Title
A Novel Systolic Architecture for Efficient Acceleration of Deconvolutional Neural Networks at the Edge

Author
Daly, Jake M

Publication Date
2021

UNIVERSITY OF CALIFORNIA SAN DIEGO

A Novel Systolic Architecture for Efficient Acceleration of Deconvolutional Neural Networks at the Edge

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science

in

Electrical Engineering (Machine Learning and Data Science)

by

Jake Daly

Committee in charge:

Professor Kenneth Kreutz-Delgado, Chair
Professor Vikash Gilja
Professor Michael Yip

2021

The thesis of Jake Daly is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2021

DEDICATION

This thesis is dedicated to my best friend and fiance, Ladan Behzadi, for all her support during my time in graduate school.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to first acknowledge Ian Colbert, who's leadership, friendship, and advice have helped me at every step of my journey over the last two years.

I would also like to acknowledge Dr. Srinjoy Das and Professor Kenneth Kreutz-Delgado who's inputs and wisdom have helped shape some of the key insights in this work.

Lastly but certain not least, I would like to sincerely thank Dr. Parimal Patel of Xilinx for his generosity with his time, and for his willingness to share his deep knowledge of complex tools and features of Xilinx products. Much of our lab's work wouldn't be possible without his help.

VITA

2016	B. S. in Electrical and Computer Engineering, University of California, Santa Barbara
2016-2019	Applications Engineer, Keysight Technologies
2019-2021	Graduate Student Researcher, UCSD Calit2 Pattern Recognition Laboratory
2020-2021	Machine Learning Intern, Advanced Micro Devices
2021	M. S. in Electrical Engineering (Machine Learning and Data Science), University of California, San Diego
2021	Machine Learning Engineer, Advanced Micro Devices

PUBLICATIONS

Ian Colbert, Jake Daly, Ken Kreutz-Delgado, and Srinjoy Das, “A Competitive Edge: Can FPGAs Beat GPUs at DCNN Inference Acceleration in Resource-Limited Edge Computing Applications?”, arXiv:2102.00294.

FIELDS OF STUDY

Electrical Engineering: Machine Learning and Data Science, Computer Architecture, Hardware Acceleration

ABSTRACT OF THE THESIS

A Novel Systolic Architecture for Efficient Acceleration of Deconvolutional Neural Networks at the Edge

by

Jake Daly

Master of Science

University of California San Diego, 2021

Professor Kenneth Kreutz-Delgado, Chair

A new era of processing has dawned: the demands for low latency and low power processing at the edge have ushered in unprecedented opportunity computer architects and embedded designers. In pursuit of new performance standards, chip designers in industry and academia have begun the march towards domain specific processors, a paradigm whose core philosophy and methods are in many ways contrary to the mantras that dominate the processors seen in today’s datacenters and technology hubs. The increasing complexity of neural networks and deep learning algorithms being deployed at these edge locations has made this pursuit anything but trivial. Some of the most powerful models that we are seeing deployed, known as deep generative models, use techniques that are effectively capable of generating new data by capturing the full joint data distribution over some input space. These models frequently use upsampling layers to take lower dimensional latent spaces to higher dimensional ones before making inferences about our world. In this work, we perform a deep analysis of one of these upsampling techniques, known as deconvolution (or equivalently transpose convolution), and propose a novel computer architecture for low latency acceleration in edge applications. Our work is the first to fuse together systolic processing and an algorithmic transformation known in this area as the TDC method [3]. We illustrate how and why this pairing is so powerful for inference acceleration and provide some preliminary performance numbers benchmarked against a pre-existing Wasserstein Generative Adversarial Network (GAN).

Chapter 1

Introduction

1.1 A New Era Of Processing

The last decade has seen an explosion of interest in artificial intelligence, powered in large part by the subfield called deep learning, which has time and time again provided state-of-the-art results in tasks like image classification, language modeling, and game playing [4]. Deep learning most often refers to the training of multi-layered neural networks, which can be deployed as universal function approximators [5]. The more data that is available to train on, the more accurately these models converge to making useful predictions, especially in highly parameterized ”deep” networks where the parameter search space is very large [4].

Of course, using more data to train these models does not come without cost. For years, we had been able to meet the ever-increasing demand for computational speed by making devices smaller and by turning up the clock frequency on our processors without many implications for power density and thermal stability. This trend came to a screeching halt in the mid 2000s with the end of Dennard Scaling [6], and a new trend subsumed its place: parallelization.

As demand for parallelization increased and neural networks progressively became even more parameterized, we've seen an explosion in popularity of parallel platforms like FPGAs and GPUs being used for deep learning acceleration. The last few years we have also seen the development of even more highly specialized deep learning accelerators pop up, for example the Google TPU that aims to optimize matrix multiplication [7]. This shift to domain-specific processors has ushered in a hardware/software co-design approach [6] in which designers journey deep into the fabric of a particular application or algorithm and look for opportunities to exploit its idiosyncrasies such that both the algorithm and architecture are jointly optimized for maximal performance.

1.2 Deconvolution at the Edge

Another driving force behind the trend to domain specific processors has been the diversification of environments where applications are deployed. The same devices that accelerate and power the training of deep learning agents in datacenters have much different resource requirements than a small edge device that might deploy the same trained network for inference. Edge computing refers to computation executed on electronic devices that are connected to but distributed away from a centralized node like a datacenter [8], and we specify this context because yielding performance gains at the edge is often more complex than simply achieving higher throughput. While the user who trains the algorithm at the datacenter likely cares more about raw throughput, the end user of an edge device may care more about her phone battery lasting all day long, or her application running quickly.

This thesis focuses on accelerating the deconvolution operator in this context - low-power, resource-constrained inference. Upsampling operators are a widely studied technique [9] and can generally be separated into two groups: (1) interpolation-based methods, and (2) learnable methods. Deconvolution (also known as transpose convolution) is a learnable upsampling operator commonly used in deep generative models such as Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs) [2, 3, 10–16]. Whereas the convolution operator is used for downsampling latent spaces in locally connected neural networks, the deconvolution operator learns a higher dimensional latent space from a lower dimensional one. Unlike most other learnable upsampling operations such as sub-pixel convolutions [17], the standard deconvolution operator does this directly, without the need for expensive post-processing operations.

This work addresses the acceleration of the deconvolutional neural network layer, with a combined architectural and data flow approach, implemented on an embedded SoC platform. The main contributions of this work are:

1. An overview and analysis of existing algorithmic and architectural approaches to DCNN acceleration

2. A novel architecture and data flow for deconvolution inference acceleration

3. An efficient means of performing the TDC transformation online

Before exploring the wealth of parallelism, pipelining, and data re-use that a systolic TDC architecture affords, we provide some background on the problems that arise when accelerating deconvolution layers, and approaches that have been proposed in recent years. After developing a basic understanding of the challenges that arise, we will observe various algorithmic, architectural, and combined approaches that have been proposed in recent years.

Chapter 2

Background

The past decade has seen an unassailable increase in quality and representational power of artificial intelligence. The genesis of this can be accredited to, aside from the increased complexity of machine learning models, the extreme extent to which models are being parameterized [1]. In Figure 2.1, we show how extreme this trend has been. To put some of these numbers into perspective, the Turing-NLG model has a number of parameters (roughly 10 billion) equivalent to over 10% of the total number of brain cells the average adult human brain is estimated to have (86 billion) [18].

Some of the most impressive results that we are seeing across traditional applications like computer vision and natural language processing come from models that are classified as deep generative models. Although generative models have been around for quite some time, an important diversion in recent years (from classical approaches to training generative models) has been the injection of deep learning into the training processes. Arguably the most intrinsic power of deep learning is that the user only needs to set a relatively small number of a neural network's hyperparameters and an optimization algorithm will do the heavy lifting of fitting the model's parameters (however many there may be) such that some objective loss signal is minimized. This is in stark contrast to the parameter estimation techniques of classical generative modeling and Bayesian statistics in which parameters are tuned by hand, evaluated, and iteratively improved upon.

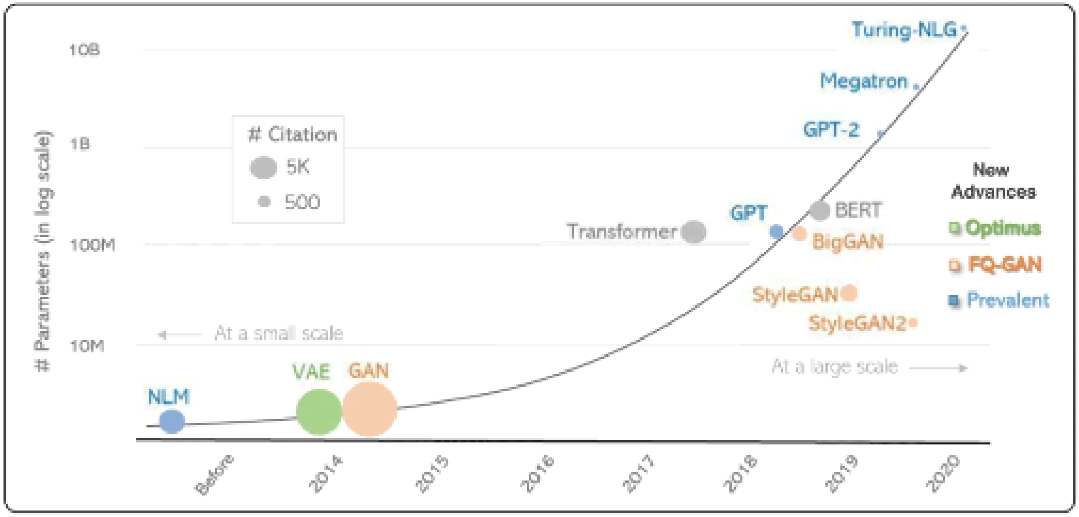


Figure 2.1. The increased capability and power of deep learning models over the past decade has been due in large part to the exponentially increasing number of parameters that these models use. Image source: Li and Gao [1]

A classic example of a deep generative model is a Generative Adversarial Network (GAN) [19] in which a discriminative model such as a convolutional neural network (CNN) is for example trained to extract features from some input image, and compress this information into a latent space. A generator is then trained to use some upsampling technique to decompress these features into a higher dimensional reconstruction of the latent information that was present in the original training data. Whereas one popular way of performing the *downsampling* or compression of information is via the convolutional operation, *upsampling* is frequently performed through the transpose convolution operation (also known as deconvolution). Before understanding the biggest problems that deconvolution poses for hardware systems, we need to review the nature of the operation.

2.1 Deconvolution Operator

The deconvolution operator can be viewed as the operation *inverse* to the traditional convolution operator: it is often used after a convolution has occurred and the original input shape is trying to be recovered [20], or more generally when we are trying to upsample from some latent space. The deconvolution operation is often implemented in practice as a neural network layer in deep generative models like GANs [9,21–25]. If we let O_C be the number of output channels of the deconvolution, and I_C be the number of input channels, we obtain each output channel by taking a set of I_C kernels, deconvolving them against the given input feature maps, and summing them per output channel, as shown in Figure 2.2. In this example, each of the eight input feature maps deconvolves with a total four kernels and the resulting outputs are summed across the input dimension (right to left in this figure) to obtain the four output feature maps.

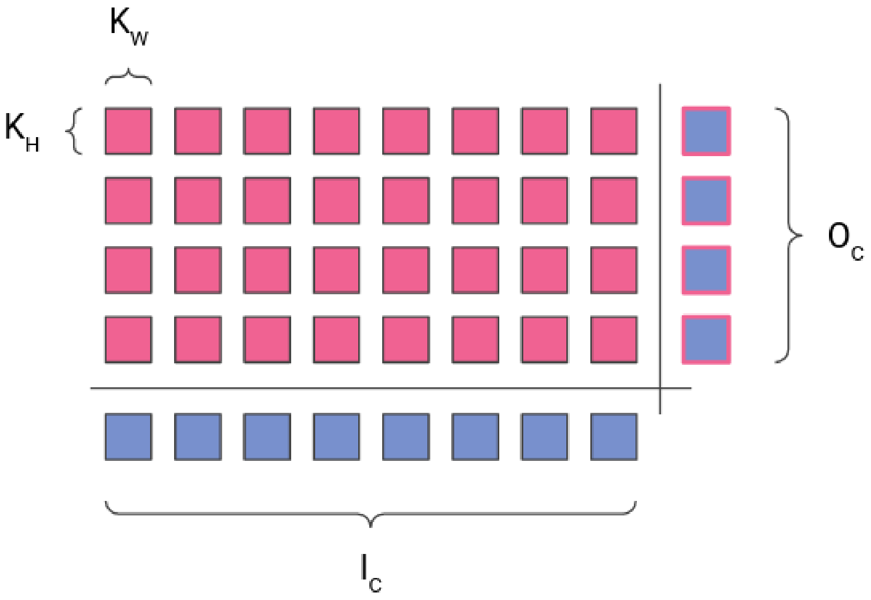


Figure 2.2. Visualization of layer-level geometry of a deconvolution, where pink tiles represent kernels (weights), blue tiles represent input feature maps, and blue tiles with a pink border represent output feature maps.

2.2 Overlapping Sums Problem

As shown in Figure 2.3, the traditional deconvolution operation involves multiplying an input element (for example a pixel in an image) by every element in a kernel, and storing this projection in some output buffer (t1). At the next time time step, the filter moves across the input to the next element, projecting it on to the output at an offset of stride S from where it had stored the previous output (t2). If S is smaller than the width of the deconvolution kernel K_D, the output of the operation ends up overlapping with the output of previous time steps (t2-t4). By the end of the computation, after the filter has deconvolved over the entire input, we may end up with an output that has a non-uniform number of sums of products.

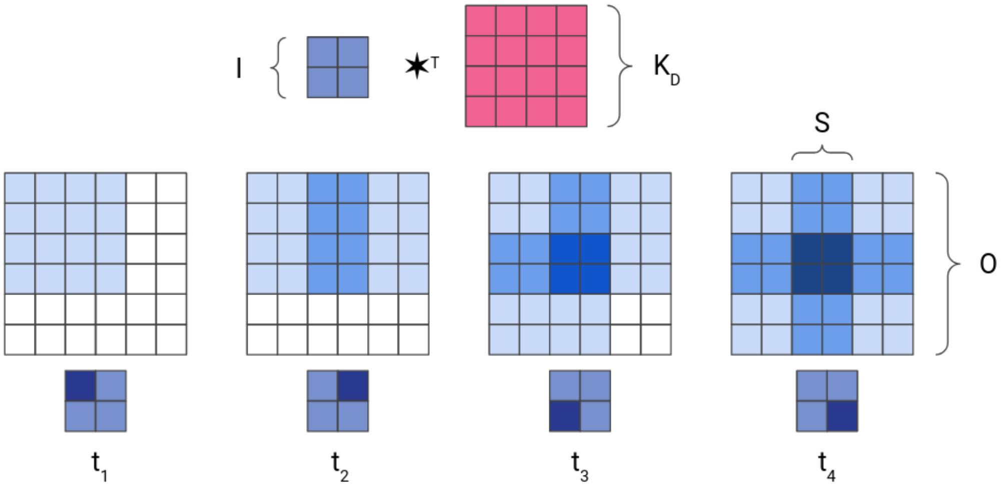


Figure 2.3. Traditional deconvolution algorithm, shown with I = 2, K_D = 2, S = 2, and O = 6. At steps t_2, t_3, and t_4 overlapping sums occur between adjacent output products, as indicated by darker regions of blue.

The need to accumulate and re-write the same locations in the output causes a big issue of efficiency and synchronization for the underlying hardware, known in the literature as the ”overlapping sums” problem [2, 3, 10, 13, 14, 16]. The inefficiency results from having to either send the incomplete, intermediate results off-chip (a waste of memory bandwidth and energy) or cache the intermediary products on-chip (a waste of space and on-chip resources).

2.3 Input Data Re-Use vs. Output Data Re-use

Data re-use refers to using the same data multiple times while it is in quick-access memory (eg. registers), rather than having to fetch this data multiple times from slow memory (eg. DDR). In an ideal world, we would like to maximize data re-use by loading data once, performing all the necessary operations on it, and then discarding it when we're done. Figure 2.2 shows why this is not always possible, especially for large networks with many input and output channels: we would ideally like to use each of the eight input feature maps across all four filters (ie. maximize data re-use over the input space). If we could finish all computations that will require each input feature map, we wouldn't have to load it again from DDR at a later point. However, if we use the input feature map against all of the kernels, we would have to store all output products somewhere, which can quickly become an overwhelming amount of data. If we consider layer 5 of the well known PGAN-LSUN network [26], this would require caching over 16 MB of data *per training sample* (assuming 32-bit data).

On the other hand, we could instead try to maximize data re-use across the output space by trying to completely finish one output feature map. Doing so would require loading each input feature map one by one, and then performing the deconvolution of each with its first kernel. The issue with this is we will have to end up loading each of the input feature maps a total of 4 times. Whether it's more efficient to maximize data re-use over the input space or the output space often comes down to the specific geometries of a network layer. For example a small neural network layer might have few enough parameters to where maximizing re-use over the input space doesn't place to high of a burden on the system. Finding a more fixed source of data re-use is, as we'll see, an easy task for a TDC/systolic combined approach.

2.4 Coarse Overview of Acceleration Approaches

When considering approaches to designing a DCNN inference accelerator that addresses the issues of overlapping sums and making efficient use of data, techniques can generally be defined by their algorithmic or architectural features [2]. Almost all research in this area reflects the more general trend discussed earlier of a domain specific, hardware / software codesign approach.

In this thesis, we define algorithmic approaches as ones that take the viewpoint of making changes to how the computation gets executed, whereas architectural approaches we define as optimizing the underlying dataflow and hardware execution units that an algorithm would require to execute. Designing at the hardware/software interface can blur the lines between what we would consider an algorithmic versus an architectural approach. A guiding principle is that if the technique could be implemented in software or a higher level programming language, it is a an algorithmic approach. For this reason, we use the term ”software-based” approaches synonymously with algorithmic approaches, and ”hard-based” approaches synonymously with architectural approaches. The codesign of the hardware and software approaches that we will discuss next can unlock powerful sources of efficiency. Before examining recent approaches to the DCNN accelerator, we will introduce some common techniques that are seen across works.

2.4.1 Algorithmic Approaches

One way to view the overlapping sums problem is that the vanilla deconvolution operation does a ’single shot’ sweep over the input space. When we say single shot over the input space, we mean that the algorithm loops over the input space, only visiting each element once. This contrasts to what would occur over the output space, where pixels might have overlapping projections onto them, and thus might be accessed more than once. Algorithmic approaches to efficient deconvolution acceleration most often champion the idea that being able to compute elements in the output space in a single shot would circumvent the overlapping sums issue.

Reverse Deconvolution

For example in Figure 2.4, we can imagine stepping over the output and asking what inputs and weights would be required to complete this output element? The authors of [2, 10] proceed in this manner, calling their method REVD (reverse deconvolution), such that each completed value could be streamed off as soon as there were enough data to make efficient use of the memory bandwidth, avoiding the overlapping sums issue.

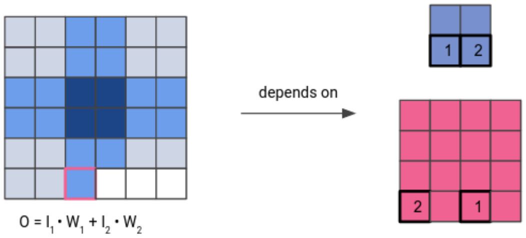


Figure 2.4. Reverse deconvolution loops over the output space and for each element, calculates the inputs and weights that are required as part of that element's sum of products.

Sparse Convolution

Another method that has been employed [11, 12, 27, 28], leverages the notion that any deconvolution can be viewed as a sparse convolution in which zeros are inserted in between input feature map elements (”fractional striding”). This way we avoid the issue of overlapping sums because the convolution operation inherently completes the output in a single shot fashion. This is shown in Figure 2.5 where we have transformed the same deconvolution that we have been using throughout the chapter. Designers employing this technique have chosen, rather than grappling with overlapping sums, to instead fight the battle of sparsity: the sparse input feature maps that are obtained when using this approach cause many wasted operations (due to zero multiplication and zero padding). To compare this with the example deconvolution we observed earlier, there are 1152/128 = 9x as many operations, of which only about 11% are non-zero operations. For problems with 2 dimensional deconvolutions (eg. any problem where we are dealing with image data), the sparse convolution transformation has a number of zero multiplications that grows quadratically with stride length, and superlinearly in the size of the deconvolution kernel (due to padding).

TDC Transform

Another algorithmic approach is to alternatively view the deconvolution as a set of S^2 smaller, dense convolutions, We obtain the convolution kernels by sampling the deconvolution

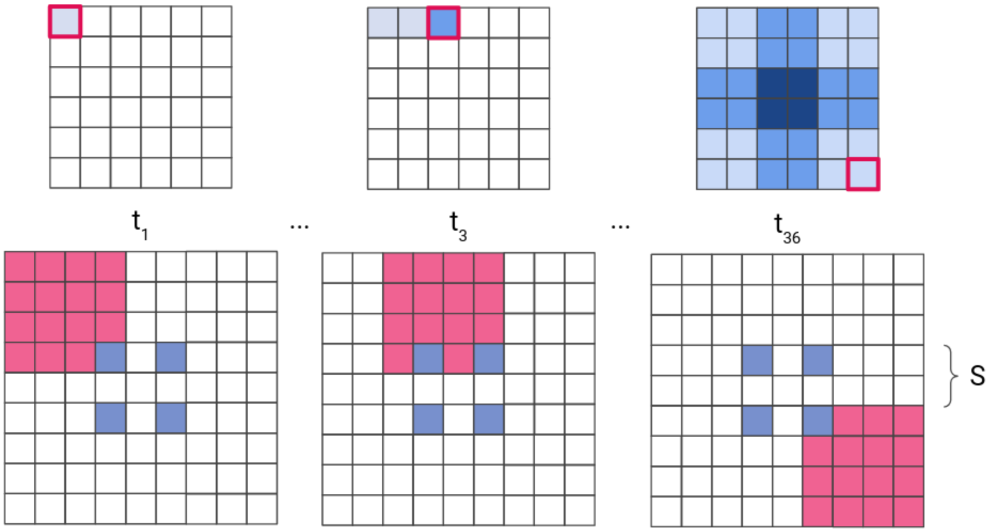


Figure 2.5. Transforming a deconvolution operation into a single sparse convolution.

kernel at stride S, and rearranging the resulting groups of samples into smaller kernels that we then convolve with the input. This method has been referred to as the TDC (**T**ransform **D**econvolution to **C**onvolution) method [14]. We will not derive the algebra that is required to perform this transformation in this work, as this has been the focus of other works [14, 15], and also because the lower level details of the transformation aren't relevant to understanding how it can be used to aid in the acceleration of a deconvolution workload.

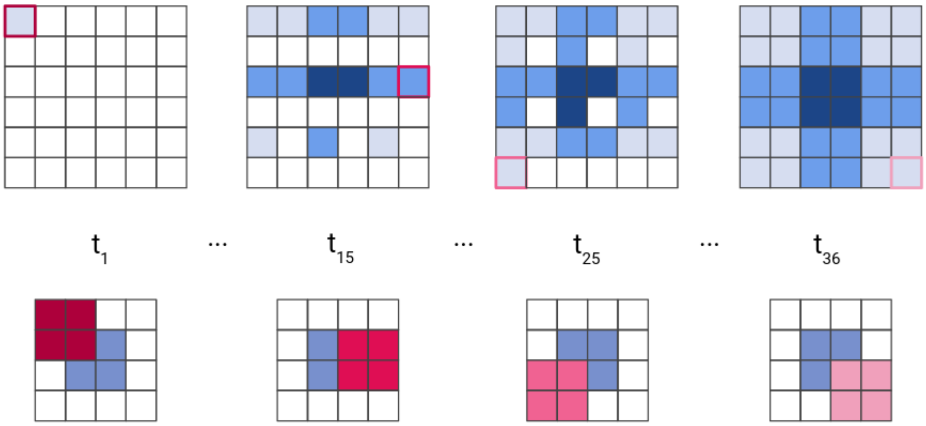


Figure 2.6. Transforming a deconvolution operation into a convolution using the TDC transform.

Figure 2.6 depicts the TDC algorithm. The deconvolution kernel gets sampled at stride S, effectively splitting it into S^2 convolution kernels. If the size of the deconvolution kernel is not evenly divisble by the stride, padding is required such that $P_{K,C} = K_C \cdot S - K_D$, where $P_{K,C}$ is the padding required for the transformed convolution kernels, K_C is the width of the new convolution kernels, S is the stride of the deconvolution operation, and K_D is the size of the original deconvolution kernel. These S^2 transformed filters are convolved across the input, which is padded with $K_C - 1$ zeros.

Because the resulting transformation requires invoking a convolution operation in place of a deconvolution, each output element is able to be computed in a single-shot. Although the TDC transformation does result in more computational steps than the traditional deconvolution 2.3, each computational step involves fewer operations because the convolution kernels are smaller than the original deconvolution kernel. With the toy example we have provided, there are 4 elements * 2 operations (one addition and one multiplication) * 36 time steps = 288 total operations, whereas the traditional deconvolution had 16 elements * 2 operations * 4 time steps = 128 operations, and the sparse convolution had 16 elements * 2 operations * 36 time steps = 1152 operations. The sparse convolution and the TDC method both circumvent the overlapping sums problem at the expense of increased operations (compared to the traditional deconvolution); however for the vast majority of layer parameters, TDC will require less operations than a sparse convolution because there is no need for zero insertion, and the padding scales with K_C rather than with K_D.

The elementary example provided in this section highlights the trade-offs that different algorithmic approaches pose. Even though one approach might expose (for example) more operations than another, we cannot say for certain whether that approach will be less efficient until we know more about how the underlying architecture can be hindered by or exploitative of these idiosyncrasies. In domain specific hardware and software codesign, we indeed would only choose one of these algorithmic approaches if we had an associated strategy that would allow us to take advantage of these attributes. This is precisely why a deep understanding of the interplay between architecture and algorithm are essential to implementing a high-performing accelerator. In the next section we give some background on common architectural approaches employed for deconvolution acceleration.

2.4.2 Architectural Approaches

The most canonical paradigm for data-parallel processing is single instruction, multiple data (SIMD) processing, in which the same instruction is issued across an array of processing elements and applied to a vector of data in parallel. Of course this method of acceleration works best for data that can be partitioned into independent chunks, which is often the case in deep learning when the majority of the computation that needs to occur is matrix multiplication. The most famous example of a SIMD architecture is a GPU, which until recently [7, 29] has been unrivaled in throughput. GPUs have been shown in recent years to have comparable or in some cases worse performance per Watt in deep learning applications compared to TPU [29] and FPGA [2].

The Tensor Processing Unit [7] (TPU) has made arguably the biggest splash in the realm of hardware accelerators in recent years. The underlying architecture is classified as a complex instruction set computer (CISC) and uses a systolic array processor to achieve throughput numbers even higher than GPU [30]. Because systolic arrays are a large part of the architecture employed in this work, we dedicate a whole section to explaining the basic concepts behind them (Section 2.5).

FPGAs are yet another type of processor that is widely employed in this area as they can be reconfigured and highly customized to do very specific tasks; they are of particular interest in the edge computing space because of their relatively high performance per watt [2]. One obstacle to designing with FPGAs is their slow design cycle and steep learning curve, however high-level synthesis (HLS) tools have come a long way in recent years and aim to lower the barrier to getting FPGA systems up and running quickly.

2.5 Systolic Arrays

The term systolic in the context of the human cardiovascular system describes a specific phase in the heart's cardiac cycle in which the muscle contracts, causing a 'systolic' pressure which results in blood being pumped to the body's organs [31]. The word invokes an analogous meaning in the context of parallel computer architectures: data is pumped through a (typically) 2D array of homogeneous processing elements, where each processor performs some operation on the data and passes the original data and/or processed data onto neighboring elements [32]. This contrasts to other well known architectures like CPU and GPU in which data moves back and forth between memory (most often registers) and a processor.

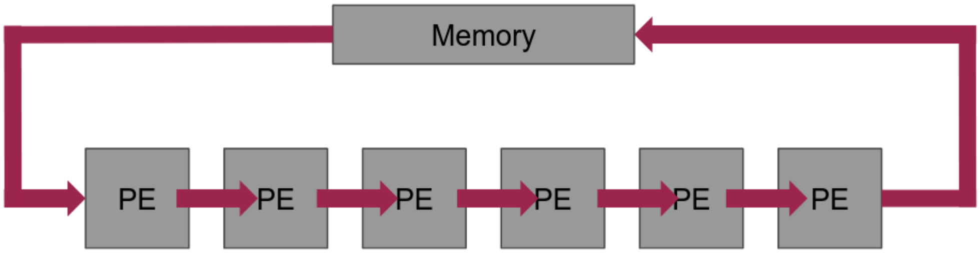


Figure 2.7. The notion behind a systolic architecture is analogous to the circulatory system in that dataflow (red) is through processing elements (PEs) before returning to memory, similar to how blood moves away from the heart (due to systolic pressure) and through organs before returning.

The systolic array philosophy is rooted deeply in two of computer architecture design's most fundamental principles: keeping architecture simple and keeping architecture small [33]. From a pure design perspective, keeping the systolic array's processing elements simple and small enables high modularity and uniformity across the units, resulting in a processor that is easy to implement and easy to adapt to different constraints [34].

More importantly than the design advantages systolic architectures pose, they are amenable to high degrees of pipelining (and thus can be run at high speeds) as well as huge data re-use [35]. These two factors coupled together imply the potential for high-speed data movement amongst processing elements, and efficient re-use of data–especially in algorithms that use the same data multiple times across their computation (for example convolution as we will see shortly). All of these would suggest a systolic architecture could be an excellent candidate for a low-latency accelerator.

Chapter 3

Existing Approaches to Deconvolution Acceleration

As cited previously, deconvolution acceleration has received a noticeable amount of attention in recent years. Most work discussed in this section builds off of the general approaches discussed in the Chapter 2, but here we elaborate on how specific approaches have been adapted and augmented over the past five years.

Acceleration via Reverse Deconvolution

One of the earliest works in the space [10] developed the reverse deconvolution algorithm, or REVD, which computes the output feature maps in a single shot, avoiding the overlapping sums problem. The authors propose a three-step design methodology to systematically optimize an accelerator to achieve optimal roofline performance by leveraging data statistics and design space exploration. The biggest criticism of the REVD method has been that the algorithm loops over the output space, and requires several expensive calculations (fixed point modulo arithmetic) on each loop iteration to obtain the required input and weight indices [3]. Colbert *et al.* [2] address the expensive address calculation arithmetic by pre-processing and caching the required calculations. This work also improves the dataflow by reordering the loops, effectively improving data re-use in the weight space and increasing the impact of their conditional execution units.

Acceleration via Sparse Convolution

It has been long known one alternative way to view deconvolution is as a sparse convolution, as we covered in Chapter 2; however, it hasn't been until recent years that works began exploiting this to avoid the overlapping sums issue. [11, 12] focus on the deployment of a deconvolution accelerator for use in GANs. Their main focus is creating an architecture that efficiently deals with the sparsity by rearranging and eliminating rows of zeros that would result in wasted computation. The resulting processor, GANAX, uses a unified MIMD-SIMD architecture that can operate in two distinct modes: it operates in SIMD mode when performing standard convolutions (required in the discriminator network of a GAN), meaning that all processing elements execute instructions that have been stored in a global instruction buffer. However, it can switch into a combined MIMD-SIMD mode when required for executing transpose convolutions, which potentially have a different number of operations per convolution window. They develop a custom instruction set for carrying out these two modes of operation, and benchmark a handful of popular GANs on their architecture. A key disadvantage of this approach is the complexity of the architecture, which switches back and forth between modes of operation and maintains a hierarchy of different instruction buffers to execute in one mode verse the other.

Xu et. al [27] investigate turning the deconvolution into a sparse convolution and accelerating it on unmodified processors using software, attempting to avoid the development of specialized hardware. They benchmark their method against off-the-shelf processors on a set of realistic benchmarks and achieve reasonable (2.41x - 4.34x) speedup using a purely software approach.

Another work employing the sparse convolution transformation is [28], who–similar to this work–employ systolic processing in their compute engines. A main goal of their work is to realize a uniform architecture for acceleration of 2D and 3D deconvolutional nets via a novel mapping strategy that tiles larger networks to efficiently execute on their architecture. Their accelerator significantly outperforms a CPU in terms of raw throughput, and outperforms both a

CPU and GPU in terms of energy efficiency.

Acceleration via TDC

Transforming the deconvolution into a set of S^2 smaller convolutions is a more recent idea that started circulating about a year after the reverse deconvolution algorithm was proposed [3,14]. The original authors targeted this idea at image super resolution, using Vivado HLS and a Virtex 7 FPGA, and claim an 81x speed up over the traditional DCNN algorithm, presumably using HLS and the same FPGA. Tu [15] also uses the TDC method targeting GANs on an unmodified CNN accelerator. The work of both Tu and Chang *et al.* perform the transformation of the deconvolution kernels into convolution kernels off-line, which creates extra pre-processing away from the accelerator, and is less end-to-end in the context of a full deconvolution accelerator.

Chapter 4

Proposed Architecture & Dataflow

The proposed approach in this work weds the extra sources of data level parallelism that have been exposed by the TDC method with the systolic architecture. In this chapter we elaborate on the synergistic relationship between the algorithm and architecture. Figure 4.1 shows a high-level overview of the system.

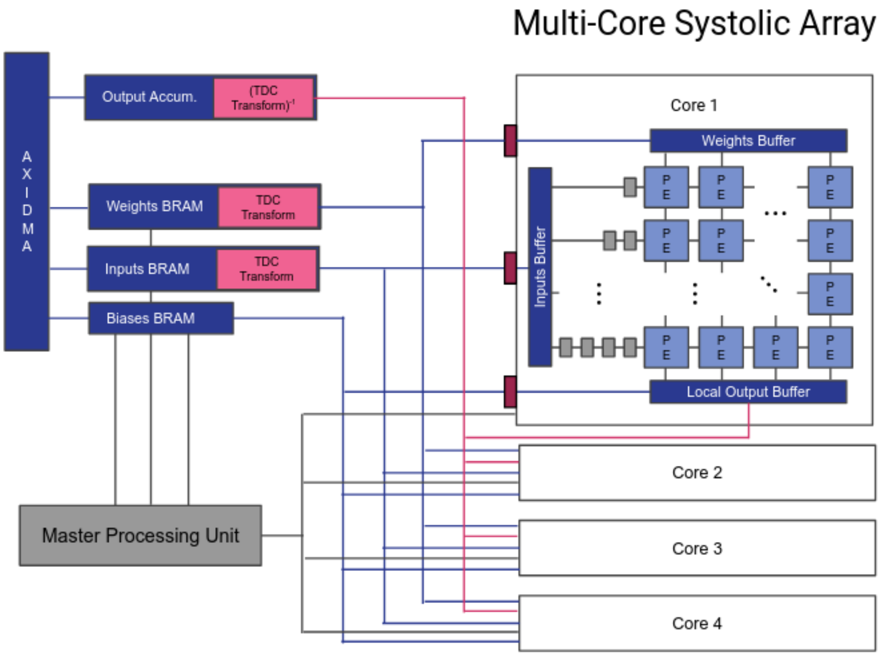


Figure 4.1. Proposed architecture with on-line TDC transform/inverse transform, and 4x parallel systolic processors.

The main philosophy of the multi-core systolic array is to have as much of the transporta-

Algorithm 1. Deconvolution Kernel. Each kernel starts the necessary data streams, then loops over output and input channels, sending each deconvolution to one of the systolic cores in a round-robin style.

```
procedure DECONVOLUTION
    StartDataStreams()
    Processor = 0
    for $o_c$ in range$(0, O_c)$ do
        SendBias(Processor)
        for $i_c$ in range$(0, I_c)$ do
            for $w_r$ in range$(0, W_R)$ do
                SendWeightRow(Processor)
            for $i_r$ in range$(0, I_R)$ do
                SendInputRow(Processor)
            WriteOutputAddress(Processor)
            Processor = Processor + 1
            if Processor == numberProcessors then
                Processor = 0
```

tion and overhead processing as possible isolated away from the master control unit, so that it can focus on keeping the cores and on chip data transport network as occupied and busy as possible. This strategy of partitioning the memory accesses away from data processing is also known as decoupled access execute [12].

4.1 On-chip acceleration of TDC

As we saw in Chapter 2, in order to transform the deconvolution to an equivalent convolution operation, the TDC method requires sampling of the deconvolution kernels and rearranging into smaller convolution kernels. This pre-processing step involves much more irregular memory access patterns compared to convolution or deconvolution operation, which in their standard forms have a sequential pattern [20]. In previous TDC works [14, 15], this pre-processing step (as well as the post-processing step that is required to 're-stitch' the split convolutions back into the deconvolution output) has been performed offline. We are the first work (to our knowledge) to build custom IP to perform this transformation and inverse transformation within the accelerator.

In large part, the transformation is applied to the weights and is done right after data is read on chip, before it is sent to the systolic cores. As data is being read into on-chip block RAM, a TDC translator waits for a full weight map to become available–this is the minimum size that would be required to fully transform a deconvolution kernel into all completed (S^2) convolution kernels. Once this condition has been met, the translator samples this portion of the block RAM according to the parameters of the deconvolutional layer, and stores them in a smaller buffer where they are ready for the master control unit to issue the instruction that transports the weights to one of the systolic cores. Because this transformation is automatically applied to weights as they are read into the design, the master control unit is not burdened with this overhead, and can focus its efforts on getting data to the different cores quickly and efficiently

For the inputs, the transform might require padding if the length or width of the transformed convolution kernels is greater than one. Rather than sending the padded input over the communication network to the cores (which would be wasting lanes on sending zero data), we initialize a buffer in the systolic core with zeros, and then implant the actual input feature map at the positions in the buffer that would be required to obtain the necessary padding.

The last step in moving data from the BRAMs to the processing cores is to move the bias (if one is required), and for the master control unit to write the output address to the core. After all of this information has been transformed and read into one of the systolic cores, a state machine is triggered which handles the cores configuration and computation so that the main controller can (in parallel) start preparing the next core for computation.

On the output side, the split outputs [15] are accumulated into a local output buffer where they are stacked one after another. When the global output buffer is ready (ie. no other cores are communicating with it), the local output buffer at the core starts reading the output data to the global output buffer, sampling in a pattern necessary to re-stitch the split outputs back together into the complete deconvolution output.

4.2 Multicasting input network

To transport the data efficiently from the global data buffers to the processing elements, we implement a multi-casting network inspired by Eyeriss [36]. The basic idea is that each processor is assigned a tag, and when we need to move data to one of the processors, we read it from the global data buffer and attach a tag to it containing the processor ID of the destination processor. The communication network broadcasts the tagged data to all processors, but any processor whose ID does not match the tag on the data will reject it. The multi-casting network contains separate buses for each of the input, weights, bias, and output address data, to ease the complexity of the routing.

4.3 Systolic State Machine Controllers

To further offload the amount of work that the master control unit has to do, we design each systolic core to be controlled by a state machine which completely controls the data movement and computation within the core once the state machine has been kicked off. The state machine's operation and sequence of events are described by the following states:

1. **Reset.** There are two types of resets which could put the systolic state machine into a reset mode: either a global system reset, or a local reset that occurs automatically when this sequence has terminated.

2. **Ready.** This state is defaulted to automatically one clock cycle after a reset, and indicates that the core is ready to be communicated with (ie. have any sort of data sent to it from the master control unit).

3. **Communicate.** Once the master control unit writes any of sort of data (weights, bias, input, or output address) to one of the cores, the systolic state machine enters into its communication state.

4. **Shift weights.** After all of the weights have been read into the local weight buffer in a core, the core will begin it's weight shifting phase in which weights are shifted downwards and stored into the local registers of the processing elements. The systolic core was designed to be able to still receive other data during this phase so that the weight shifting could be overlapped with other data communication, effectively eliminating the cost of shifting weights into the array.

5. **Compute.** After *all* data has been received by the core and all the weights have been shifted into the array, the compute phase kicks off, in which inputs are read from the input buffer on every clock cycle into the array (for full details on dataflow, see Section 4.4). This state finishes when all S^2 convolutions (ie. one complete deconvolution) has finished writing into the local output buffer.

6. **Idle.** The core then waits in an idle state for the global output buffer to signal that it is ready to communicate with the core.

7. **Write.** In this state, the core first writes the output address to the module controlling the global output buffer. After this module checks where the data is being stored, it signals back to the core that it is ready to receive data. The core then writes all of the data contained in its local buffer to the global output buffer, and when it has completed this writing, it automatically enters a local reset in which variables are re-initialized to their starting values so that core is ready for its next computation.

4.4 Systolic flow for TDC

As mentioned earlier, TDC exposes a new source of data-level parallelism that is not available in any of the other algorithmic transformations mentioned in Chapter 2. **Aside from enabling S^2 new sources of spatial and temporal parallelism, we also see an equivalent gain in data re-use: each these S^2 TDC kernels are convolved with the same exact input**

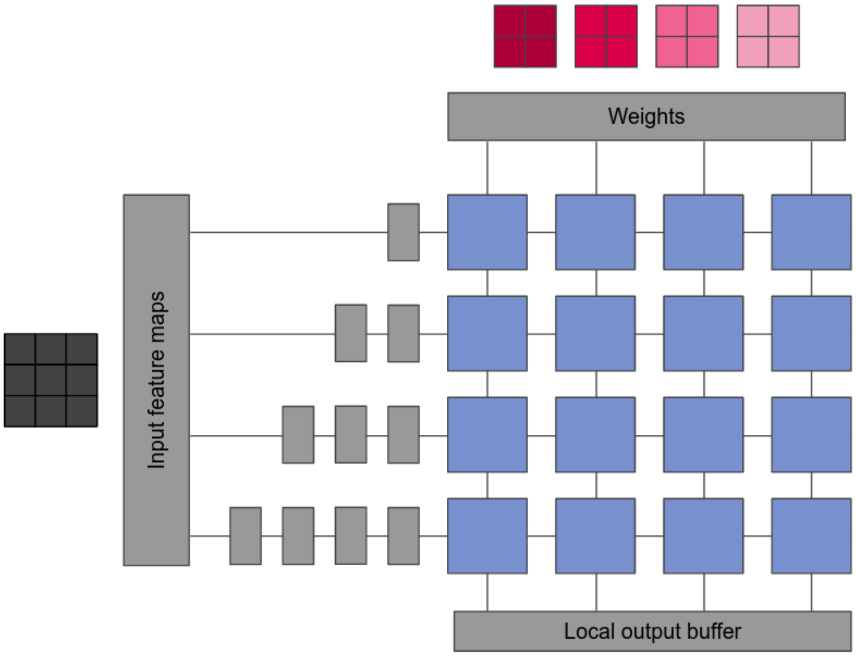
elements. To illustrate this, we will show the computation being performed for the first layer of WGAN-MNIST.

Figure 4.2a shows the dataflow through the array, which can be classified as a weight stationary dataflow [36] because the weights are shifted in to the processing elements and remain stationary until the computation completes. The processing core itself is organized as follows: each core contains an array of processing elements K_C^2 rows long by S^2 columns wide. Each of the S^2 transformed kernels are shifted into the columns of the array, such that each of their K_C^2 elements reside in one row of the column, and that the columns preserve the same order of kernel elements; for example, each column in the Figure 4.1b (from top to bottom) contains the elements $w_{c,1,1}, w_{c,1,0}, w_{c,0,1}, w_{c,0,0}$, where $c \in [0,3]$ represents any of the four transformed convolution kernels.

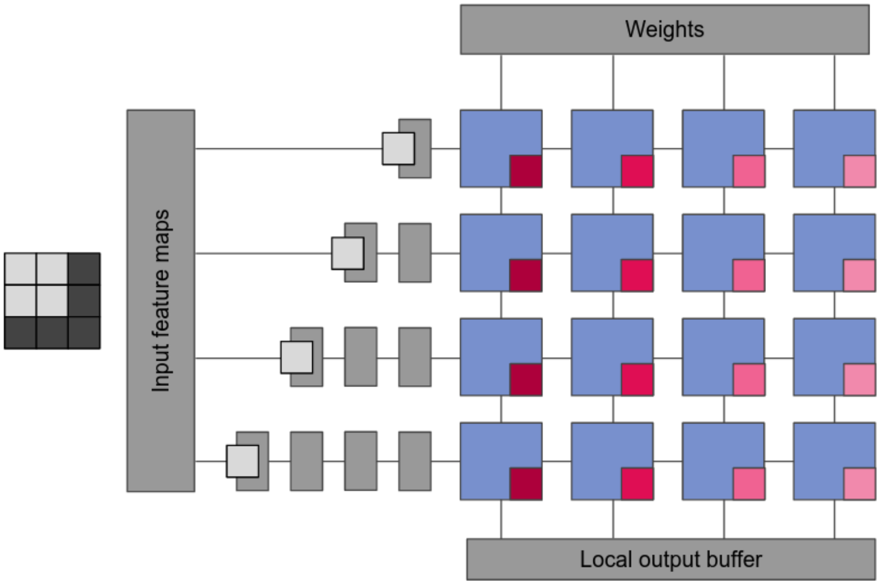
After filling the array with weights, input feature maps begin streaming in from the input buffer on the left 4.0c. Each clock cycle, K_C^2 inputs are read out of the input buffer, which correspond to the first matrix multiplication of each of the S^2 convolutions. Immediately after being read out from the input feature map buffer, each read value will hit a number of delay elements equal to the row that it has been read into. These delay elements allow the remaining three products of the first column's *first* matrix multiplication to meet with the accumulating value as it passes down the first column on the correct clock cycle. With each pass, it picks up one of the remaining values, until it reaches the final processing element, where the final accumulated value containing all four products is stored in a local output buffer until the processing cycle completes 4.-1d. Due to the massive opportunity the systolic array provides for pipelining computation, the remaining three matrix multiplications that are required to complete the convolution of i and $w_{0,:,:}$ are trailing directly behind this one.

Meanwhile, while the products of a given matrix multiplication are being accumulated and passed downwards, the input data shifts from left to right across the rows of the array. If we look again at 4.-1d, input $i_{0,0}$ will be re-used across all four convolution kernels. With each advancement of $i_{0,0}$ into a new column, a new matrix multiplication is started between the c^{th}

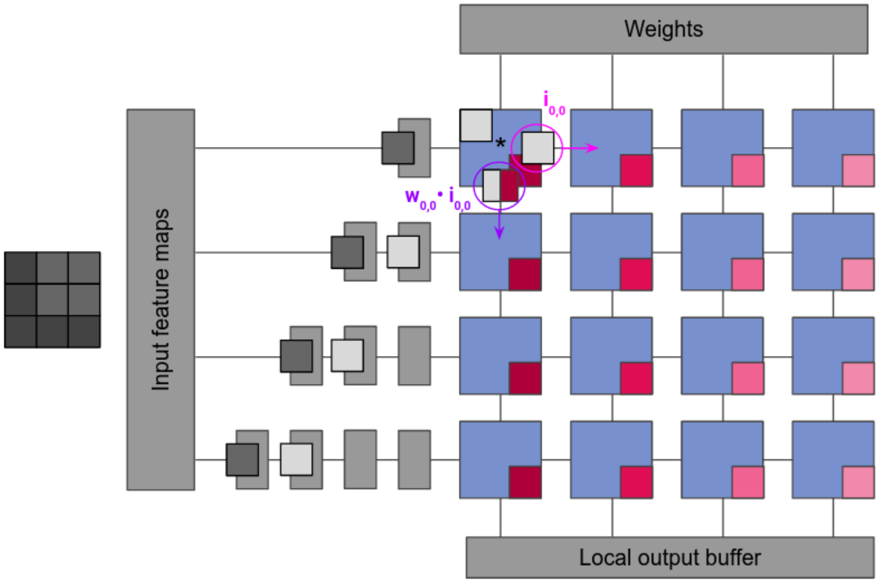
convolution kernel and an equally sized portion of the input feature map. As a result, the entire computation shown in the first column of 4.-1d is effectively replicated in every column against a new convolution kernel, with the only difference being the increasing delay.



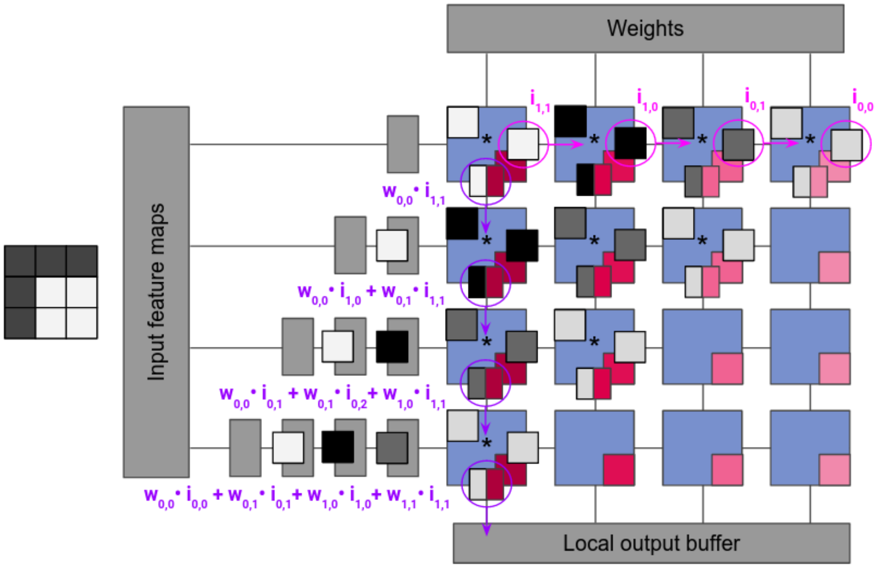
(a) Systolic array after weights and inputs have been loaded into their buffers.



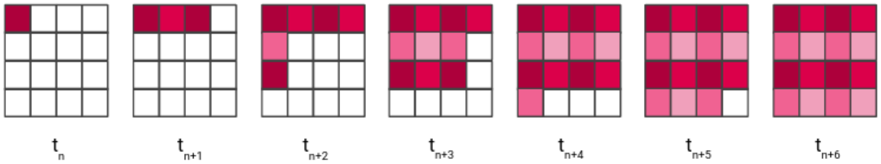
(b) After the weight buffer is full, the local systolic controller will shift the weights into the columns of the systolic array and then begin shifting in the inputs. The first 2x2 input is shown being loaded here, which will be used across all S^2 convolution kernels.



(c) As the weights start shifting into the array, they are delayed according to their row so that the resulting products meet up with their counterparts on the appropriate clock cycle.


Weights
$i_{1,1}$
$i_{1,0}$
$i_{0,1}$
$i_{0,0}$
Input feature maps
*
$w_{0,0} \cdot i_{1,1}$
$w_{0,0} \cdot i_{1,0} + w_{0,1} \cdot i_{1,1}$
$w_{0,0} \cdot i_{0,1} + w_{0,1} \cdot i_{0,2} + w_{1,0} \cdot i_{1,1}$
$w_{0,0} \cdot i_{0,0} + w_{0,1} \cdot i_{0,1} + w_{1,0} \cdot i_{1,0} + w_{1,1} \cdot i_{1,1}$
Local output buffer

(d) The first output of the array shown here is the first matrix multiplication of the first convolution. In the next 3 clock cycles, this same column will produce the remaining three matrix multiplications to complete the first convolution.



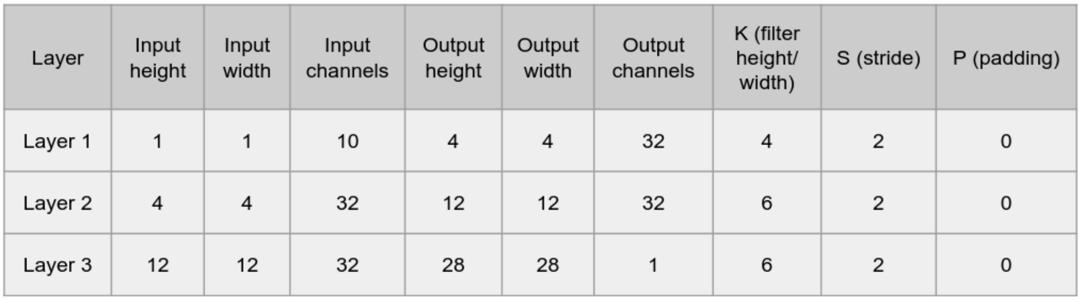
(e) If (d) shows the first output being completed at time t_n, here we see which outputs complete (and the corresponding convolution filters they belong to) on the following clock cycles until the computation has finished all S^2 convolutions (equivalent to 1 deconvolution)

Figure 4.-2. Data flow through the systolic array (b,c, and d), demonstrating which output products are accumulating at various stages (c, d) and the sequence of completed matrix multiplications writing to the local output buffer (e).

Chapter 5

Experiments

We designed the entire architecture using Verilog, and implemented the design using Vivado for the Xilinx PYNQ SoC, which has a dual ARM core CPU and programmable logic roughly equivalent to the Artix-7 FPGA.

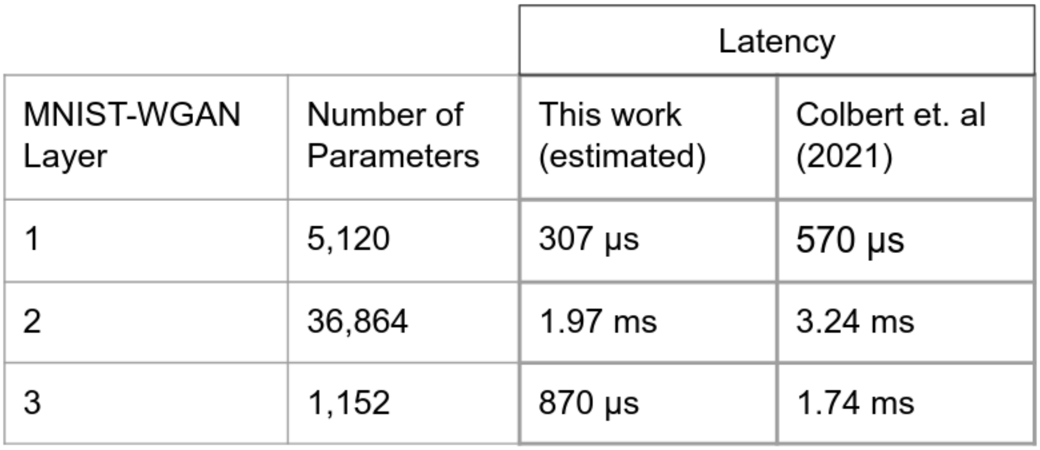


Layer	Input height	Input width	Input channels	Output height	Output width	Output channels	K (filter height/ width)	S (stride)	P (padding)
Layer 1	1	1	10	4	4	32	4	2	0
Layer 2	4	4	32	12	12	32	6	2	0
Layer 3	12	12	32	28	28	1	6	2	0

Figure 5.1. Layer level parameters for the implemented WGAN-MNIST network.

The network we benchmarked our architecture with is WGAN-MNIST, a Wasserstein GAN trained to generate the popular dataset of handwritten digits that was also used in [2] (the network has the layer level parameters shown in Figure 5.1). To make the most efficient use of the processing elements within the systolic array in terms of utilization, we use an array width equal to S^2 elements (this allows all S^2 convolution kernels to be fit across the columns of the array, as was shown in the data flow example in the 4). Through experimentation, we find that using four separate systolic cores is the minimal number which allows the master control unit to never be standing idle waiting for a core to be ready. Taking all of this into consideration, we only use roughly 30% of the FPGA's DSP slices, a relatively small number on an already small, edge FPGA.

We obtain the estimates for latency (Figure 5.2) by taking the simulated network runtime, and adding to it twice the average time that was measured to write over the memory mapped IO (MMIO) interface between the processing system and programmable logic. The MMIO interface is faster than the general purpose IO (GPIO) interface because it allows the user to write directly to the memory mapped portion of the address space. The estimates shown were for an operating frequency of 50 MHz, which leaves us optimistic that we will be able to reduce the latency even further when we look to optimize and pipeline circuit paths in future work.



		Latency	
MNIST-WGAN Layer	Number of Parameters	This work (estimated)	Colbert et. al (2021)
1	5,120	307 µs	570 µs
2	36,864	1.97 ms	3.24 ms
3	1,152	870 µs	1.74 ms

Figure 5.2. Performance estimates for latency obtained for the WGAN-MNIST network, in comparison with previous work [2]

Chapter 6

Conclusions & Future Work

In this work, we highlighted some of the challenges of deconvolution accelerator design, and showed the merits and drawbacks of existing approaches. We analyzed some of the common algorithmic transforms that are performed to rearrange the deconvolution into a form that is easier to design hardware for, and then observed how various architectures look to exploit these transformed dataflows. Our proposed solution binds one of these transformed dataflows, the TDC transform, with a systolic array architecture that is able to take full advantage of the extra sources of parallelism that the TDC method creates, while simultaneously capitalizing on strong data re-use and high degrees of pipelining. All of these advantages have resulted in an improved latency over a similar work (employing a different architecture and dataflow but using the same network) by an average of 45% across all layers.

In future work, we look to make several optimizations to the architecture. The first optimization we will look to make is to increase the pipelining in the master control unit where the longest combinational circuit paths were, which prevented us from passing a static timing analysis at a higher frequency. Our design was .01 ns away (WNS) from passing a static timing analysis at 100 MHz operating frequency, so we are optimistic that with little effort we will be able to increase this frequency to 100 MHz and beyond. We will also look to unroll the loops in our assembly code to cut down the overhead of the loops in our program, which we suspect will further cut down latency. We will also start benchmarking the design against larger networks with many more parameters, so we can quantitatively assess how well the architecture scales.

Bibliography

[1] C. Li and J. Gao, "Scaling up-researchers advance large-scale deep generative models," Apr 2020.

[2] I. Colbert, J. Daly, K. Kreutz-Delgado, and S. Das, "A competitive edge: Can fpgas beat gpus at dcnn inference acceleration in resource-limited edge computing applications," arXiv preprint arXiv:2102.00294, 2021.

[3] J.-W. Chang and S.-J. Kang, "Optimizing fpga-based convolutional neural networks accelerator for image super-resolution," in Proceedings of the 23rd Asia and South Pacific Design Automation Conference, pp. 343–348, IEEE Press, 2018.

[4] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," arXiv preprint arXiv:1703.09039, 2017.

[5] B. C. Csaji, "Approximation with artificial neural networks," 2001.

[6] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," in 2018 ACM/IEEE 45th Annual Internal Symposium on Computer Architecture (ISCA), 2019.

[7] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," CoRR, vol. abs/1704.04760, 2017.

[8] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," IEEE Internet of Things Journal, vol. 3.5, 2016.

[9] Z. Wang, J. Chen, and S. C. Hoi, "Deep learning for image super-resolution: A survey," IEEE transactions on pattern analysis and machine intelligence, 2020.

[10] X. Zhang, S. Das, O. Neopane, and K. Kreutz-Delgado, "A design methodology for efficient implementation of deconvolutional neural networks on an fpga," arXiv preprint arXiv:1705.02583, 2017.

[11] A. Yazdanbakhsh, M. Brzozowski, B. Khaleghi, S. Ghodrati, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "Flexigan: An end-to-end solution for fpga acceleration of generative adversarial networks," in 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 65–72, IEEE, 2018.

[12] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "Ganax: A unified mimd-simd acceleration for generative adversarial networks," in Proceedings of the 45th Annual International Symposium on Computer Architecture, pp. 650–661, IEEE Press, 2018.

[13] S. Liu, C. Zeng, H. Fan, H.-C. Ng, J. Meng, Z. Que, X. Niu, and W. Luk, "Memory-efficient architecture for accelerating generative networks on fpga," in 2018 International Conference on Field-Programmable Technology (FPT), pp. 30–37, IEEE, 2018.

[14] J.-W. Chang, K.-W. Kang, and S.-J. Kang, "An energy-efficient fpga-based deconvolutional neural networks accelerator for single image super-resolution," IEEE Transactions on Circuits and Systems for Video Technology, 2018.

[15] K. Tu, "Accelerating deconvolution on unmodified cnn accelerators for generative adversarial networks–a software approach," arXiv preprint arXiv:1907.01773, 2019.

[16] J.-W. Chang, S. Ahn, K.-W. Kang, and S.-J. Kang, "Towards design methodology of efficient fast algorithms for accelerating generative adversarial networks on fpgas," in 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 283–288, IEEE, 2020.

[17] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang, "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network," in Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 1874–1883, 2016.

[18] F. A. Azevedo, L. R. Carvalho, L. T. Grinberg, J. M. Farfel, R. E. Ferretti, R. E. Leite, W. J. Filho, R. Lent, and S. Herculano-Houzel, "Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain," Feb 2009.

[19] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in Advances in neural information processing systems, pp. 2672–2680, 2014.

[20] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," arXiv preprint arXiv:1603.07285, 2016.

[21] C. Dong, C. C. Loy, and X. Tang, “Accelerating the super-resolution convolutional neural network,” 2016.

[22] T. Tong, G. Li, X. Liu, and Q. Gao, “Image super-resolution using dense skip connections,” 2017.

[23] M. Haris, G. Shakhnarovich, and N. Ukita, “Deep back-projection networks for super-resolution,” 2018.

[24] S. D. Das, N. A. Shah, S. Dutta, and H. Kumar, “Dsrn: an efficient deep network for image relighting,” 2021.

[25] Z. Li, J. Yang, Z. Liu, X. Yang, G. Jeon, and W. Wu, “Feedback network for image super-resolution,” 2019.

[26] T. Karras, T. Aila, S. Laine, and J. Lehtinen, “Progressive growing of gans for improved quality, stability, and variation,” arXiv preprint arXiv:1710.10196, 2017.

[27] D. Xu, Y. Wang, K. Tu, C. Liu, B. He, and L. Zhang, “Accelerating generative neural networks on unmodified deep learning processors - a software approach,” arXiv preprint arXiv:1907.01773v3, 2020.

[28] D. Wang, J. Shen, M. Wen, and C. Zhang, “Towards a uniform architecture for the efficient implementation of 2d and 3d deconvolutional neural networks on fpgas,” in 2019 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–5, IEEE, 2019.

[29] Y. E. Wang, G.-Y. Wei, and D. Brooks, “Benchmarking tpu, gpu, and cpu platforms for deep learning,” 2019.

[30] B. Chin, “Lecture notes in parallel computation,” December 2020.

[31] G. J. Betts, Anatomy and Physiology. OpenStax, 2013.

[32] H. Kung and C. E. Leiserson, “Systolic arrays for vlsi,”

[33] D. A. Patterson and J. L. Hennessy, Computer Organization and Design, Fifth Edition: The Hardware/Software Interface. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2013.

[34] H. Kung, “Why systolic architectures?,” 1982.

[35] S. M. Afroze, “Lecture notes in advanced logic synthesis.”

[36] Y.-H. 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), 2016.