Efficient Inference Acceleration

By

Michael A. Mishkin

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy
(Electrical and Computer Engineering)

at the
UNIVERSITY OF WISCONSIN–MADISON
2019

Date of final oral examination: 5/17/2019

ACKNOWLEDGMENTS

A friend recently asked me what it is that I like about being a graduate student. The truth is that I am happy to be graduating. Completing a PhD and finally participating in what turned out to be a rather high throughput diploma ceremony has provided much of the motivation for the work contained within this dissertation. However, reaching these new heights required solid grounding for which I am ever so grateful to the people who have supported my efforts throughout this endeavor.

First I would like to thank Professor Nam Sung Kim and Professor Mikko Lipasti for their guidance and support, without which this dissertation would not have been possible. I would also like to extend my deepest gratitude to all members of the PhD Defense Committee for their critical role in this achievement.

I would like to acknowledge colleagues at Intel, who provided an internship experience that introduced me to tools and concepts that have proven valuable within my research. I would also like to thank colleagues at AMD Research for providing an internship that encouraged innovation and architectural research.

Finally, I would like to thank my parents, my grandparents, and my sister for their continued support throughout my pursuit of a PhD at the University of Wisconsin Madison.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Forward progress in computing technology is expected to involve high degrees of heterogeneity and specialization. Emerging applications integrating neural networks are becoming more common and as a result development of specialized hardware designed for acceleration of neural networks is increasingly economical. As Moore's law wanes and applications utilizing neural networks benefit from high-performance and low-power execution provided by widely available specialized hardware, algorithms using neural networks are poised to continue to outpace alternative approaches.

This dissertation explores the design space of neural network inference accelerators, spanning from monolithic systolic arrays with off-chip DRAMs for weight storage to tiled matrix-vector units with tightly coupled on-chip weight storage to supply high bandwidth weights without dependence on off-chip memory, targeting efficient microarchitectural techniques and neural network inference sequencing schemes, identifying three key design points of interest. The first is a monolithic systolic array based accelerator where pipeline depths are reduced in order to eliminate clocked element overheads. These optimizations primarily target energy-efficiency but also improve performance subject to bandwidth limitations. The accelerator includes weight permutation considerations required to better support processing convolutional layers on wide arrays using scheduling policies that preserve temporal locality of weight sub-matrices.

The second accelerator uses codebook quantization for both weights and activations to reduce power associated with both on-chip communication and synapse calculation. Codebook based quantization and dequantization are tightly integrated into the accelerator data-path enabling the bulk of on-chip communication to remain in the quantized format. Training experiments are presented to provide insight into training techniques for inference accelerators utilizing codebook quantization of both activations and weights.

The third accelerator design considers communication power reduction within a tiled accelerator using temporally coded interconnects for both activations and weights. Tolerance for the latency of the temporal codes within neural network accelerators is achieved by scheduling schemes that facilitate reuse of temporally communicated values and buffer capacities provisioned to support these schedules. Within the accelerator with temporally coded links, these adverse effects amount to performance degradations rather than high power consumption.

1 INTRODUCTION

...there is surely no reason today for taking seriously a position that attributes a complex human achievement entirely to months (or at most years) of experience, rather than to millions of years of evolution or to principles of neural organization that may be even more deeply grounded in physical law...

— Noam Chomsky

1.1 A Brief History of Intelligent Computation

The notion of intelligent machinery [135] was well established long before the inception of Moore's law [103, 62]. Yet even as many tasks far exceeding innate human ability at reasonable time and energy expense are delegated to digital systems, computers remain mostly incapable of other tasks taken for granted within biological systems. As the capability gap narrows, we find digital systems integrated across industries for progressively more sophisticated applications. Just as personal computers have provided a more intelligent means of expression than typewriters and as today's smartphones contain far more compute power than is necessary to make a phone call, we find computing seamlessly integrated into the world around us, and we are dependent upon it for a growing set of trivial tasks. Recent advances in machine learning and more specifically neural networks are among the foremost achievements narrowing the capability gap and augmenting the digital backplane supporting our high-tech lifestyles. The emergence of neural networks as an algorithm of choice is facilitated and strengthened by the emergence of hardware capable of efficiently processing them.

For the better part of a century, computer scientists, neuroscientists, and electrical engineers alike have studied the computational paradigms of the neural circuits that are the basis of intelligence [105, 125, 23, 97]. Early descriptions of neural computation can be traced to the Pitts-McCulloch Neurons [96] which modeled networks of high fan-in automata that communicated digitally using 1's and 0's, where a 1 represents a spike of electric potential, similar to the electrical properties of biological neurons that were observed by Hodgkin and Huxley [58]. The descriptions of programmable computing machinery introduced by Turing and von Neumann within the same time frame involved a computational paradigm quite different from the neuron models of Pitts and McCulloch. Although the Pitts-McCulloch neuron is readily expressed as a Turing machine, the converse is not as easily achieved [111]. It is the versatility of the von Neumann model of computation that is responsible for its continued applicability in describing general purpose compute platforms as we know them today, and it is the physical realizations of these machines upon which the bulk of neural network research has been conducted.

Research into neural networks for machine learning purposes has provided a rich set of algorithms that can be trained to accomplish a wide range of tasks [44]. These algorithms are gaining accelerating widespread interest as semiconductor technologies, following a von Neumann paradigm, have grown progressively more powerful. The great strides in compute power provided by high bandwidth general purpose GPU computing in particular have summoned the rise of Deep Learning by enabling high performance training of large neural networks with data sets far exceeding the complexity that can be easily captured by deterministic representations [115, 11]. Even as neural network based machine learning methods experience more widespread use, the computational substrates facilitating artificial intelligence remain vastly different from the densely interconnected electrical signaling networks of neurons found in biological systems. The reason for the discrepancy is firmly rooted in fundamental limitations of semiconductor based technologies, however many opportunities for more efficient silicon lie within adoption of principles embodied by neural models of computation.

The trends observed by Gordon Moore regarding transistor feature size reduction provided highly predictable silicon scaling, both accurately modeling and setting the pace for these trends in the semiconductor industry for decades, with profound impacts observed in all dependent technologies. Robert Dennard's CMOS parameter scaling guidelines [27] fueled adherence to Moore's law for many years but became infeasible as feature sizes decreased toward near atomic dimensions [102] where a minimum supply voltage only slightly greater than the transistor threshold voltage prevents continued application of Dennard's model. Transistor scaling itself continued after Dennard scaling became infeasible, however as power and area efficiency were no longer easily achieved at the device level, proper utilization of available transistors to address these concerns would require careful consideration at the architecture and system level [30]. Moore's law itself will soon come to an end as well [73, 81], so the modes of continued progress within a thriving computing industry must adapt through techniques sometimes speculatively developed within academic research.

Among the techniques providing a competitive edge in emerging computing systems is the integration of domain specific accelerators which provide higher performance and better energy efficiency than general purpose hardware. The CPU's primary purpose requires it to be generally programmable with semantics of operation separated by several layers of abstraction from the algorithms they are used to implement. Transistor scaling provided consistent CPU performance boosts realizable across all applications as the abundance of transistors available to architects enabled integration of micro-architectural features making CPU's better and better at intelligently churning through streams of instructions, employing techniques such as branch prediction, speculative prefetching, and out-of-order execution [56] to dynamically adapt to the characteristics of workloads drawn from a diverse range of algorithmic domains. The ubiquitous, highly programmable CPU core remains an essential component for management of modern computer systems within an ecosystem augmented by alternative computational hardware, such as GPUs, FPGAs, and ASICs, which are capable of more efficiently processing applications within certain workload classes. Such heterogeneous ecosystems are increasingly prevalent as the end of Moore's law approaches and domain specific acceleration provides recourse [73] as the best path for sustained progress in an industry that continues to push the cutting edge of computing technology.

One symptom of the waning of Moore's law is an increasing difficulty meeting power budgets that can require guarantees of some amount of transistor inactivity. This transistor inactivity may be achieved in several ways ranging from integration of more efficient logic to reduced utilization of inefficient logic [139]. Using lower clock frequencies is an effective power saving technique that results in a linear reduction of dynamic power and can be coupled with voltage scaling for further power reduction, however there is a limit to how low the voltage can be set without incurring exponential delay increases and unreliable operation [29]. The clock frequency plateau observed within the microprocessor market can be viewed as a manifestation of power wall mitigation that was required once Dennard's scaling principles ceased to hold. Achieving high performance with processors running at lower clock frequencies required scaling out to multiple cores. As the power wall has increased in significance with advancing process generations, the notion of dark silicon has emerged as both a limiting factor preventing further integration of additional cores [30] and a design principle which influences new processor designs [133]. Ultimately, making optimal use of transistors requires modern processor designs to achieve energy efficiency under strict power budgets. This may be simply resolved through larger cache to core ratios or through more inventive techniques involving more efficient computational logic design. The computational paradigm of the human brain is extremely dark [132], so it is reasonable to expect that looking toward the brain can provide insights into techniques for achieving efficient computation under dark silicon constraints. However, as the architecture of the brain is highly specialized, these insights may only be applicable within the context of domain specific accelerators, if at all.

In the mobile device space architects are faced with extremely strict power envelopes, due to a lack of server-scale cooling mechanisms within mobile devices and the limited battery capacities of portable devices. Low-power architectural design principles are necessary to satisfy these restrictive constraints while delivering the high performance applications that users have come to expect. One approach used within these systems is a big-little paradigm, in which large high perform out-of-order cores are coupled with smaller more efficient in-order cores, with a common ISA [78] where task to core mapping can be based on QoS requirements and predictive program phase analysis [94], using smaller cores to run less critical tasks at lower power. ASICs can provide better energy efficiency than the small cores and therefore achieve higher performance as well as saving power but sacrifice general programmability. The emergence of integrated multi-accelerator systems within a single chip, commonly known as SoCs, is now prevalent among mobile processors. Within mobile SoCs, the die area is divided among a heterogeneous set of diverse accelerator IPs, each optimized for performing a task with high efficiency, and the number of IP's per chip is increasing every year [57].

Within data centers, the form factors are more forgiving and much higher TDP's can be sustained with ample cooling capability and power budgets bound by infrastructure capacity. Over-provisioned data center power infrastructure often accounts for a substantial portion of capital costs [110], where over-provisioning is required to support peak utilization, upgradability, and fault-tolerance. Several techniques exist for power management [110, 92], which can reduce infrastructure requirements, however reducing power requirements can also be accomplished by using more energy-efficient hardware [98]. These settings are increasingly heterogeneous, with more specialized hardware like GPU's being used to achieve higher performance on workloads that run less efficiently on CPU's. In fact, recent systems leading the top500 list are gaining more FLOPS from GPU's than from CPU's [32].

The evolution of GPUs is primarily driven by the high performance demands of the computationally intensive and highly parallel real-time 3D rendering tasks of computer gaming. But the GPU execution pipelines have undergone transformation from highly specialized graphics accelerators to large arrays of general purpose cores, capable of efficiently executing thousands of threads in parallel under an execution paradigm that has come to be known as single instruction multiple thread (SIMT) [86]. This transformation began as a means of improving programmability to support the computational demands of different types of graphics shaders with the same compute units but has also been adopted for a growing set of non-graphics tasks, also amenable to a bulk synchronous parallel execution model. One of the more noteworthy examples of applications for which GPGPU has become the preferred hardware platform is found in deep learning, where high performance neural network training provided by GPGPU computing is arguably responsible for the rising tide of artificial intelligence.

Recent advances in machine learning have spawned a surge of interest in artificial intelligence as the feasibility of utilizing compute, memory, and bandwidth intensive neural network algorithms has been bolstered by both an abundance of data and the compute power necessary to support large scale training. Deep Neural Networks (DNNs) in particular have demonstrated high accuracy within the domains of speech recognition, image recognition, the game of Go, and internet search [63]. The general availability of machine learning algorithms and hardware capable of efficiently running them has created a market for increasingly intelligent systems embedded in edge devices and cloud services. As the best choices of hardware for deployment of future deep neural networks continue to evolve, specialized platforms that sacrifice programmability for efficiency are becoming increasingly prevalent for these workloads. Several companies with large data centers that experience high volumes of machine learning workloads have recently invested in developing in-house acceleration for artificial intelligence workloads [64, 5].

1.2 Neuroscience Comparisons and Insights

It is not surprising that the algorithms that have been shown to be most competitive at perceptual tasks such as processing real world images and languages are those that resemble biological systems that evolved processing similar real-world stimuli. As the evolution of biological nervous systems was subject to many of the same charge transport principles that are the basis of electronic systems, there are also certain parallels that can be drawn between efficient circuit design techniques and the architecture and mechanisms of the computational paradigm of the brain.

Many of the complexities of the cellular structures instrumental in the brain's evolution preclude adoption of certain aspects of the brain's underlying functionality within an engineering context. Factors such as power demands, planar layouts, and economic pressures affecting CMOS technologies constrain the form factors of silicon technologies and curtail the interconnectivity of the processing elements themselves as compared to biological neurons. So, the manner in which the influence of common physical principles are expressed differ between brains and neural network accelerators. Nonetheless, there are many useful engineering insights that can be gained from understanding the characteristics, behaviors, and organization of neural circuits [83], including efficient communication protocols, low precision operation, asynchrony, locality, parallelism, specialization, and heterogeneity.

1.2.1 Locality

The brain itself is an extremely efficient system, operating under a power envelope of about 20 Watts, which accounts for 20% of the energy that a body's metabolism must support [8], about 80% of which is associated with signaling and communication [35]. The human brain contains about 10^8 meters of wiring [23], where the connectivity between neurons maintains a "small-world" characteristic that incorporates sparse long distance connections alongside dense local connections. The heterogeneous distribution of functionality within different areas of the brain can be partially attributed to this tendency toward small world network connectivity. *Heterogeneity* exists within many hierarchical levels within the brain [8], from different neuron cell types to different functional brain regions. Much like the motivation for integration of acceleration into SoCs and data centers described earlier in this chapter, specialization of functionality within the brain enables functional structures optimized for particular tasks to operate more efficiently.

This preference for *local connectivity* is mirrored at many levels of abstraction within the computing ecosystem, like the connections between transistors in an integrated circuit. Whether communication is facilitated by electrical signals between routers or action potentials propagating along axons [88], the energy consumed by communication and the latency of communication is related to the distance that signals must travel. So, it is not surprising that different systems constrained by energy consumption and latency of communication would tend toward locality preserving formations, whether derived by millions of years of evolutionary processes or by sound engineering principles. Local connectivity is the basis for the energy efficiency achieved by the links between neighboring computational elements found in systolic arrays, which will be discussed in detail in Chapter 3.

1.2.2 Relaxed Synchronization

Biological neural circuits do not depend on centralized clocks for operation and are therefore more like asynchronous than synchronous circuit designs [148], where synchronous behaviors are facilitated by the intrinsic timing characteristics of the neurons themselves [23]. In digital circuits, the notion of asynchrony is typically associated with self-timed circuits, found in wave-front arrays [80] and the quasi-delay insensitive circuits used in TrueNorth [2]. Circuit timing in self-timed circuits is controlled based on the intrinsic propagation delays within each logic block. These asynchronous circuit design techniques have some attractive properties, like elimination of the overheads associated with sharing a global clock across many logic blocks and innate robustness to parametric variation. However, asynchronous circuits require completion signaling logic for each combinational logic block and handshaking logic between logic blocks, both of which have associated overheads [114].

Synchronous design techniques have mature and easily accessible tool-chains that do not require these additional asynchronous overheads but suffer from rigid synchronization requirements controlled by a centralized clock. The clock distribution nets within digital circuits can account for a substantial portion of power consumption, which can be mitigated by circuit design techniques that minimize the number clocked elements. The data propagation patterns through systolic arrays involve temporal semantics that require precise timing. As a result, the clocked elements within these arrays can account for a substantial portion of the power consumption within the systolic array. The discussion in Chapter 3 explores design techniques that minimizes the number of clocked elements in systolic arrays to eliminate extraneous overheads associated with sequential logic elements. The resulting computational array designs will be referred to as diastolic arrays within this dissertation to differentiate from systolic arrays.

The diastolic name is not unique to this work. Surprisingly, only one paper has been identified that uses the diastolic name. This other work is about reconfigurable arrays [20] and is entirely unrelated to the notion of "diastolic arrays" described within this dissertation.

1.2.3 Low Precision

There is evidence that neuron scaling played a pivotal role in the course of human evolution in that miniaturization of the neuron led to reduction of the energy consumption of neurons and enabled efficient construction of more complex neural circuits. Some have hypothesized that human evolution may have even reached a lower limit on neuron size, reminiscent of the end of Moore's law, imposed by reduced precision of the neuron's ion channels at smaller scales [35], introducing a trade-off between the tolerance of natural neural networks to the imprecision inherent in their neuronal components and their energy consumption. This loss of precision is conceptually similar to the increased susceptibility to some of the effects of parametric variation in deep-submicron CMOS, which are generally managed through establishing voltage-frequency domains [68] where proper functionality is shielded from being affected by variation.

One of the ideas that has been widely adopted by recent work on efficient neural networks has been the use of *low precision* numeric representations, which will be discussed in more detail in Chapter 4, where the impact of both low precision weights and activations are considered as well as efficient circuit design techniques for accelerating low precision neural networks. An accelerator designed for codebook based quantization is considered in which a small set of representative values are selected for the weights and activations of each layer such that each weight and activation can be stored and communicated as codebook indices. This non-linear mapping from bit sequence to represented value achieves compression ratios that provide better accuracy than comparable linear mappings, as will be demonstrated in Chapter 5, and provides the flexibility necessary to achieve an optimal mapping that is not necessarily present within structured quantization mappings derived from evaluation of a parameterized mathematical expression rather than from a codebook lookup.

1.2.4 Functional Signal Timing

The role of intrinsic timing behavior of biological neurons is deeply embedded within their computational paradigm. There are logic styles that utilize delays for computational purposes, known as race logic [125, 95], in which multiple path delays within a winnertake-all regime are used for computation. Although the evaluation in [95] is primarily synchronous, they do describe an asynchronous design based on intrinsic propagation delay through configurable memristive devices, which they suggest would consume less energy than synchronous delays.

Functional delay based circuitry has also been proposed for on-chip communication [100], where the timing of signal toggles is used to encode bit sequences. These *temporal communication* protocols are reminiscent of the way biological neurons communicate by propagation of spikes of potential along an axon. A system that communicates using spikes is capable of being more energy efficient than a system where all information must be captured by bits. This is as true for the brain as it is for digital circuits. Both types of spikes represent changes to electric potential caused by movement of charged particles and in both cases a significant portion of the average energy consumption is a linear function of the spike rate [40, 6]. When integrated into digital systems, the overhead associated with translation between binary and temporal formats can become significant, so these protocols are only appropriate within interconnects that are sufficiently capacitive to provide greater energy savings than the introduced transmission overhead. The latency associated with temporal protocols can also present a problem within systems with tight latency constraints. Within humans, this latency is on similar order as the response times that we experience, with a minimum spacing between spikes of about 1KHz. Within integrated circuits the latency constraint imposed on the transmission protocol is dependent upon the purpose of the interconnect, which may be more forgiving for some applications than others.

The discussion in Chapter 6 explores the integration of temporal codes as the interconnect supporting communication of synaptic weights within a neural network accelerator in order to reduce the energy consumed during weight and activation retrieval. Within this context, the data reuse characteristics of the computation schedules and cache capacities at each processing element dictate the latency tolerance of the accelerator. While increasing buffer capacities provide better flexibility, provisioning of buffer capacity to target tolerance for particular schedules can enforce adherence to power budgets by building preference for efficient schedules into the accelerator.

1.2.5 von Neumann

The *parallel* operation of neurons provides an execution model that achieves very high throughput despite the low operating frequency of the neurons themselves. Although parallelism itself is essential for high performance operation of the many subsystems within digital processing technologies, the high degree of parallelism found in biological nervous systems is in stark contrast to the serial operation of the von Neumann inspired processors we are most familiar with. These digital systems depend heavily upon multiplexing of functional units to execute sequences of instructions, multiplexing of cache hierarchies to reduce data movement, multiplexing of memories to support large virtual address spaces, and multiplexing of interconnect as packet switched on-chip networks. Multiplexing decouples resources from their application. Von Neumann architectures separate memory resources from compute resources to multiplex compute resources across many applications.

When processing neural networks on general purpose platforms, synaptic weights are stored in memory, just as any other data would be, and the same functional units are used to process many different neurons. In contrast, each neuron within the brain represents an instance of tightly-coupled weights and computation. Multiplexing does occur within the brain, in the ways that memories are recalled [8], however the mechanisms of plasticity within the brain seem far more complex than computer programming.

Neural network accelerators generally require multiplexing of compute resources for processing of many neurons as well. Energy efficient neural network inference is generally achieved by scheduling computations in such a way that maximizes data reuse and thereby minimizes data movement. Energy efficiency can also be achieved by maintaining synaptic weights in close proximity to dedicated compute resources. However, the cost inefficiency of specialization under low utilization is difficult to justify.

1.3 Thesis Contributions

Within this dissertation, three different neural network inference accelerator designs are considered. The first is a monolithic systolic array based accelerator where the pipeline depth within the systolic array is reduced in order to eliminate the overheads associated with clocked elements. These arrays, referred to as diastolic arrays, use adder tree based reduction combined with systolic summation in order to eliminate the need for clocked elements imposed by the temporal semantics of systolic arrays. This technique primarily improves energy efficiency by performing larger portions of the neuron dot-product before storing partial sums to clocked elements. These techniques also improve performance subject to other bandwidth limitations.

The accelerator description includes considerations for computation sequencing based on weight permutation mechanisms used to support convolutional layers that are not wide enough to span the array without processing multiple locations at a time. With these mechanisms in place, the processing sequence can exploit temporal locality within a relatively small weight buffer that can be used to reduce the number of memory accesses associated with loading multiple permutations of the same set of weights. A sensitivity study is conducted to evaluate the impact of array size for a set of CNNs drawn from winners of the ImageNet competition that demonstrates that array sizes of 128x128 are the largest array sizes which achieve high enough utilization to benefit from the energy efficiency associated with systolic arrays. Furthermore, given analysis of tiled architectures containing multiple diastolic arrays of different dimensions but equivalent total compute capacity, a tiled architecture of four 128x128 diastolic arrays is shown to achieve a favorable energy-delay product due to the performance improvements provided by tiling combined with the energy efficiency of the diastolic design technique.

The second accelerator uses codebook quantization for both weights and activations in order to reduce power associated with both on-chip communication and synapse calculation. While some previous accelerators have considered codebook quantization of weights, codebook quantization of activations is rarely discussed in the research literature. Several interesting points within the design space of wide data-path accelerator processing elements with integrated codebook based dequantization are considered. This includes a way to reduce the number of multipliers when the number of weight codebook entries is smaller than the output bandwidth of the processing element as well as a lookup table based multiplier, which take advantage of the compressed format of both the weights and activations to replace the multiplication operation associated with synapse calculation with reads from small storage arrays. The scaling properties of these techniques are noteworthy since they demonstrate a spectrum of accelerator designs where dependent upon the target codebook sizes and the properties of the underlying technology, the optimal accelerator may be any one the accelerator design points considered.

Training techniques for codebook quantization of activations are considered, including discussion of recent related work which has followed similar approaches for comparable activation encoding schemes. Some training experiments are presented which demonstrate a continuous-discrete learning method (CDLM) [22] based training technique that is capable of recovering accuracy after retraining underlying weights for a network with fixed codebook entries determined using k-means for both weights and activations. This technique is compared with a technique that attempts weight codebook entry training, similar to the more recently proposed Deep Compression [49] technique, however better accuracy is generally achieved by training the underlying weights. These experiments are based on networks trained using the Cifar-10 data set which provides abbreviated training times in comparison to more complex data sets.

The third accelerator design considers data movement power reduction within a tiled accelerator by using temporally coded interconnects. Unlike natural neural networks where weights are tightly coupled to the processing element and activations are communicated as temporal sequences of spikes, this accelerator design uses temporal codes for both activations and weights. Tolerance for the latency of the temporal codes is achieved by optimizing for batch sizes of four, such that sub-optimal processing schedules result in performance degradation rather than peak power elevation.

1.4 Thesis Organization

The thesis is organized as follows. Accelerators based on diastolic arrays are described in Chapter 3. Accelerators designed for codebook based quantization of both activations and weights using lookup table based synapse calculation are described in Chapter 4, with associated training experiments detailed in Chapter 5. An accelerator with temporal coded interconnects is discussed in Chapter 6. The simulation infrastructure used for evaluation of each accelerator is discussed in Chapter 7. A broad discussion of neural network acceleration is included in Chapter 2, with some concluding remarks in Chapter 8.

2 NEURAL NETWORK ACCELERATION

Nature is the great visible engine of creativity, against which all other creative efforts are measured.

— Terence McKenna

2.1 Artificial Neural Networks

Artificial neural networks (ANNs) are algorithms that process information in a manner that resembles the computational mechanisms of natural neural networks. As a thriving sub-field of machine learning, neural networks have been engineered specifically to address problems entirely independent of the biological systems from which their name is derived. As such, many inconsistencies exist between the artificial models of neural networks and biological models, including but not limited to the numeric representations used by computer systems [101] and decoupling of the network model from the mechanisms of computation. Within the context of the study of neuroscience, neural networks represent but one level of abstraction useful for describing the way the brain operates. As such, the neural networks of biological interest must be consistent with other levels of abstraction from the molecular structures involved in synaptic activity, to the cellular structures of neurons, to the time scales at which motor responses are produced based on sensory inputs [23]. Although simulations exploring each level of abstraction are useful within neurobiological research, the successful application of artificial neural networks in practical applications serves as more of proof of concept that these mechanisms are effective, from a scientific perspective.

Uninhibited by the evolutionary processes responsible for the genetic underpinnings of the brain's construction, novel domain specific acceleration designs are readily optimized for a particular class of applications [122, 131, 70]. Neural networks are well suited for hardware acceleration, due to their repeated use of simple computational elements and high parallelizability. Within the realm of machine learning, neural networks are not constrained by their dependence upon the underlying biology necessary for supporting living cells. The underlying technology supporting accelerator implementations provides a large design space for engineers to work within, and the wide range of applications of artificial intelligence provides excellent motivation for production of neural network accelerators. Popularity of neural networks for machine learning problems will most likely continue to grow as neural network acceleration providing higher performance and better efficiency than alternative techniques becomes more easily available.

The artificial neuron, illustrated in Figure 2.1, models synapses as the product of synaptic weights and input stimuli. A neuron's synaptic activity is aggregated as a sum of products, and a non-linear activation function is applied to the weighted sum, as in Eq. 2.1

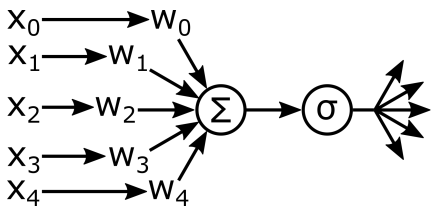


Figure 2.1: Artificial Neuron Model

$$n_j = \sigma\left(\sum_{i=0}^{n-1} W_{ij} a_i\right), \qquad (2.1)$$

where σ is the activation function, W_{ij} are elements of the weight matrix, a_i are elements

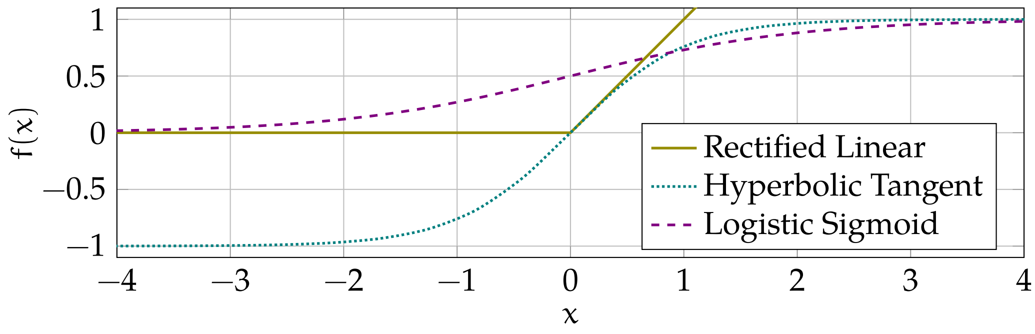


Figure 2.2: Common Activation Functions

of the input activation vector, and n_j are elements of the output activation vector.

The inputs to each ANN neuron are generated as the output of an activation function. These activation functions can be used to represent the probable firing rate within biological neurons in response to accumulated membrane potentials [26, 39]. Three common ANN activation functions, plotted in figure 2.2, are the ReLU, tanh, and sigmoid. The linearity of the ReLU makes it easy to calculate. The tanh and sigmoid are non-linear and involve more complicated calculations that are sometimes implemented as piece-wise linear lookup tables [134] to more efficiently approximate the activation calculation.

Neural networks can be either recurrent or feed-forward in nature. Deep neural networks (DNN's) are layer based architectures which are for the most part feed-forward with some exceptions. The long-short term memory units (LSTMs) used within some DNN's contain units that are internally recurrent to maintain a state vector from one iteration to the next. Other recurrent networks, such as those found in reservoir computing and echo state machines include a pool of neurons with random synaptic connections that may exist between any neurons in the pool. Recurrent neural networks are useful for processing data that is sequential in nature, like speech or text. Convolutional neural networks (CNN's) are a purely feed-forward type of DNN that is useful for processing data that is spatial in nature, such as images. CNN's include convolutional layers which use a two-dimensional grid of spatially distributed neurons where a kernel defines a common set of weights used by neurons at each location in the grid.

Feed-forward neural networks following the multi-layer perceptron (MLP) model contain an input layer, one or more hidden layers, and an output layer, as in Figure 2.3. The input layer accepts the inputs to the model which may be images, sometimes with some pre-processing applied. Labels are associated with the output layer of the MLP such that each output neuron represents a particular classification. The synaptic weights for all layers of the MLP are typically trained by an iterative process known as stochastic gradient descent.

Convolutional neural networks contain convolutional layers where shared weights represent filter banks used for spatial feature extraction. Sharing weights enables many

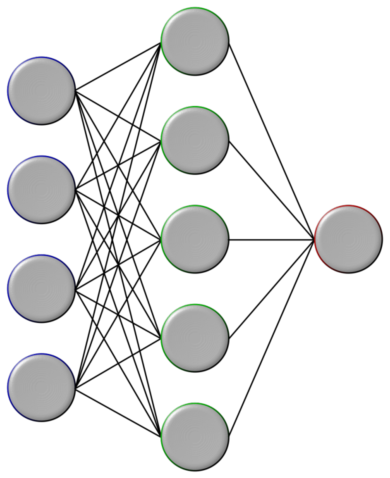


Figure 2.3: Multilayer Perceptron

neurons to be represented by a relatively small set of weights befitting of storage space constrained computing environments. The convolution operation of the convolutional layer applies a set of filters to a stack of two dimensional feature maps. The filters are applied as a two dimensional sliding window, like conventional convolution, where each output value is determined by synaptic weights applied at a location within the feature map. The filter sets are therefore four dimensional, where two of the dimensions are spatial and the other two dimensions are source and destination feature channels. The filters in convolutional neural networks are typically square with odd spatial dimensions. Typical filter sizes include 1x1, 3x3, 5x5, or 7x7. A stride parameter is sometimes specified as a spatial step size, but this step size is often 1. When stride>1 the output shrinks spatially relative to the input, similar to the effect of pooling. The convolutional layers compress a relatively large number of computations into a smaller set of kernel weights. The kernel weights of a convolutional layer can represent a substantial memory footprint, despite their reuse properties.

The widespread rise of interest in deep learning was sparked once convolutional neural network based classifiers began winning the ImageNet challenge [118], which started in 2012 with AlexNet [77] and continued in each subsequent year of the challenge with CNN topologies such as VGG16, Resnet, and GoogLeNet [130, 124, 55], with the most recent winners being based on ensembles of multiple neural networks. The rising interest in deep learning has led to rising interest in neural network accelerators capable of efficiently processing neural networks.

2.2 Training

Since the time of AlexNet's introduction, many hardware cognizant neural network designs have emerged as well as many accelerator architectures targeting neural networks. While some accelerator proposals have targeted both inference and training [16, 138], many focus exclusively on inference. There are a few possible explanations of the preference for inference acceleration, the first of which is that a well trained model will typically be used for inference many more times than the time spent training the model, which makes acceleration of repeated inferences more valuable than the one time cost of training a given network.

Although inference accelerators can be used for portions of the training process, training introduces certain complexities requiring support for operations which would not need to be supported by an accelerator designed exclusively for inference. For example, the softmax operation and loss evaluation should not present a significant bottleneck since there are generally much fewer neurons in the output layer than other layers, such that acceleration of these operations would provide little benefit over general purpose implementations. The forward propagation inference step is a part of training, so inference accelerators can be used during forward propagation. Backward propagation requires similar operations to forward propagation, so it is possible to use an accelerator designed for inference to compute gradients, where the complexities associated with backpropagation would represent incremental modifications to an inference accelerator.

Finally, the memory usage during training is significantly higher than during inference. Intermediate layer outputs, which are transient during inference, must be stored for later use during back propagation. For convolutional layers, the storage requirement for intermediate feature maps eclipses that of the convolutional kernels [117], so acceleration of training has less of an impact on overall efficiency of the training system, since memory overheads begin to outweigh the latency reductions and energy savings associated with acceleration.

The more significant acceleration opportunity exists within inference, suggesting that a computing ecosystem that supports development of training techniques that are uninhibited by the limitations imposed by a training accelerator would enable exploration of training techniques that target efficient inference using particular accelerators. Today, training tasks that previously would have required a week of training on a single GPU have been demonstrated to complete training to target accuracy within minutes using a cluster of thousands of GPU's [143, 61, 99] or TPU's [145]. These training techniques use very large mini-batches to achieve high throughput, which can increase convergence times if not done carefully. Hyper-parameter tuning is generally required to achieve these convergence rates.

2.3 Inference Acceleration

The notion of a neural network accelerator can take many forms. The most general form includes basically any hardware that augments a CPU based system to provide improved performance or efficiency when processing neural networks. This accelerator may be IP tightly integrated on-chip into an SoC or it may be a separate chip accessed over PCIe.

An accelerator can be programmable like GPUs and DSPs, they can be reconfigurable like FPGAs, or they can be specialized fixed function ASICs. Due to their predefined data-flow characteristics, domain specific ASICs do not require many of the complex microarchitectural mechanisms of single thread optimized processors that provide features such as general purpose branch prediction and out-of-order execution. Although programmable general purpose compute platforms such as CPUs and GPUs are good options for portability, they can be less efficient than more specialized implementations where data paths, control flow and interconnects can be tailored to the domain of interest. Portions of the calculations involved in evaluating a neural network are inherently serial in nature, so there may not be enough parallelism to benefit from execution of all neural network layers on architectures with massively parallel sets of processing elements. So, performing inference on a GPU is not necessarily worthwhile, however recent NVIDIA GPUs have integrated Tensor cores, which essentially builds in a matrix multiplication accelerator within the GPU to enable more efficient inference evaluation than previous generations [71].

FPGAs provide a reconfigurable substrate for implementation of either domain specific accelerators or programmable processors. FPGAs provide greater data path configurability and therefore can provide greater efficiency than GPUs [50, 140, 136, 106] and in some cases retain portability provided by OpenCL [7]. FPGA's are generally less efficient than hard-wired implementations of the same architecture, but due to their specificity, widespread usage of the class of applications addressed by an accelerator is fundamentally required for silicon to be cost effective.

Artificial intelligence and more specifically artificial neural networks are an increasingly prevalent target for domain specific acceleration due to the many applications for which these accelerators can be used. Although this dissertation focuses for the most part on ASICs, FPGAs are a cost effective alternative for inference acceleration [142] which provides homogeneously upgradable hardware [113], that can even be dynamically reconfigured to other domain specific needs.

2.3.1 Scaling

As new generations of deep nets become more accurate, there is generally an associated rise in the size of the model, either as rising network depth [55] or width [147]. Adding depth to a topology tends to improve its ability to generalize [17] while increasing width improves memorization [17]. Both can improve achievable network accuracy and support for both types of scaling is worthwhile. Increasing model sizes require systems processing neural networks to scale to accommodate increasing storage and memory requirements, whether width or depth are being scaled. Width scaling results in more active memory per layer that benefits from higher intra-layer throughput, which can be provided by processing elements capable of large vector dot-products. Depth scaling results in an elongated critical path which benefits from higher inter-layer throughput, which may be better provided by a pipeline of distributed processing elements [36], when the system's processing capacity is over-provisioned with respect to a single layer.

In general, there are two ways to scale system storage capacity to accommodate increasing model sizes: scaling up of memory per node and scaling out to multiple nodes. The preferred mode of scalability is dependent upon the form factor of the system, in that scaling out may not be possible within a spatially constrained mobile device, while scaling up memory in a data center setting may not provide as much throughput gain as scaling out to multiple nodes. It may be more cost effective to over-provision the memory capacity for the accelerator in order to avoid the need to invest in additional systems, however accelerators with enough on-chip memory capacity to store a full set of weights may be able to alleviate their dependence on memory accesses after scaling out and not experience gains from scaling up.

Inference accelerators that maintain large on-chip memories allocated to weights like DaDianNao [16] and True North [2] are highly efficient. However, without a high bandwidth memory system to fall back on, the on-chip memory capacity can become a limiting factor, such that scaling-out to multiple nodes is required to accommodate larger network topologies. These architectures can minimize movement of synaptic weights and provide an alternative to implementations where off-chip memory is used to scale to larger networks, where memory bandwidth can easily become a bottleneck.

Accelerators with greater on-chip memory capacities can avoid more DRAM accesses by using mapping and scheduling schemes that exploit temporal locality within data retrieved from DRAM. Although reducing the number of DRAM accesses reduces power, larger on-chip memories have increased static power and access energy. Well established techniques exist for mitigating static power in SRAMs, including high-κ gate dielectrics and FinFET based gates, as well as dynamic techniques that place SRAM into low power states with cache-line granularity [33] or use power gating of unused capacity [112]. However, the dynamic power consumption associated with each access is largely a function of the data movement distance and the data values being retrieved. Distributing the memory capacity across many compute tiles can reduce the distance between data and its compute resource but at the expense of capacity available to each tile and with potential duplication of data within different partitions of the on-chip weight storage. This notion of collocating weights with memory can also be seen in accelerator proposals that integrate 3D stacked DRAM [37, 72] to reduce data movement distance of DRAM accesses. Several other proposals have considered arrays of memristive storage devices where activations are brought to the location of weight storage and computation takes place within the arrays [121, 19].

2.3.2 Schedules

Activations, weights, and partial sums involved in active calculations can be maintained in storage structures closer to the processing elements in order to reduce data movement outside of the processing element, assuming a processing schedule is used that makes use of these storage resources to exploit the algorithm's locality properties. The size and organization of this active data dictates how efficiently the on-chip storage can be used and therefore how much locality can be preserved. Data reuse can exist for neuron inputs due to sharing of inputs across multiple neurons, for neuron outputs since many synapses must accumulate to calculate each output, and for weights within convolutional layers or when large batch sizes are used. Weight reuse is also present in recurrent neural networks where heuristics which allow multiple time steps to be computed simultaneously can be utilized to achieve temporal locality of weights [151].

Feed forward neural networks group neurons into layers where the inputs to each layer are the outputs of the preceding layer. So feed forward networks are typically processed one layer at a time to minimize the number of neurons that are actively being computed at once. When processing convolutional layers where only a subset of activations is required to process each neuron, multiple layers can be actively evaluated at once [4]. Inter-layer schedules are useful for layers with large spatial dimensions and are particularly well suited for accelerators implemented using reconfigurable substrates where computational resource allocation can be tailored to a neural network topology.

Weight movement can be very expensive, so holding weights local to processing can be beneficial. In an extreme case, if the weights of a systolic array are held constant throughout the duration of an inference and only change through seldom reconfiguration, the processing elements are underutilized and many systolic array instances are required to support a full network. While this would be a viable scheduling scheme for a large network of TPU's, it also resembles the designs being proposed for in-situ memristive storage of weights in which the weights are stored within memristive cells and dot products can be performed in the analog domain. This tight coupling of memory to computation represents a step away from the von Neumann model of computation and better resembles the distributed model of computation found in the brain. Processing in memory is therefore an attractive computational paradigm for neural networks.

Many array based neural network accelerator architectures include some amount of storage within the processing elements. Eyeriss [15] contains a register-file within each processing element which can be used for different purposes depending upon the data-flow configuration. FlexFlow [93] integrates designated FIFO's for weights and inputs and is reconfigurable for many different data flows with different reuse characteristics. Chapter 3 discusses scheduling techniques for an accelerator design with static data flows that attempts to minimized weight memory accesses by holding weights constant within the array for short epochs while either many convolutional neurons sharing the same set of weights are processed or fully-connected neurons are evaluated for multiple input samples of a larger batch. While similar data-flows are achievable within reconfigurable accelerators, a trade-off exists between the overheads of reconfigurability and the applicability of the accelerator to different dataflows, where an ability to achieve high utilization with alternative dataflows can offset the cost of reconfigurability [93].

2.3.3 Quantization

Data formats have a significant impact on bandwidth, storage, and processing requirements of the accelerator. Many data formats have been considered for neural networks from traditional single-precision floating points and integers to codebook based quantization. Quantization schemes designed for low-precision fixed point arithmetic have been one mainstay of several accelerator designs due to their computational efficiency and their demonstrated inference accuracy. Different precision requirements have been demonstrated for different neural network layers [65, 123] and some have proposed quantization to as low as a single bit [25, 116].

Floating point units are significantly more complex than fixed point units since floating point arithmetic combines mantissa operations with exponent calculation, both of which can be expressed in terms of integer operations but require additional logic for normalization of the mantissa result and adjustment of the exponent according to the leading bit of the mantissa calculation. So fixed point arithmetic can be significantly more efficient than floating point.

Neural network accelerators built for reduced precision achieve energy efficiency improvements for two reasons. The first is that low precision accelerators only need to support low precision fixed point arithmetic and thereby achieve higher efficiency than general purpose CPUs and GPUs which often support a wider range of numeric formats. The second is that low precision synaptic weights experience low memory bandwidth requirements and efficient utilization of storage resources. Processing neural networks using low-precision fixed point arithmetic is easily integrated into software implementations of neural networks [137, 141].

Encoding schemes that use codebook quantization and variable length encodings [48, 49] increase compression ratios for further improvement to resource utilization, without affecting the underlying arithmetic operations. However, codebook quantized values need to be decoded before they can be used for computation. So data encodings based on codebook quantization place indirection for decoding along the arithmetic data path, which introduces codebook bandwidth as a potential bottleneck. In general purpose hardware, a lookup table residing in cache or scratchpad memory may be used to access a codebook fairly quickly. However, caches are subject to unpredictable contention with other memory requests and are therefore more prone to becoming a bottleneck than designated local storage. Scratchpads on the other hand are not included in most general purpose processors [74]. GPUs are an exception since they contain some specialized memory structures (constant, shared, etc.), but each structure is subject to its own preferred access patterns [60] that may not perform well on divergent memory requests. In contrast, specialized memory structures in ASICs can be designed specifically to address the needs of the targeted workloads, and high bandwidth codebook lookups can be integrated into the accelerator data path.

2.3.4 Sparsity

When zeroes are present within either the activation or weight vectors, the result of the individual scalar multiplication can be eliminated from the multiply and accumulate if they can be detected prior to performing the calculation. Doing so has a dramatic impact on the accelerator data path, particularly for wide data paths where sparsity may not map well to wide processing elements.

Weight Sparsity

Several studies [51, 49] have demonstrated that when weight pruning techniques are applied to neural networks, accuracy is often recoverable with a high compression ratio. The resulting sparse model formats can be expressed in a data format containing both indexes and value as either compressed sparse column (CSC) [48] or compressed sparse row (CSR) [146] which use relative indexing to map sparse weight values to the weight matrix.

Execution of sparse neural networks involves indirection along the data path which can degrade performance. Thus, accelerators designed specifically for sparse neural networks have been proposed that integrate this lookup tightly into the processing element. This has included proposals optimized for convolutional layers [108] as well as fully connected layers [48].

Activation Sparsity

Some researchers have designed accelerators that remove zeroes from the activations that are broadcast to the other neurons [48, 3]. When zero activations are skipped without compressing synapse memory then the identifier for the activation arriving at a processing element can be used to directly index into the synapse storage [3]. When zero activation skipping is combined with synapse sparsity, it becomes necessary to walk through the indirection tables to locate and retrieve the corresponding synapse weights [48]. In contrast, if weights are sparse and zero activations are not skipped then weight prefetching can be done non-speculatively.

The fundamental motivation for activation sparsity is to eliminate ineffectual computations within the networks. Within networks using ReLU activation functions many activation values are zeroes creating an opportunity for elimination of these zero values from the computations. Further pruning of the set of activations may be accomplished by using thresholding to eliminate activations subject to accuracy losses [3].

2.4 Chapter Summary

In this dissertation, the efficient synapse calculations provided by wide data paths are used to implement accelerators that exhibit high performance and energy efficiency. Although there have been accelerators proposed for sparse data format processing within wide data paths, they tend to introduce additional overhead associated with logic for packing the sparse format into vectors that can be processed using the wide data paths [150, 153]. In addition, sparsity has been demonstrated to degrade the classification confidence produced by sparse DNN's [144]. We avoid these added complexities associated with sparse neural networks; however they are worth noting as a highly efficient alternative to neural network processing based on dense matrices.

This dissertation focuses on design techniques for inference accelerators built to process neural networks expressed as dense matrices. It touches upon both the design of the processing elements of these accelerators and the schedules that efficiently harness them. Accelerators based on both monolithic systolic arrays which store weights in off-chip DRAM are considered as well as tiled architectures with tightly coupled weight memory within each tile. The analysis of the impact of tiling within the accelerator that uses off-chip weight storage provides some insights into bandwidth limitations that motivate tightly coupled high bandwidth storage within each tile. The preferred scaling modalities of these two design points differ due to the memory independence associated with large on-chip weight storage.

The data formats supported by an accelerator play a critical role in processing efficiency. Codebook based compression provides higher compression ratios with comparable accuracy to fixed point formats, yet there are only a few accelerator proposals that use codebook based compression. One of the benefits of these compression schemes is their reduction of data movement costs, which can be further improved upon by designing on-chip networks tailored to the intended data access pattern.

3 DIASTOLIC ARRAYS

Don't be trapped by dogma - which is living with the results of other people's thinking. Don't let the noise of others' opinions drown out your own inner voice. And most important, have the courage to follow your heart and intuition.

— Steve Jobs

Systolic array based neural network accelerator architectures use algorithm-specific parallel processing fabrics to efficiently process neural networks. Some recent developments in neural network acceleration approaches have included systolic array designs at scales far exceeding commonly studied systolic architectures. Thus, there is a need to revisit the design space implications associated with large scale arrays designed for neural network inference. In this chapter we explore the design space of an accelerator based on Google's Tensor Processing Unit (TPU), considering circuit design techniques intended to mitigate the power consumption associated with clocked elements within the baseline TPU's systolic array.

Clocked element elimination techniques are considered along each of the interwoven pipelines of the systolic array, including the summation pipeline which expands to adder tree based cells, the activation propagation pipeline which expands to share input operands across multiple dot-product lanes, and the weight distribution pipeline which expands to buses shared across multiple weight matrix columns. Parameter distribution mechanisms supporting multi-permutation weight stationary data-flows are developed within the context of arrays with expanded cells, as well as mapping and scheduling techniques that improve sub-matrix permutation locality in order to minimize memory traffic. We further explore the trade-offs between monolithic arrays and tiling of multiple arrays within the same package. Exploration considering the cell expansion techniques, permutation based scheduling, and tiling have demonstrated batch sized one inference with better than 2x energy-efficiency and better than 3x performance improvement in comparison to a TPU-like baseline, with 72% and 79% improvement on average respectively for performance and efficiency.

3.1 Introduction

Large scale acceleration of neural network inference requires systems with high memory bandwidth and parallel compute fabrics capable of supporting the highly parallelizable data flows characteristic of neural networks. Data movement can account for a substantial portion of the energy consumed when evaluating neural networks, so mechanisms for exploiting locality that alleviate memory bandwidth pressure are critical components of efficient neural processing systems. Proper provisioning of specialized resources for neural network data flows considers on-chip storage capacities, off-chip bandwidths, and compute bandwidth provided by processing elements with wide access semantics, like Google's Tensor Processing Unit (TPU). The TPU contains a massive 2D systolic array capable of 4096 MACs per cycle.

Accelerators with large monolithic 2D arrays of functional units [63, 82, 15] benefit from operation of all functional units collectively, which requires relatively simple array control logic. Alternative implementations with processing distributed across multiple nodes containing smaller 2D arrays of functional units [14, 16, 91, 150, 34] where simplification of the control logic is achieved when the arrays are operated in lock-step with synchronization points to preserve coordinated memory accesses, which depending upon neural network mapping may produce multi-cast opportunities.

The TPU's weights are stored off chip, such that weight memory bandwidth can be critical for workloads demanding high weight throughput [63]. Optimal processing sequences preserve access locality as much as possible, reduce the number of memory accesses which both reduces memory access power consumption as well as avoids the memory bottleneck that may be present. Other on-chip bandwidths can impede the computational throughput as well. The computational bandwidth of the array represents the peak processing throughput, which is only achievable provided sufficient memory bandwidth to support workload sequencing demands and given workloads with array mappings that achieve high utilization.

The 2D arrays described above are implemented as 2D systolic arrays, which comprise an interwoven set of pipelines with regular data propagation patterns between modular processing elements that exploit operand reuse to achieve high throughput, even with modest memory bandwidths [79], provided workloads can achieve high array utilization. The multi-node designs expose more of the total computational bandwidth to on-chip interconnects which operate less efficiently than the data propagation patterns within the systolic array.

Convolutional neural networks (CNNs) make use of structured reuse patterns to provide high ratios of feature computation to weight memory accesses. Fully-connected layers lack synaptic weight reuse unless batch sizes larger than one are provided, but it is often the case that large batches are not available for inference tasks [34, 63]. So fully-connected layers are generally memory bound and therefore may not be able to achieve the same throughput as convolutional layers. Compute fabrics capable of supporting high data movement bandwidths are ideal for supporting fully-connected layers, but accelerators that target fully-connected layers by providing weight bandwidths that closely match the available compute resource bandwidths can result in designs where additional compute bandwidth would result in significant CNN speedups. CNNs often contain many more convolutional layers than fully-connected layers, so accelerators that target convolutional layers may sacrifice fully-connected layer processing latency without a noticeable performance impact for many CNNs. However, within accelerators targeting both fully-connected and convolutional layers, large processor arrays can be designed to support both the high weight bandwidths required for efficiently processing fully-connected layers and the local weight reuse capacity required for efficiently processing convolutional layers. The main caveat of these designs is that not all convolutional layers are easily mapped to extremely large arrays.

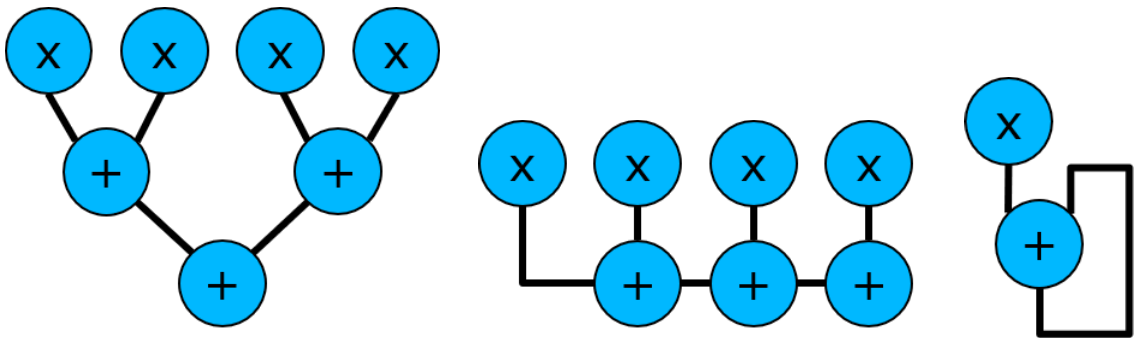
Proper techniques for mapping and sequencing the calculations of a convolutional layer are dependent upon the capabilities and distribution of resources of the underlying architecture [15, 37]. Making use of the compute resources provided by large processor arrays requires processing schedules that favor reuse of the synaptic weights of convolutional layers which have lower memory bandwidth requirements than fully-connected layers. Convolutional kernel reuse patterns are easily mapped to wide data paths provided by TPU-like matrix multiplication accelerators when the dimensions of the processing elements do not exceed the number of input and output feature maps.

So, the convolutional layers of typical CNN benchmarks, which can often have from 64 to 512 features per convolutional kernel, can often be easily mapped to fairly large arrays without considering multiple locations in a single sub-matrix multiplication. However, convolutional kernels with fewer channels are not as easily mapped to over-sized arrays without under-utilization of the computational resources. Due to data alignment differences from one location to the next, these mappings can require data permutation support in order to reuse weights loaded to the array for input and output neurons requiring different data permutations.

The vast majority of operations performed during neural network inference are the multiplication and accumulation associated with matrix multiplication and vector inner products of the standard artificial neuron equation

$$n_j = \sigma\left(\sum_j a_i w_{ij}\right), \tag{3.1}$$

where n_j is the j^{th} neuron output, a_i is the i^{th} input activation, w_{ij} is the synaptic weight, and σ is a non-linear activation function applied to the sum. The manner in which an inner product is performed by a processor can therefore have a tremendous impact on performance and power characteristics of neural network evaluation, including the way calculations are sequenced on the accelerator. Computing inner products on a scalar processor requires that multiply-accumulate operations be based on iterative accumulation, as in Figure 3.1c. Alternatively, systolic arrays (Figure 3.1b) and adder trees (Figure 3.1a) perform spatial reductions on vector operands, requiring fewer



(a) Adder-Tree (b) Systolic Array (c) Iterative

Figure 3.1: Alternative Inner Product Unit Architectures.

iterations per dot product. The notion of an inner-product unit is easily extended to a two dimensional matrix-vector product unit through replication of the dot-product unit with vector operand sharing. These matrix-vector units efficiently make use of the locality inherent in repeated use of the vector for multiplication along each row of the matrix. The primary benefits of the systolic architecture are its modular layout and simple interconnects between cells. Data propagation through the array follows a temporal pattern that is reflected within the cascaded arrival time requirements of input operands delivered to the array. Adder trees have a slightly more complex physical layout than systolic arrays but require fewer clock cycles of reduction latency.

Due to the balanced structure of the adder tree, shallower pipelines can be used with relaxed timing constraints to achieve high throughput with fewer clocked storage elements than systolic arrays. Systolic arrays constructed using chains of adder tree based cells benefit from the power reductions associated with fewer clocked elements while retaining the modularity and high compute bandwidths associated with systolic arrays. Within this chapter, the term diastolic arrays is used to describe arrays which employ circuit design techniques requiring fewer clocked elements than the baseline systolic array. These diastolic techniques are considered along three data-flows of the 2D systolic array, the most effective of which is the summation data-flow described above. The second data-flow under consideration is the activation propagation, where operands can be shared by adjacent dot-product lanes within a single cycle rather than propagating from one lane to the next. The third data-flow considered is weight matrix loading, for which an equivalent bandwidth weight propagation network is considered that requires fewer clocked elements than baseline weight propagation.

This chapter makes the following contributions:

1. Detailed discussion of the diastolic array design space and its low-latency and energy efficient characteristics.

2. Development of a novel scheduling scheme for processing CNNs on large 2D systolic arrays and description of the simple hardware capabilities required to support it.

3. Evaluation of the power, performance, and energy efficiency characteristics of the diastolic arrays.

4. Sensitivity studies pertaining to array size and multi-array tiling.

5. Demonstrate nearly 79% energy efficiency and 72% performance improvements over the TPU-like baseline architecture for batch size 1 inference.

3.2 Motivation

Systolic arrays are densely packed parallel compute fabrics with inter-processor communication links supporting simple and well defined data flow patterns [79]. The modular layout of systolic arrays allows for an abbreviated VLSI design cycle yielding highly area efficient substrates. Due to the well-defined data flows of the array, systolic arrays are capable of high compute throughput with a comparatively low input and output bandwidths without dependence on auxiliary local storage structure within the array. However, achieving high utilization with a systolic array is only possible with certain types of algorithms, known as regular iterative algorithms [129], having operand communication patterns corresponding to the regular inter-processor links of the array.

Systolic arrays are not ideal for general purpose hardware due to the difficulty involved in generally mapping algorithms to the arrays. Although many algorithms can be mapped to systolic arrays, including convolution, IIR filters, DFT, matrix arithmetic, QR decomposition, sorting, searching, string matching and regular expressions [79], the systolic architectures that are capable of each of these algorithms may be different. Some general purpose spatial dataflow architectures like DySER [43] and TRIPS [120] bridge the gap with reconfigurable interconnects, and many FPGA's include DSP blocks that can be readily configured for systolic operation [142]. The overheads of reconfigurability may be preferable to the limited programmability of specialized architectures for many purposes, however there are overheads associated with reconfigurability that are not necessary within accelerators designed for specific data flows.

Domain specific architectures designed specifically for neural network inference can be highly efficient and are applicable to a wide range of inference tasks. Growing interest in accelerators specifically designed for processing neural networks has led some architects to turn to systolic arrays as the basis for neural network acceleration. A prominent example of this is found in Google's Tensor Processing Unit (TPU) [63].

In general, the benefit of data-path pipelining is high throughput stemming from reduction of the minimum clock period required for timing closure at a fixed voltage. However, there are several inefficiencies that are introduced by pipelining that affect power, latency, and area. Latency overheads are introduced by setup and hold times, which account for increasingly significant portions of the clock cycle at high frequencies. The additional clocked storage elements introduced by pipelining can account for a significant portion of dynamic power drain when left unmitigated by techniques like clock gating. Ideally, division of the data-path would result in evenly divided segments with equal latencies that sum to the same latency of the unpipelined path. However, in practice the clock period is bounded by the slowest pipeline stage and since it is not always possible to evenly divide the pipeline, there is generally a net increase in data-path latency that results from pipeline imbalance [56]. Although this is not a factor for cascaded single-cycle systolic array cells, it can become a problem with more complicated processing elements, potentially benefiting from retiming techniques for a more balanced pipeline [87].

When pipelining for the purpose of performance alone, the optimal pipeline depth is the one that provides the highest throughput. In typical CPU pipelines, instruction throughput is generally limited by hazards [56] and speculation accuracy [53], so the degree of pipelining that is worthwhile is dictated by the effectiveness of microarchitectural techniques like superscalar pipelines, dynamic scheduling, and speculation that make use of the instruction capacity provided by deeper pipelines. However, when power is a significant concern, judicious pipelining practices that avoid the power overheads of additional clocked elements save power while potentially sacrificing some of these performance benefits. Many of the associative structures and prediction logic associated with high IPC in CPU's are unnecessary within domain-specific accelerators, especially those with sufficiently predictable data flows to be amenable to a systolic array based implementation. The data flows associated with neural network inference are highly predictable and therefore much of the instruction handling logic associated with high performance CPU front ends is not required to achieve high throughput non-speculatively in neural network accelerators. Nevertheless, systolic arrays maintain very deep pipelines where each stage includes additional compute resources. The systolic array itself can represent a significant portion of power consumption with clocked elements alone representing 40% of the power drawn by the systolic array. Therefore, this work advocates systolic array designs that attempt to reduce the number of flip-flops contained within the systolic array without reducing the number of compute resources. The techniques used to reduce pipeline depths will are described in more detail in Section 3.4 after the baseline accelerator design and its operation are described in Section 3.3.

3.3 Accelerator Scheduling

The analysis conducted in this chapter uses an architecture loosely based on that of the TPU [63, 64]. The TPU operates as a coprocessor controlled by a CPU which delivers instructions to the TPU over a PCIe x16 I/O bus, obviating any need for a fetch unit. The instructions are large, and each represents a substantial sequence of operations such that the TPU is capable of DNN inference with minimal CPU interaction [64]. The first generation TPU contains a 256x256 systolic array of 8-bit MAC units with 16-bit outputs, 4MB of 256-element 32-bit accumulator storage (4096 entries), 8GB of off-chip read-only weight memory, a weight FIFO with sufficient capacity for four "tiles" worth of synaptic weights, a 24MB local activation storage, and a DMA interface to the host [64]. The architecture depends on data prefetching from weight memory in a manner that utilizes the ample opportunity to coordinate memory accesses during the long latency of each matrix operation.

The baseline architecture under consideration is shown in Figure 3.2. It contains a two-dimensional square $A \times A$ systolic array with an output buffer capacity of R_n rows of width A. Operations such as pooling and activation functions are performed external to the processing array after accumulation has completed. Input activations for the next layer are stored within the input activation buffer and weights are retrieved from the weight memory into the weight buffer, from which the weights are loaded into the matrix multiply unit. Each neural network layer is calculated by a sequence of matrix-multiplications, where weights are assumed to be stationary throughout each

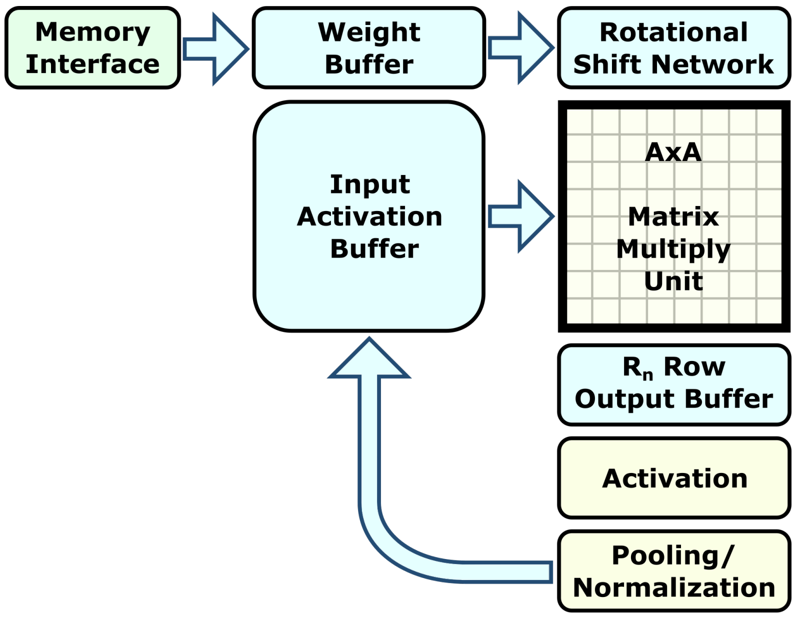


Figure 3.2: Baseline Accelerator Architecture

matrix-multiplication operation. The sequence of calculations impacts when weight sets are retrieved, the composition of those weight sets, and the amount of reuse that can be achieved for a given weight set. The following discussion pertains to the intricacies of sequencing CNN calculations on the array. Much of the discussion focuses on convolutional layers where the bulk of the weight reuse opportunity exists. However, it is instructive to first consider fully-connected layers, as the matrix multiplication partitioning takes the same general form, which can be considered to be a special case of convolutional layers where kernels and feature maps all have spatial dimensions of 1×1.

3.3.1 Fully-Connected Layers

Evaluating the neuron equation of Eq. 3.1 requires transformation of the calculation to a sequence of matrix multiplications constructed according to the configuration of the accelerator architecture and the layer dimensions of the neural network topology. Fully-connected layers are easily expressed as matrix-vector multiplications, or as matrix-matrix multiplications when batch size is larger than 1. Evaluating a fully-connected layer with M inputs, N outputs, and a batch size of B involves a matrix multiplication of the form

$$\underset{N\times B}{O} = \underset{N\times M}{W} \times \underset{M\times B}{I}, \tag{3.2}$$

where the weight matrix W with dimensions $N \times M$ is multiplied by the input matrix I with dimensions $M \times B$ to produce the output matrix O with dimensions $N \times B$. If the full matrix W fits into the array then scheduling the computation is easy. However, given a processing array with dimensions $A \times A$ where either $M > A$ or $N > A$, the matrix must be partitioned. When $M > A$, multiple component operations are required to calculate a given set of neurons. When $N > A$ the input matrix must be read multiple times to complete the full set of neurons for the layer. The number of rows in the output buffer, R_n affects the schedule when the batch size is large since if $B > R_n$ then each weight sub-matrix needs to be loaded into the array multiple times.

3.3.2 Convolutional Layers

Schedules for convolutional layers are a little more complicated to express using typical matrix operations since they involve multi-dimensional tensors that are not as easily converted to matrix multiplications. Due to the repetition of weights in convolutional layers, there are many ways to express the same convolution as a matrix multiplication. One way to lower the convolution operation into a matrix multiplication [18] is to create a weight matrix where each row contains the full set of kernel weights for each of N output channels as

$$\underset{N\times XY}{O} = \underset{N\times MK^2}{W} \times \underset{MK^2\times XY}{I}. \tag{3.3}$$

This formulation of the convolution packs all of the weights into a single matrix W without any weights repeated within the matrix. This same weight matrix can then be used at each output location but requires that the input layer be transformed into a matrix where each input is repeated several times within the matrix to account for the full set of output neurons. In the following discussion, we devise a minimal set of data alignment mechanisms necessary to support the matrix multiplication sequences of convolutional layers.

Kernel Matrix Partitioning

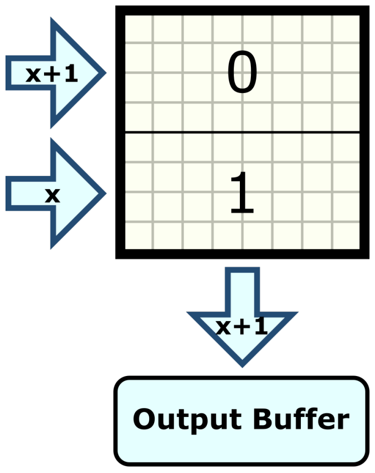
Alignment of weight data to the array's inputs either requires logic to shuffle input data to construct each sub-column of the input matrix or it requires that each input sub-column is itself aligned within the input buffer. Many convolutional layers have sufficiently many input and output channels to fill the array with weight sub-matrices pertaining to just one position of the filter at a time. For these layers, aligning the inputs to the array dimensions is easily achieved by constructing the weight matrix such that the rows are grouped by input channel and the matrix-matrix product is calculated by partitioning the weight matrix into sub-matrices each pertaining to a single input channel, such that only one filter position is considered at a time.

$$\underset{N\times XY}{O} = \sum_{k_y}\sum_{k_x} \underset{N\times M}{W_{k_x,k_y}} \times \underset{M\times XY}{I} \tag{3.4}$$

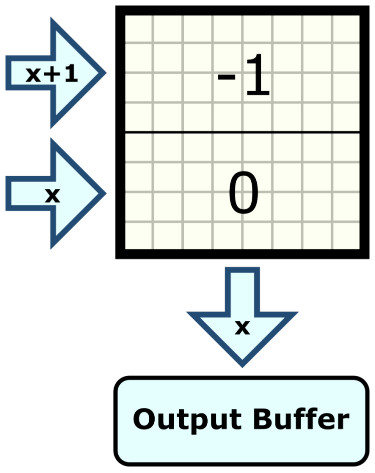
This is equivalent to Eq. 3.3, except this representation requires no repetitions within the input matrix and the matrix can be easily constructed by sequences of row aligned input buffer accesses.

Column-wise Permutation

Layers with fewer input channels than the width of the processing array are also common within convolutional neural networks, especially within the first layers where the input channels often represent the image color channels. These cases would suffer from under-utilization of the array unless multiple input positions are processed with each



(a) Odd Permutation



(b) Even Permutation

Figure 3.3: Column-wise Permutation Examples with input activations pertaining to two consecutive locations of the feature map arriving from the left and outputting to one output location, where column-wise alignment of weights differs depending upon the output location.

weight sub-matrix. Assuming aligned input buffer accesses, multiple permutations of the weight matrix are required to process the full set of inputs.

Considering a simple case where the number of input channels is half the width of the array and the number of output channels is equal to the height of the array, it is easy to see that peak array utilization is achieved when the set of weights loaded to the array at a given time represents two input positions and one output position. Assuming that the input rows are retrieved by accesses aligned to the array dimensions, each input address pertains to a full set of input channels for a pair of adjacent x-coordinates (one even and one odd) and a single y-coordinate. Accounting for all output positions having synaptic connections to the same pair of input x-coordinates requires two column-wise permutations of the same weight matrix, one even and one odd, as is depicted in the examples in Figure 3.3. The output neurons are divided into subsets each pertaining to a different column-wise permutation of the weight sub-matrices,

$$\underset{N\times\frac{XY}{2}}{O^{h}} = \sum_{k_x\in h}\sum_{k_y} \underset{N\times 2M}{W^{h}_{k_x,k_y}} \times \underset{2M\times\frac{XY}{2}}{I} \quad \forall h \in \{\texttt{even}, \texttt{odd}\}, \tag{3.5}$$

where O^{even} and O^{odd} represent two separate subsets of the output neurons. When more x-coordinates are contained per input row, the weight matrix requires as many column-wise permutations as there are input positions considered in a single row. We only consider input rows with multiple coordinates along one dimension of the feature map. This is because packing both multiple x and y coordinates into a single row can require multiple buffer accesses, since the input buffer is addressed only along one dimension of a typical SRAM.

Thus far, we have only considered weight matrix construction for layers with sufficiently many output channels to fill the height of the processing array. Therefore all permutations of the weight matrix are simply column-wise rotations of the same set of columns. Assuming column-wise addressing of the weight memory, the logic required to create these permutations within the array is quite simple, requiring mechanisms to control placement of columns within the array and zeroing beyond kernel edges. Importantly, there is no need to store multiple permutations in weight memory since these permutations can be easily constructed at run-time, as is shown in Figure 3.4. Alternative implementations may include column-wise permutation logic at the activation input of the array; however this is not required for a functionally correct accelerator.

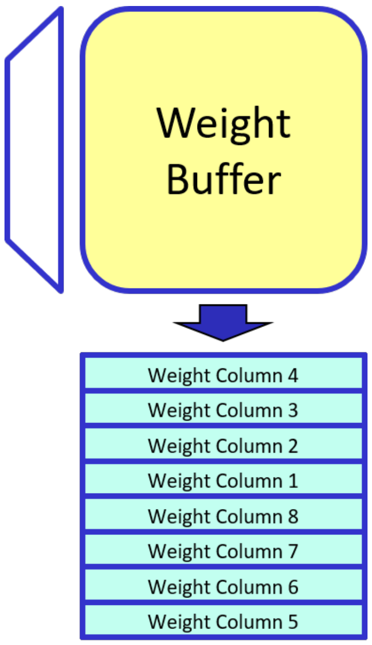


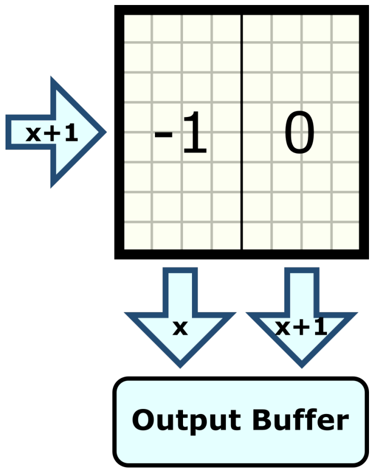
Figure 3.4: Permutation by Column-wise construction.

Row-wise Permutation

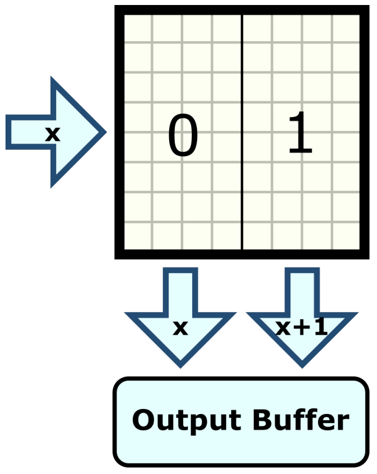
Although most convolutional layers contain fairly large sets of output channels, if the height of the processing array is larger than the number of output channels in the layer then weight sub-matrices that include multiple output locations within each column are appropriate and multiple row-wise permutations of the weight sub-matrices are required to align the kernel matrices to the output buffer. The input rows are divided into subsets pertaining to each row-wise permutation of the weight sub-matrices and accumulated as

$$\underset{2N\times\frac{XY}{2}}{O} = \sum_{\nu\in\{\texttt{even,odd}\}} \sum_{k_y} \sum_{k_x\in\nu} \underset{2N\times M}{W^{\nu}_{k_x,k_y}} \times \underset{M\times\frac{XY}{2}}{I^{\nu}}, \tag{3.6}$$

where each output neuron must accumulate synapses from both row-wise permutation sets of the input activations, as is depicted in Figure 3.5.



(a) Odd Permutation



(b) Even Permutation

Figure 3.5: Row-wise Permutation Examples with input activations pertaining to a single location within the feature map arriving from the left and outputting to two consecutive output locations, where row-wise alignment of weights differs depending upon the input location.

Support for these cases can be accomplished either by storing permuted copies of the weight sub-matrices in memory or using mechanisms for row-wise permutation of weight columns from the weight buffer, as is shown in Figure 3.6. Row-wise permutations are only necessary when the array dimensions exceed the number of output channels by a factor of two or more. For processing arrays of size 64 $\times$ 64 or smaller, very few layers require row-wise permutations. But as the size of the array increases, more layers require multiple permutations of the weights to fully utilize the array width. The layers with few enough channels to require permutations tend to have outputs with large spatial dimensions, so a few layers requiring permutations can represent a significant portion of memory traffic. Mechanisms for constructing permutations instead of retrieving them from memory are useful for alleviating the memory access overhead in these cases.

Similar to the multi-positional input rows, multi-positional output rows should only be drawn from one of either the x or y spatial dimension, due to one dimensional SRAM

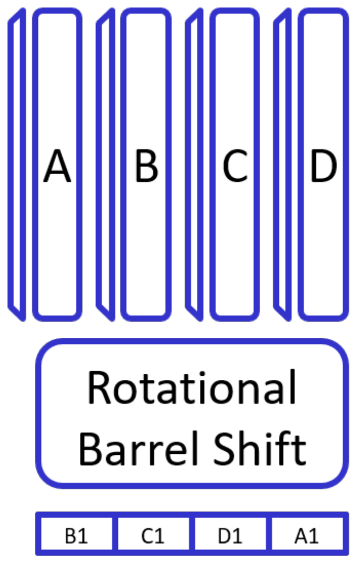


Figure 3.6: Row-wise Permutation by Rotated Column-wise construction.

access semantics which prevent combining multiples of both x and y dimensions within a single row. Furthermore, the row composition guidelines must be identical for both the input and output since the output of one layer becomes the input of the next and the row format should not need to change between layers. The exception is the input to the very first layer of the CNN, which never needs to reside in the output buffer.

Combined Permutation

The combination of the row-wise and column-wise sub-matrix permutation capabilities outlined above is sufficient to support layers where both input rows and output columns contain multiple positions. We can extrapolate the form of the sums based on combinations of the example equations 3.5 and 3.6 above to be

$$\underset{L_n N \times \frac{XY}{L_n|\mathcal{H}|}}{O^h} = \sum_{v \in \mathcal{V}} \sum_{k_x \in hv} \sum_{k_y} \underset{L_n N \times L_i M}{W^{h,v}_{k_x,k_y}} \times \underset{L_i M \times \frac{XY}{L_i|\mathcal{V}|}}{I^v} \quad \forall h \in \mathcal{H}, \tag{3.7}$$

where $\mathcal{H}$ is the set of column-wise permutations and $\mathcal{V}$ is the set of row-wise permutations. Each *W* submatrix takes a block-Toeplitz form where each block has dimensions of $N \times M$.

Assuming L_n locations per output row and L_i locations per input row, the dimensions of each weight sub-matrix is $L_n N \times L_i M$. The number of different permutations of the weight matrix is the product of the number of column-wise and row-wise permutations ($P = |\mathcal{H}||\mathcal{V}|$). If L_i and L_n are co-prime then $|\mathcal{H}| = L_i$ and $|\mathcal{V}| = L_n$. However, assuming consistent buffer alignment with respect to multi-location buffer rows, the more general expressions for the number of permutations is expressed in terms of the greatest common denominator (GCD) of L_i and L_n as $|\mathcal{H}| = \frac{L_i}{\texttt{GCD}(L_i, L_n)}$ and $|\mathcal{V}| = \frac{L_n}{\texttt{GCD}(L_i, L_n)}$, and the general expression for the number of unique permutations of weight matrices is $P = \frac{\texttt{LCM}(L_i, L_n)}{\texttt{GCD}(L_i, L_n)}$. Therefore, layers with the same number of input and output features do not require multiple permutations. Layers with feature counts that are powers of two are easily mapped to arrays with dimensions that are powers of two and therefore achieve higher temporal locality for weights loaded into the array. For these layers with power of two feature counts where the number of output features is a multiple of the number of input features, row-wise permutations are never required.

3.3.3 Sequencing

The number of times that each weight sub-matrix is loaded into the processing array can be minimized by identifying sets of outputs that share the same set of weight sub-matrices and processing those neurons as a single unit. When only one input position and one output position are addressed per sub-matrix, this is easily accomplished since all neurons sharing the same output channel share the same set of sub-matrices. When multiple input locations are addressed by the same sub-matrix, a striding pattern with step size equal to the number of column-wise permutations identifies the set of output positions that share the same set of weight sub-matrices. Similarly, when multiple output locations are addressed by the same sub-matrix, all neurons of a given output channel share the same set of submatrices, however a striding pattern is necessary while accessing the input rows. These effects are independent, so when both the inputs and outputs contain multiple positions per row the superposition of the striding patterns

```
// $N_n$: number out channels
// $N_i$: number in channels
// $A$ : buffer row size
// $H$ : Column-wise permutations
// $V$ : Row-wise permutations

for($n = 0$; $n < N_n$; n+=A)
  for $P_{out} \in \{1, ..., H\}$
    for $P_{in} \in \{1, ..., V\}$
      for $W_{sub} \in P_W(P_{in}, P_{out})$
        for($i = 0$; $i < N_i$; i+=A)
          $O(n, P_{out})$ += $W_{sub}(n, i) \times I(i, P_{in})$
```

Figure 3.7: Convolutional Layer `For` loop structure of neuron accumulation based on permuted weight sub-matrices.

can be applied to identify the input sets and output sets pertaining to the same set of weight sub-matrices.

Output neurons are grouped into weight locality sets defined by their associated input and output permutations where all neurons in a set use the same weight sub-matrix permutations, as is shown in Figure 3.7, where the inner loop pertains to accumulation of all rows within the output buffer belonging to the same locality set. The listing does not include output buffer allocation, which simply results in additional outer loops to partition the computation across multiple output buffer allocations where each neuron is allocated within the output buffer once and fully accumulates before proceeding to a new allocation. In order to address all synapses for the neurons contained in the output buffer, an outer loop of the computation schedule must be based on the output buffer contents. If there are more output channels than the output buffer row size, then only one row's worth of output channels should be contained within the output buffer at a time, since different output channel sets use different sets of weights. If an output striding pattern is required, the contents of the output buffer should give priority to neurons within the same locality set. If the output buffer has sufficiently many rows to accommodate multiple locality sets, then it is worthwhile to maintain sets from the same output channel set requiring different column-wise permutations of the weight matrices, in order to reuse the matrices in the weight buffer and minimize off chip memory accesses.

3.3.4 Implementation

Column-wise permutations are easily constructed based on the sequence of weight buffer accesses where weight buffer addresses must be calculated based on an offset corresponding to the column-wise permutation amount. The sequence is controlled using automated address iteration logic which associates a base address, permutation number, location offset and the layer dimensions to generate the sequence of weight addresses for each weight sub-matrix.

Row-wise permutations of the weight sub-matrices cannot be constructed based solely on aligned accesses to the weight buffer and therefore require additional architectural support to construct the permuted matrix column. The row-wise permutation is created by rotation of each column read from the weight buffer. The rotation is performed by a circular shift network between the weight buffer read port and the array's weight input, as is shown in Figure 3.2. The circular shift network is implemented as a crossbar barrel shifter with minimal wiring overhead in comparison to the size of the buffers and the size of the array.

The primary motivation for this implementation of row-wise permutation is to

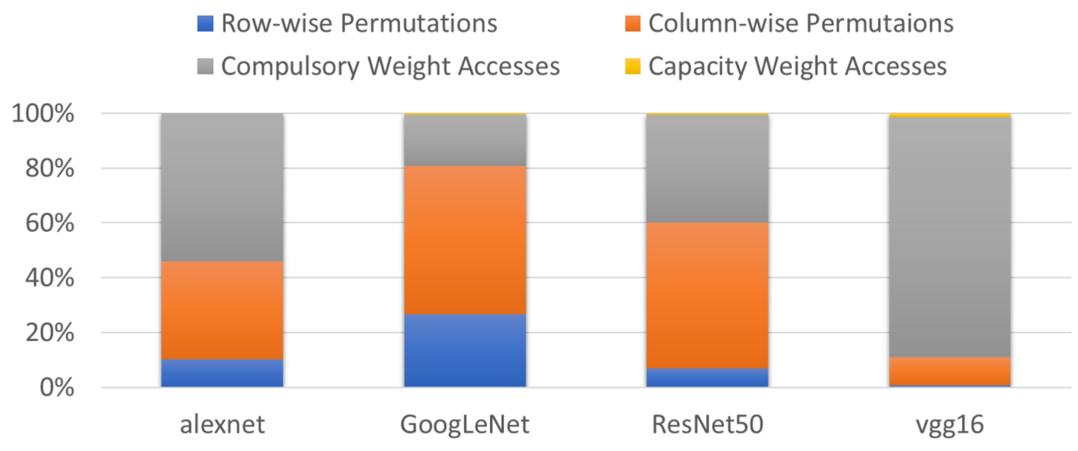


Figure 3.8: Weight memory access breakdown based on TPU output buffer size (4096 entries) and array size (256x256).

prevent impeding the weight memory bandwidth. One alternative approach would store permutations in memory and load them into the weight buffer as separate sub-matrices, but this requires additional data to transfer from the weight memory, which is an inefficient use of the weight bandwidth. Another alternative approach for row-wise permutations would be to store multiple copies of the same weights within the buffer rows requiring permutation such that permutations are easily constructed using aligned weight buffer accesses. Although this approach does not require additional data transfers from memory, it does increase the number of writes to the weight buffer and therefore would impede the buffer's write bandwidth. Furthermore, storing multiple copies reduces the effective capacity of the buffer.

The data in Figure 3.8 demonstrates the proportion of weight memory accesses that can be satisfied by permutations constructed from contents of the weight buffer. The data is collected based on an accelerator with TPU based parameters, so it is interesting to note that nearly all weight accesses for these networks can be satisfied either by permutation

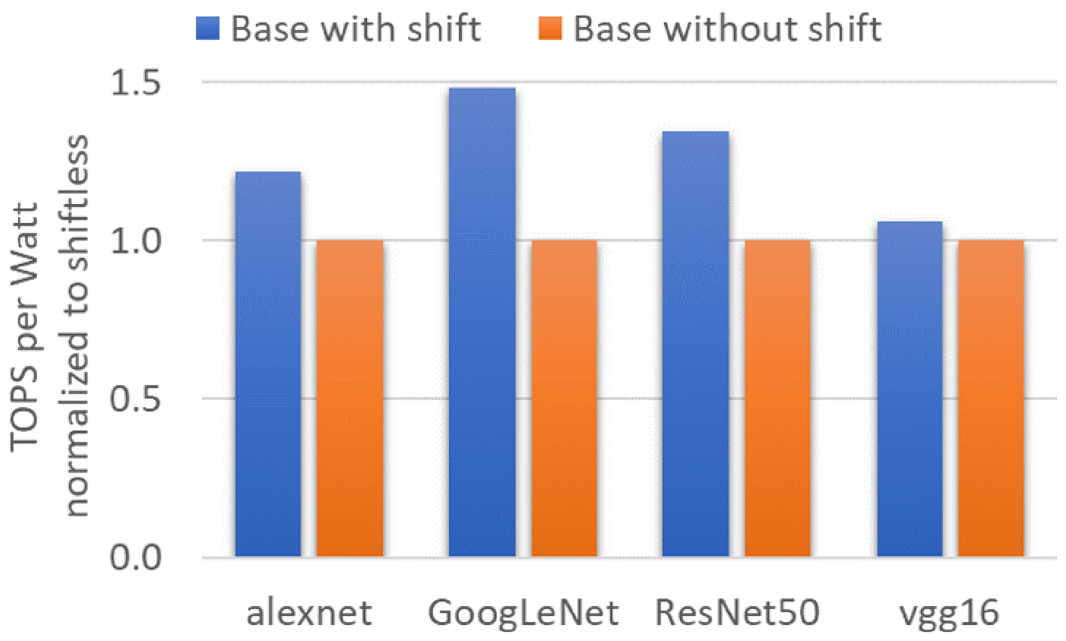
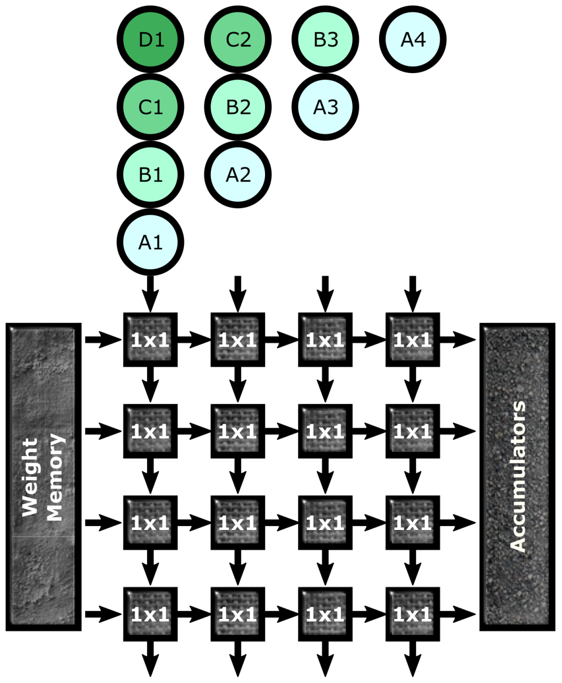


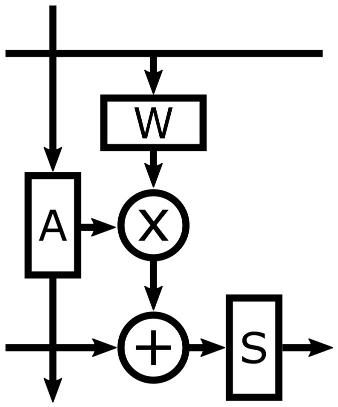
Figure 3.9: Energy efficiency of row-wise permutation support.

construction or by compulsory memory accesses. One thing that can be observed from this data is that the computation sequence is exploiting permutation locality well. The large portion of the weight accesses that are considered permutations demonstrates the importance of the permutation support. Furthermore, since the experiment is based on the TPU parameters, this suggests that there is indeed some permutation support within the TPU, but whether it is similar to the support described above is yet to be determined.

The energy efficiency improvement resulting from exploitation of weight buffer locality for the same 256x256 systolic array based accelerator operating at a clock frequency of 667MHz with one LPDDR4 module storing 8-bit weights for batch size one inference using each network is shown in Figure 3.9. A 50% energy efficiency improvement is observed for GoogLeNet, which requires the most permutations of the networks consid-



(a) Diagonal Wave-front Incident upon Systolic Array input with A, B, C and D representing different columns of the input matrix.



(b) Multiply Accumulate Cell of Systolic Array

Figure 3.10: Baseline Systolic Array Diagrams.

ered. This data was collected using the simulation methodology described in Section 3.5. Permutation support is assumed to be present for all experiments in other sections. The following section describes the diastolic array based accelerator optimizations.

3.4 Diastolic Accelerator

In order to explore the impact of pipeline depth reductions, expanded pipeline stages along each of the data-flows in the systolic array are considered. The flow of input activation data into the baseline systolic array follows a diagonal wave-front [56], shown in Figure 3.10a, where inputs are propagated downward. Weight sub-matrices are statically mapped directly to the cells of the systolic array during computation of each component matrix operation and partial dot-products are accumulated through horizontal propagation along each row of the array. The partial sums produced by each row of the array are combined with previously accumulated partial dot-products within the accumulators at the output of the array. The baseline systolic array cell contains a single 8-bit fixed point scalar multiplier for synapse calculations and a single 16-bit fixed-point scalar adder for accumulation with partial sums propagated from adjacent cells. A schematic of the matrix-multiplication systolic array cell is shown in Figure 3.10b, which includes a weight register, an input activation register, and an accumulated sum register, representing the three data flows of the array. Pipeline depth adjustments for each of these three dataflows are considered within this section.

3.4.1 Weights

Assuming an array with sufficient storage within each cell for weights corresponding to multiple component matrix-multiply operations to be fetched prior to calculations, the latency of loading weights can be mostly overlapped with the latency of calculation. In cases where the weight loading latency is greater than the processing latency, the weight bandwidth imposes a lower bound on array latency. In order to simultaneously process synapses and load a new weight matrix, each column of the array contains two weight registers, one for the active calculation and one for the active transfer. Weights propagate through the array along segments spanning multiple cells which are used as a shared input to weight registers of a set of cells along the segment. The sequence starts with the matrix rows destined for the furthest array cells such that the longer latency of traversing multiple hops through the array does not affect the overall latency of loading the weight matrix. Once weights have propagated to all shift registers, weights are loaded from to a set of weight registers from the shift registers for all segments in unison, where they remain for the duration of a given matrix-multiply operation. An implementation of the weight distribution bus segment is given in Figure 3.11, where a bus is shared across a subset of weight registers. Each row of the array is composed of a sequence of these weight distribution bus segments. This implementation of weight distribution is more efficient than purely systolic implementations of pipelined data propagation, as is demonstrated experimentally in Figure 3.12a for two implementations of the weight register, one of which is flip-flop based and the other latch based. The latch based design is more efficient, most likely because it directly integrates clock gating logic more effectively than flip-flops with multiplexer based enable ports. The clock-gating that is applied during these experiments is based on the Synopsys automated clock gating compilation flag.

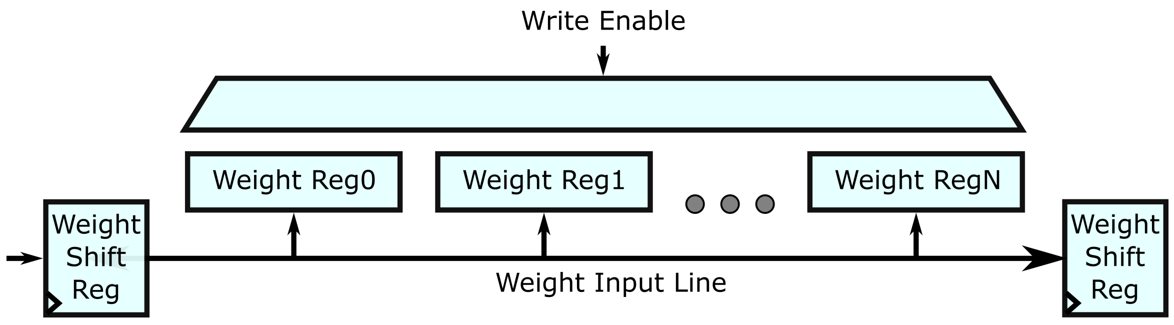
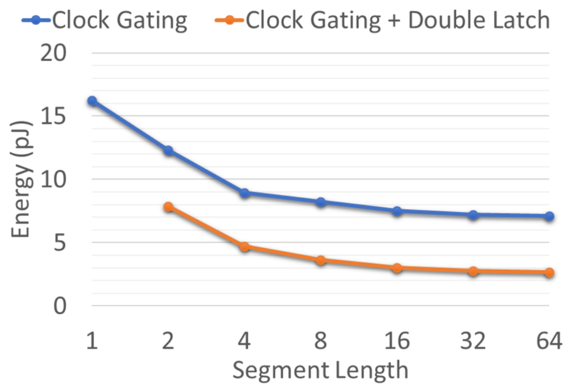
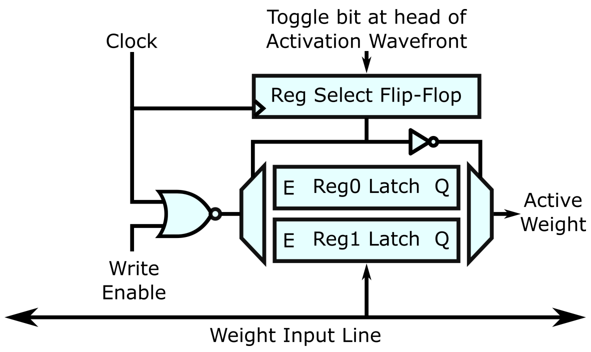


Figure 3.11: Weight Distribution Bus Segment



(a) Energy per 8-bit weight loaded to a 256x256 array lane, by segment length.

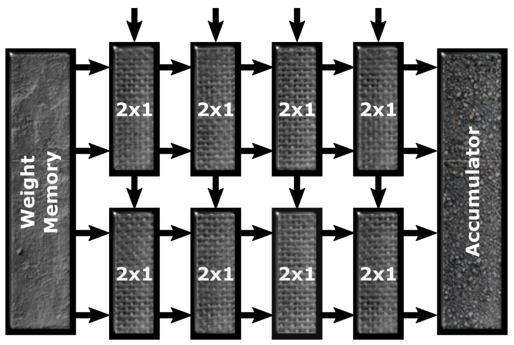


(b) Latch based implementation of weight register.

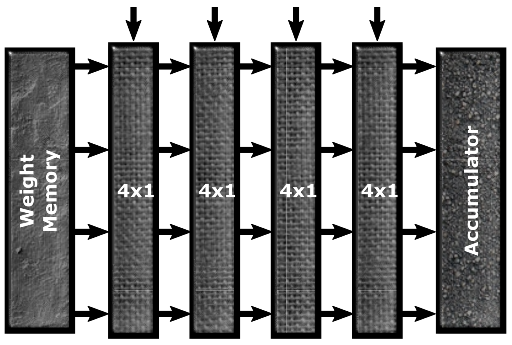
Figure 3.12: Comparison of latch based implementation to typical clock-gated flip-flop implementation of dual weight registers at various weight distribution bus segment lengths.

3.4.2 Activations

Expansion of the systolic array cells along the direction of input propagation, as is illustrated in Figure 3.13, involves elimination of intermediate activation registers such that each input activation is shared by multiple lanes of inner product calculation at a time. This reduces the number of registers required for input propagation while also



(a) Cell Height of 2

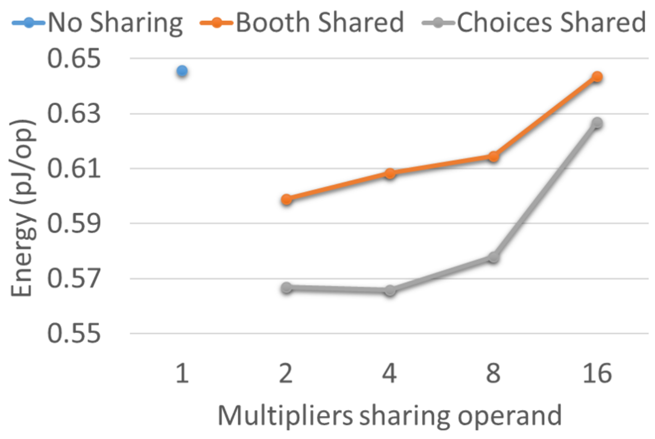


(b) Cell Height of 4

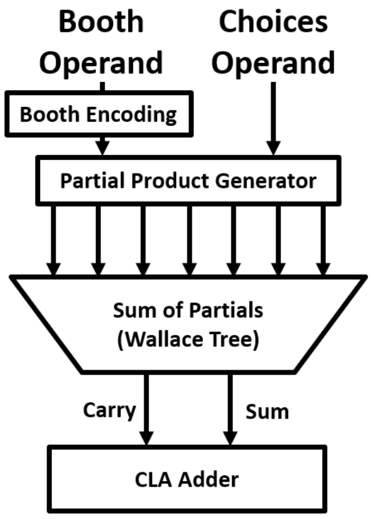
Figure 3.13: Cell height in diastolic arrays.

increasing the fan-out of each activation register. This has two conflicting effects on energy where although the number of flip-flops decreases, the per flip-flop energy is increased due to an increased fan-out that can require the synthesis tool to increase the size of the register's flip-flops in order to meet timing constraints through the functional units. Thus the primary benefit of the expansion of activation propagation to multiple dot-product lanes is the latency reduction that it provides, which is valuable if array latency is critical. The extent to which input activation registers can feasibly be merged is bounded by the cell's timing constraints. We will refer to cell size along the direction of input propagation as cell *height*.

The activation register provides one of the input operands to the fixed-point multiplier. The operand inputs to a fixed-point multiplier are asymmetric, so the choice of



(a) Energy per multiply when sharing either the booth encoded(A) or multiplexed(B) input across variable number of multipliers.



(b) Modified Booth Fixed-Point Multiplier.

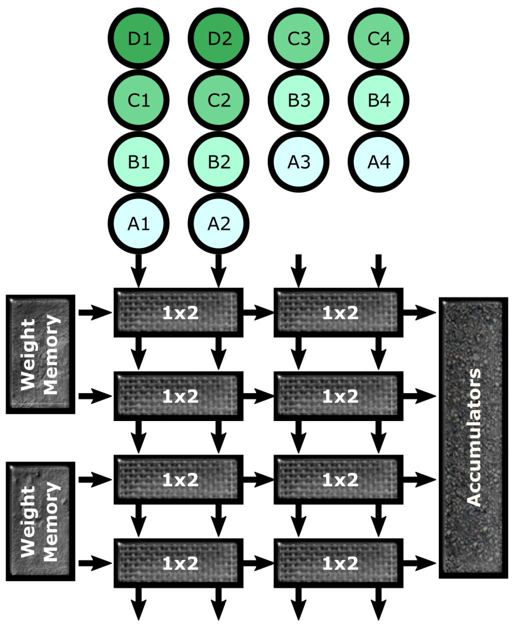
Figure 3.14: Energy per 8-bit multiplication sharing different multiplier input across variable number of booth radix-2 multipliers. Minimum is observed sharing the multiplexed input across 2-4 multipliers.

which multiplier input is used for the shared activation can affect the efficiency of multiplication. Figure 3.14 demonstrates that sharing of the multiplexed input of the radix-2 modified booth encoded multiplier is often more efficient than sharing the activation across the booth encoded input. This observation is likely due to a shorter timing path through the multiplexed input. Although the multiplier in the context of the accelerator is synthesized based on a more generic specification than in these experiments, the analysis in Figure 3.14 demonstrates the multiplier asymmetry considerations that must be considered by the synthesis tool. Based on these results, it is reasonable to expect than multipliers synthesized with shared operands would similarly select the multiplexed input as the shared input, and custom implementations should likely do the same.

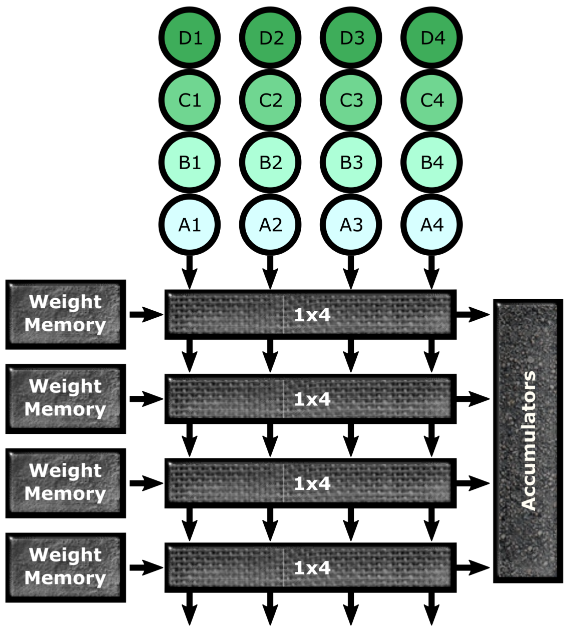
3.4.3 Summation

Expansion of the systolic array cells along the direction of output accumulation involves integration of more complex inner product units within each cell, with adder-tree based reductions. The cell size along the direction of accumulation will be referred to as the cell *width*. Adder trees provide an implementation of spatial reduction with single cycle semantics in which reductions are performed on all data incident upon the tree inputs within a given cycle. The data flows used for wide diastolic arrays must therefore follow a piece-wise diagonal wave-front, as is shown in Figure 3.15, where all inputs to a given cell are delivered within the same cycle with data from the same row appearing at the adjacent cell within the following clock cycle. The latency of the adder tree's calculation is dependent upon the pipeline depth of the adder tree itself, which is dependent upon the selected clock frequency and has a logarithmic relationship to the input count.

Due to the greater complexity of wiring within the adder tree structure in comparison to the systolic array, it is conceivable that the adder tree could introduce some additional wire energy overhead, however this would appear to not be the case based on the place and route results, shown in in Figure 3.16. This data demonstrates the energy savings associated with using adder trees for accumulation in place of systolic accumulation. The energy savings is attributed to elimination of flip-flops by using adder trees without internal flip-flops for dot-product reductions.



(a) Cell Width of 2



(b) Cell Width of 4

Figure 3.15: Cell width in diastolic arrays.

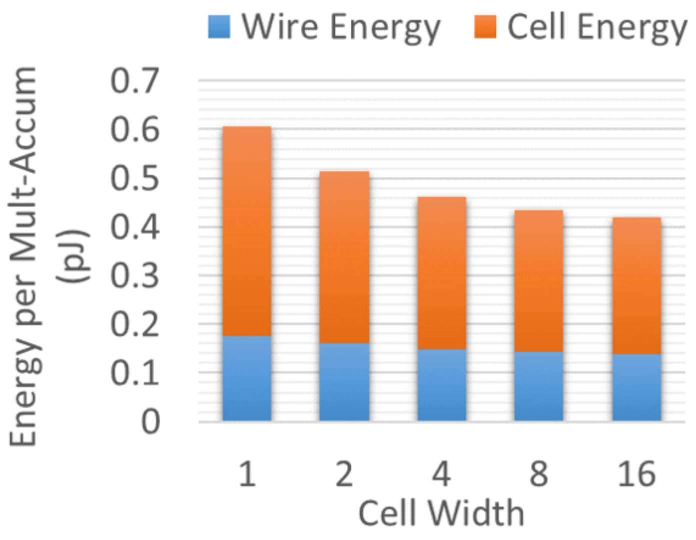


Figure 3.16: Energy per 8-bit multiply-accumulate within array cells of varied width, without internal pipelining, at a clock frequency of 667MHz.

3.4.4 Section Summary

In this section we considered modes of expansion within each of the three dataflows of the systolic array. For weight loading, bus segment based weight propagation is considered in comparison to typical systolic propagation. For activation propagation, the impact of sharing across multipliers was demonstrated to have diminishing returns due to a long critical path with lower power requirements when the multiplexed input of the radix-2 modified booth encoded multiplier is shared as opposed to the booth input. For summation, adder tree based dot-product reduction is shown to be more efficient due to lower pipelining requirement than systolic reduction at 667MHz, with the added benefit of using step-wise wave-fronts with lower latencies. In the following sections full chip experiments are conducted for various accelerators with these optimizations.

3.5 Experimental Methodology

Experiments are designed to consider the performance and power characteristics of a systolic array processing a set of neural networks drawn from ImageNet winners. The set of convolutional neural network benchmarks considered are given in Table 3.1, with the total number of neurons and synapses for each network listed. The last column in the table is the ratio of Fully connected synapses to convolutional synapses in each network, included to emphasize the relatively high number of fully-connected synapses contained in AlexNet. The calculations associated with each neural network layer are subdivided into component matrix-multiplications, each sized to fit the dimensions of the processing array. The performance model uses the guidelines outlined in Section 3.3.3 to determine the sequence of component matrix-multiplications in order to exploit permutation locality. In a diastolic array, the latency of processing each of the component matrix-multiply operations is a function of the cell height and width. The latency of loading the weight matrix for each component operation is unaffected by diastolic expansions. Weights are loaded one column at a time so the latency of loading each weight sub-matrix is proportional to the number of columns. With a capacity of weight set within

Table 3.1: CNN Topologies.

	Neurons	Synapses	
Network	Total	Total	FC:Conv
Alexnet [77]	0.7×10^6	0.7×10^9	0.09
GoogLeNet [130]	3×10^6	1.6×10^9	0.0007
Resnet50 [55]	10.6×10^6	3.9×10^9	0.0005
VGG16 [124]	13.6×10^6	15.4×10^9	0.008

the array at a time, the array weight loading would need to wait the full duration of array computation latency before loading a new set of weights. So the array is assumed to have sufficient capacity for two weight sub-matrices such that one stationary set of weights is held while a new set of weights loads. With capacity for two weight sets, the computation latency is only critical when it exceeds the latency of loading two sets of weights, since we assume that the first new set of weights can be used immediately by a new input wave front once it is loaded.

Considered individually, the processing latency for each calculation is given in Eq. 3.8 as

$$t_{op} = t_{prop} + t_{cell} + t_{acc} + t_{cols} + t_{weights}, \tag{3.8}$$

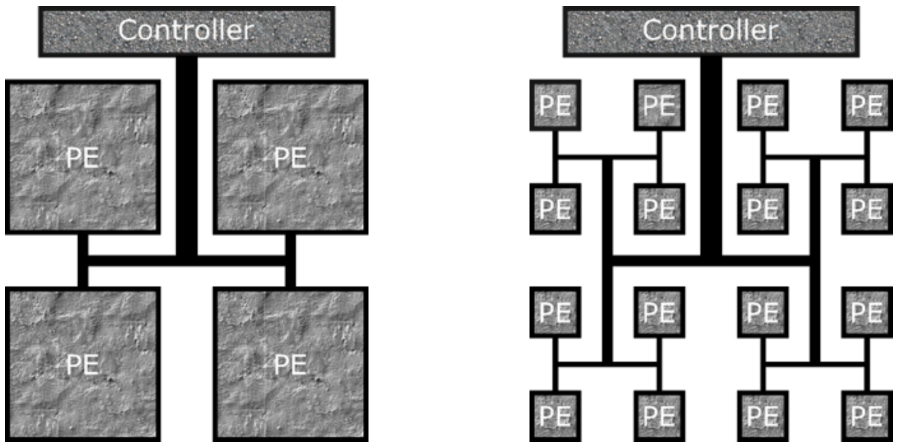
where t_{prop} is the number of cycles required for an input to propagate through the array and t_{acc} is number of cycles involved in accumulation of each partial sum. t_{cell} is the latency of processing within each cell, which is a function of the clock frequency and the cell width since it depends on the pipeline depth of the cell. t_{cols} represents the latency of each additional input matrix column after the first and essentially just adds an additional cycle for each additional column of the input matrix. $t_{weights}$ is the overhead of loading the array with a new weight matrix, which is dependent on the bandwidth of the weight loading data path including both the weight buffer access bandwidth and the weight memory bandwidth.

Many of these latency components are concurrent. We assume ideal prefetching of weights, such that a new set of weights is loaded into the weight buffer as soon as there is sufficient capacity for the new set. *Memory Stalls* only occur when weights cannot be loaded into the array because they have not yet been retrieved from memory. Once a weight sub-matrix is loaded into the weight buffer, many permutations of the sub-matrix can be loaded into the array. In order to load a new sub-matrix permutation into the array, the computation involving the previously loaded submatrix to be replaced must have completed. *Compute Stalls* occur when all sub-matrices are involved in active computations, causing loading of the next weight set to stall until the full $t_{prop} + t_{cell} + t_{acc}$ computation latency of the older input set completes. Once a weight sub-matrix is fully loaded into the array, the corresponding input sub-matrix can begin to propagate as a diagonal wave-front. The *Input Loading* latency is similar to the t_{col} of Eq. 3.8. For computations involving input matrices with fewer columns than there are weight columns, the *Weight Loading* latency causes a delay between successive input sets.

The performance model also supports configuration with tiled architectures where each tile is a separate array and all tiles share the same memory bandwidth and a common neuron memory where activations are written to and read from through a high bandwidth tree based interconnect, like DaDianNao's neuron memory [16]. A tiled design partitions a set of multiply-accumulate units evenly into separate systolic arrays such that the total compute bandwidth is the same as a monolithic configuration with the same number of multiply-accumulate resources. The smaller arrays have lower weight loading and compute latencies, which can improve performance. However, the on chip bandwidth for inputs and weights required to keep the arrays active is greater for a tiled architecture. The performance model assumes sufficient on-chip bandwidth to support all tiled arrays and evaluates the energy impact based on memory accesses and interconnect usage. The interconnect model is based on H-tree distance of traversal to the array edge, as is shown in Figure 3.17.

The mapping of layers to arrays partitions by both output channels and spatial dimensions. A lower bound on channel partitioning is imposed by the array output width to avoid row-wise permutations. A lower bound on spatial partitioning is imposed by the number of weight columns in order to avoid stalling inputs due to weight loading latency.

The power model is primarily based on results collected after place-and-route of the processing elements using Synopsys tools and a 45nm TSMC technology library. Designs for three different clock frequencies are synthesized with each cell's pipeline depth set to meet the associated timing constraints. The pipeline depth sometimes varies depending upon the width of each cell since as the depth of the adder tree increases, deeper pipelines may be required for timing closure. The pipeline depths of



(a) 2x2 Arrays of 128x128 MAC units each.

(b) 4x4 Arrays of 64x64 MAC units each.

Figure 3.17: Examples of Tiled Architectures.

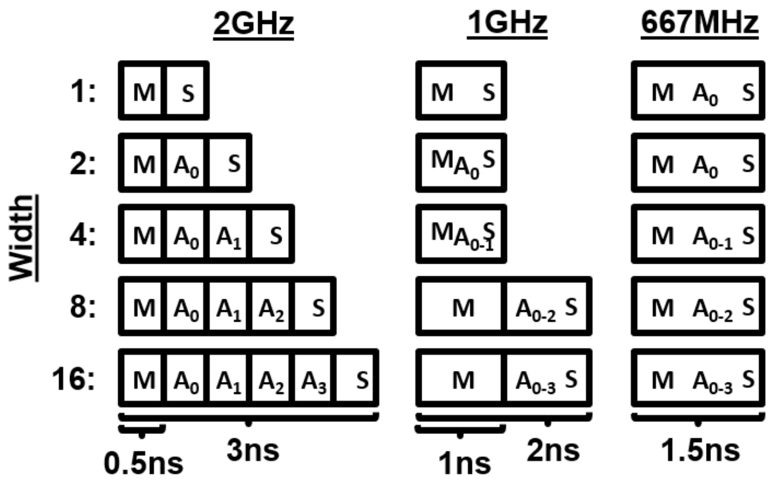
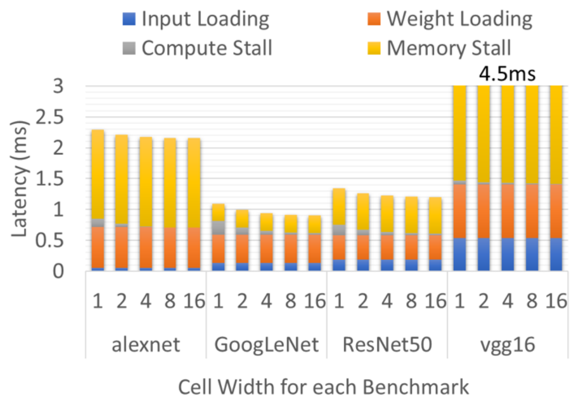
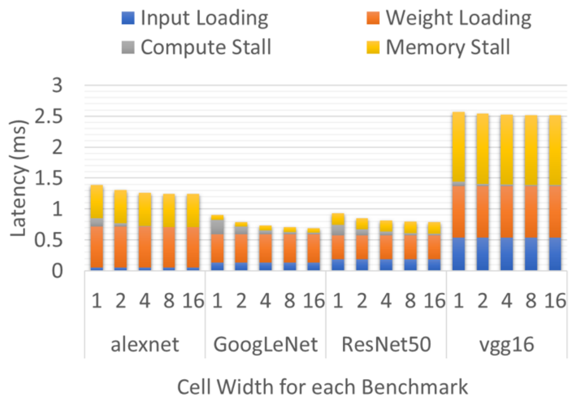


Figure 3.18: Multiply-accumulate unit pipeline depths at different clock frequencies and cell widths.

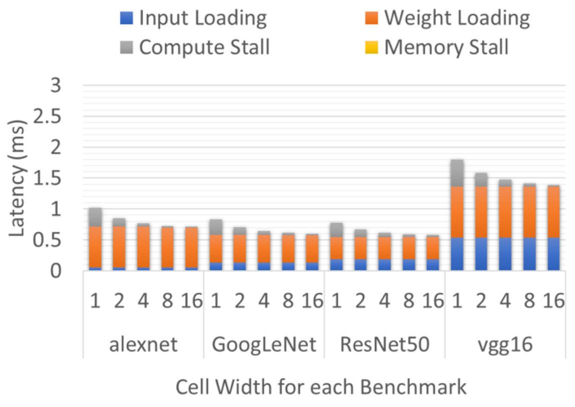
the partial dot-product multiply accumulate pipelines under consideration are shown in Figure 3.18. These pipeline depths are the basis for the t_{cell} latency of Eq. 3.8. All designs considered within these experiments use 8-bit operands for weights and activations, with 16-bit sums accumulated from one cell to the next, and 32-bit sums accumulated within the output buffer. Some additional power model constants are derived from cacti used to model SRAMs. DRAM energy models are based on power numbers reported within recent DRAM research literature [85, 107] or otherwise publicly available. Most of the experiments assume an 8GB LPDDR4-4266 mobile DRAM package [119] with a bandwidth of 34GB/s. Scaling to multiple LPDDR4 packages enables incremental analysis of the impact of memory bandwidth. We also consider HBM2 which is not as efficient as LPDDR4 but achieves much higher bandwidth of 256GB/s per package.



(a) LPDDR4



(b) 2xLPDDR4



(c) HBM2

Figure 3.19: Cell Width effect on batch size 1 inference latency when applied in conjunction with different memory technologies.

3.6 Results

The results include performance and power characterizations of various cell widths and heights within the context of an accelerator with a monolithic 256x256 processing array with SRAM buffers output, input, and weights with capacities of 4MB, 28MB, and 256KB respectively. A sensitivity study is conducted which varies the array dimensions with weight and output buffer capacities scaled in proportion to the array weight capacity and output width of the array, such that the number of output entries and the number of arrays worth of weights remains constant. We also compare monolithic systolic arrays to tiled architectures where each tile is itself a systolic array and multiple tiles share a common memory controller. Total compute capacity and output buffer capacity remains constant for designs in the tiling study.

3.6.1 Monolithic Performance

Width adjustments affect both the latency and energy of calculations, as is demonstrated for each benchmark in Figure 3.19 for a configurations with a monolithic 256x256 processing array with cell height fixed at one, operating at a 667MHz clock frequency. The three graphs represent a comparison of latency results between systems with different memory packages. The system with HBM2 has sufficient bandwidth to completely avoid memory stalls, so the reduction of compute stalls with increasing cell width can clearly be seen in Figure 3.19c. When the latency of propagation through the array is less than twice the latency required to load a new set of weights from the weight buffer, the compute stalls become mostly negligible.

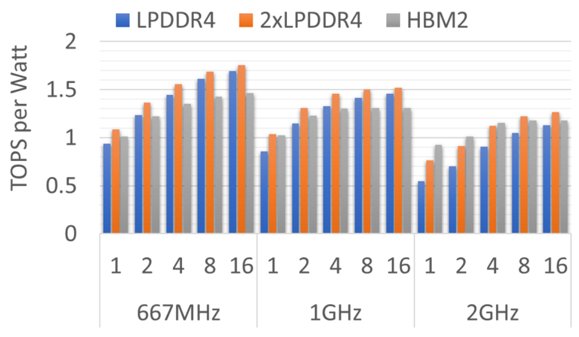
The LPDDR4 bandwidth is low enough that a significant portion of the latency is attributed to stalling while weights are loaded into the weight buffer for some of the benchmarks. Weight stalls can sometimes overlap with the compute stall interval and overshadow the latency reductions of cell widening. The memory stalls affect AlexNet and VGG16 more severely than GoogLeNet and ResNet50. Both GoogLeNet and ResNet50 often require multiple permutations, which improves their tolerance of lower memory bandwidths.

The weight loading and input loading latencies are unaffected by adjustments to cell width and memory technology because they are primarily dependent upon the bandwidths of the array itself, which remains constant when cells are expanded. Based on the comparatively large weight loading latencies in Figure 3.19c, increasing the weight loading bandwidth can improve performance contingent upon sufficient memory bandwidth and further reducing the processing latency.

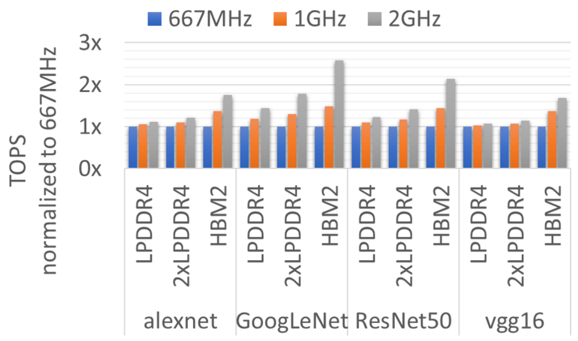
3.6.2 Monolithic Power

The power efficiency of expanded cells is related to both the latency reductions that help to eliminate stalling conditions and the elimination of flip-flops from adder trees with relaxed timing constraints. The results in Figure 3.20a clearly demonstrate that wider cells are more efficient at each clock frequency with each memory configuration. At higher frequencies, memory stalls become dominant and the power efficiency is degraded, even for the system with HBM2 memory. Power efficiency improves as more memory packages are included since the memory bandwidth increases with roughly the same per access energy costs. All further experiments are based on an array with cell width 16 and a cell height 2, which provides sufficient latency reduction to avoid computation stalls given the weight loading bandwidth.

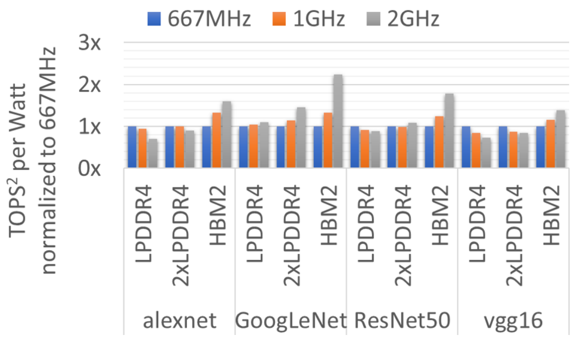
The overall inference performance associated with clock frequency adjustments is highly dependent upon memory bandwidth. Figure 3.20b shows the tera-ops per second (TOPS) for each memory system and each frequency, normalized to a 667MHz



(a) TOPS per Watt for array with cell height 1 averaged across benchmarks for five cell widths.



(b) TOPS for array with cell height 2 and width 16 for each benchmark.



(c) TOPS^2 per Watt for array with cell height 2 and width 16 for each benchmark.

Figure 3.20: Batch size 1 inference on a 256x256 array at three clock frequencies with different memory system configurations.

clock frequency in order to easily capture data across the set of benchmarks within the same graph. The benchmarks that were previously identified to not be tolerant of low memory bandwidths do not scale well to higher clock frequencies without high memory bandwidth.

In order to combine the observations regarding power efficiency and performance, Figure 3.20c considers the TOPS^2 per Watt, which is an energy-delay metric [52, 127]. Increasing the clock frequency is observed to often represent degraded efficiency by this metric for lower bandwidth memory systems but an improvement in efficiency for higher bandwidth memory systems. If high performance must be achieved within a

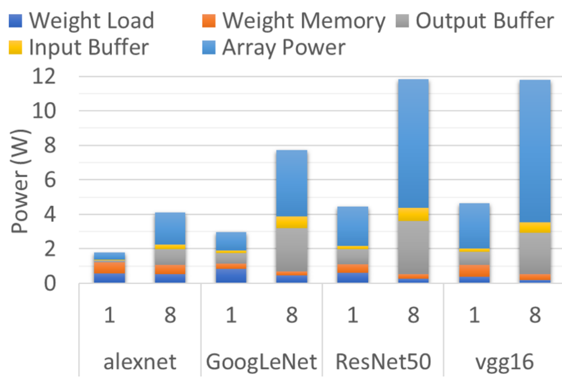


Figure 3.21: Power breakdown for inference with batch sizes of 1 and 8 on a monolithic 256x256 array.

single package then maximizing energy-delay metrics under the peak power constraint is worthwhile. If scaling out to multiple accelerators operating in parallel to achieve high performance is preferred, then maximizing power efficiency of each accelerator may be preferable for scalability to an efficient system.

The next set of results uses the design point that demonstrated the highest average power efficiency in Figure 3.20a, which supports two LPDDR4-4266 packages at an accelerator clock frequency of 667MHz. Figure 3.21 shows the power components of each benchmark running on this design for batch size one and batch size eight inference. In general, the weight related power components become less significant with larger batch sizes and all other components become more significant. Reaching the peak power consumption requires high weight locality to achieve high utilization of the available compute resources. Increasing batch size improves exactly that. ResNet50 and VGG16 both consume more power and are more efficient than the other two benchmarks because of their high array utilization.

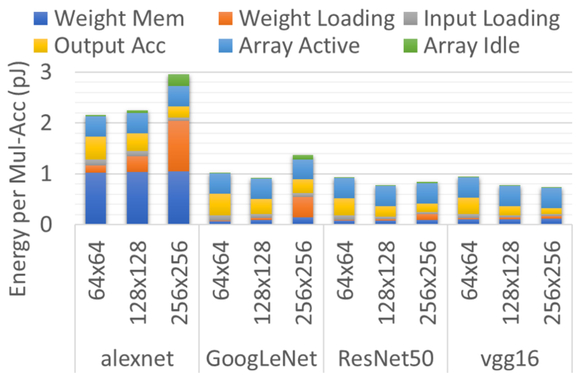
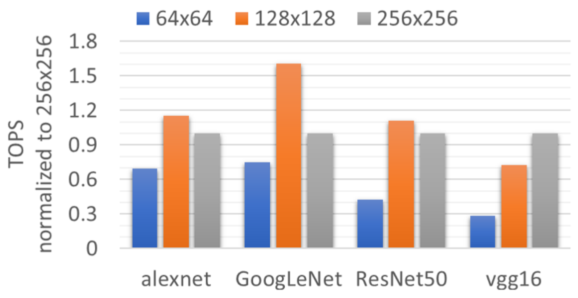


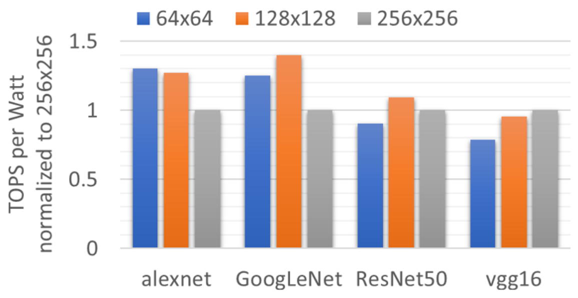
Figure 3.22: Energy per multiply-accumulate breakdown for batch size one inference with varied monolithic array sizes.

3.6.3 Array Size Sensitivity Study

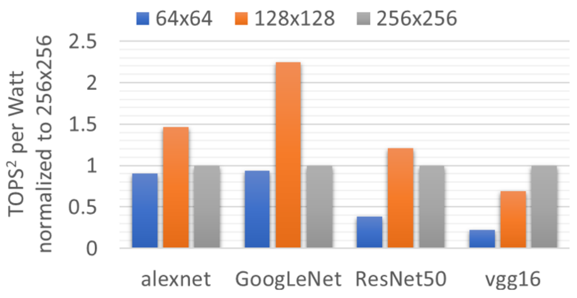
The data in Figure 3.22 shows the impact of reducing array dimensions on the energy required per multiply-accumulate operation. The elimination of permutations is observed as a reduction of the weight loading energy, although weight loading energy is also reduced by the reduced capacity of the weight buffer for the smaller array. Reducing the array dimensions also reduces array idle time by more closely matching the memory bandwidth to the array's weight loading bandwidth. Output accumulator utilization and input buffer read accesses both increase since more partial matrix-vector products need to be accumulated for each layer mapped to the smaller array. When the array size is too small, the input and output energy components outweigh the energy reductions derived from better array utilization. For Resnet50 and GoogLeNet the minimum energy per multiply-accumulate is observed at an array size of 128x128. For AlexNet the minimum is observed at an array size of 64x64. For VGG16, the minimum is observed at the original array size of 256x256. Based on the performance impact and other efficiency



(a) TOPS



(b) TOPS per Watt



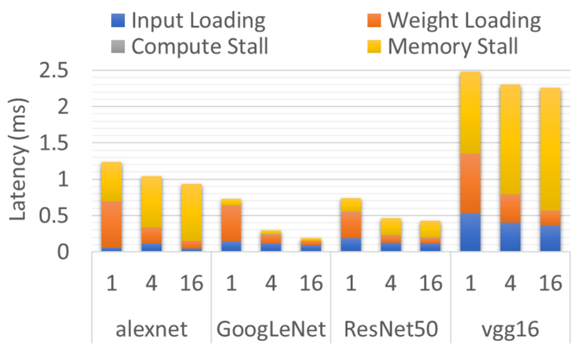
(c) TOPS^2 per Watt

Figure 3.23: Effect of adjustments to array dimensions.

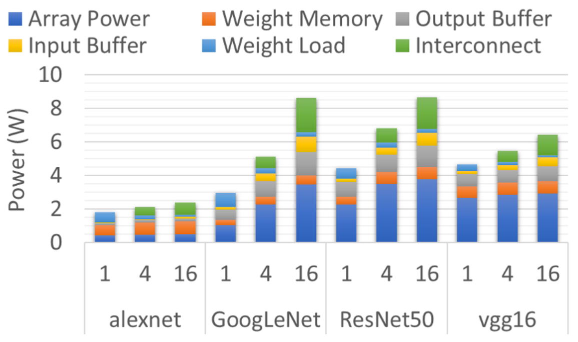
metrics shown in Figure 3.23, it appears that an array size of 128x128 would be preferable for all benchmarks other than VGG16 which experiences a significant performance degradation with smaller array sizes due to the compute bandwidth limitation.

3.6.4 Multi-tile Architecture

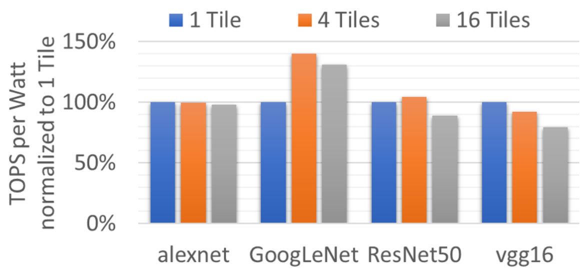
Within tiled architectures composed of multiple smaller systolic array tiles, both high utilization and high compute bandwidth can be achieved. One of the difficulties that arises with a tiled architecture is a net increase in array interface bandwidths, which raises the bandwidth demands on the interconnect, buffers, and memory system. The tiled architectures are arranged as in Figure 3.17. Sufficient on-chip bandwidth to support operation of all arrays in parallel is assumed, only subject to the memory bandwidth.



(a) Latency



(b) Power



(c) TOPS/W

Figure 3.24: Sensitivity Study of the impact of tiling on each benchmark.

Thus the interconnect design is not optimal, but it allows for exploration of the power implications of supporting these high bandwidths.

As can be seen in Figure 3.24a, the performance of the baseline 256x256 monolithic diastolic array is substantially lower than the tiled architectures, with the latency reduction of weight loading outweighing the increased memory stall due to higher bandwidth demand for all benchmarks. Reducing the array dimensions can also introduce serialization of dot-product computations since each neuron is allocated to only one tile, however this would appear to not be a significant factor in the overall latency as input loading latency appears to typically decrease with the increased tiling degree, and input loading represents only a small portion of the overall latency.

As the performance improves for the tiled architectures, an increasingly substantial portion of the power consumption is due to the interconnect, as can be seen in Figure 3.24b. Benchmarks that experience improved array utilization due to tiling also demonstrate increasing array power and buffer power components, similar to the power observations made regarding increasing batch size in Figure 3.21. Unlike increased batch size however, the memory power tends to increase due to the increased memory bandwidth exposed by tiling.

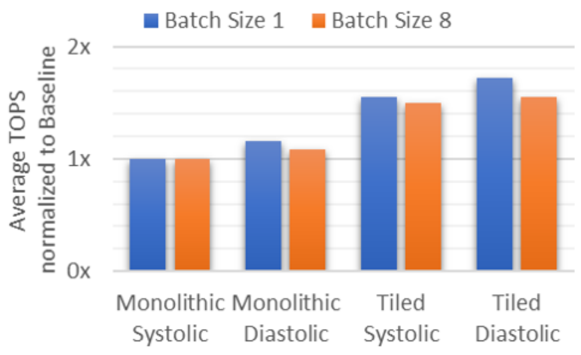
In general, tiling tends to increase both performance and power consumption for all benchmarks. However, the amount that each metric increases varies from one benchmark to the next. Power efficiency is considered to determine which benchmarks benefit from tiling by having greater performance improvements than power demands. Figure 3.24c demonstrates that GoogLeNet experiences higher power efficiency than other configurations on four tiles each with a 128x128 array. Other benchmarks do not demonstrate significant power efficiency improvements with tiling, and VGG16 demonstrates a power efficiency degradation. The 16-tile architecture demonstrates a degradation in comparison to the 4-tile TOPS/W, which is primarily caused by the increasing interconnect overhead which may be possible to reduce with better engineering of the interconnect than the ideal bandwidth configuration used in this model. For the purposes of this study the tiled architecture under consideration is the 4-tile configuration, which demonstrates a favorable TOPS^2/W metric across the set of benchmarks.

3.6.5 Results Summary

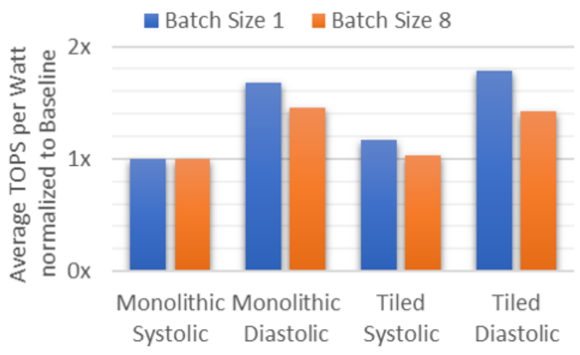
Figure 3.25 contains results pertaining to four architectures. The baseline design is the 256x256 monolithic systolic array. The other three designs include a monolithic 256x256 diastolic array, 4-tiled 128x128 systolic array and a 4-tiled 128x128 diastolic array. The diastolic array contains cells with a width of 16 and a height of 2, meaning that each cell of the diastolic array contains two 16-wide adder-tree based dot-product units that share their activation input operands to form a 2x16 matrix-vector multiplication unit within each cell. Results are considered for batch size 1 and batch size 8 inference averaged over the set of benchmarks.

The results in Figure 3.25a demonstrate the incremental performance improvements resulting from both tiling and diastolic cell expansion. Diastolic cell expansion alone provides 16% and 9% average performance improvements for batch size 1 and batch size 8 inference respectively, while tiling alone respectively provides 55% and 50% improvements. The combined average performance improvements are 72% and 55%, respectively, with tiling being responsible for the bulk of the improved performance.

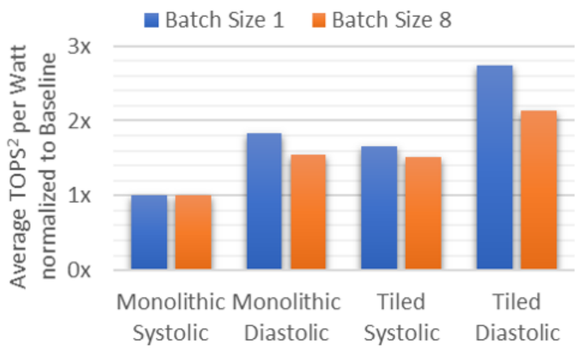
The results in Figure 3.25b pertain to the average TOPS/W for each architecture. Diastolic cell expansion is responsible for a 68% and 46% improvement to TOPS/W, respectively for batch sizes of 1 and 8. Although tiling does typically require increased power consumption, it does also provide a small power efficiency improvement of 17% and 3%, respectively. Overall, the average TOPS/W improvement of the baseline for the tiled diastolic design is 79% and 43%.



(a) Average Performance



(b) Average TOPS/W



(c) Average TOPS^2/W

Figure 3.25: Comparison of the Baseline Monolithic 256x256 Systolic array and the four Tiles of 128x128 Systolic arrays with and without Diastolic cells with width 16 and height 2 (baseline is width 1 and height 1). The memory system for each architecture contains two LPDDR4-4266 packages and the accelerator clock frequency is 667MHz.

The results in Figure 3.25c pertain to the average TOPS^2/W representing the combined impact of performance and power-efficiency improvements. The improvement due to diastolic arrays alone is 84% and 54% for batch size 1 and 8 respectively. The improvement over the baseline due to tiling alone is 65% and 51%. The overall improvement over the baseline average TOPS^2/W is 173% for batch size one and 113% for batch size eight. So, although the improvements with larger batch sizes are not as much as those for batch size one, there is still a sizable improvement resulting from these combined optimizations.

3.7 Chapter Summary

In this chapter, a model for a systolic accelerator architecture is developed for evaluating the performance and power characteristics of a systolic array with expanded cells, each containing multiple lanes of dot-product units which have shallow adder-tree based pipelines that exhibit lower latency than typical systolic alternatives for dot-product calculation. Expanding systolic array cells enables elimination of many of the flip-flops of the systolic array, and in this manner result in a 30% reduction of array area, with significant energy-efficiency improvements and power savings as well as higher performance inference with small batch sizes, where the array would otherwise be underutilized. A sensitivity study is considered in order to identify the ideal dimensions of the 2D arrays of a neural network inference accelerator composed of multiple tiled arrays with equivalent computational resources to the monolithic arrays. For the set of CNN benchmarks considered, the four tiles of 128x128 arrays is shown to be a more efficient choice for evaluating these networks, and when expanded cells are integrated into this accelerator we observe an average performance improvement of 72% over the baseline design for batch size one inference and more than 3x speedup for some benchmarks, with TOPS/W improving by 79% on average and better than 2.5x observed for some benchmarks.

4 CODEBOOK QUANTIZATION SPECIALIZED ACCELERATION

If you're walking down the right path and you're willing to keep walking, eventually you'll make progress.

— Barack Obama

Data quantization is one of the most effective low-power techniques used by efficient neural network inference accelerators. The low energy requirements associated with usage of quantized data formats are associated with simplifies arithmetic logic, reduces storage requirements, and lowers communication bandwidth requirements. Codebook based quantization is an alternative to linear quantization that achieves a high compression ratio by representing values as an index into a table of higher precision values. Codebook based compression can achieve higher accuracy at comparable compression ratios to linear quantization, but unlike linear quantization requires higher precision arithmetic logic and a codebook lookup to translate from codebook indices to weight values. This chapter considers accelerator design techniques that integrate specialized codebook storage structures into accelerators designed for codebook quantization of both weights and activations. Codebook based quantization of both weights and activations can be used to reduce the computation power component by tailoring the multiplication stage of the neural processing element to the codebook format. The accelerator maintains low on-chip communication and storage energy by using an output-stationary computation sequence that avoids communication of partial sums external to the processing element until the activation function is evaluated to a codebook index.

The accelerator considered is a tiled architecture similar to DaDianNao [16], operating at 667MHz with a peak performance of about 5.5 TeraOps/s. Using 16-entry 16-bit codebooks, a power reduction of more than 70% is demonstrated resulting from only modifying on-chip storage and interconnect requirements enabled by codebook based compression. A 74% reduction in compute power is achieved by using lookup tables indexed by both a weight index and an activation index to determine each 16-bit product in place of the 16-bit truncated multiplication used in the baseline design, representing a further 26% overall power reduction beyond what was achieved by interconnect and storage modifications alone. This compute energy is comparable to that of 8-bit multiplication, which is slightly less precise than truncated 16-bit multiplication. To compare the compute energy between implementations using 8-bit multipliers and dual-index codebook look-ups, the scaling characteristics of the lookup based design is considered as the number of weight and activation codebook entries are adjusted to be smaller than the vector width of the processing element, demonstrating better scaling characteristics for relatively smaller codebooks than multiplier based designs.

4.1 Introduction

Codebook based quantization of neural networks achieves high compression ratios with better achievable accuracy than linear quantization schemes that have comparable compression ratios. Codebook based quantization stores indices into codebooks containing higher precision values such that only compressed indices are stored and communicated, and the high precision codebook entry is used for calculations. In general, codebook based representations of a distribution of values can be determined based on clustering techniques which converge to an optimal set of representative values to be used as codebook entries used in place of the original values during neural network evaluation.

Accelerators designed for low precision fixed point formats gain both storage capacity and interconnect bandwidth efficiency as well as simplified arithmetic logic requirements. However, as better compression ratios with comparable accuracy can be achieved codebook based quantization [48, 49, 84], accelerator architectures that provide support for codebooks stand to be more efficient than purely fixed-point alternatives. Codebook based quantization reduces the memory and bandwidth requirements, however since arithmetic operations cannot be applied directly to codebook indices, an additional decoding step is necessary before calculations can take place. In neural network accelerators designed to support codebook quantization, codebook lookups within the processing elements can enable dequantization of weights along the accelerator data-path [48, 84] without disrupting performance.

While previous work on accelerators supporting codebook based quantization has mainly focused on weight quantization [48, 84], encoding activations as codebook indices

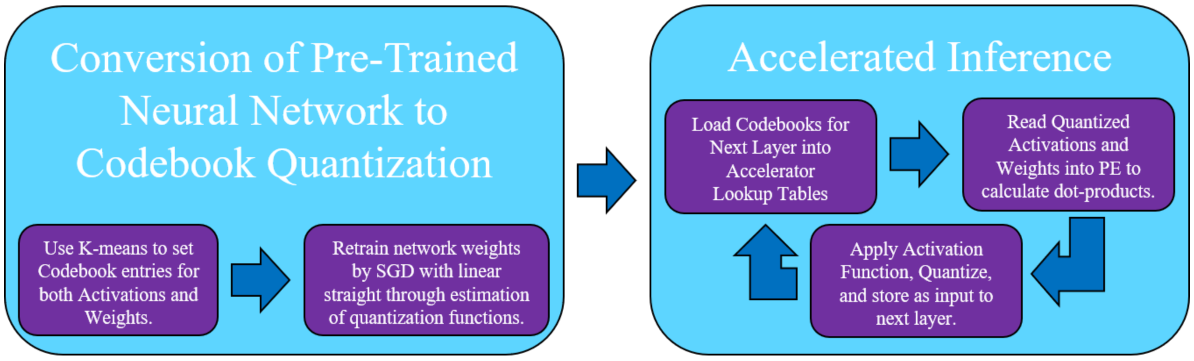

Conversion of Pre-Trained Neural Network to Codebook Quantization
Use K-means to set Codebook entries for both Activations and Weights.
Retrain network weights by SGD with linear straight through estimation of quantization functions.
Accelerated Inference
Load Codebooks for Next Layer into Accelerator Lookup Tables
Read Quantized Activations and Weights into PE to calculate dot-products.
Apply Activation Function, Quantize, and store as input to next layer.

Figure 4.1: End-to-end inference acceleration and training for codebook quantization of activations and weights.

is also beneficial as it reduces both the activation storage and communication overhead. Furthermore, accelerators designed for codebook quantization of both activations and weights can use processing elements that replace multiplication logic with codebook lookups based on both codebook indices. Assuming common weight and activation codebooks across all neurons in a layer, lookup tables (LUT) can be shared across the dot-product lanes of a wide matrix-vector unit such that an activation shared as the input to many neurons requires only one lookup for a set of dot-product lanes. As is shown in Figure 4.1, the codebook LUTs are populated from memory before processing each layer. The amount of computations required for each layer should be large enough that the overhead of loading the codebook is insignificant. Synapses are calculated based on the values retrieved from the codebook lookups for each weight and activation and accumulated for each neuron. Once a neuron's synapses have fully accumulated, an activation function implemented as codebook based thresholding is applied to produce a quantized value that can be written to neuron storage by the processing element. Processing schedules for these processing elements must consider the amount of data that can be maintained near the processing elements and the cost of data movement for each data type.

This chapter makes the following contributions:

1. Details the design of a neural processing accelerator where on-chip communication is primarily in a compressed format.
2. Incorporates codebook based quantization of both weights and activations to achieve energy efficient computation based on LUT lookups.

3. Integrates activation function evaluation logic that handles quantization, activation, and normalization within a single operation.
4. Considers the scheduling implications of integration of activation quantization within the processing element.
5. Evaluates energy impacts of data-path adjustments for a set of representative neural networks.

4.2 Background

There exists some leeway in the numeric precision necessary for accurate outcomes from neural network inference, which creates opportunities for trade-offs between precision and efficiency [54]. Often, digital implementations of neural networks written for general purpose hardware use 32-bit floating point parameters, which is generally sufficient precision to accommodate neural network algorithms designed for continuous number systems. On hardware that supports lower precision arithmetic, it can be worthwhile to use reduced numerical precision for neural network inference, leading many of the recently proposed neural network inference accelerators to embrace low precision number formats as a means of improving efficiency [14, 63, 48]. However, it becomes much more difficult to achieve high accuracy at very low precisions so specialized training techniques are required to properly utilize low-precision capabilities.

Different quantization schemes can be categorized according to the properties of the function used to map to the low precision representations, which is generally a piecewise-constant mapping from continuous values to the quantized representations. *Structured* quantization schemes can be described by mathematical expression, such as the linear mapping of a fixed point representation or the logarithmic mapping of a floating point exponent. These structured formats are easily integrated into arithmetic data paths since arithmetic operations can be easily applied directly in the quantized format. *Codebook* quantization schemes represent a non-linear mapping from an index to a value using a codebook for translation from the encoded format to a representative value. Codebook based representations are more flexible in that they are not tied to any mathematical function, but they are also more difficult to integrate into computational data-paths since a codebook lookup is required prior to calculation. The codebook entries themselves are typically determined using data clustering techniques such as k-means [41, 49, 21].

Training techniques that apply updates directly to variables in their quantized format can exhibit sub-optimal convergence characteristics. Stochastic rounding [46] provides an approach for directly training quantized weights where randomness within rounding decisions evades stale states. However, alternative techniques that train high-precision shadow weights that get quantized during inference can have even better convergence characteristics than stochastic rounding based techniques [89]. Deep Compression [49] is a training technique specifically designed for codebook based quantization which first associates each parameter with a cluster and then trains the values representing each cluster. Thus, rather than maintaining a high precision version of the weights, weights are stored compactly in their quantized form and although the quantized weights are fixed the network is still trainable. However, quantization techniques that involve retraining of a network trained at full precision can still exhibit better accuracy than quantization techniques that include quantization throughout the training process [45], as will be demonstrated in Chapter 5.

Many of the neural network accelerators designed for codebook quantization target codebook quantization of weights only [48, 84]. Weight quantization impacts model size and makes networks more portable. Activation quantization impacts transient storage overheads and improves inference efficiency. This chapter discusses efficiency of processing element designs for an accelerator that supports codebook quantization of both weights and activations.

When activations are quantized, training of neural networks by gradient back-propagation requires propagation through the quantization functions. The piecewise-constant functions representing quantization are non-differentiable and therefore incompatible with traditional back-propagation based stochastic gradient descent. Training schemes for networks with quantized activations therefore use what has come to be known as a straight through estimator (STE) [152, 116, 25, 10], which approximates the back propagated gradients by either transparently passing through the quantization function or replacing the quantization function by a simplified piecewise-differentiable function. Activation quantization has been demonstrated to be effective down to one-bit per activation [116, 25], however accuracy generally improves with quantization schemes that represent each activation using more bits [10, 149]. Similar to the data based approaches used for identifying the quantization levels for weights, the techniques used for identifying non-linear quantization levels for activations have included clustering techniques, such as Lloyd's algorithm [10] or using block coordinate descent [149]. Layer-wise codebooks [149, 49] can provide a favorable ratio of codebooks to synapses.

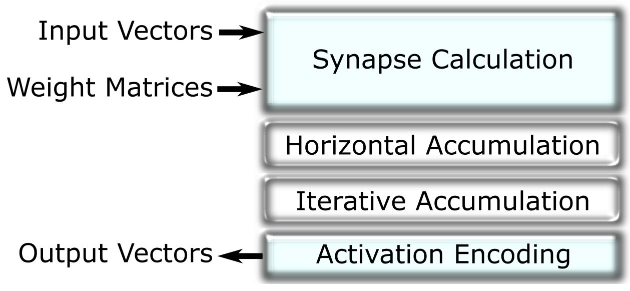
Given the different precision and memory requirements for training and inference of quantized networks, different hardware is appropriate for each task. Inference accelerators which do not need to support training can be optimized for the inference task only. In this chapter, a description of the design of a neural processing unit which exploits codebook based quantization of both weights and activations to efficiently store, communicate, and process data is developed in Section 4.3. The design features LUT based computation tailored to the codebook based quantization scheme. The neural processing unit is evaluated in Section 6.4 based on the experimental setup described in Section 4.4. The next chapter will return to the discussion of training techniques that target an accelerator supporting codebook quantization of both activations and weights, such as the one described in this chapter.

4.3 Accelerator Design

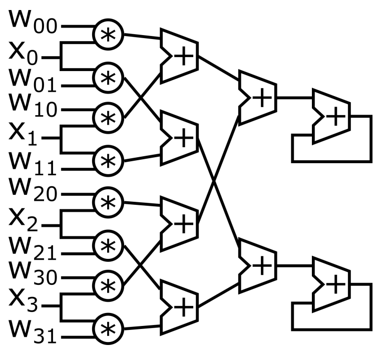
The vast majority of operations performed during neural network inference are the multiply-accumulation operations associated with synapse aggregation in Eq. 2.1. Therefore, neural network accelerators typically contain many multiply-accumulation units to support high computational throughput. Figure 4.2b is a schematic for a matrix-vector product unit with an input width of four and an output width of two. An adder tree is used to accumulate the dot-product corresponding to subsets of each matrix row.

The inputs to the unit are an activation vector and a weight matrix, where each input activation is paired with a weight corresponding to each output neuron. This mode of parallel synapse calculation exploits the reuse of inputs and outputs to achieve high computational throughput with relatively low input and output bandwidth requirements. Computing the matrix-vector product between large weight matrices and activation vectors involves accumulation of multiple component matrix-vector products partitioned according to the dimensions of the processing element.

The neuron processing data flow follows the sequence illustrated in Figure 4.2a. Synapse calculation is performed within the first stage of the processing element where both the weight matrix and the activation vector are provided as codebook indices. The synapses for each output are summed across the set of inputs as in a typical matrix-vector product. This reduction takes place within the second stage where a set of parallel adder trees produce a set of partial sums that is combined with previously calculated partial sums maintained in an output buffer within the third stage. Once all synapses have been accumulated for a given output, the activation function is applied within the fourth



(a) Neural processing unit stages



(b) 4x2 Matrix-Vector Unit

Figure 4.2: Neural Processing Unit

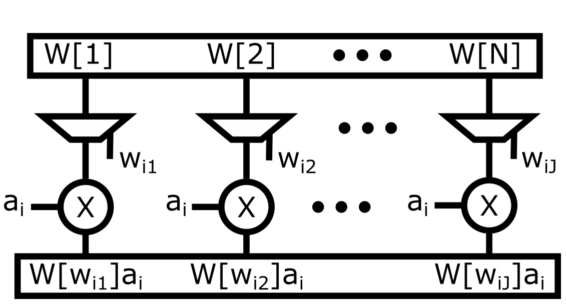
stage.

Neural network accelerator architectures designed for codebook based quantization must decode operands prior to synapse calculation. Codebook decoding integrated closer to the functional units leads to more efficient use of the compressed weight format. In this section we incrementally develop the design of a neural processing element with integrated decoding/encoding of weights and activations, focusing on the first and last stages of the pipeline where dequantization and quantization take place.

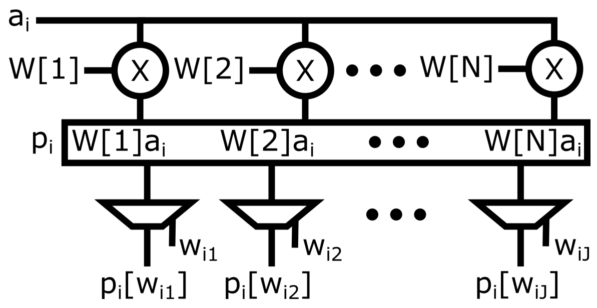
4.3.1 Synapse Calculation

Codebook quantized synapse weights are stored in weight memory and communicated as indices corresponding to codebook entries that must be decoded for use with conventional multiplication logic. Performing high throughput matrix-vector products requires that multiple codebook lookups be performed in unison, which can be implemented using a single LUT, since all weights share the same codebook. The synapse calculation implementation in Figure 4.3a contains a set of multiplexers sharing the codebook array as their inputs to select codewords corresponding to each synapse weight index. The output of each of these multiplexers is the weight operand input to the multiplier that produces the synapse calculation, such that each multiplier is paired with a multiplexer to select its weight operand. The matrix-vector unit contains multiple instances of the structure in Figure 4.3a, according to the input activation vector size of the unit.

If the number of weight codebook entries is small in comparison to the number of multiplications per activation, then calculating a full set of possible multiplications



(a) Shared Codebook Access



(b) Precomputed Products

Figure 4.3: Single input, J output multiplication.

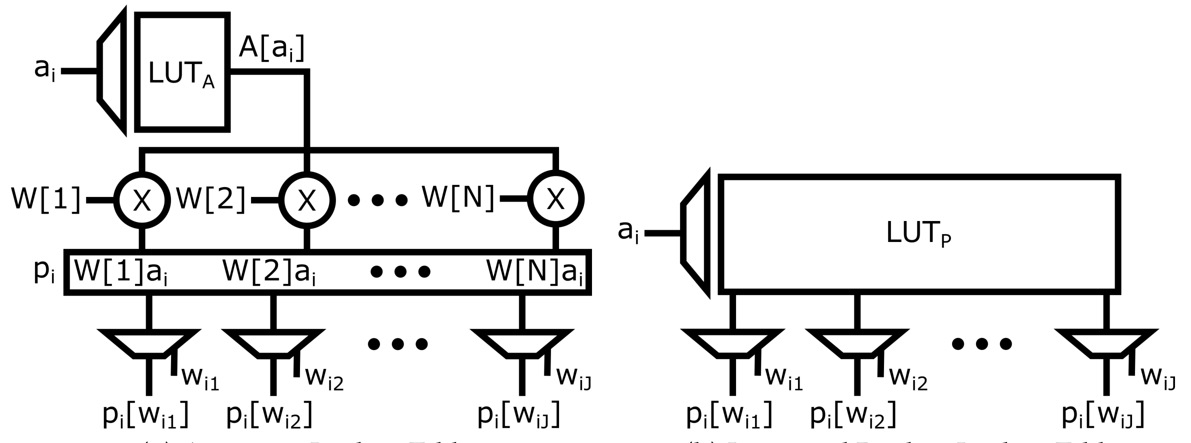
between an activation and each codebook entry would require no more multiplications than to calculate the products with each weight. We take advantage of this property of the codebook by placing synapse weight multiplexing at the output of the multipliers, as is shown in Figure 4.3b. The resulting neuron calculation can be expressed in terms of the weight index of an array (p_i) containing the set of multiplications between an activation (a_i) and the set of weight codebook entries (W) as in Eq. 4.1,

$$n_j = \sigma\left(\sum_{i=0}^{n-1} p_i\left[w_{ij}\right]\right) \qquad \textit{for} \qquad p_i[c] = W[c] \times a_i, \tag{4.1}$$

where w_{ij} is the codebook index stored for the synapse between input i and output j. The codebook entries of W are static input operands for the set of multipliers and the other operand input to each multiplier is an input activation value, which is shared among as many multipliers as there are weight codebook entries. The result of the set of multiplications with activation a_i is output by the LUT (p_i) from which the synapse weight indices select the corresponding calculated synapse.

Selection of the optimal synapse calculation implementation between the two designs in Figure 4.3 is based on the ratio of codebook entries to output ports. In both designs, the inputs to the multiplexers are high fanout nets, however since the design in Figure 4.3a multiplexes a fixed codebook, the switching activity on these nets remains low. The design in Figure 4.3b, on the other hand, can suffer from high switching activity at the multiplexer inputs. Given input activation sequences with repeated values, multiple outputs can be processed based on the same calculations, which provides some energy savings, but the common case given typical scheduling schemes for convolutional layers would involve processing a new activation every cycle resulting in high activity factors on these nets. Although the multiplexing in 4.3b is more expensive than in 4.3a, the implementation in Figure 4.3b can still be beneficial if multiplexing is less expensive than multiplication. This is because the implementation in Figure 4.3a does not scale well as the codebook size is reduced. Reduction of the number of codebook entries in 4.3a only affects the multiplexers, which are already inexpensive. Reducing the number of codebook entries in 4.3b eliminates multipliers, which can save quite a bit of energy.

This design for the synapse calculation logic is itself similar to a design previously described as q-tables [84], where significant energy savings are demonstrated for quantized LSTMs. A fundamental difference between q-tables and our design is that we include multi-ported access to the arrays containing the set of products while q-tables are single ported SRAMs. Our design is intended to exploit activation reuse across the lanes of the matrix-vector product unit. Adding multiplexers at the output of the array increasing compute throughput without requiring additional multipliers for additional lanes.



(a) Activation Lookup Table (b) Integrated Product Lookup Table

Figure 4.4: LUT based multiplication using activation index.

Activation codebook support in these designs is implemented as a small lookup table accessed prior to synapse calculation. In the design depicted in Figure 4.4a, a lookup table access translates from each activation index to a representative value. The result of multiplication with each of the weight quantization levels is the input to a set of multiplexers, just as in the previous design in Figure 4.3b with the activation lookups included to obtain the activation value. The activation codebook lookups are inexpensive due to the low bit-width and few entries in the table. However, the design does not scale well when the number of activation codebook entries are reduced. The expression for the neuron output for the processing element design that includes the activation LUT is given in Equation 4.2

$$n_j = \sigma\left(\sum_{i=0}^{n-1} W\left[w_{ij}\right] \times A\left[a_i\right]\right), \tag{4.2}$$

where W and A represent the weight and activation lookup tables respectively.

An alternative implementation subsumes all multiplications of Figure 4.4a within a single lookup table containing all possible products between the weight and activation codebook values, as is shown in Figure 4.4b. This provides an implementation of codebook based multiplication that does not require any multiplication logic. The product lookup table has as many entries as the activation lookup table in Figure 4.4a, however each entry corresponds to the product of one activation with all entries in the synaptic weight codebook. The size and access energy of the product lookup table are dependent upon the number of quantization levels used for activations, the number of weight quantization levels, and the bit-width of the stored products, such that reductions of either the number of weight entries or activation entries can result in notable energy reductions. The neuron calculations performed using product LUT lookups are expressed as

$$n_j = \sigma\left(\sum_{i=0}^{n-1} P\left[a_i\right]\left[w_{ij}\right]\right), \tag{4.3}$$

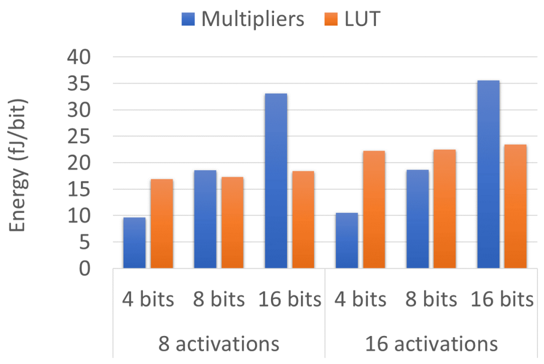
where P represents the product set lookup table with a row addressed by the activation index a_i and the column selected by the weight index w_{ij} to produce representative values for each product to be passed to adder tree reduction.

The LUTs used for synapse calculation are updated relatively infrequently so the update overhead remains minimal. An additional small set of multipliers, useful for populating the LUTs, is included within the accelerator with minimal energy impact due to infrequent use. This design decision is rather inconsequential however, since computing the codebook entries amounts to a small set of infrequent simple multiplications that does not need to be accelerated.

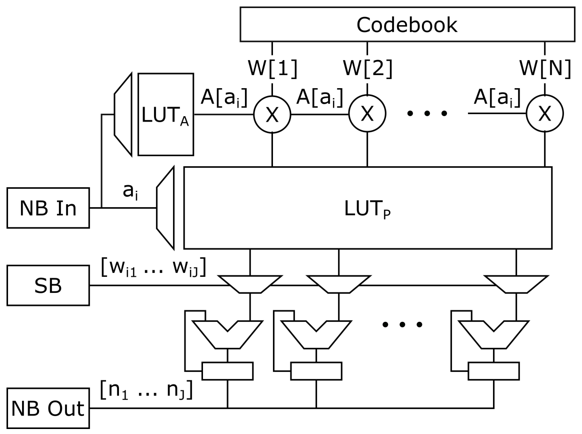
Providing mechanisms for populating the LUTs directly from memory provides an opportunity to load the product LUTs with values other than the "true" products of the two codebooks. The possibility that alternative set of representative synapse values not derived using the multiplication operation could produce better accuracy than the "true" codebook products is intriguing, however the relative simplicity of training layers based on linear functions would seem to indicate that "true" product storage is the most reasonable use case.

To provide some insight into the scaling properties of the synapse calculation design alternatives, the data presented in Figure 4.5a illustrates the difference in scaling behavior of a set of parallel multipliers as compared to a wide lookup-table. While the energy per bit calculated using the multipliers is most significantly impacted by the bit-width of the operands, the energy per bit read from the LUT is most significantly impacted by the number of LUT entries. Therefore, the product LUT based synapse calculation may become undesirable if the number of activation quantization levels becomes too large of if the bit-width becomes too small.

If the activation codebook size exceeds the maximum worthwhile number of LUT entries, then a combined approach in which more common activation values are maintained within the LUT while a multiplier array is used for infrequent codewords may be preferable, such as the one shown in Figure 4.5b. When a new activation arrives, either a tag check or a threshold check would take place to determine if the products are present in the lookup table, and if not, it is calculated using the multipliers and activation codebook and may either populate the LUT or bypass the LUT. Identifying



(a) Comparison of energy required to access LUT vs perform multiplications.



(b) Combined LUT and multiplier array implementation of synapse calculation logic.

Figure 4.5: Scaling Considerations.

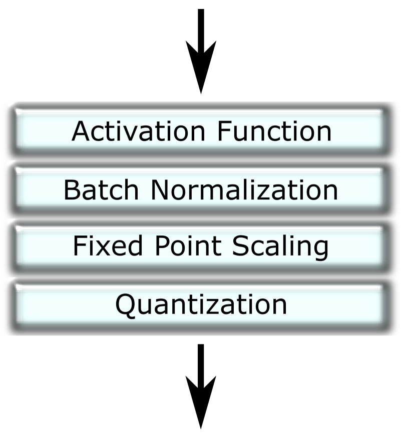
the optimal LUT size will vary depending upon multiplier implementations, codeword bit-widths, and technology. Ultimately, the primary benefit of including these additional multipliers is scalability to more activation levels than the original design.

Providing additional multipliers as an alternative data path can be worthwhile, since some implementations of extreme quantization have demonstrated better accuracy when the input layer is not quantized [24]. Other layers would have encoded input activations generated within the accelerator, while the input layer would require less preprocessing since the accelerator would accept these inputs in their raw fixed point format.

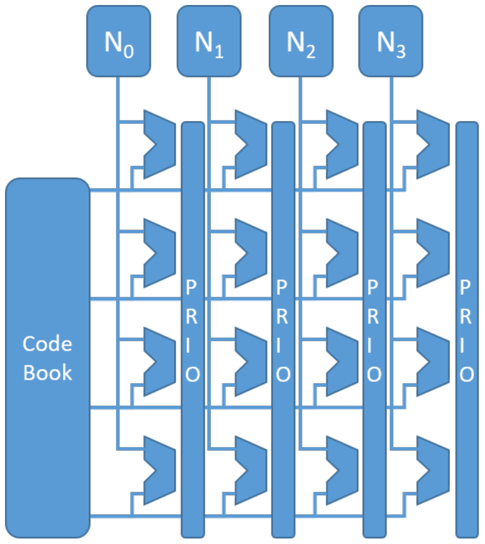
4.3.2 Activation Calculation

Once full accumulation of a neuron completes, the activation function is applied. Different neural networks may use different activation functions including rectified linear functions, sigmoids, or hyperbolic tangents. When using codebook based quantization of activations, the activation function can be easily integrated into the quantization logic in order to produce a single codebook index without evaluating intermediate steps. Unlike the codebooks used during synapse calculation which map to representative values used as arithmetic operands, the activation calculation codebook stores thresholds between quantization levels corresponding to the codebook that will be used to decode activations in the subsequent layer. As is shown in Figure 4.6a, the activation encoder subsumes batch normalizations, fixed-point scaling, the activation function and the discretization function of the next layer by using the composition of these functions as the basis for the threshold values stored in the activation encoding codebook.

A high throughput activation encoder is implemented as a set of comparators that evaluate the accumulated sum of products against each codebook threshold, as is shown in Figure 4.6b. The quantization level indices are sorted monotonically such that the output of a priority encoder maps to the codebook index pertaining to the lowest surpassed threshold. The accelerator evaluates any max pooling operations directly



(a) Operations subsumed by quantizer



(b) Parallel Codebook Quantizer

Figure 4.6: Codebook quantizer design

on codebook indices while writing the activations to neuron memory by comparing previously stored indices at the same address in read-modify-write fashion.

4.4 Methodology

In order to evaluate the efficacy of the LUT based accelerator design approach, the processing elements are synthesized using Synopsys Design Compiler and a 45nm TSMC standard cell library operating at a frequency of 667MHz with a supply voltage of 0.9V. Energy characterizations are based on the post-synthesis netlists with switching activity collected from RTL simulation for uniformly distributed codebooks and indices. An architectural performance model of the accelerator is used to gather run-time data pertaining to common DNN benchmarks, which is used in conjunction with the energy characterization to assess the energy impact of different scheduling schemes and architectural design decisions for common DNN benchmarks. Experiments are conducted for architectures with equivalent performance characteristics in order to study the impact on energy of the micro-architectural optimizations and adjustments discussed in the previous section.

The baseline accelerator design is loosely based on DaDianNao [16]. It is a tiled architecture with 16 tiles each containing a 16x16 16-bit matrix-vector product based neural processing unit. In DaDianNao, each tile contains 2MB of local eDRAM weight storage and there is a 4MB central eDRAM for neuron storage. We assume the same tiled topology (depicted in Figure 4.7a), operand bit-widths, vector dimensions, and storage capacities as DaDianNao for the baseline. One difference between the baseline and DaDianNao is that all memory structures are modeled as 45nm SRAM, where the technology node is selected to match the 45nm TSMC library used to model the processing element.

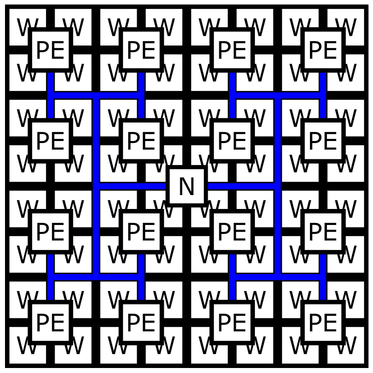
A DaDianNao-like model is frequently used as a baseline for studies pertaining to neural network accelerators, which is one of the reasons that it was selected for this study. Furthermore, DaDianNao is representative of a class of NPU designs that locates synaptic weights in large memory structures close to the processing elements, which can be appropriate for inference accelerators where the weights are expected to mostly remain read-only. The large on-chip memory structures that result from this design decision greatly benefit from data compression and are therefore well suited for codebook quantization. Designs like Google's TPU [63] and Cambricon [91], which depend on off-chip DRAM for network parameters, would gain similar benefits in data-path efficiency from codebook quantization and lookup tables. However, compression techniques demonstrate greater benefits for designs such as DaDianNao, where weights are stored on chip.

The tiled architecture under consideration has three specialized interconnects. The first is a set of bidirectional point-to-point links between a centralized on-chip neuron memory and each processing element used for communication of quantized activations. The second is for communication of quantized synaptic weights between each processing element and its designated local weight memory. The third is for off-chip communication and is used primarily for setup of on-chip memory structures between neural network layers, including communication of input and output sets.

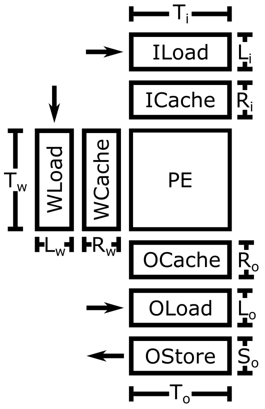
The cost of communication is related to the distance between the processing elements and their associated storage structures. The more expensive, frequent communication of high precision partial sums remains within the processing element between accumulators and the output buffer. Synaptic weights are less expensive to communicate than they would be if they were not compressed, however weights require high bandwidth which motivates collocating the synapse memory with the processing element as well. The less expensive, lower bandwidth communication of quantized activations utilizes longer distance links between the processing element and the shared neuron memory located further from the processing elements.

We consider several different architectures, each designed for a particular quantized format. The neuron memory and weight memory capacities and port widths are scaled down by a factor of four, proportional to the quantization scheme's compression ratio, such that a codebook quantization scheme with 16 entry 16-bit codebooks for both weights and activations has on-chip memories and interconnect scaled down by a factor of four in comparison to the baseline. This codebook size was chosen primarily because prior work has demonstrated accurate inference using codebook quantization to 16 codebook entries [48, 149], but also because 16-entry LUTs can be easily synthesized as full-swing latch based arrays, and because our own training experiments demonstrated that high accuracy can be achieved with 16 activation and weight codebook entries.

Processing elements are modeled as in Figure 4.7b with an input buffer, weight buffer, output buffer, and a processing element capable of all processing stages shown in Figure 4.2a. Each buffer of the performance model contains a cache and a communi-



(a) Chip Layout



(b) PE Node Model

Figure 4.7: Simulation model

cation buffer where data resides only during active transfers. The cache replacement policy is assumed to be LRU, which is sufficient to exploit reuse opportunities within the sequential access patterns of DNN processing. The energy model considers the interconnect between each processing element tile and the central neuron memory as well as traffic between each processing element and its associated weight buffers. For the purposes of these experiments, weights are assumed to be preloaded into the weight buffers, and the overhead associated with initially loading weights into the buffers is not modeled. We assume sufficient buffer capacity to accommodate at least two layers at a time such that all weight loads are compulsory and easily prefetched under a layer-wise scheduling scheme. Reconfiguration is presumed to be infrequent with minimal energy overhead. In either case, analysis that excludes an off-chip traffic model is informative even though the benefits of model compression are not considered for off-chip accesses.

The sequencing of the component calculations within the processing element directly impacts the data movement involved in calculating the matrix-vector product. When using codebook quantization of activations and weights, the intermediate partial sums have much larger bit widths than the codebook indexes and are therefore more costly to communicate. Output stationary scheduling schemes that prevent communication of partial sums outside of the processing elements are therefore appropriate. Neurons are mapped to PEs by partitioning into sets of output channels until there are not enough output channels to fill the width of the processing element, then partitioning of the layer is done spatially. This represents a preference for avoiding unnecessarily duplicating kernel features across multiple tiles, which results in more efficient weight storage utilization than spatial partitioning of a layer's neurons alone.

Within each tile, the output buffer is iteratively allocated to sets of neurons which fully accumulate before the buffer is reallocated. Thus, greater output buffer capacity provides a better opportunity for data reuse by having more active neuron calculations allocated to each PE at a time. This reuse opportunity represents a tradeoff between output buffer access energy and data reuse that will be analyzed in more detail in section 6.4.

For convolutional layers, weights can be reused from one input set to the next. Using an outer-output/inner-weight (O/W) loop ordering strategy effectively exploits the reusability of the weight sub-matrices. The O/W ordering loads multiple positions of the same output feature set to the output buffer at a given time such that each time a weight sub-matrix is loaded it can be evaluated for the full set of allocated outputs.

All fully-connected layer weights are unique, which places a higher bandwidth demand on the weight memory since fully connected weights are only reused with batch sizes larger than one. In these cases where the batch size is one, an outer-output/inner-input (O/I) ordering exploits the sharing of the inputs by all outputs of the fully-connected layers. When the O/I scheme is used, either the LUT outputs can be held constant for multiple output sets or the same output set can be accumulated for multiple inputs. It turns out that although accessing the LUT requires less energy than multiplication, it does consume more energy than accessing the output buffer, so the sequencing used in the following section prefers holding the input activations constant and iterating through the output buffer.

The input and weight buffers are read-only so their communication buffers are only for loading data. The output buffer needs to store the calculated neuron outputs and therefore requires a store buffer. The output buffer also may need to load values in some designs where partial sums or biases need to be retrieved from sources external to the processing node in order to accumulate the full neuron potential. However, in an accelerator aware neural network, biases can be expressed as weights whose input activation is a constant activation index.

4.5 Evaluation

The numeric format used in typical matrix-vector processing element arithmetic units has a significant impact on their energy requirements when performing multiply-accumulate operations. Figure 4.8 demonstrates the impact of reducing numeric precision on compute energy per synapse. First considering half precision floating point, we observe that the addition operation requires almost the same energy expense as multiplication for half-precision floating points. Fixed point addition is significantly more efficient than floating point, representing 15-30% of the multiply-accumulate energy without truncation and 10-20% when the multiplier outputs are truncated and a reduced bit-width adder tree is used. Truncated multiplication [14] retains high order bits after multiplication, thereby supporting a wider range of values than low bit-width multiplication with equivalent output width. The primary energy reduction resulting from truncation is in the adder trees since most of the multiplier logic is still required to calculate the truncated products. Fixed point addition scales approximately linearly with the operand bit-width, while multiplication experiences higher order energy reduction with reduced bit-widths.

Figure 4.9 compares average power consumed by the 16-bit accelerator without codebook quantization to the accelerator with 16 entry 16-bit codebooks, which have essentially the same processing elements with peripheral dequantization. We observe a 70% power reduction resulting from resizing the neuron and weight SRAMs according to the quantization compression ratio of 4:1. These power results are averages across

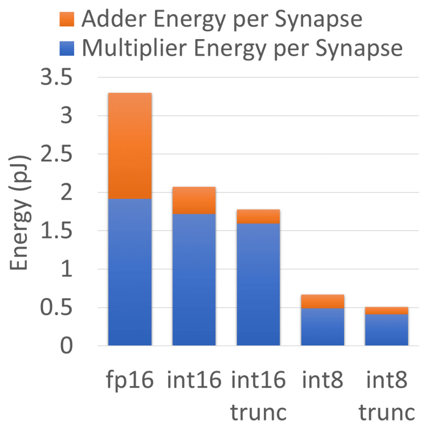


Figure 4.8: Compute energy per MAC

GoogLeNet, ResNet50, AlexNet, and VGG16. The 8-bit accelerator without codebook quantization consumes slightly less power than the 16-bit codebook quantized design since the 16-bit multiplication consumes much more power than the 8-bit multiplication. When 16-entry codebooks are used with 8-bit operands, further power savings are achieved due to the reduced memory access power.

Figure 4.10 zooms in to the compute power component of the 8-bit codebook design so that the difference between the power consumed during synapse calculation by the different designs that were introduced with Figure 4.3 and Figure 4.4 can be observed. The data labeled as premult represents the synapse calculation implementation shown in Figure 4.3a where codebook multiplexing takes place prior to multiplication. The data labeled as postmult represent the synapse calculation implementation shown in Figure 4.3b and Figure 4.4a, where codebook multiplexing takes place after multiplication with the full set of weight codebook entries. The data labeled as lut represents the synapse calculation implementation shown in Figure 4.4b, where multipliers have been replaced by a single lookup table.

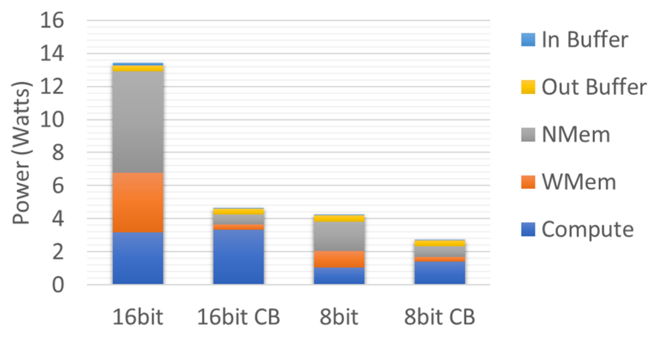
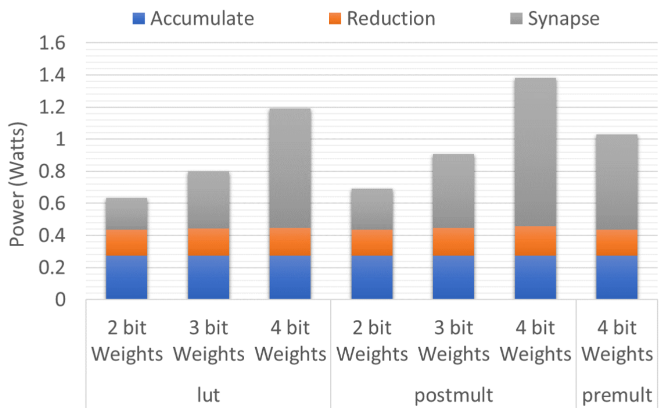
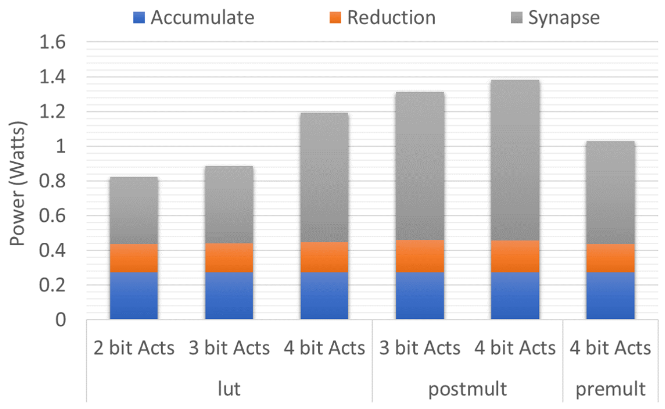


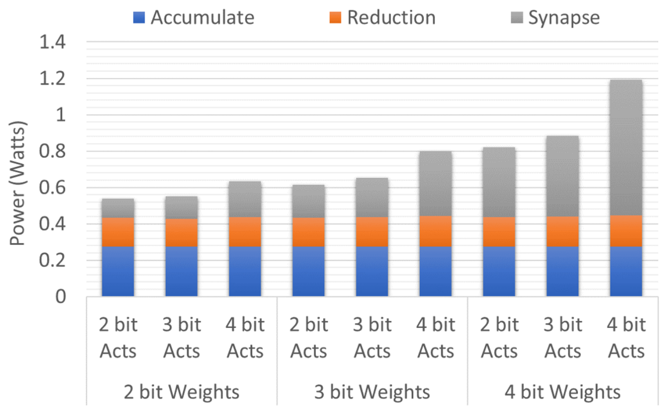
Figure 4.9: Power Breakdown of accelerators with 16-bit and 8-bit operands, with and without 4-bit codebook quantization.



(a) Variable weight codebook size with 4-bit activation indices.



(b) Variable activation codebook size with 4-bit weight indices.



(c) LUT based accelerator with various weight and activation codebook sizes.

Figure 4.10: Compute Power Breakdowns of 8-bit accelerators with weight and activation codebooks of various sizes, considering each of synapse computation architectures described in Section 4.3.1.

As can be seen in Figure 4.10a and Figure 4.10b, the premult design consumes less power than the other two when both codebooks contain 16 entries. This is because the inputs to the multiplexers are constant for the premult design while the multiplexers themselves consume more power in the other two designs. As the number of weight codebook entries decreases in Figure 4.10a, the power consumed by the lut and postmult designs decreases linearly with the size of the codebook, while the premult design is mostly unaffected since the multiplexer energy of premult is already mostly minimal. As the number of activation codebook entries decreases in Figure 4.10b, the power consumed by the lut design decreases, while the other two designs are mostly unaffected. Figure 4.10c shows the codebook scaling impact on the lut design, which demonstrates greater sensitivity to the weight codebook size than the activation codebook size but favorable scaling properties until the multiplexer energy is more dominant than the lookup energy.

The output buffer size plays a pivotal role in determining the optimal processing sequence. Operand reuse increases with larger output buffers but so does the power consumed by accesses to the output buffer. We observe that when the output buffer is sized to 4KB the trade-off between these two effects appears to be at a minimum, as is shown in Figure 4.11. Sizing the input buffers to have at least as many rows as the output buffers captures much of the reuse opportunity with the input buffer access energy remaining mostly negligible. A 4KB output buffer was used for all other experiments.

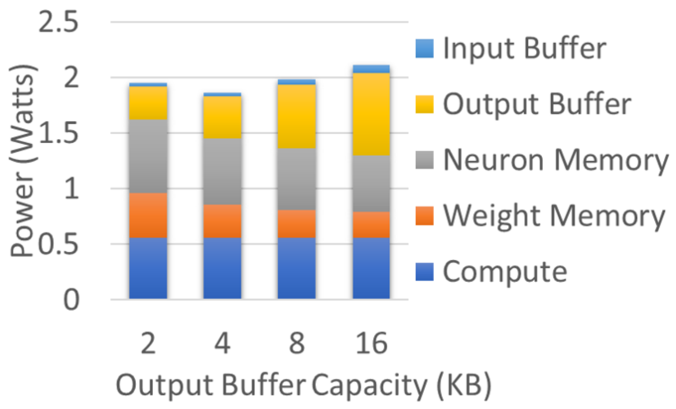


Figure 4.11: Power impact of output buffer capacity adjustment.

4.6 Chapter Summary

Codebook quantization schemes provide high compression ratios that improve memory efficiency when applied to neural network activations and weights which can be further leveraged to achieve substantial energy reductions within specialized neural network accelerators. Lookup-table based multiplication based on codebook indices provides a light-weight alternative to traditional multiplier implementations that can be easily integrated into such specialized accelerators. In comparison to a baseline 16-bit design containing 16x16 matrix-vector units similar to DaDianNao, energy reductions of better than 80% are achieved by tailoring the accelerator for codebook quantization of activations and weights. Power reductions achieved by replacing the processing element's multipliers with lookup table based multiplication account for more than 50% reduction of the compute power component, which represents a 30% overall improvement in comparison to an accelerator using codebook quantization with processing elements containing 16-bit fixed-point multipliers.

5 CODEBOOK TRAINING

I view this year's failure as next year's opportunity to try it again. Failures are not something to be avoided. You want to have them happen as quickly as you can so you can make progress rapidly.

— Gordon Moore

Users deploying previously deployed neural networks on hardware designed to operate more efficiently using a different numeric format may need to retrain the networks in order to achieve optimal accuracy under the different constraints imposed by a new platform. Ideally pretrained neural networks would be easily portable to a new set of parameters. And while some pretrained networks may be a suitable starting point for initialization of the retraining process, retraining for a more heavily constrained platform risks significant loss of accuracy. This is sometimes the case for low precision fixed point conversion, and more so for codebook based quantization. Rounding approaches are dependent upon proper selection of fixed point quantization parameters which can be approximated by statistical analysis of the floating point representation [90] to ensure coverage of a sufficiently wide range of values. However, simply rounding parameters from a network trained as floating point to a network executed as fixed point is only capable of attaining comparable accuracy up until a certain precision, that depends upon the complexity of the trained model and datasets [67, 66]. The representation requirements vary from network to network and from layer to layer within the same network. When compression is achieved using reduced precision fixed point numeric formats in place of floating point, arithmetic operations can be done more efficiently than when using floating points. However, the compression ratios achieved by simply converting to fixed point are still sub-optimal.

The magnitude of a value required for a network to operate properly and the precision with which the number is represented are not necessarily related. However, the linearity of fixed point number systems imposes a direct relationship between the range of relative magnitudes that can be expressed and the precision with which the numbers are expressed. The distributions of values taken by weights and activations are non-uniform, with small values occurring much more frequently than large values [59, 10]. High value activations have a significant impact on downstream neurons but occur infrequently in comparison to low value activations. When a neuron generates a high value activation, it is important that that magnitude be preserved, however the precision of the large activation may not be quite as significant. None the less, linear quantization schemes use as many bits as are required to represent the maximum value at the same precision as is used to represent smaller values. The fact that weight and activation values are distributed non-uniformly has led researchers to explore representations based on non-linear quantization functions that better fit these distributions. Some have proposed structured nonlinear number systems that lend themselves to being efficiently incorporated into the composition of functions involved in neural network computation [47]. However, the structure imposed by these number systems may not be ideal in all cases either, which motivates use of codebooks. Coding techniques that select representative values decouple the numeric precision from the numeric representation and thereby retain the benefits of low precision fixed point arithmetic but compress the values that are communicated and stored external to the compute elements.

Neural networks with imprecise numeric representations can suffer from some degradation of expressiveness, which is manifested as poor inference accuracy. Determining a lossy compression scheme that maintains neural network expressivity while minimizing the storage overhead is not necessarily an equivalent problem to simply selecting representative values that are close to the continuous parameter and activation values. The optimal selection of representation must minimize the loss function associated with neural network inference. Determining the best codebook based representation requires accelerator specific training techniques designed for achieving high accuracy inference under structural constraints imposed by the inference acceleration hardware. Although training for discrete quantization in neural networks remains an open problem, there have been recent studies exploring discrete quantization of weights [49] and there are well established techniques for training quantized networks [22] that we demonstrate on a small scale in this section.

5.1 Training Methodology

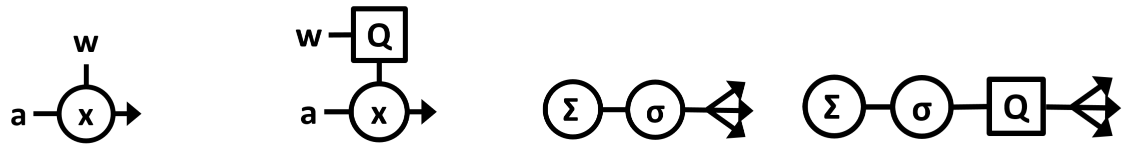
Codebook quantization can be expressed in terms of a discretization function and a quantization vector where the discretization function maps to a quantization level and the vector contains the values associated with each level, as in Eq. 5.1

$$\hat{\mathbf{x}} = \vec{\mathsf{q}}^{\mathsf{T}} \vec{\mathsf{r}}(\mathbf{x}), \tag{5.1}$$

where $\vec{q}$ is the quantization vector and $\vec{r}(\mathbf{x})$ is the discretization function that given a scalar value x returns a vector of indicator functions for each quantization level. The discretization function is not differentiable due to the discontinuities between quantization levels. Typical back propagation requires differentiability, so approximations are required in place of the actual derivatives of the quantization functions when training either the quantization function's representative values for each cluster or training continuous weights at the input of weight quantization functions.

Inference accuracy with lower precision quantization can be improved upon by retraining techniques that fine tune the weights for the lower precision discrete representation [22, 49]. One such technique, known as the continuous-discrete learning method (CDLM), involves fine tuning of continuous weights based on gradients calculated for a loss function evaluated using discrete weights. CDLM [22] only applies updates to continuous weights and does not affect the discretization function used to convert from the continuous representation to the discrete representation.

SGD based fine tuning after codebook initialization involves back propagation of gradients through the quantization functions. Since the quantization function itself is not differentiable, the quantization function must be replaced by a differentiable approximation which allows back-propagation during training. The approximation referred to as "straight through estimation" [24, 59] involves replacing the quantization function with a linear function that is easily differentiated during back propagation. We used a variant of the straight-through estimator to approximate partial derivatives for back propagation to the inputs of the quantization function with the gradient of



(a) Weighted Synapse (b) Quantized Weight (c) Activation Function (d) Quantized Activation

Figure 5.1: Quantized Neural Network Flow Elements

quantization estimated as is shown in Eq. 5.2, which essentially replaces the quantization function with its inputs during back propagation. This approximation is analogous to the continuous discrete learning method (CDLM) [22] without explicitly maintaining a discrete network during training, and instead applying the quantization function during network evaluation.

$$\frac{\partial \hat{\mathbf{x}}}{\partial \mathbf{x}} = \vec{q}^{\mathsf{T}} \frac{\partial \vec{r}(\mathbf{x})}{\partial \mathbf{x}} \approx 1, \tag{5.2}$$

This approach is well suited for integration into popular neural network toolboxes such as TensorFlow which represent the neural network as a directed graph of functions. When quantized activations are considered, it is necessary to back propagate through activation discretization in order to apply updates to all layers of the neural net. We will demonstrate in this chapter that CDLM is effective for training the weights of a network with quantized activations.

TensorFlow and other toolboxes that use automatic differentiation to apply gradient updates to neural network parameters are based on flow graphs defined with matrix multiplications, convolutions, and activation functions as nodes. In TensorFlow, the gradient calculation is performed by internally creating a second graph that mirrors the first but applying chains of gradients in the reverse direction [1], similar to typical back propagation. Both fixed point conversion functions and discrete quantization functions are easily integrated into the directed flow graph representation of the neural network, using the pattern shown in Figure 5.1.

The key to easily implementing the straight through gradient calculation in TensorFlow is the `stop_gradient` function which essentially acts as a diode within the flow graph, allowing computation in the forward direction and preventing back propagation through the `stop_gradient`. Figure 5.2 shows how the graph for quantization functions is implemented using a `stop_gradient` to cancel out the input for the forward pass but allow the gradient to propagate as if the input were included on the backward pass. The quantization function is non-differentiable with respect to the input so the gradient is back-propagated straight through to the input. Other than the quantization functions added to the flow graph, the neural network topology remains identical to the topology of the original continuous network and the graph that TensorFlow generates internally is performing back propagation through the continuous network, as is described for CDLM [22]. These same quantization blocks are used both for the activations, where the input to the quantization function is the previous layer's output, and for the weights, where the input to the quantization is the continuous weight. The fixed point conversion block follows a similar pattern except, instead of using discretization and quantization to convert from a continuous value to a

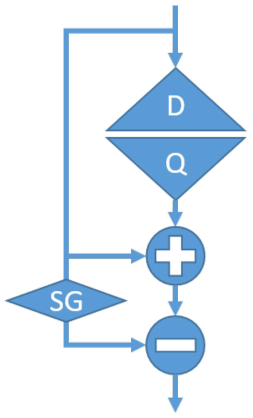


Figure 5.2: Quantization function with `stop_gradient` used to implement CDLM in TensorFlow.

quantized value, subtraction of an offset and division by a step size followed by rounding is used to more easily convert from a continuous value to a fixed point format with many more possible output values than discrete quantization. The discrete quantization blocks are followed by fixed point conversion blocks which represent the fixed point format of the codebook entries.

Another quantization approach, included in the deep compression (DC) model compression scheme [49], is designed for codebook based discretization functions in which the quantized value associated with each discrete codebook entry can be trained as a shared weight. DC only updates the codebook entries during the fine tuning SGD iterations and not the underlying continuous weights or discretization thresholds. Thus, DC decouples the thresholds of the discretization function from the representative values used for quantization. Training the quantization vector does not require backpropagation through the discretization function, so stop-gradients are not needed to implement DC-like codebook quantization of weights.

Initialization procedures that identify good discretization functions are necessary due to the rigidity of the discretization functions during the fine tuning stages of CDLM. The k-means clustering algorithm is extremely effective at mapping the distribution of a set of data points to a small set or representative data points. The k-means objective function optimizes for minimal discrepancy between a small set of representative data points with respect to the underlying data. Selection of representative codebook quantization levels using k-means provides a good starting point for network retraining by SGD. We use k-means clustering to initialize both the activation quantization functions and the weight quantization functions. The k-means analysis of activations is based on samples of the activation function outputs from several forward passes through a previously trained continuous version of the neural net topology. The data samples used for k-means in activation quantization are the same as those used for determining an appropriate fixed point representation. K-means is computationally intensive, but the results will demonstrate that it is much more effective at identifying a small set of representative quantization than fixed point representations are capable of.

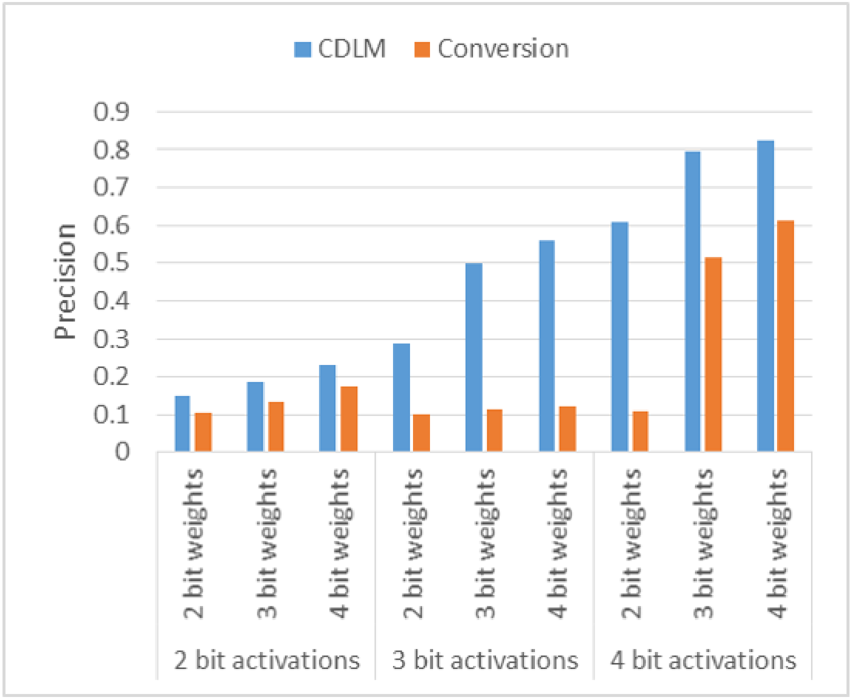


Figure 5.3: Comparison of direct fixed-point conversion to the result of subsequent CDLM for ResNet-32 network trained on cifar-10 with an initial accuracy of 90%.

5.2 Training Experiments

The CDLM is appropriate for training of low precision fixed point networks as well as those using codebook based quantization. The impact of conversion from single precision floating point to low bit-width fixed point on the network accuracy can be significant. Figure 5.3 demonstrates the impact of linear quantization to extremely low bit-width representations of the ResNet-32 trained using Cifar10. Initially, the mapping from floating point to fixed point generated by direct conversion appears to be insufficient. With retraining some accuracy is recovered at low bit-widths, however it is difficult to train the network with high accuracy using fixed point quantization at such low bit widths. As one would expect, an accuracy boost is observed as the bit-width increases. But the original single precision network was trained to 91%, and even after retraining with CDLM, we still observe a sizable gap between the retrained accuracy and the original accuracy. These results provide a point of reference for comparison with the use of discrete quantization.

Expressivity of deep neural networks is related to the precision of the network representation and calculations. As the precision of a network is decreased, the finite set of discrete hyperplanes that can be represented may not include a viable discriminator [28]. Therefore, the selection of the number of clusters, which is a prior of the k-means algorithm, can have a significant effect on inference accuracy. We explore several values of K in our experiments to demonstrate the importance of this parameter.

The results in Figure 5.4 are based on floating point codebooks and provide a likely upper bound on our expectations for these techniques when applied to fixed point codebooks. The codebooks for both the weights and activations are generated using the k-means approach described above. The accuracy evaluated immediately after applying k-means is fairly low when small codebooks are used. However k-means is quite effective for codebooks with 16 entries. To improve the accuracy of the network, we compare two retraining strategies. One of the training strategies is CDLM and the other is similar to the quantization step used by Deep Compression [49], in that instead of retraining the weights of the original continuous network, only the codebook entries are trained as if they are shared weights. The threshold values are unaffected by this training procedure, so the codebook indexes produced by the discretization function remain constant throughout the training procedure. Codebook retraining does generally improve upon the results using k-means alone, however the evaluated accuracy also generally falls short of the accuracy exhibited by using CDLM, as is shown in Figure 5.4. The results in Figure 5.4 demonstrate the ability of CDLM training to recover the accuracy after codebook quantization of both weights and activations for a ResNet-32 [55] originally trained to 91% accuracy on the cifar10 [76] dataset using single precision floating point parameters. The figure provides a comparison between the inference results immediately after codebook initialization using k-means and the results after retraining the weights using both CDLM and the Deep Compression-like technique.

The results in Figure 5.5 demonstrate the impact that fixed point quantization of the codebook entries has on the accuracy achieved when training using CDLM for

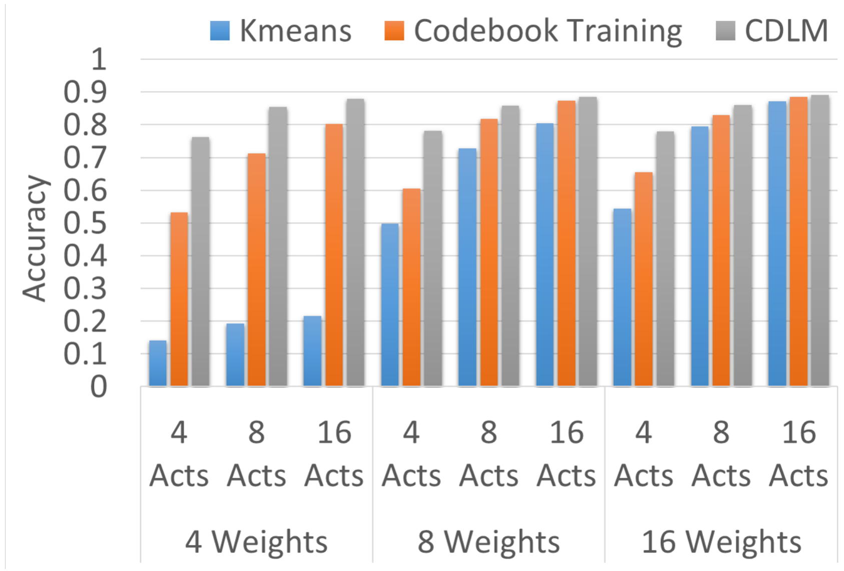


Figure 5.4: Comparison of discrete quantization retraining results.

retraining. According to this data, 8-bit codebooks appear to be generally sufficient for achieving high accuracy on the cifar10 ResNet-32. The number of activation codebook entries appears to be more critical than the number of weight codebook entries, although increasing the number codebook entries in either case is generally beneficial to accuracy.

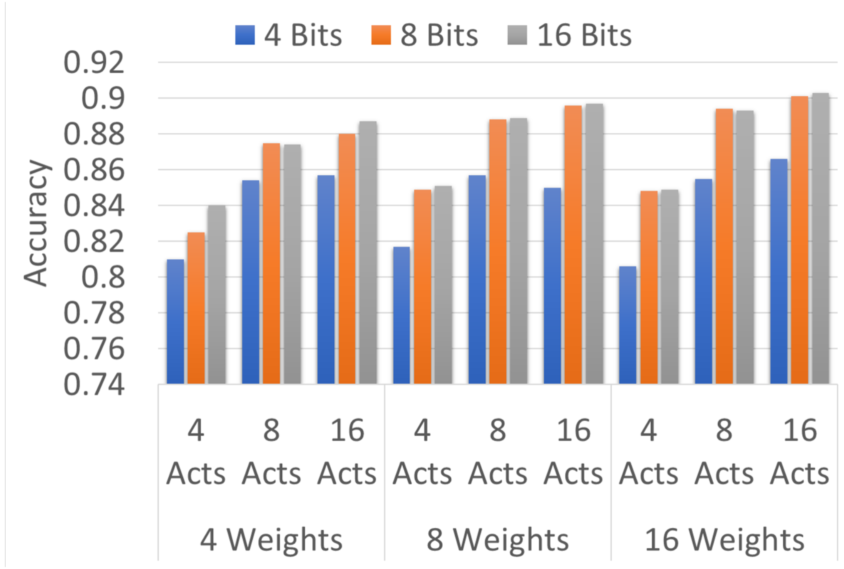


Figure 5.5: Comparison of Discrete Quantization retraining with CDLM and various codebook entry bit widths.

This result is specific to the particular network topology and dataset of the experiment. More complex datasets would likely require greater precision to achieve similar accuracy recovery. These results are encouraging and identify a viable training regime for fine tuning of neural networks with discrete quantization of both weights and activations. Future work will involve extending these experiments to larger networks with more complex data sets. Although these preliminary experiments indicate that k-means clustering provides a representative set of quantization levels, fine tuning of quantization parameters for higher accuracy remains an open question that we leave for future work. In addition, a training regime that trains the entries within the product LUTs is very interesting. For the purposes of this work, we have assumed that the values stored in the product LUTs correspond to the cartesian product of results of multiplications between the activation and weight codebooks.

5.3 Chapter Summary

The main caveat of inference using a codebook based number system is that general purpose platforms may not be well equipped to efficiently apply discrete multiplication, however an accelerator designed with built-in decoding and encoding logic can efficiently accommodate these discrete representations by using specialized lookup table structures. Hardware designed specifically for reduced precision arithmetic can impose constraints upon the parameters of the neural networks that can be deployed, requiring offline training techniques like those described above. But this is by no means the only training approach.

While conducting these experiments other techniques were considered as well but were omitted because they either converged too slowly, diverged entirely, or converged to lower accuracy than the approach described here. These included attempts to train weight and activation codebook entries as well as training the underlying weights. The different combinations of training weight codebook entries, activation codebook entries, and underlying weights amount to seven different training approaches. Ultimately, the technique that demonstrated most favorable convergence properties turned out to be when the codebook entries were set using k-means clustering to select representative centroids of each layer prior to retraining and subsequently training the underlying weights with fixed codebook entries, which is equivalent to CDLM.

We also included experiments based on the same approach where instead of training underlying weights, the codebook index for each weight was fixed, the underlying weights were discarded, and weight codebook entries were retrained. This approach is similar to the one used by Deep Compression, so the results were included in this chapter as a basis for comparison, although the accuracy achieved using this technique was not better than CDLM. Although these results provide evidence that CDLM would be the preferred approach, the results are by no means conclusive in this regard. So the comparison between Deep Compression and CDLM remains an open question, which may have important implications for companies that adapt aspects of the Deep Compression techniques within production training regimes.

SGD based approaches for setting the activation codebook entries remains an open problem with regard to this dissertation, since my attempt to do so were ineffective. When training for the LUT based accelerator described in section 4.3 it would be interesting if the discretization functions and codebook entries could be trained directly using SGD as this would represent a departure from the traditional linear neural network algorithms, however attempts to do so were similarly ineffective.

6 TEMPORAL SYNAPSE WEIGHTS AND ACTIVATIONS

The really good idea is always traceable back quite a long way, often to a not very good idea which sparked off another idea that was only slightly better, which somebody else misunderstood in such a way that they then said something which was really rather interesting.

— John Cleese

Within tiled accelerators with collocated weight storage, on-chip interconnect energy can account for a sizable slice of overall accelerator energy consumption. This section considers a technique for energy reduction within the accelerator's on-chip interconnect using an encoding scheme that reduces the switching activity factors associated with on-chip communication. The encoding is based on a temporal coding scheme which associates timing of signal toggles with data. More efficient communication is achieved when more bits can be expressed by a single toggle, however latency constraints imposed by the target system restrict the protocol design space and limit its energy efficiency. The energy savings achieved using this technique is evaluated using a tiled accelerator design similar to the one described in Chapter 4, demonstrating 10% reductions in overall energy consumption for a set of representative neural network benchmarks.

6.1 Introduction

Wire energy is predominantly dynamic energy, which is related to the length of the wires that signals traverse, as well as the amount and diversity of data moving over the wires. Within tiled accelerators with large local weight stores within each tile, wire distances both within tiles and across tiles can be quite large with high dynamic power dissipation. One approach for mitigating dynamic energy over these wires is to use data encoding protocols that reduce the activity factors associated with communication [128, 9, 100]. In this chapter, a temporal coding technique is integrated into the interconnects of a tiled accelerator in order to lower the interconnect power component.

The temporal coding scheme translates the timing of the signal toggles to codewords. Temporal communication protocols are in general most effective when latency constraints are sufficiently high to create a large time window for the transmission. Therefore, workloads with communication patterns that either provide sufficient slack to accommodate extended delays or involve large bulk data transfers that can be sent in parallel are particularly well suited for temporal codes. Domain specific accelerators for applications with these characteristics can therefore integrate specialized temporally coded interconnects designed to exploit these communication patterns.

The communication patterns associated with neural network processing are dependent upon characteristics of the computation schedule such as stationarity and mapping. Schedules for efficient processing of DNN's typically reduce data movement by exploiting data reuse opportunities. This can introduce slack into the access patterns for reused data items, which improves tolerance for longer latency communication protocols. Elevated latency constraints represent an opportunity for more energy efficient temporal protocols. Selecting processing schedules that maintain high computational throughput throughout the duration of a temporally coded transmission place less strain on weight memory, thereby providing energy efficiency benefits realizable even within a system without a strictly provisioned power budget.

Encoding and decoding of temporal codes adds to the energy overhead of utilizing these protocols, which limits the systems and interconnect links for which temporal coding can be beneficial. Similar to biological neural networks, additional energy efficiency could potentially be achieved by integrating the temporal format of the communication protocol into computational elements capable of processing information in the temporal format. Tight integration of the encode/decode logic into functional blocks at the endpoints of the interconnect amortizes the energy overheads associated with the temporal transmission as computational progress. Domain specific accelerators can therefore integrate custom receivers and/or transmitters that bridge temporally coded interconnects specialized data paths. However, due to the data reuse typically exploited by neural network accelerators, such tight integration would be inefficient since each retrieval of data would require a remote retransmission instead of a local cache access.

We herein consider the design space of neural network accelerators that utilize temporal codes to improve communication efficiency. The encoded formats considered take advantage of DNN tolerance for low precision numeric formats and codebook based quantization. For fixed point formats the temporal encoders and decoders would be more easily integrated into arithmetic units. However, more extreme codebook quantization schemes provide higher compression ratios with higher accuracy inference than would be possible with linear quantization.

Decoded values that are cached close to the processing element need not be retransmitted. Minimizing retransmission by increasing data reuse can improve both performance and efficiency of DNN processing. The optimal processing schedule for performance is tightly linked to the transmission latency, the cost of retransmission, and the reusability of decoded values. Schedules that do not optimally exploit reuse opportunities can cause high power consumption if not properly accounted for by the hardware. Accelerators that integrate temporal codes within the interconnect reduce the peak power by reducing the maximum activity factors associated with communication. Furthermore, long latency codes impose a minimum latency restriction on each data transmission, and in turn impose a preference for high local data reuse patterns that achieve high performance despite the limitations imposed on storage access rates.

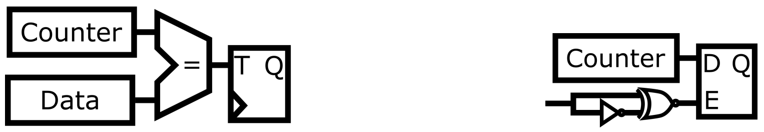
6.2 Temporal Coding

Temporal coding schemes encode multi-bit sequences as a signal toggle placed in time. Each clock cycle during a transmission is associated with a different symbol. Encoding a given symbol involves driving a transition onto the interconnect within the corresponding clock cycle. Protocols with longer latency transmissions can encode more symbols per transition and can therefore achieve higher energy efficiency, due to lower communication activity factors. Therefore, systems with communication patterns that are tolerant of long latency transmissions are good candidates for integration of interconnects communicating with temporal codes. However, due to the energy efficiency provided by the encoding scheme, the transmitters and receivers can be run at a higher frequency to provide lower communication latencies, without transmission overhead overshadowing the power reduction provided by temporal coding [100].

The encoding and decoding to and from the temporal domain and the binary domain is performed by transmitters and receivers comparable to those given in Figure 6.1. The basic temporal code transmitter (Figure 6.1a) comprises a counter, a digital comparator, and a toggle flip-flop. The basic temporal code receiver (Figure 6.1b) is a storage element with two input terminals: one for a counter value and one for the temporally encoded signal. When an encoded toggle arrives, the contents of the counter at that moment are captured within the receiver to become the output of the receiver.

The transmitters and receivers introduce some translation overhead which should be low in comparison to the wire energy to justify integration of temporal codes into the interconnect. Since wire energy is proportional to wire distance, there is a threshold average interconnect distance at which temporal codes are appropriate. When measured using a 45nm process, the wire distance at which temporal codes are beneficial is about 1mm [100]. The interconnect energy becomes more significant with progressive process generations [69], and the translation overhead would be expected to decrease more quickly with future process generations than the wire energy. So this threshold distance is also expected to decrease at smaller scales, with respect to the nominal gate length.

There is an opportunity in some ASICs with compatible communication patterns to efficiently integrate the transmitters and receivers into functional blocks that would either accept temporal operands or output results in temporal format and thereby productively

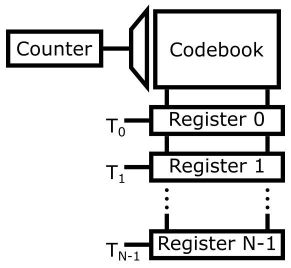


(a) Transmitter (b) Receiver

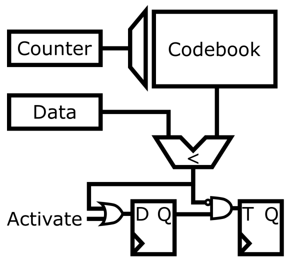
Figure 6.1: Basic Temporal Transmitter and Receiver Designs.

amortize data conversion overhead as computational progress. For interconnects used to communicate quantized values, the codebook look-ups and their implications can be integrated as a part of the temporal receivers. However, in cases where received values are stored for reuse, as is often the case with neural networks, it is preferable to retain a compressed format in a local cache.

The codebook is a single ported lookup table SRAM addressed by an encoded index corresponding to a quantization level. When the temporal sequence counter is used to generate look-up table addresses, each cycle in the temporal sequence can be associated with a different quantization level represented by values with larger bit-width than the counter width. The temporal decoding receiver based on codebook sequencing is similar to the one in Figure 6.1b except storing codebook entries instead of counter values. When a toggle is received, the value at the address indicated by the counter value at that moment is stored into a register. This receiver design is easily extended to one that accepts multiple parallel temporally encoded signals, as is shown in Figure 6.2a. The wide receiver is implemented as multiple registers each with its own temporally encoded input wire and sharing the lookup table output as the data input.



(a) Receiver



(b) Transmitter

Figure 6.2: Codebook Temporal Transmitter and Receiver Designs.

Codebook based transmission involves determining the appropriate quantization level given an input value. Unlike the basic transmitter in Figure 6.1a where the set of possible data values and the set of counter values are equivalent, the quantizing transmitter maps each level to an interval of possible data values. Since encoding is based on intervals that may be of any size, the values stored in the codebook must be the thresholds between adjacent quantization levels. Identifying the quantization level for a given data word involves two comparisons with the upper and lower thresholds of each quantization level. Assuming the temporal code sequence is monotonic, these comparisons can be performed by a single comparator within the temporal encoding transmitter, such as the one shown in Figure 6.2b. This transmitter design is easily extended to a wide transmitter containing multiple comparators operating in parallel that share the same counter and codebook.

The contents of the codebooks in Figure 6.2 can be designed to meet application specific needs. Within the context of the processing elements of a neural network accelerator designed to support encoded activations, the temporal code and the associated codebook are defined according to the activation quantization function, just as was the case for activation encoding in Section 4.3.

When the encoded signal is to be used for communication purposes only, then every possible bit sequence must be represented by the counter sequence, but the order of the counter sequence need not be incremental. The counter finite state machine can therefore be a grey code counter or an incrementer or a decrementer or something else. In the following section we develop the microarchitecture of the processing element designed for codebook based quantization of both activations and weights, where the weights and activations are communicated to the processing element temporally and decoded to indices.

6.3 Implementation

6.3.1 Temporal Code Receivers

We assume temporal codes that encode four bits per transition and therefore have a latency of $2^4 = 16$ cycles per transmission. The temporal code receivers for weights are double buffered to avoid pipeline stalls. The weight SRAMs collocated with each processing element are banked such that each processing element can actively receive four transmissions concurrently into four different receiver buffer pairs, one from each of the four banks. The high weight bandwidth provided by banked accesses improves tolerance to the temporal transmission latency within network layers lacking reuse of weights.

Achieving high utilization of the processing element throughout the duration of the temporal transmission requires sufficient reuse opportunity, which depends on the neural network topology and scheduling scheme. The output/input buffer capacities are provisioned to allow for a continual flow under certain conditions, however when reuse is not possible, the pipeline stalls until the next set of weights is ready. For fully connected layers, the opportunity for weight reuse is only available when processing batches of multiple inputs concurrently. A batch size of four is sufficient to maintain a continual flow with fully connected layers. The same batch sizing applied to convolutional layers maintains a continual flow, however due to weight sharing in convolutional layers there are alternative schedules that make more efficient use of weights.

The activations are also encoded temporally. The activation receiver stores the codebook indexes within a small buffer local to the processing element. The input buffer is accessed with each iteration of synapse calculation to index into the lookup table containing the set of outputs to select for that activation. The activation reuse pattern must both accommodate reuse of the same weight set with multiple activation sets and reuse of the same activation sets with multiple weight sets. An input buffer with at least eight entries would be necessary to enable a continual flow, however a 32 entry buffer is used to improve flexibility. The input buffer has a capacity of 256 bytes.

6.3.2 Sequencing

Weight codebook indexes make multiplexer selections to output calculated synapse products which are then accumulated. The output of synapse calculation is subject to adder tree reduction to calculate the neuron dot products. The combined partial sums output by the adder tree are accumulated with partial sums previously stored to the output buffer. The size of the output buffer is set to accommodate enough neurons to support active calculations without stalling due to the activation transmission latency.

The partial sums stored in the output buffer are 32 bits each. Output buffer space is provisioned to be large enough for all output sets to be fully accumulated before being transmitted to the global neuron memory. Each of the 16 outputs of the processing element are stored within a 512 byte SRAM which provides sufficient capacity for reuse of four input sets throughout the duration of transmission of a subsequent four input sets while also buffering a full set of completed accumulations pending transmission. With a large output buffer, many outputs can be processed for the same set of input activations before advancing to the next input set. This both reduces the number of times that the same inputs must be retrieved and favors re-retrieval of read-only inputs over spilling partial sums. The partial sums are not compressed so they are more expensive to evict from the output buffer and re-retrieve than input activations. Therefore, provisioning of local buffer space to partial sums (as an output buffer) is assumed to be more efficient than using this storage space for inputs, which can be communicated at lower energy expense.

When accumulations are sequenced to make use of input activation temporal locality, the sequence can be represented as a set of nested `for` loops as in Figure 6.3, which describes the sequence of calculations used for a fully connected layer. This sequencing essentially allocates the output buffer to a set of neurons, loads all input activation sets accumulating synapses between each input activation and all neurons within the output buffer, applies the activation function once accumulation of all synapses completes, then allocates the output buffer to a new set of activations and repeats until all neurons have been processed. This sequencing results in reretrieval of the full set of activations as many times as the output buffer is reallocated ($\lceil \mathtt{Tnnn}/\mathtt{Tnn} \rceil$). This same schedule is performed by processing elements for different non-overlapping subsets of the output neurons.

```
// Tnnn: the number of neurons
//       calculated by this unit
// Nb:   the samples per batch
// Ni:   the total number of inputs
// Tnn:  the output buffer capacity
// Ti:   the per cycle input width
// Tn:   the per cycle output width

<Load all codebooks for layer>
for(nnn=0; nnn<Tnnn; nnn+=Tnn)
  for(ii=iii; ii<Ni; ii+=Ti)
    <Load Next Set of Inputs>
    for(nn=nnn; nn<Tnn+nnn; nn+=Tn)
      <Load Next Set of Weights>
      for(s=0;s<Nb;s++)
        //Iterate batch samples
        for(i=ii;i<Ti+ii;i++)
          for(n=nn;n<Tn+nn;n++)
            A=activation[s][i];
            W=synapse[n][i];
            sum[s][n] += LUT[A][W];
  for(nn=nnn; nn<Tnn+nnn; nn+=Tn)
   <Apply activation functions>
```

Figure 6.3: For loops with temporal locality of weights across batch.

The weights of the fully connected layer are represented by a two dimensional array with dimensions corresponding to the inputs and outputs. Weight data is stored in memory such that each address corresponds to a row of 256 weights representing the synapses between one input set containing 16 activations and one output set containing 16 neurons. Multiple samples from a batch of inferences are processed each time that a new weight set is read in order to maintain high utilization.

When the batch based inner loop scheduling approach is taken for convolutional layers, the scheduling approach taken for fully-connected layers is directly applicable only if there are sufficiently many output channels to saturate the output buffer with neurons from a single coordinate. If the layer does not contain enough output channels to saturate the output buffer, then it can be worthwhile to allocate the output buffer to the full set of output channels at multiple adjacent coordinates. This gives rise to non-uniform temporal locality across the output buffer with respect to different activations. Since not all convolutional layer neurons residing in the output buffer at a given time correspond to the same coordinates, not all neurons in the buffer complete at the same time and the activation functions are only applied to a subset of the output buffer at a time, thus fewer rows of the output buffer may need to be reserved for activation encoding of some convolutional layers with few output channels than for fully-connected layers.

Sharing of convolutional kernels provides opportunities to exploit temporal locality of weight retrievals across multiple different neurons. Therefore, scheduling schemes that do not rely on batches for weight reuse are capable of retaining throughput under the conditions imposed by temporal codes as well. These schemes would only schedule from as many channels as are contained in one output set (16 neurons) at a time, since the output sets only pertain to a single location.

The differences between these two approaches to mapping and scheduling convolutional layers fall into two classes of output stationary, classified by Eyeriss [15] as multiple output channel (MOC) and multiple output position (MOP). When applied to our tiled architecture, the mapping of the MOC scheme requires that the feature maps be divided along spatial dimensions to different tiles. When using a MOC scheme, all kernel weights must be stored within each tile and multiple batches remains the primary means of weight reuse. The mapping of a MOP scheme requires that only the channels that are processed by a given processing element be stored within the tile's weight memory.

6.3.3 Activation Transmission

The temporal transmitter is implemented as a single row of comparators, as in Figure 6.2b, with the same width as the output buffer row. The storage side decoders simply convert from temporal format to a codebook index that can be stored in the compressed format. The storage side encoders and decoders are therefore implemented as in Figure 6.1 with a single threshold codebook shared by an array of transmitters.

6.4 Evaluation

When using codebook based quantization, much lower bit widths are required for storage and communication than for compute, so a significant energy and area reduction can be achieved. The impact of quantization on the energy required to read a 16x16 matrix of weights from a 45nm SRAM with the same capacity as the DaDianNao weight memory is shown in Figure 6.4. The Figure demonstrates an energy reduction greater than the compression ratio that is achieved by adjusting the width of the read port for the compressed data format without adjusting the total capacity of the memory. A proportional capacity adjustment reduces energy only slightly since the H-Tree energy is not significantly affected by capacity adjustments. This is because although the area of

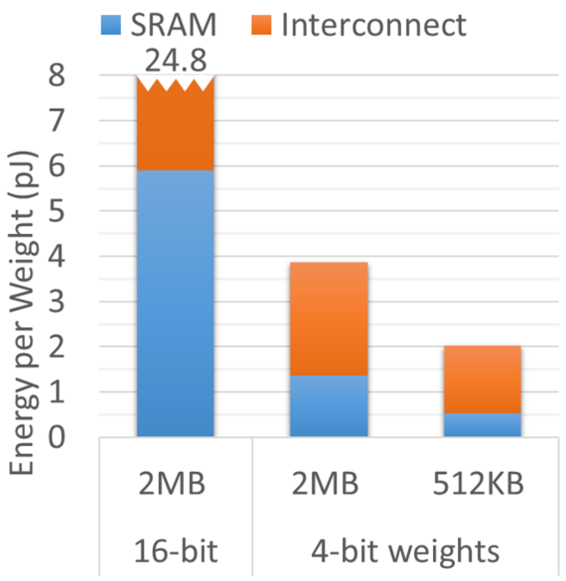


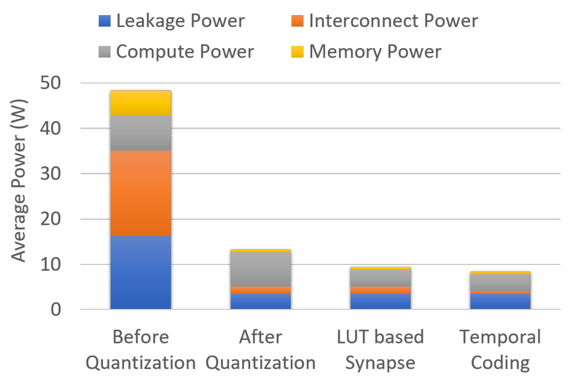
Figure 6.4: Cacti based Weight Memory Access Energy per row for different number formats, different SRAM capacities, and both binary and 4-bit temporal code.

the SRAM is roughly proportional to the area, the Manhattan distance traversed by the H-Tree wires (and their associated capacitance) is roughly proportional to the square root of the SRAM capacity. In all cases using binary encoding the H-Tree remains the dominant energy component, representing 65%-75% of the access energy. Therein lies the opportunity for temporal codes to reduce energy, as is illustrated in the figure.

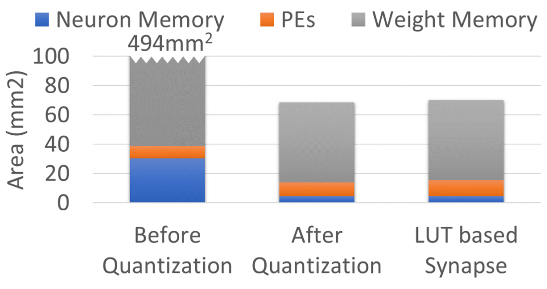
Transistor scaling trends have had less of an impact on interconnect energy than on data-path and SRAM energy, so the discrepancy between the array access energy and communication energy in Figure 6.4 is even more severe at more modern process nodes. Based on the data provided in [69] we would expect a reduction in compute and SRAM energy by roughly a factor of 6 but only a factor of 2 reduction in communication energy when comparing identical chips at the 40nm and 10nm nodes. This provides motivation for low energy communication techniques such as temporal coding which will be described in the following section.

6.4.1 Progressive Optimization

To demonstrate the impact of each energy reduction optimization, we start with an unreasonably large design modeled after DaDianNao. DaDianNao was implemented using a 28nm process and eDRAM based memories which are much more dense than SRAM. Our analysis is based on a 45nm process using SRAM, yet for the purpose of comparison we show data regarding a hypothetical DaDianNao implementation which consumes nearly 50W of power, over 15W of which is leakage power. The energy data is collected based on a schedule designed for the implementation using temporal coding in order to make a direct comparison, so this does not represent the peak power consumption for this design. None the less, we can observe a 75% power reduction resulting from resizing the neuron and weight SRAMs according to the quantization compression ratio of 4:1, which significantly reduces the memory and leakage power



(a) Average Power Results based on AlexNet, ResNet32, and VGG16



(b) Area Results

Figure 6.5: Progressive impact of each optimization based on inference with batch size 4.

but has a negligible effect on the compute power which becomes the dominant power component. Resizing the memories also has a significant effect on the area of the design, which is reduced by over 85% to an area comparable to the 28nm implementation of DaDianNao, as can be seen in Figure 6.5b.

Replacing the 256 16-bit fixed point multipliers per processing element with 16 lookup tables each causes a 16% increase in the area of the design but reduces the compute power by over 50%, representing a 30% decrease in the overall power as compared to the design with quantized weights and neurons using multipliers. The final bar demonstrates how using temporal coding further reduces the power consumption by about 10%. Compute power and leakage remain the dominant power components accounting for 90% of power consumption.

6.4.2 Performance Evaluation

The computational throughput of a processing element using temporally coded activations can be impacted by the latency associated with encoding and decoding the activations. However, due to concurrency within the processing element pipeline, this latency does not cause stalling for neural network layers containing sufficiently many synapses per activation and synapses per neuron to continue accumulation throughout the duration of the activation decoding and encoding. A network layer with too few activations per neuron will stall during encoding having completed accumulation for a new set of neurons while the previous set is still being encoded. A network layer with too few neurons per activation will stall during decoding having completed all accumu-

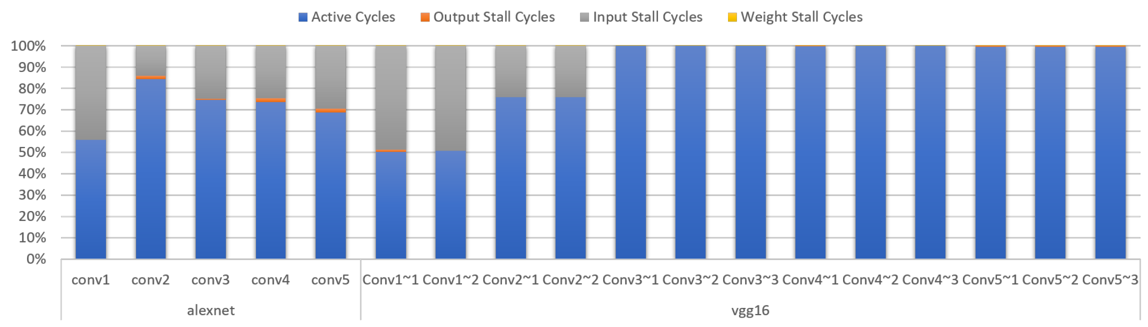


Figure 6.6: Breakdown of cycles spent active or stalling for inference performed on a batch of size 4 for a set of convolutional layers based on a naive for loop based scheduling scheme.

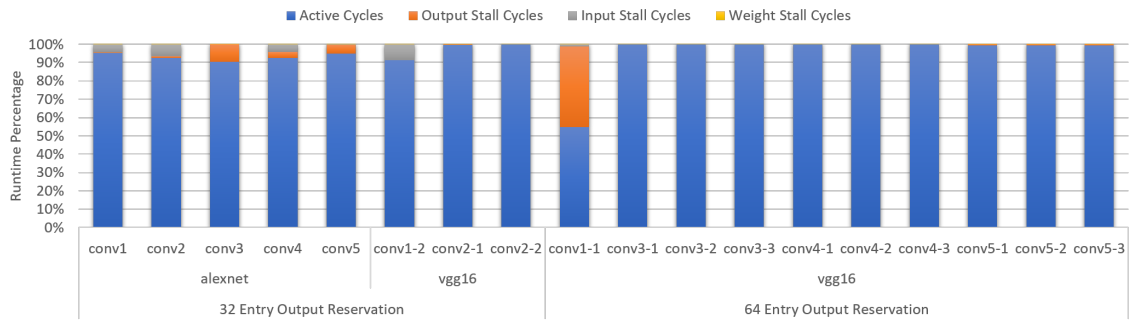


Figure 6.7: Breakdown of cycles spent active or stalling for inference performed on a batch of size 4 for a set of convolutional layers. The better performing run of either a reservation of 32 or 64 output buffer entries for neurons pending activation.

lations for the previous set of activations while the new set is decoding. Constructing a schedule that maintains this property for all neurons is not always possible, especially for convolutional layers with few output channels.

Figure 6.6 represents a typical for loop based scheduling scheme and demonstrates the performance degradation that occurs within layers where input activations are repeatedly reretrieved. Figure 6.7 demonstrates a substantial improvement for most of the layers impacted by the naive scheduling scheme. The difference is that the schedules in Figure 6.7 will partially release completed accumulations from the output buffer when only a subset has completed due to differing positions of neurons simultaneously residing in the output buffer, as was described in Section 6.3.2. There is a total of 128 entries in the output buffer. When a layer has enough output channels to saturate half of the output buffer, the other half can be allocated to neurons pending activation. Layers with few output channels sometimes benefit from a smaller allocation for pending activations. Two sets of experiments were run with pending allocations of size 64 and 32 entries and the better performing of the two for each layer is shown in the figure. All but one layer achieves at least 90% efficiency.

6.4.3 Peak Power Enforcement

When the accelerator is presented with a suboptimal schedule, there is a significant difference in behavior between the quantized accelerator with and without temporal coding. Figure 6.8 demonstrates the impact on power when using a schedule for only a single sample as opposed to a batch of four samples. A bandwidth limitation in the temporally coded design reduces the performance by a factor of four for the single sample inference. The accelerator without temporal coding executes a batch size of one with high performance and there is a significant interconnect and memory power penalty. The accelerator with temporal coding, on the other hand, stalls and thereby prevents these high power states

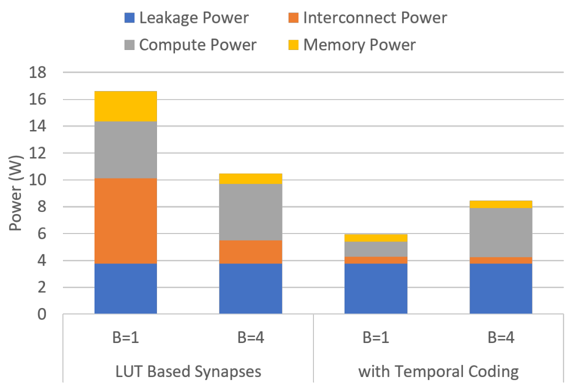


Figure 6.8: Power components of accelerators with and without temporal coding for various batch sizes.

from occurring.

Ultimately, the performance degradation that is occurring here is a result of a design decision to limit the weight retrieval bandwidth to one new weight every four cycles. This is intended to impose a preference for schedules that maintain low power by reusing weights. The temporal codes are primarily a means of making use of link bandwidth for the purpose of energy reduction rather than extracting higher performance which exacerbates power consumption. Running the temporally coded links at a higher clock frequency would be one means of achieving higher bandwidth without incurring as much power consumption as a binary coded baseline, due to the low activity factors of the protocol that are the basis for their energy efficiency.

6.5 Chapter Summary

In this work we have described a neural network accelerator designed for synaptic computations using codebook based compression of activations and weights. The design utilizes an output stationary computation sequence to minimize the on chip traffic that is not encoded. Temporal coding enables low energy communication of encoded activations. The design demonstrates energy reductions in comparison to a 45nm DaDianNao baseline of better than 80% while enforcing a peak power by preventing execution schedules that would result in rapid memory accesses in favor of schedules that efficiently make use of memory. The performance degradation caused by temporal coding is only about 4% in comparison to uninhibited performance using the same schedule, due to the schedule's adherence to the data access rates imposed by the communication protocol.

7 SIMULATION INFRASTRUCTURE

We generalize from one situation to another not because we cannot tell the difference between the two situations but because we judge that they are likely to belong to a set of situations having the same consequence.

— Roger N. Shepard

A growing set of machine learning applications are dependent upon neural networks and the topologies of these networks are evolving with time. There is a need for a design space framework for exploration within the domain of neural network accelerators in which different neural network topologies can be evaluated at various design points within the space. Such a framework is useful for three purposes:

1. Exploration of different accelerator architectures within their design phase.
2. Analysis of micro-architectural implications for different accelerator architectures.
3. Evaluation of deployment strategies including mapping of neurons and their calculation schedules within particular accelerator architectures.
4. Design of new neural network topologies tailored to deployment on particular accelerator architectures.

An accelerator data-path consists of memories, buffers, caches, registers, interconnects, and processing elements. In order for an accelerator to fully utilize its available compute throughput it must have sufficient memory bandwidth to support the data access patterns of the processing schedule. This does not mean that the memory bandwidth and compute bandwidth must be equal since operands can be reused or passed systolically. However, insufficiently provisioned bandwidths can become significant bottlenecks and therefore must be carefully considered within the data-path model. These bandwidth interactions are sufficient for high level analytical modeling in early accelerator design stages [57], however fully capturing the interactions between neural network topologies, scheduling schemes, and architectural parameters requires more detailed models.

This dissertation has involved several iterations of performance model development, all following common design principles but representing the accelerator with varying degrees of detail. The performance simulation framework used in Chapters 4 and 6 contains more detailed cache modeling than the model used in Chapter 3. The version of the simulator used in Chapters 4 and 6 will be referred to as *Tempura*, and the other simulator version will be referred to as *Dim Sum*. All of the simulators use a timing model based on interval simulation [38], which provides high performance cycle level modeling of behaviors within the accelerator. The simulator models and monitors the data path stalling conditions at each of the accelerator interfaces in order to identify stall events. A separate clock is maintained for each stage of the accelerator pipeline. A stall event is indicated when a stage's clock advances beyond a dependent stage's clock. When a stall event occurs the dependent clock advances to the end of the stall and simulation continues. By logging the stall events, it is possible to construct latency breakdowns similar to CPI stacks [31].

Although this chapter sets the groundwork for design space exploration, a truly comprehensive design space would be intractable so certain assumptions must be made to establish a reasonable space that is both broad enough for coverage and narrow enough for tractability, while still including leeway for elements of architectural novelty. One of the ways that the simulation framework accomplishes this task is by not being bound to a particular ISA. Front end bottlenecks would emerge with instruction sets where instructions exert fine grained control over the data-paths. For example, the slight performance difference between Cambricon [91] and their DaDianNao-like design point occurred when processing convolutional layers that required more fine grained control flow between matrix operations. In this case, the DaDianNao-like baseline uses a single instruction to describe an entire DNN layer and maintains special purpose logic to facilitate layer execution. Cambricon uses a more expressive instruction set with matrix based instructions and in doing so can support many more layer types than DaDianNao. But for the layer types that DaDianNao does support, it is slightly faster than Cambricon since it is better at avoiding bubbles in the instruction sequence. There are many ways to implement the front end of an accelerator, but the front end accounts for only a very small portion of the accelerator logic, which can be engineered to optimally support the data-path requirements. Data-path performance is significantly influenced by factors such as the number of compute units, their access semantics, and the on-chip buffer hierarchy, so these elements are carefully modeled.

In general, the performance models do not include control mechanisms. Another example of an important control mechanism which is idealized by the simulator is data prefetching. Due to the highly specialized nature of these accelerators, accelerator control flow is highly predictable. Within an architecture where control hazards are not a significant concern, non-speculative prefetching based on a decoupled access-execute model [126] is readily integrated [12, 109, 64, 13]. Using a bandwidth-based memory mode, the idealized data access model assumes that as soon as buffer capacity is available, bandwidth is the only limiting factor in loading the next set of data items into the buffer. This is of course a simplification that overlooks the impact of non-deterministic memory access latencies and reactive access initiation. However, with sufficient buffer capacity, the memory access latency is easily amortized. This is of course not a significant issue when all weights remain in on-chip storage, which provides much lower access latency. However, if temporal codes are integrated, this latency becomes more significant, so the performance model that supports temporal codes must consider stalling conditions that occur as a result of the transmission latency.

7.1 Tempura

The simulation framework used in Chapter 4 and Chapter 6 has come to be known as Tempura, due to its support for modeling the latency of temporal codes.

7.1.1 Architecture Specification

Processing elements are modeled as in Figure 4.7b with an input buffer, weight buffer, output buffer, and a processing element. The architecture specification is primarily based on a set of parameters that define the dimensions of the matrix-vector multiplication unit and its buffers. The input width and output width of the unit are used to infer

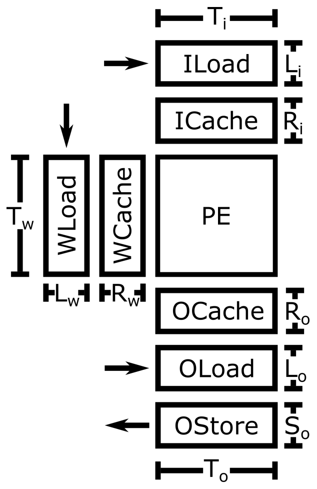


Figure 7.1: Processing Element Node model

the number of multipliers and the adder tree dimensions, as well as the width of the weight registers. Additional parameters define the number of rows in each of the buffers feeding the input, output, and weight registers. The width of each row of these buffers is assumed to be equal to the width of the corresponding registers.

Each buffer contains one cache of data that can be reused and one or two communication FIFOs where data can reside only during an active transfer and is eliminated once consumed. These FIFO's also act as prefetch buffers, enabling early retrieval of data. The input and weight buffers are read-only so each is associated with one communication FIFO used for loading data. The output buffer needs to store the calculated neuron outputs and therefore requires a store buffer as well. The output buffer also may need to load values in some designs where partial sums or biases need to be retrieved from sources external to the processing node in order to accumulate the full neuron membrane potential. Capacity and latency parameters are used to describe the behavior of these communication FIFOs.

7.1.2 Layer Specification

Layers are described in terms of a small set of parameters including the number of input channels (N_i), number of output channels (N_n), batch size (N_b), the spatial dimensions of output (N_x, N_y), the spatial dimensions of the convolutional kernel (Kx, Ky), and the stride (N_s). Fully connected layers are expressed as a special case of a convolutional layer with all spatial parameters set to one. The simulator loads sets of layer specifications from a csv file containing descriptions of all layers for a set of neural networks.

7.1.3 Loop Specification

The loop specification describes the range of indices for which the current calculation applies. These are constructed at runtime and associated with the scope of each loop within the computation schedule. The specification is expressed in terms of a range of values and therefore defines a contiguous subset of a neural network layer to be processed within the given loop iteration. The order of the calculations within this

Variable	Description
b	batch
n	output channel
nx	output X coordinate
ny	output Y coordinate
i	input channel
ix	input X coordinate
iy	input Y coordinate
kx	kernel X coordinate
ky	kernel Y coordinate

Table 7.1: Loop Specification Variables

region of a neural network layers is determined according to a scheduling specification. The set of variables associated with a buffer pertain to the indices of the data structure associated with the buffer. The set of variables used to describe the loop specification are shown in Table 7.1. Some of these variables are dependent upon one another and must always remain consistent.

7.1.4 Schedule Specification

The simulator facilitates scheduling given a set of scheduling guidelines for each buffer, called the *Schedule Specification*, which is represented as a set of ordered lists of data structure dimensions defining the order of nested loops for each buffer. The simulator sequences computation according to buffer contents in order to exploit locality in a manner that is not necessarily easily expressed in terms of nested `for` loops iterating over indices of the neural network's multi-dimensional data structures. The computation sequence is loop based, but instead of having each loop represent a variable, each loop represents a buffer's contents and all index variables that the buffer contents encompass. In many cases, a schedule may easily reduce to one expressed in terms of component variables, however the representation in terms of buffer contents expands the set of schedules that can be expressed beyond these. The layout of data within each buffer is based on the buffer's own loop ordering specification and the format expected by the processing element side registers, which may have a vector width of many elements that is assumed to match the buffer read/write width.

The processing element's local storage registers also have loop order settings that dictate the order in which data in the buffers are accessed. For processing elements with wide registers that process multiple operands in parallel, the format of the data within the registers is based on the schedule specification for the registers. This data ordering affects the format that data must be stored and must be reflected within the format of each row of data within the buffers. The loop specification for a buffer must represent a multiple of the *row format* as is defined by the processing element register.

For workloads with batch sizes that exceed one input sample, all data within a row pertains to a single sample from the batch. Thus, for an input register, the set of variables defining the row format includes i, ix, and iy. For an output register, the set of variables defining the row format include n, nx, and ny. For an adder tree based processing element, the weight register must contain the set of weight mappings from each input activation in the input register to each output value in the output register. Therefore, the weight row format is derived from the input and output row formats.

The assumption that either the input or output register formats pertain to multiple channels at a single location simplifies the weight row format since it need not contain duplicate weights at a given time. Furthermore, when both the input and output register row formats pertain to only one location in the input and output volume respectively, the alignment of weight data never varies. Enforcing that either the input or the output register format should remain 1 along each dimension allows the weight register contents to be expressed in terms of i, n, kx and ky, where i is the same as the input register, n is the same as the output register, and kx and ky are $\max(ix, nx)$ and $\max(iy, ny)$, respectively. Typically, the output format can be assumed to only pertain to a single location when an output width of 16 is used.

The row format for each buffer differs from one network layer to the next. It is dependent upon the dimensions of the layer's data structures, the size of the row, and the schedule specification for the row. To determine the row format, the simulator iterates through the schedule specification's list of dimensions and fills the row from each dimension. Typically, this results in a row format where a row is filled by sequencing along the first dimension specified with all other dimensions held constant within the row. However, if the first dimension is smaller than the row capacity then the row format will be filled by multiples of the first dimension size along the second dimension in the schedule specification, and so forth.

The schedule specification also defines the order in which rows are retrieved into each buffer using a similar approach. The dimensions of the schedule specification are considered in order incrementing the size of the buffer contents by multiples of the row size until the buffer capacity is reached, subject to constraints imposed by the layer size and calculation region imposed by other buffer contents. The resulting contiguous region of data can be expressed as a loop specification representing the current buffer contents.

7.1.5 Loaders

The canonical accelerator contains three buffers and a processing unit. The contents of the buffers at any given time defines the set of calculations that can be performed by the processing unit. This set of calculations is the intersection of the calculations pertaining to the contents of each buffer. Therefore, the schedule for data loaded into one buffer is tightly coupled to that of the other buffers. The ordering of loop nesting therefore influences both the amount of calculation that can take place per buffer fill and the amount of reuse of buffer entries per buffer fill. Reuse is of course desirable since it decreases the number of times that buffers must be refilled, and finding an optimal schedule for a given layer and architecture involves considering different loop nest orderings, intra-buffer loop iteration orderings, and row formats. Simulation of each of these settings is facilitated by the loader which determines based on these settings what should be loaded into each buffer at a given time.

Each data structure (ie. weights, inputs, and outputs) is associated with its own buffer. A loader controls iteration through the data structures associated with each buffer. The loader has a `loop` function that accepts a *loop specification* as input and generates a sequence of sub-*loop specifications* that collectively span the input loop specification, each of which can be loaded into and reside within the corresponding buffer at once. Each `loop` function generates *loop specifications* that can be used as input to the `loop` function of a different buffer within a nested set of loops. The outermost `loop` has a *loop specification* input pertaining to the full set of calculations in the layer and each level of nesting refines the active calculation region.

The scheduling aspects of an accelerator's stationarity when processing a particular layer are controlled by the ordering of these nested loops. For example, an output stationary schedule would use the output buffer's loader as the outermost loop. Different output stationary schedules are defined by the *schedule spec* of the output buffer and the ordering of the inner loops. The *schedule spec* is used to define which subset of the outputs are active at a given time and the inner loops define the order in which calculations on these are performed.

The loader may be associated with a DFG describing the set of calculations required for the layer, in which case it can dynamically determine which data should be loaded based on the progress that has been made. When the DFG is not used, the loader sequencing is based on iteration through the `for` loops associated with the buffer. Although the loop specifications themselves are not necessarily aligned to the row boundary, the loader guarantees that the loop specifications do not exceed the maximum number of rows available.

Each time that a loop specification is returned by the loader's `loop` function it is first inserted into a cache that tracks the rows that have been loaded and determines whether new rows are hits or misses as well as deciding which rows to evict when misses occur. The list of missed rows is used to impose a timing constraint on the load time of the new data which must take place after the last access of the evicted rows.

7.1.6 Timing

The simulator models various stalling conditions that arise from buffer capacity, bandwidth limitations, and cache contention.

The cache tracks row aligned ranges where each row has an associated tag based on the minimal indices of the row along each relevant dimension for the corresponding data structure. Each time that a row is accessed, the access time is logged for the associated tag. The contents of the cache are maintained in LRU order. When a new block is inserted it is compared to the current contents of the cache and when an overlap is identified with an existing block, the overlapping section is removed from the existing block to be reinserted at the end of the list with the new block.

Evictions from the cache occur based on a predefined capacity during insertion of new blocks into the cache. When an eviction occurs, the cache's local clock is compared to the last access time for the evicted block. If the evicted block was last accessed at a time that is later than the cache clock then a stall occurs affecting the timing of the new insertion.

Data structures such as the output activations that are not read-only and must be written can cause additional delays when data is evicted due to bandwidth constraints on the write port. These constraints may be imposed either at the processing side or the memory side of the cache's store FIFO. The set of evicted tags are stored in a free list along with the time that the associated block becomes available according to the clock maintained within the store buffers. This free list is consulted when the rows are accessed to determine whether the processing element's execution sequence must stall due to free list availability and if so, the stall advances the global clock and increases a stall counter within the cache.

A load buffer bandwidth is also considered when determining whether loading of data must stall. The resulting performance measurement represents an ideal prefetching schedule where data is evicted and loaded as soon as would be possible, independent of the accelerator front end control logic.

7.1.7 Data Flow Graph

A data flow graph (DFG) is maintained while processing each layer to represent the outputs that are actively being accumulated. Within the DFG, outputs are grouped into sets which accumulate in lock-step. Each of these output sets is associated with a unique key built from N, Nx, Ny, and B. While an output is actively being processed, it is allocated a live input tree which is updated each time that an input is processed for that output. Once all inputs have been processed for a given input, the output key is added to a dead output tree to keep track of completed outputs. The DFG can be used for both validating a computation schedule and facilitating more dynamic scheduling than simply iterating through every *loop specification*.

Trees are used to improve performance and memory usage for the DFG. When a live input tree is first allocated for a given output, the set of live inputs includes all inputs that are used by a given output key. The depth of the tree is three nodes, where each tree level represents one dimension of the input (X, Y, and channel). Each tree layer is a linked list where the list nodes contain a pointer to the next tree level and the next node in the linked list at the same tree level. In addition, each node contains a minimum and maximum value pertaining to the represented range of remaining inputs. Each time that a calculation takes place, the DFG is updated by removing the corresponding input range from the tree corresponding to the output key. Removal from the tree involves recursive identification of overlapping nodes in the tree and removal. When nodes partially overlap, a duplicate node is created with a copy of all sub-nodes and inserted into the linked list. The range of the duplicate and original nodes are set to be non-overlapping where one node represents the item to be removed, where recursion takes place. Once reaching the end of the tree, the overlapping node is removed along with nodes higher in the tree that no longer have any sub-nodes. When a tree has been completely removed, this represents the end of the calculation for a given output key which is then inserted into the dead output tree. The purpose of the dead output tree is primarily verification and preventing recalculations.

The dead output tree has a similar node structured to the live input tree in that each tree level contains a linked list pertaining to one dimension of the output. The primary difference is that when the tree is updated, it merges nodes together rather than splitting them apart as was the case for the live input tree. Nodes are merged when they represent consecutive ranges and all sub-nodes represent full ranges. At the end of a layer's simulation, all output keys should be accounted for so the entire tree should be merged into a single node. This merging is done each time that a key is added to the tree in order to keep the size of the data structure small and easy to search quickly.

7.1.8 Schedules

When the DFG is used for dynamic scheduling, the minimal output key is used as the basis for selecting the next set of inputs. Incidental overlaps with other outputs may then result in calculations which may not occur using increment based scheduling alone. Increment based scheduling selects successive ranges to fill the buffers and assumes that once a range of values has been visited it is not revisited. While this scheme is appropriate for some DNN schedules, convolutional layers may not be optimally treated due to the reuse of inputs with nonuniform locality properties. Neurons within the same layer that pertain to different locations in that layer do share some of the same set of presynaptic neurons as inputs. Thus, when multiple locations reside in the output buffer simultaneously, the buffer contents can be divided into multiple locality sets, each composed of neurons which necessarily share the same set of presynaptic neurons. Assuming some overlap between the locality sets, it is worthwhile to share input activations among multiple locality sets concurrently residing in the output buffer.

When scheduling calculations with multiple locality-sets in the output buffer, the locality sets complete at different times. To account for this, the scheduling scheme designates one of the sets as the leading set and prioritizes loading inputs and weights that pertain to the leading set. Other locality sets can then opportunistically perform calculations that exploit locality from leading set calculations.

7.1.9 Energy Model

The simulator maintains a set of event counters used for making power calculations. The equations used to calculate power use these counter values in combination with a set of constants collected from either RTL synthesis or Cacti6.5. The following table lists some of the power components and their associated simulation counters and coefficient sources:

Power Component	Simulation Counter	Coefficient Source
Multiplier Energy	mvm.WProc	RTL
Adder Tree Energy	outreg accesses	RTL
Accumulation Energy	outreg accesses	RTL
Input Buffer Access	inreg miss	RTL or Cacti
Weight Buffer Access	weightreg miss	RTL or Cacti
Output Buffer Access	outreg miss	RTL or Cacti
Input Buffer Fill	inbuf miss	Cacti and Wire
Weight Buffer Fill	weightbuf miss	Cacti
Output Buffer Drain	outtbuf miss	Cacti and Wire

Table 7.2: Power components and the associated counters.

7.1.10 Multi-node Approximation

In order to simulate multi-node architectures with high performance, the simulator divides the loop specification representing a full layer according to a mapping specification. The mapping specification is essentially just used to specify how many partitions should be used along each dimension of the calculation. Due to symmetry of the calculation across the multiple nodes, simulation is based on one node only with worst case latencies under otherwise synchronized assumptions used to model the contention introduced by using multiple nodes.

7.2 Dim Sum

A second version of the simulator was developed to evaluate the high latency, extremely wide systolic array based processing elements that are Diastolic arrays. While the simulator used in Chapter 3, shares much of the underlying infrastructure and guiding principles with the version used in the other two chapters, it makes several simplifying assumptions which greatly improve its performance. The primary difference is that instead of using configurable scheduling, a static O/W scheduling scheme is used in order to reduce the off-chip memory accesses associated with loading each new set of weights. The other simplification is the elimination of buffers. The weight buffer is treated as a FIFO only and input buffering is not considered other than as a centralized data store. This eliminates the cache related stalling conditions of the previous version such that processing latencies are based on memory bandwidth, diastolic cell dimensions, and array dimensions which affect its input and output bandwidths. Without caching, the loader based infrastructure of the previous version is also unnecessary.

7.2.1 Sequencing

For each layer, the processing schedule follows a sequence of matrix multiplications following the formulation presented in Section 3.3, where each layer is expressed as a summation over a series of matrix multiplications. The dimensions of each input matrix of the series is

$$L_i N_i \times \left\lceil \frac{N_b N_y \left\lceil \frac{N_x}{L_i} \right\rceil}{V P_s} \right\rceil, \tag{7.1}$$

and the dimensions of each weight matrix is

$$L_o \left\lceil \frac{N_o}{P_c} \right\rceil \times L_i N_i, \tag{7.2}$$

where P_s and P_c represent the degree of spatial and channel partitioning across nodes in a tiled architecture. When these dimensions exceed processing array dimensions $(A_i \times A_o)$, the matrices are divided into sub-matrices, each of which is one iteration of the innermost loop of the static schedule's loop nest. In order to achieve permutation locality, the innermost loops pertain to different permutations of the same set of weights, which includes all row-wise permutations and as many column-wise permutations as can be accommodated by the output buffer. An outer for loop is responsible for a second level of column-wise permutation iteration in cases when the output capacity is exceeded. The static loop ordering scheme is as follows: Loop 1 partitions the output across channels

1. Output channel set
2. Outer column-wise permutation set
3. Feature Map Spatial Dimension set
4. Input channel set
5. Kernel Y-dimension
6. Kernel X-dimension set
7. Row-wise permutations
8. Inner column-wise permutations

Figure 7.2: Loop Ordering Scheme for Diastolic Arrays (from outermost to innermost).

according to the width of the output buffer. Loops 2 and 3 partition the neurons to fill the capacity of the output buffer. Loop 4 partitions the input across channels according to the width of the activation array input. Loop 5 and 6 iterate over the X and Y coordinates of the kernel, addressing one Y-coordinate per sub-matrix computation but potentially many X-coordinates depending upon the number of input and output locations per sub-matrix computation. The number of iterations of loop 6 is determined based on the number of alignments of the kernel to an input row as well as how many of those alignments will be considered by different row-wise permutations. Thus, the number of iterations of loop 6 is calculated as $\left\lceil \frac{N_{xk}+L_i-1}{\texttt{LCM}(L_i,L_o)} \right\rceil$, where the LCM term accounts for both the number of row-wise permutations and the number of locations per input row. Loop 7 iterates through row-wise permutations, all of which are required to compute a given neuron output. Loop 8 iterates through the column-wise permutations corresponding to the locality sets contained within the output buffer.

The reason that column-wise permutation is handled by the innermost loop is because activation input sets can be reused across the column-wise permutations. The row-wise permutation is the next outer loop in order to exploit reusability of the weight buffer for different permutations. The iteration order for the spatial dimensions of the filter is selected to improve the likelihood of reuse of input buffer contents, which is dependent upon input buffer locality. The outer most loops pertain to output buffer allocation.

To expand upon some of the explanation of permutations in Section 3.3, the following set of equations describe how the number of permutations is calculated based on the processing element dimensions $(A_i \times A_o)$ and the number of input channels (N_{ci}), output channels (N_{co}), and spatial dimensions of the layer (N_{xi}, N_{xo}).

$$L_o = \min\left(\max\left(\left\lfloor \frac{A_o}{N_{co}} \right\rfloor, 1\right), N_{xo}\right) \tag{7.3}$$

$$L_i = \min\left(\max\left(\left\lfloor \frac{A_i}{N_{ci}} \right\rfloor, 1\right), N_{xi}\right) \tag{7.4}$$

$$V = \frac{L_o}{\texttt{gcd}\left(L_o, L_i\right)} \tag{7.5}$$

$$H = \frac{L_i}{\texttt{gcd}\left(L_o, L_i\right)} \tag{7.6}$$

where L_o and L_i are the number of locations per output row and input row, respectively, and V and H are the number of row-wise and column-wise permutations, respectively. Rounding errors can occur when not all permutations require the same number of spatial kernel iterations, which is handled by detecting these iterations when they occur and skipping them.

7.2.2 Timing Model

The simulator maintains multiple "clocks" associated with different accelerator components in order to facilitate interval based simulation. Each component's clock advances independently of other components while until a dependency on another component is encounter at which point a stalling condition may be identified by time-stamp comparisons, at which point the stalling clock is advanced to the time when the dependence is resolved. These dependencies can take the form of either a true-dependence or an anti-dependence. An anti-dependence requires resource availability and causes stalls due to resource contention where the dependence is resolved once the resource is no longer busy. An anti-dependence stall is indicated when the contended resource clock is ahead of the client clock. A true-dependence requires a producer component to supply a consumer component, such that when the producer is not ready a stall condition occurs that advances the consumer component clock. The true-dependence stalling condition is indicated when the producer clock is ahead of the consumer clock, representing that the producer will not be ready until the time held by its clock.

Whenever a new weight sub-matrix is loaded from memory it is loaded into the weight buffer. This buffer acts as a FIFO with finite capacity, representing an anti-dependence in which the time-stamp associated with a FIFO's most recent eviction is compared with the memory access clock. In larger buffers, the evicted block represents data that was processed further in the past and therefore should provide a better opportunity for prefetching into the buffer. Once this stalling condition is accounted for, the memory access timing is based on a latency calculation involving the achievable bandwidth and the size in bytes of data to be retrieved. A true-dependence exists between the weight buffer and the processor arrays, so if the clock associated with the memory prefetch advances beyond that of the array then a *memory stall* occurs, representing a situation when the array is ready but weight data still needs to be retrieved from memory first.

An anti-dependence exists between the weight buffer and the processing array as well, which delays loading of weights when it is encountered. Specifically, weights cannot be loaded into the array until all computations pertaining to the oldest set of weights resident within the array have completed. Thus when the completion timestamp of the oldest resident computation within the array exceeds the weight buffer clock, a *compute stall* occurs, in which neither new inputs nor weights can be loaded into the array until resolution. The anti-dependence must be considered for each permutation of a weight set, while the true-dependence is only evaluated for the first permutation of each weight set.

Weights are optimistically permitted to begin loading into the array once the time remaining for retrieving weights from memory is less than the latency of loading the full set of weights into the array and the compute anti-dependence resolves. The true-dependence between the weight buffer and the array results in a *weight loading* stall when the latency required to load a new set of weights prevents back to back operation of the array. This is indicated when the timestamp of the weight load exceeds the array clock. Where the compute clock represents the time that the last input row of the previous sub-matrix multiplication first arrived at the array.

During each iteration, the compute clock advances with each stalling condition and according to the input rows processed within that iteration. The additional processing latency associated with propagation activations through the array, performing calculations, and propagation of accumulating summations affect the timestamp placed into the active computation FIFO, and therefore only influence the compute clock through the weight loading stall. The compute clock is the clock used to measure computation latency.

7.2.3 Energy Model

The energy model for Dim Sum is similar in principle to that of Tempura. However, different RTL models and different Cacti configurations are used to gather data. The table above lists the energy components that are tracked by the performance model as well as the source of constants used to produce power estimates. The power constants for processing elements are based on RTL simulation and synthesis. The power constants for SRAMs are collected from comparable Cacti [104] configurations. The interconnect model is based on experiments conducted within previous work [100], and DRAM

Power Component	Simulation Counter	Coefficient Source
Mult-Acc Energy	Multiply-Accumulate Count	RTL
Array Idle Energy	Compute Clock	RTL
Accumulation Energy	Output Buffer Access Count	Cacti
Input Buffer Access	Input Buffer Access Count	Cacti
Input Interconnect	Input Buffer Access Count	Interconnect Model
Weight Loading	Permutation Count	RTL and Cacti
Weight Memory	Weight Set Count	DRAM Spec
Weight Interconnect	Weight Set Count	Interconnect Model

Table 7.3: Power components and the associated counters.

power constants are collected from literature [85, 107, 119].

7.3 Chapter Summary

This chapter presented the simulation infrastructure used for the studies conducted in Chapter 3, Chapter 4, and Chapter 6. The first simulator version, called Tempura, has greater architectural configuration flexibility with detailed cache content tracking. The second simulator version, called Dim Sum, is more rigid and does not include a hierarchical cache model, but is able to simulate with much greater performance. Both simulators are capable of using symmetry assumptions to accelerate simulation of tiled architectures. The results provided by both simulators pertain to both performance and power estimates.

8 CONCLUSION

What we usually consider as impossible are simply engineering problems... there's no law of physics preventing them.

— Kaku, Michio

8.1 Conclusion

Three studies of the neural network accelerator design space are conducted within this dissertation. The first study considers three circuit design techniques that reduce the number of clocked storage elements within the processing array of an accelerator based on the Tensor Processing Unit [64] (TPU) demonstrating reductions of both processing latency and power consumption. This study also considers permutation aware scheduling mechanisms and policies for reducing off-chip memory accesses pertaining to permutations of previously loaded weights. Also within this study, the memory bandwidth tolerance of monolithic systolic array processing elements is compared against that of accelerators containing multiple smaller systolic arrays but equivalent computational capacity. These tiled architectures have higher on-chip bandwidth demands but better array utilization for certain neural network topologies than the monolithic accelerators. The optimal array size for the energy efficiency tradeoff between global interconnect power and array utilization is found to be 128x128 as opposed to the 256x256 arrays of the baseline design. The primary benefit of tiling is a performance improvement while the primary benefit of clocked storage element elimination is energy efficiency. The two accelerator modifications are entirely compatible and result in a 2.8x average energy-delay improvement when combined.

The second study considers specialization for codebook quantization of both activations and weights within an accelerator based on DaDianNao [16]. The weight storage for this accelerator is entirely within on-chip memory which significantly reduces the data movement distance for synaptic weights. However, due to the large size of the weight storage, the power consumption of the baseline accelerator is dominated by interconnect energy. This power component is significantly impacted by the data compression provided by codebook quantization, such that after specializing on-chip storage structures and interconnects for the encoded format, the dominant power component is the synapse computation stage of neuron processing. The study includes analysis replacing the 16-bit truncating multipliers with lookup tables (LUTs), which reduces the synapse computation power to be comparable to that of using 8-bit multipliers. The study further demonstrates energy reductions when scaling to smaller codebook sizes, enabled by LUT based synapse computation. The analysis includes an exploration of the tradeoff between the number of output buffer entries and activation retransmissions to determine that power consumption is minimized by using an output buffer of size 4KB for each processing element.

The third accelerator builds upon the second to further reduce the interconnect energy by using temporal codes to communicate both weights and activations. An alternative convolutional scheduling scheme, in which the output buffer allocation may include multiple locations at a time which complete accumulation at different times and therefore release only a portion of the buffer after each time step, leading to better transmission latency tolerance for some workloads. Supporting activation reuse in this manner requires high weight bandwidth which is supported by four banked weight memories providing concurrent temporal transmissions of four weight sets at a time.

8.2 Reflections and Future Directions

8.2.1 Diastolic Array Extensions

Recent versions of the TPU have included support for bfloat16, which is a relatively new floating point format with the same dynamic range as IEEE single precision floating points but with a mantissa containing only seven bits [42]. Extending diastolic arrays to support floating points is non-trivial since logic to handle the floating point exponent increases the critical paths of the addition logic. One way to approach supporting floating points within an array using wide cells would be to maintain primarily fixed point arithmetic within each cell by using a block floating point approach in which a maximum exponent within a vector is used as the exponent for the entire vector with all significands shifted appropriately after performing multiplication. This is similar to the approach taken for the Microsoft BrainWave [34], where a single exponent is extracted from vectors of size 128. The main caveat of this approach is the loss of precision with large vector sizes that would only be exacerbated by the already limited precision of the bfloat16.

The fixed dataflows of the diastolic array are a major limiting factor in achieving high utilization. While it would be nice to support additional data flows, the cell expansion based design techniques are based on an assumed data flow, so reconfigurability within diastolic arrays would not necessarily be easily achieved. Ideally, reconfigurability could be integrated without disrupting cell expansions. This requires data-flow reconfigurations only be applied at the granularity of the expanded cells, which may not be worthwhile. For the time being, it would seem that reconfigurability and diastolic arrays are based on diametrically opposed physical design principles. The energy-efficiency provided by reconfigurable arrays, like flexflow [93], is achieved by high utilization at the expense of some reconfiguration overhead, while the energy-efficiency provided by diastolic arrays assumes a fixed data-flow thereby degrading utilization in some cases.

One form of reconfigurability that would seem to be appropriate for integration into diastolic arrays is a dynamically reconfigurable pipeline depth. This works in conjunction with DVFS mechanisms to enable low voltage, high frequency operation by dynamically increasing the pipeline depth. This provides the option of a high performance turbo boost without increasing the frequency, or perhaps a low power mode for arrays already operating close to V_{min}. The pipeline depth adjustment mechanisms previously described by Koppanalil et. al. [75] would be appropriate for this purpose within diastolic arrays.

Although not considered within the original design of the array, integration of column-wise permutation logic at the array input, as is shown in Figure 8.1, would provide a performance improvement at the expense of the power requirements of a rather large switch. Similar to the row-wise weight permutation logic, only rotations would need to be supported by the switch, however many more rotations need to be supported

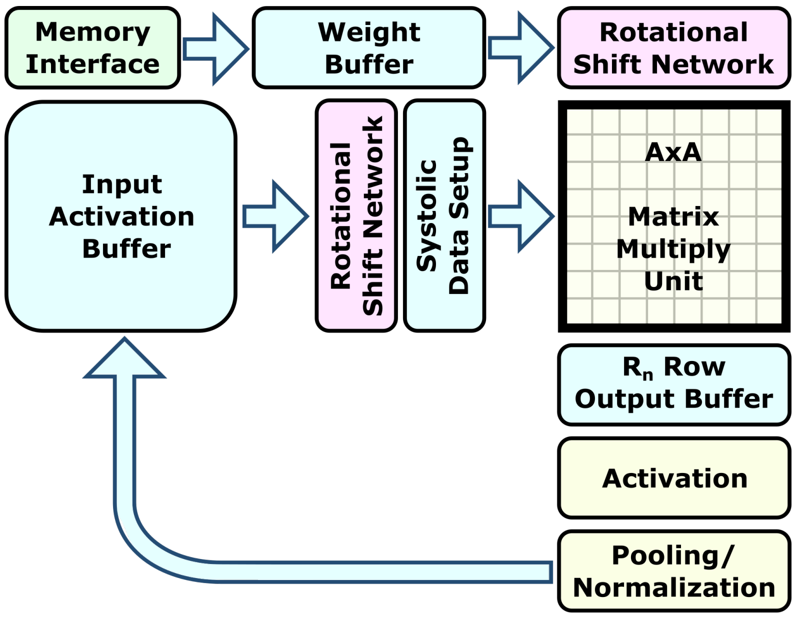


Figure 8.1: Alternative accelerator design with one permutation network supporting permutation of activations and one permutation network supporting permutation of weights.

for activations than for weights, so the capacitance of the input permutation switch would be much higher than that of the weight permutation network, although still less than the energy per access of the input activation buffer. The performance improvement is derived from an ability to begin processing a new column-wise permutation without reloading weights into the array, thus eliminating the weight loading latency when there is sufficient output buffer capacity to support additional column-wise permutation sets. As the weight loading latency accounts for a significant portion of overall latency in Figure 3.19, a performance improvement can be derived from reducing the number of column-wise permutations loaded into the array. This latency reduction would need to be weighed against the power consumed by the additional permutation support and the fact that the memory bandwidth imposes a lower bound on overall latency.

8.2.2 Extending LUT based multiplication

It would seem that the diastolic array design techniques are compatible with integration of the lookup tables described for quantized activation support, where the notion of diastolic cell height could be used to represent the sharing of activations used for synapse calculations in Figure 4.3 and Figure 4.4 is sufficiently large to share lookup tables across many dot-product lanes, preferably greater than the number of weight quantization levels. These two techniques would seem to be mostly orthogonal, in that the only modification that would be required to support lookup tables within a diastolic array would be additional support for their reconfiguration, which can be implemented with similar overheads as weight loading. However, an incompatibility may arise within the physical layouts where cells containing LUTs within a large array lead to area inefficiencies that were easily hidden alongside massive weight storage in Chapter 4, but would be amplified and could become quite significant within larger processing arrays.

Although diastolic arrays containing LUT based synapses represents a feasible design point, the added complexity associated with integration of LUT based processing elements into an array of processing elements requires more careful consideration regarding the area of LUT cells, particularly with regard to the area agnostic experiments presented in Figure 4.10, where the preferred implementation of synapse computation for a given set of codebook sizes remains an open question which may vary from one process node to the next.

One would expect the impact of compression on the sensitivity studies considered in Section 3.6.4 to have a similar reduction of the interconnect power and memory power components, which likely leads to a stronger preference for tiled architectures over monolithic arrays when specialized for codebook quantization. Overall, codebook based compression is certainly worthwhile within array based accelerators. Although the power reductions observed in the high bandwidth weight memories collocated with processing elements of Chapter 4 are not present within the array based design, the compressed format could instead be harnessed to increase weight bandwidth by loading more than one weight per row per cycle. For the weight loading data-paths described in Section 3.4.1 increased weight bandwidths could be implemented using multi-column interleaved weight storage formats, where weights for multiple array columns are contained within each weight buffer row.

8.2.3 Temporal Code Consideration

There is an intriguing corner of the tiled design space in which compute elements are coupled with smaller SRAMs per tile, such that more tiles can be integrated into a given accelerator. The reason this area of the design space is intriguing is because it reduces the weight access overhead that was addressed by temporal coding in Chapter 6. If the weight memories are small enough then there is no longer a benefit to temporal coding of weights. However, as each processing element of this massively tiled architecture supports the synapses of fewer neurons, the computational bandwidth of each tile is available to a smaller set of neurons. When the computation bandwidth per tile is too low, activation locality is not effectively exploited across neurons without broadcasts spanning many tiles. When the computational bandwidth is high, small weight stores result in under-utilization of this bandwidth while larger weight stores exacerbate energy per access concerns. Although, this under-utilization of many distributed computational resources is consistent with biological models of neural networks, severe under-utilization of computational resources is not cost effective. All that aside, this could represent one way of harnessing dark silicon for a low power system design. As far as temporal codes are concerned, the high radix activation network required by these systems could potentially benefit from temporal coding.

8.3 Closing Remarks

The process of preparing this dissertation has involved many iterations of analysis with decidedly different accelerators and many incremental successes and failures along the way. The accelerators studied within this dissertation cover many design points, targeting different mechanisms of achieving energy efficiency. Although the exact designs considered may never see the light of day, each study does provide insights into approaches to addressing important considerations within the accelerator design space which have also been observed in related work and are undoubtedly reflected in the accelerator design process that is necessary for engineering new neural network accelerator designs. For those who read through this dissertation, I sincerely hope that they can derive insights that positively influence their own approach and perspective on the emerging computing ecosystem, destined to incorporate neural network acceleration in a very significant way.

BIBLIOGRAPHY

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015.

[2] F. Akopyan et al. "TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10 (Oct. 2015), pp. 1537–1557.

[3] J. Alberico et al. "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing". In: *2016 ACM/IEEE 43rd International Symposium on Computer Architecture (ISCA)*. June 2016.

[4] M. Alwani et al. "Fused-layer CNN Accelerators". In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan, Oct. 2016, pp. 1–12.

[5] Amazon. *Overview of Amazon Web Services*. https://docs.aws.amazon.com/aws-technical-content/latest/aws-overview/aws-overview.pdf. Dec. 2018.

[6] D. Attwell and S. B. Laughlin. "An energy budget for signaling in the grey matter of the brain". In: *J. Cereb. Blood Flow Metab.* 21.10 (Oct. 2001), pp. 1133–1145.

[7] Utku Aydonat et al. "An OpenCL™Deep Learning Accelerator on Arria 10". In: *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: ACM, 2017, pp. 55–64. ISBN: 978-1-4503-4354-1.

[8] V. Balasubramanian. "Heterogeneity and Efficiency in the Brain". In: *Proceedings of the IEEE* 103.8 (Aug. 2015), pp. 1346–1358.

[9] Mahdi Nazm Bojnordi and Engin Ipek. "DESC: Energy-efficient Data Exchange Using Synchronized Counters". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 234–246. ISBN: 978-1-4503-2638-4.

[10] Zhaowei Cai et al. "Deep Learning with Low Precision by Half-Wave Gaussian Quantization". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. 2017, pp. 5406–5414.

[11] Kumar Chellapilla, Sidd Puri, and Patrice Simard. "High Performance Convolutional Neural Networks for Document Processing". In: *Tenth International Workshop on Frontiers in Handwriting Recognition*. Ed. by Guy Lorette. Université de Rennes 1. La Baule (France): Suvisoft, Oct. 2006.

[12] T. Chen and G. E. Suh. "Efficient data supply for hardware accelerators with prefetching and access/execute decoupling". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–12.

[13] Tianqi Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 578–594. ISBN: 978-1-931971-47-8. URL: `https://www.usenix.org/conference/osdi18/presentation/chen`.

[14] Tianshi Chen et al. "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 269–284.

[15] Y. Chen, J. Emer, and V. Sze. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. June 2016, pp. 367–379.

[16] Y. Chen et al. "DaDianNao: A Machine-Learning Supercomputer". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2014, pp. 609–622.

[17] Heng-Tze Cheng et al. "Wide & Deep Learning for Recommender Systems". In: *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*. DLRS 2016. Boston, MA, USA: ACM, 2016, pp. 7–10. ISBN: 978-1-4503-4795-2.

[18] Sharan Chetlur et al. "cuDNN: Efficient Primitives for Deep Learning". In: *CoRR* (Oct. 2014).

[19] P. Chi et al. "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. June 2016, pp. 27–39.

[20] Myong Hyon Cho et al. "Diastolic Arrays: Throughput-driven Reconfigurable Computing". In: *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD '08. San Jose, California: IEEE Press, 2008, pp. 457–464.

[21] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. "Towards the Limit of Network Quantization". In: *Proceedings of the 5th Annual International Conference on Learning Representations*. ICLR '17. Toulon, France, 2017.

[22] Fiesler Choudry et al. "A Weight Discretization Paradigm for Optical Neural Networks". In: *in Proceedings of the International Congress on Optical Science and Engineering*. SPIE, 1990, pp. 164–173.

[23] Patricia Smith Churchland and Terrence J. Sejnowski. *The Computational Brain*. 1st. Cambridge, MA, USA: MIT Press, 1994. ISBN: 0262531208.

[24] Matthieu Courbariaux and Yoshua Bengio. "BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1". In: *CoRR* abs/1602.02830 (2016).

[25] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'15. Montreal, Canada: MIT Press, 2015, pp. 3123–3131.

[26] P. Dayan and L.F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. Computational Neuroscience Series. Massachusetts Institute of Technology Press, 2001.

[27] R. H. Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268.

[28] Sorin Draghici. "On the Capabilities of Neural Networks Using Limited Precision Weights". In: *Neural Netw.* 15.3 (Apr. 2002), pp. 395–414. ISSN: 0893-6080.

[29] R. G. Dreslinski et al. "Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits". In: *Proceedings of the IEEE* 98.2 (Feb. 2010), pp. 253–266.

[30] Hadi Esmaeilzadeh et al. "Dark Silicon and the End of Multicore Scaling". In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA '11. San Jose, California, USA: ACM, 2011, pp. 365–376.

[31] Stijn Eyerman et al. "A Performance Counter Architecture for Computing Accurate CPI Components". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: ACM, 2006, pp. 175–184. ISBN: 1-59593-451-0.

[32] Michael Feldman. *New GPU Accelerated Supercomputers Change the Balance of Power on the TOP500*. June 2018.

[33] K. Flautner et al. "Drowsy caches: simple techniques for reducing leakage power". In: *Proceedings 29th Annual International Symposium on Computer Architecture*. May 2002, pp. 148–157.

[34] J. Fowers et al. "A Configurable Cloud-Scale DNN Processor for Real-Time AI". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. June 2018, pp. 1–14.

[35] Douglas Fox. "Neuroscience: The Limits of Intelligence". In: *Scientific American* 305.1 (July 2011), pp. 36–43.

[36] Mingyu Gao et al. "Tangram: Optimized Coarse-Grained Dataflow forScalable NN Accelerators". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA, 2019.

[37] Mingyu Gao et al. "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: ACM, 2017, pp. 751–764.

[38] D. Genbrugge, S. Eyerman, and L. Eeckhout. “Interval simulation: Raising the level of abstraction in architectural simulation”. In: *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. Jan. 2010, pp. 1–12.

[39] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep Sparse Rectifier Neural Networks”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*. Vol. 15. 2011, pp. 315–323.

[40] David H. Goldberg, Arun P. Sripati, and Andreas G. Andreou. “Energy efficiency in a channel model for the spiking axon”. In: *Neurocomputing* 52–54 (2003). Computational Neuroscience: Trends in Research 2003, pp. 39 –44.

[41] Yunchao Gong et al. “Compressing Deep Convolutional Networks using Vector Quantization”. In: *CoRR* abs/1412.6115 (2014). arXiv: `1412.6115`. URL: `http://arxiv.org/abs/1412.6115`.

[42] Google. *Cloud TPU System Architecture*. Dec. 2018.

[43] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. “Dynamically Specialized Datapaths for Energy Efficient Computing”. In: *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*. HPCA ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 503–514.

[44] Alex Graves, Greg Wayne, and Ivo Danihelka. “Neural Turing Machines”. In: *CoRR* abs/1410.5401 (2014).

[45] Yunhui Guo. “A Survey on Methods and Theories of Quantized Neural Networks”. In: *CoRR* abs/1808.04752 (2018). URL: `http://arxiv.org/abs/1808.04752`.

[46] Suyog Gupta et al. “Deep Learning with Limited Numerical Precision”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 1737–1746. URL: `http://proceedings.mlr.press/v37/gupta15.html`.

[47] P. Gysel, M. Motamedi, and S. Ghiasi. “Hardware-oriented Approximation of Convolutional Neural Networks”. In: *ArXiv e-prints* (Apr. 2016).

[48] S. Han et al. “EIE: Efficient Inference Engine on Compressed Deep Neural Network”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. June 2016, pp. 243–254.

[49] Song Han, Huizi Mao, and William J. Dally. “Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding”. In: *CoRR* abs/1510.00149 (2015). URL: `http://arxiv.org/abs/1510.00149`.

[50] Song Han et al. “ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA”. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’17. Monterey, California, USA, 2017, pp. 75–84. ISBN: 978-1-4503-4354-1.

[51] Song Han et al. “Learning Both Weights and Connections for Efficient Neural Networks”. In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’15. Montreal, Canada: MIT Press, 2015, pp. 1135–1143.

[52] A. Hartstein and T. R. Puzak. “Optimum power/performance pipeline depth”. In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* Dec. 2003, pp. 117–125.

[53] A. Hartstein and Thomas R. Puzak. “Optimum Power/Performance Pipeline Depth”. In: *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 36. IEEE Computer Society, Dec. 2003, pp. 117–125.

[54] Atif Hashmi et al. “Automatic Abstraction and Fault Tolerance in Cortical Microachitectures”. In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA ’11. San Jose, California, USA: ACM, 2011, pp. 1–10.

[55] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). arXiv: `1512.03385`.

[56] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[57] Mark Hill and Vijay Janapa Reddi. “Gables: A Roofline Model for Mobile SoCs”. In: *Proceedings of the 2019 IEEE 25th International Symposium on High Performance Computer Architecture*. HPCA ’19. Washington, D.C., USA, 2019.

[58] A. L. Hodgkin, A. F. Huxley, and B. Katz. “Measurement of current-voltage relations in the membrane of the giant axon of Loligo”. In: *The Journal of Physiology* 116.4 (1952), pp. 424–448.

[59] Itay Hubara et al. “Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations”. In: *CoRR* abs/1609.07061 (2016).

[60] W. M. Hwu et al. “Compute Unified Device Architecture Application Suitability”. In: *Computing in Science Engineering* 11.3 (May 2009), pp. 16–26.

[61] Xianyan Jia et al. *Highly Scalable Deep Learning Training System with Mixed Precision: Training ImageNet in Four Minutes*. 2018. arXiv: `1807.11205 [cs.LG]`.

[62] M.I. Jordan and S. Russell. “Computational intelligence”. In: *The MIT Encyclopedia of the Cognitive Sciences*. Ed. by R.A. Wilson and F.C. Keil. Cambridge, MA: MIT Press, 1999.

[63] Norman P. Jouppi et al. “A Domain-specific Architecture for Deep Neural Networks”. In: *Commun. ACM* 61.9 (Aug. 2018), pp. 50–59.

[64] Norman P. Jouppi et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA ’17. Toronto, ON, Canada: ACM, 2017, pp. 1–12.

[65] P. Judd et al. “Stripes: Bit-serial deep neural network computing”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–12.

[66] P. Judd et al. “Stripes: Bit-serial deep neural network computing”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–12.

[67] Patrick Judd et al. “Proteus: Exploiting Numerical Precision Variability in Deep Neural Networks”. In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS ’16. Istanbul, Turkey: ACM, 2016, 23:1–23:12.

[68] U. R. Karpuzcu, N. S. Kim, and J. Torrellas. “Coping with Parametric Variation at Near-Threshold Voltages”. In: *IEEE Micro* 33.4 (July 2013), pp. 6–14.

[69] S. W. Keckler et al. “GPUs and the Future of Parallel Computing”. In: *IEEE Micro* 31.5 (Sept. 2011), pp. 7–17.

[70] Brucek Khailany et al. “A Modular Digital VLSI Flow for High-productivity SoC Design”. In: *Proceedings of the 55th Annual Design Automation Conference*. DAC ’18. San Francisco, California: ACM, 2018, 72:1–72:6.

[71] Paresh Kharya. *Intel Highlighted Why NVIDIA Tensor Core GPUs Are Great for Inference*. May 2019.

[72] D. Kim et al. “Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. June 2016, pp. 380–392.

[73] N. S. Kim et al. “Heterogeneous Computing Meets Near-Memory Acceleration and High-Level Synthesis in the Post-Moore Era”. In: *IEEE Micro* 37.4 (2017), pp. 10–18.

[74] Rakesh Komuravelli et al. “Stash: Have Your Scratchpad and Cache It Too”. In: *Proceedings of the 42nd International Symposium on Computer Architecture*. ISCA. Portland, Oregon: ACM, June 2015, pp. 707–719.

[75] Jinson Koppanalil et al. “A Case for Dynamic Pipeline Scaling”. In: *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. CASES ’02. Grenoble, France, 2002, pp. 1–8. ISBN: 1-58113-575-0.

[76] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. University of Toronto, 2009.

[77] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105.

[78] R. Kumar et al. “Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction”. In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* Dec. 2003, pp. 81–92.

[79] Kung. “Why systolic architectures?” In: *Computer* 15.1 (Jan. 1982), pp. 37–46.

[80] Sun-Yuan Kung. “On supercomputing with systolic/wavefront array processors”. In: *Proceedings of the IEEE* 72.7 (July 1984), pp. 867–884.

[81] H. Kwon and T. Krishna. “OpenSMART: Single-cycle multi-hop NoC generator in BSV and Chisel”. In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Apr. 2017, pp. 195–204.

[82] Kiseok Kwon et al. “Co-design of Deep Neural Nets and Neural Net Accelerators for Embedded Vision Applications”. In: *Proceedings of the 55th Annual Design Automation Conference*. DAC ’18. San Francisco, California: ACM, 2018, 148:1–148:6.

[83] Simon B. Laughlin and Terrence J. Sejnowski. “Communication in Neuronal Networks”. In: *Science* 301.5641 (2003), pp. 1870–1874. eprint: `http://science.sciencemag.org/content/301/5641/1870.full.pdf`.

[84] J. Lee, D. Shin, and H. J. Yoo. “A 21mW low-power recurrent neural network accelerator with quantization tables for embedded deep learning applications”. In: *2017 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. Nov. 2017, pp. 237–240.

[85] S. Lee et al. “Understanding power-performance relationship of energy-efficient modern DRAM devices”. In: *2017 IEEE International Symposium on Workload Characterization (IISWC)*. Oct. 2017, pp. 110–111.

[86] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. *Fermi Architecture White Paper*. Tech. rep. 2010.

[87] Charles E. Leiserson, Flavio M. Rose, and James B. Saxe. “Optimizing Synchronous Circuitry by Retiming (Preliminary Version)”. In: *Third Caltech Conference on Very Large Scale Integration*. Ed. by Randal Bryant. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 87–116.

[88] Peter Lennie. “The Cost of Cortical Computation”. In: *Current Biology* 13.6 (2003), pp. 493 –497.

[89] Hao Li et al. “Training Quantized Nets: A Deeper Understanding”. In: *CoRR* (2017).

[90] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. “Fixed Point Quantization of Deep Convolutional Networks”. In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML’16. New York, NY, USA: JMLR.org, 2016, pp. 2849–2858.

[91] S. Liu et al. “Cambricon: An Instruction Set Architecture for Neural Networks”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. June 2016, pp. 393–405.

[92] Yanpei Liu, Stark C. Draper, and Nam Sung Kim. "SleepScale: Runtime Joint Speed Scaling and Sleep States Management for Power Efficient Data Centers". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 313–324.

[93] W. Lu et al. "FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2017, pp. 553–564.

[94] A. Lukefahr et al. "Composite Cores: Pushing Heterogeneity Into a Core". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2012, pp. 317–328.

[95] Advait Madhavan, Timothy Sherwood, and Dmitri Strukov. "Race Logic: A Hardware Acceleration for Dynamic Programming Algorithms". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 517–528.

[96] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1943), pp. 115–133.

[97] Carver Mead. *Analog VLSI and Neural Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0-201-05992-4.

[98] D. Meisner and T. F. Wenisch. "Does low-power design imply energy efficiency for data centers?" In: *IEEE/ACM International Symposium on Low Power Electronics and Design*. Aug. 2011, pp. 109–114.

[99] Hiroaki Mikami et al. *Massively Distributed SGD: ImageNet/ResNet-50 Training in a Flash*. 2018. arXiv: `1811.05233 [cs.LG]`.

[100] M. Mishkin, N. S. Kim, and M. Lipasti. "Temporal codes in on-chip interconnects". In: *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. July 2017, pp. 1–6.

[101] Thomas M. Mitchell. *Machine Learning*. 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997. ISBN: 0070428077, 9780070428072.

[102] G. E. Moore. "No exponential is forever: but "Forever" can be delayed! [semiconductor industry]". In: *2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.* Feb. 2003, 20–23 vol.1.

[103] Gordon E. Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (Apr. 1965).

[104] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. Dec. 2007, pp. 3–14.

[105] John von Neumann. *The Computer and the Brain*. New Haven, CT, USA: Yale University Press, 1958.

[106] Eriko Nurvitadhi et al. "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: ACM, 2017, pp. 5–14. URL: `http://doi.acm.org/10.1145/3020078.3021740`.

[107] Mike O'Connor et al. "Fine-grained DRAM: Energy-efficient DRAM for Extreme Bandwidth Systems". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 '17. Cambridge, Massachusetts: ACM, 2017, pp. 41–54.

[108] Angshuman Parashar et al. "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 27–40. ISBN: 978-1-4503-4892-8.

[109] Michael Pellauer et al. "Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: ACM, 2019, pp. 137–151.

[110] Steven Pelley et al. "Power Routing: Dynamic Power Provisioning in the Data Center". In: *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XV. Pittsburgh, Pennsylvania, USA, 2010, pp. 231–242.

[111] Gualtiero Piccinini. "The First Computational Theory of Mind and Brain: A Close Look at Mcculloch and Pitts's "Logical Calculus of Ideas Immanent in Nervous Activity"". In: *Synthese* 141.2 (Aug. 2004), pp. 175–215.

[112] Michael Powell et al. "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories". In: *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*. ISLPED '00. Rapallo, Italy, 2000, pp. 90–95.

[113] Andrew Putnam et al. "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 13–24.

[114] Jan M. Rabaey. *Digital Integrated Circuits: A Design Perspective*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996. ISBN: 0-13-178609-1.

[115] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. "Large-scale Deep Unsupervised Learning Using Graphics Processors". In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. Montreal, Quebec, Canada: ACM, 2009, pp. 873–880.

[116] Mohammad Rastegari et al. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks". In: *14th European Conference on Computer Vision (ECCV)*. Amsterdam, The Netherlands, Oct. 2016, pp. 525–542.

[117] Minsoo Rhu et al. "vDNN: Virtualized Deep Neural Networks for Scalable, Memory-efficient Neural Network Design". In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, 2016, 18:1–18:13. URL: `http://dl.acm.org/citation.cfm?id=3195638.3195660`.

[118] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: `10.1007/s11263-015-0816-y`.

[119] *Samsung Rolls Out Industry's First 8GB LPDDR4 DRAM Package*. URL: `https://news.samsung.com/global/samsung-rolls-out-industrys-first-8gb-lpddr4-dram-package`.

[120] Karthikeyan Sankaralingam et al. "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture". In: *Proceedings of the 30th Annual International Symposium on Computer Architecture*. ISCA '03. San Diego, California: ACM, 2003, pp. 422–433.

[121] A. Shafiee et al. "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. June 2016, pp. 14–26. DOI: `10.1109/ISCA.2016.12`.

[122] Yakun Sophia Shao et al. "Aladdin: A Pre-RTL, Power-performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 97–108.

[123] Hardik Sharma et al. "Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)* (2018), pp. 764–775.

[124] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations*. 2015.

[125] J. Smith. "Space-Time Algebra: A Model for Neocortical Computation". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. June 2018, pp. 289–300.

[126] James E. Smith. "Decoupled Access/Execute Computer Architectures". In: *Proceedings of the 9th Annual Symposium on Computer Architecture*. ISCA '82. Austin, Texas, USA: IEEE Computer Society Press, 1982, pp. 112–119.

[127] V. Srinivasan et al. "Optimizing pipelines for power and performance". In: *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.* Nov. 2002, pp. 333–344.

[128] Mircea R. Stan and Wayne P. Burleson. "Bus-invert Coding for Low-power I/O". In: *IEEE Trans. Very Large Scale Integr. Syst.* 3.1 (Mar. 1995), pp. 49–58. ISSN: 1063-8210.

[129] Earl E. Jr. Swartzlander. *Systolic Signal Processing Systems*. New York, NY, USA: Marcel Dekker Inc., 1987.

[130] Christian Szegedy et al. "Going Deeper with Convolutions". In: *Computer Vision and Pattern Recognition (CVPR)*. 2015.

[131] A. Takach. "Design and verification using high-level synthesis". In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. Jan. 2016, pp. 198–203.

[132] M. B. Taylor. "A Landscape of the New Dark Silicon Design Regime". In: *IEEE Micro* 33.5 (Sept. 2013), pp. 8–19.

[133] M. B. Taylor. "Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse". In: *DAC Design Automation Conference 2012*. June 2012, pp. 1131–1136.

[134] O. Temam. "A defect-tolerant accelerator for emerging high-performance applications". In: *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. June 2012, pp. 356–367.

[135] A. M. Turing. "Computers and Thought". In: ed. by Edward A. Feigenbaum and Julian Feldman. Cambridge, MA, USA: MIT Press, 1995. Chap. Computing Machinery and Intelligence, pp. 11–35.

[136] Yaman Umuroglu et al. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference". In: *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA, 2017, pp. 65–74.

[137] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. "Improving the speed of neural networks on CPUs". In: *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*. 2011.

[138] Swagath Venkataramani et al. "ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 13–26. ISBN: 978-1-4503-4892-8.

[139] Ganesh Venkatesh et al. "Conservation Cores: Reducing the Energy of Mature Computations". In: *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XV. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 205–218.

[140] S. Vogel, A. Guntoro, and G. Ascheid. "Efficient hardware acceleration for approximate inference of bitwise deep neural networks". In: *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. Sept. 2017, pp. 1–6.

[141] Pete Warden. *How to Quantize Neural Networks with TensorFlow*. May 2016. URL: `https://petewarden.com/2016/05/03/how-to-quantize-neural-networks-with-tensorflow/`.

[142] Xuechao Wei et al. "Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC '17. Austin, TX, USA, 2017, 29:1–29:6.

[143] Masafumi Yamazaki et al. *Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds*. 2019. arXiv: `1903.12650 [cs.LG]`.

[144] R. Yazdani et al. "The Dark Side of DNN Pruning". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. June 2018, pp. 790–801.

[145] Chris Ying et al. *Image Classification at Supercomputer Scale*. 2018. arXiv: `1811.06992 [cs.LG]`.

[146] J. Yu et al. "Scalpel: Customizing DNN pruning to the underlying hardware parallelism". In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. June 2017, pp. 548–560.

[147] Sergey Zagoruyko and Nikos Komodakis. "Wide Residual Networks". In: *CoRR* abs/1605.07146 (2016).

[148] Semir Zeki. "A massively asynchronous, parallel brain". In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 370.1668 (2015), p. 20140174. DOI: `10.1098/rstb.2014.0174`.

[149] Dongqing Zhang et al. "LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks". In: *European Conference on Computer Vision (ECCV)*. 2018.

[150] S. Zhang et al. "Cambricon-X: An accelerator for sparse neural networks". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–12.

[151] X. Zhang et al. "Towards Memory Friendly Long-Short Term Memory Networks (LSTMs) on Mobile GPUs". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2018, pp. 162–174.

[152] Shuchang Zhou et al. "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients". In: *arXiv preprint arXiv:1606.06160* (2016). URL: `https://arxiv.org/abs/1606.06160`.

[153] X. Zhou et al. "Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2018, pp. 15–28.