The Dissertation Committee for Kimble Derek Houck certifies that this is the approved version of the following dissertation:

Hardware Implementation of Inference in Deep Neural Networks

Committee:

Risto Miikkulainen, Supervisor

Dana Ballard

Don Fussell

Thibaud Taillefumier

Hardware Implementation of Inference in Deep Neural Networks

by

Kimble Derek Houck

Dissertation

Presented to the Faculty of the Graduate School
of the University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

The University of Texas at Austin
May 2022

Acknowledgments

First of all I'd like to thank the team I was a part of at Centaur Technology that designed an actual SoC implementation of the hardware design paradigm described in this document, including Glenn Henry for coming up with the original idea for the design (and recruiting me and supporting my continuing my PhD work while working for Centaur). Additionally I'd like to thank my colleagues Jim Donahue, Terry Parks, Kyle O'Brien, Parviz Palangpour, Tyler Walker, Benjamin Seroussi, Bryce Arden, Scott Gardner, Scott Petersen, Jonathan Johnson, Mark Ebersole, Anna Slobadova, Patrick Roberts, Doug Reed, Paul Zucknick and everyone else at Centaur who helped with the hardware design or encouraged my work towards my dissertation.

Second, I'd like to thank my advisor, Risto Miikkulainen, and my committee members Dana Ballard, Don Fussell, and Thibaud Taillefumier for their patience and encouragement along this journey. Additionally, I'd like to thank my masters thesis advisor, Andrew Fagg, for encouraging me to apply to UT Austin in the first place.

Finally, I'd like to thank my family and friends for all their support and encouragement, and especially my parents for their financial support.

Abstract

Hardware Implementation of Inference in Deep Neural Networks

by

Kimble Derek Houck, Ph.D.

The University of Texas at Austin, 2022

Supervisor: Risto Miikkulainen

Deep learning neural network algorithms, including convolutional and recurrent networks, have risen to popularity in recent years. Along with this popularity has come a wide range of implementations that optimize the performance of these algorithms on existing hardware, including GPU architectures and with modern x86 CPU SIMD capabilities. Likewise, effort has been put into developing hardware specifically for running these algorithms, either focusing on specific algorithms or on a range of building block operations common to many deep learning variations. While some of these architectures, have large power requirements and are generally designed to run in a datacenter environment, hardware architectures that are designed to run most deep learning well while being small, low cost and/or power are also important for applications where these are limiting factors.

In this work I will describe the implementation of both convolutional and recurrent network layer types on such a novel hardware architecture. This novel ultra-wide SIMD architecture is built around a ring of simple data movement and register units that feed simple arithmetic units, attached accumulator registers and post-processing units. Unlike many other architecture designs however, this class of hardware designs posses few methods for efficiently rearranging data over even moderate distances in memory but rather relies on shifting data between adjacent or nearby data units in the ring. Thus, neural network implementations that take the geometry of the

inputs into account as much as possible are needed. I present one such implementation, M^3inM^2V, and show that it allows such simple hardware architectures to be efficiently used for neural network inference, analyzing both it's performance on the described novel architecture and the very different AVX-512 SIMD architecture.

Furthermore, I show the applicability of recurrent network architectures to a novel domain; the decoding of information encoded in the electrical spiking activity observed from ensembles of neurons. By comparing the ability of a classifier to infer different pieces of information from the data and/or comparing classifiers trained using different methods of transforming the observed activity into feature vectors inferences can be made about what information is encoded in the neural signal, and how. By showing that deep learning classifiers can perform useful classification on such a dataset, possibly with less parameter tuning than other classifiers, I show that such tools can contribute to increasing scientific understanding of the brain. Furthermore, for future applications for decoding neural signals such as the control of prosthetic devices, the ability to run the decoding algorithms on relatively low power hardware would be highly advantageous.

Table of Contents

Chapter 1

Introduction

This dissertation presents a novel algorithm for performing convolution operations like those used in deep learning networks, on a proposed class of massively parallel, yet relatively simple, hardware architectures. Specifically, these architectures sacrifice flexibility in the ability to rearrange data in the SIMD vectors in exchange for the ability to support SIMD widths in the thousands of bytes. My algorithm facilitates this trade-off by performing the convolution in a geometry aware manner that eliminates the need for many complex data movement patterns, thus allowing it to be run on hardware that provides limited data movement capabilities.

1.1 Motivation

The rise in popularity of deep learning has spawned the development of new tools and hardware capabilities to run the algorithms more efficiently. This includes methods of computing deep network layers that take advantage of operations that have already been optimized on various architectures using standard compute libraries such as BLAS (Chellapilla et al., 2006, Mathieu et al., 2013). It also includes higher level software packages that separate the selections of these optimized methods and hardware on which to run them from the design of the deep networks themselves (Dieleman et al., 2015, Al-Rfou et al., 2016). On the hardware side new architectures have even been proposed, such as those by Chen et al. (2014), Qadeer et al. (2015), or Liu et al. (2016), which provide varying levels of generalizability to different and potentially novel deep-learning architectures. Some of these architectures have large power requirements and are generally designed to run in a datacenter environment.

However, hardware architectures that are designed to run most deep learning architectures well while being small, low cost and/or power are important for applications where these are limiting factors. One obvious route for supporting deep learning algorithms with such hardware is expanding the on-chip parallelism while keeping the circuitry as small and simple as possible. This can be achieved either by (1) increasing the number of cores or arithmetic units on the chip, or by (2) increasing the number on simultaneous arithmetic operations that each core can perform

at once. The first approach leads to duplication of overhead for each core and the need for more complex control mechanisms to coordinate between them. This design goes against the goals of simplicity and efficiency. The second approach, increasing Same Instruction, Multiple Data (SIMD) capabilities for a smaller number of cores requires less overhead to implement and extends the trend started by the growth of Intel's AVX instruction set to include 512 bit wide SIMD capability (Intel, 2021). However, while the amount of circuitry required to perform arithmetic operations on each element scales linearly with the SIMD vector width, the requirements for many data movement operations do not. Thus, algorithms' data movement requirements are the main obstacle in making ultra-wide SIMDs practical.

1.2 Proposed approach

This dissertation proposes a solution to this limitation in the context of inference in deep learning. The idea is to show that the flat approach to parallelism is feasible for these applications by implementing deep learning algorithms using the types of local data movement operations that do scale linearly with SIMD width. This approach, called ***M**emory **M**ovement **Min**imizing **M**atrix **M**ath - **V**ectorized **Conv**olution* (M^3inM^2VConv or $MinMVConv$) is the main contribution of this dissertation. In this work I will describe the implementation of both convolutional network and recurrent network algorithms on such a novel, ultra-wide SIMD vector hardware architecture.

The target hardware in question, *ncore* by Centaur Technology (Henry et al., 2020), aims to provide enough flexibility to implement a wide range of current and future deep learning algorithms, as will be shown by the implementation of a wide range of such algorithms on an emulator of the hardware. The hardware design attempts to create this flexible platform by way of an ultra-wide SIMD architecture, utilizing simple arithmetic units arranged in a ringlike fashion that allows for some simple data shift operations to be performed with extremely high efficiency. If this architecture were implemented in a form that kept cost and power consumption down it could be deployed in a wider range of commodity devices including small servers, laptops, gaming platforms and perhaps even smaller devices, allowing for the increased use of pretrained networks for things like facial, voice, gesture and/or hand-

writing recognition closer to the "edge". These tools could be applied to a wide range of use cases, from data input and device control to interacting with video games. Furthermore, efficient, widely deployable hardware support for deep learning models would facilitate support for such capabilities without the speed, security and privacy concerns associated with offloading these functions to the cloud.

1.3 Real world application: decoding neural signals

Building on the above approach, this dissertation shows that learning algorithms on such hardware provides an advantage in solving practical problems. In addition to consumer applications such a hardware accelerator might be applicable to a range of other scientific and industrial uses, especially if it addresses power consumption concerns. While "traditional" deep learning applications such as computer vision/image processing, audio and speech recognition and text processing have applications in a wide range of domains, one could imagine that other applications of deep learning algorithms could expand this applicability even further.

As an example of a lesser explored application of deep learning I will look at the application of recurrent deep networks to computational neuroscience, specifically the decoding of information encoded in the electrical spiking activity observed from ensembles of biological neurons. Simpler recurrent network models with a limited number of learned parameters have been recently shown to be effective at decoding neural spiking activity (Sussillo et al., 2012). In this work I will show that such meaningful information can be gleaned by gated recurrent networks from a dataset consisting of behaviorally correlated activity of neurons in an auditory processing area of the rat brain (Sloan et al., 2009, Sloan, 2009).

While most of the computational time is spent on training the networks, running such neural decoding algorithms on hardware that is optimized for inference using deep networks is still relevant. Specifically, while much current work on observed neural signals is focused simply on figuring out how the brain encodes information, many researchers hope that someday such decoding of neural signals could be used to drive brain-machine interface and neuro-prosthesis systems (Smith et al., 2013). While most research applications can make use of relatively large compute clusters, simpler hardware with lower power requirements would be better suited for running

a controller for a prosthetic device.

One limitation of this approach is that the training stage will not be considered on the proposed hardware architecture. However, the ability to decode neural signals using a trained network running on small and relatively low-power hardware is still an important advancement towards eventual applications in the domain of neural controlled prosthetic devices. Even without training capability such applications would benefit from the fast and potentially low power inference capabilities that could be offered by hardware architectures like the ultra-wide SIMD. Having such highly mobile and low power hardware available when the rest of the scientific and engineering hurdles have been cleared to make such prosthetics a reality, will be the final piece of the puzzle. Furthermore, there are likely many other applications in other domains, where the ability to perform inference using deep learning without the need for large and power hungry hardware will similarly help facilitate the deployment of practical innovations stemming from basic scientific advancement.

The field of neuroscience, along with many modern other biology disciplines, increasingly reliant on machine learning algorithms to not only analyze data, but to build models that can provide clues as to what theories might be fruitful to test experimentally, or what variables to look for in the data. Even old datasets can be reanalyzed with new algorithms that were previously computationally intractable to yield new insights. Thus, given that most researchers' budgets are finite, anything that increases the amount of usable computational power available for that budget increases the ability to analyze data.

1.4 Guide to the reader

The dissertation begins in Chapter 2 with an overview of related work, algorithms and hardware. Then I will provide an in depth description of the proposed hardware that MinMVConv will run on. Chapter 3 follows with the details of the convolution algorithm itself, including some interesting variants. Next, in Chapter 4, the performance of my proposed algorithm is analyzed as a function of the hardware parameters.

In order to provide an example of implementation of MinMVConv on existing, widely available hardware, an AVX-512 implementation is presented in Chapter 5.

My accompanying analysis of this implementation compares the hardware utilization efficiency to that on the proposed class of ultra-wide SIMD architectures for which the geometry-aware convolution algorithm was developed. While the number of data elements contained within a 512b SIMD line is insufficient to fully showcase the strengths of my MinMVConv, the implementation does still show that the algorithm can be implemented and run in real, existing hardware.

In further support of the general utility of algorithms like MinMVConv Chapter 6 addresses the issue of recurrent network layers. While the implementation of even "complex" recurrent layer types, such as LSTMs, is relatively simple. The fact that they can be implemented with an MinMVConv like approach shows that the proposed hardware and programming approach is capable of supporting networks that contain these layers, possibly in combination with convolutional layers.

To demonstrate the practical utility of MinMVConv to a new domain, Chapter 7 discusses the application of convolutional and recurrent deep networks to the emerging area of using deep learning to decode neural signals. While interesting in of itself from a scientific perspective, the proposed MinMVConv algorithm and the simple, ultra-wide SIMD architecture that it supports open the door for low power requirement applications of neural decoding, such as the control of prosthetic devices.

Chapter 2

Background

From the first perception model proposed by Rosenblatt (1958), neural networks have evolved into often large modern networks composed of many layers of varying types. These networks have inspired new algorithms and hardware architectures in order to run them, and have found their way into a wide range of applications. In this chapter I will give a brief overview of some of the neural network layer types used in modern deep neural networks. I will then describe some of the algorithms and hardware architecture proposed to train and perform inference on these networks, including a description of the proposed class of new ultra-wide SIMD architectures whose development I was involved with and that my work presented here describes algorithms for. Finally I will describe emerging applications of deep learning to the decoding of information from the spiking activity of real neurons.

2.1 Neural nets and deep learning

Starting from the simple perception model, neural network algorithms have evolved to include networks with many layers, and often with diverse topologies among these layers. Here I will briefly discuss this evolution, and then I will describe two of the major topologies that expand on the classical fully connect layer; convolutional networks and recurrent networks.

2.1.1 Deep learning and the resurgence of Neural Network Algorithms

While artificial neural network algorithms are much older than the current popularity of "deep learning" several factors have made the training and evaluation of large scale neural networks practical. The first factor is the increase in available computing power. The second factor is related to this, and it is the increase in the amount of training data available in certain domains. This is especially image processing, where the ability to crowd source much of the work in generating a dataset led to the creation of the well known ImageNet dataset (Deng et al., 2009), which then gave rise

to successful deep networks such as AlexNet (Krizhevsky et al., 2012). While more ambitious datasets and networks have been developed since, they all share common building blocks in their use of variations on convolutional, or ConvNet layers. Unlike fully connected layers, ConvNets utilize relatively small filters that are then used to perform a 2D convolution with the input. This requires fewer total parameters, and allows the networks to more easily capitalize of the spatial relationships within the input.

Likewise, for domains such as text and speech processing, where temporal information is important, improvements in recurrent network algorithms have made such algorithms practical in more applications. These improvements, which include adding gates to the recurrent units that determine what information is used to determine the unit's output (Hochreiter and Schmidhuber, 1997). These gates help prevent useful information from being lost over larger timestep intervals, and aid in training the networks.

Convolutional Networks

The first well known use of convolutional networks was the use of the LeNet architecture on the MNIST handwritten digit dataset (LeCun et al., 1998). While much simpler than most modern convolutional network applications this network had all of the building blocks used in those newer networks, including the reuse of weights through the convolution of the same filters with all locations across the input image, and the use of max pooling to reduce the dimensionality of each layer's output.

As an example I will discuss a simplified version of the LeNet network, with fewer layers, that is used as part of the tutorial for the deep learning toolkit Theano Al-Rfou et al. (2016). This network takes as it's input the 28×28 images from the MNIST dataset. Unlike the $224 \times 224 \times 3$ RGB inputs that AlexNet uses from the ImageNet dataset the MNIST inputs have only one channel. This means that while convolutional network filters are generally described as being $R \times S \times C$ filters where R is the number of rows, S is the number of columns and C is the number of channels the filters for this LeNet variation on the single channel MNIST dataset are $5 \times 5 \times 1$ filters. There are 20 such filters in the first layer of this network, are they are commonly referred to as *filter groups*.

The term filter groups makes more sense in terms of the second layer of the

network. Here each of the 50 filter groups take a 20 channel input (for a 5×5 filter). Each channel is the output of the previous layer. Within a filter group the 2D filter for each channel is convolved with that channel's 2D input by sliding the window across the input and summing the products of the filter coefficients with the "pixels" covered by the filter. The 2D matrices resulting for the convolutions of all channels are then summed to create a single output per filter group.

If *same size* convolutions are used then the input image is padded with zeros and the output of the convolution for each filter group has the same number of rows and columns as the input. If only the *valid* convolutions are used then the output is slightly smaller than the input, as convolutions where the filter hangs off the edge are not used. In either case, since the outputs of each filter group have roughly the same 2 dimensional size as each input channel the total size of the input to each layer will grow if the layers have more filter groups than input channels. To combat this, and to build some degree of translational invariance into the network a technique known as *max pooling* is used.

In max pooling the final output of each filter groups convolution is divided up into small sectors. In the basic case these sectors are small non-overlapping squares, and in the case of the modified LeNet network example discussed above are 2×2. Out of the convolution results in this square only the largest value is selected as part of the actual output to the full layer. Thus for the LeNet MNIST example, the 24×24 output of each filter group of the first convolution is subsampled via max pooling to yield a $12 \times 12 \times 20$ output across all filter groups in the layer.

Recurrent Networks

Convolutional layers can be mixed with fully connected layers to form powerful classifiers for inputs that represent a single point in time (or consist of information properly combined over a period of time). They cannot, however, form a representation that includes information about inputs that the network has recently seen, without modifying the weights through training. Recurrent neural networks can accumulate information over time, however. Thus they are natural choices for application where the data is best viewed as a timeseries. Practical applications include speech and text processing, music identification, and even handwriting recognition when sequence of the actual penstrokes is known (Cho et al., 2014, Greff et al., 2015)

The most basic recurrent network layers simply consist of fully connected layers where the outputs are connected back to the units in that layer, as well as being used as inputs to the next layer. However, many modern recurrent layers make use of gating mechanisms to control what gets output from each unit. These gates are sigmoid functions of the inputs (recurrent and non-recurrent) to each unit, or cell. The first such recurrent architecture to be proposed was the Long-Short Term Memory(LSTM) networks proposed by Graves and Schmidhuber (2005).

A LSTM cell contains three gates, along with a cell state. The three gates, know as the *input*(i), *forget*(f) and *out*(o) gates are computed as follows, where σ is the standard sigmoid function, with a range between zero and one:

$$i = \sigma(W_i * x_t + U_i * h_{t-1} + w_{ci} * c_{t-1} + b_i) \tag{2.1}$$

$$f = \sigma(W_f * x_t + U_f * h_{t-1} + w_{cf} * c_{t-1} + b_f) \tag{2.2}$$

$$o = \sigma(W_o * x_t + U_o * h_{t-1} + w_{co} * c_t + b_o) \tag{2.3}$$

The $W * x$ and $U * h_{t-1}$ terms are vector multiplications between the weights and in incoming and recurrent connections respectively. The $w * c$ terms are scalar multiplications that are used to include the cell state in the calculation of the gate values themselves. These connections are known as *peephole* connections, and some or all are left out of some LSTM variants. It should be noted that if peephole connections are used the outgate (Eq. 6.3) uses the cell state for that timestep. The input and forget gets, which are used to calculate the cell state, get their peephole connections from the cell state at the previous timestep.

The new cell state is computed using the input and forget gates as follows, where the input gate controls the contribution of the incoming and recurrent connections to the new state, while the forget get determines the contribution of the previous timestep's cell state:

$$c_t = i * tanh(W_c * x_t + U_c * h_{t-1} + b_c) + f * c_{t-1} \tag{2.4}$$

The cell's actual output is then determined by taking the *tanh* of the state and

scaling it by the output gate:

$$h_t = o * tanh(c_t) \tag{2.5}$$

While LSTM networks have proven effective on a wide range of tasks, the model is complex, and each gate adds a potentially large number of parameters to train. This is especially true as the number of cells in a layer increases, as the number of recurrent connections is N^2, where N is the number of cells in the layer. A newer type of gated recurrent layer, known as Gated Recurrent Units (GRUs) reduces the number of parameters by reducing the number of gates by one, and eliminating the cell state (Cho et al., 2014).

The remaining two gates in the GRU model are known as the *update*(z) and *reset*(r) gates, and are computed as follows:

$$z_t = \sigma(W_z * x_t + U_z * h_{t-1} + b_z) \tag{2.6}$$

$$r_t = \sigma(W_r * x_t + U_r * h_{t-1} + b_r) \tag{2.7}$$

Like the LSTM cells, the *tanh* function is then used in the calculation of the cell output, however the GRU cell determines it's output directly from the gates and the previous output, with no need to maintain a seperate state value:

$$h_t = z * h_{t-1} + (1 - z) * tanh(W_h * x_t + r * (U * h_{t-1}) + b_h) \tag{2.8}$$

It should be noted that the $z * h_{t-1}$ term is a scalar multiplication between the update gate and the previous timestep's output for *that* cell, while the $U * h_{t-1}$ term is the dot product between the recurrent portion of the weight matrix and the previous timestep's output from all cells in the layer. Thus the intuition behind the update and reset gates is that the update gate determines how much the previous timestep's output for the cell determines the new output directly, compared to the influence of the connections coming into the cell. The reset gate on the other hand determines how much influence the recurrent connections have on the incoming connections' part of the output.

2.2 Computing hardware and deep learning

While, historically limitations created by contemporary compute hardware hindered the development of neural network algorithms, in recent years neural networks and deep learning have driven the development of algorithms and even new hardware architectures to facilitate more efficient training and evaluation of neural networks. This includes both efforts to simply speed up the matrix/vector multiplication inherent in evaluating all neural network layers, especially fully connected and recurrent layers, and algorithms and hardware to specifically speed on convolutional layers. It is convolutional layers that I will focus on most specifically here, but with respect to algorithms for their evaluation on standard CPU and GPU hardware, and on hardware architectures designs specifically for neural network computations.

2.2.1 Implementation of Convolutional Networks across Hardware Architectures

Due to their popularity and use is a wide array of research and production applications, many optimized implementations of ConvNets have been developed for different architectures, including standard x86 CPUs, GPUs and other prototype hardware architectures. Many of these implementations are part of deep learning tools such as Theano/Lasagne (Al-Rfou et al., 2016), Caffe, or TensorFlow. These tools often break up networks into modular layers, such as a convolution layer or a recurrent layers. Layers of a specific type have meta-parameters for their size, training options etc, and allow for the architecture of the network itself to be specified with limited consideration for the actual implementation or even the hardware it will be run on. It is the implementation however, that I am concerned about in the first part of my work. Here I will discuss implementations that others have developed, for x86 based CPU architectures, GPU architectures and novel, deep learning specific hardware.

2.2.2 x86 Implementations

The most naive approach to implementing a convolutional or recurrent network is to simply calculate the output to each layer serially, performing single multiply-add operations between each weight and the appropriate input value(s). While this might

yield code that is easy to understand and debug, it does not take full advantage of the capabilities of modern CPUs. Even if the work were parceled out such that each core is performing calculations, the SIMD capabilities of the cores are ignored, with only one multiply-add operation be performed at once as opposed to the eight to 16 such single precision floating point operations that can be performed simultaneously on many modern Intel CPUs (Intel, 2021). Common compilers, including Intel's C/C++ compilers and the open source `gcc` and `g++` compilers, have optimization options that can attempt to allow code to automatically make use of these SIMD capabilities, and many libraries exist with optimization implementations of common mathematical operations.

For a fully connected neural network layer it is simple to take advantage of this SIMD capability, through the use of these optimized libraries. Assuming that the weights are arranged properly the bulk of the computational load can be performed as a simple matrix dot product, with an element wise addition for the bias terms. These are both operations for which there are many well optimized libraries to perform, such as the well known BLAS library. The connectivity of more complex network layer times is not this simple, however any many cases it can still be transformed into a matrix operation. Likewise, the primary computations for simple fully connected recurrent layers can be broken up into the sum of two dot products, one for the layer input, and the other for the recurrent connections. For gated recurrent networks the bulk of the computation comes from multiplying the values from the incoming and recurrent connections to compute the values for the gates and cell state or output. In the case of an LSTM layer, this means calculating the product of the same incoming and recurrent inputs with weights for the in, forget and out gates along with the cell state (Hochreiter and Schmidhuber, 1997). All of these values can be computed at once using a similar sum of dot products strategy.

Unlike many basic and recurrent neural network layer types, one key feature of convolutional networks is that they are not fully connected (LeCun et al., 1998). This however means that more complex manipulations are required to formulate the layer's computations in a way such that they can be carried out using a small number of standard matrix computations. One such method is proposed by Chellapilla et al. (2006). This method duplicates the input values to create a matrix where each row contains all of the values needed to compute a given output. The weight values

are then arranged appropriately, with no duplication, to allow the majority of the computations needed to compute the layer output to be performed as part of a single dot product.

2.2.3 GPU implementations

While the matrix method of performing convolutions can was developed to take advantage of optimized matrix multiplication libraries and SIMD capabilities of modern CPUs the same method can also be used on GPUs. One such implementation is cuDNN (Chetlur et al., 2014). Another method developed for implementing large convolutional networks on GPUs involves the use of an Fast-Fourier Transform (FFT) (Mathieu et al., 2013). The FFT transform is used to transform a layer's filters and inputs from 2d spatial images to the frequency domain, where the the actual convolution operation can be performed with a single step. Like the duplication of data in the matrix method, the FFT transform incurs extra overhead, however since the transformed values can be reused the overhead is made worthwhile by the savings in computational operations across all computations performed.

2.2.4 Custom and experimental hardware implementations

Besides optimized CPU and GPU implementations, a third route for implementing neural network algorithms is to use hardware that was designed primarily for running neural network algorithms. While still much less flexible in their capabilities than a modern CPU or GPU some of this architectures have been designed for the general types of calculations performed as part of neural network algorithms, and posses the flexibility to implement a wide range of neural network algorithms.

In their DianNao architecture, Chen et al. (2014) focus on such an approach, implementing a SIMD architecture that is designed with the memory locality structure of convolutional networks in mind, maximizing the usage of data elements that have been fetched from RAM. They take advantage of the structure of input (x,y, channel) and of the 2D convolution to inform arrangement of data in memory and order of operations and accomplish this by having a circular input buffer that feeds the data to the SIMD unit. This buffer is filled with input values from different channels but the same x, y location, and all calculations that need this values are accomplished and

stored in another buffer before the initial set of inputs are replaced with a new one. Their hardware design is somewhat unusual, however, in that it broadcasts the input channels the 16 inputs that it can handle at once to all 16 arithmetic units, while the 16^2 total weight values read in together are divided up among the arithmetic units. The focus behind this scheme is memory reuse. It achieves reuse of the data values, as multiple partial calculations that use the same X,Y locations are completed before that data is removed from the buffer nd replaced with new data. The temporary storage registers for the partial sums do add additional complexity to the hardware however, and the number of such buffers places an upper limit on the number of partial sums that can be stored, and by extension limits data reuse.

A successor architecture to DianNao, Cambricon (Liu et al., 2016), addresses some of the flexibility concerns of DianNao by implementing more general matrix operation primitives. However, both proposed architectures are limited in the width of their SIMD capability and would likely not scale well. This means that a higher clock speed would be required to get the same throughput as more highly parallel methods.

Another novel design, proposed by Han et al. (2016), focuses specifically on accelerating deep networks with sparse weights. This proposed architecture, the Efficient Inference Engine, consists of many, relatively small SIMD compute units. The authors claim improvements in inference speed and power consumption compared to CPU or GPU based inference algorithms. However, this speed and efficiency is achieved by putting strict requirements on the sparsity of both the data and the weights, and by constraining the final weight values of the network to belong to a small set of discrete values. While this might work for some networks and result in a very efficient hardware design for certain situations it is unlikely to work for all networks, limiting the architecture's usefulness as a general deep learning inference engine.

2.2.5 TPUs

Another major hardware architecture that should be mentioned is Google's Tensor Processing Unit (TPU), as it has seen production use as a data center based device designed for neural network inference. The TPU utilizes pipelined CISC instructions that allow for many clock cycles to by spent in each pipeline stage. Additionally, the pipeline is designed such that data movement operations and auxiliary post-processing operations can be performed simultaneously with the main fused-multiply-add (FMA)

workload, preventing these background operations from delaying the arithmetic operations that need their results whenever possible (Jouppi et al., 2017).

The primary arithmetic operations are either $8b \times 8b$, $8b \times 16b$, or $16b \times 16b$ integer multiplies added to a 32b accumulator, with wider operations taking multiple clock cycles. The FMAs are performed in chunks of 256 elements at a time inside a 256×256 systolic array *Matrix Multiply Unit*, and once they leave the Matrix Multiply Unit the accumulators' values can be stored for later or sent to a separate hardware unit that handles activation functions and similar post-processing operations (Jouppi et al., 2017). While custom ASICs could previously be tailored to a specific neural network workload, the TPU's neural network specific design secures is a first as a drop in, neural network specific hardware solution for use in production systems.

2.3 AVX-512 Overview

However, just because solutions exist that are specifically targeted for neural network inference exist doesn't mean that general purpose hardware shouldn't be considered - especially since most neural networks are part of a bigger system made up of more general code. Thus, the final SIMD paradigm that deserves more through discussion is Intel's AVX-512 instruction set, as it is built in to many modern high end Intel server and consumer grade CPUs (Intel, 2019), and has many open source libraries that take advantage of it, including tensorflow (Abadi et al., 2015). unlike the previously mentioned architectures, the AVX-512 instruction set was not built with neural network workloads specifically in mind, but instead builds on a history of SIMD technology developed for Intel x86 CPUs for multimedia, communication and general purpose SIMD operations. This history begins with the MMX instruction set extensions introduced on Pentium II CPUs, which provided integer operations on a 64b vector width, mostly targeted at multimedia and internet communications applications. Later improvements to SIMD functionality came with the Streaming SIMD Extensions (SSE) which introduced 128b wide SIMD vectors along with floating point support. This expanded vector width then grew to 256b with the introduction of Advanced Vector Extension (AVX and AVX2) instructions, which also introduce fused multiply-add capability which is commonly used in neural network applications (Intel, 2021). Finally, the AVX-512 instruction set increased the SIMD vector width

to 512b, being released initially as a specialized many-core compute device in the first generation of Xeon Phi, known as *Knight's Landing*. AVX-512 was brought to standard CPUs designs with the Skylake-X consumer focused line, and as of 2022 can now be found in most recent Intel Core (consumer market) and Xeon Scalable (server) CPUs, although it's not available on the latest 12th generation "Core" CPUs (Intel, 2019).

2.3.1 Changes from previous AVX instructions

The most obvious change between AVX-512 and previous AVX instructions is the increase in SIMD width. However, AVX-512 also follows the trend of increasing register space for SIMD instruction with each generation of SIMD capability. Specifically, while MMX instructions introduced only 8 64b SIMD registers, the SSE instructions increased this to eight larger 128b *xmm* registers could be split into 16 64b for SSE/SSE2 instructions. The original AVX instructions introduced the 16 256b *ymm* registers, which doubled in both number and size for 32 512b wide *zmm* registers used for AVX-512 instructions (Intel, 2021). This increased register width allows for a greater amount of data to be kept immediately available for use. Additionally, while previous SIMD instruction sets have had a few iterations (e.g. SSE, SSE2, etc), AVX-512 consists of multiple groups of instructions in addition to the original "Foundation" instructions (AVX-512F) (Intel, 2021). While it is possible to detect which of these instruction groups are supported on a given CPU, this ambiguity as to what constitutes AVX-512 makes generalized development more difficult, especially since some the instruction groups include things like 16b floating point support that are potentially useful for neural network applications (Intel, 2021).

2.4 Ultra-wide SIMD hardware description

Finally, I will describe the proposed ultra-wide SIMD architecture for which the algorithms described in this document were developed. This proposed architecture, known as *ncore*, trades off the ability to perform a wider array of data movement operations such as those available in AVX-512 for the ability to do as many simple arithmetic operations at once as possible, with SIMD widths an order of magnitude wider than the TPU. This section describes this hardware design and the philosophy

behind it.

2.4.1 New Ultra-wide SIMD Hardware

Relatively narrow SIMD operations, such as the 128, 256 and 512*b* wide SIMD operations provided by current or upcoming x86 ISAs provide a larger amount of flexibility in their use, since they operate on a relatively small amount of data at once, compared to the size of many matrix operations used in deep learning applications. Architectures with a higher degree of parallelism offer the opportunity to complete a larger chunk of a matrix operation within a single instruction. New implementations of deep learning primitives however are likely needed in order to take advantage of of the greater parallelism offered by such a new architecture. Here I will describe one such proposed ultra-wide SIMD architecture, in order that I can define new methods to use the architecture for deep learning calculations. The hardware was developed by a team that I worked with at Centaur Technology in Austin, TX, and is the hardware for which I will develop implementations of convolutional and recurrent deep network algorithms. In order to aid in the understanding of my implementations I will first describe the basic functionality and unique aspects of the hardware on which it will be run.

2.4.2 Hardware and Instruction Set Architecture (ISA) Overview

The primary building block of this proposed hardware architecture is a hardware "slice" concept outlined by Henry (2020), consisting of memory, a *Neural Data Unit (NDU)*, a computational unit (*Neural Processing Unit (NPU)*) that can take two input values and perform at least the limited set of arithmetic operations needed for common neural network architectures, including a large number of multiply accumulates, and a output/post-processing unit. These are arranged to form a deterministic pipeline, where each stage feeds the next while allowing data movement and post-processing operations to take place simultaneously with the basic addition and multiply operations that make up the bulk of neural network computations.

The main pipeline begins with the data movement stage, the NDU, that controls the NPU inputs, allowing for inputs from memory registers, previous outputs, and operations that allow the previous operation's input(s) to a unit to be be sent to an

adjacent unit at a given distance and direction within a relatively smaller number of SIMD units, either as a rotate, local broadcast, or similar operation. This movement capability essentially makes a logical ring out of the processing units, facilitating local data movement without the overhead of large scatter/gather operations, but still allowing data to (eventually) reach any location across the SIMD width using a sequence of rotations, but without changing the relative order of the data elements.

The data movement stage feeds the NPUs, where each computational unit includes an arithmetic unit that performs FMAs and related operations, and an "accumulator" register that is used to store the intermediate results of such sequences of simple operations. The result of a string of these arithmetic unit operations can then be sent to an Output/Activation unit, similar to that in the TPU, where a provided function (e.g. `tanh`) or other post-processing step can be applied to the accumulator value before it is either be written back to memory, or used as input to further arithmetic operations.

This simplest such network layer that one can evaluate on this architecture is a fully connected layer whose input and output sizes are less than or equal to the SIMD width. Using this architecture the output of a single such layer can be computed as follows:

1. The input to the network along with the first row of the weight matrix are loaded from cache/memory.

2. Prior to performing any calculations the accumulator registers of all computational units are set to zero.

3. Element-wise multiplication is performed between the network input vector and the first row of the weight matrix, and the result added to the computational units' accumulator registers.

4. The values from the input vector are then shifted, such that the value that was used by computational unit i for the last multiply-accumulate operation is transferred from the input register of unit i to computational unit $i + 1$. This value is then used as an input to the next element-wise multiply-accumulate operation, along a new row of weights fetched from memory.

5. This process is repeated until all elements in the input vector have been provided

to each computational unit along with the appropriate weight values. At this point the accumulator register of each computational unit holds the weighted sum of the inputs one of the layer's neurons.

6. The bias value is then added to each computational unit's accumulator register as a single SIMD operation.

7. The appropriate activation function is applied to the contents of each computational unit's accumulator register, and the result written back to memory.

In fully connected layers where there are more inputs to the layer than can be fit across a single SIMD row the additional inputs can be placed in another row. Steps three through five are then repeated for each SIMD row of input data. Likewise, layers that have more neurons in them than computational units in the hardware can be computed be repeating the whole process M times, where $M = \frac{ceiling(layer_{size}}{computational_{units})}$.

While the above example uses the values in the computational units' accumulator registers to write the layer output back to memory, as state previously, the capability to use a function of these registers' contents directly as an input back into the computational units is also provided. This functionality is important for efficiently implementing recurrent neural networks. To further facilitate the implementation of gated recurrent networks limited ability use the computational units' output as input to adjacent computational units is also provided.

2.4.3 Motivation for Ultra-wide Paradigm

While the high-level goal driving the design of ncore was performance per total system cost due to its nature as a on-die co-processor, the low-level design was driven by multiply-accumulate (MAC) efficiency, and design scalability (Henry, 2020). Specifically, multiply-accumulate (MAC) efficiency meant that both the hardware and software algorithms were designed to maximize the usage of the main arithmetic units each clock cycle, whether fused multiply-add (FMA) operation, or other simple arithmetic operations that could reuse most of the hardware components of the FMA units. The repetition of this simple hardware design in the slice concept described above allows for groups of identical arithmetic units along with their supporting data movement and output functions to be repeated across the SIMD width as a larger

hardware module, simplifying the hardware design process and allowing the SIMD width to be changed by simply changing the number of these modules.

While at a high-level ncore's pipeline is similar to the TPU, this design highlights a major difference in low-level design. Namely, the organization of ncore around a simple, repeated "slice" reduces the size of the hardware design problem, simplifying the process of design and testing, allowing for a new version of the hardware to be easily designed with a different SIMD width as dictated or allowed by hardware technology or application requirements, and potentially allowing more compute units on the same physical chip by facilitating optimization. Plus, some would argue that such simpler design paradigms are less vulnerable to the ever increasing scourge of security vulnerabilities (Henry, 2021).

2.5 Machine Learning and Decoding Neural Signals

The reason that so many hardware solutions have been developed for neural network training and/or inference is that these algorithms have proven themselves useful in a large array of fields. Well known examples include computer vision and speech processing, but neural network algorithms can be used many fields where sensor data or other similarly large volume input needs to be interpreted or have a label assigned to it. One such field is computational neuroscience and the interpretation of observed neural signals.

Neuroscience researchers utilize a wide range of methods for observing, either directly or indirectly, the activity of neurons. These methods range from invasive methods for directly observing the activity of small numbers of individual neurons in animal studies, to non-invasive methods, such as fMRI or EEG, that allow for the observation of the entire human brain, albeit at a much coarser temporal and/or spatial scale. The types of data generated by these observations varies. However, almost all methods yield large amounts of data whose interpretation often requires powerful analysis tools.

2.5.1 Machine Learning Techniques for Neuroscience Time-series Data

Some such analysis methods focus on inferring other aspects of the data such as structure, as in (Huth et al., 2015) for example. However, here I will focus on situations where some form of classification or regression is performed on the data, often to infer something about the stimulus that drove the observed neural activity, or the behavior(s) of the organism that that resulted from it. My primary focus will be on the challenges of inferring the information encoded in spikes observed from individual or groups of neurons. However, I will also discuss techniques used for other modalities of neural data, as most such methods share the challenge of extracting meaningful information from a noisy and incomplete observation of the brain.

2.5.2 Simple Classifiers

Perhaps the simplest method of quantifying the relationship between the spikes observed from a neuron or group of neurons in correlation to a stimulus that has been presented to the system is to simply count the number of spikes within a specific time period after a given stimulus is presented. These spike totals can be collected for a single neuron repeatedly as some parameter of the stimulus is varied. For example Hubel and Wiesel (1962) used simple bars of varying angles to map the receptive fields of cells in the cat visual cortex.

While this spike counting approach is extremely simple, counting spikes over a long period relies on the assumption that any information encoded by changes in spiking within that period is not relevant. Foffani and Moxon (2004) expand this concept to build a histogram of observed spikes over multiple time bins during and shortly after the period in which a given stimulus is input into the system. These histograms can then be used to create a simple classifier that can infer which a stimulus label for a new histogram of spikes. A smooth version of a PSTH histogram can also be created by convolving a Gaussian kernel with a vector called a *spike train*, which contains a one at all points in time where a neuron was observed to be active, and a zero at all other locations. While this method, similar to that employed by Smith et al. (2013) in their analysis of neural responses in the primate auditory system, helps remove noise from the signal of observed neural activity further processing is needed to be

ability to decode or quantify information about the stimulus that drove the observed neural activity.

2.5.3 Machine Learning

Smith et al. (2013) use principal component analysis to reduce the dimensionality of their histograms of neural activity and then use a linear discriminate analysis (LDA) classifier to decode the stimulus that drove the activity. Similarly, Support Vector Machines have been used to expand the categorical classifier into a multidimensional space.

2.5.4 Neural Network Algorithms

Besides direct classification on the observed data, machine learning tools that build a lower dimensional representation of their input can also be used to transform the stimulus thought to have evoked observed neural activity input a form that can more easily be related to that activity. For example, Agrawal et al. (2014) utilize convolutional neural networks to predict voxel intensities for fMRI data for vision related brain regions ranging from V1 to higher level visual areas. They do this by presenting the stimulus images used in the fMRI study to a ConvNet trained on the ImageNet dataset. A feature vector can then be extracted from the intermediate layers of the network that can be used to predict the observed voxel intensities using a simple regression method.

Modern deep learning methods developed as machine learning tools are far from being faithful models of real neurons. However, their simplified nature makes them tractable models of the brain that still can be valuable in some cases. One obvious area where deep learning networks designed for machine learning problems might still be helpful for modeling the brain is for low level vision. One surprising property of even early deep convolutional networks trained on natural images is their tendency to learn filters in their initial layers that resemble the receptive fields of cells in the primary visual cortex (V1). Furthermore, Lee et al. (2007) show that higher layers of some deep networks can learn filter basis functions that resemble receptive fields found in a subsequent layer of the visual cortex (V2).

It is this tendency to learn representations at lower network levels that are sim-

ilar to known receptive fields at lower levels of sensory processing in the brain that make them good candidates for generating feature vectors and representation for applications such as that of Agrawal et al. (2014). Furthermore, this tendency of deep networks to mimic the known receptive fields of actual cells in the brain extends beyond convolutional networks and the visual system. For example, Kanitscheider and Fiete (2016) use LSTM networks as a model of areas of the hippocampus used for navigation. Similar to computer vision networks trained on object recognition from input image, these recurrent networks are trained to predict location in a simulated environment using inputs similar to those believed to feed into regions of the hippocampus responsible for navigation tasks. The recurrent cells in the hidden layers networks display a tendency to learn "receptive fields" similar to those observed in actual cells in the hippocampus.

The task of decoding observed neural activity related to a sensory task in the brain is different from using a classifier to perform a similar task. However if you consider sensory processing in the brain to be a phenomenon that occurs in stages from input to decision or behavior, then decoding neural activity observed at an intermediate stage can be thought of as designing a classifier to mimic the functionality of subsequent stages. Such a classifier need not necessarily process the the information in the same manner as these later stages of processing in the brain. However, the fact that deep networks tend to learn similar representations to those in the brain, with the representations at subsequent deep network layers resembling subsequent known stages over sensory processing, makes these networks a good candidate for mimics the function of the later stages for such decoding applications.

2.5.5 Timeseries Algorithms for Neural Data

Some modalities of neural data, such as the fMRI study discussed above, have low temporal resolution. In such cases algorithms that take temporal information into account are less likely to be useful. However, other modalities of measuring activity in the brain, such as EEG or MEG, or the observation of the actual spiking activity from individual cells, has very high temporal resolution. In the case of data from such an observation modality, using models or algorithms that take into account time and/or frequency, which describes the variation of a signal over time might be more useful.

Buzsaki (2006) mentions the dichotomy between time and frequency domain measures of a signal, and compares the analysis of neural timeseries to the analysis for speech or other acoustic timeseries where information from both the time and frequency domains are important. LSTMs address time directly, frequency less so, but speech and music processing are common applications of recurrent neural networks (Greff et al., 2015), showing their potential to handle such information. Furthermore, LSTMs have been applied to decoding attention or emotion information from EEG data (Soleymani et al., 2014). This shows that not is the application of LSTMs to EEG data feasible, but that at least in some cases problems such as noisy training sets of limited size are not barriers to the use of recurrent deep learning algorithms.

Chapter 3

Convolutional Network Computations on Ultra-wide SIMD Hardware

Many algorithms already exist for implementing convolutional network layers on GPU hardware, or traditional CPU SIMD architectures. However, due to the unique features and limitations of the previously described ultra-wide SIMD hardware, I have implemented a new algorithm to compute the output of a convolutional network layer that capitalizes on the strengths of this architecture and avoids its weaknesses. It should be noted that this algorithm is only concerned with the forward pass through the layer. However, it could be used as part of a training algorithm, with additional steps to perform the backwards weight update step which would involve more traditional CPU instructions. However, the main goal of the algorithm is to facilitate fast inference on new data by a previously trained network.

In the following sections I will describe how the input data and weight values are laid out in memory to be used by the algorithm. Then I will describe the algorithm itself, along with some extensions and modifications to maximize its efficiency across a wide variety of network sizes and variations on the standard convolution algorithm.

3.1 Insight

By its nature, convolution is a spatially local operation - to generate each output a few filter coefficients are convolved with a few, adjacent elements of the input. Moreover, spatially adjacent outputs have similarly adjacent input windows in all dimensions. Thus, if the data is arranged correctly, convolution can be performed using only (relatively) local movements of data within a SIMD row. This design facilitates the implementation of an efficient convolution operation on SIMD hardware that does not support scatter/gather operations across the full length of the row.

For convolution with a single channel of input this algorithm is straightforward. Assuming the ability to shift data right by one element within the SIMD row, a row of the input tensor begins lined up with the element of the output row for which they are the rightmost input. These values are then multiplied by the rightmost fil-

ter coefficient for the first row of the filter and the input rotated right for the next filter element until the first row of the convolution is complete. This process is then repeated with new inputs for each horizontal row of the convolution filter.

In practice however, deep learning convolution operation neither involves one channel of input, or is the input normally even close to filling the element width of the proposed SIMD architecture. This observation about the data leads to the second main insight of my algorithm: the division of the SIMD row into equal width blocks. The input tensor is then flattened such that row n and channel a of the input tensor is initially placed in the first block of the input row, channel b of row n in the second block, and so forth. This design differs from simply flattening the input matrix in the row and then channel dimension, in that a small number of pad elements may be inserted between each channel in order to maintain even number of equal size blocks across the SIMD row, and allows for a slightly larger, but still local relative to the total SIMD width, rotation to be used to line each block of input up with a new combination of output elements.

Thus, by preserving the original geometry of the each input data, but packing the rows of each channel to maximize the amount of data stored in each SIMD row, convolution can be performed efficiently while performing only local data movement operations within the rows. This allows the algorithm to be run on hardware that does not support global scatter/gather type data movement, or where such operations are inefficient.

3.2 Algorithm

The insights outlined above provide the framework for describing both a novel algorithm, known as M^3inM^2VConv, for performing a multi-channel convolution on the ultra-wide SIMD hardware in question, as well as a data format to facilitate this algorithm. Since the algorithm requires that the data be in a specific, Geometry-preserving layout, I will describe this layout before describing the algorithm itself. Also included in the description of the layout are methods for padding the data to address the fact that unless a network was designed with performance on the particular hardware in mind, it will likely not fit perfectly within the hardware's SIMD width.

Once I have established the required data layout I will describe the algorithm itself. Once I describe its canonical form I will explore the modifications required to support several variations on a standard multi-channel convolution algorithm. These variations include depthwise, strided and dilated convolutions.

3.2.1 Layout of input data in cache memory

Unlike the matrix multiplication method for computing the output of a convolutional neural network layer on SIMD hardware that was proposed by Chellapilla et al. (2006), my method does not duplicate the layer's data input values. Instead, I leave the inputs in their original 2D form, with each row of the input occupying a different SIMD row. While these SIMD rows could be laid out in physical memory in any manner that makes sense for the hardware, it is helpful to think of these SIMD blocks as being laid out in a matrix whose number of columns is equal to the SIMD width and whose number of rows is equal to the number of SIMD blocks that can be fit in the cache or other memory allocated to the network. Visualizing the memory in this way, the input values to a layer are laid out in memory in the same manner as they would be if they were pixel values in an image being displayed on the screen.

Unfortunately, the width of the inputs to most ConvNets layers is less than the SIMD width of the proposed hardware (for example AlexNet, a successful ConvNet model utilizing the ImageNet dataset, takes three channels of 224×224 input (Krizhevsky et al., 2012)). Obviously if you had a SIMD width of 512, or even greater, it would be inefficient to use an entire SIMD row for each row of a single channel's input.

To address this issue I introduce the concept of a block, which I rely on heavily in my convolutional network layer algorithm, Specifically, to allow for the utilization of memory elements across the entire SIMD width the memory words are divided up into blocks, where the width of a block is the smallest power of two that is wide enough to fit the input width. For example, for a 224 pixel wide input and a 512 word wide SIMD row two blocks of width 256 can be used, though some padding elements are still required if the input width is not exactly a power of two. In the case of the input of width 224, $256 - 224 = 32$ pad elements would be needed within each block. Thus, in this case one 512-element-wide memory row could hold one row of the red and green input channels of an ImageNet example. In the most basic version of this

scheme the corresponding input row from the blue channel would then be stored in its own memory row, where only one of the two 256 word blocks are used. If hardware's SIMD width were wider, say 1024 elements, then all three channels would fit across one set of SIMD rows, with a 4th block of 256 elements that would remain unused,

In general, if the SIMD row is divided into k blocks based on the width of the input to the layer then the input data from up to k channels from that layer can be fit in h memory rows, where h is the height of the layer's input. This principle is illustrated in the top row of blocks in Figure 3.1, highlighted in purple.

Figure 3.1: Basic ConvNet memory layout for both data and weight values. The first number in each green weight block is the filter group number, and the second number is the input channel number whose filter weights are stored in those SIMD rows. This layout arranges the data in a form to efficiently feed an implementation of the convolution algorithm that takes advantage of the hardware's strengths, while minimizing its weaknesses.

It should be noted that within these blocks of size k, it might be the case that not all elements within the memory row contain values that will be used in the calculation of the layer's output. Specifically, if the actual width of the input is less than k then

there will be some columns of "pad" elements within the block of h memory rows used for the k channels of input data. This block layout aligns both the data and the overall structure of the convolution to the structure of the hardware. As such it's use will be continued for the convolution's other data and computational elements.

3.2.2 Layout of weight values in cache memory

In order to be able to not duplicate the input data elements to facilitate computation of the convolution layer output the weight values do have to be duplicated. However, the duplication of the weight coefficients offers an advantage over duplicating the data values, in that as long as the memory exists to store these duplicated values the work of duplicating the value only needs to be done once, after which an unlimited number of inputs examples can be processed.

Like for the input data, the SIMD rows that store the weight values are divided up into blocks. However, instead of one input channel being assigned to each block each block is assigned values from one filter group. However for a given SIMD row, each block only contains one weight value, duplicated r times, where r is the width of the layer's output for that filter group. Thus for a single channel filter of size $m \times n$ blocks from $m \times n$ SIMD rows are needed to hold the weight values.

While it is uncommonly seen in real networks, the simplest case weight layout is one where each filter group only receives one input channel. Here the filter groups are assigned to blocks such that each filter's weight values are assigned to the block in the same position in the SIMD row as the block containing the input data for that filter group's channels. In the more common case where there are multiple input channels, each filter group receives all channels as input. Assuming each filter group is of size $m \times n \times c$, where c is the number of channels, then in this case the first $m \times n$ rows contain weight values from the channel whose input values are aligned in the same block. There are then $c - 1$ more sets of $m \times n$ SIMD rows, where the location of each channel's filter weights are rotated by one additional block after each group of $m \times n$ rows. The green regions of Figure 3.1 show this layer for k filter groups, each with k input channels. The reason for staggering the weight values in this manner is to take advantage of the rotational ring of registers to allow the single copy of each element in the layer input to be lined up with every arithmetic unit that needs it.

3.2.3 Convolution Operation

Using this block paradigm for the structure of the input data, a description of the algorithm to perform the actual convolution arithmetic can be described. Keeping with the block motif described above, the implementation of the convolution operation for the previously described ultra-wide SIMD hardware leaves the layer inputs in their original 2D form to perform the convolution. Each row of the convolution output is then computed all at one time, shifting the convolution input around the ring of registers on the inputs the arithmetic units until each value has been provided to all arithmetic units that need it. Figure 3.2 shows an example of this operation with a single block, for a single input channel. The other blocks across the SIMD row can be performing the same convolution simultaneously for other filter groups, as the register shift operation is shifting the values around across the entire SIMD row. The first three panes show the computations for the first row of the 3×3 filter, while the fourth pane shows the first computation for the second filter row. Note that the rightmost column of each filter row is handled first, and then the data shifted right in the registers. Once the multiply-add operations for all columns in all rows of the filter have been performed the result in the arithmetic units' accumulators can can be written back to memory and the computations for the next row of the convolution output begun.

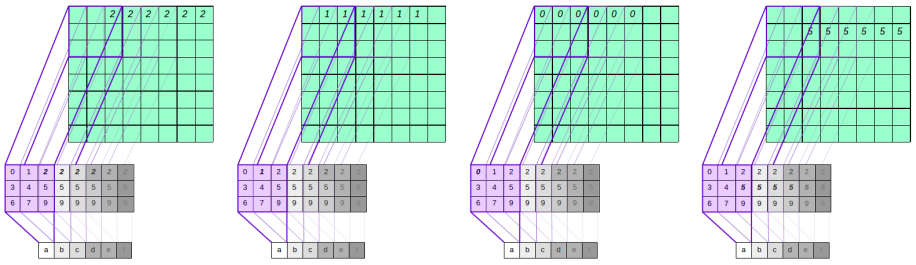

Figure 3.2: Example of part of a single block of the convolution operation. Note that all columns in a single row of the convolution output are being computed simultaneously. The same filter value (green) is used for all arithmetic lanes in a given step of a convolution. This filter element is highlighted in bold and all data inputs that are convolved with this value are marked by its index number. Performing the convolutions in this manner, rather than reshaping the input data, avoids the need to perform data shuffle operations during network evaluation, something that the proposed hardware does not support effectively.

It should be noted that the convolution represented in Figure 3.2 is a *valid* convolution in that the filter never hangs off the edge of the image. Often however a *same size* convolution is used, where the filter is allowed to "hang off" the edge (or the image is padded with zeros), this implementation produces a convolution output of the same dimensions as the input. The convolution method described in Figure 3.2 can be easily modified to support same size convolutions, either by padding the input with zeros or by replacing the filter coefficients with zero valued weights when they don't line up with an actual input value. The latter method is preferable when the width layer input size is close enough to the power-of-two block size such that adding zero pad values to the actual SIMD memory row would necessitate the use doubling of the block size, reducing the number of filter groups that can be processed simultaneously across the SIMD row.

Finally, this simple description of the convolution operation assumes that each filter group has only one channel of input. This is rarely the case in reality. However an extension for handling multiple channels is relatively simple. Assuming that the input data width is such that the SIMD row is divided into as many blocks as there are input channels, then after the computations for the current channel for each filter row the input data is shifted further around the ring of input registers until it is lined up to with a new filter group block. The computations for that row of the filter and the new channel are then performed. This process is then repeated until all channels have been processed for that filter row. At that point new data is loaded into the input registers from memory and the process repeated for the next row of the filter.

This is the reason for the staggered layout of the filter coefficients by channel in Figure 3.1. For ease of visualization all rows for a given channel from each filter group are grouped together. However in reality the only the first n rows of the $m \times n$ SIMD rows holding filter weights for each channel are used, corresponding to the n columns in the first row of the filter. Once all channels in the first row of the filter have been processed the next groups of n SIMD rows will be used, corresponding to the columns for each channel in the next row of the filter.

The hardware assembly instructions for this convolution operation starts out similarly to the computation to a fully connected layer, in that an initial multiply-accumulate is performed between a SIMD width memory row containing data values and the row containing the first set of weights for the layer. However, unlike the

algorithm for computing the output of a fully connected layer, there is no need to line every input up with every computational unit. Instead, the basic premise of my convolution algorithm is to divide the computational units into blocks corresponding to those used to store the data and weights in memory. A single row of output to a convolutional layer is then performed as follows:

1. Set the values in all accumulator registers to zero.

2. Load the first (or next) SIMD row of input data and on weight values into the arithmetic units' input registers, and perform the multipy-add operations of the first step of the convolution operation. On the first iteration this is the convolution of the first row and column of the first channel of each filter group assigned to a block in a SIMD row's worth of blocks and is the beginning of the next row on subsequent iterations.

3. Shift the input data over one element in its register, load the next SIMD row's work of weight values, and perform the next multiply-add operation. Repeat this until all columns for the current row and channel within the current SIMD row's worth of filter groups have been processed.

4. *If unprocessed channels for this row remain:* Shift the data in the registers containing the input data such that the channels in each block of input data are aligned with a new filter group's block of accumulators. This may require additional clock cycles beyond that required for the first multiply-add with the channels aligned with the new filter.

5. *If unprocessed channels for this row remain:* Load the new SIMD row's worth of weight values, for the first column of the same filter row for the channel now aligned with each filter groups.

6. *If unprocessed channels for this row remain:* Shift the input data over one element in its register, load the next SIMD row's work of weight values, and perform the next multiply-add operation. Repeat this until all columns for the current row and channel within the current SIMD row's worth of filter groups have been processed.

7. *If unprocessed channels for this row remain:* Branch back to step 4, *Else:* continue.

8. *If unprocessed rows remain in filters:* Branch back to step 2, *Else:* continue.

9. Write contents of all accumulator registers back to memory.

To reach peak efficiency this algorithm assumes certain sizes for the input data and network layers that fit the block motif perfectly.

Variations for improved efficiency

Of course, not all layers will have k input channels, or a multiple thereof. Unfortunately due to the nature of the convolution algorithm proposed above, and the need to rotate the blocks around the ring to ensure that each convolution output sees all of its inputs, in the general case there will be wasted effort if all k blocks cannot be filled with input data. In this case the unused block(s) can be though of as a pause, or "gap" in the computation for whichever filter group's computational units with which they are aligned. To ensure that these computational units' accumulator registers are not modified while this gap in the data is aligned with them (and the other units are performing useful calculations), the weight values that will coincide with the gap blocks as they are rotated around the ring are set to zero. Figure 3.3 shows what the memory layout might look like for such a case where this is a single gap block.

As alluded to above, some layers may have more than k input channels. This case can easily be handled by first running the above described algorithm on the first k input channels. Then, instead of writing the result for an output row back to memory after the first k channels, the value is held in the arithmetic units' accumulators, and the computations for the next k channels are performed. This process is repeated until all channels are processed, possibly with the last iteration of the algorithm having only some subset of the k blocks in its SIMD rows filled with non-zero values. The full result for the convolution output row can then be written back to memory and the computations for the next row begun.

If a layer contains more than k filter groups the solution is even simpler. The above process is repeated for the first k filter groups, until all convolution output

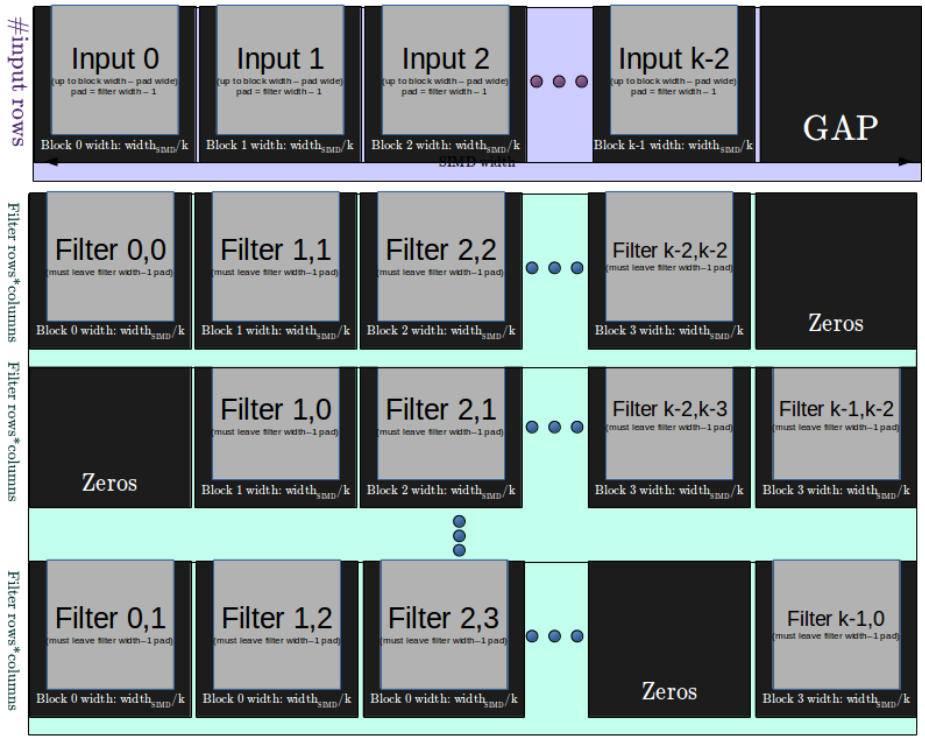

Figure 3.3: Memory layout with $k-1$ input channels and the k^{th} block as a single "gap" block. This method of arranging the inputs reduces efficiency, as on any given cycle some or the arithmetic units are essentially doing nothing. However, this method allows for the 2D convolution method to still be used even when not all blocks can be filled on some SIMD rows, trading reduced arithmetic unit utilization for the avoidance of costly and complex data shuffling operations.

rows for those groups have been completed. Then the computations for the next k filter groups are performed, with the possibility that some of the blocks or arithmetic units across the SIMD row do not have useful computations to perform, if the total number of filter groups is not a multiple of k.

Depending on the implementation details, the algorithm described above achieves close to 100% hardware utilization on network layers have input channel counts that are multiples of a power of two, when the width of such inputs is an exact power of two. If the network has been designed from scratch to run on this hardware then using meta parameters that meet these requirements would likely be a non-issue for everything except the initial input to the network, as inputs such as three channels for color images are common. However the hardware also needs to run networks that

were designed for other hardware, or without any hardware considerations in mind. In these cases the networks can still be run on the ultra-wide SIMD hardware, albeit at less that full efficiency.

Despite this, the block motif based convolution algorithm introduced in this section as M^3inM^2VConv presents an efficient way to perform convolutional network layer inference on ultra-wide SIMD hardware like ncore by organizing both the inputs (data and weights) and the algorithm itself to fit the structure of the hardware. This core algorithm however is designed for a basic convolutional layer, while in reality a wide array of variations of convolution are found in real networks. Thus, to be useful M^3inM^2VConv must be shown to be modifiable to work a wide range of these variations.

3.3 Variations to accommodate new convolution algorithms

As convolutional networks have become more popular in a wide range of applications, new variations on have been developed, either to improve network accuracy, or to decrease the computational requirements and/or parameters. Examples of these variations include depthwise, strided convolutions, and dilated convolutions. I will briefly describe these variations and show how the M^3inM^2VConv algorithm can be modified to accommodate them. Furthermore, I argue that this versatility shows that my algorithm provides a good framework by which to perform other variations on convolution like operations using the described ultra-wide SIMD hardware.

3.3.1 Depthwise Convolution

Depthwise convolution performs a 2d convolution on each channel separately, with each output channel being derived from inputs from a single input channel (Chollet, 2016). Thus the depthwise convolution algorithm is simply the standard convolution algorithm abbreviated such that input is never rotated to align a new set of channels with each output. The proposed convolution algorithm for ultrawide SIMD easily accommodates this trivial modification, simply by setting the number of iterations for the loop over channels to one.

$a_{0,0}$	$b_{0,0}$	$a_{0,1}$	$b_{0,1}$	$a_{0,2}$	$b_{0,2}$	$a_{0,3}$	$b_{0,3}$	$c_{0,0}$	$d_{0,0}$	$c_{0,1}$	$d_{0,1}$	$c_{0,2}$	$d_{0,2}$	$c_{0,3}$	$d_{0,3}$
$a_{1,0}$	$b_{1,0}$	$a_{1,1}$	$b_{1,1}$	$a_{1,2}$	$b_{1,2}$	$a_{1,3}$	$b_{1,3}$	$c_{1,0}$	$d_{1,0}$	$c_{1,1}$	$d_{1,1}$	$c_{1,2}$	$d_{1,2}$	$c_{1,3}$	$d_{1,3}$
$a_{2,0}$	$b_{2,0}$	$a_{2,1}$	$b_{2,1}$	$a_{2,2}$	$b_{2,2}$	$a_{2,3}$	$b_{2,3}$	$c_{2,0}$	$d_{2,0}$	$c_{2,1}$	$d_{2,1}$	$c_{2,2}$	$d_{2,2}$	$c_{2,3}$	$d_{2,3}$

Table 3.1: Element locations for three rows of four interleaved output groups of a M^3inM^2VConv strided convolution

3.3.2 Strided Convolution

Another convolution variation, used in popular networks such as *mobilnet*, is strided convolution (Howard et al., 2017). Strided convolution is a form of down-sampling that takes advantage of the fact that adjacent elements of a convolution are generally very similar. Downsampling methods like max-pooling compute all output elements, and then downsample by applying some function, such as max, to reduce groups of adjacent outputs to one element. In contrast, strided convolution simply only computes every n^{th} element, where n is the stride.

While in theory strided convolution reduces the total number of computations required by a factor of n, it presents potential problems for convolution algorithms that rely on the geometry of the standard convolution problem, as in the case of the previously described ultra-wide convolution algorithm. However, assuming that n is very small compared to the SIMD width, as is generally the case for a useful stride convolution, the geometry-preserving convolution algorithm can be modified to compute the down-sampled outputs of strided convolution at near full utilization, preserving both the efficiency of strided convolution and of the ultra-wide SIMD hardware.

Strided convolution can be handled by interleaving the **elements** of two output groups, allowing the inputs that are "skipped" by one output channel to be utilized as inputs to another. Thus, most/all available output elements are computing something useful every clock cycle, albeit at the expense of some simple post-processing to separate the final outputs.

To facilitate this interleaving of outputs, the weights for the two channels need to also be interleaved, however the input data remains in the standard data block format. Table 3.1 shows the first three rows of four such interleaved output groups of width four, originating from inputs blocks of width eight.

The input to a stride of two convolution stays in "normal" form, and like standard

zero padding convolution is shifted left $\lfloor filter\_width/2 \rfloor$ elements such that the first inputs to output elements in even groups (a, c, e...) start lined up with those outputs. On the first step of the inner loop (filter columns) only even output groups (a, c, e...) receive valid inputs. The weights for the other output groups are set to zero to mask out these multiples, as shown in Fig 3.4.

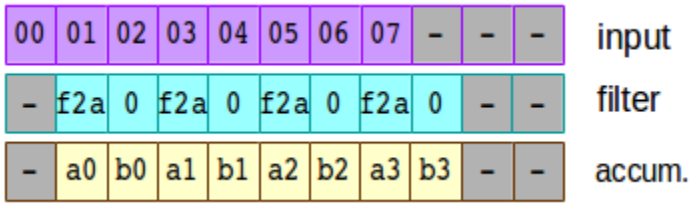

Figure 3.4: The initial step for a row of stride two convolution. Only output channel a receives valid inputs, and as such the weights for channel b have been set to zero. On the next step at least some elements from both channels will receive valid inputs

On the next next step, and on all remaining steps except the final one elements in both groups within each block receive valid inputs. As the input is rotated the output elements in the odd groups (b, d,...) see the input that was just seen by the associated even group element, while the even group element get its input for the next filter column to the left. Note that after middle step in the convolution (Fig 3.4), as is the input gets rotated further right, elements on the left need to be masked out to prevent contamination with data from the convolution block on the left (Fig 3.6).

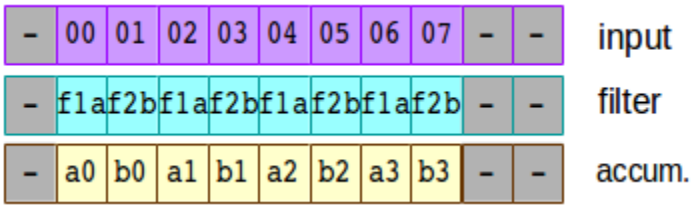

Figure 3.5: The middle step for a row of stride two convolution - note that all output elements have a valid input and that there is no need for zero valued weights or other masking

On the final step (Fig 3.7) of the inner loop only the even output groups (b, d, f, ...) receive input. Thus the inner loop consists of $filter\_width$ plus one step, one more than stride of one convolution.

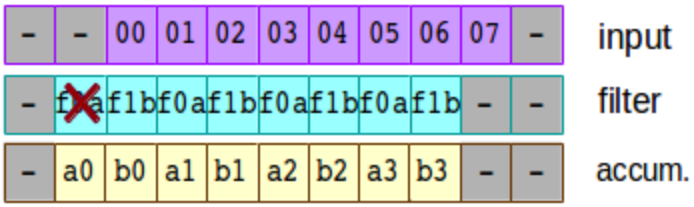

Figure 3.6: First step of a stride 2 interleave convolution where some output elements of output channel a have seen all of their inputs

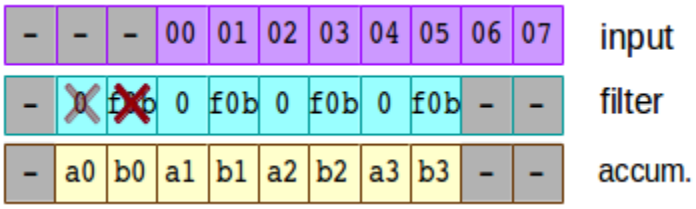

Figure 3.7: Final step for a row of stride 2 convolution interleaved output. Note that the only non-zero weights are for output channel b

Due to the first and the last step of this algorithm for a geometry preserving stride two convolution only providing inputs for one of the two interleaved channels in each block 100% efficiency cannot be achieved. However, depending on the size of the input it can provide !00% utilization of the compute elements and output registers on half of the multiplication steps for convolutions with a filter width of three, and more for larger filter widths.

One disadvantage of this method is the requirement that the interleaved output channels be separated after the convolution step has been completed, to facilitate their use as input to another layer. However, for an output channel width of N, the interleaved channels take up blocks $2N$ elements wide, where even $2N$ is assumed to be significantly narrower than the total SIMD width. Assuming that local gather operations and circular rotates of the SIMD row are available and efficient on the scale of $2N$ then the output channels can be de-interleaved using only a few steps per row.

Thus, while its efficiency is not perfect, strided convolution can be implemented using the M^3inM^2VConv framework while maintaining a high utilization of the ultra-wide SIMD hardware's compute capability.

3.3.3 Dilated Convolution

Another convolution variant, dilated (atrous) convolution, transforms the filter to allow the same set of filter weights to detect features in multiple scales. This is done by convolving the filter with every R^{th} image element, where R is the dilation rate. For example, an atrous convolution with a dilation rate of two would convolve the filter with every other element, such as elements zero, two and four from rows zero, two and four. A standard convolution can be thought of as having a dilation rate of one, as the filter is convolved with adjacent elements. A naive approach to doing this is to rearrange the inputs into R matricies that only contain, as adjacent elements, those elements involved in the same convolution. Then standard matrix-based methods for performing a convolution can be applied, and the results then appropriately rearranged. Some versions of dilated convolution in TensorFlow use this matrix transformation approach (tensorflow, 2018). On hardware that performs convolutions in matrix form efficiently such a method allows the main computational work to be performed efficiently, at the cost of some (somewhat) inefficient pre and post processing.

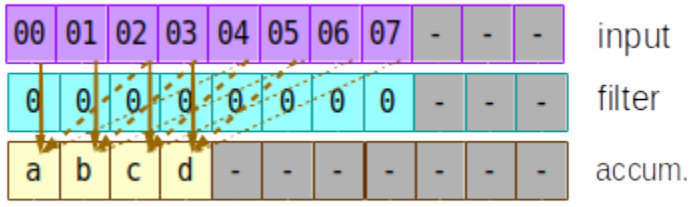

Figure 3.8: Initial step in the geometry-preserving convolution algorithm, modified to support dilated convolution. Note that besides the presence of few valid output elements, this computation begins the same way as that for standard convolution

However, just as the described ultra-wide SIMD hardware does not lend itself well to performing convolution as a matrix operation, it does not lend itself to the rearranging of data required for this "default" implementation. Thus, a new solution is needed. After observing that the sets of inputs from a given row used by adjacent output elements in a dilated convolution are shifted by one (e.g. $[0, 3, 6]$ and $[1, 4, 7]$), a variant of the basic geometry-preserving convolution algorithm can be developed to provide these input patterns using only ultra-wide SIMD friendly local shifts within each row of data. Specifically, instead of shifting the data in each row by one in

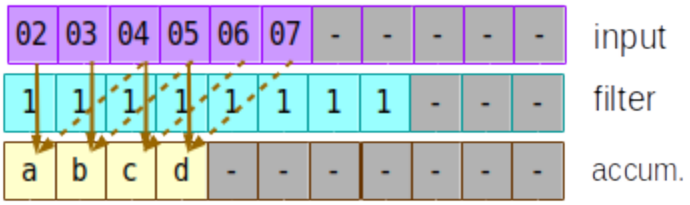

Figure 3.9: The next step to compute a single row of a filter in the geometry-preserving dilated convolution algorithm. Note that data has been rotated by R elements, which in this case is two.

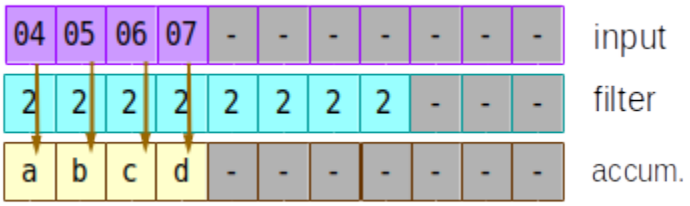

Figure 3.10: The final step to compute a single row of a filter of width three in the geometry-preserving dilated convolution algorithm.

between filter elements, the data is shifted by R. Figure 3.8 shows the data and weight layout for the first step of a row of a dilated convolution, in which the data is aligned just as is would be for a standard convolution. Figure 3.9 shows the next step, with the data rotated by R instead of one. Finally, Figure 3.10 shows the final step of a convolution with a three element wide filter, having once again rotated the input row by R.

Assuming that the ability exists to rotate a SIMD row by R elements in a single clock dilated convolution is just as efficient as standard convolution. Furthermore, since no "rearranged" form of the data is needed to perform a dilated convolution using this method the same input data can be reused for multiple dilation sizes (including standard convolution).

3.3.4 Other convolution variants

While the convolution variants described here are obviously not the entire space of possible variations, it can be argued that even showing adaptions to this range of variations proves the flexibility of M^3inM^2VConv. Similar methods are likely to allow M^3inM^2VConv to be adapted to other variations, although the performance

of these variations might vary, causing convolution variants that normally are faster than standard conv to not experience the same speedups on the ultra-wide SIMD hardware, if for example the computation of elements intended to be dropped must still be performed, then later dropped on the ultra-wide SIMD hardware.

3.4 Summary of M^3inM^2VConv Convolution Algorithms

In this chapter a baseline implementation of M^3inM^2VConv, a geometry-preserving convolution algorithm for ultra-wide SIMD hardware is presented. This algorithm is based on a block layout analogous to the layout of the hardware itself, minimizing the need for complex data movements during the the algorithm itself. Further, by providing modifications to the algorithm for several common convolution variations it is shown that M^3inM^2VConv is versatile and is likely to be able to be applied to future convolution variations.

However, while it is shown that M^3inM^2VConv successfully implements convolution using the limited data movement capabilities of the ultra-wide SIMD hardware, it still needs to be shown that it is efficient. Specifically, the algorithm should minimize the memory used for padding or repetition of data that requires more space that the initial size of the input. Additionally, M^3inM^2VConv should ensure that as many arithmetic units are used as possible each clock cycle (MAC efficiency). Both of these performance considerations will be discussed in the following chapter.

Chapter 4

Quantitative Analysis on Proposed Ultra-Wide SIMD Hardware

In this chapter I will focus on two different aspects of the efficiency of the above described geometry preserving convolution algorithm: arithmetic unit utilization rate, and memory usage requirements. Since the described algorithm is for a proposed class of ultra-wide SIMD architectures, without set values for things like clock speed, power requirements memory interfaces or even the actual SIMD width one cannot compute more traditional measures of performance such as floating (or fixed) point operations per second (FLOPs). The equations I derive should however be able to be used to estimate the maximum performance of any real incantation of this architecture. Moreover, the estimates of memory usages can be used to help estimate the speed of the memory interface needed to allow an actual implementation to approach the theoretical maximum compute efficiency.

Additionally, due to the nature of the convolution algorithm previously described, the exact geometry of the network can have a large effect both on memory usage and computational efficiency. The reasons for this will be described when the memory usage is quantified, however the same utilization principles apply to computational efficiency.

4.1 Memory Usage

The following equations outline the memory usage of the above described method for performing convolution operations as part of a convolutional network layer. The size of the blocks that the SIMD row is divided into is denoted by B. The maximum block size is $B < 2 * WIDTH_{input}$, since if B were greater than or equal to the $2 * WIDTH_{input}$ the next smaller power of two could be used as the block size.

According to the weight coefficient layout described above where the weight values are replicated to align one copy with each arithmetic unit that is to us that weight on a given clock. Thus, each weight requires B words of memory, either for the replications of the weight value, or for the padding around it in it's block (Zeros in

weight RAM due to a "gap" to fill in a block to which a filter group is not assigned are not considered here). Thus, if F is the number of filter groups, R and S are the number of rows and columns in each filter respectively, and C is the number of channels in the input data, the total number of words needed to hold the weight coefficients in the form used as input to the previously described algorithm is as follows:

$$TOTAL\_WORDS_{weights} = F \times R \times S \times C \times B \tag{4.1}$$

Putting the block size in terms of the input width this equation becomes

$$TOTAL\_WORDS_{weights} < F \times R \times S \times C \times 2 \times WIDTH_{input} \tag{4.2}$$

This is an increase by a factor of $2 * WIDTH_{input}$ over the original wordcount for the weights, which is $O(n)$. It should also be noted that the replication of the original weight values needs to only occur once, assuming that there is room to store the expanded weight matrix somewhere, either in ncore RAM or in main memory.

From this total number of words required to store the weight in the format that the algorithm needs one can determine the number of SIMD rows work of memory required. The number of filter groups in a given ConvNet layer might not require all SIMD rows to be filled with weights. However, since the ncore memory is only addressable by row the total number of memory rows required is, where $BLOCKS\_PER\_ROW = \frac{WIDTH_{SIMD}}{WIDTH_{BLOCK}}$:

$$TOTAL\_MEMORY\_ROWS_{weights} = ceil\Big(\frac{F}{BLOCKS\_PER\_ROW}\Big) * R * S * C \tag{4.3}$$

While this is clearly an increase over the $F * R * S * C$ words needed to store the weights when the convolution is performed as a matrix multiply, it still presents the advantage that it's the filter coefficients in this case that are being replicated, not the input data. That means that assuming there this room to store it, it can be done once and no further preprocessing operations are needed, as opposed to the need to expand and reshape the input data matrix every time, at the input to every layer.

Likewise, the layout and memory requirements for each layer's input data can be described in terms of the network parameters F, R, S, and C, along with the width and height of the input data. Unlike the weight values, or the input data in

the method by Chellapilla et al. (2006) to implement the convolution operation as a matrix multiply, the data here isn't actually duplicated (beyond optional tiling across SIMD rows that are less than half full). However padding to fit the data into the selected block size does cause it to take up additional memory space. Since one block within a SIMD row is used for each row and channel of the input the total number of words needed is equal to:

$$TOTAL\_WORDS_{data} = B \times HEIGHT_{input} \times C \tag{4.4}$$

Using the previous observation that $B <= 2 \times WIDTH_{input}$

$$TOTAL\_WORDS_{data} < 2 \times WIDTH_{input} \times HEIGHT_{input} \times C \tag{4.5}$$

Since each SIMD row may only contain data from different channels with the same row of input data the total number of SIMD rows required is as follows:

$$TOTAL\_MEMORY\_ROWS_{data} = ceil\Big(\frac{C}{BLOCKS\_PER\_ROW}\Big) \times HEIGHT_{input} \tag{4.6}$$

This is an $O(2)$ increase over the original size of the data. However this should be compared to the method of Chellapilla et al. (2006), where the input data is expanded into a matrix of size $R * S * C \times P * Q$ where P and Q are the height and width of the output, respectively. For valid only convolutions $P = HEIGHT_{input} - R$ and $Q = WIDTH_{input} - S$. For same-size convolutions $P = HEIGHT_{input}$ and $Q = WIDTH_{input}$. Thus

$$TOTAL\_WORDS\_MATRIX_{data} <= R \times S \times C \times HEIGHT_{input} \times WIDTH_{input} \tag{4.7}$$

Even in the worst case where almost half of the elements in each block are filled with zeros in the 2D convolution method the matrix method uses $\frac{1}{2} * R * S$ more data words. Even for a small, 3×3 filter kernel this is a $4.5x$ increase.

Furthermore the use of the 2D convolution method reduces the amount of transformation required for the data going into each network layer on the ultra-wide SIMD architecture. Since whole output rows for multiple filter groups are computed together and output as one SIMD row the output from the previous layer will already be in a

form very close to the required 2D form. The max-pooling operation used after most convolutional layers will result in smaller block sizes being needed. However these can be handled using simpler compression and rotation operations that take advantage of the hardware's circular ring of registers, and do not require the use of arbitrary swizzle or shuffle operations.

4.1.1 Effect of Network Parameters on Effecient Memory Usage

Due to the geometry based representation of the data in the proposed ultra-wide SIMD convolution algorithm the exact network meta-parameters can have a large impact on the utilization rate of the individual compute elements. Specifically, the algorithm relies on the ability to rotate the data in such a fashion that each input channel remains aligned with within some output "lane" as the input channels are aligned as input to each output requiring them. In order to do this the SIMD row must be divided into lanes of equal width with no remainder. Assuming that the SIMD width itself is some power of two then the width of these lanes must also be a smaller power of two, regardless of whether the entire lane can be filled with data. Thus networks whose layers have input widths of non power of two sizes will "waste" memory with padding value to accommodate the algorithm. The worst such width is $2^n + 1$, as such inputs do not fit within a 2^n wide block, requiring the use of a 2^{n+1} wide block, with $2^n - 1$ pad elements.

Another source of padding elements comes from situations where a layer does not have enough channels, C, to completely fill up a SIMD line's worth of data (or n SIMD lines). Specifically, if $BLOCKS\_PER\_ROW$ is not evenly divisible by C then some blocks within the "remainder" SIMD line will have to be filled with pad elements, the worst case of this being cases where $C\%BLOCKS\_PER\_ROW == 1$.

4.1.2 Summary of Memory Usage Analysis

In conclusion, while M^3inM^2VConv does use extra memory space for storing the filter weight values, this is preferable to expanding or otherwise duplicating the input, as the weight arrays tend to be smaller and are reused. Additionally, the addition of even a local broadcast capability to the hardware, as described in (Henry, 2020), can

considerably reduce the space needed to store the input weight arrays. Likewise, while some padding might be needed to pack the input into the SIMD width input rows, this padding is limited. Additionally, since the ideal case the padding is non-existent designing a network width zero or minimal padding would be possible.

4.2 Computational Efficiency

For a network with ideal size parameters 100% arithmetic computation efficiency can be achieved, assuming that the hardware can perform the data movement operations needed to feed the input to each arithmetic operation can be performed in parallel with the arithmetic operations. Of course, in practice, less than 100% efficiency can often occur. I go over the three reasons for this below; network parameters, data movement and the effects of variations on the classical convolution algorithm

4.2.1 Effect of Network Parameters on Computational Efficiency

Like the pad elements in the memory representation of the input data, depending on the network parameters some computational units may go unused when calculating the output to a layer. There are two situations that can lead to idle to computational units dues to less than ideal network parameters - within block padding and whole unused blocks.

If the input to a computational layer requires pad elements within each block due to non power of two input sizes then there will be at least as many unused computational units as pad elements in the input. Whether there are additional computational units depends on whether the convolution itself is a "same-size" or a "'valid" convolution. Same-size convolutions allow a partial convolution of the filter with the edges of the input, leading to an output that is the same size as the input, despite the elements at the edges not being full convolutions. In contrast, a valid convolution only includes elements where the filter can be fully convolved with the input elements, ensuring valid values with no edge effects at the expense of making the output smaller than the input.

Additionally, just as entire blocks of a SIMD line in RAM can be filled with pad elements in there are insufficient channels in the input to fill the line, the results from

whole blocks of computational units can be unused if there are insufficient channels in the layer output to titillate the full SIMD width. Specifically, the number of unused blocks of arithmetic units is equal to the following:

$$C_{output}\%BLOCKS\_PER\_ROW \tag{4.8}$$

where C_{output} is the number of channels in the layer output.

Thus the ideal convolution layer from an efficiency standpoint is one were the width of the data is some power of two such that the entire lane of input contains valid input data, and that uses "same-size" convolution, such that the entire lane of computational units is producing useful output values as well. In fact, with the described convolution algorithms the same-size convolution is always computed unless the edge elements are masked out. And thus in the case of a valid convolution these computational units are essentially unused, as are those being fed only pad elements as input. Furthermore, the maximally efficient network has numbers of channels in each layer that are either power of two, or otherwise multiples of the appropriate number of $BLOCKS\_PER\_ROW$.

4.2.2 Effect of Data Movement on Computational Efficiency

For purposes of this metric it is assumed that the ultra-wide SIMD hardware as a unit can be fed data from the outside world (system RAM, disk, etc) as fast as it can process it. Thus the concern here is with computational cycles being utilized by the ultra-wide SIMD unit *solely* for data movement purposes.

It is assumed that through the use of pipelining and/or parallel execution paths that the ultra-wide SIMD hardware can perform data movement operations at the same time as arithmetic operations. Thus most data movement will be hidden and not have a effect on computational density. However, complex data movement operations that take multiple clock cycles, or those that must be performed on the result of one computation before the next computation is performed have the potential to result in "dead clocks" where the computational unit utilization id 0%.

For basic convolution the primary way that these "dead clocks" can occur is for convolutions with small filters ($1x1$ or $3x3$ for example) and data widths such that the SIMD row must be divided into blocks that are wider than the maximum amount

that the hardware can rotate the data within the SIMD row within a single timestep.

4.2.3 Effect of Convolution Variations of Computational Efficiency

Finally, I will discuss the effects of various modifications to the convolution algorithm on computational efficiency. Specifically I will address the three variants described above; depth-wise convolution, strided convolutions and dilated convolution. The depthwise and dilated convolution variants can theoretically achieve the same performance as standard convolution for the same layer input and output sizes. The cases of strided convolution is more complicated however, as the algorithm itself potentially introduces unused cycles for subsets of the arithmetic units as well as further processing steps for the final output.

Depthwise Convolution

The major difference between depthwise convolution and standard convolution is that depthwise convolution only provides one channel of input to each output channel. Thus assuming that the fact that the data is not "reused" by using each channel of input as input to multiple output channels does not cause the data input bandwidth requirements to exceed the hardware's capabilities the depthwise convolution variant shows no reduction in computational efficiency compared to standard convolution.

Stride Two Convolution

Because stride of two convolutions are performed by interleaving two channels of output within one block they are able to achieve better utilization of the input, despite greater difference the set of input elements received by adjacent outputs compared to standard convolution. However, this method of arranging the output computations within the SIMD row still results in some instructions where only half of the arithmetic units are doing work, even for ideal size parameters. Since these instructions are the first and last instruction of the computation of each row of the convolution filter, the total utilization rate increases with the filter size(width).

For example, for a 3×3 filter four arithmetic instructions are needed to compute each row of the convolution filter, with the first and the last instruction having only

50% utilization. Thus for a 3×3 filter the overall utilization is 75%. In contrast, for a 7×7 filter eight arithmetic instructions are needed per filter row, so with the two half utilization instructions the overall utilization is $\frac{7}{8}$, or 87.5%.

In addition to this incomplete utilization of the arithmetic units, stride two convolution potentially occurs an additional performance penalty due to the need to deinterleave the resulting output channels after the computation is complete. It is assumed that the same mechanism that rotates/transforms the data for input into the arithmetic units can perform the local shifts and blend required the deinterleave the output, however co-opting this for this purpose may hold up the availability of data for subsequent arithmetic operations.

Dilated Convolution

For same size convolution and the proper input sizes dilated convolution can theoretically achieve 100% arithmetic unit utilization. The assumption here is that the dilation rate is of a size such that the SIMD line can be rotated by the appropriate amount a single clock and be ready for the next arithmetic operation. If this is not the case the arithmetic unit utilization can drop rapidly, with 50% utilization for dilation rates requiring two rotations, and 33% for those requiring three.

Like for regular convolution, dilated convolution using only the "valid" outputs will lead to unused arithmetic units within each block even when the input has no padding. However, due to the wider effective filter width ($WIDTH_{FILTER} * R$, where R is the dilation rate) the amount of padding in the output due to the use of valid only convolution will increase with R.

4.3 Performace Conclusions

In conclusion, while it's possible that tweaks might exist to improve the algorithm's performance in at least some cases, overall M^3inM^2VConv performs well on the ultra-wide SIMD hardware with respect to both memory usage and computational efficiency. Additionally, adaptations to the algorithm to account for modified convolution algorithms do not cause this performance to unravel.

Finally, it should be noted that this analysis is for the basic ultra-wide SIMD paradigm described at the beginning of this document and the ring data movement

paradigm. It does not take into account additions to the hardware's data movement capabilities such as local broadcast that could help optimize M^3inM^2VConv without requiring longer distance data movement.

Next the performance of M^3inM^2VConv will be evaluation on another relatively wide SIMD architecture, AVX-512, that was designed assuming that arbitrary data movements are common, as they are much easier to implement efficiently in hardware at narrower SIMD widths.

Chapter 5

Implementation on Existing AVX-512 Hardware

While the width of Intel's AVX-512 standard isn't of the same order of magnitude as the Ultra-wide SIMD architecture proposed in this document, it's ability to accommodate 16 element wide f32 operations (or 32 element wide f32 operations on some varients) allows it to support at least small implementations of the above proposed convolution algorithm. Importantly, chips that support most of these instructions have actually been produced and are available to the public, including the Xeon Phi many-core processors, some high end Xeon server parts and the Skylake-X series of desktop CPUs (Intel, 2019).

5.1 Purpose: Prove algorithm generalizes to other wide hardware

While the proposed geometry-preserving convolution algorithm, MinMV, wasn't designed with a narrower SIMD architecture in mind, implementing the algorithm for AVX-512 shows that the algorithm can be applied to general purpose SIMD architectures that either already exist or that might be invented in the future, with out without deep learning in mind as the sole, or primary, application. While many extensions to the AVX-512 standard have been released and are available in different subsets on various AVX-512 capable CPUs, this implementation uses instructions in the AVX-512F foundation standard, which is available on Skylake-X consumer grade desktop parts (i7 78xxX and i9 79xxX X-series) (Intel, 2019).

However, while AVX-512 represents a twofold increase in SIMD vector width over previous x86 SIMD instructions, both the AVX-512 vector width and immediately accessible memory register size are orders of magnitude smaller than those of the *ncore* ultra-wide SIMD architecture paradigm for which $MinMVConv$ was designed. Despite this, it is possible to implement $MinMVConv$ for a $3x3$ convolution on a small $8x8x2$ floating point input using AVX-512. While not practical for real world applications, this implementation still allows $MinMVConv$ to be analyzed on AVX-512, yielding insights into what other types of hardware $MinMVConv$ might perform

well on.

5.2 Implementation

The following C code is an example implementation of the previously described geometry-preserving convolution algorithm utilizing Intel's AVX-512 instruction set for a tiny $3x3$ convolution, computing two same-size output channels on two $8x8$ input channels. While tiny, this convolution illustrates the geometry-preserving convolution algorithm on AVX-512, while fitting within the 32 available 512b *zmm* registers (Intel, 2021). The full source code is included in appendix A, however the psuedocode is provided in this chapter to make understanding the implementation easier.

Table 5.1: AVX-512 Convolution Setup

1. Declare zmm registers for expanded filters, input data, and working memeory for a single output row
   - `expanded_filters[18]`
   - `output_2chan`
   - `input_2chan[16]`
2. Declare arrays of idencies to perform ncore style rotates using AVX-512 permute operations
   - `filter_column_rotate_mask:` `{15,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14}`
   - `filter_column_initial_rotate_mask:` `{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0}`
   - `channelwise_rotate_mask:` `{11,12,13,14,15,0,1,2,3,4,5,6,7,8,9,10}`
3. Load filters and input into registers.

The convolution function starts with the setup steps described in Table 5.1. First the *zmm* registers to be used are declared, and loaded with the appropriate values. Of note are the use of registers to hold all rows of a pair of input channels (eight registers

in this case), registers to hold the expanded arrays of filter coefficients (18 in this case, for a $3x3$ filter times two input channel configurations). These are values that are stored in the register space for geometry-preserving convolution on the proposed ultra-wide SIMD hardware for which the algorithm was developed. Since AVX-512 does not include a concept of rotating elements around a circular SIMD width buffer like the proposed ultra-wide SIMD architecture does, additional registers are need to support the general permute instructions that are supported. These masks are constants for a given input configuration. The $filter\_column\_rotate\_mask$ is used to rotate the input data to align the input properly for each filter column, with $filter\_column\_initial\_rotate\_mask$ having been used to align the data elements one element to the left to ensure that the output is aligned with the original input. Finally, the $channelwise\_rotate\_mask$ index mask will be used to realign the input channels as input to the other convolution output channel.

Once the filter coefficients have been loaded into their allocated *zmm* registers the actual convolution loop can be performed. One output row is computed at a time, with this out_row consisting of two eight element wide channels. At the beginning of the loop for each output row out_row is set to zero, and then the convolution is performed for each filter input row, and each of the two channels therein, for each of the two input channels the inner loop of the convolution is performed, multiplying the coefficients for each filter column in the current filter row by the appropriately aligned inputs, as shown in Table 5.2. Since during same size convolution with no padding inputs from one channel get rotated such that they are aligned with the output lanes of the other channel the masked form of fused multiply add is used, allowing for the appropriate bit mask for each filter column to be used to prevent the output from being contaminated with other channels' data. It should be noted that this mask is separate from the index masks used with the $\_mm512\_permutexvar\_ps$ intrinsic to rotate the data. Finally, after all the filter input rows have been computed for both channels the contents of the *zmm* register being used to store the output is store back to memory so that the next row on convolution outputs can be computed.

Of course, this implementation of the algorithm is limited to a much smaller convolution then is realistic. Even networks now considered "toy" problems, such is LeNet (LeCun et al., 1998) have channel widths wider than 8 elements, and even the narrowest layers have more the two input channels.

Table 5.2: Convolution inner loop - unrolled with specific intrinsics for $3x3$ filter, two input and two output channels.

1. perform multiply-add for rightmost element of conv filter row:
   `_mm512_mask3_fmadd_ps(input[row], filters[i], output, 0x7F7F)`
2. rotate one element for next column:
   `_mm512_permutexvar_ps(filter_column_rotate_mask, input[row])`
3. perform multiply-add for middle element of conv filter row:
   `_mm512_mask3_fmadd_ps(input[row], filters[i], output, 0xFFFF`
4. rotate one element for next column:
   `_mm512_permutexvar_ps(filter_column_rotate_mask, input[row])`
5. perform multiply-add for leftmost element of conv filter row:
   `_mm512_mask3_fmadd_ps(input[row], filters[i], output, 0xFEFE)`
6. Rotate one element for next column:
   `_mm512_permutexvar_ps(filter_column_rotate_mask, input[row])`
7. rotate input row to align the input to the other output channel:
   `_mm512_permutexvar_ps(channelwise_rotate_mask, input[row])`
8. Repeat 1-6 with appropriate filter elements

5.3 Computational Efficiency

One major challenge with respect to performing the geometry-preserving convolution algorithm using general purpose SIMD AVX-512 hardware is that while it can be assumed that on the proposed ultra-wide SIMD architecture that data rotations can be pipelined such that they are at least partially performed in parallel with the arithmetic operations, no such pipelining is available with the AVX-512 standard, as a data manipulation instruction must be completed and returned the results returned to one of the AVX-512 general purpose registers, not fed directly to a subsequent arithmetic operation, as with ncore. This means that in contrast to the ultra-wide SIMD case where 100% efficiency can be achieved for an ideal network, the best case efficiency for the AVX-512 implementation the inner loop alone is at best 50%, has

only half of its instructions perform actual arithmetic, with every other instruction performing a non-simultaneous data movement operation instead.

5.4 Register/memory usage

The second major challenge in implementing the geometry-preserving M^3inM^2vConv algorithm on AVX-512 is accounting for a massive difference in directly accessible register space. Specifically, the total combined space in the 32 512b registers available for input for AVX-512 instructions is of the same order of magnitude as the amount of space available to feed arithmetic and data operations on the hardware for which M^3inM^2vConv was designed. In fact, the memory space from which the previously described ultra-wide SIMD architecture is assumed to be able to feed data to the data manipulation and arithmetic units directly like a register, is of a total size whose order of magnitude is similar to a Skylake-X CPU's total cache size (Intel, 2019).

The lack of immediately accessible register space hobbles the application of the geometry preserving convolution algorithm to even small input and filter sizes on AVX-512. However, it is still posiible to implement simple networks on AVX-512. Thus, I will analyze a simple network's AVX-512 implementation's usage of the 32 AVX-512 SIMD registers, which at at 16kB is three orders of magnitude smaller than that of the hardware on which M^3inM^2vConv was designed to run real networks.

Recall that in Eq 5.1 for the M^3inM^2vConv in the ultra-wide SIMD hardware:

$$TOTAL\_MEMORY\_ROWS_{data} = ceil\Big(\frac{C}{BLOCKS\_PER\_ROW}\Big) \times HEIGHT_{input} \tag{5.1}$$

where C is the number of channels in the input data and $BLOCKS\_PER\_ROW = \frac{WIDTH_{SIMD}}{WIDTH_{input}}$. For the example case about this means that $TOTAL\_MEMORY\_ROWS_{data} = ceil\Big(\frac{2}{\frac{16}{8}}\Big) \times 8$, for a total of eight memory rows filling eight SIMD registers.

Further, recall that in Eq 5.2:

$$TOTAL\_MEMORY\_ROWS_{weights} = ceil\Big(\frac{F}{BLOCKS\_PER\_ROW}\Big) * R * S * C \tag{5.2}$$

where F is the number of filter groups, R and S are the number of rows and columns in each filter respectively, and C is the number of channels in the input data. Thus, a $3x3$

filter for 2 input and 2 output channels requires $ceil\left(\frac{2}{BLOCKS\_PER\_ROW}\right) * 3 * 3 * 2$ registers to store the weight coefficients. An input with a width of 8, as used in the example above, allows for a maximum of 2 $BLOCKS\_PER\_ROW$. Thus, $ceil\left(\frac{2}{2}\right) * 3 * 3 * 2 = 18$ registers are required to hold the weight coefficients.

In the example above this means that in total for two eight element wide inputs there is room for all eight input rows of the $8x8x2$ input, along with both sets of nine registers for weight coefficients. This makes for $18 + 8 = 26$ registers in total, and does not leave room to support any more input channels.

5.5 Actual Run

While the above analysis' show why the M^3inM^2vConv algorithm is not well suited to the smaller AVX-512 architecture, and that it is not a practical algorithm for running real networks, it is still important to show the correctness of the algorithm on even a toy problem in order to show that the above analysis is valid. To do this the algorithm was run an an Intel eon Platinum 8375C by way of an Amazon EC2 instance running Ubuntu 20.04. See B for full output from convolution run showing correctness of all output values.

The log first shows the input to the test network - small values with only an occasional zero chosen to both make the log easy to read and ensure that arithmetic errors get detected. Next, the inputs for computations made by the AVX-512 implementation for each output row are shown, with the 16 values across the SIMD width grouped into their eight element wide channels. Finally, the output of the AVX-512 implementation from A is compared to the output of a simple nested loop implementation also found in A. To account for differences in floating point execution order, values that matched the reference implementation within 0.0001 were counted as correct. Further, several output values were checked by hand to ensure that the reference implementation was also correct.

5.6 Lessons Learned

While it has been shown that M^3inM^2vConv can be correctly implemented on AVX-512, it is also clear that it is not a good fit for the architecture. The issue of

only being able to handle two channels worth of input in the *zmm* registers at once can be mitigated by not loading all the input rows into the registers at once, allowing space for multiple channel pairs worth of data for the rows currently being fed to the convolution. The downside to this however is that the filter coefficients would also have to be moved into and out of the *zmm* registers, since even for a 3×3 filter this is only room for one pair of input channels' worth of coefficients.

One option to make better use of the limited *zmm* register space is to stream in the input rows, as the latter rows of the input are not needed for the initial convolutions, and after the initial convolutions the first few rows of the input are no longer needed. This would allow for two channels worth of 16 element wide input to be supported at once, as the 20 *zmm* registers left available after the 18 registers have been assigned for the coefficients for two channels worth of $3x3$ convolution is sufficient to support the six registers (three rows times two channels) needed for the currently needed rows of 16 element wide inputs, along with extra space for the accumulated result of the convolution FMAs, and extra room for new rows of input being loaded in. However, three input channels would require 27 registers to hold the filter coefficients, which does not leave enough registers for the inputs required for the current convolution output row, even in the case of eight element wide inputs, where two channels times three rows worth of input would still fit in six *zmm* registers.

New AVX-512 instructions to allow multiple filter coefficients to be stored in the same *zmm* register and only temporarily expanded out in a "scratch space" *zmm* register right before they are used, or a new AVX-512 implementation with more SIMD registers, would make small scale implementations of the proposed geometry-preserving convolution algorithm feasible of current x86 SIMD hardware,though likely with reduced computational efficiency due to necessary "weight expansion" step.

Thus, barring such new developments, the geometry preserving SIMD algorithm is best suited for less general purpose architectures that were designed for extremely wide SIMD and that have a large amount of onboard, readily accessible, register-like memory and a more efficient means to immediately use the output of local data movement operations as arithmetic inputs. However, despite not being well suited for AVX-512 architectures, implementing even a simple version of M^3inM^2vConv in AVX-512 was still useful as it helped to highlight these properties of hardware on which M^3inM^2vConv excels.

Chapter 6

Gated Recurrent Network Computations on Ultra-wide SIMD Hardware

While my implementation of convolution algorithms on the described ultra-wide SIMD hardware is the primary focus of this work, depending on the application many modern networks also incorporate other types of neural network layers, including fully connected and recurrent layers.

Assuming the ability to feed both input data and the associated weights to the compute hardware fast enough, a fully connected layer is trivial on this hardware. The input line can simply be rotated until every output has seen every input, with the appropriate weights being provided for the multiply-add at each step. As I will show below, the inference step of common recurrent neural network algorithms, including the seemingly complicated gated recurrent networks are simply made up of fully connected layers, with a few, computationally small, extra steps.

6.1 Insight

The main insight for performing computations for recurrent neural network layers on the proposed ultra-wide SIMD hardware is the fact that the bulk of the computation for any recurrent neural network layer is a matrix multiply, the same as for a classical, fully connected neural network. In fact, for a "vanilla" or classical recurrent neural network where each neuron simply has as it's input the layer input and direct recurrent connections between all neurons in the layer the computation is simply a matrix multiply for two sets of fully connected inputs - the incoming connections and the recurrent ones. This same idea applies to more complex recurrent networks that use gates, such as LSTMs (Graves and Schmidhuber, 2005) or GRUs (Cho et al., 2014). Likewise, a similar principle can be applied to more exotic variants such as qRNNs (), which use convolution operations as part of the recurrent layer.

6.2 LSTM Algorithm

I will discuss Long Short-Term Memory (LSTM) layers as a canonical example of a gated recurrent network layer. While other types of gated recurrent layers exists, such as the Gated Recurrent Unit, this same principles apply to their implementation. Unlike "vanilla" recurrent networks, whose output is simply a weighted sum of the layer's inputs at that timestep and it's outputs at the previous timestep gated recurrent networks use gates to control how much the new inputs effect the output, and how much information from the old output is allowed to remain, creating a longer range form of memory. These gates are themselves controlled by the weighted sum of the incoming and recurrent connections, with unique weights for each gate, with each neuron, or *cell* having it's own set of gates:

$$i = \sigma(W_i * x_t + U_i * h_{t-1} + w_{ci} * c_{t-1} + b_i) \tag{6.1}$$

$$f = \sigma(W_f * x_t + U_f * h_{t-1} + w_{cf} * c_{t-1} + b_f) \tag{6.2}$$

$$o = \sigma(W_o * x_t + U_o * h_{t-1} + w_{co} * c_t + b_o) \tag{6.3}$$

The key to implementing a gated recurrent layer is to recognize that the computation consists of two parts, the matrix multiply for the (fully connected) input and recurrent connections for each gate, and a few scalar arithmetic steps for each gate. While this can be computationally demanding if the fully connected multiplies are large the steps are simple. Once the weighted sums from the inputs are computed and the correct activation function applied to them the only new step is the gate equations themselves. For example, for an LSTM the cell state(c_t) and output(h_t) are computed as follows:

$$c_t = i * \hat{c} + f * c_{t-1} \tag{6.4}$$

where

$$\hat{c} = tanh(W_c * x_t + U_c * h_{t-1} + b_c) \tag{6.5}$$

and the final output is:

$$h_t = o * tanh(c_t) \tag{6.6}$$

The data for the matrix multiply can be arranged in two ways to provide inputs for these final gate equations. One approached, used by major deep learning frameworks such as Tensorflow, is to interleave the weighted sums for the gates and candidate state (i, f, o and $\hat{c}$) (tensorflow, 2018). While this is definitely possible on the proposed ultra-wide SIMD hardware it then requires that the values in the resulting SIMD row be locally shifted in order to line of the appropriate values to multiply together. Since such operations are difficult, even on a local scale on the proposed hardware, a better approach is to compute a full SIMD line's worth of input gates (i), store it, then compute a full line worth of cells f gates, and so forth until a line's worth of all four values have been computed. The final state and output values can then be computed for a full SIMD line's worth of cells at once, with only multiply-add operations and no data shifting.

6.3 Hybrid Recurrent-Convolution: Quasi Recurrent Neural Networks

Quasi Recurrent Neural Networks () expand of the idea of recurrent gated networks by employing LSTM-like gates that are controlled by the output of convolutions over the input, combining the computational efficiency of convolution with the memory capabilities of a gated recurrent network. While this sounds complicated, the building block approach used for standard LSTMs can also be applied here - first the gate control and input signals are computed, using the previously described geometry preserving convolution algorithm in this case, and then those output can be used as the inputs for the LSTM gate equations in the same way as the result of the standard fully connected multiply is.

Thus, while recurrent neural networks appear complicated and are often computationally demanding to compute the bulk of this computation is the same as that used for other, non-recurrent networks. By combining the right algorithms and adding a few extra steps recurrent networks can easily be implemented on the proposed ultra-wide SIMD hardware. While the resulting algorithms are not as interesting as the geometry-preserving convolution algorithm used to make convolution efficient on the ultra-wide hardware showing that recurrent layers can also be implemented on the proposed hardware, allowing all layers of a network with both convolution and

recurrent connections to be executed on the hardware. This flexibility helps show that the hardware can be used for problem domains beyond simple convolution based image processing, including domains with a temporal element, language and signal processing.

Chapter 7

Proposed Application in Decoding Neural Signals

Furthermore, I will show the applicability of gated recurrent units, which will also be implemented on the hardware, to a new domain; the decoding of observed neural activity. Specifically, I use gated recurrent unit networks to predict the stimuli that coincided with the observed activity of real neurons. GRU networks were trained using a dataset consisting of the observed spikes from a small number (10 to 16) of clusters of neurons in the rat primary auditory cortex (A1) recorded while the animals performed a frequency discrimination task. The task stimuli consisted of a variable length series of identical single frequency tones, followed by a "target" tone of a slightly different, single frequency. The animals were rewarded if they responded correctly to this target stimulus. Additional details regarding the behavioral task and the recording of the neural data found in this dataset can be found in Sloan et al. (2009).

Previous analysis of this data show that the coincident stimulus can be predicted from the observed neural activity using both the PSTH based classifier Foffani and Moxon (2004) discussed previously and a support vector machine (SVM) based classifier. The more powerful SVM based classifier was shown to be more accurate than the PSTH classifier (Dodd et al., sion, Houck, 2012), however there is much room for improvement over the performance of both classifiers. Preliminary results have shown that GRU networks are capable of decoding information from the neural signals. Thus expanding the usefulness of gated recurrent networks to this domain.

I will further fine-tune the performance of these networks, training them using Theano (Al-Rfou et al., 2016), with layer implementations from the Lasagne framework (Dieleman et al., 2015). I will then use the algorithm that I have developed to evaluate GRU layers on the described hardware to perform inference on held out test examples.

7.1 Problem Definition

While many deep learning algorithms were developed for domains involving the processing of visual spectrum images or video, speech or text, there is no reason

that these algorithms cannot be applied to other signal processing domains. Here I present one such domain that has received limited attention: the decoding of neural signals, and specifically the decoding of information from the spiking activity of single neurons or clusters of neurons. While recurrent neural networks are an obvious fit for this domain due to the temporal element, I will also show that convolutional networks convolving over time windows can also perform the decoding operation. First however I will better define the problem by describing the dataset that I use, along with the performance metric I have chosen and why. Then I will discuss a method that I developed to augment the relatively small dataset to reduce overfitting, and then finally I will describe my results with both recurrent and convolutonal networks, showing the relevance to deep learning to this domain, and then finally discuss why the specific benefits of being able to implement these algorithms on simple, power efficient hardware

7.1.1 Predicting Task Relevant Stimulus Frequency Change from Rat Auditory Cortex Spike Data

While I have not implemented LSTM or GRU network layers for the described hardware yet, it is meant to run these algorithms as well as convolutional network layers. In order to show the versatility of recurrent network layers, and show the value of running these network types on the hardware due to their wide applicability I apply gated recurrent networks to a new domain. Specifically, I use gated recurrent unit networks to predict the stimuli that coincided with the observed activity of real neurons. GRU networks were trained using a dataset consisting of the observed spikes from a small number (10 to 16) of clusters of neurons in the rat primary auditory cortex (A1) recorded while the animals performed a frequency discrimination task. The task stimuli consisted of a variable length series of identical single frequency tones, followed by a "target" tone of a slightly different, single frequency. The animals were rewarded if they responded correctly to this target stimulus. Additional details regarding the behavioral task and the recording of the neural data found in this dataset can be found in Dodd et al. (sion).

Previous analysis of this data show that the coincident stimulus can be predicted from the observed neural activity using both the PSTH based classifier Foffani and Moxon (2004) discussed previously and a support vector machine (SVM) based clas-

sifier. The more powerful SVM based classifier was shown to be more accurate than the PSTH classifier (Dodd et al., sion, Houck, 2012), however there is much room for improvement over the performance of both classifiers.

Intuition Behind Measure of Classifier Performance

Due to the fact that in the dataset I use examples of neural responses to tones that were the same as the previous tone are much more numerous than responses to tones that were different. Thus, a degenerate classifier could achieve high accuracy simply by assigning the most common label to every example ("reference"/"same", in this case). This problem is a fairly common one in a wide variety of domains, and performance measures have been developed to provide a more meaningful representation of classifier or forecast accuracy. One such measure is called Peirce Skill Score (PSS) (Peirce, 1884), which is defined as follows for a two label classification problem:

$$PSS = \frac{hits}{hits + misses} - \frac{false\_alarms}{false\_alarms + correct\_rejections} \tag{7.1}$$

Here *hits* are defined as the number of instances where the classifier correctly assigned the label 'True' to an example of class $True$. A *miss* is defined as the number of times that the label 'False' was assigned to an example of class 'True'. Conversely, the term *false_ alarms* is the total number of times that the classifier assigned the label 'True' to an example for class 'False', and *correct_ rejections* represents the number of times that the classifier correctly assigned a label of "False" to an example of class "False". Thus the first term in Equation 7.1 represents the percentage of examples of class "True" that the classifier correctly identified. The second term is the proportion of examples of class 'False' that the classifier incorrectly identified, which can can also be written as $1 - percent\_correct(False)$. Thus PSS for a binary classifier can also be written as:

$$PSS = percent\_correct(True) + percent\_correct(False) - 1 \tag{7.2}$$

A perfect classifier would have a PSS of 1 $(1 + 1 - 1)$, and truly random classifier a PSS of 0 $(0.5 + 0.5 - 1)$. However, unlike when the total percentage correct is used as the performance measure, a classifier that always assigns the most common label will also have a PSS of zero, regardless of how unbalanced the labels are. While PSS

is not the only such metric for solving this problem, the selection of some such metric is necessary in order to represent classifier accuracy for problems in which one label, often the label of interest, is extremely uncommon.

7.2 Method for augmenting training data for neural spike datasets

Large amounts of training data is crucial to training neural network effectively. While in some domains such as computer vision or speech recognition large amounts of data can be pulled from the internet, often already labeled or at least easily labeled by humans without specially skills. Unfortunately, in other domains data can be harder to come by, and unfortunately in the case of neural spike datasets this is often the case, and thus any method to create representative synthetic data would be helpful.

Overfitting and Dataset Extension

A specific challenge in the analysis of this data is the relatively small amount fo data that was recorded simultaneously from any given set of neurons. Data was recorded from the same animal in multiple sessions spanning days or even weeks using a chronically implanted electrode array, as described in Sloan (2009). However, there is no guarantee that the electrodes were detecting the activity of the same group of neurons during subsequent days/sessions. Previous attempts to concatenate multiple sessions from the same animal into one dataset to train the previously mentioned nearest neighbor or SVM classifiers were unsuccessful, likely due to this inconsistency.

Due to the lack of feasible method to combine the individual sessions into one large dataset, any deep learning classifier trained on this data will need to learn network parameters using the small number of training examples available from a single session once validation and tests sets have been removed. Unfortunately, when only provided with this small training set even very small gated recurrent networks simply memorize the training set within a few epochs, while failing to generalize to high performance on the validation set. Figure 7.1 shows an example of such overfitting, with the network achieving 100% accuracy on the training set, while validation set performance actually dropped.

One logical approach for eliminating this rather dramatic case of overfitting is to use the available training examples to generate "artificial" training examples that

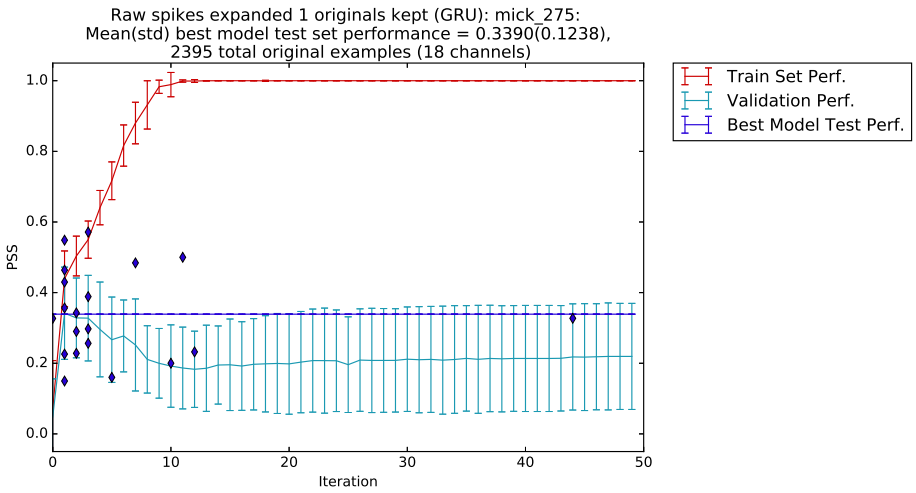

Figure 7.1: Example GRU Network training (red) and validation (aqua) set performance by epoch, averaged over 20 fold cross validation. The blue diamonds represent test set performance for the epoch with the best validation set performance within each fold, and the blue line the mean test set performance. This shows the importance of augmenting the small initial training set in order to achieve better generalization performance.

provide a larger training set that is still representative of real neural activity patterns observed in conjunction with each stimulus type. Such approaches are commonly employed on image training sets for convolutional networks. Simard et al. (2003) discuss a range of such transforms that could be used for images, from simple, small translations, to complex nonlinear warping of the original training images. While a wide range of transformations are presented as useful, the stipulation is that the transformation create new, artificial, training examples whose occurrence would be feasible if the original training set were larger.

The challenge for applying a similar approach for expanding a dataset consisting of the spiking activity of a small number neurons or clusters of neurons as opposed to pixel values in an image is selecting an appropriate transformation with which to create new examples. One possibility would be to add noise in some form to the existing training examples, creating several versions of each example, each with slight variations.

However, other, unique properties of the dataset might yield transforms that provide more variation in the expanded training set, while not producing unrealistic

examples that bear no resemblance to neural activity that would actually be observed. One such potential property is the fact that individual neurons don't always spike reliably in response to a given input

Specifically, I take advantage of the fact that spikes were recorded from multiple multi-unit clusters at once. I create new training examples for stimulus class by randomly sampling a sequence of spikes for each electrode channel from the pool of all original sequences recorded from that electrode concurrent with a stimulus from that class. This works on the assumption that correlations in activity between these electrode channels that were important to the encoding of the stimulus property in question (same different) were present across all instances of a given stimulus label, modulo some level of noise in the data. Thus these relevant correlations would be preserved in the constructed training examples. If this theory is true then one would expect overfitting to be reduced and performance increased by augmenting the training set with examples built by independently selecting the spike-train for each channel based on all observed responses on that channel in response to a stimulus with the example's label.

7.3 Recurrent networks for decoding auditory cortex data

While recorded neural spike activity in response to a stimulus can be have it dimensionallity reduced to aid in classification by binning or other methods that reduce the temporal nature of the data, the signal is still by it's nature a time series. Thus, while convolutional networks can be used wit time series data, recurrent networks are by their nature designed are input in serial form, whether a time series or another type of sequence, with gated networks being most suited for data where relations between input values at points further apart in the series might be important.

Network Architecture

Due to the relatively small nature of the data, the gated recurrent networks used here to perform the classification task described above are quite small be modern terms. Specifically, two layers of Gated Recurrent Unit (Cho et al., 2014) cells were used, the first with eight units and the second with four. On top of that is a two output linear softmax layer with a batch size of 400, with a learning rate of 0.01.

Spikes from all channels that were observed in the first 50ms after stimulus onset were used as input to the network. The spikes from each channel were represented as a binary vector with a resolution of 1ms, with zeros for all times where no spikes were observed, and ones at timesteps where a spike was observed. Separate networks where these spikes were summed into five 10ms bins were also trained, to mirror previous work on the dataset with classifiers that could not handle the dimensionality created by the higher resolution and to test whether the gated recurrent networks can take advantage of that higher resolution.

The networks were trained using Theano (Al-Rfou et al., 2016), with layer implementations from the Lasagne framework (Dieleman et al., 2015).

Results

In all I trained GRU networks using the parameters described above on 28 different sessions from the dataset, from four different animals and seven separate days each. For each session I trained networks using two temporal resolutions for each channel of spike data; raw spike times encoded with zeros (no spike) or ones(spike) in a $1ms$ resolution vector, and the same spike data binned into five $10ms$ bins across the $50ms$ window.

In addition to training networks using the original training set examples for each fold I also trained separate networks using a training set expanded to 16 times the size of the original training set using the electrode channel sampling method described previously. Finally, as a control I trained networks for each session and at each temporal resolution using training data where the electrode channels were shuffled using the same method as the $16x$ expanded training sets, but where the total number of training examples was kept equal to the original. Table 7.1 shows the mean performance of the networks trained on both temporal resolutions and all three training set variations.

When trained with the larger training set created by augmenting the training set the networks outperform those trained on the original training data when tested on held out examples for each fold. While expanding the training data $16x$ may or may not be optimal, this shows that this method of training set augmentation is potentially helpful for this type of data. Specifically, since the networks are also no longer able to achieve 100% performance on the training set, as shown in Figure 7.2,

	Orig Data	16x Data	Shuffled Channels Only
$10ms$ bins	0.1739	0.2474	0.2155
Raw spikes	0.1711	0.2390	0.2103

Table 7.1: Performance (PSS) of GRU networks on all three training set variations, for both temporal resolutions. Using a paired t-test across all sessions and training set variations there is not a statistically significant difference between the performance of the classifiers on the binned data compared to the full resolution data ($p = 0.1972$). This shows that the full temporal resolution of the data might not be needed, meaning networks that process fewer total timesteps and are faster to train and to run can perform as well as those that use more, higher resolution timesteps.

the overfitting problem has been mitigated. While information might still be present in the relationship between the spiking activity observed on each electrode channel, the loss of this information has less impact than the gain in network generalization performance afforded by the augmentation of the training set.

If relevant information were being lost by shuffling the channels between examples, and this loss were simply being made up for by the gains found in reducing overfitting, then one would expect the networks trained on channel shuffled training data where the total number of training examples was not increased to perform worse than networks trained on the original data for that session. Surprisingly however, this is not the case. Simply shuffling the channels within training examples that share the same classification label improves performance. This occurs despite the fact that the networks overfit the training data similarly to those trained on the original training examples. As shown in the example learning curve in Figure 7.3 the networks still essentially memorize the training set, typically achieving 100% performance.

This surprising result implies that information useful to training the networks may not be being lost at all by shuffling the electrode channels within each classification label.

Overall, networks trained on both the expanded and merely channel shuffled both outperform networks trained on the original training data. Table 7.2 shows the p-values for paired t-tests between each session compared to training on the original training sets, for both temporal resolutions and training set variations.

The classifiers trained with a 16x augmented training set show no statistically significant difference in performance compared to SVMs trained on the original dataset

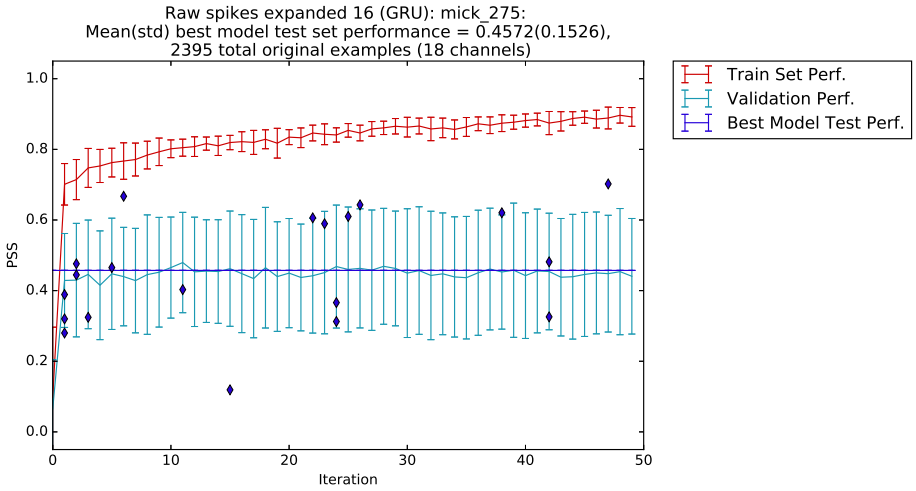

Figure 7.2: Example GRU Network training (red) and validation (aqua) set performance by epoch, averaged over 20 fold cross validation. The blue diamonds represent test set performance for the epoch with the best validation set performance within each fold, and the blue line the mean test set performance. This network was training on training examples where each channel had been randomly sampled from a pool of all original examples with that label. The total size of the training set is 16x the original. This shows that even simple methods of training set augmentation are useful for successfully applying gated recurrent network algorithms to relatively small neuroscience datasets.

binned with 50ms bins (RBF kernel, C=1, gamma determined automatically by sklearn's implementation). I also trained SVMs with a channel shuffled training set the same size as the original training set, which unlike for the GRU network classifiers did not improve performance.

7.4 Conclusion

In conclusion, this work shows the usefulness of neural networks in interpreting neural spike timeseries, along with a potential tool for increasing available training set size. This highlights the potential usefulness of neural network inference hardware for running models to interpret such data both in a research context and eventually potentially for the control of various prosthetics. Additionally, this highlights the potential of neural networks in other signal processing fields that might not have the gigantic amounts of data available that things like speech or image processing do, but

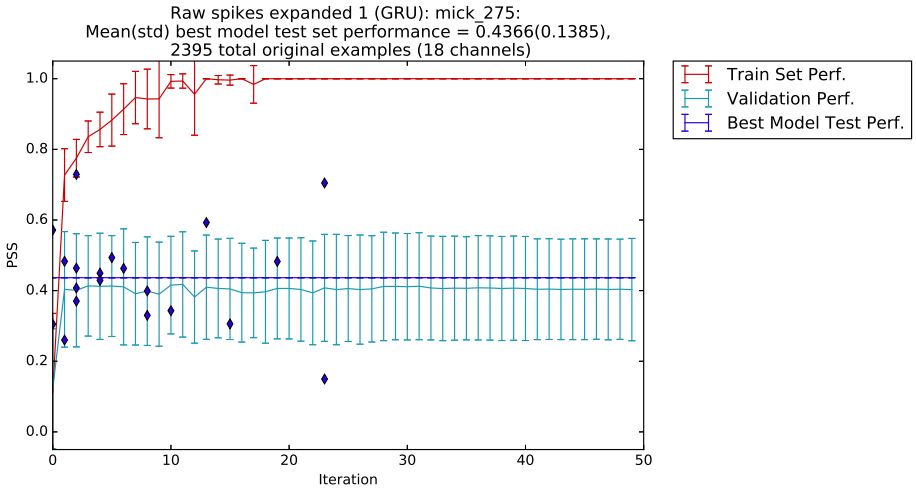

Figure 7.3: Example GRU Network training (red) and validation (aqua) set performance by epoch, averaged over 20 fold cross validation. The blue diamonds represent test set performance for the epoch with the best validation set performance within each fold, and the blue line the mean test set performance. This network was training on training examples where each channel had been randomly sampled from a pool of all original examples with that label. The total size of the training set did not change however. This implies that while increasing the size of the training set is helpful the simple step of shuffling the channels within each label in the training data helps improve decoding performance.

that might have significant economic, scientific, or humanitarian value.

	16x Data	Shuffled Channels Only
$10ms$ bins	5.5660×10^{-8}	5.2323×10^{-7}
Raw spikes	1.9393×10^{-7}	2.6867×10^{-5}

Table 7.2: Paired t-test p-values. Tests compared network performance on pairs of sessions between networks trained on the original training set and each training set variation. The consistent increase in performance confirms that the channel shuffling method is helpful for this type of data, making it's further investigation worthwhile.

Chapter 8

Discussion and Future Work

This work presents algorithms for running various convolutional and recurrent neural network algorithms on a novel class of simple but very wide SIMD hardware architectures. It shows the application of convolutional and recurrent neural networks to new domains in computational neuroscience. However, there are further questions that can be answered regarding both the types of hardware architectures on which the M^3inM^2VConv algorithm performs well, and exploring new domains that can take advantage of the speed and simplicity of the class of ultra-wide SIMD on which M^3inM^2VConv excels.

8.1 Exploring Performance on Other Hardware Types

While most other hardware types, such as GPUs and Google's TPU were designed with other algorithms in mind, if implementing M^3inM^2VConv is practical then doing so still might be useful, if for nothing else than to learn what hardware aspects result in M^3inM^2VConv being efficient.

8.2 Other domains - Cybersecurity applications such as anomaly and malware detection

Another area that could be explored is other domains to which neural networks are being or could be applied and that would benefit from the use of hardware on which the M^3inM^2V algorithms excel. One example of such a domain of increasing social and economic importance where machine learning and neural networks have the potential to play an increasing role is the domain of cybersecurity. Some common subdomains of cybersecurity where machine learning has proven useful include source code/binary analysis, intrusion/anomaly detection, and penetration testing (Hu et al., 2020, Goh, 2021). To narrow the scope I will focus on possible applications to source code and binary analysis and intrusion detection, along with special considerations for the IoT domain.

8.2.1 Source and Binary Code Analysis

The first application of machine learning to cybersecurity for which the applicability of M^3inM^2V neural network implementations will be discussed is the area of source code and binary analysis. This is important for the detection of embedded malware, software vulnerabilities, and even potentially hardware bug exploits (Xue et al., 2019), all of which have potential for both significant economic cost (e.g. ransomware), the disruption of critical infrastructure, and/or the compromise of corporate secrets or personal privacy. One recent high profile example is the Solar Winds hack, where software from what should have been a trusted source was compromised (Alkhadra et al., 2021). This incident has furthered the demand of a "Software Bill of Materials" detailing exactly which libraries and other third party components a software package contains.

Some current subdomains of source or binary code analysis to which machine learning have been applied include malware detection and function identification, using both techniques borrowed from natural language processing (NLP), and techniques that take advantage of software's formal structure. (Xue et al., 2019). Specific neural network based tools include one by Guo et al. (2018) that uses recurrent networks for both malware detection and binary code analysis, and Li et al. (2021) perform binary analysis using a neural network model based on one used for NLP.

While machine learning is a clearly a valuable tool in the domains of source code and even binary analysis, as this is frequently a development, or at least deployment time activity, and as such, it stands to benefit less from speed and efficiency gains than algorithms that need real time, or near real time results. There is still some potential benefit however, especially since any power savings for routine tasks supports green computing initiatives. Additionally, if ultra-wide SIMD accelerators geared towards M^3inM^2V algorithms are present on end user devices then the use of acceleration could allow of robust virus and malware scanning with minimal impact on system performance and if applicable, battery life.

8.2.2 Intrusion and Anomaly Detection

Another domain where the acceleration of machine learning algorithms would have greater benefit is machine learning for intrusion and anomaly detection. That is, the

real time detection of malware activity and/or compromised systems by analyzing network traffic, application behavior, system logs and other sources for malicious or simply unusual behavior. While at it's core the Solar Winds hack was a software supply chain issue, it's recognized that network monitoring and anomaly detection can still provide a last line of defense against compromised software (Alkhadra et al., 2021), helping to identify in real time unexpected and potentially harmful behavior by compromised applications. Additionally, intrusion/anomaly detection can identify unauthorized system access from other sources, including previously unknown exploits, malicious users, or accounts compromised through social engineering. One recent example of such a compromise is the Colonial Pipeline ransomware attack, where a stolen VPN password ultimately lead to a pipeline shutdown and major fuel shortages on the east coast of the United States. (Turton and Mehrotra, 2021).

While accuracy is always valued, such detection systems don't have to be perfect. Existing heuristic based Security Information and Event Monitoring (SIEM) systems already have a high false positive rate, and if a human-in-the-loop approach is maintained, even a reduced number of such false positives is an improvement, as current systems often create too many reports for human analysis to properly address (Feng et al., 2017).

However, for machine learning based intrusion/anomaly detection systems to be used they first must be trained, and one challenge in using neural networks or machine learning in general for network monitoring applications is the lack of standardized training data, both due to a general hesitancy to release datasets that could reveal information about private networks, and a wide range of types of information that could be included in a dataset, along with different ways to represent that data. However, there has been a recent effort to generate useful public datasets for training intrusion and anomaly detection systems, using both real and synthetic data and often focusing on different subdomains (Ring et al., 2019). These datasets have allowed for the training of various network intrusion/anomaly detection systems with different specializations, include detecting DDOS attacks (Elsayed et al., 2020), high risk users (Feng et al., 2017),

The (near) real time nature of intrusion and anomaly detection creates a high potential benefit from the use of accelerators. Additionally, in many cases efficiency is important, if one does not want to have to dedicate a significant amount of a

system's resources to security scanning, or dedicate too much power, data center space, etc to single purpose scanning appliances.

8.2.3 IoT and Edge Security

Most domains of cybersecurity apply in some form to IoT, often with the added challenge of limited power, compute capability and/or network bandwidth. However, the need for security at (or near) the edge grows with the growing deployment of IoT devices in applications ranging from safety critical domains such as medical, automotive, or critical infrastructure; to the ever growing list of appliances and consumer devices that are only available in an internet connected form, and whose security and privacy, while generally not a matter of life and death, is still very important.

Because power is important at the edge, or even for an "edge" server, any security application that actually runs at the edge can stand to benefit from the potential power savings of simple, ultra-wide SIMD hardware and M^3inM^2V implementations of neural network algorithms. While most code and binary analysis functions can be offloaded to the cloud, only allowing previously verified software to be installed on IoT devices, offloading real-time monitoring to the cloud is more difficult, as the necessary data must be uploaded, both using potentially limited bandwidth and, depending on the application, transferring potentially private data to the cloud. Additionally, for some applications, the need for more sophisticated machine learning tools is potentially greater, as a sufficiently skilled human may not be readily available to monitor the output of a SIEM system. Thus, a system with a low false positive rate that can either send easy to understand alerts to a untrained user or take action on it's own may be required.

Chapter 9

Conclusion

This chapter summarizes the contributions of this dissertation, and its broader impact with respect to both machine learning accelerator hardware and machine learning applications.

9.1 Contributions

This document starts off with a discussion of various machine learning hardware acceleration schemes, both hypothetical and implemented, for neural network inference. An overview of the design scheme for the ultra-wide SIMD accelerator, "ncore", for which the later described M^3inM^2V neural network algorithm implementations are written is then presented, this establishes the concept of a new class of SIMD arithmetic accelerator, for which new neural network implementations are needed in order to take advantage of the new design.

Next, the basic convolution inference M^3inM^2V algorithm itself is described, showing that convolutional network layer inference can be performance efficiently on the previously described ultra-wide SIMD hardware, and providing the basis of the implementation of other convolution variations, Next, algorithms for several such variations on a basic convolutional layer are described, including strided and dilated convolution. Following the introduction of the M^3inM^2V convolutional algorithm and some practical variations, an analysis of their performance characteristics is presented, both on hardware similar to the ultra-wide SIMD ncore architecture for which the algorithm was designed, and for comparison, on $x86\_64$ using AVX-512. This analysis shows that convolution can be preformed with high computational efficiency and memory requirements reasonable to the hardware design, showing that the previously described ultra-wide SIMD hardware design as a viable approach to a co-processor style, SoC integrated neural network accelerator.

Then, covering another major class of layers used by modern neural networks, a description of a M^3inM^2V approach to inference for recurrent network layers is presented, thus showing that another major class of neural network inference layer is implementable on the ultra-wide SIMD hardware architecture in question. Next, the

performance characteristics of these proposed recurrent network implementations on hardware designs fitting in the ultra-wide SIMD ncore architecture class are presented, showing once again that such hardware designs are viable from running inference on networks that contain recurrent layers.

Finally, to illustrate to breadth of domains in which neural networks, and specifically, the proposed M^3inM^2V algorithms, a summary is presented of work that I performed applying recurrent and convolutional network algorithms to computational neuroscience, decoding information contained in neural activity recorded in rodent sensory systems. Then, to further emphasize the diversity of the domains to which neural networks can be applied I give an overview of current applications of neural networks in cybersecurity, as well as future applications to which M^3inM^2V and hardware for which it is designed might be useful.

9.2 Broader Impact

More broadly, this dissertation establishes two main sets of characteristics: those of ultra-wide SIMD hardware on which M^3inM^2V algorithms perform well, and those of hardware on which M^3inM^2V algorithms may not be the best choice. Specifically, hardware on which M^3inM^2V performs well is capable of extremely wide SIMD operations, of the order of hundreds, or better yet, thousands, or elements wide. Furthermore, such hardware needs to be pipeline in such a way that arithmetic and supported localized data movement operations can be performed simultaneously to minimize or avoid wasted clock cycles where no arithmetic is performed while waiting for the result of a data operation, as seen in the AVX-512 example. Furthermore, while it is possible to implement M^3inM^2V on hardware that in addition to the above features also supports global memory scatter/gather capabilities that allow for matrix elements to be efficiently fed to arithmetic operations regardless of their locations in memory or geometric context in relation to other data elements, existing algorithms likely exist for such hardware that take advantage of a wider range of data movement options. This however leads to much more complex hardware features, which hardware designed with M^3inM^2V can avoid if such data movement features aren't needed for other purposes.

Of course, one must ask why are such hardware designs worth looking at. A pri-

mary reason is that the ultra-wide SIMD designs that lend themselves to M^3inM^2V algorithms provide another tool for hardware engineers designing systems that must run machine learning, or potentially other matrix math heavy algorithms, efficiently. While in some cases more complex parallel designs, such as those used in the TPU or in GPUs might be appropriate, having other options available allows designers to pick an approach (or combination of approaches) that best fit the cost, performance and power consumption needs of the application and physical constraints of the SoC or hardware device in question.

9.3 Next Steps: Continued Development

The algorithms introduced in this work shows that neural network inference can be performed on the described class of ultra-wide SIMD hardware, despite the hardware's limited data manipulation capability, establishing the hardware design as a viable option for neural network inference. This section takes a step back and addresses the importance of edge-based AI accelerators, the advantages of having them on the same SoC as the general purpose CPU, and why M^3inM^2V and the ultra-wide SIMD design paradigm it enables provide a promising approach to such on-die AI accelerators. Additionally reasons why such hardware might be useful even in an "unlimited power" datacenter environment are discussed.

The use of AI accelerators is important for edge applications for multiple reasons, including bandwidth, latency, data privacy, and security. AI accelerators at the edge enable use of neural network algorithms even when bandwidth is not available to send data to centralized cloud servers, when network latency is too large for the application, or when the application requires the system be robust to loss of network connectivity. In an extreme case, such as critical infrastructure, you might even have an air-gapped system that is not connected to the internet at all. Additionally, even when a system has a sufficient internet connection, acceleration at the edge protects privacy by reducing the amount of data that must be sent to the cloud, ideally reducing it to zero if the intended operation is a purely local one (e.g. "turn on the lights" instead of "get today's weather forecast"). Not only is this important from an ethical standpoint, but growing public interest, and in some countries, regulation regarding privacy such as the General Data Protection Regulation (GDPR) in Europe,

will drive demand for systems that avoid sending unnecessary data to the cloud where it must be protected.

However, to move AI inference to the edge, suitable hardware must be available. The hardware limitations of many edge applications favor SoC integrated AI accelerators, the most prominent of these limitations being size and power. Even edge "servers" that can rely on grid power might now have have at least soft power or space limitations. For example, a smart home hub designed to protect privacy by processing local sensor data instead of sending it to the cloud might not be popular with consumers if it required a large form factor to accommodate GPU-like accelerator cards, made too much noise, or generated too much heat. Other applications might have very little leeway if there are hard power or size limitations due to battery or form factor considerations.

Of course, placing an AI accelerator of the same die as a general purpose CPU places limits on the accelerator design, and thus favors simple and flexible designs. The M^3inM^2V algorithms introduced in this document, combined with the ultra-wide SIMD hardware design paradigm, provide a practical way to include a such a flexible neural network accelerator on the same die as a high power CPU (Henry et al., 2020). Specifically, the ring-based ultra-wide SIMD and M^3inM^2V combination described in this work offer a scalable approach that be parameterized to fit the available space on an SoC. This approach is possible because the arithmetic unit and associated data handling and storage hardware form a simple, replicatable unit, and the hardware size grows essentially linearly with number of arithmetic units, with a small amount of extra overhead for the distribution of control signals (Henry, 2020). The only requirement imposed by software on SIMD width is that M^3inM^2VConv requires a power-of-two width for the block scheme to work. In addition to allowing the SIMD width to be scaled to fit the hardware size supported by a particular SoC design, the individual unit based approach to the design allows for the potential to add additional operations, such as logical comparisons or bitwise operations, to each unit without modifying the overall design paradigm. Such operations would be much harder with the less flexible systolic array approach, which is built around multiply-add operations only, often with completely separate data handling. As long as these additional features are physically small, they have minimal effect on the hardware layout, but enable much more flexible software with the ability to support a wider

range of functionality. This design yields software that is more likely to be able to run any future neural network design, or even more general forms of signal processing, with less chance of having even a single new "exotic" operation not supported by hardware that as a result requires an entire network layer to be transferred to the general purpose CPU and back again.

Finally, while likely most valuable at the edge, on die AI accelerators also have a place in datacenter applications, for two reasons. First, it simplifies the system, as there is no GPU or other separate hardware to buy, make space for in a chassis, power, and cool - all things which require money, space, and energy. Thus on-die AI accelerators lend themselves to cheaper and "greener" datacenter operations than discrete accelerator based inference. Secondly, on-die accelerators allow for closer integration with the CPU. This reduces latency and allows for more complex yet performant software that performs tightly coupled operations on both the accelerator and standard CPU hardware, since data transfer from an on-die accelerator, while potentially time consuming, should still be significantly faster then transferring the same data from a separate, discrete accelerator.

Therefore, the combination of a new class of simple, but ultra-wide SIMD hardware designs, and the accompanying M^3inM^2V algorithms presented in this dissertation, improves the efficiency, and even feasibility, of a broad spectrum of neural network inference applications, thus laying a foundation for the expansion edge-based AI inference. Additionally, by enabling improved on-chip AI acceleration of edge applications, the algorithms described in this document contribute technical solutions addressing a range of important social, economic, and larger system engineering issues, including network bandwidth congestion, energy efficiency, and data security and privacy.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Agrawal, P., Stansbury, D., Malik, J., and Gallant, J. L. (2014). Pixels to voxels: Modeling visual representation in the human brain. *arXiv preprint arXiv:1407.5104*.

Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., Bengio, Y., Bergeron, A., Bergstra, J., Bisson, V., Bleecher Snyder, J., Bouchard, N., Boulanger-Lewandowski, N., Bouthillier, X., de Brébisson, A., Breuleux, O., Carrier, P.-L., Cho, K., Chorowski, J., Christiano, P., Cooijmans, T., Côté, M.-A., Côté, M., Courville, A., Dauphin, Y. N., Delalleau, O., Demouth, J., Desjardins, G., Dieleman, S., Dinh, L., Ducoffe, M., Dumoulin, V., Ebrahimi Kahou, S., Erhan, D., Fan, Z., Firat, O., Germain, M., Glorot, X., Goodfellow, I., Graham, M., Gulcehre, C., Hamel, P., Harlouchet, I., Heng, J.-P., Hidasi, B., Honari, S., Jain, A., Jean, S., Jia, K., Korobov, M., Kulkarni, V., Lamb, A., Lamblin, P., Larsen, E., Laurent, C., Lee, S., Lefrancois, S., Lemieux, S., Léonard, N., Lin, Z., Livezey, J. A., Lorenz, C., Lowin, J., Ma, Q., Manzagol, P.-A., Mastropietro, O., McGibbon, R. T., Memisevic, R., van Merriënboer, B., Michalski, V., Mirza, M., Orlandi, A., Pal, C., Pascanu, R., Pezeshki, M., Raffel, C., Renshaw, D., Rocklin, M., Romero, A., Roth, M., Sadowski, P., Salvatier, J., Savard, F., Schlüter, J., Schulman, J., Schwartz, G., Serban, I. V., Serdyuk, D., Shabanian, S., Simon, E., Spieckermann, S., Subramanyam, S. R., Sygnowski, J., Tanguay, J., van Tulder, G., Turian, J., Urban, S., Vincent, P., Visin, F., de Vries, H., Warde-Farley, D., Webb, D. J., Willson, M., Xu, K., Xue, L., Yao, L., Zhang, S., and Zhang, Y. (2016). Theano:

A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688.

Alkhadra, R., Abuzaid, J., AlShammari, M., and Mohammad, N. (2021). Solar winds hack: In-depth analysis and countermeasures. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–7.

Buzsaki, G. (2006). *Rhythms of the Brain*. Oxford University Press, USA.

Chellapilla, K., Puri, S., and Simard, P. (2006). High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft.

Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. (2014). Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. (2014). cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Chollet, F. (2016). Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE.

Dieleman, S., SchlÃ¼ter, J., Raffel, C., Olson, E., SÃ¸nderby, S. K., Nouri, D., Maturana, D., Thoma, M., Battenberg, E., Kelly, J., Fauw, J. D., Heilman, M., de Almeida, D. M., McFee, B., Weideman, H., TakÃ¡cs, G., de Rivaz, P., Crall, J., Sanders, G., Rasul, K., Liu, C., French, G., and Degrave, J. (2015). Lasagne: First release.

Dodd, O. T., Houck, K. C., Palmer, T. J., Sloan, A. M., Kilgard, M. P., Miller, D. P., Fagg, A. H., and Rennaker, R. L. (in revision). Neural correlates of frequency discrimination in behaving rats.

Elsayed, M. S., Le-Khac, N.-A., Dev, S., and Jurcut, A. D. (2020). Ddosnet: A deep-learning model for detecting network attacks. In *2020 IEEE 21st International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, pages 391–396.

Feng, C., Wu, S., and Liu, N. (2017). A user-centric machine learning framework for cyber security operations center. In *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 173–175.

Foffani, G. and Moxon, K. (2004). PSTH-based classification of sensory stimuli using ensembles of single neurons. *Journal of Neuroscience Methods*, 135(1-2):107 – 120.

Goh, K. C. (2021). *Toward Automated Penetration Testing Intelligently with Reinforcement Learning*. PhD thesis, Dublin, National College of Ireland.

Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2015). Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*.

Guo, W., Mu, D., Xu, J., Su, P., Wang, G., and Xing, X. (2018). Lemna: Explaining deep learning based security applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 364–379, New York, NY, USA. Association for Computing Machinery.

Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. (2016). EIE: efficient inference engine on compressed deep neural network. *CoRR*, abs/1602.01528.

Henry, G. (2020). Stanford seminar - centaur technology's deep learning coprocessor. https://www.youtube.com/watch?v=5Z7cmyYakAw, Accessed on 4/10/2022.

Henry, G., Palangpour, P., Thomson, M., Gardner, J. S., Arden, B., Donahue, J., Houck, K., Johnson, J., O'Brien, K., Petersen, S., et al. (2020). High-performance deep-learning coprocessor integrated into x86 soc with server-class cpus industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 15–26. IEEE.

Henry, G. G. (2021). From mainframes to microprocessors. *IEEE Micro*, 41(6):89–96.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780. - Claim that LSTM network can hold onto information for 1000+ timesteps.

Houck, K. (2012). Automatically identifying stimuli from firing patterns in the auditory cortex of rat. Master's thesis, University of Oklahoma.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861.

Hu, Z., Beuran, R., and Tan, Y. (2020). Automated penetration testing using deep reinforcement learning. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 2–10. IEEE.

Hubel, D. H. and Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154.

Huth, A. G., Griffiths, T. L., Theunissen, F. E., and Gallant, J. L. (2015). Pragmatic: a probabilistic and generative model of areas tiling the cortex. *arXiv preprint arXiv:1504.03622*.

Intel (2019). Intel product specifications. `https://ark.intel.com/content/www/us/en/ark.html`.

Intel (2021). *Instruction Set Reference. In Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 1,2.

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

Kanitscheider, I. and Fiete, I. (2016). Training recurrent networks to generate hypotheses about how the brain solves hard navigation problems. *arXiv preprint arXiv:1609.09059*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Lee, H., Ekanadham, C., and Ng, A. (2007). Sparse deep belief net model for visual area v2. In *Advances in neural information processing systems*, pages 873–880.

Li, X., Yu, Q., and Yin, H. (2021). Palmtree: Learning an assembly language model for instruction embedding. *arXiv preprint arXiv:2103.03809*.

Liu, S., Du, Z., Tao, J., Han, D., Luo, T., Xie, Y., Chen, Y., and Chen, T. (2016). Cambricon: An instruction set architecture for neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 393–405. IEEE Press.

Mathieu, M., Henaff, M., and LeCun, Y. (2013). Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*.

Peirce, C. (1884). The numerical measure of the success of predictions. *Science*, IV(93):453–454.

Qadeer, W., Hameed, R., Shacham, O., Venkatesan, P., Kozyrakis, C., and Horowitz, M. (2015). Convolution engine: Balancing efficiency and flexibility in specialized computing. *Communications of the ACM*, 58(4):85–93.

Ring, M., Wunderlich, S., Scheuring, D., Landes, D., and Hotho, A. (2019). A survey of network-based intrusion detection data sets. *Computers & Security*, 86:147–167.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408.

Simard, P. Y., Steinkraus, D., and Platt, J. C. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, volume 3, pages 958–962.

Sloan, A. (2009). *Spectrotemporal dynamics of neural responses in auditory cortex during frequency discrimination.* Phd dissertation, The University of Oklahoma, United States - Oklahoma.

Sloan, A., Dodd, O., and Rennaker, R. (2009). Frequency discrimination in rats measured with tone-step stimuli and discrete pure tones. *Hearing Research*, 251:60–69.

Smith, E., Kellis, S., House, P., and Greger, B. (2013). Decoding stimulus identity from multi-unit activity and local field potentials along the ventral auditory stream in the awake primate: implications for cortical neural prostheses. *Journal of Neural Engineering*, 10(1):016010.

Soleymani, M., Asghari-Esfeden, S., Pantic, M., and Fu, Y. (2014). Continuous emotion detection using eeg signals and facial expressions. In *2014 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6. IEEE.

Sussillo, D., Nuyujukian, P., Fan, J. M., Kao, J. C., Stavisky, S. D., Ryu, S., and Shenoy, K. (2012). A recurrent neural network for closed-loop intracortical brain–machine interface decoders. *Journal of neural engineering*, 9(2):026027.

tensorflow (2018). Tensorflow r1.10 (commit f2ebb29e20efb61c267de2f4008ad39d5758f29a). https://github.com/tensorflow/tensorflow/tree/f2ebb29e20efb61c267de2f4008ad39d5758f29a.

Turton, W. and Mehrotra, K. (2021). Hackers breached colonial pipeline using compromised password. *Bloomberg. Available online at: https://www.bloomberg. com/news/articles/2021-06-04/hackers-breached-colonial-pipeline-using-compromised-password.*

Xue, H., Sun, S., Venkataramani, G., and Lan, T. (2019). Machine learning-based analysis of program binaries: A comprehensive study. *IEEE Access*, 7:65889–65912.

Appendix A

AXV-512 Convolution Algorithm C Code (with intrinsics)

```
#include "immintrin.h"
#include <stdio.h>
#include <math.h>

#define TOTAL_ZMM_REGS 32
#define NUMBER_OF_INPUT_CHANNELS 2
#define NUMBER_OF_OUTPUT_CHANNELS 2
#define NUMBER_OF_CHANNELS 2
#define CHANNEL_WIDTH 8
#define CHANNEL_HEIGHT 8

// ***************************************************************************
// Unoptimized convolution one channel at a time to provide ground truth *
// ***************************************************************************
void slow_3x3x1_same_conv_8x8x2_input(float* input, float* filter_ch_1,
    float* filter_ch_2, float* output)
{
  int input_row, filter_row, output_row, output_column, filter_column,
      input_column, input_channel, i;
  for (output_row=0; output_row<CHANNEL_HEIGHT; output_row++)
  {
    for (output_column=0; output_column<CHANNEL_WIDTH; output_column++)
    {
      //zero out output element
      output[output_row*CHANNEL_WIDTH+output_column] = 0;
      //apply the filters
      for (filter_row=0; filter_row<3; filter_row++)
      {
        input_row = output_row-1+filter_row;
        // skip out of bounds input rows
        if ( (input_row < 0) || (input_row >= CHANNEL_HEIGHT) ) continue;
```

```
      for (filter_column=0; filter_column<3; filter_column++)
      {
        input_column = output_column-1+filter_column;
        // skip out of bounds columns
        if ( (input_column < 0) || (input_column >= CHANNEL_WIDTH) )
            continue;
         // add this filter element's contribution to the output
         output[output_row*CHANNEL_WIDTH+output_column] +=
              filter_ch_1[filter_row*3+filter_column] *
              input[input_row*CHANNEL_WIDTH*2+input_column]
              +
              filter_ch_2[filter_row*3+filter_column]*
              input[input_row*CHANNEL_WIDTH*2+CHANNEL_WIDTH+input_column];
      }//end for each filter column
     }//end for filter row
    }//end for each output column
  }//end for output row
}//end slow_3x3x1_same_conv_8x8x2_input



//
    ***************************************************************************
// AVX-512 Convolution - Same size, 8x8x2 input, 3x3x2 filter, 8x8x2 output
    *
//
    ***************************************************************************
void avx512_3x3xn_same_conv_8x8xn_input(float* input, float* filter, float*
    output, int number_of_input_channels)
{
  /*************************************************************
   * Declare zmm register for expanded filter                  *
   * //each filter coefficient comes from RAM preexpanded,     *
   * assume time to build these vectors more costly than RAM space *
   *************************************************************/
```

```
__m512 expanded_filters[18];

/*************************************
 * Declare zmm register for output row *
 * Only a single output row needs to be *
 * stored in the zmm registers at once. *
 *************************************/
__m512 output_2chan;

/*********************************************************
 * Declare remaining zmm registers for the input data *
 * 16 registers to hold inputs, room for 2 input channels *
 *********************************************************/
__m512 input_2chan[8];

/*******************
 * rotate indecies *
 *******************/
int channelwise_rotate_mask_array[16] __attribute__ ((aligned (64))) =
       {11,12,13,14,15,0,1,2,3,4,5,6,7,8,9,10}; //swap channels from
           partially rotated state
__m512i channelwise_rotate_mask =
    _mm512_load_epi32(channelwise_rotate_mask_array);

int filter_column_rotate_mask_array[16] __attribute__ ((aligned (64))) =
       {15,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};//rotate all elements
           left one
__m512i filter_column_rotate_mask =
    _mm512_load_epi32(filter_column_rotate_mask_array);

int filter_column_initial_rotate_mask_array[16] __attribute__ ((aligned
    (64))) =
             {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0};//rotate all
                 elements left one
```

```
    __m512i filter_column_initial_rotate_mask =
        _mm512_load_epi32(filter_column_initial_rotate_mask_array);

/***********************************
 * Load initial data into registers *
 ***********************************/
 int i;
 for (i =0; i<18; i++)
   expanded_filters[i] = _mm512_load_ps(&(filter[i*16]));

 int number_of_input_channel_blocks = ceil(number_of_input_channels/2.0);
 for (i=0; i<number_of_input_channel_blocks*8; i++)
 {
   printf("Loading row starting with %f into input register %u\n",
       input[i*16], i);
   input_2chan[i] = _mm512_load_ps(&(input[i*16]));
   //rotate block of two input channels left by one, to set up for
       rightmost element of same-size convolution
   input_2chan[i] =
       _mm512_permutexvar_ps(filter_column_initial_rotate_mask,
       input_2chan[i]);
 }//end for each input data element

/*********************************************
 * For each row of output, perform convolution *
 *********************************************/
 int out_row, input_channel_block, channel, filter_row, filter_column,
     filter_index, input_source_row;
 float zero_array[16] __attribute__ ((aligned (64))) =
     {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
 printf("Computing 8 output rows, from %d input channel blocks (%d input
     channels)\n",
       number_of_input_channel_blocks, number_of_input_channels);
 float debug_row[16] __attribute__ ((aligned (64)));
```

```
   for (out_row=0; out_row<8; out_row++)
   {
     printf("[DEBUG] output row %d\n", out_row);
     //zero output register
     output_2chan = _mm512_load_ps(zero_array);
//   for (input_channel_block=0;
    input_channel_block<number_of_input_channel_blocks;
    input_channel_block++)
//   {
     input_channel_block=0;
     for (filter_row=0; filter_row<3; filter_row++)
     {
      input_source_row = input_channel_block*8+out_row+filter_row-1;
      if ( ((out_row+filter_row)<1) || ((out_row+filter_row) > 8))
        continue;
      printf(" [DEBUG] filter row %d\n", filter_row);
       for (channel=0; channel<2; channel++)
       {
         printf("   [DEBUG] channel %d, filter_row %d, out_row %d\n",
             channel, filter_row, out_row);

        //
            ********************************************************************************
        // Un-roll the loop over filter columns since a different mask is
            needed each time
        //
            ********************************************************************************

        // *****************
        // Filter column 2 *
        // *****************
        printf("     [Filter column 2]\n");
        filter_column = 2;
         filter_index = filter_row*3*2+channel*3+(2-filter_column);
             //subtract the index from 2 since the data is packed with the
```

```
    rightmost column first
_mm512_store_ps(debug_row, expanded_filters[filter_index]);
printf("     Current filter input: [%2.1f, %2.1f, %2.1f, %2.1f,
    %2.1f, %2.1f, %2.1f, %2.1f], ",
          debug_row[0], debug_row[1], debug_row[2], debug_row[3],
          debug_row[4], debug_row[5], debug_row[6], debug_row[7]);
printf("[%2.1f, %2.1f, %2.1f, %2.1f, %2.1f, %2.1f, %2.1f, %2.1f]\n",
          debug_row[8], debug_row[9], debug_row[10], debug_row[11],
          debug_row[12], debug_row[13], debug_row[14],
              debug_row[15]);

_mm512_store_ps(debug_row, input_2chan[input_source_row]);
printf("     Current input vector: [%.2f, %.2f, %.2f, %.2f, %.2f,
    %.2f, %.2f, %.2f], ",
         debug_row[0], debug_row[1], debug_row[2], debug_row[3],
         debug_row[4], debug_row[5], debug_row[6], debug_row[7]);
printf("[%.2f, %.2f, %.2f, %.2f, %.2f, %.2f, %.2f, %.2f]\n",
         debug_row[8], debug_row[9], debug_row[10], debug_row[11],
         debug_row[12], debug_row[13], debug_row[14], debug_row[15]);

// perform multiply-add for rightmost element of conv filter row
    (column 2)
output_2chan = _mm512_mask3_fmadd_ps(input_2chan[input_source_row],
                                     expanded_filters[filter_index],
                                     output_2chan, 0x7F7F);
// rotate one element for next column
input_2chan[input_source_row] =
    _mm512_permutexvar_ps(filter_column_rotate_mask,
                                                  input_2chan[input_source_row]);

// *****************
// Filter column 1 *
// *****************
printf("     [Filter column 1]\n");
filter_column = 1;
```

```
 //subtract the index from 2 since the data is packed with the
    rightmost column first
 filter_index = filter_row*3*2+channel*3+(2-filter_column);
 _mm512_store_ps(debug_row, expanded_filters[filter_index]);
 printf("    Current filter input: [%2.1f, %2.1f, %2.1f, %2.1f,
    %2.1f, %2.1f, %2.1f, %2.1f], ",
          debug_row[0], debug_row[1], debug_row[2], debug_row[3],
          debug_row[4], debug_row[5], debug_row[6], debug_row[7]);
 printf("[%2.1f, %2.1f, %2.1f, %2.1f, %2.1f, %2.1f, %2.1f, %2.1f]\n",
          debug_row[8], debug_row[9], debug_row[10], debug_row[11],
          debug_row[12], debug_row[13], debug_row[14],
              debug_row[15]);

 _mm512_store_ps(debug_row, input_2chan[input_source_row]);
 printf("    Current input vector: [%.2f, %.2f, %.2f, %.2f, %.2f,
    %.2f, %.2f, %.2f], ",
         debug_row[0], debug_row[1], debug_row[2], debug_row[3],
         debug_row[4], debug_row[5], debug_row[6], debug_row[7]);
 printf("[%.2f, %.2f, %.2f, %.2f, %.2f, %.2f, %.2f, %.2f]\n",
         debug_row[8], debug_row[9], debug_row[10], debug_row[11],
         debug_row[12], debug_row[13], debug_row[14], debug_row[15]);

//perform multiply-add for middle element of conv filter row
output_2chan = _mm512_mask3_fmadd_ps(input_2chan[input_source_row],
                                     expanded_filters[filter_index],
                                     output_2chan, 0xFFFF);
 //rotate one element for next column
 input_2chan[input_source_row] =
     _mm512_permutexvar_ps(filter_column_rotate_mask,
                                                    input_2chan[input_source_row]);

 // *****************
 // Filter column 0 *
 // *****************
 printf("    [Filter column 0]\n");
```

```
 filter_column = 0;
 //subtract the index from 2 since the data is packed with the
    rightmost column first
 filter_index = filter_row*3*2+channel*3+(2-filter_column);
 _mm512_store_ps(debug_row, expanded_filters[filter_index]);
 printf("    Current filter input: [%2.1f, %2.1f, %2.1f, %2.1f,
    %2.1f, %2.1f, %2.1f, %2.1f], ",
          debug_row[0], debug_row[1], debug_row[2], debug_row[3],
        debug_row[4], debug_row[5], debug_row[6], debug_row[7]);
printf("[%2.1f, %2.1f, %2.1f, %2.1f, %2.1f, %2.1f, %2.1f, %2.1f]\n",
          debug_row[8], debug_row[9], debug_row[10], debug_row[11],
        debug_row[12], debug_row[13], debug_row[14], debug_row[15]);

 _mm512_store_ps(debug_row, input_2chan[input_source_row]);
 printf("    Current input vector: [%.2f, %.2f, %.2f, %.2f, %.2f,
    %.2f, %.2f, %.2f], ",
         debug_row[0], debug_row[1], debug_row[2], debug_row[3],
         debug_row[4], debug_row[5], debug_row[6], debug_row[7]);
 printf("[%.2f, %.2f, %.2f, %.2f, %.2f, %.2f, %.2f, %.2f]\n",
        debug_row[8], debug_row[9], debug_row[10], debug_row[11],
        debug_row[12], debug_row[13], debug_row[14], debug_row[15]);

 //perform multiply-add for leftmost element of conv filter row
output_2chan = _mm512_mask3_fmadd_ps(input_2chan[input_source_row],
                                     expanded_filters[filter_index],
                                     output_2chan, 0xFEFE);
//rotate one element for next column
input_2chan[input_source_row] =
    _mm512_permutexvar_ps(filter_column_rotate_mask,
                                                    input_2chan[input_source_row]);


 // ******************************************************
// rotate input row (swap channels) to align the channels
// as input to the other output channel
```

```
        // ******************************************************
         input_2chan[input_source_row] =
             _mm512_permutexvar_ps(channelwise_rotate_mask,
                                   input_2chan[input_source_row]);
      }//end for each channel
     }//end for each filter row
// }//end for each input channel block
   //store this output block
   _mm512_store_ps(&output[out_row*16], output_2chan);
   printf("completed output row %u, first element stored: %f\n", out_row,
       output[out_row*16]);
  }//end for each output row
}// end avx-512 conv




int main()
{
  float input[CHANNEL_WIDTH*CHANNEL_HEIGHT*NUMBER_OF_INPUT_CHANNELS]
      __attribute__ ((aligned (64)));
  float
      filter[3*NUMBER_OF_INPUT_CHANNELS*3*NUMBER_OF_OUTPUT_CHANNELS*CHANNEL_WIDTH]
      __attribute__ ((aligned (64)));
  float output[CHANNEL_WIDTH*CHANNEL_HEIGHT*NUMBER_OF_OUTPUT_CHANNELS]
      __attribute__ ((aligned (64)));
  float ground_truth_output_channel_0[CHANNEL_WIDTH*CHANNEL_HEIGHT];
  float ground_truth_output_channel_1[CHANNEL_WIDTH*CHANNEL_HEIGHT];

  printf("Loading values into test filter...\n");
  //load values into filter, including zeros for mask
  unsigned int row, column, i, packed_column;
  float filter_a_1[3*3] = {0, 1, 2, 10, 11, 12, 20, 21, 22};
  float filter_b_1[3*3] = {0, 2, 4, 20, 22, 24, 40, 42, 44};
  float filter_a_2[3*3] = {1, 2, 10, 11, 12, 20, 21, 22, 0};
```

```
float filter_b_2[3*3] = {2, 4, 20, 22, 24, 40, 42, 44, 0};
printf("Filter B:\n");
printf(" Input channel configuration 1:\n");
for (row=0; row<3; row++)
{
  printf("[ ");
  for (column=0; column<3; column++)
  {
     printf("   %2.1f ", filter_a_1[3*row+column]);
  }
  printf("]\n");
}
printf(" Channel 2:\n");
for (row=0; row<3; row++)
{
  printf("[ ");
  for (column=0; column<3; column++)
  {
     printf("   %2.1f ", filter_a_2[3*row+column]);
  }
  printf("]\n");
}
printf("Filter B:\n");
printf(" Input channel configuration 1:\n");
for (row=0; row<3; row++)
{
  printf("[ ");
  for (column=0; column<3; column++)
  {
     printf("   %2.1f ", filter_b_1[3*row+column]);
  }
  printf("]\n");
}
printf(" Channel 2:\n");
for (row=0; row<3; row++)
```

```
{
  printf("[ ");
  for (column=0; column<3; column++)
  {
     printf("   %2.1f ", filter_b_2[3*row+column]);
  }
  printf("]\n");
}


printf("Loading values into test filter...\n");
for (i=0;
    i<3*NUMBER_OF_INPUT_CHANNELS*3*NUMBER_OF_OUTPUT_CHANNELS*CHANNEL_WIDTH;
    i++) filter[i] = 0;
for (row=0; row<3; row++)
{
  for (column=0; column<3; column++)
  {
     packed_column = 2-column;
     for (i=0; i<CHANNEL_WIDTH; i++)
     {
       filter[(2*row)*3*CHANNEL_WIDTH*NUMBER_OF_INPUT_CHANNELS +
     packed_column*CHANNEL_WIDTH*NUMBER_OF_INPUT_CHANNELS + i] =
         filter_a_1[3*row+column];

  filter[(2*row)*3*CHANNEL_WIDTH*NUMBER_OF_INPUT_CHANNELS +
     packed_column*CHANNEL_WIDTH*NUMBER_OF_INPUT_CHANNELS + CHANNEL_WIDTH
         + i] = filter_a_2[3*row+column];

  filter[(2*row+1)*3*CHANNEL_WIDTH*NUMBER_OF_INPUT_CHANNELS +
     packed_column*CHANNEL_WIDTH*NUMBER_OF_INPUT_CHANNELS + i] =
         filter_b_1[3*row+column];

  filter[(2*row+1)*3*CHANNEL_WIDTH*NUMBER_OF_INPUT_CHANNELS +
```

```
      packed_column*CHANNEL_WIDTH*NUMBER_OF_INPUT_CHANNELS + CHANNEL_WIDTH
          + i] = filter_b_2[3*row+column];
      }
    }
}

printf("Expanded filters:\n");
for (row=0; row<18; row++)
{
  if (row%2 == 0)
    printf("   Row %d, filter A, both channels: [ ", row);
  else
    printf("   Row %d, filter B, both channels: [ ", row);
  for (i=0; i<CHANNEL_WIDTH*NUMBER_OF_INPUT_CHANNELS; i++)
  {
    printf("%2.0f ", filter[row*NUMBER_OF_INPUT_CHANNELS*CHANNEL_WIDTH+i]);
  }
  printf("]\n");
}

printf("Loading fake input data...\n");
for (i=0; i<CHANNEL_WIDTH*CHANNEL_HEIGHT*NUMBER_OF_INPUT_CHANNELS; i++)
{
  input[i] = (i%16)*0.1 + (i/(CHANNEL_WIDTH*NUMBER_OF_INPUT_CHANNELS));
}//end for i
int channel;
for (row=0; row<CHANNEL_HEIGHT; row++)
{
  printf("Row %d: ", row);
  for (channel=0; channel<NUMBER_OF_INPUT_CHANNELS; channel++)
  {
    printf("[ ");
    for (column=0; column<CHANNEL_WIDTH; column++)
    {
      printf("%2.2f ", input[row*NUMBER_OF_INPUT_CHANNELS*CHANNEL_WIDTH +
```

```
            channel*CHANNEL_WIDTH + column]);
    }//end for each column
    printf("] ");
  }//end for each channel
  printf("\n");
}//end for each row

printf("Computing ground truth...\n");
slow_3x3x1_same_conv_8x8x2_input(input, filter_a_1, filter_b_1,
    ground_truth_output_channel_0);
//note that the filters are swapped for output channel 2, since in the
    original input channel
//configuration channel 1 is convolved with filter b, but for this test
    case swapping the filters
//is easier
slow_3x3x1_same_conv_8x8x2_input(input, filter_b_2, filter_a_2,
    ground_truth_output_channel_1);

printf("Running conv...\n");
avx512_3x3xn_same_conv_8x8xn_input(input, filter, output,
    NUMBER_OF_INPUT_CHANNELS);

printf("Checking the results...\n");
int correct = 0;
float ground_truth_conv_value;
int output_channel;
for (i=0; i < CHANNEL_WIDTH*CHANNEL_HEIGHT*NUMBER_OF_OUTPUT_CHANNELS; i++)
{
  printf("output[%u] = %4.4f ", i, output[i]);
  if (i%(NUMBER_OF_OUTPUT_CHANNELS*CHANNEL_WIDTH) < CHANNEL_WIDTH)
  {
    ground_truth_conv_value =
       ground_truth_output_channel_0[CHANNEL_WIDTH*(i/(NUMBER_OF_OUTPUT_CHANNELS*CHANNEL_WI
          +
                                    i%(NUMBER_OF_OUTPUT_CHANNELS*CHANNEL_WIDTH)];
```

```
      output_channel = 0;
    }//end if channel 1
    else
    {
      ground_truth_conv_value =
             ground_truth_output_channel_1[CHANNEL_WIDTH*(i/(NUMBER_OF_OUTPUT_CHANNELS*CHANNE
                 +
                                       i%(NUMBER_OF_OUTPUT_CHANNELS*CHANNEL_WIDTH)
                                           - CHANNEL_WIDTH];
      output_channel = 1;
    }//end if channel 2
    //check if value correct
    if (abs(ground_truth_conv_value - output[i]) < 0.0001)
    {
      printf(" ~= %4.4f (Correct)", ground_truth_conv_value);
      correct++;
    }//end if correct
    else printf(" != %4.4f (WRONG)", ground_truth_conv_value);

    printf(" (Output channel %d, row %d)\n", output_channel,
        i/(NUMBER_OF_OUTPUT_CHANNELS*CHANNEL_WIDTH));

  }//end for i (check)

  printf("%d out of %d output element correct!\n", correct,
      CHANNEL_WIDTH*CHANNEL_HEIGHT*NUMBER_OF_OUTPUT_CHANNELS);

  return 1;
}//end main
```

Appendix B

AXV-512 Convolution Algorithm C Run Log

```
Loading values into test filter...
Filter B:
  Input channel configuration 1:
[     0.0     1.0     2.0 ]
[     10.0     11.0     12.0 ]
[     20.0     21.0     22.0 ]
  Channel 2:
[     1.0     2.0     10.0 ]
[     11.0     12.0     20.0 ]
[     21.0     22.0     0.0 ]
Filter B:
  Input channel configuration 1:
[     0.0     2.0     4.0 ]
[     20.0     22.0     24.0 ]
[     40.0     42.0     44.0 ]
  Channel 2:
[     2.0     4.0     20.0 ]
[     22.0     24.0     40.0 ]
[     42.0     44.0     0.0 ]
Loading values into test filter...
Expanded filters:
    Row 0, filter A, both channels: [  2  2  2  2  2  2  2  2 10 10 10 10 10 10 10 10
    Row 1, filter B, both channels: [  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2
    Row 2, filter A, both channels: [  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1
    Row 3, filter B, both channels: [  4  4  4  4  4  4  4  4 20 20 20 20 20 20 20 20
    Row 4, filter A, both channels: [  2  2  2  2  2  2  2  2  4  4  4  4  4  4  4  4
    Row 5, filter B, both channels: [  0  0  0  0  0  0  0  0  2  2  2  2  2  2  2  2
    Row 6, filter A, both channels: [ 12 12 12 12 12 12 12 12 20 20 20 20 20 20 20 20
    Row 7, filter B, both channels: [ 11 11 11 11 11 11 11 11 12 12 12 12 12 12 12 12
    Row 8, filter A, both channels: [ 10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11
```

```
    Row 9, filter B, both channels: [ 24 24 24 24 24 24 24 24 40 40 40 40 40 40 40 40
    Row 10, filter A, both channels: [ 22 22 22 22 22 22 22 22 24 24 24 24 24 24 24 2
    Row 11, filter B, both channels: [ 20 20 20 20 20 20 20 20 22 22 22 22 22 22 22 2
    Row 12, filter A, both channels: [ 22 22 22 22 22 22 22 22  0  0  0  0  0  0  0
    Row 13, filter B, both channels: [ 21 21 21 21 21 21 21 21 22 22 22 22 22 22 22 2
    Row 14, filter A, both channels: [ 20 20 20 20 20 20 20 20 21 21 21 21 21 21 21 2
    Row 15, filter B, both channels: [ 44 44 44 44 44 44 44 44  0  0  0  0  0  0  0
    Row 16, filter A, both channels: [ 42 42 42 42 42 42 42 42 44 44 44 44 44 44 44 4
    Row 17, filter B, both channels: [ 40 40 40 40 40 40 40 40 42 42 42 42 42 42 42 4
Loading fake input data...
Row 0: [ 0.00 0.10 0.20 0.30 0.40 0.50 0.60 0.70 ] [ 0.80 0.90 1.00 1.10 1.20 1.30 1.
Row 1: [ 1.00 1.10 1.20 1.30 1.40 1.50 1.60 1.70 ] [ 1.80 1.90 2.00 2.10 2.20 2.30 2.
Row 2: [ 2.00 2.10 2.20 2.30 2.40 2.50 2.60 2.70 ] [ 2.80 2.90 3.00 3.10 3.20 3.30 3.
Row 3: [ 3.00 3.10 3.20 3.30 3.40 3.50 3.60 3.70 ] [ 3.80 3.90 4.00 4.10 4.20 4.30 4.
Row 4: [ 4.00 4.10 4.20 4.30 4.40 4.50 4.60 4.70 ] [ 4.80 4.90 5.00 5.10 5.20 5.30 5.
Row 5: [ 5.00 5.10 5.20 5.30 5.40 5.50 5.60 5.70 ] [ 5.80 5.90 6.00 6.10 6.20 6.30 6.
Row 6: [ 6.00 6.10 6.20 6.30 6.40 6.50 6.60 6.70 ] [ 6.80 6.90 7.00 7.10 7.20 7.30 7.
Row 7: [ 7.00 7.10 7.20 7.30 7.40 7.50 7.60 7.70 ] [ 7.80 7.90 8.00 8.10 8.20 8.30 8.
Computing ground truth...
Running conv...
Loading row starting with 0.000000 into input register 0
Loading row starting with 1.000000 into input register 1
Loading row starting with 2.000000 into input register 2
Loading row starting with 3.000000 into input register 3
Loading row starting with 4.000000 into input register 4
Loading row starting with 5.000000 into input register 5
Loading row starting with 6.000000 into input register 6
Loading row starting with 7.000000 into input register 7
Computing 8 output rows, from 1 input channel blocks (2 input channels)
[DEBUG] output row 0
  [DEBUG] filter row 1
    [DEBUG] channel 0, filter_row 1, out_row 0
      [Filter column 2]
```

```
    Current filter input: [12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0], [20.0,
    Current input vector: [0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80], [0.90,
    [Filter column 1]
    Current filter input: [11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0], [12.0,
    Current input vector: [0.00, 0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70], [0.80,
    [Filter column 0]
    Current filter input: [10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0], [11.0,
    Current input vector: [1.50, 0.00, 0.10, 0.20, 0.30, 0.40, 0.50, 0.60], [0.70,
  [DEBUG] channel 1, filter_row 1, out_row 0
    [Filter column 2]
    Current filter input: [24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0], [40.0,
    Current input vector: [0.90, 1.00, 1.10, 1.20, 1.30, 1.40, 1.50, 0.00], [0.10,
    [Filter column 1]
    Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [24.0,
    Current input vector: [0.80, 0.90, 1.00, 1.10, 1.20, 1.30, 1.40, 1.50], [0.00,
    [Filter column 0]
    Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [22.0,
    Current input vector: [0.70, 0.80, 0.90, 1.00, 1.10, 1.20, 1.30, 1.40], [1.50,
[DEBUG] filter row 2
  [DEBUG] channel 0, filter_row 2, out_row 0
    [Filter column 2]
    Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [0.0, 0
    Current input vector: [1.10, 1.20, 1.30, 1.40, 1.50, 1.60, 1.70, 1.80], [1.90,
    [Filter column 1]
    Current filter input: [21.0, 21.0, 21.0, 21.0, 21.0, 21.0, 21.0, 21.0], [22.0,
    Current input vector: [1.00, 1.10, 1.20, 1.30, 1.40, 1.50, 1.60, 1.70], [1.80,
    [Filter column 0]
    Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [21.0,
    Current input vector: [2.50, 1.00, 1.10, 1.20, 1.30, 1.40, 1.50, 1.60], [1.70,
  [DEBUG] channel 1, filter_row 2, out_row 0
    [Filter column 2]
    Current filter input: [44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0], [0.0, 0
    Current input vector: [1.90, 2.00, 2.10, 2.20, 2.30, 2.40, 2.50, 1.00], [1.10,
```

```
      [Filter column 1]
      Current filter input: [42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0], [44.0,
      Current input vector: [1.80, 1.90, 2.00, 2.10, 2.20, 2.30, 2.40, 2.50], [1.00,
      [Filter column 0]
      Current filter input: [40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0], [42.0,
      Current input vector: [1.70, 1.80, 1.90, 2.00, 2.10, 2.20, 2.30, 2.40], [2.50,
completed output row 0, first element stored: 244.799988
[DEBUG] output row 1
  [DEBUG] filter row 0
    [DEBUG] channel 0, filter_row 0, out_row 1
      [Filter column 2]
      Current filter input: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0], [10.0, 10.0, 10
      Current input vector: [0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80], [0.90,
      [Filter column 1]
      Current filter input: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0], [2.0, 2.0, 2.0,
      Current input vector: [0.00, 0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70], [0.80,
      [Filter column 0]
      Current filter input: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [1.0, 1.0, 1.0,
      Current input vector: [1.50, 0.00, 0.10, 0.20, 0.30, 0.40, 0.50, 0.60], [0.70,
    [DEBUG] channel 1, filter_row 0, out_row 1
      [Filter column 2]
      Current filter input: [4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0], [20.0, 20.0, 20
      Current input vector: [0.90, 1.00, 1.10, 1.20, 1.30, 1.40, 1.50, 0.00], [0.10,
      [Filter column 1]
      Current filter input: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0], [4.0, 4.0, 4.0,
      Current input vector: [0.80, 0.90, 1.00, 1.10, 1.20, 1.30, 1.40, 1.50], [0.00,
      [Filter column 0]
      Current filter input: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [2.0, 2.0, 2.0,
      Current input vector: [0.70, 0.80, 0.90, 1.00, 1.10, 1.20, 1.30, 1.40], [1.50,
  [DEBUG] filter row 1
    [DEBUG] channel 0, filter_row 1, out_row 1
      [Filter column 2]
      Current filter input: [12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0], [20.0,
```

```
    Current input vector: [1.10, 1.20, 1.30, 1.40, 1.50, 1.60, 1.70, 1.80], [1.90,
    [Filter column 1]
    Current filter input: [11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0], [12.0,
    Current input vector: [1.00, 1.10, 1.20, 1.30, 1.40, 1.50, 1.60, 1.70], [1.80,
    [Filter column 0]
    Current filter input: [10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0], [11.0,
    Current input vector: [2.50, 1.00, 1.10, 1.20, 1.30, 1.40, 1.50, 1.60], [1.70,
  [DEBUG] channel 1, filter_row 1, out_row 1
    [Filter column 2]
    Current filter input: [24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0], [40.0,
    Current input vector: [1.90, 2.00, 2.10, 2.20, 2.30, 2.40, 2.50, 1.00], [1.10,
    [Filter column 1]
    Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [24.0,
    Current input vector: [1.80, 1.90, 2.00, 2.10, 2.20, 2.30, 2.40, 2.50], [1.00,
    [Filter column 0]
    Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [22.0,
    Current input vector: [1.70, 1.80, 1.90, 2.00, 2.10, 2.20, 2.30, 2.40], [2.50,
[DEBUG] filter row 2
  [DEBUG] channel 0, filter_row 2, out_row 1
    [Filter column 2]
    Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [0.0, 0
    Current input vector: [2.10, 2.20, 2.30, 2.40, 2.50, 2.60, 2.70, 2.80], [2.90,
    [Filter column 1]
    Current filter input: [21.0, 21.0, 21.0, 21.0, 21.0, 21.0, 21.0, 21.0], [22.0,
    Current input vector: [2.00, 2.10, 2.20, 2.30, 2.40, 2.50, 2.60, 2.70], [2.80,
    [Filter column 0]
    Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [21.0,
    Current input vector: [3.50, 2.00, 2.10, 2.20, 2.30, 2.40, 2.50, 2.60], [2.70,
  [DEBUG] channel 1, filter_row 2, out_row 1
    [Filter column 2]
    Current filter input: [44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0], [0.0, 0
    Current input vector: [2.90, 3.00, 3.10, 3.20, 3.30, 3.40, 3.50, 2.00], [2.10,
    [Filter column 1]
```

```
      Current filter input: [42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0], [44.0,
      Current input vector: [2.80, 2.90, 3.00, 3.10, 3.20, 3.30, 3.40, 3.50], [2.00,
      [Filter column 0]
      Current filter input: [40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0], [42.0,
      Current input vector: [2.70, 2.80, 2.90, 3.00, 3.10, 3.20, 3.30, 3.40], [3.50,
completed output row 1, first element stored: 448.200012
[DEBUG] output row 2
  [DEBUG] filter row 0
    [DEBUG] channel 0, filter_row 0, out_row 2
      [Filter column 2]
      Current filter input: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0], [10.0, 10.0, 10
      Current input vector: [1.10, 1.20, 1.30, 1.40, 1.50, 1.60, 1.70, 1.80], [1.90,
      [Filter column 1]
      Current filter input: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0], [2.0, 2.0, 2.0,
      Current input vector: [1.00, 1.10, 1.20, 1.30, 1.40, 1.50, 1.60, 1.70], [1.80,
      [Filter column 0]
      Current filter input: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [1.0, 1.0, 1.0,
      Current input vector: [2.50, 1.00, 1.10, 1.20, 1.30, 1.40, 1.50, 1.60], [1.70,
    [DEBUG] channel 1, filter_row 0, out_row 2
      [Filter column 2]
      Current filter input: [4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0], [20.0, 20.0, 20
      Current input vector: [1.90, 2.00, 2.10, 2.20, 2.30, 2.40, 2.50, 1.00], [1.10,
      [Filter column 1]
      Current filter input: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0], [4.0, 4.0, 4.0,
      Current input vector: [1.80, 1.90, 2.00, 2.10, 2.20, 2.30, 2.40, 2.50], [1.00,
      [Filter column 0]
      Current filter input: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [2.0, 2.0, 2.0,
      Current input vector: [1.70, 1.80, 1.90, 2.00, 2.10, 2.20, 2.30, 2.40], [2.50,
  [DEBUG] filter row 1
    [DEBUG] channel 0, filter_row 1, out_row 2
      [Filter column 2]
      Current filter input: [12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0], [20.0,
      Current input vector: [2.10, 2.20, 2.30, 2.40, 2.50, 2.60, 2.70, 2.80], [2.90,
```

```
    [Filter column 1]
    Current filter input: [11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0], [12.0,
    Current input vector: [2.00, 2.10, 2.20, 2.30, 2.40, 2.50, 2.60, 2.70], [2.80,
    [Filter column 0]
    Current filter input: [10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0], [11.0,
    Current input vector: [3.50, 2.00, 2.10, 2.20, 2.30, 2.40, 2.50, 2.60], [2.70,
  [DEBUG] channel 1, filter_row 1, out_row 2
    [Filter column 2]
    Current filter input: [24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0], [40.0,
    Current input vector: [2.90, 3.00, 3.10, 3.20, 3.30, 3.40, 3.50, 2.00], [2.10,
    [Filter column 1]
    Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [24.0,
    Current input vector: [2.80, 2.90, 3.00, 3.10, 3.20, 3.30, 3.40, 3.50], [2.00,
    [Filter column 0]
    Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [22.0,
    Current input vector: [2.70, 2.80, 2.90, 3.00, 3.10, 3.20, 3.30, 3.40], [3.50,
[DEBUG] filter row 2
  [DEBUG] channel 0, filter_row 2, out_row 2
    [Filter column 2]
    Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [0.0, 0
    Current input vector: [3.10, 3.20, 3.30, 3.40, 3.50, 3.60, 3.70, 3.80], [3.90,
    [Filter column 1]
    Current filter input: [21.0, 21.0, 21.0, 21.0, 21.0, 21.0, 21.0, 21.0], [22.0,
    Current input vector: [3.00, 3.10, 3.20, 3.30, 3.40, 3.50, 3.60, 3.70], [3.80,
    [Filter column 0]
    Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [21.0,
    Current input vector: [4.50, 3.00, 3.10, 3.20, 3.30, 3.40, 3.50, 3.60], [3.70,
  [DEBUG] channel 1, filter_row 2, out_row 2
    [Filter column 2]
    Current filter input: [44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0], [0.0, 0
    Current input vector: [3.90, 4.00, 4.10, 4.20, 4.30, 4.40, 4.50, 3.00], [3.10,
    [Filter column 1]
    Current filter input: [42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0], [44.0,
```

```
      Current input vector: [3.80, 3.90, 4.00, 4.10, 4.20, 4.30, 4.40, 4.50], [3.00,
      [Filter column 0]
      Current filter input: [40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0], [42.0,
      Current input vector: [3.70, 3.80, 3.90, 4.00, 4.10, 4.20, 4.30, 4.40], [4.50,
completed output row 2, first element stored: 655.200012
[DEBUG] output row 3
  [DEBUG] filter row 0
    [DEBUG] channel 0, filter_row 0, out_row 3
      [Filter column 2]
      Current filter input: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0], [10.0, 10.0, 10
      Current input vector: [2.10, 2.20, 2.30, 2.40, 2.50, 2.60, 2.70, 2.80], [2.90,
      [Filter column 1]
      Current filter input: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0], [2.0, 2.0, 2.0,
      Current input vector: [2.00, 2.10, 2.20, 2.30, 2.40, 2.50, 2.60, 2.70], [2.80,
      [Filter column 0]
      Current filter input: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [1.0, 1.0, 1.0,
      Current input vector: [3.50, 2.00, 2.10, 2.20, 2.30, 2.40, 2.50, 2.60], [2.70,
    [DEBUG] channel 1, filter_row 0, out_row 3
      [Filter column 2]
      Current filter input: [4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0], [20.0, 20.0, 20
      Current input vector: [2.90, 3.00, 3.10, 3.20, 3.30, 3.40, 3.50, 2.00], [2.10,
      [Filter column 1]
      Current filter input: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0], [4.0, 4.0, 4.0,
      Current input vector: [2.80, 2.90, 3.00, 3.10, 3.20, 3.30, 3.40, 3.50], [2.00,
      [Filter column 0]
      Current filter input: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [2.0, 2.0, 2.0,
      Current input vector: [2.70, 2.80, 2.90, 3.00, 3.10, 3.20, 3.30, 3.40], [3.50,
  [DEBUG] filter row 1
    [DEBUG] channel 0, filter_row 1, out_row 3
      [Filter column 2]
      Current filter input: [12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0], [20.0,
      Current input vector: [3.10, 3.20, 3.30, 3.40, 3.50, 3.60, 3.70, 3.80], [3.90,
      [Filter column 1]
```

```
    Current filter input: [11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0], [12.0,
    Current input vector: [3.00, 3.10, 3.20, 3.30, 3.40, 3.50, 3.60, 3.70], [3.80,
    [Filter column 0]
    Current filter input: [10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0], [11.0,
    Current input vector: [4.50, 3.00, 3.10, 3.20, 3.30, 3.40, 3.50, 3.60], [3.70,
  [DEBUG] channel 1, filter_row 1, out_row 3
    [Filter column 2]
    Current filter input: [24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0], [40.0,
    Current input vector: [3.90, 4.00, 4.10, 4.20, 4.30, 4.40, 4.50, 3.00], [3.10,
    [Filter column 1]
    Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [24.0,
    Current input vector: [3.80, 3.90, 4.00, 4.10, 4.20, 4.30, 4.40, 4.50], [3.00,
    [Filter column 0]
    Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [22.0,
    Current input vector: [3.70, 3.80, 3.90, 4.00, 4.10, 4.20, 4.30, 4.40], [4.50,
[DEBUG] filter row 2
  [DEBUG] channel 0, filter_row 2, out_row 3
    [Filter column 2]
    Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [0.0, 0
    Current input vector: [4.10, 4.20, 4.30, 4.40, 4.50, 4.60, 4.70, 4.80], [4.90,
    [Filter column 1]
    Current filter input: [21.0, 21.0, 21.0, 21.0, 21.0, 21.0, 21.0, 21.0], [22.0,
    Current input vector: [4.00, 4.10, 4.20, 4.30, 4.40, 4.50, 4.60, 4.70], [4.80,
    [Filter column 0]
    Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [21.0,
    Current input vector: [5.50, 4.00, 4.10, 4.20, 4.30, 4.40, 4.50, 4.60], [4.70,
  [DEBUG] channel 1, filter_row 2, out_row 3
    [Filter column 2]
    Current filter input: [44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0], [0.0, 0
    Current input vector: [4.90, 5.00, 5.10, 5.20, 5.30, 5.40, 5.50, 4.00], [4.10,
    [Filter column 1]
    Current filter input: [42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0], [44.0,
    Current input vector: [4.80, 4.90, 5.00, 5.10, 5.20, 5.30, 5.40, 5.50], [4.00,
```

```
      [Filter column 0]
      Current filter input: [40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0], [42.0,
      Current input vector: [4.70, 4.80, 4.90, 5.00, 5.10, 5.20, 5.30, 5.40], [5.50,
completed output row 3, first element stored: 862.200012
[DEBUG] output row 4
  [DEBUG] filter row 0
    [DEBUG] channel 0, filter_row 0, out_row 4
      [Filter column 2]
      Current filter input: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0], [10.0, 10.0, 10
      Current input vector: [3.10, 3.20, 3.30, 3.40, 3.50, 3.60, 3.70, 3.80], [3.90,
      [Filter column 1]
      Current filter input: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0], [2.0, 2.0, 2.0,
      Current input vector: [3.00, 3.10, 3.20, 3.30, 3.40, 3.50, 3.60, 3.70], [3.80,
      [Filter column 0]
      Current filter input: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [1.0, 1.0, 1.0,
      Current input vector: [4.50, 3.00, 3.10, 3.20, 3.30, 3.40, 3.50, 3.60], [3.70,
    [DEBUG] channel 1, filter_row 0, out_row 4
      [Filter column 2]
      Current filter input: [4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0], [20.0, 20.0, 20
      Current input vector: [3.90, 4.00, 4.10, 4.20, 4.30, 4.40, 4.50, 3.00], [3.10,
      [Filter column 1]
      Current filter input: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0], [4.0, 4.0, 4.0,
      Current input vector: [3.80, 3.90, 4.00, 4.10, 4.20, 4.30, 4.40, 4.50], [3.00,
      [Filter column 0]
      Current filter input: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [2.0, 2.0, 2.0,
      Current input vector: [3.70, 3.80, 3.90, 4.00, 4.10, 4.20, 4.30, 4.40], [4.50,
  [DEBUG] filter row 1
    [DEBUG] channel 0, filter_row 1, out_row 4
      [Filter column 2]
      Current filter input: [12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0], [20.0,
      Current input vector: [4.10, 4.20, 4.30, 4.40, 4.50, 4.60, 4.70, 4.80], [4.90,
      [Filter column 1]
      Current filter input: [11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0], [12.0,
```

```
    Current input vector: [4.00, 4.10, 4.20, 4.30, 4.40, 4.50, 4.60, 4.70], [4.80,
    [Filter column 0]
    Current filter input: [10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0], [11.0,
    Current input vector: [5.50, 4.00, 4.10, 4.20, 4.30, 4.40, 4.50, 4.60], [4.70,
  [DEBUG] channel 1, filter_row 1, out_row 4
    [Filter column 2]
    Current filter input: [24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0], [40.0,
    Current input vector: [4.90, 5.00, 5.10, 5.20, 5.30, 5.40, 5.50, 4.00], [4.10,
    [Filter column 1]
    Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [24.0,
    Current input vector: [4.80, 4.90, 5.00, 5.10, 5.20, 5.30, 5.40, 5.50], [4.00,
    [Filter column 0]
    Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [22.0,
    Current input vector: [4.70, 4.80, 4.90, 5.00, 5.10, 5.20, 5.30, 5.40], [5.50,
[DEBUG] filter row 2
  [DEBUG] channel 0, filter_row 2, out_row 4
    [Filter column 2]
    Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [0.0, 0
    Current input vector: [5.10, 5.20, 5.30, 5.40, 5.50, 5.60, 5.70, 5.80], [5.90,
    [Filter column 1]
    Current filter input: [21.0, 21.0, 21.0, 21.0, 21.0, 21.0, 21.0, 21.0], [22.0,
    Current input vector: [5.00, 5.10, 5.20, 5.30, 5.40, 5.50, 5.60, 5.70], [5.80,
    [Filter column 0]
    Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [21.0,
    Current input vector: [6.50, 5.00, 5.10, 5.20, 5.30, 5.40, 5.50, 5.60], [5.70,
  [DEBUG] channel 1, filter_row 2, out_row 4
    [Filter column 2]
    Current filter input: [44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0], [0.0, 0
    Current input vector: [5.90, 6.00, 6.10, 6.20, 6.30, 6.40, 6.50, 5.00], [5.10,
    [Filter column 1]
    Current filter input: [42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0], [44.0,
    Current input vector: [5.80, 5.90, 6.00, 6.10, 6.20, 6.30, 6.40, 6.50], [5.00,
    [Filter column 0]
```

```
      Current filter input: [40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0], [42.0,
      Current input vector: [5.70, 5.80, 5.90, 6.00, 6.10, 6.20, 6.30, 6.40], [6.50,
completed output row 4, first element stored: 1069.199951
[DEBUG] output row 5
  [DEBUG] filter row 0
    [DEBUG] channel 0, filter_row 0, out_row 5
      [Filter column 2]
      Current filter input: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0], [10.0, 10.0, 10
      Current input vector: [4.10, 4.20, 4.30, 4.40, 4.50, 4.60, 4.70, 4.80], [4.90,
      [Filter column 1]
      Current filter input: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0], [2.0, 2.0, 2.0,
      Current input vector: [4.00, 4.10, 4.20, 4.30, 4.40, 4.50, 4.60, 4.70], [4.80,
      [Filter column 0]
      Current filter input: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [1.0, 1.0, 1.0,
      Current input vector: [5.50, 4.00, 4.10, 4.20, 4.30, 4.40, 4.50, 4.60], [4.70,
    [DEBUG] channel 1, filter_row 0, out_row 5
      [Filter column 2]
      Current filter input: [4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0], [20.0, 20.0, 20
      Current input vector: [4.90, 5.00, 5.10, 5.20, 5.30, 5.40, 5.50, 4.00], [4.10,
      [Filter column 1]
      Current filter input: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0], [4.0, 4.0, 4.0,
      Current input vector: [4.80, 4.90, 5.00, 5.10, 5.20, 5.30, 5.40, 5.50], [4.00,
      [Filter column 0]
      Current filter input: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [2.0, 2.0, 2.0,
      Current input vector: [4.70, 4.80, 4.90, 5.00, 5.10, 5.20, 5.30, 5.40], [5.50,
  [DEBUG] filter row 1
    [DEBUG] channel 0, filter_row 1, out_row 5
      [Filter column 2]
      Current filter input: [12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0], [20.0,
      Current input vector: [5.10, 5.20, 5.30, 5.40, 5.50, 5.60, 5.70, 5.80], [5.90,
      [Filter column 1]
      Current filter input: [11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0], [12.0,
      Current input vector: [5.00, 5.10, 5.20, 5.30, 5.40, 5.50, 5.60, 5.70], [5.80,
```

```
    [Filter column 0]
    Current filter input: [10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0], [11.0,
    Current input vector: [6.50, 5.00, 5.10, 5.20, 5.30, 5.40, 5.50, 5.60], [5.70,
  [DEBUG] channel 1, filter_row 1, out_row 5
    [Filter column 2]
    Current filter input: [24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0], [40.0,
    Current input vector: [5.90, 6.00, 6.10, 6.20, 6.30, 6.40, 6.50, 5.00], [5.10,
    [Filter column 1]
    Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [24.0,
    Current input vector: [5.80, 5.90, 6.00, 6.10, 6.20, 6.30, 6.40, 6.50], [5.00,
    [Filter column 0]
    Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [22.0,
    Current input vector: [5.70, 5.80, 5.90, 6.00, 6.10, 6.20, 6.30, 6.40], [6.50,
[DEBUG] filter row 2
  [DEBUG] channel 0, filter_row 2, out_row 5
    [Filter column 2]
    Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [0.0, 0
    Current input vector: [6.10, 6.20, 6.30, 6.40, 6.50, 6.60, 6.70, 6.80], [6.90,
    [Filter column 1]
    Current filter input: [21.0, 21.0, 21.0, 21.0, 21.0, 21.0, 21.0, 21.0], [22.0,
    Current input vector: [6.00, 6.10, 6.20, 6.30, 6.40, 6.50, 6.60, 6.70], [6.80,
    [Filter column 0]
    Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [21.0,
    Current input vector: [7.50, 6.00, 6.10, 6.20, 6.30, 6.40, 6.50, 6.60], [6.70,
  [DEBUG] channel 1, filter_row 2, out_row 5
    [Filter column 2]
    Current filter input: [44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0], [0.0, 0
    Current input vector: [6.90, 7.00, 7.10, 7.20, 7.30, 7.40, 7.50, 6.00], [6.10,
    [Filter column 1]
    Current filter input: [42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0], [44.0,
    Current input vector: [6.80, 6.90, 7.00, 7.10, 7.20, 7.30, 7.40, 7.50], [6.00,
    [Filter column 0]
    Current filter input: [40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0], [42.0,
```

```
      Current input vector: [6.70, 6.80, 6.90, 7.00, 7.10, 7.20, 7.30, 7.40], [7.50,
completed output row 5, first element stored: 1276.199951
[DEBUG] output row 6
  [DEBUG] filter row 0
    [DEBUG] channel 0, filter_row 0, out_row 6
      [Filter column 2]
      Current filter input: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0], [10.0, 10.0, 10
      Current input vector: [5.10, 5.20, 5.30, 5.40, 5.50, 5.60, 5.70, 5.80], [5.90,
      [Filter column 1]
      Current filter input: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0], [2.0, 2.0, 2.0,
      Current input vector: [5.00, 5.10, 5.20, 5.30, 5.40, 5.50, 5.60, 5.70], [5.80,
      [Filter column 0]
      Current filter input: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [1.0, 1.0, 1.0,
      Current input vector: [6.50, 5.00, 5.10, 5.20, 5.30, 5.40, 5.50, 5.60], [5.70,
    [DEBUG] channel 1, filter_row 0, out_row 6
      [Filter column 2]
      Current filter input: [4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0], [20.0, 20.0, 20
      Current input vector: [5.90, 6.00, 6.10, 6.20, 6.30, 6.40, 6.50, 5.00], [5.10,
      [Filter column 1]
      Current filter input: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0], [4.0, 4.0, 4.0,
      Current input vector: [5.80, 5.90, 6.00, 6.10, 6.20, 6.30, 6.40, 6.50], [5.00,
      [Filter column 0]
      Current filter input: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [2.0, 2.0, 2.0,
      Current input vector: [5.70, 5.80, 5.90, 6.00, 6.10, 6.20, 6.30, 6.40], [6.50,
  [DEBUG] filter row 1
    [DEBUG] channel 0, filter_row 1, out_row 6
      [Filter column 2]
      Current filter input: [12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0], [20.0,
      Current input vector: [6.10, 6.20, 6.30, 6.40, 6.50, 6.60, 6.70, 6.80], [6.90,
      [Filter column 1]
      Current filter input: [11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0], [12.0,
      Current input vector: [6.00, 6.10, 6.20, 6.30, 6.40, 6.50, 6.60, 6.70], [6.80,
      [Filter column 0]
```

```
    Current filter input: [10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0], [11.0,
    Current input vector: [7.50, 6.00, 6.10, 6.20, 6.30, 6.40, 6.50, 6.60], [6.70,
  [DEBUG] channel 1, filter_row 1, out_row 6
    [Filter column 2]
    Current filter input: [24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0], [40.0,
    Current input vector: [6.90, 7.00, 7.10, 7.20, 7.30, 7.40, 7.50, 6.00], [6.10,
    [Filter column 1]
    Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [24.0,
    Current input vector: [6.80, 6.90, 7.00, 7.10, 7.20, 7.30, 7.40, 7.50], [6.00,
    [Filter column 0]
    Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [22.0,
    Current input vector: [6.70, 6.80, 6.90, 7.00, 7.10, 7.20, 7.30, 7.40], [7.50,
[DEBUG] filter row 2
  [DEBUG] channel 0, filter_row 2, out_row 6
    [Filter column 2]
    Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [0.0, 0
    Current input vector: [7.10, 7.20, 7.30, 7.40, 7.50, 7.60, 7.70, 7.80], [7.90,
    [Filter column 1]
    Current filter input: [21.0, 21.0, 21.0, 21.0, 21.0, 21.0, 21.0, 21.0], [22.0,
    Current input vector: [7.00, 7.10, 7.20, 7.30, 7.40, 7.50, 7.60, 7.70], [7.80,
    [Filter column 0]
    Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [21.0,
    Current input vector: [8.50, 7.00, 7.10, 7.20, 7.30, 7.40, 7.50, 7.60], [7.70,
  [DEBUG] channel 1, filter_row 2, out_row 6
    [Filter column 2]
    Current filter input: [44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0], [0.0, 0
    Current input vector: [7.90, 8.00, 8.10, 8.20, 8.30, 8.40, 8.50, 7.00], [7.10,
    [Filter column 1]
    Current filter input: [42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0], [44.0,
    Current input vector: [7.80, 7.90, 8.00, 8.10, 8.20, 8.30, 8.40, 8.50], [7.00,
    [Filter column 0]
    Current filter input: [40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0], [42.0,
    Current input vector: [7.70, 7.80, 7.90, 8.00, 8.10, 8.20, 8.30, 8.40], [8.50,
```

```
completed output row 6, first element stored: 1483.199951
[DEBUG] output row 7
  [DEBUG] filter row 0
    [DEBUG] channel 0, filter_row 0, out_row 7
      [Filter column 2]
      Current filter input: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0], [10.0, 10.0, 10
      Current input vector: [6.10, 6.20, 6.30, 6.40, 6.50, 6.60, 6.70, 6.80], [6.90,
      [Filter column 1]
      Current filter input: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0], [2.0, 2.0, 2.0,
      Current input vector: [6.00, 6.10, 6.20, 6.30, 6.40, 6.50, 6.60, 6.70], [6.80,
      [Filter column 0]
      Current filter input: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [1.0, 1.0, 1.0,
      Current input vector: [7.50, 6.00, 6.10, 6.20, 6.30, 6.40, 6.50, 6.60], [6.70,
    [DEBUG] channel 1, filter_row 0, out_row 7
      [Filter column 2]
      Current filter input: [4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0], [20.0, 20.0, 20
      Current input vector: [6.90, 7.00, 7.10, 7.20, 7.30, 7.40, 7.50, 6.00], [6.10,
      [Filter column 1]
      Current filter input: [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0], [4.0, 4.0, 4.0,
      Current input vector: [6.80, 6.90, 7.00, 7.10, 7.20, 7.30, 7.40, 7.50], [6.00,
      [Filter column 0]
      Current filter input: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [2.0, 2.0, 2.0,
      Current input vector: [6.70, 6.80, 6.90, 7.00, 7.10, 7.20, 7.30, 7.40], [7.50,
  [DEBUG] filter row 1
    [DEBUG] channel 0, filter_row 1, out_row 7
      [Filter column 2]
      Current filter input: [12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0, 12.0], [20.0,
      Current input vector: [7.10, 7.20, 7.30, 7.40, 7.50, 7.60, 7.70, 7.80], [7.90,
      [Filter column 1]
      Current filter input: [11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0], [12.0,
      Current input vector: [7.00, 7.10, 7.20, 7.30, 7.40, 7.50, 7.60, 7.70], [7.80,
      [Filter column 0]
      Current filter input: [10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0], [11.0,
```

```
      Current input vector: [8.50, 7.00, 7.10, 7.20, 7.30, 7.40, 7.50, 7.60], [7.70,
    [DEBUG] channel 1, filter_row 1, out_row 7
      [Filter column 2]
      Current filter input: [24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0, 24.0], [40.0,
      Current input vector: [7.90, 8.00, 8.10, 8.20, 8.30, 8.40, 8.50, 7.00], [7.10,
      [Filter column 1]
      Current filter input: [22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0, 22.0], [24.0,
      Current input vector: [7.80, 7.90, 8.00, 8.10, 8.20, 8.30, 8.40, 8.50], [7.00,
      [Filter column 0]
      Current filter input: [20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0, 20.0], [22.0,
      Current input vector: [7.70, 7.80, 7.90, 8.00, 8.10, 8.20, 8.30, 8.40], [8.50,
completed output row 7, first element stored: 582.799988
Checking the results...
output[0] = 244.8000   ~= 244.8000 (Correct) (Output channel 0, row 0)
output[1] = 372.6000   ~= 372.6000 (Correct) (Output channel 0, row 0)
output[2] = 401.4000   ~= 401.4000 (Correct) (Output channel 0, row 0)
output[3] = 430.2000   ~= 430.2000 (Correct) (Output channel 0, row 0)
output[4] = 459.0000   ~= 459.0000 (Correct) (Output channel 0, row 0)
output[5] = 487.8000   ~= 487.8000 (Correct) (Output channel 0, row 0)
output[6] = 516.6000   ~= 516.6000 (Correct) (Output channel 0, row 0)
output[7] = 343.4000   ~= 343.4000 (Correct) (Output channel 0, row 0)
output[8] = 115.2000   ~= 115.2000 (Correct) (Output channel 1, row 0)
output[9] = 220.0000   ~= 220.0000 (Correct) (Output channel 1, row 0)
output[10] = 245.8000   ~= 245.8000 (Correct) (Output channel 1, row 0)
output[11] = 271.6000   ~= 271.6000 (Correct) (Output channel 1, row 0)
output[12] = 297.4000   ~= 297.4000 (Correct) (Output channel 1, row 0)
output[13] = 323.2000   ~= 323.2000 (Correct) (Output channel 1, row 0)
output[14] = 349.0000   ~= 349.0000 (Correct) (Output channel 1, row 0)
output[15] = 310.8000   ~= 310.8000 (Correct) (Output channel 1, row 0)
output[16] = 448.2000   ~= 448.2000 (Correct) (Output channel 0, row 1)
output[17] = 666.9000   ~= 666.9000 (Correct) (Output channel 0, row 1)
output[18] = 696.6000   ~= 696.6000 (Correct) (Output channel 0, row 1)
output[19] = 726.3000   ~= 726.3000 (Correct) (Output channel 0, row 1)
```

```
output[20] = 756.0000   ~= 756.0001 (Correct) (Output channel 0, row 1)
output[21] = 785.7000   ~= 785.7000 (Correct) (Output channel 0, row 1)
output[22] = 815.4000   ~= 815.4000 (Correct) (Output channel 0, row 1)
output[23] = 533.1000   ~= 533.1000 (Correct) (Output channel 0, row 1)
output[24] = 289.8000   ~= 289.8000 (Correct) (Output channel 1, row 1)
output[25] = 495.0000   ~= 495.0000 (Correct) (Output channel 1, row 1)
output[26] = 524.7000   ~= 524.7000 (Correct) (Output channel 1, row 1)
output[27] = 554.4000   ~= 554.4000 (Correct) (Output channel 1, row 1)
output[28] = 584.1000   ~= 584.1000 (Correct) (Output channel 1, row 1)
output[29] = 613.8000   ~= 613.8000 (Correct) (Output channel 1, row 1)
output[30] = 643.5000   ~= 643.5000 (Correct) (Output channel 1, row 1)
output[31] = 517.2000   ~= 517.2000 (Correct) (Output channel 1, row 1)
output[32] = 655.2000   ~= 655.2000 (Correct) (Output channel 0, row 2)
output[33] = 963.9000   ~= 963.9000 (Correct) (Output channel 0, row 2)
output[34] = 993.6000   ~= 993.6000 (Correct) (Output channel 0, row 2)
output[35] = 1023.3000   ~= 1023.3000 (Correct) (Output channel 0, row 2)
output[36] = 1052.9999   ~= 1053.0000 (Correct) (Output channel 0, row 2)
output[37] = 1082.7001   ~= 1082.7000 (Correct) (Output channel 0, row 2)
output[38] = 1112.4000   ~= 1112.4000 (Correct) (Output channel 0, row 2)
output[39] = 722.1000   ~= 722.1000 (Correct) (Output channel 0, row 2)
output[40] = 487.8000   ~= 487.8000 (Correct) (Output channel 1, row 2)
output[41] = 792.0000   ~= 792.0000 (Correct) (Output channel 1, row 2)
output[42] = 821.7000   ~= 821.7000 (Correct) (Output channel 1, row 2)
output[43] = 851.4001   ~= 851.4000 (Correct) (Output channel 1, row 2)
output[44] = 881.1000   ~= 881.1000 (Correct) (Output channel 1, row 2)
output[45] = 910.8000   ~= 910.8000 (Correct) (Output channel 1, row 2)
output[46] = 940.5000   ~= 940.5000 (Correct) (Output channel 1, row 2)
output[47] = 724.2000   ~= 724.2000 (Correct) (Output channel 1, row 2)
output[48] = 862.2000   ~= 862.2000 (Correct) (Output channel 0, row 3)
output[49] = 1260.9000   ~= 1260.9000 (Correct) (Output channel 0, row 3)
output[50] = 1290.6001   ~= 1290.6001 (Correct) (Output channel 0, row 3)
output[51] = 1320.2999   ~= 1320.3000 (Correct) (Output channel 0, row 3)
output[52] = 1350.0000   ~= 1350.0000 (Correct) (Output channel 0, row 3)
```

```
output[53] = 1379.7000   ~= 1379.7000 (Correct) (Output channel 0, row 3)
output[54] = 1409.4000   ~= 1409.4000 (Correct) (Output channel 0, row 3)
output[55] = 911.1000   ~= 911.1000 (Correct) (Output channel 0, row 3)
output[56] = 685.8000   ~= 685.8000 (Correct) (Output channel 1, row 3)
output[57] = 1089.0000   ~= 1089.0000 (Correct) (Output channel 1, row 3)
output[58] = 1118.7000   ~= 1118.7000 (Correct) (Output channel 1, row 3)
output[59] = 1148.4000   ~= 1148.4000 (Correct) (Output channel 1, row 3)
output[60] = 1178.1000   ~= 1178.1001 (Correct) (Output channel 1, row 3)
output[61] = 1207.8000   ~= 1207.8000 (Correct) (Output channel 1, row 3)
output[62] = 1237.5000   ~= 1237.5000 (Correct) (Output channel 1, row 3)
output[63] = 931.2000   ~= 931.2000 (Correct) (Output channel 1, row 3)
output[64] = 1069.2000   ~= 1069.2000 (Correct) (Output channel 0, row 4)
output[65] = 1557.9000   ~= 1557.9000 (Correct) (Output channel 0, row 4)
output[66] = 1587.6000   ~= 1587.6001 (Correct) (Output channel 0, row 4)
output[67] = 1617.2999   ~= 1617.2999 (Correct) (Output channel 0, row 4)
output[68] = 1647.0000   ~= 1647.0000 (Correct) (Output channel 0, row 4)
output[69] = 1676.7000   ~= 1676.7000 (Correct) (Output channel 0, row 4)
output[70] = 1706.4000   ~= 1706.4000 (Correct) (Output channel 0, row 4)
output[71] = 1100.1000   ~= 1100.1001 (Correct) (Output channel 0, row 4)
output[72] = 883.8000   ~= 883.8000 (Correct) (Output channel 1, row 4)
output[73] = 1386.0000   ~= 1386.0000 (Correct) (Output channel 1, row 4)
output[74] = 1415.7000   ~= 1415.7000 (Correct) (Output channel 1, row 4)
output[75] = 1445.4000   ~= 1445.4000 (Correct) (Output channel 1, row 4)
output[76] = 1475.1000   ~= 1475.1001 (Correct) (Output channel 1, row 4)
output[77] = 1504.8000   ~= 1504.7999 (Correct) (Output channel 1, row 4)
output[78] = 1534.5000   ~= 1534.5000 (Correct) (Output channel 1, row 4)
output[79] = 1138.2001   ~= 1138.2000 (Correct) (Output channel 1, row 4)
output[80] = 1276.2000   ~= 1276.2000 (Correct) (Output channel 0, row 5)
output[81] = 1854.9000   ~= 1854.9000 (Correct) (Output channel 0, row 5)
output[82] = 1884.6000   ~= 1884.6001 (Correct) (Output channel 0, row 5)
output[83] = 1914.2999   ~= 1914.2999 (Correct) (Output channel 0, row 5)
output[84] = 1944.0000   ~= 1944.0000 (Correct) (Output channel 0, row 5)
output[85] = 1973.7000   ~= 1973.7000 (Correct) (Output channel 0, row 5)
```

```
output[86] = 2003.4000   ~= 2003.4000 (Correct) (Output channel 0, row 5)
output[87] = 1289.1000   ~= 1289.1001 (Correct) (Output channel 0, row 5)
output[88] = 1081.8000   ~= 1081.8000 (Correct) (Output channel 1, row 5)
output[89] = 1683.0000   ~= 1683.0000 (Correct) (Output channel 1, row 5)
output[90] = 1712.7001   ~= 1712.7000 (Correct) (Output channel 1, row 5)
output[91] = 1742.4000   ~= 1742.4000 (Correct) (Output channel 1, row 5)
output[92] = 1772.0999   ~= 1772.1001 (Correct) (Output channel 1, row 5)
output[93] = 1801.8000   ~= 1801.7999 (Correct) (Output channel 1, row 5)
output[94] = 1831.5001   ~= 1831.5000 (Correct) (Output channel 1, row 5)
output[95] = 1345.2000   ~= 1345.2000 (Correct) (Output channel 1, row 5)
output[96] = 1483.2000   ~= 1483.2000 (Correct) (Output channel 0, row 6)
output[97] = 2151.9001   ~= 2151.8999 (Correct) (Output channel 0, row 6)
output[98] = 2181.6001   ~= 2181.6001 (Correct) (Output channel 0, row 6)
output[99] = 2211.3003   ~= 2211.2998 (Correct) (Output channel 0, row 6)
output[100] = 2241.0000   ~= 2241.0000 (Correct) (Output channel 0, row 6)
output[101] = 2270.7000   ~= 2270.7000 (Correct) (Output channel 0, row 6)
output[102] = 2300.3999   ~= 2300.3999 (Correct) (Output channel 0, row 6)
output[103] = 1478.1000   ~= 1478.1001 (Correct) (Output channel 0, row 6)
output[104] = 1279.8000   ~= 1279.8000 (Correct) (Output channel 1, row 6)
output[105] = 1980.0001   ~= 1980.0000 (Correct) (Output channel 1, row 6)
output[106] = 2009.7000   ~= 2009.7000 (Correct) (Output channel 1, row 6)
output[107] = 2039.3999   ~= 2039.4001 (Correct) (Output channel 1, row 6)
output[108] = 2069.1001   ~= 2069.1001 (Correct) (Output channel 1, row 6)
output[109] = 2098.8000   ~= 2098.7998 (Correct) (Output channel 1, row 6)
output[110] = 2128.5000   ~= 2128.5000 (Correct) (Output channel 1, row 6)
output[111] = 1552.2000   ~= 1552.2000 (Correct) (Output channel 1, row 6)
output[112] = 582.8000   ~= 582.8000 (Correct) (Output channel 0, row 7)
output[113] = 816.6000   ~= 816.6000 (Correct) (Output channel 0, row 7)
output[114] = 827.4000   ~= 827.4000 (Correct) (Output channel 0, row 7)
output[115] = 838.2000   ~= 838.2000 (Correct) (Output channel 0, row 7)
output[116] = 849.0000   ~= 849.0000 (Correct) (Output channel 0, row 7)
output[117] = 859.8000   ~= 859.8000 (Correct) (Output channel 0, row 7)
output[118] = 870.6000   ~= 870.6000 (Correct) (Output channel 0, row 7)
```

```
output[119] = 537.4000   ~= 537.4000 (Correct) (Output channel 0, row 7)
output[120] = 932.2000   ~= 932.2000 (Correct) (Output channel 1, row 7)
output[121] = 1204.0000   ~= 1204.0000 (Correct) (Output channel 1, row 7)
output[122] = 1220.8000   ~= 1220.8000 (Correct) (Output channel 1, row 7)
output[123] = 1237.6000   ~= 1237.6000 (Correct) (Output channel 1, row 7)
output[124] = 1254.4000   ~= 1254.4000 (Correct) (Output channel 1, row 7)
output[125] = 1271.2000   ~= 1271.2000 (Correct) (Output channel 1, row 7)
output[126] = 1288.0000   ~= 1288.0000 (Correct) (Output channel 1, row 7)
output[127] = 608.8000   ~= 608.7999 (Correct) (Output channel 1, row 7)
128 out of 128 output element correct!
```