chen [13] shows that Row Stationary outperformes both by compromising the best of these two dataflows. As such, this study is only focused on the memory access patterns of two different implementations of row stationary.

2.4.1 Mapping

Convolution is originally mapped as a filter (3D in case of CNNs) moving through a larger tensor. In an ideal world, this is done through a weight stationary dataflow, however with varying filter sizes and fixed hardware this is not feasible. As such, the compiler will divide the operations by input window. An input window is the largest set of data that can be mapped into the systolic array. This approach divides the operation by available hardware. For both implementations of row stationary this is done in the following sequence:

1. Load Data into A &B: Instructions 1 and 2 are used to load the filters into RFB and the input data into RFA.

2. Convolve: Use instruction 0 once the data is in its respective registers. Convolve will fill up all register A in each PE with consecutive data from RFA. e.g. PE(0,0) will have data Input pixel 0(RFA0-0), PE(0,1) will have data from input pixel 1 (RFA0-1)... For the two variations of row stationary each row in the systolic Array will have data from the next channel or the next row respectively. Since all the PEs in a row are applying the same filter to different data, one value of each RFB will be used for the entire row. At the end of this operation, each PE will contain the resulting partial sum however, there is no information inside the PE that ties this result to its destination.

3. Reduce and store: Once all the data is in all the PEs instruction 3 will is used to reduce all the rows that are writing to the same address in each column and write them back to memory.

2.4.1.1 Order of operations

An input window can only produce a set of dot product of one dimensional rows per SA row. As such each SA row will compute a 1D row of a row of the output. That is PE(0,0) will compute

1 row for output pixel 0, PE(0,1) will compute a row for output pixel 1... For initial layers, the output will likely exceed the number of PEs per row in the SA. When we SA rows saturate, it is necessary that operation is *folded* in time. This is that the operation will be broken down by the available hardware and scheduled sequentially in time. For example if the output map consists of 10 pixels but the array is has 4 columns, this operation will be divided by computing pixels 0-3 in parallel then 4-7 and finally 8&9. This is called folding as it is analogous to folding a sheet of paper.

Once all the rows are folded the next step is folding the vertical dimension . If the filter size (for heightwise) or the channel size (for channelwise) exceed the number of rows in the SA, they are folded next. For channelwise, the goal is to reuse the same 2D horizontal plane so all rows of a filter that align. As such, all the rows of a filter that apply to that same row of the tensor are applied sequentially. After a filter and input are mutually exhausted a new filter is brought in and filters are applied sequentially. Finally, the filters are moved to the next row in the output pixel. Folding only happens for the inner two dimensions (output width and number of channels). After that, each operation is mapped sequentially and time has no constraints. If more parallelism is to be extracted, an extra level of folding is required. For a detailed version of how memory is read, addresses are generated, and registers are used see the code in scripts folder. For a high abstraction overview, the pseudocode to how this operation is mapped is shown bellow.

```
for row in output_map
    for filter in filters
        for f_row in filter_rows
            #map to adjacect columns in systolic array
            parallel for x in output_map
                if x>SA_cols
                    fold_cols()
                #map to different rows in systolic array
                parallel for channel in channels
```

```
                    if f_row > number of columns
                        fold_rows()
                    #accumulate into output pixel
                    output_pixel += f_row*row
```

On the other hand, the original heightwise variant of row stationary, will fold output width and filter height. This will perform a 2D dot product. Once the convolution is done for an entire output row, the next row is computed to reuse the same filter. This is followed by applying a different filter in order to produce all output channels that use that same 2D input window. Finally a the next channel is computed . The pseudocode for this operation in shown bellow.

```
for channel in channels
    for filter in filters
        for row in output_map
            #map to adjacect columns in systolic array
            parallel for x in output_map
                if x>SA_cols
                    fold_cols()
                #map to different rows in systolic array
                parallel for f_row in filter_rows
                    if f_row > number of columns
                        fold_rows()
                    #accumulate into output pixel
                    output_pixel += f_row*row
```

2.4.2 Operation Correctness Verification

The simulator produces an output memory map, this is used to verify that throughout each dataflow, the correct data is obtained at the end of the run. In the process array utilization and memory accesses per cycle are logged. As stated before, ensuring higher utilization will translate to lower latency (better performance). This because higher utilization means that more operations can be ran in parallel. For a dataflow this means that given a fixed input window, more operations can be mapped at the same time. In addition to this, the memory metrics leave a print of reuse. The goal is to map operations such that accesses are grouped together, increasing temporal locality with it. In addition to this, maximizing the access of adjacent datapoints will help increase spatial locality. These accesses will be used to determine cache size.

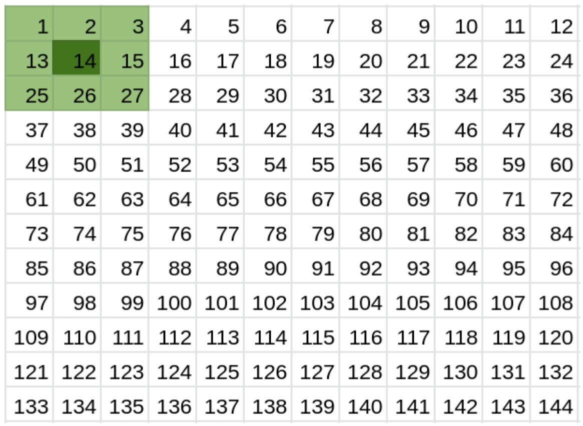
As stated before, correctness of the output is the most important goal. Since correctness of the operation is not enforced by the architecture, ensuring that the output is correct is the responsibility of the coder who writes the compiler. Currently there is no proven method for validating results in Accelerators, especially CNN ones. Inability to formally prove this a significant the factors why Google [10] does not sell their TPU unit to customers. On a non formal sense, verification can be done by pre-computing the expected output, running the program and checking the final state of memory to verify that the output matches the input. A helpful way to do this is to design tensors and filters that are designed to test that the operation being performed is actually convolution. This as opposed to using real CNN data.

A scalabe data generator was built in order to automate this for any layer. The input tensor is composed by numbers counting from 1 to MxN where M and N are the tensor width and height respectively. Orthogonal pixels (same X,Y position but different channel) have the same magnitude but every other channel is multiplied by ± 1. This way for an odd number of channels all the results but the ones of the first layer will cancel each other out. For an even number of channels, the first two will just be halfed so when added, they make one and the rest of the even number of channels cancel out. For example assume the result of convolution in one channel (assume one filter in one position) is 5. If the number of channels are odd, the data generated will make sure that each other channel is the negative of the first. If we add the result of, for example, 7 channels we will get 14 +14-14 +14-14 +14-14 = 14.

The filters generated by the data generator are far more simple. Each value in the filter is 1/(mxn) where m is the filter width and n is the filter height. In CNNs square filters with odd numbers are common as they are centered, help account for rotation invariance and all pixels have

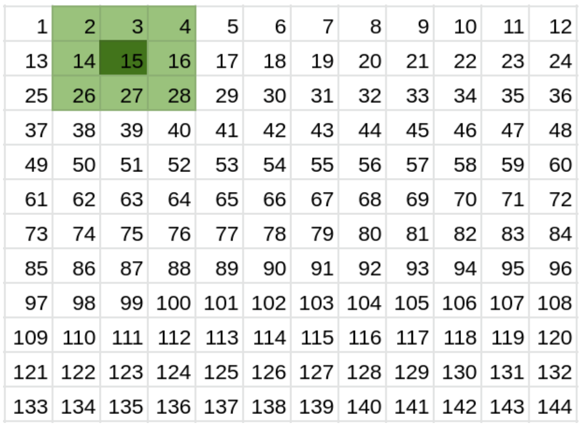
several isometric counterparts from the center pixel. With an odd m and n an having m=n, the result of each channel of convolution will always be the center pixel as the isometric pixels average each other out. For example figure 2.4 depicts the computation of a 2D input going from 1 to 144 (12x12) for a single channel tensor. If the filter (3x3 green square) is composed of nine 1/9 set in a 3x3 the result of the convolution for fig 2.4-a will be 14 (depicted in bold green) because
$\frac{1}{9} * 1 + \frac{1}{9} * 2 + \frac{1}{9} * 3 + \frac{1}{9} * 13 + \frac{1}{9} * 14 + \frac{1}{9} * 15 + \frac{1}{9} * 25 + \frac{1}{9} * 26 + \frac{1}{9} * 27 = 14.$

For the next pixel (figure 2.4-b) the result of the convolution will be 15, then 16... Using a filter that averages out all the values in a channel and outputs the an integer that matches the center pixel from floating point numbers makes it easy to verify for correctness.



1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72
73	74	75	76	77	78	79	80	81	82	83	84
85	86	87	88	89	90	91	92	93	94	95	96
97	98	99	100	101	102	103	104	105	106	107	108
109	110	111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130	131	132
133	134	135	136	137	138	139	140	141	142	143	144

(a) Pixel 1

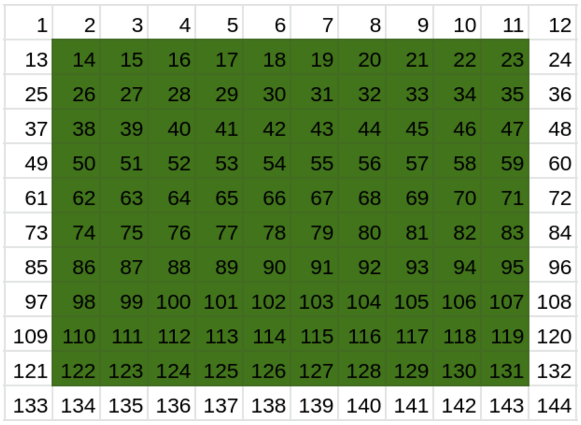


1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72
73	74	75	76	77	78	79	80	81	82	83	84
85	86	87	88	89	90	91	92	93	94	95	96
97	98	99	100	101	102	103	104	105	106	107	108
109	110	111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130	131	132
133	134	135	136	137	138	139	140	141	142	143	144

(b) Pixel 2

Figure 2.4: Example of 1 Channel convolution

The result of this convolution will be the green square depicted in figure 2.5. This result will be the same with the channel value flipping. Another way to check is just keeping the magnitude the same and checking that the result is equal to the center pixels*number of channels.



1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72
73	74	75	76	77	78	79	80	81	82	83	84
85	86	87	88	89	90	91	92	93	94	95	96
97	98	99	100	101	102	103	104	105	106	107	108
109	110	111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130	131	132
133	134	135	136	137	138	139	140	141	142	143	144

Figure 2.5: 1 channel convolution final result

3. RESULTS

The workloads selected to test the new dataflow and compiler were YOLO (which is the state of art image vision world) and AlexNet. Although other models (workloads) have shown better accuracy, these two were selected because they are purely convolutional. More modern models make use of inception layers which have a high number of 1x1 "convolutions" that do not allow sharing in space/time. With the current compiler 1x1 convolutions would perform drastically better in channelwise row stationary than the original one, however, no implementation of row stationary is optimal since rows are 1 pixel wide. Further studies from the compiler side will explore these workloads.

3.1 Memory access logging and timing

Although reducing the cycle count is the ultimate goal, memory accesses are crucial in order to figure this out. The issue with selecting a dataflow lies not in extracting parallelism, as there is no scarcity of it, but in extracting parallelism while using adjacent sets of data. In order to quantify this, memory accesses are logged. The end goal is not only achieving higher systolic array utilization but more localized memory accesses. Especially for the input feature map. Gao et. al [15] point out that by reducing off chip memory accesses, not only one saves latency but also dynamic power consumption. Latency is trivial since the workload is deterministic and known at compile time and all necessary data can be pre-fetched. However energy utilization is important especially on edge devices that are designed to do inference with low power.

Although the original implementation of row stationary [13] provides an efficient solution to maximizing reuse by partitioning dot products into partial sums of rows, their decision of parallelizing partial sums across the filter height introduces too much variability in the global memory requirements, as shown in table 1.1. Gao et. al. [15] propose a solution to this by using a 256kB small cache that functions at speeds comparable to a non-pipelined fused multiply and add operation. This was done by getting the timing in a 45nm process for the systolic array logic and a similar process for Cacti [16] in the cache. As such in this analysis we will not focus in timing but in raw cycle count and the case whether partition is required or not for each layer.

3.1.1 Memory overlap and partitioning

Data is partitioned in "vaults" each vault is to be streamed from memory and operations are mapped such that same data is used. If the size of the input window exceed the vault size, data needs to be evicted and re-streamed. If the eviction rate is too high (for example for filters) data completly bypasses the local buffer. Both TETRIS [15] and EYERISS [13] define their lowest level of reuse as a 2D plane that cuts across channels with size width x height as shown in figure 3.1(a). If this size exceeds local cache data needs to be partitioned into overlapping dimensions. The proposed channelwise plane stationary cuts the data across width into slices that are height x channels as seen in figure 3.1(b).

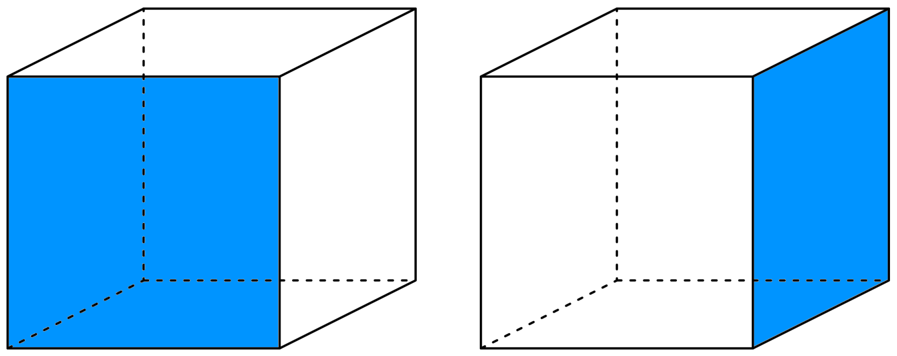
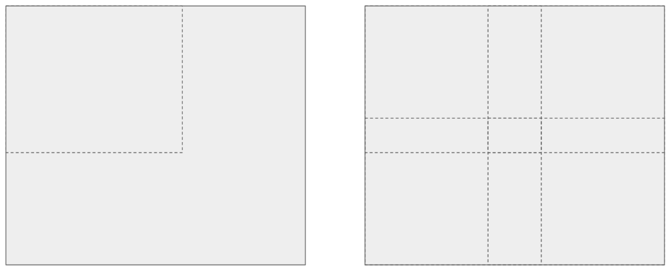

(a) Heightwise IF tensor partition (b) Channelwise IF tensor partition

Figure 3.1: Tensor partitioning overlap for different dataflows

For the original row stationary, since the filter is moving in two dimensions across the vertical and horizontal axis, partitions have two dimensions of overlap. This means that data evicted will get re-fetched incurring in unnecessary dynamic memory usage. This is depicted in figure 3.2(a) in where subdividing an input window into four vaults will require two-dimensional overlap. On the other hand partitioning across channels is relatively cleaner since channels the same filter can be split without any overlap as seen in figure 3.2(b) where the vertical access is the channels and horizontal is the tensor height.


(a) Heightwise IF partitioning partition

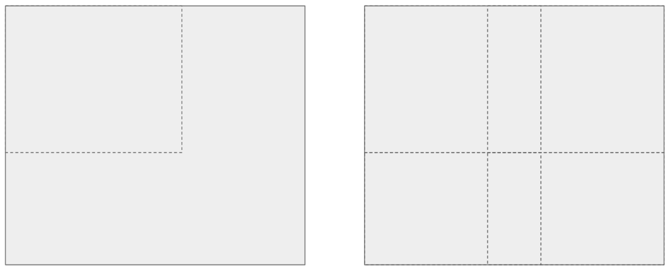

(b) Channelwise IF partitioning partition

Figure 3.2: Tensor partitioning for different dataflows

3.1.2 Memory Access

The channel-wise row stationary dataflow was built around having similar size vaults independent of the layer (as shown on table 1.1). This as a result of channel increase while tensors get narrow as higher level features are extracted. Using this, the primary goal of this dataflow was to fetch one vault at a time and use it to exhaustion and evict it. This ensures less dynamic power usage as there is less fetching from memory. In order to exemplify this, the memory accesses at each clock cycle were graphed for a tensor size 52x52x16 and 4 filters size 7x7x16. In figure 3.3 we can see that for the original height-wise row stationary, the input windows are 52x52 (x2 as each datapoint is a 2 byte bfloat16/FP16) or 5.4kB. On the other hand we can see that for channelwise row stationary the vaults are a lot smaller in 52x16 (x2) or 1.6kB. This is depicted in figure 3.4.

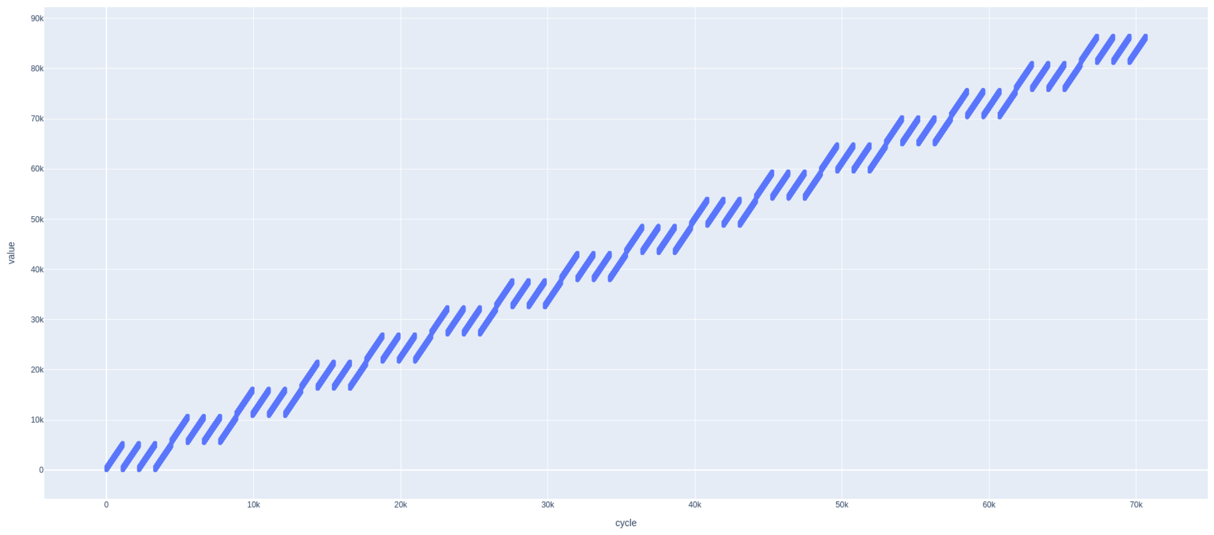


Figure 3.3: Input Feature map accesses with height-wise row stationary.

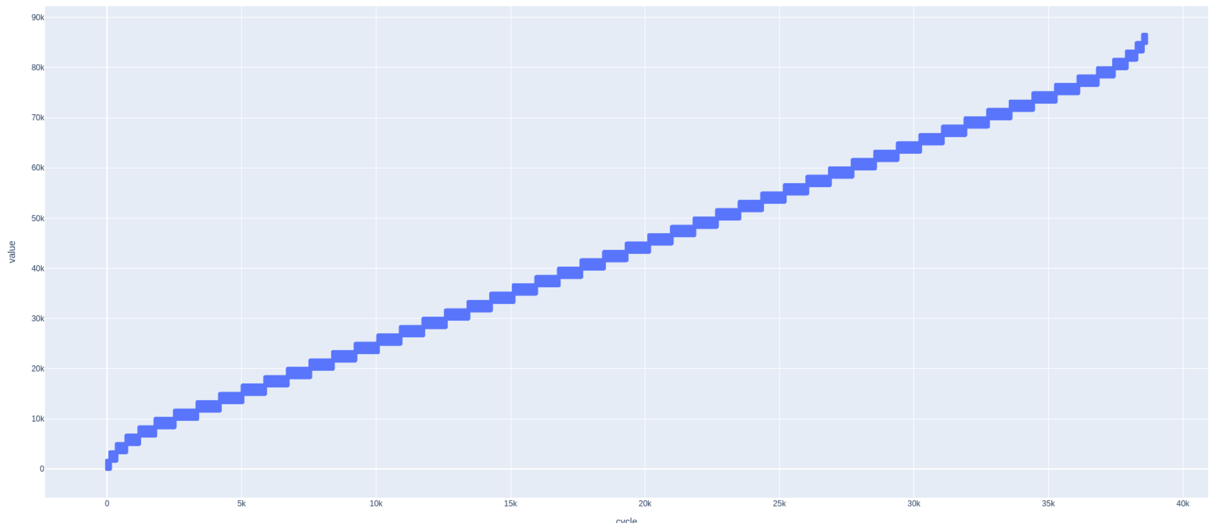


Figure 3.4: Input Feature map accesses with channel-wise row stationary.

Localizing memory accesses helps lowering cache utilization and overall reducing the need for partitioning for larger tensors. These memory access patterns demonstrate that memory requirements can be calculated by looking at the workload without necessarily running the simulator. However, when designing and adapting different dataflows, it is necessary to run the simulator to verify for correctness. Plotting these logs helps drive exemplifies this. It is also worth noting that for this case, channel-wise uses 40k clock cycles while height-wise uses 70k. This will be explored in the next section.

Knowing that we can calculate memory requirements from tensor sizes tables 3.2 and 3.1 gives us a glimpse as to the cache sizes that we require. In these tables, lower and more stable is better. We can see that for the initial layer of YOLO tiny, the vault exceeds the size 256kB planned cache, requiring partition. This problem will only exacerbate in CNNs with larger inputs like the original YOLO. This partitioning for early layers will incur in extra dynamic energy utilization but since it doesn't affect utilization (because the data is pre-fetched) it will not be examined. It is worth noting that the extra energy usage is process dependent and there's only one layer that needs partitioning. As such, this effect was not taken into account.

Layer	CW IF vault	HW IF vault	CW filter vault	HW filter vault
1	1,344	100,352	66	242
2	39,744	85,698	960	50
3	6,656	338	1,536	18
4	9,984	338	2,304	18
5	9,984	338	2,304	18

Table 3.1: Alexnet memory requirements in Bytes

Looking at the vaults on tables 3.1 and 3.2 we can extract that the **standard deviation of required cache space for heightwise vs chanelwise is 3.4x higher for Alexnet. For narrower and deeper networks like YOLO, this standard deviation is 23.25x on HW vs CW**. This deviation just emphasizes how the tensor partitioning though channels that is necessary for height-wise row

Layer	CW IF vault	HW IF vault	CW filter vault	HW filter vault
1	2,496	346,112	18	18
2	6,624	85,698	96	18
3	6,592	21,218	192	18
4	6,656	5,408	384	18
5	6,656	1,352	768	18
6	6,656	338	1,536	18
7	11,264	242	3,072	18
8	18,432	162	6,144	18
9	14,336	98	2,048	2

Table 3.2: YOLO memory requirements in Bytes

stationary is sub-optimal from a cache perspective.

3.2 Per Layer utilization and cycle count

The goal of using accelerators is reducing the overall cycle count that is needed to perform a layer of convolution. This study focuses on comparing the novel channelwise row-stationary dataflow vs the baseline row stationary in a 32x64 PE systolic array. The results for each layer of Alexnet and YOLO are found in tables3.3 and 3.4.We can see from these results that for the first layer, in where filter height is more than or equal than number of channels, the original row stationary performs equally or outperforms channelwise. This because utilization is a function of number of rows that can be used at a given time and this is dependent on either number of channels or number of rows. Although they might outperform, systolic array utilization is rather low and latency is incredibly high. Once we go past the first layer, we can see that utilization stagnates in the original row stationary while it blows up in the channelwise implementation. This increase in utilization directly correlates to the drop in cycle count.

From table 3.3 we can see that the slowest slowdown in cycle count is 0.33x of channelwise vs heightwise given that the first layer has a filter height of 11 and only 3 channels. Other than that one layer, **overall speedup gained by using height-wise vs channelwise row stationary** ranges from 1x (layer one YOLO) to 32x (layer 7 YOLO) with an average of per layer speedup of 5.8x and a **cycles to cycles speedup of 3.06x for Alexnet and 4.60x for YOLO-Tiny**. From tables

Layer	Cycles HW	Utilization HW	Cycles CW	Utilization CW
1	7,334,209	16.42%	22,146,433	4.48%
2	314,302,465	7.00%	85,487,361	67.99%
3	6,488,065	1.61%	1,368,577	34.38%
4	9,732,097	1.61%	2,052,865	34.38%
5	6,488,065	1.61%	1,368,577	34.38%

Table 3.3: Alexnet run latency and utilization

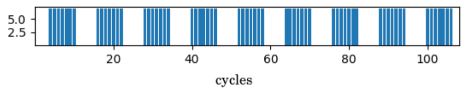
Layer	Cycles HW	Utilization HW	Cycles CW	Utilization CW
1	1,550,017.00	4.66%	1,550,017.00	4.66%
2	4,408,321.00	0.43%	1,515,361.00	22.88%
3	4,964,353.00	3.70%	1,241,089.00	39.45%
4	394,531.00	3.66%	1,036,801.00	78.13%
5	4,718,593.00	3.52%	995,329.00	75.00%
6	8,650,753.00	1.61%	1,824,769.00	34.38%
7	28,311,553.00	13.18%	884,737.00	28.13%
9	29,360,129.00	1.03%	9,289,729.00	21.88%
10	3,584,001.00	0.34%	350,001.00	21.88%

Table 3.4: YOLO run latency and utilization

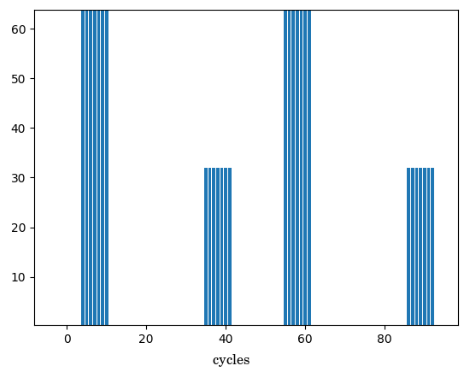
3.3 and 3.4 we can also see that cycle count is a function of utilization, however, the relationship is not linear. The micro-architecture spends more time performing reduction of partial sums than computing them and since this is a rather serial task, it takes a larger amount of cycles. This task can be parallelized by pipelining but this will explored in future publications.

Finally, it is worth noting that overall utilization does not tell the full story. In order to analyze why even through channelwise, utilization is still somewhat poor, one has to analyze the per cycle utilization of the systolic array. In figure 3.5 we can see the first 100 cycles of runtime in a workload that has a filter size 7x7 and 96 channels. We can see that in heightwise the systolic array row utilization maxes out at 7 as this is the filter height. For the channelwise implementation we can see that the filter saturates the array utilizing 64 of its rows, then folding happens and the next 32 rows are processed after the reduction is performed. We can also see that the array utilization is more sparse in channelwise, given that 64 values need to be reduced vs 7. This will be further addressed with studies into pipelining and the accumulation engine.

In addition to reducing accumulation cycles, better scheduling techniques can be designed that concurrently map the *edge cases* that result after the array is no longer saturated. The edges resulting after saturation can be seen starting around cycle 35 and 86 in figure 3.5(b). These two edges would saturate the array as well. Further studies into concurrently scheduing edge cases in dataflows that saturate need to be perfomed on the compiler side.



(a) Heightwise Systolic Array utilization



(b) Channelwise Systolic Array utilization

Figure 3.5: Systolic Array Utilization during the first 3 cycles

4. SUMMARY AND CONCLUSIONS

In conclusion, this study introduces a Scalable Systolic Array architecture that separates the study of different micro-architectural implementations from dataflow mappings. This framework allows a vast number of explorations not only in shapes and sizes of the hardware but also on how to extract parallelism while reducing the memory footprint in order to achieve lower latency when processing fairly large neural networks.

Using this set of tools two compilers were designed. An implementation of the original row stationary and a channel trans-versing one that maps different channels as opposed to different rows in the filer in parallel. Using this new dataflow it was proven that the average cache usage for the input map in channel-wise row stationary is 36.2% of what it is for the original using AlexNET and 17.3% of the usage for YOLO. It was also shown that there is better locality since the standard deviation of input partitions is 3.4x and 23.25x higher for the original row stationary. Showing with this better locality and less variance with the partitioning enabled with chanelwise row stationary. Finally in terms of raw cycle count, by saturating the array, Channelwise is able to get a 3.06x and a 4.60x speedup on AlexNet and YOLO-Tiny respectively. This shows an improvement over the standard dataflow but it was shown that there is area to grow, especially in large arrays and large CNNs.

4.1 Further Studies

Although throughout this study it is shown that the compilers generate code for performing 2D convolution with 3D tensors, there are multiple areas where the simulator needs study and the Architecture needs expansion. These areas include but are not limited to:

- **Caching**: Throughout the simulator, the cache system determined the clock cycle assuming the worst case scenario. This leaves room to explore in-depth timings in depth based on the cache's ability to retrieve specific data with different cache arrangements. An existing cache simulator can be replaced in the RAM ports.

- **Reduction**: Micro-architectural changes to the collection and reduction engine targeting minimization of output memory accesses and cycles used.

- **Pipelining**: Micro-architectural changes that sort out dependencies between parts in hardware in order to allow concurrent use of the stages andpipelining.

- **Generative Adversarial Networks** A study into how Channel-wise row stationary behaves with Encoder/Decoder style Neural Network topoligies.

- **Multicore Designs** A study into increasing parallelism by scaling up (going multicore) in order to parallelize inception layers that operate on the same input data but sequentially diverge converge.

- **Activation and Pooling** The results of each layer need to be pooled and activated. The architecture needs expansion in order to support this.

- **Edge Cases** Compiler studies into concurrently mapping edge cases that results as an effect of inevitable saturation.

REFERENCES

[1] X. Han, Y. Zhong, L. Cao, and L. Zhang, "Pre-trained alexnet architecture with pyramid pooling and supervision for high spatial resolution remote sensing image scene classification," *Remote Sensing*, vol. 9, p. 848, Aug 2017.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[3] F. Rosenblatt, *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory), Spartan Books, 1962.

[4] *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 1969.

[5] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Comput.*, vol. 1, p. 541551, Dec. 1989.

[6] E. M. Pete Bannon, "Tesla autonomy day."

[7] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–379, June 2016.

[8] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014.

[9] S. Kung, "Vlsi array processors," *IEEE ASSP Magazine*, vol. 2, pp. 4–22, Jul 1985.

[10] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho,

D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *CoRR*, vol. abs/1704.04760, 2017.

[11] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, pp. 2295–2329, Dec 2017.

[12] Y. Chen, J. Emer, and V. Sze, "Using dataflow to optimize energy efficiency of deep neural network accelerators," *IEEE Micro*, vol. 37, no. 3, pp. 12–21, 2017.

[13] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, pp. 262–263, 2016.

[14] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," *SIGPLAN Not.*, vol. 53, p. 461475, Mar. 2018.

[15] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 17, (New York, NY, USA), p. 751764, Association for Computing Machinery, 2017.

[16] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit.*

Code Optim., vol. 14, June 2017.

APPENDIX A

ON BACKPROPAGATION AND OPTIMIZATION

Learning is hard. It is uncomfortable. A Neural Network will not alter itself during training if it performs a right prediction. If it fails, it has to perform the arduous task of propagating through each layer the correct answer. We only learn through discomfort. Through performing and action and failing. That discomfort, that pain of failure is what triggers the process of readjusting, of relearning. As such, the act of learning is inherently uncomfortable.

From an evolutionary perspective, our brains ultimate goal is optimization. Maximizing the reward and minimizing the effort/discomfort. The rewards can take a multitude of effects but eventually they are all rewarded with dopamine. Those dopamine hits have no analogous in our electronic counterparts. They do not get directly rewarded by a correct prediction. They work solely through negative feedback. Us on the other hand, knowing that learning is uncomfortable, our brains eventually wire themselves to optimize out learning. That is why as we age it is uncomfortable to pick new skills, learn new languages, explore outside the areas we have built our entire careers in. Today I thank everyone who stepped out of their comfort zone so I could keep learning.