Imperial College of Science, Technology and Medicine
Department of Electrical and Electronic Engineering

Hardware Aware Convolutional Neural Network (CNN) Training Acceleration

Diederik Adriaan Vink

Supervised by
Professor Christos-Savvas Bouganis

Submitted in part fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering of Imperial College London
and the Diploma of Imperial College, August 2023

Abstract

Convolutional Neural Networks (CNNs) have emerged as a powerful deep learning tool, revolutionizing various domains such as computer vision. From applications like self-flying drones to self-driving cars, CNNs have demonstrated their effectiveness in enabling autonomous systems. As the demand for higher accuracy increases, CNN models are growing in complexity and time required to train. CNN training requires increasingly sophisticated hardware to handle the computational and memory requirements associated with training and inference tasks. However, CNNs are becoming increasingly intricate and have increasingly specific layers [3, 4, 5, 2, 1, 6, 7]. Additionally, the increasingly varying workloads and number representations is growing accordingly [8, 9, 10, 11].

One approach to address the issue of long training times is through the use of low-precision data representations and computations. In this thesis, a novel training strategy called MuPPET (Multi-Precision Policy Enforced Training) is proposed to that encompasses multiple precisions, including low-precision fixed-point representations.

Beyond long training times, CNNs provide a large variety of workloads throughout training. Currently accelerators are struggling to find the best hardware architecture to allow for efficient utilization of the available resources throughout training [12, 13]. Field-programmable gate arrays (FPGAs) provide a high degree of flexibility, unlocking the potential to adapt designs to the incoming workload. Caffe Barista integrate FPGAs into CNN training frameworks, providing a custom convolution kernel to accelerate training. Following Caffe Barista, FPGPT is a complete toolflow consisting of a state-of-the-art high performance FPGA convolution unit. This unit supports runtime workload adaptation providing the option to execute a variety of workloads on the same compiled design.

Finally, the thesis culminates in the creation of an acceleration policy building on MuPPET. This work utilizes the synergism between these two works to address the issues addressed by MuPPET and FPGPT better than either work could do individually.

Acknowledgements

Starting with my supervisor Christos-Savvas Bouganis, I would like to thank you for guiding me throughout my PhD and providing me with a vast amount of ideas and feedback for research. Additionally, you taught me how to clearly and effectively communicate my thoughts and work to others both verbally and in writing. Over the years of my PhD you have helped me develop as an engineer, a researcher and a communicator.

I would also like to thank Aditya Rajagopal; my flatmate, fellow PhD student and above all friend. Over the 9 years of knowing and working alongside you, you have been always been an guiding example for me to follow. You help brainstorm various ideas, giving honest and critical advice, especially at times when I was most stuck. My PhD would not have been the same without you in the lab.

I would also like to thank the rest of my friends that have been a wonderful distraction from work when I needed it, making my the time during my PhD unforgettable and for all the encouragement in the final stretches of my PhD.

To my partner, Katarina Boskovic, I want to thank you for being a constant source of love and understanding throughout the years of my PhD. You showed endless patience during my PhD, happily listening to me ramble on about my ideas, work and frustrations over the years, always there to support me.

Finally I would like to thank my family. I would like to thank my parents, Joyce and Patrick, who encouraged me to always pursue my goals and passions with everything I had and to never give up on my dreams, no matter what. I would also like to thank my brother, Jasper, for setting an incredible example for me to follow throughout my life. Your example made me strive to be the best version of myself and produce the best work I could. As a family, you have all shown me endless love, support and encouragement which truly helped me to complete my PhD.

Statement of Originality

I hereby certify that the contents of my PhD thesis are a product of the work that I have undertaken throughout the course of my PhD, of which 2 works were in collaboration with Aditya Rajagopal. All works and sources that have been used to help create this thesis have been cited and acknowledged.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the recent past, machine learning has taken the world by storm. As the popularity of the machine learning models grows so do the ever expanding applications. This increase in popularity and applicability has increased the attention this field has received from both academia and industry. Both academia and industry are looking to improve machine learning as well as apply it to new fields and problems. As they develop, the applications become more demanding and widespread. This results in the complexity of the machine learning techniques to continue to escalate. To keep up with the rising complexity, increasingly powerful hardware is being developed. Massive compute servers hosting hundreds of GPUs are exclusively dedicated to training these ever growing models.

CNNs are one of the most common deep learning methods that do not require the obscene compute power of other modern machine learning techniques like Large Language Models [7, 8]. This makes them accessible to train and implement in a variety of scenarios on a range of hardware. Nonetheless, the complexity of training modern CNNs is still demanding and often takes days, weeks and sometimes even months on commonly available compute setups. For a single training run for GoogLeNet on a standard setup of a CPU with a mid-range GPU takes 155 hours, which is 6.5 days. This assumes that the first training run returns the desired results the first time, and no further training runs are required. Such long training times means developing new architectures or iterating over novel techniques is an expensive and time consuming process.

Outside of the cost of training, the available hardware itself is another limiting factor for CNN researchers. There are a multitude of companies providing custom accelerators. These companies are each vying for the largest possible market share, hoping to convince potential clients their static architecture provides the biggest advantages. Some focusing on large batch sizes and others on small batches. Some companies target the types of network that contain multiple small convolutions while others aim to optimize for networks sporting the largest convolutions setting up for massive parallelism [9, 10, 11, 12]. What they all have in common is that they have to pick an approach and stick to it. In turn, this forces most researchers to focus their work based on a restriction placed by the hardware they purchase. Only large companies are uniquely able to afford a variety of different hardware accelerators to work with greater flexibility. This power further increasing their advantage over most academic or smaller scale researchers.

It is possible to address the issue of static hardware by utilizing reconfigurable hardware, often as FPGAs. Although FPGAs provide increased flexibility and adaptability, designing an optimized architecture is a time consuming process. Furthermore, they have limited clock speeds causing them to struggle to compete in terms of wall-clock execution time. On the other hand, they do provide incomparable efficiency saving overall power consumption, lowering training costs. Unfortunately, none of the popular training frameworks have integrated FPGAs into their ecosystem. Each of these factors adds an extra hurdle, dissuading most researchers from profiting off of the benefits provided by FPGAs.

1.1 Motivation

As discussed previously, the rising complexity of training CNNs has started to create an ever increasing barrier to entry. The increased hardware, time and monetary requirement is limiting the options researchers have. Currently training times for a modern CNN such as GoogLeNet when employing a 2080TI NVIDIA GPU is 152 hours. That being said, the machine learning world outside of CNNs is up to 835,584 GPU hours for GPT-3 [13] with the expectation for it to reach 63,000,000 GPU hours on the highest end GPUs available to train for GPT-4. Over time this is creating a gap between large corporate research and all other researchers. Works that investigate making training CNNs more accessible hope to abridge this increasing gap.

The basis of this thesis is to attempt to make initial strides to abridge this gap, but not limiting the scope to just small scale research. Additionally, to narrow the focus of the work, convolutions were made the focal point of the work. Convolutions make up 60-90% of all computation in CNNs. Being such a dominant compute factor made convolutions a natural initial target for improving CNN training. Following on, the work presented in this thesis follow two trajectories, focusing on either the algorithmic or hardware perspectives.

The first track to be explored is the acceleration of training, with a focus on quantization. This field has a variety of approaches, including mixed-precision training [14], reduced precision training [1, 15, 16, 17, 18, 19, 20, 21, 22, 23], distributed training [24, 25, 26, 27] and many more. Of all these fields, distributed training has attracted the most attention both in academia and industry. Distributed training is only applicable when the user has a large number of accelerators to train on. As mentioned previously, having access to such a large number of hardware accelerators is not common place making this work beneficial only to a subset of ML practitioners. As it stands right now both distributed and federated learning rely on distributed accelerator infrastructures. Many of the these works rely on quantizing data to reduce communications overheard [18, 28, 29, 30, 31, 27]. Additionally, most available hardware accelerators are producing specific reduced precision cores [11, 9]. Due to the increasing potential of quantized computation this specific hardware is now commonly available. Considering all of these factors, the work here focuses on how quantized training can be utilized to accelerate CNN training. The works presented in this thesis provide benefits to both single and distributed accelerator infrastructures as both can employ quantization and the works are not dependent on either infrastructure setup.

Training a CNN using reduced precision requires and understanding of how the error incurred by reducing the precision affects the training process. Currently the impact of quantization error of a convolution assuming a linear system is well established. Yet CNNs are inherently non-linear, limiting the effectiveness the established understanding of the impact of quantization. Lacking quantization error analysis techniques for CNNs requires the development of novel techniques to mitigate this potentially detrimental source of error. Exploring this field provides the promise of implementable acceleration for any ML practitioner.

Although the large hardware accelerator companies are adding fixed point computation units, these units are still limited to 8bit computation units. There are no lower precision options or native hardware for custom data-types available in production, although the research community is constantly exploring new precision types for efficient yet accurate computing such as in [32]. This leads to the second track that is explored throughout this thesis, finer grained quantized precision. As will be demonstrated in this thesis, reduced precision training can benefit from every potential precision. FPGAs are uniquely able to adapt to any precision the user aims to use as long as the architecture is designed to support it. They are even able to support custom data-types as long as they are defined by the hardware designer. Such adaptability allows for the exploration of more creative quantized training without needing to wait for large manufactures to catch up.

Employing FPGAs to operate at unique precisions that GPUs cannot provide, opens up an additional, hardware based opportunity. The convolutions in CNNs, especially in the forward pass, have a predictable data access pattern based on the dimensions of the layers. Inference acceleration architectures use this to their advantage, limiting their genericism to hyper-optimize for 1x1, 3x3, 5x5 and 7x7 convolutions. When examining the backward pass the details of the convolution vary drastically, rendering the aforementioned specific optimizations ineffectual. This leads to using more general techniques such as general matrix multiplication (GEMM), winograd or fast fourier transform (FFT) techniques. To execute a convolution as a GEMM, the data is reshaped from a 4-dimensional tensor to a 2-dimensional matrix. This is an expensive process and often negates any acceleration benefits acquired from hardware acceleration if left unaddressed. This is demonstrated early on in the thesis and is followed up by further exploration into mitigating this hurdle.

Finally, providing a small workload to a highly parallelized hardware accelerator will result in inefficient utilization of the available hardware. This leads to excessive power consumption with suboptimal runtime performance. A dynamic, reconfigurable architecture on the other hand provides the opportunity to tailor the architecture to the workload in question. If designed with sufficient foresight, this can even become a runtime parameter allowing for rapid adaptation with limited performance penalty.

Taking both an algorithmic and hardware approach to accelerating CNN training results in a unique opportunity for synergy. Quantized training acceleration techniques can benefit greatly from access to specific, tailored hardware. Additional information about the data can be extracted at minimal to no performance cost and the hardware can adapt to the needs of the quantized training scheme. As the demands on the training schemes advance and the adaptability of FPGAs progresses, the benefit of their synergism only grows.

1.2 Objectives and Minimum Requirements

This section aims to clarify the overall objectives of each branch of exploration presented in this thesis. Additionally, various minimum performance requirements are put in place to ensure the research is useful and relevant. These minimum requirements in some cases surpass what is provided by the current state-of-the-art. There are three aspects that are provided with objectives and minimum requirements.

1.2.1 Quantized Training

For the quantized training, the overall goal is to accelerate the training of a CNN model that operates in 32bit floating point (FP32) when fully trained, by utilizing quantization throughout training. The performance will be based on wall clock time and all baselines must be collected on the same hardware configuration. Furthermore, the completely trained model must be in FP32 and have an accuracy that is competitive with standard FP32 techniques. Finally, the training methodology should focus on quantizing convolutions for its performance gains. The acceleration does not have to outperform orthogonal acceleration works like distributed learning. The reason for this is that quantized convolutions can be applied alongside distributed training, compounding their benefits.

For the FPGA convolution works, the overarching goal is to produce a high performance, energy efficient convolution engine. Memory size and transfer rates are common bottlenecks limiting the performance potential of FPGAs. Additionally, this work should automate the configuration and generation of the FPGA bitstreams which enables both experienced and inexperienced FPGA and ML researchers to adopt training on FPGAs. This process should be flexible, adapting to a variety of CNNs, datasets and FPGAs.

1.2.2 Hardware Architecture

For the hardware architecture there are four definitive minimum requirements in place. They relate to the supported CNN networks, operational numerical precision and design throughput.

First, the architecture must be able to support all convolutions that are incurred for a specific CNN architecture throughout training. Additionally, the architecture must be able to be configured to work for a variety of different CNN architectures. This entails handling both small (down to 1x1) and very large (as large as the largest activation) convolutions kernels. Such a requirement demands a combination of flexibility and efficient memory utilization on the part of the architecture.

Secondly, the architecture must be able to accommodate a variety of modern networks that run on ImageNet. The majority of current FPGA-based CNN training works only operate on CIFAR-10 which is not considered relevant enough for this work. Other works support ImageNet but only for a specific or outdated networks.

Thirdly, the design must be able to operate in floating point. If the design only supports lower fixed point precisions, there is no guarantee the same design will work at floating point precision. On the other hand, if the design can support 32bit floating point, producing a basic implementation of the same architecture at a lower precision of either floating or fixed point merely requires a type change. There are of courses further optimizations that can be exploited at lower precisions. This flexibility to adapt any precision allows for easy integration with any research objective seeking to use FPGAs for training, regardless of desires to quantize or not.

Lastly, the design must consistently operate with a throughput of one. This means that at each cycle, each component of the FPGA is ready to access a new set of data. Such a requirement ensures that data is constantly flowing through the FPGA, ensuring the design is always ready to accept new data from memory. Designing the hardware this way allows for burst reading, a feature of FPGAs that enables significantly more efficient data access. This throughput must be maintained regardless of the convolution dimensions.

Regarding the objectives, the design must be able to compete with the state-of-the-art designs, while adhering to all of the minimum requirements. Ideally, this is in regards the average and convolutions GOPS. That being said, if those metrics are not available, then any like for like comparison is accepted that indicates similar or better runtime performance. When compared to a GPU solution, the main focus is to outperform the GPU in the areas and FPGA excels at. This entails at least outperforming the GPU on $\frac{GOPS}{W}$ and ideally on $\frac{GOPS}{DSP}$ as well. Regarding outperforming the GPU purely on GOPS (or any accepted metric variation thereof) is considered a long shot objective, but an objective nonetheless.

1.2.3 Ecosystem

Beyond the training and architecture requirements, the ecosystem itself also has specific requirements. The ecosystem decides which configuration the hardware architecture is best suited for the target network, configures compilations and integrates the FPGA execution and reconfiguration into CNN training frameworks. These requirements do not focus on performance, but rather focus on providing ease of use and adoption. The ecosystem should allow anyone interested in using FPGAs for CNN training to effortlessly get started without needing to make drastic changes to their existing standard code. The ecosystem must be able to accommodate a variety of networks and datasets as well. Additionally, it must be able to adapt the design to the incoming workload, scaling the design to operate at peak performance. Finally, the ecosystem must ensure the FPGA bitstream can be natively executed from within a commonly used machine learning frameworks such as Caffe, PyTorch or TensorFlow.

1.3 Metrics

Throughout this thesis, there will be different metrics analyzed depending on the work in question. As the thesis follows two key tracks, the metrics are split up accordingly. There are also common metrics that apply to all works.

The main metric used for the quantized training works is the Top1 and Top5 accuracy. This accuracy is reported for ImageNet and CIFAR-100 networks. Each CNN dataset has a set of classes. For example, ImageNet has 1000 potential classes an input can be classified as, CIFAR-100 has 100 such classes. As CNNs are classification systems, when used for inferences, for each input they assign a probability that it is of a certain class. The Top1 accuracy refers to how often the class assigned the highest probability for an input is the correct class. The Top5 accuracy refers to how often the classes assigned with the 5 highest probabilities for an input contain the correct class.

For the hardware accelerator works, the metrics are not related to training accuracy. They are expected to be perfect replacements of current compute accelerators (GPU or CPU) and produce the same output for a specific input. Instead they focus on the compute performance. The main metrics of interest here are tera operations per second (TOPS), giga operations per second (GOPS), Peak GOPS, average GOPS, convolution GOPS, GOPS per DSP ($\frac{GOPS}{DSP}$) and GOPS per watt ($\frac{GOPS}{W}$), . GOPS and TOPS solely look at how many operations the hardware processes per second. This is a common metric used to measure hardware performance. It is important to note this metric is only useful when analyzing the exact same computation. Comparing the same overall implementation but with different execution strategies could change the overall number of operations, making the GOPS metric a far less meaningful comparison.

Peak GOPS, average GOPS and convolution GOPS are specific ways of viewing GOPS. The

Peak GOPS in this context looks at the maximum GOPS achieved at any point in time throughout training. This metric demonstrates the upper limits of the hardware accelerator, but does not account for fact that workloads vary throughout training. Average GOPS analyzes the performance of the heterogeneous system (CPU and hardware accelerator) considering every operation and overall runtime. This metric is far more indicative of how the accelerator will perform in a real world scenario when the workload is not specifically tailored to maximize performance. Convolution GOPS analyzes the average performance of the hardware accelerator across only the convolutions encountered during training. This allows for a direct comparison of hardware accelerators by removing any heterogeneity, but still considers unfavorable workloads by considering all convolutions encountered during training.

$\frac{GOPS}{DSP}$ and $\frac{GOPS}{W}$ are metrics scaling the GOPS. The GOPS used in these metrics are the convolution GOPS metric. $\frac{GOPS}{DSP}$ scales the GOPS to the number of DSPs on the hardware accelerator. This shows how efficiently the hardware architecture utilizes the available resources. $\frac{GOPS}{W}$ is a metric of power efficiency. This metric takes into account how much power is required to perform the same number of computations. These metrics demonstrate two types of efficiency, useful for computing the cost of employing each technique.

Finally wall-clock time is a metric shared across all works. Throughout the thesis this is sometimes also measured as hours per epoch. This metric simply defines how long it would take to train a certain metric. Although it is not as insightful as the performance metrics presented so far, it can easily be applied across all works.

1.4 Publications

Based on the goals listed above and with the aim to bring improved training through quantized training, training on FPGAs and a merger of the two, the following publications were produced.

1.4.1 MuPPET

MuPPET is borne from the focus on algorithmic acceleration of CNN training, addressing the first objective. MuPPET utilizes a range of slowly increasing fixed point precisions to accelerate early stages of training. Using the Gradient Diversity metric the progress of training is tracked [33]. When the gradient diversity metric indicates the networks are no longer efficiently learning, MuPPET increases the precision of training. This is repeated until FP32 is reached and a final model is produced. This has shown to produce up to 1.84x wall clock training time acceleration. MuPPET was published in ICML 2020. For MuPPET I was a co-first-author. The algorithmic design and testing work as split evenly between the two primary authors. Aditya Rajagopal focused on the implementation of the MuPPET algorithm whereas I focused on the implementation of the quantization execution and integration into PyTorch.

1.4.2 Caffe Barista

Caffe Barista is an initial attempt at creating a hardware convolution unit, addressing objectives two and three, Caffe Barista, as indicated by the name, is integrated into the Caffe training framework. Barista consists of a systolic array, with the toolflow taking care of transforming the 4D input tensors to 2D matrices for a general matrix multiplication convolution. To be able to adapt to various workloads the tool breaks the convolution into tiles. Sections of the data in the form of tiles are sent to the targeted hardware. The final result is created by recombining the produced partial results after computation. This work suffered in terms of performance, but indicated the bottlenecks and opportunities required for improvement and competitive performance.. Caffe Barista was published at FPL 2020.

1.4.3 FPGPT

FPGPT is the realization of what Barista aimed to be. The foundation of FPGPT is based on a systolic array performing a general matrix multiply just like Barista. The architecture for

FPGPT differentiates itself by including a variety of additional features. FPGPT onboards the transformation process for all forward and backward convolutions. FPGPT's architecture is able to adapt to the incoming workload both at compilation time and runtime. The adaptability and onboarding alleviate a substantial amount of work from the CPU, accelerating the overall convolution time. Furthermore, FPGPT is able to maintain a throughput of 1 regardless of the precision it operates in or the workload provided. Additionally, FPGPT's runtime adaptability means it can execute any convolution efficiently, as long as the convolution's resource requirements don't exceed the compiled design limits.

Regarding the toolflow, FPGPT provides a specific design space exploration (DSE) flow. Due to the runtime adaptability, FPGPT is able to share compiled kernels across multiple different convolution sizes. The performance impact due to sharing a compile kernel is minimal, with some sharing possible without any performance sacrifice. Furthermore, the DSE contains a verified, highly accurate performance model. This model is used to guide the DSE to determine the optimal configuration. Finally, FPGPT is natively integrated into PyTorch (who acquired caffe), allowing for seamless transitioning from standard to FPGA based training. FPGPT is in the process of being submitted to TRETS as a journal paper.

1.5 Contributions

In the previous section each of the works for this thesis is introduced. Here the contributions of the thesis are discussed and how the works have helped to improve training CNNs. Training is considered improved based on how the work performs based on the metrics presented in the Sec. 1.3.

1.5.1 Quantized Training

In the field of quantized training, MuPPET and `GM` both provided contributions. MuPPET is one of a few works that utilizes the benefits of quantized training while targeting full precision inference networks. MuPPET explores to what extent quantization could be introduced during training to allow for acceleration. Additionally, MuPPET investigates the capability to use relatively simple metrics to track training progress when perturbed by induced error, for example due to quantization. This work will allow anyone who has access to reduced precision hardware or can benefit from quantized values to train full precision networks without consequence. One field already utilizing quantization is distributed and federated learning. MuPPET would allow these field to exploit the benefits they gain from quantization while mitigating the loss in accuracy that may be experienced.

`GM` takes the ability to quantize without recourse introduced by MuPPET and aggrandizes its effect. `GM`, a merger of MuPPET and FPGPT, is the final work in this thesis and treats each convolution as an individualized unit, adapting the precision locally. `GM` lowers each individual precision based on metrics collected by a modified FPGPT design, relying on the FPGPT performance model to determine the performance impact. Utilizing the FPGPT hardware, `GM` further enhances the benefits introduced by MuPPET. Additionally, it investigates and demonstrates the benefit of performing localized analysis on each convolution layer rather than solely looking at the network as a whole.

1.5.2 Hardware Acceleration

The hardware architectures presented in this thesis explore potential designs to perform CNN training on modern networks and datasets along with their corresponding bottlenecks. The initial work in Caffe Barista explore an initial design, clearly indicating the bottlenecks and limitations in performing a convolution as a GEMM on an FPGA. FPGPT progresses this to demonstrate the benefits of alleviating these bottlenecks. FPGPT investigates the dimensions and requirements of each convolution during training. Using this exploration FPGPT explores how to best utilize the reconfigurability of an FPGA as well as runtime configurable flexibility to best suit the large variety of workloads encountered during training. The limitations of the FPGPT design demonstrate how this field can further be progressed to ideally end up with a design that is competitive with a GPU in terms of wall-clock time, not just energy efficiency.

1.5.3 Ecosystem

Regarding ecosystem contributions, FPGPT, MuPPET and `GM` have contributed to the PyTorch ecosystem. Caffe Barista, as the name implies, contributed to the Caffe framework. As Caffe is now part of PyTorch, most of the focus will be on the PyTorch ecosystem development.

MuPPET and `GM` have created a modular system to allow for custom quantized training policy providing control over the precision of each convolution unit. The largest ecosystem development is from FPGPT. FPGPT developed an automated system to analyze a network & dataset combination to produce a configured FPGA bitstream. Additionally, FPGPT provides an extension to PyTorch that allows for native integration of FPGA hardware into PyTorch. This is a novel development and extends to applications beyond FPGPT, as any FPGA architecture can make use of this feature. These contributions to the PyTorch ecosystem will allow any user to have access to FPGA-based CNN training, regardless of experience with FPGAs. This makes FPGAs a far more attractive and feasible method to train CNNs for many users whether in industry, academia or for private purposes.

Chapter 2

Background Theory

2.1 Introduction

In the introduction section of this thesis the key objectives of this work are outlined. To properly investigate these, three core principles must be understood. These principles comprise of convolutional neural network (CNN) training, reconfigurable hardware for machine learning (ML) and quantized ML. Each chapter contains a specific contribution that relies on at least 2 of these 3 concepts.

First, convolutional neural networks (CNNs) are explored as these are the main target of optimization, and appear in each chapter. This if followed by an introduction to the methods of performing the convolutions seen in CNNs. Moving away from convolutions, quantization basics, and common error metrics to understand the impact of quantization are discussed. CNN and quantization basics are brought together to introduce the concept of quantized training. Finally reconfigurable hardware is discussed to complete the required background theory to understand the work presented in this thesis.

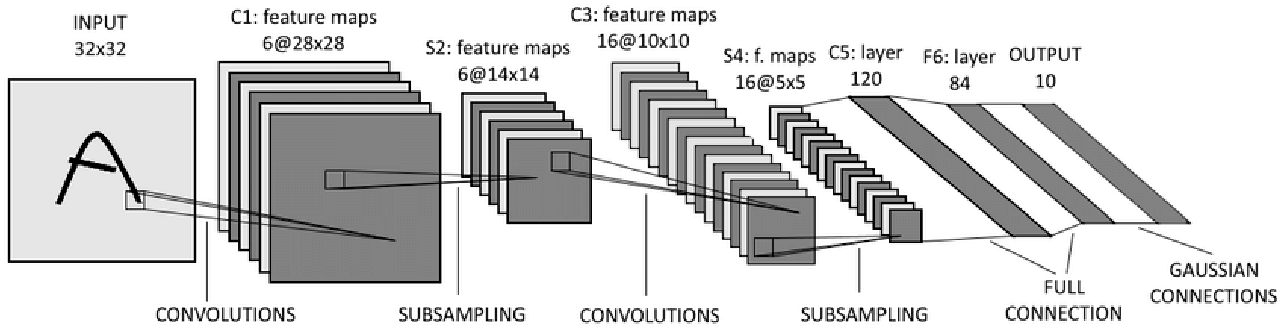


Figure 2.1: LeNet5 Architecture [2]

2.2 Convolutional Neural Networks (CNNs)

CNNs are a specialized supervised deep learning technique commonly used in image processing applications. They often excel at image detection and recognition. The focus on image based input data has led to a focus on the receptive field of each layer [34]. General deep neural networks consist of fully connected layers, connecting each output node of a layer to each input node of the next layer. Each of these connections is weighted based on how relevant that connection is to providing the desired output. CNNs instead rely on a receptive field represented by a weights kernel. This kernel is then shifted across the input, generating localized dense connectivity. Mathematically, a kernel sliding across an input performing per location computation is best represented by a convolution. Consequently, receptive field architectures rely primarily on multiple non-linear convolution layers. This is reflected by the fact that 50-90% of CNN workload is convolution layers [35, 36, 37]. Additionally, utilized convolutions allows for diminishing spatial dimensions while maintaining image information critical for classification.

The most basic CNN construction can be seen in Fig. 2.1. Each network layer consists of nodes that are connected (directly or indirectly) to the nodes in the next layer. Among all these connections, certain ones provide more important information towards feature extraction and classification. There connections are given a higher weight, referred to as a layer's weights.

Modern CNNs all have varying architectures, but nonetheless generally consist of a subset of commonly found layers. These layers consist of convolution layers, fully connected layers, ReLU layers, pooling layers, dropout layers and batchnorm. The details of each layer and how they interact in a network are discussed below.

2.2.1 Layers

The core to understanding CNNs is the various layers that are employed in networks. Each CNN architecture contains differences in how layers are connected and consequently how an input flows through the network. Nonetheless, the layers utilized and the order in which they are placed are generally similar across networks. Initial layers are responsible for feature extraction. These layers extract various features getting more specific deeper into the network. Later layers focus on classification from the initially extracted features [38].

Convolution

The key layer in CNNs is the convolution layer after which this class of network is named. These layers are responsible for extracting extracting features from images by convolving the input with a filter. In essence they create linear combinations of sets of values in the input. Convolution has always been used in image processing as a common filtering technique and proved highly effective in neural networks.

ReLU

Often directly after a convolution layer is the ReLU (Rectified Linear Unit) layer. This is a non-linear layer introduced to combat the vanishing-gradient problem. This issue refers to the fact that activations functions often reduce the input size between 0 and 1, squishing large changes in input to small space, resulting in a small gradient. As more layers get added, this gradient gets increasingly small slowing down or even preventing successful training [39]. This layer is non-linear clipping layer, clipping all negative values to 0.

Pooling

Next is the pooling layer. Pooling layers are used or dimensional reduction. The pooling layer is a sliding window that moves across its input grouping together data. Common pooling techniques are average and max pooling, with max pooling being the most popular. Max pooling will select the largest value of each position the sliding window visits passing only that value along. Pooling is used to remove redundant features encountered during the convolution.

Fully Connected

At the end of a network, there are usually at least 2 fully connected layers. These layers connect every node of one layer to the next. Early DNNs consisted exclusively of these kinds of layers. These layers are used solely to perform final classifications steps, ensuring the result is reduced to the number of outputs required to generate a probability for each potential output class. As there layers can be expressed as a convolution, they are occasionally replaced by specifically constructed convolution layers.

BatchNorm

The four layers discussed so far almost universally seen across different CNN architectures. Batchnorm is a more recent addition to CNNs. Batchnorm (batch normalization) layers are often placed after a ReLU. They are used to stabilize training by collecting distribution statistics of a layers' activations. If a layers weights are changed, the resultant sudden change in activations statistics lead to inaccuracy in succeeding layers. As a result the batchnorm layer mitigates this effect by minimizing the difference between batches [40].

Dropout

Dropout layers will drop a random set of connections at each iteration. These layers are usually placed further down the network closer to the fully connected layers. This is done so that key information is not dropped in early feature extraction phases which might hinder early learning. By occasionally dropping connections and temporarily changing the network structure, this layer helps reduce overfitting to the training dataset [41].

Overall Structure

The specific unique combinations of various CNN architectures are discussed in Section. 2.2.4. Nonetheless, there is a general construction setup that is followed by a majority of CNN architectures.

Usually in a CNN a convolution layer is followed by a ReLU (Rectified Linear Unit) layer. In standard networks a ReLU is followed by a pooling layer. If the network contains batchnorm layers, these will sit in between the convolution and ReLU layer. Similarly dropout layers tend to sit at the same location. The combination of a convolution layer with a batchnorm, dropout and/or ReLU can be called a convolutional block. More advanced blocks like residual, fire and squeeze modules are introduced in later works.

The beginning of the network is commonly filled with a series of convolution blocks with pooling layers between them for spatial and dimensional reduction. This is responsible for feature extraction and some degree of classification. The later layers, usually the last 2-3 are fully connected layers to finalize any classification process. For extremely deep networks like ResNet150, classification is handled by more convolutions layers as compared to ResNet18, as the fully connected layers are a smaller percentage of all the layers.

It is important to note that these structures are commonly place, but by no means a rule. There are discussions about the optimal placing of batchnorm and dropout layers as well as new methods of directly chaining convolution layers [6, 5].

2.2.2 Training

Training a CNN is the process of determining the correct weights for the CNN to produce the highest accuracy results. The most popular form of training is utilizing the back propagation algorithm utilizing stochastic gradient descent (SGD) as the optimizer. This allows for the training to be cast as an optimization problem through a high-dimensional hyper-space.

CNN training consist of 2 different types of passes: the forward pass and the backward pass. Initially weights are randomly set. Data is presented to the CNN, often as a batch of images, which pass through the network in a forward fashion. The network computes a classification and utilizes a loss function to determine the quality of the results. This loss value is then passed back to the last layer. This layer computes the partial gradient with respect to the weights and the forward input. The gradient with respect to the weights are used to update the network weights while the gradients with respect to the input are propagated back to the previous layer. This preceding layer then uses these incoming gradients to compute their own gradient with respect to input and weights. This then propagates all the way down back to the first layer. This step is referred to as the backward pass.

The computed loss value is used as the error gradient, steering the direction of training. This steers the model through the high-dimensional space. When performing the back-propagation, the activations (outputs) of each layer are required to calculate the required gradients.

2.2.3 Inference

Inference is the operational method of CNNs upon deployment. This is exclusively a forward pass on a fully trained network. A key distinction between the forward pass in training and inference, is each iteration of training will have a different set of weights of the forward pass. During inference, the forward pass will remain static no matter how many iterations the inference performs.

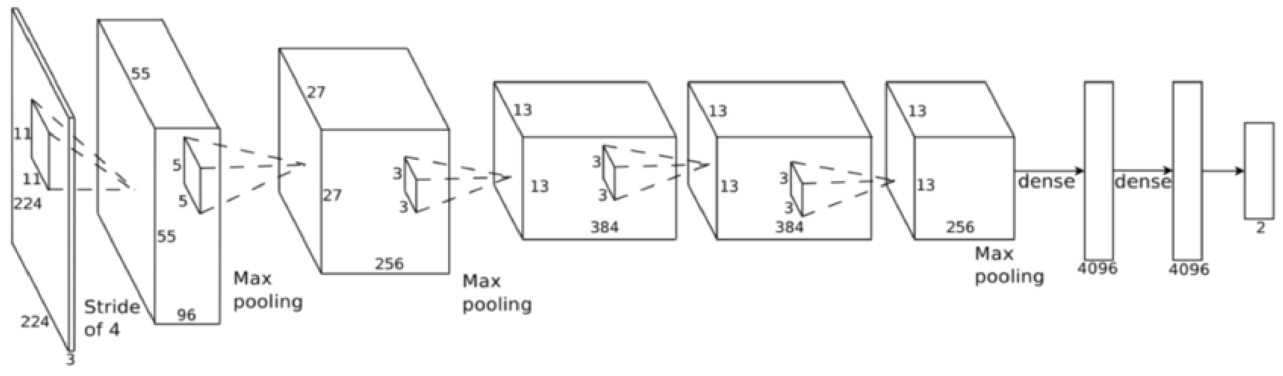


Figure 2.2: AlexNet Architecture [3]

2.2.4 Network Specific

As research in CNNs progresses, networks becoming increasingly complex. This demonstrates itself in two aspects. The first is adding new types of layers such as dropout layer and batchnorm layers. The second and most uniqifying aspect is different connection setups and unique convolution types.

AlexNet [3]

Of all the networks that will be discussed in this thesis, AlexNet is the most basic. AlexNet is the first network that was able to meaningfully classify the ImageNet dataset. As shown in Fig. 2.2, AlexNet contains 5 convolution layers, each followed by a ReLU layer. After the first, second and fifth convolution & ReLU layer a max pool is placed. This is finalized with fully connected layers to complete classification.

ResNet [4]

The ResNet architecture is a newer design that introduces residual learning. Unlike AlexNet which has a static structure, the residual learning block can be applied multiple times creating networks of various depths. ResNet networks are a combination of the residual blocks shown in Fig. 2.3 As shown by the curved arrow, a residual block adds the input to the residual block back after a set of convolutions. This is referred to as a bypass connection. The key function

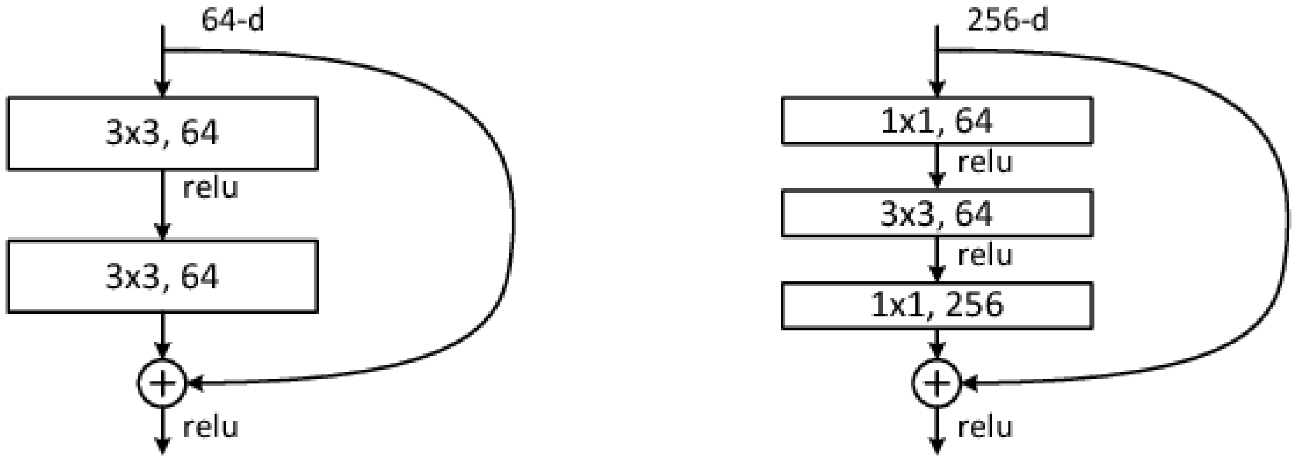


Figure 2.3: Residual Learning Block [4]

of the bypass connection is to allow a set of convolutions to adopt weights of zero, creating an identity function. A fully zeroed out residual block is able to function as a pass-through of the input.

This architecture came from the trend that blindly adding depth to networks initially provides improved accuracy but has diminishing returns eventually leading to accuracy degradation. The residual blocks ensure that if a convolution would not meaningfully contribute to classification, it can become identity and ensure its output is not worse than the previous residual block. This guarantee does not exist in more simple network architectures. For example, this enables the ResNet-152 network which supports 152 convolution layers to function properly and improve upon ResNet-101.

SqueezeNet [5]

Unlike ResNet which attempts to improve accuracy by adding more layers and mitigating the impact, SqueezeNet aims to reduce complexity while maintaining high accuracy. Running 152 convolution layers requires a large amount of memory and compute power. SqueezeNet on the other hand has $\frac{1}{50}^{\texttt{th}}$ of the number of parameters of AlexNet, let alone ResNet-152. SqueezeNet achieves such low complexity while maintaining through the application of the follow three 'strategies':

1. Replace 3x3 convolutions with 1x1 convolutions for a 9x drop in parameters.

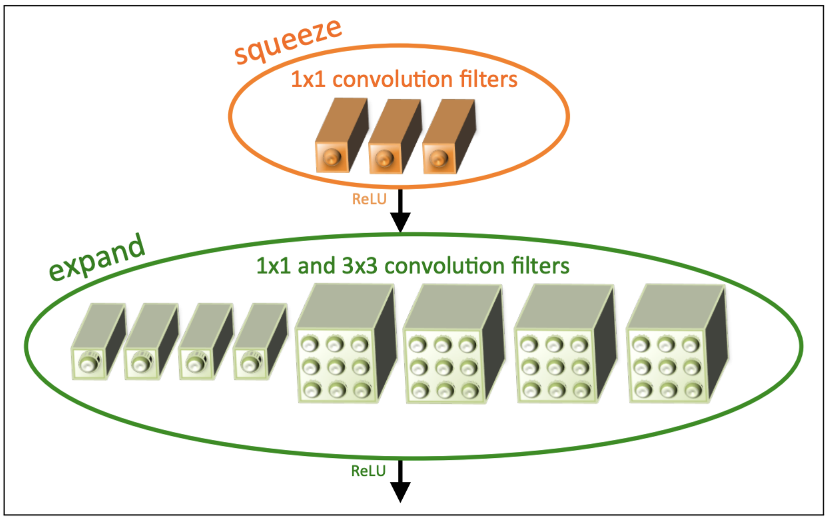


Figure 1: Microarchitectural view: Organization of convolution filters in the **Fire module**. In this example, $s_{1x1} = 3$, $e_{1x1} = 4$, and $e_{3x3} = 4$. We illustrate the convolution filters but not the activations.

Figure 2.4: Fire Module [5]

2. Decrease input channels to 3x3 convolutions as they scale faster than 1x1 convolutions.

3. Delay downsampling until later on in the network to maintain large activation maps.

These three strategies are realized with the implementation of fire modules. A fire module is displayed in Fig. 2.4. The principle behind a fire module is comprised of two steps, the squeeze and the expand section. In the squeeze section, strategy 1 is employed replacing standard 3x3 convolutions with 1x1 convolutions. The expand section has a combination of 1x1 and 3x3 convolutions. An expand section will always have a greater number of convolutions than the preceding squeeze section ensuring a low number of input channels to the expensive 3x3 convolutions. Employing less convolutions in the squeeze stage ensures strategy 2 is utilized. Finally strategy 3 is unrelated to the fire module and comes into play decided when to apply pooling to cause downsampling. The output of the expand layers are all concatenated together and passed on to to the succeeding fire module.

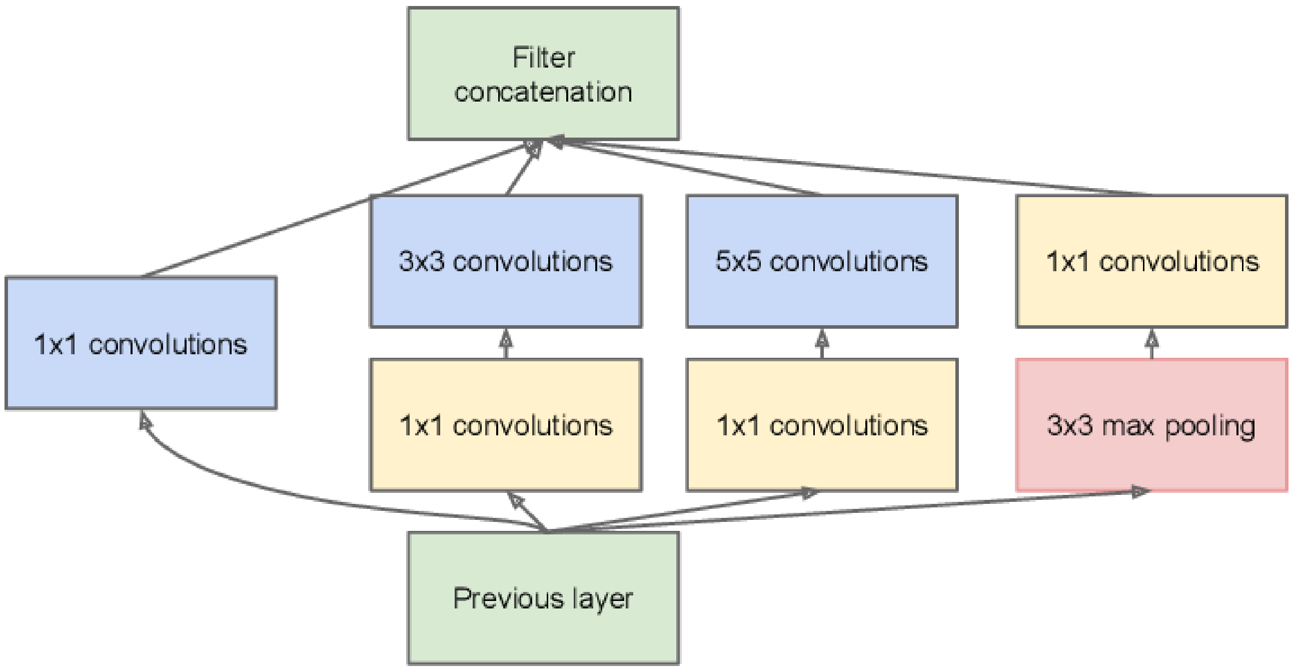


Figure 2.5: Inception Modules [6]

GoogLeNet [6]

The GoogLeNet architecture, also known as Inception V1, has similar properties to SqueezeNet, but with a slightly different focus. Comparably to SqueezeNet, GoogLeNet creates a unique module placed in sequence with occasional max pooling for downsampling. Fig. 2.5 portrays GoogLeNet's Inception module. This modules consists of a 1x1, 3x3 and 5x5 convolution (blue). The variety of convolution kernel (KERN) sizes allows different levels of granularity. Smaller convolution kernels allow for a more localized analysis with larger convolutions able to extract broader features. To ensure the complexity does not get out of hand, 1x1 convolutions (yellow) are used to reduce the number of input channels. This is similar to Strategy 2 from SqueezeNet. Clearly for more modern networks like SqueezeNet and GoogLeNet, there are a large amount and dependence upon 1x1 convolutions.

2.3 Convolution Layer Execution

In this section, initially the mathematical breakdown of convolution layers is presented. This is then followed with the various computational implementation methods of the convolution layer are discussed. The GEMM approach is discussed in greatest detail as it is the most relevant to all the work presented in this thesis.

All CNNs convolve 4D tensor structures (KERN and INP) producing a 4D tensor (OUT). First the basics of the convolutions encountered during training are discussed. This is succeeded by the most common convolution technique called Direct convolutions, also known as sliding window convolution. Following this up is a discussion of how KERN and INP structures are manipulated to perform a 4D convolution through a GEMM operation. The discussion highlights the data reordering requirements for performing a convolution through a GEMM operation, a process that is targeted for acceleration in the hardware chapters of this thesis. Finally this is finished by a discussion of Winograd and FFT convolutions.

2.3.1 Convolution Details

Convolution layers are the core of CNNs as well as all the work presented in this thesis. These layers vital to this work and are of a higher complexity compared to other layers. Therefore a more in-depth description of the what occurs in convolution layers during the forward and backward pass operation are described.

A convolution operation takes as inputs a convolution kernel (KERN) an input data structure (INP), and produces an output data structure (OUT).

During the forward pass, for each convolution layer (CONV), a convolution of a batch of input feature maps (IFMs) with the layer weights is performed producing a set of output feature maps (OFMs), as represented by Eq. 2.1.

$$Fwd := \underbrace{OFM}_{\text{OUT}} = \underbrace{(IFM)}_{\text{INP}} *dilate \underbrace{(weight)}_{KERN} \tag{2.1}$$

In accordance with the introduced terminology the IFM corresponds to INP, the layer weights to KERN and the OFM to OUT. Each CONV accepts as input an IFM tensor of dimensions $B_{size} \times Ch \times H_{IFM} \times W_{IFM}$, denoting the batch size, number of IFM channels, IFM height and IFM width respectively. Assuming the layer weights have D filters consisting of Ch channels per filter this produces a tensor with dimensions $D \times Ch \times H_{weights} \times W_{weights}$. Note that the number of channels for INP and KERN must be the same, and therefore: $Ch_{weight} = Ch_{IFM}$. In each convolution layer the IFM is convolved with the layer weights to produce an OFM of dimensions $B_{size} \times D \times H_{OFM} \times W_{OFM}$, which are batch size, the OFM channels (equal to D), OFM height and OFM width respectively. The OFM is then passed to the next layer in the network and the same process is repeated again. After passing through all networks layers, the forward pass concludes by calculating the loss which is then passed to the backward pass.

The backward pass uses the backpropagation algorithm for updating the network parameters. Each CONV is provided with the gradient w.r.t. OFM ($Grad_{OFM}$) $\frac{\delta loss}{\delta OFM}$ by the previous layer during the backpropagation algorithm. In a CNN, the OFM of a layer becomes the IFM of the following layer. As a result, in the backward pass, the $Grad_{IFM}$ of one layer becomes the $Grad_{OFM}$ of the following layer. For example, (from the forward pass perspective) layer 3's OFM is layer 4's IFM. In the backward pass this means layer 4's $Grad_{IFM}$ is passed backwards to become layer 3's $Grad_{OFM}$.

Once the $Grad_{OFM}$ is received, the gradient w.r.t. weights ($Grad_{weight}$) (Eq. 2.2) is computed through a convolution operation and is used to update the weights.

$$Grad_{weight} := \underbrace{\frac{\delta loss}{\delta weight}}_{\text{OUT}} = perm\left(\underbrace{perm\left(IFM\right)}_{\text{INP}} * perm\left(\underbrace{dilate\left(\frac{\delta loss}{\delta OFM}\right)}_{\text{KERN}}\right)\right) \tag{2.2}$$

Here, $perm$ refers to the permute operation of the matrix from $N \times C \times H \times W$ to $C \times N \times H \times W$. Furthermore, $dilate$ refers to dilating the $Grad_{OFM}$ by the layers dilation factor, which is the stride value from the forward pass, where $*$ denotes the convolution operation. The gradient w.r.t. IFM ($Grad_{IFM}$) (Eq. 2.3) is computed using a convolution operation and then passed back to the previous layer as that layer's $Grad_{OFM}$.

$$Grad_{IFM} := \underbrace{\frac{\delta loss}{\delta IFM}}_{\text{OUT}} = \underbrace{dilate\left(\frac{\delta loss}{\delta OFM}\right)}_{\text{INP}} * perm\left(rot180\underbrace{\left(weight\right)}_{\text{KERN}}\right) \tag{2.3}$$

Here $rot180$ refers to rotating the affected matrix by 180 degrees at its lowest two dimensions, H_{weight} and W_{weight} in this case. Using (Eq. 2.2) and (Eq. 2.3) the updating of the model parameters is performed according to a policy defined by the optimization algorithm employed.

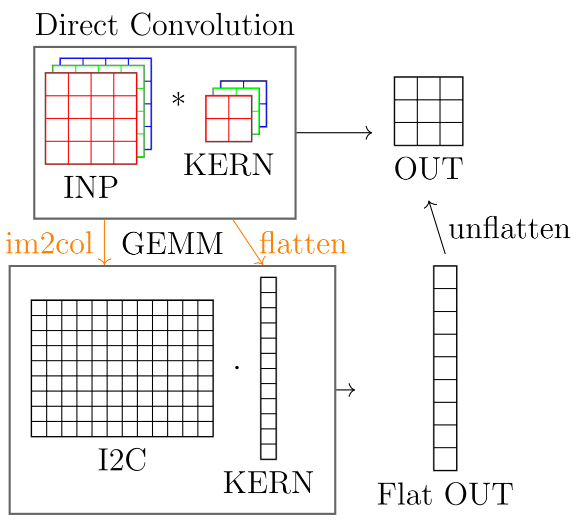
2.3.2 Direct a.k.a Sliding Window Convolution

Direct convolutions are also known as sliding window convolutions. The operation consists of sliding the convolution kernel across the input image. At each location the kernel performs an element-wise multiplication summing all of the results. The KERN window then 'slides' to the next position and repeat the multiplication and accumulation (MAC). The distance the window slides is based on the stride parameter.

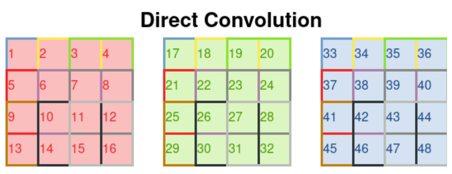
In a direct convolution, a 2D tile of KERN of size $H_{KERN} \times W_{KERN}$ and a 2D tile of INP of size $H_{INP} \times W_{INP}$ are convolved. Moving to 3D KERN and INP tensors, the third dimension are the channels depicted by the colored squares in Fig. 2.6b. The results of the per channel 2D convolutions are summed together to produce a 2D OUT matrix of $H_{OUT} \times W_{OUT}$. Convolving the 3D KERN tensor of $C_{KERN} \times H_{KERN} \times W_{KERN}$ with the 3D INP tensor $C_{INP} \times H_{INP} \times W_{INP}$ produces a single 2D OUT matrix, where $C_{KERN} = C_{INP}$.

In the case where a number of inputs are processed in parallel (Batch size > 1), 4D convolutions have to be performed. Let's assume N_{KERN} 3D KERN tensors (full 4D KERN tensor) and N_{INP} 3D INP tensors (full 4D INP tensor). Each of the N_{KERN} 3D KERN tensors is convolved with each of the N_{INP} 3D INP tensors, producing $N_{INP} \cdot N_{KERN}$ 2D OUT matrices. Ordering the $N_{INP} \cdot N_{KERN}$ 2D OUT matrices produces the 4D OUT tensor of dimension $N_{OUT} \times C_{OUT} \times H_{OUT} \times W_{OUT}$ as $N_{OUT} = N_{INP}$ and $C_{OUT} = N_{KERN}$. Please note each of the $N_{INP} \cdot N_{KERN}$ 2D OUT matrix computations is independent.

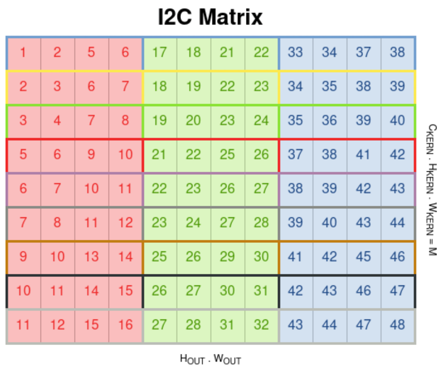
Although this version of convolving is very straight forward, it does not scale well. The maximum degree of parallelism is always directly correlated to the dimensions of the convolution kernel. For example when looking at inference, most convolutions are usually 1x1 and 3x3 convolutions. This would mean that unless there are enough 1x1 convolutions to perform simultaneously, a processor with low memory but exceptionally high amount of parallel DSPs will waste a lot of resources. The problem could of course be split up to run across all the DSPs, but this would require recombining the solution using even more resources. Similarly for the 3x3 convolutions, a processor with dimensions not a multiple of three will similarly waste a



(a) GEMM Convolution and Direct Convolution



(b) Direct Convolution



(c) im2col: The 3D input tensor is transformed into a single 2D I2C matrix

Figure 2.6: Im2col Visualization where $N_{KERN} = 1, C_{KERN} = 3, H_{KERN} = 2, W_{KERN} = 2$ and $N_{INP} = 1, C_{INP} = 3, H_{INP} = 4, W_{INP} = 4$

lot of compute being unable to neatly fit to the problem space. As a result, small edge FPGAs are often a very good application for these kinds of convolutions [42, 43, 44].

2.3.3 General Matrix Multiply (GEMM)

The combination of high performing libraries and hardware for GEMMs (i.e. GPUs) has led to wide adoption of computing convolutions through GEMM operations. In the case where the convolution operation is performed through a matrix multiplication, the INP and KERN structures are transformed into matrices such that their multiplication produces the same result as their convolution. This transformation is called the 'im2col' transformation. For brevity, all mentions of dilation, rotation and permutation are left out. It is assumed that the KERN and INP tensors discussed in this section have already been dilated, rotated and permuted.

Let's consider the same case as above, where a convolution needs to be performed between the 3D INP and 3D KERN tensors. As the process is independent of the number of 3D KERN, N_{KERN}, and 3D INP, N_{INP}, we will consider the case where $N_{KERN} = N_{INP} = 1$ for simplicity. Firstly, the 3D INP and KERN tensors are converted into 2D matrices, $Matrix_{INP}$ of dimensions $O \times M$ and $Matrix_{KERN}$ of dimensions $M \times P$ respectively.

$Matrix_{KERN}$ is formed by vectorizing the 3D KERN tensors and stack them on top of each other, forming a vector with dimensions $M \times P$, where $M = (C_{KERN} \cdot H_{KERN} \cdot W_{KERN})$ and $P = 1$. Fig. 2.6a shows the transformation of KERN from a $3 \times 2 \times 2$ tensor to a 12×1 matrix.

$Matrix_{INP}$ is formed by considering the corresponding tile position in the case of a direct convolution for one of the channels, and form part of the row of the of the $Matrix_{INP}$. Successive tile positions are stacked one under the other filling up the rows of the matrix. The process is repeated for the rest of the channels filing out the remaining part of the columns, resulting in the I2C matrix. This is visually represented by each colored square from Fig. 2.6b (representing a location for of a KERN tile) mapping to a row in I2C matrix in Fig. 2.6c.

It is worth noting that in the case where the convolution stride is less than W_{KERN}, elements of the 3D INP tensor will be replicated in I2C matrix. As such, in the case where the I2C is fully computed and stored before the convolution and $stride < W_{KERN}$, the memory footprint for the I2C is larger than the memory footprint of the original INP structure. For example in GoogLeNet the I2C:INP size ratio reaches up to 25:1.

Multiplying $Matrix_{INP}$ with $Matrix_{KERN}$ produces a matrix with dimensions $O \times P$, where P=1. Finally, the matrix is unflattened into a square matrix of size $H_{OUT} \times W_{OUT}$ as shown in Fig. 2.6a, creating the desired 2D OUT matrix. Similar to the direct convolution process, this is repeated for all $N_{KERN} \cdot N_{INP}$ 3D KERN and 3D INP tensor combinations to produce the 4D OUT matrix.

The process of converting the INP to I2C matrix is referred as "im2col"procedure in the literature. In the context of the forward pass, the INP structure corresponds to the IFM structure of a layer, KERN to the layer weights and OUT to the OFM of that layer. In the context of the backward pass, INP corresponds to $Grad_{OFM}$ and the layer weights for the $Grad_{weight}$ and $Grad_{input}$ convolutions respectively. The permute, dilation and rotation operations that are required are applied to the INP and KERN structures before the im2col procedure.

The above discussion shows how the KERN and INP tensors are transformed from their original state to one that can be utilized in a GEMM based convolution. As shown by the example in Fig. 2.6c, the amount of data required by an I2C matrix is often greater than that of the INP. Being able to perform this operation on the fly can provide the potential for a notable reduction in memory required for the convolution, as well as to benefit the latency of computing the above in case of the existence of parallel hardware.

The GEMM method has the highest time complexity of $O(N^3)$. That being said, this does not necessarily mean the worst performance, as there are extensive optimization libraries for highly parallelized execution.

2.3.4 Winograd

Winograd convolution is an alternative to convolving through a sliding window convolution or a GEMM. The core principle is based on winograd's minimal filtering algorithm. This entails moving the computation to the Winograd spatial domain by transforming both the input and the weights, performing an element-wise multiplication and transforming it back. Winograd convolution provide the advantage of reducing the number of operations required to perform the convolution and needs a very low runtime memory. Although winograd can technically be applied to any size of convolution, performing huge transforms leads to unacceptable levels on inaccuracy according to [45]. Therefore it is often broken down into multiple smaller sub-convolutions which need to have their partial results combined at a cost. As there is an optimal size for the winograd convolution, both the convolution and the processor must be of specific sizes to elicit the low number of FLOPs and low memory profile. For large convolutions like those seen in the backward pass, this limitation on size becomes a problem. Therefor the tradeoff for winograd is that it has the least flexibility out of all discussed convolution approaches. Winograd is primarily useful for smaller, well fitting convolution kernels that fit the specific filter sizes that elicit the benefits provided by the winograd algorithm [45].

2.3.5 Fast Fourier Transform (FFT)

Lastly, FFT convolutions transform the input to the frequency domain through a Fourier Transform. Unlike Winograd convolutions, FFTs operate most efficiently on large convolution kernels. Larger kernels allow for greater amortization of the conversion cost moving transforming to the frequency domain [45]. Although FFTs have a higher time complexity of $O(N \log N)$ vs $O(N)$ for winograd and usually a higher memory requirement, they allow for greater flexibility [7] by not being forced to smaller problem sizes like winograd. Furthermore, larger convolutions are far more amenable to highly parallel accelerators such as GPUs, TPUs and server grade FPGAs. This also makes FFTs more applicable to the larger convolutions seen in the backward pass. In an ideal system, all three types of convolutions are applicable and can be employed depending on the convolution workload and processor specifications at hand.

2.4 Reconfigurable Hardware

Two of the four content chapters present custom reconfigurable architectures for performing the convolutions present in CNN training as GEMMs. These focus on using field programmable gate arrays (FPGA) as the target hardware. FPGAs consist of a grid of configurable logic blocks (CLBs). Each CLB consists of a lookup table (LUT), mux and flip-flops. Each CLB is linked to all surrounding CLBs through an array of interconnects. Outside of the CLB, the FPGA has an array of DSPs for more complex operations and on-chip memory (BRAM and URAM). By programming the FPGA with a bitstream, all CLBs are setup and connected through the interconnects in a fashion representing a custom circuit. This reprogrammability allows the FPGA to adapt its custom architecture depending on the scenario it is placed in. The customizability along with the interconnected grid structures makes FPGAs uniquely able to provide a high degree of parallelism and energy efficiency.

Furthermore, both of the FPGA works in this thesis target the FPGA through an abstract language called High Level Synthesis (HLS). This is a library that allows a custom compiler to interpret a variation of C++ to be compiled down to an FPGA executable bitstream. An alternative to using HLS is by designing the architecture in a hardware description language (HDL), a lower level register level language. Most works are written in HDLs as these have better built tools and support, whereas HLS is new and not fully able to deliver upon its claims.

2.5 State of the Art: Quantized Training

This section will cover the state of the art (SotA) in the field of quantized training. The various leading techniques for both low- and high-precision final networks are discussed with a discussion of the key approach taken.

All training discussed so far assumes 32-bit floating point (FP32) numerical precision for all operations encountered. Over time CNNs gained popularity, especially in edge-device applications such as self-driving cars and drones. Furthermore networks became more computation and memory intensive through ever increasing depth. Edge devices are already notoriously low on memory and compute power, and with increasing network complexity CNNs started to become unusable on edge-devices. As a result, complexity reduction became a popular field of research. This led to four main directions: 1) network pruning, 2) sparse accelerators, 3) network architecture search and 4) quantization. In this work, the focus is placed exclusively on quantization, so only those methods will be discussed.

Regarding quantization aware training (QAT), there are two key distinctions in the works. The vast majority aim to create a network that operates at a reduced precision. The goal for these low-precision networks is to ensure inference can occur at a lower precision while maintaining a competitive test accuracy. The other potential goal is using low precision during training to accelerate training but still aiming to produce an FP32 network.

2.5.1 Low-precision Network

The production of low-precision network places a focus of accelerating inference through quantization while maintaining a test accuracy similar to an FP32 baseline. Most works in this space are not concerned with the impact on training time. These works often generate complex methods quantizing as well as complex methods of analyzing the effect of quantization. This makes them strong inference accelerators, but almost always negatively impact training time. This often leads to creating highly complex, non-uniform quantization schemes that take long to train but result in highly competitive results. Within this category of creating low-precision networks, there are two subclasses: 1) post-training quantization and 2) reduced-precision training.

Post-training Quantization (PTQ)

Post-training Quantization is a form of quantization training that takes a fully trained model and attempts to manipulate the existing network to operate at a target reduced precision [46, 47, 48, 49, 50, 51]. This often requires a few iterations of training to make any necessary adjustments based on a target goal or metric. Various approaches include analyzing each layers impact on the final accuracy [46], with some including the computation complexity of various quantization schemes upon those layers [47]. The authors of [49] posed expressing the task loss as a Taylor series expansion with quantization posed as a quadratic unconstrained binary optimization problem. Other works focus on optimizing the clipping of the quantization itself, using the Hessian of the data [48] or modeling clipping using Laplace or Gaussian models [51]. In [50], the authors mitigate outlier impact by duplicating the channel and halving the values before setting the clipping range.

Quantized Aware Training (QAT)

On the other end of spectrum from post training quantization is quantized aware training. QAT knows the target network precision before training starts and incorporates this into the training process. This usually slows down the training process but allows for highly competitive accuracy with lower bitwidths than PTQ when looking at the state of the art. The works divide into three main categories: 1) custom quantization schemes, 2) bias analysis and adaptation and 3) incorporating quantization into the loss function.

The custom quantization schemes range from analyzing the MSE induced by various clipping strategies [52] to creating quantization strategies that have 2 quantized ranges for high and low values [16, 17]. These custom quantization strategies still operate on the concept of fixed point computation. Another approach is modifying the standard floating-point system. This has lead to custom precision types like bfloat [53] and block minifloat [54]. Bfloat16 is a 16bit floating point representation that differs from the standard floating point representation by devoting sacrificing bits in the mantissa only [53]. This allows for less detail but ensures the entire range of values is covered, operating on a similar assumption to [16, 17]. Block minifloat takes the custom floating point design even farther by sharing the exponent across multiple values called a block. This allows the values within each block to benefit from greater precision within their range before getting scaled by the shared exponent [54]. Furthermore, works like [18] and [55] utilize custom quantization not to quantize the model, but to distribute the load of backpropagation, easing channel communication.

The second main approach is to address the bias introduced by quantization. Incorporating a bias factor after the convolution is a common occurrence in CNNs. Works such as [19, 56, 57] actively attempt to counteract the induced quantization bias by modifying the existing bias to correct for the mean-activation shift. Others only quantize a random subset of the data during training to mitigate the effect of the quantization bias [20].

Lastly, the authors of [58] incorporate the clipping factor into the loss function, to allow the quantization scheme to be a part of the optimization process.

2.5.2 High-precision Network

A less popular field of quantized training is with the focus on accelerating training itself. The state-of-the-art method in accelerating training with reduced precision is mixed-precision training [14]. The authors propose to employ low-precision FP16 computations in the training stage of high-precision CNNs that perform inference in FP32. Along the training phase, the algorithm maintains a high-precision FP32 copy of the network's weights, known as a *master copy.* At each minibatch, the inputs and weights are quantized to FP16 with all computations of the forward and backward pass performed in FP16, yielding memory footprint and runtime savings. Under this scheme, each stochastic gradient descent (SGD) update step entails accumulating FP16 gradients into the FP32 master copy of the weights, with this process performed iteratively throughout the training of the network. Micikevicius et al. [14] evaluate their scheme over a set of state-of-the-art models on ImageNet, and show that mixed-precision training with FP16 computations achieves comparable accuracy to standard FP32 training.

2.5.3 Gradient Diversity

Gradient diversity [33] is a metric of measuring the dissimilarity between sets of gradients that correspond to different minibatches. The gradient diversity of a set of gradients is defined as

$$\begin{aligned}\Delta_{\mathcal{S}}(\mathbf{w}) &= \frac{\sum_{i=1}^{n} ||\nabla f_i(\mathbf{w})||_2^2}{||\sum_{i=1}^{n} \nabla f_i(\mathbf{w})||_2^2} \\ &= \frac{\sum_{i=1}^{n} ||\nabla f_i(\mathbf{w})||_2^2}{\sum_{i=1}^{n} ||\nabla f_i(\mathbf{w})||_2^2 + \sum_{i \neq j} \langle \nabla f_i(\mathbf{w}), \nabla f_j(\mathbf{w}) \rangle}\end{aligned} \tag{2.4}$$

where $\nabla f_i(\mathbf{w})$ is the gradient of weights $\mathbf{w}$ for minibatch i.

The key point to note in Eq. (2.4) is that the denominator contains the inner product between two gradients from different minibatches. Thus, orthogonal gradients would result in high gradient diversity, while similar gradients would yield low gradient diversity. The proposed framework, MuPPET, enhances this concept by considering gradients between minibatches across epochs and proposes the developed metric as a proxy for the amount of new information gained in each training step.

2.6 State of the Art: FPGA CNN Training Hardware

This section covers the state-of-the-art (SotA) of the of works related to training CNNs on FPGAs.

These works pertain to Caffe Barista and FPGPT which both solely look at accelerating CNN training through the computation of convolutions as GEMMs. This provides it two areas of comparable works. There are the SotA works in performing convolutions as GEMMs and the works that enable CNN training on FPGAs.

2.6.1 FPGA-based Convolutions through GEMMs acceleration

The works described in this section provide optimized GEMM hardware kernels that can be adapted for the training process of CNNs but are not inherently built for such a purpose. To be able to adapt these units to function in CNN training, the input data often needs to be preprocessed, introducing overheads, as they lack features to make them directly applicable to the training problem. These overheads include: a) the latency of the *im2col* process for the INP structure which now needs to be performed on the CPU, which also increases the amount of data that needs to be transferred to the FPGA from the host CPU, and b) cost for performing the necessary permutations and rotations that are required to happen on the CPU incurring further latency penalties.

Despite these works published in isolation and not integrated with common machine learning frameworks such as PyTorch, a discussion of these works is presented here for completeness.

GEMM-HLS [59] is currently the state-of-the-art in matrix multiplication computation when an FPGA device is targeted, with no specific focus on convolutions or CNN workloads. [60] creates a high-efficiency 2D-convolution processor for only 4 parallel 3x3 and 5x5 convolutions, which will not be efficiently utilized across CNN models that usually contain a variety of convolutions dimensions. This makes the design less efficient in the context of training CNNs. [61] propose an efficient convolution architecture which however is not designed to target the CNN training process as it does not address issues such as dilation, permutation and rotation, as well as focusing on 2D convolutions and therefore not incorporating efficient techniques to deal with 4D tensors as seen in CNN workloads. Both [60, 61] do not provide the flexibility required to execute various convolution sizes with a variety of filters as is required for CNN training. Finally, like in the case of GEMM-HLS, for both [60, 61] all permutations, rotations and accumulations must occur on the CPU as these units have not accounted for these specific use cases incurring further latency penalties.

2.6.2 FPGA-based CNN Training

Previous work related to the training of CNNs with FPGAs share the common goal of offloading the entire training process onto the FPGA. They range from multi-FPGA solutions [15, 1, 62, 63] to single FPGA solutions [64, 65, 66, 67, 68, 69, 70]. [15, 1, 63] design training accelerators focused on FPGA-clusters, employing both model and layer parallelism, with a custom workload balancing methodology to allow for linear performance scaling. F-CNN [62], employs a streaming datapath for runtime reconfiguration by overlapping computation and utilizing multi-device platforms. Their approach is to design custom hardware for a subset of CNN layers and distributed them across the devices.

In the realm of single FPGA implementations, [66, 67] focus on placing the entire network on the FPGA. [66] create an ISA for a multi-layer processor, where [67] creates an automatic RTL compiler tailoring Verilog modules from a library to the input network. The work in [65] utilizes low-bit, fixed-precision to reduce architecture complexity and resource consumption. As a trade-off to be able to fit the entirety of training onto a single FPGA, these works assume the entire CNN and its intermediate data can fit on on-chip memory. As a result, none of these works are able to operate on large datasets like ImageNet or large modern networks like GoogLeNet and SqueezeNet.

The work presented in the previous chapter, Barista [64], natively integrated into ML frameworks, performs tiled GEMMs but having the im2col occurring on the host CPU resulted in a memory bounded accelerator that added additional CPU processing time to perform its tiling operations. [15, 1, 65] are all limited by design to small CNN models and/or batch sizes, as they make exclusive use of on-chip memory in order to increase performance, prohibiting their applicability on modern models. [71] presents a highly dynamic GEMM unit that supports compile and runtime configurations. It can support a variety of precisions and number representations and arbitrary GEMM sizes making it suitable for training but no training performance data was presented. [68] presents training in a reduced precision environment, adopting custom number representations like block floating point and a modified stochastic weight averaging low-precision (SWALP) format to train networks on the CIFAR-10 and MNIST datasets to high accuracy.

There are also works creating entire FPGA training frameworks. The NITI training framework presented in [70] is of particular interest. NITI provides a scheme to accelerate training through low-precision number representation with a custom approach to ensure high accuracy results with a demonstrated FPGA acceleration capacity. NITI focuses mainly on the algorithmic aspect of low-precision training, but do present an FPGA implementation, but not results on training CNNs are presented for the FPGA application, only for the GPU. Additionally, DarkFPGA [69] also developed a reduced precision FPGA based training framework. Differing from NITI, DarkFPGA bases its quantization technique on WAGE [72] rather than the 'switch-and-round' technique presented in NITI. For more details on 'shift-and-round' please refer to the introduction to Ch. 3. Additionally, DarkFPGA places more of a focus on a hardware-software co-design that focuses on using batch-level parallelism to accelerate FPGA-based CNN training. NITI on the other hand focuses on integrating custom reduced precision quantization techniques in their loss and weight update functionality that provide additional benefits on custom hardware. DarkFPGA only train on MNIST and CIFAR-10 in the evaluation of their framework. Both of these works present their own frameworks and therefore do require the user to switch to this custom framework.

[15, 1, 65, 62, 66, 67] assume that the entire CNN model can be loaded on the on-chip memory of an FPGA. This means that all intermediate data, like the activations produced at the forward pass that are used for the backward pass computations for $Grad_{IFM}$, need to be stored on the FPGA. For large networks operating on ImageNet inputs that start at 224x224 (compared to 32x32 on CIFAR), the required memory exceeds what can be provided by current FPGAs. This limits the potential network choice and batch sizes that can be utilized, as even on some of the largest FPGAs that Xilinx offers can only fit outdated smaller networks on CIFAR-10.

The work presented in EF-train [73] mainly targets the framework over the FPGA implementation itself. It is aimed at edge-device training and utilizes data restructuring in the framework itself to enable energy efficient training in FP32, enabling AlexNet and VGG16 training on ImageNet.

All aforementioned works require recompilation when targeting new networks, reducing the claimed gains when the compilation times need to be accounted for as part of the training time and hindering their application in cases where the tuning of the topology of the network is also part of training process. The problem is more significant in solutions that target multiple FPGAs as multiple designs need to be compiled. To place this problem in context, as mentioned in Chapter 3 when discussing MuPPET, it takes around 152 hours to train GoogLeNet on ImageNet. Compiling a large FPGA design written in Vitis HLS takes around 12-24hrs. If multiple bitstreams need to be compiled, this can take days, making it almost as expensive as training the network itself.

Moreover, most do not natively integrate with an ML framework due to their custom interfaces, in an attempt to improve the attainable performance. ML Frameworks provide data organized in memory in a specific way, mainly compatible with GPU accelerators. If the FPGA hardware kernel is not built to operate on the memory structure being presented in the way dictated by the framework, a costly memory transformation needs to occur on the host machine increasing runtime latency. To combat this, the authors of FeCaffe [74], created an high efficiency FPGA integration into the Caffe ML framework. Utilizing the existing OpenCL NDRange kernels as the architecture for their FPGA kernels, they were able to offload computation onto the FPGA with seamless integration into Caffe. FeCaffe was designed with the focus on integrating FPGA designs that optimize specific operations or layers seen in CNN training with a popular ML framework, and the authors did not attempt to efficiently utilize the FPGA device. This is the only work that is able to target modern networks such as GoogLeNet and SqueezeNet at FP32 training on the ImageNet dataset.

In summary, FeCaffe [74] is currently the SotA in framework integration as well as allowing for FP32 training of modern networks on relevant datasets. From a hardware design perspective, EF-train [73] is the only work to create an FPGA architecture specifically for convolution able

	Networks	**Dataset**	**Representation**	**FPGAs**
FeCaffe [74]	AlexNet, VGG16, SqueezeNet, GoogLeNet	ImageNet	FP32	1× Stratix 10
EF-Train [73]	AlexNet, VGG16	ImageNet	FP32	1× PYNQ-Z1, 1× ZCU102
FPDeep [1]	AlexNet, VGG16	CIFAR-10	FXP16	15-100× XC7VX690T

Table 2.1: SotA FPGA Convolution Breakdown Summary

to provide high accuracy results due to FP32 training and high performance, but lacking integration characteristics. From a peak performance perspective, FPDeep [1] is currently the SotA for achievable performance obtaining the highest GOPS design, but at FXP16 on the CIFAR-10 dataset and using 15 server-grade FPGAs. Tab. 2.1 summarizes the key characteristics of these three SotA works.

Departing from the previous approaches, the FPGA based works in this thesis aim to be able to compete with the three SotA works by being the first unified toolchain providing a parameterizable architecture of an FPGA-based system for training CNNs at FP32 that does not impose any restriction on the characteristics of the input topologies (size or type of layer), or target dataset, and at the same time is integrated to an existing widely used ML framework: PyTorch [75].

Chapter 3

MuPPET: Multi-Precision Policy Enforced Training

Accelerating CNN training through quantized fixed-point precision

Training CNNs is a computation and resource intensive process that can take weeks to train high accuracy networks. In recent years, Convolutional Neural Networks (CNN) have massively increased in complexity and therefore compute requirements. Initial networks had around 60,000 parameters and 4 layers (LeNet-5) where modern networks have over 138 million parameters (VGG16) and over 200 layers (DenseNet) [76, 77]. As a result, a lot of research has focused on optimizing the inference stage of CNNs on FPGAs and edge-GPUs to allow for low power implementations [78, 42, 79, 80, 81]. Yet, compared to the inference stage which runs in sub-second time, training time for CNNs takes days days, frequently becoming weeks. This difference in computation time is due to the multiple iterations that are required for identifying the training parameters that vary as a function of the network's topology and target dataset. The high resource demand means that only those with expansive and expensive distributed processing clusters can efficiently train CNNs. Thus, to combat the extensive training time, cost and power consumption, the acceleration of the training stage is increasingly attracting attention from industry and academia with state-of-the-art research [64, 1, 65, 62, 1, 15, 82, 66, 67, 63, 73, 74].

The research community largely focuses on distributed training operating on expensive data processing clusters, unavailable to the average user. These works focus on improving communication and scalability within the computing cluster to allow for better performance. Even when accelerated, distributed training requires the financial and logistical capability to host multiple server grade processors i.e. GPUs, TPUs or IPUs [83]. To address a more commonly available setup, the aim for this chapter is to explore acceleration on a heterogeneous single CPU, single GPU setup.

Existing works accelerating heterogeneous CNN training approach the problem from two aspects. Some works focus on an algorithmic perspective, improving distribution, data-movement, pruning, amongst others [84, 31, 30, 28, 29, 12, 25, 26]. Alternatively, some works provide optimized hardware that efficiently execute some or all layers when training CNNs [15, 1, 62, 78, 66, 65, 63]. As research in these areas advances, the idea of hardware-software co-design is growing accordingly.

This chapter focuses on the algorithmic acceleration of a single CPU, single GPU, non-distributed CNN training. There are two core branches to this algorithmic acceleration approach. One algorithmic approach for accelerating CNN training is data management. These techniques rely on improved data reuse, cache optimization, scheduling to minimize memory movement and minimized host-to-processor transfer latency, amongst others [27, 85, 86, 87].

The second approach is reduced precision training. Computation cores utilizing 4, 8 and 16bit precision have been introduced in modern server grade GPUs, such as the Turing architecture for NVIDIA GPUs [11]. The state-of-the-art method in training with reduced precision is mixed-precision training [14]. The authors propose to employ low-precision FP16 computations in the training stage of high-precision CNNs that perform inference in FP32. Along the training phase, the algorithm maintains a high-precision FP32 copy of the network's weights, known as a *master copy*. At each minibatch, the inputs and weights are quantized to FP16 with all computations of the forward and backward pass performed in FP16, yielding memory footprint and runtime savings. Under this scheme, each stochastic gradient descent (SGD) update step entails accumulating FP16 gradients into the FP32 master copy of the weights, with this process performed iteratively throughout the training of the network. Micikevicius et. al. [14] evaluate their scheme over a set of state-of-the-art models on ImageNet, and show that mixed-precision training with FP16 computations achieves comparable accuracy to standard FP32 training. Wang et al. [23] also a method to train an FP32 model using 8-bit floating-point (FP8). The authors propose a hand-crafted FP8 data type, together with a chunk-based computation technique, and employ strategies such as stochastic rounding to alleviate the accuracy loss due to training at reduced precision. For AlexNet, ResNet18 and ResNet50 on ImageNet, Wang et al. [23] demonstrate comparable accuracy to FP32 training while performing computations in FP8. Want et. al. [70] present a newer approach to the same problem that is investigated in [23]. The authors present NITI, an integer only based training framework utilizing block fixed point number representation with a custom 'shift-and-round' scheme to capture as much information when quantizing INT32 matrix multiplication results back to INT8 during training. They also replace the common weight update parameters with a custom learning heuristic that combines the update and the value rounding in a single calculation to maximize information transfer. NITI trained reduced precision networks achieve comparable accuracy compared to FP32 in CIFAR-10 and ImageNet for AlexNet.

At the hardware level, 8-bit fixed-point multiplication uses 18.5× less energy and 27.5× less area with up to 4× lower runtimes than FP32 [88, 89], demonstrated the acceleration potential provided by employing reduced precision training. Consequently, this work attempts to push the boundaries of reduced-precision training by combining the processing performance gains of low-bitwidth fixed-point computations with the floating-point-level accuracy of training an FP32 model.

Building upon [14] is the novel work introduced in this chapter, MuPPET. The MuPPET (Multi-Precision Policy Enforced Training) framework demonstrates the potential of reduced precision when utilized to accelerate CNN training. The goal is to accelerate training without sacrificing the final accuracy of produced networks. Training exclusively in low precision utilizing reduced precision hardware will be notably faster than standard FP32 training. Consequently the network will have a suboptimal final accuracy compared to FP32 training [16]. To prevent any loss in final accuracy, MuPPET progressively increases the precision at which training is performed. By still running a large number of training epochs at reduced precision, MuPPET leverages reduced precision hardware to accelerate CNN training. To ensure the precision in increased at appropriate moments throughout training, MuPPET employs a novel, custom precision switching policy. Additionally, MuPPET is generalizable across a variety of networks and datasets. This ensures MuPPET is a well rounded solution for accelerating CNN training, that is not exclusive to specific CNN architectures or training datasets.

The novel contributions of this chapter are as follows:

- Runtime policy for increasing precision throughout training, generalizable across networks and datasets.
- Accelerated training framework (MuPPET) that suffers no significant penalty in final validation accuracy.

3.1 Core Principles

This section discusses the three underlying principles of full framework of MuPPET. Each individual principle controls a separate part of reduced-precision training MuPPET framework. The combination of these principles allow MuPPET to successfully accelerate training while producing a FP32 model without sacrificing accuracy. The first principle is Gradient Diversity which tracks how effectively the network is learning during the training process at a point in time. This underlies how the precision switching policy decides when to change precision. The second principle is how the quantizations are performed for each input during training. The final principle is multilevel optimization which defines how various levels of quantization are utilized to train a full precision model.

3.1.1 Gradient Diversity

The first core principle is gradient diversity. Gradient diversity was introduced by Yin et al. [33] as a metric for measuring the dissimilarity between sets of gradients that correspond to different minibatches. The gradient diversity of a set of gradients is defined as

$$\begin{aligned}\Delta_{\mathcal{S}}(\mathbf{w}) &= \frac{\sum_{i=1}^{n} ||\nabla f_i(\mathbf{w})||_2^2}{||\sum_{i=1}^{n} \nabla f_i(\mathbf{w})||_2^2} \\ &= \frac{\sum_{i=1}^{n} ||\nabla f_i(\mathbf{w})||_2^2}{\sum_{i=1}^{n} ||\nabla f_i(\mathbf{w})||_2^2 + \sum_{i \neq j} \langle \nabla f_i(\mathbf{w}), \nabla f_j(\mathbf{w}) \rangle}\end{aligned} \tag{3.1}$$

where $\nabla f_i(\mathbf{w})$ is the gradient of weights $\mathbf{w}$ for minibatch i, and n is all minibatches in an epoch.

The key point to note in Eq. (3.1) is that the denominator contains the inner product between the gradients from two *different minibatches*. The more orthogonal the gradients, the smaller the absolute value of the inner product. Thus, orthogonal gradients would result in high gradient diversity, while similar gradients would yield low gradient diversity. Therefore, gradient diversity can be seen as metric of difference in information provided by each set of gradients.

MuPPET uses gradient diversity as a proxy for the amount of new information gained in each training step. For the last minibatch in each epoch, MuPPET computes and stores the average gradient across all layers. Using the history of gradients, MuPPET computes the gradient diversity *across epochs*. The gradient diversity across epochs is used to analyze training progress over time. When the gradient diversity becomes to low and remains as such, this indicates no significant change to the gradients over time. This would indicate that no additional information is being gained across successive epochs, stagnating learning. To counter this stagnation of learning, MuPPET reacts and alters the precision to compensate.

3.1.2 Quantized Training

The second core principle of MuPPET is training in a quantized fashion. This principle is responsible for providing the runtime acceleration itself. Before elaborating on data flows through the network at each iteration, the quantization method is described first. This allows for a clearer understanding of the quantizer components.

Quantization Strategy

In order to implement quantized training, a quantization strategy needs to be defined. The proposed quantization strategy utilizes block floating-point arithmetic also known as dynamic fixed-point (DFXP). In DFXP each fixed-point number is represented as a pair consisting of a $\mathrm{WL}^{\mathrm{net}}$-bit signed integer x and a scale factor s. The number's value is represented as $x \times 2^{-s}$.

During both the forward and backward passes of the training process the weights and feature maps are both quantized. Consequently the corresponding multiplication operations are performed at the same low precision. The quantization method employs a stochastic rounding methodology [22].

Throughout training, and mainly in the convolution layers, there are a large amount of Multiple-Accumulate (MAC) operations. The accumulation stage of the MACs is done with a 32-bit fixed-point value to prevent overflow on the targeted networks. The accumulator word-length is large enough to accommodate the current CNN models without overflow. The result of this convolution is subsequently quantized back to $\mathrm{WL}^{\mathrm{net}}$, the target word-length, before being passed on as an input to the next layer. Within a CNN, the weights and feature maps are represented as a multi-dimensional matrix. Following the dynamic fixed-point scheme, the quantization is performed such that a single scale factor is shared by all values within such a multi-dimensional matrix. The quantization configuration for the i-th level of optimization and the l-th network layer, q_l^i, and the full set of configurations, q^i, are given in Eq. (3.2) and (3.3) respectively,

$$q_l^i = \left\langle \mathrm{WL}^{\mathrm{net}}, s_l^{\mathrm{weights}}, s_l^{\mathrm{act}} \right\rangle^i, \quad \forall l \in [1, |\mathcal{L}|] \tag{3.2}$$

$$q^i = \left\langle q_l^i \mid \forall l \in [1, |\mathcal{L}|] \right\rangle \tag{3.3}$$

where $|\mathcal{L}|$ is the number of layers of the target network, $\mathrm{WL}^{\mathrm{net}}$ is the fixed word-length across the network, s_l^{weights} and s_l^{act} are the scaling factors for the weights and activations respectively, of the l-th network layer for the i-th level of optimization. As a result, for N levels, there are N distinct quantization schemes; $N-1$ of these schemes are with varying fixed-point precisions and the finest level of quantization, q^1, is single-precision floating-point (FP32).

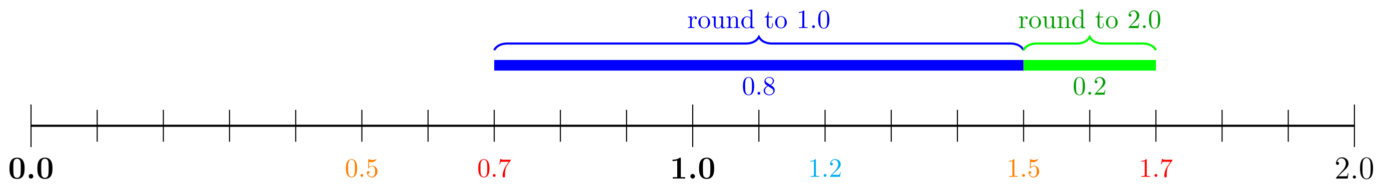


Figure 3.1: Stochastic Rounding

The scaling factor for a matrix $\mathbf{X}$ is first calculated as shown in Eq. (3.4) and individual elements are quantized as in Eq. (3.5).

$$s^{\{\text{weights,act}\}} = \left\lfloor \log_2 \left(\min \left(\frac{\text{UB} + 0.5}{\mathbf{X}_{\text{max}}^{\{\text{weights,act}\}}}, \frac{\text{LB} - 0.5}{\mathbf{X}_{\text{min}}^{\{\text{weights,act}\}}} \right) \right) \right\rfloor \tag{3.4}$$

$$x_{\text{quant}}^{\{\text{weights,act}\}} = \left\lfloor x^{\{\text{weights,act}\}} \cdot 2^{s^{\{\text{weights,act}\}}} + \text{Unif}\left(-0.5, 0.5\right) \right\rceil \tag{3.5}$$

where $\mathbf{X}_{\{\text{max,min}\}}^{\{\text{weights,act}\}}$ is either the maximum or minimum value in the weights or feature maps matrix of the current layer, LB and UB are the lower and upper bound of the current wordlength WL^{net}, and Unif(a,b) represents sampling from the uniform distribution in the range $[a, b]$.

Eq. (3.5) adds a uniformly randomly sampled value in the range $[-0.5, 0.5]$ to the scaled product followed by rounding the sum. This introduces a weighted probability to round to the next absolute value compared to the standard rounding value. The probability is weighted based on how far the scaled value is from its standard rounding value. Figure 3.1 demonstrates the process of the weighting the probability. In this scenario the value to be rounded is 1.2 (shown in cyan). 0.7 and 1.7 (shown in red) indicate the possible range of values of the sum. There is an 80% chance of of rounding to 1.0 as indicated by the blue bar, and a 20% change of rounding to 2.0 as indicated by the green bar. The further a value moves from its standard rounding value, the greater the probability of not rounding to the standard rounding value. For example, if the value to be rounded is 0.6, there would be a 60% change of rounding to 1.0 and a 40% change of rounding to 0.0

Eq. (3.4) adds 0.5 and -0.5 to the upper and lower bounds respectively. These represent the maximum value that could be added due to the uniform sampling required by the stochastic

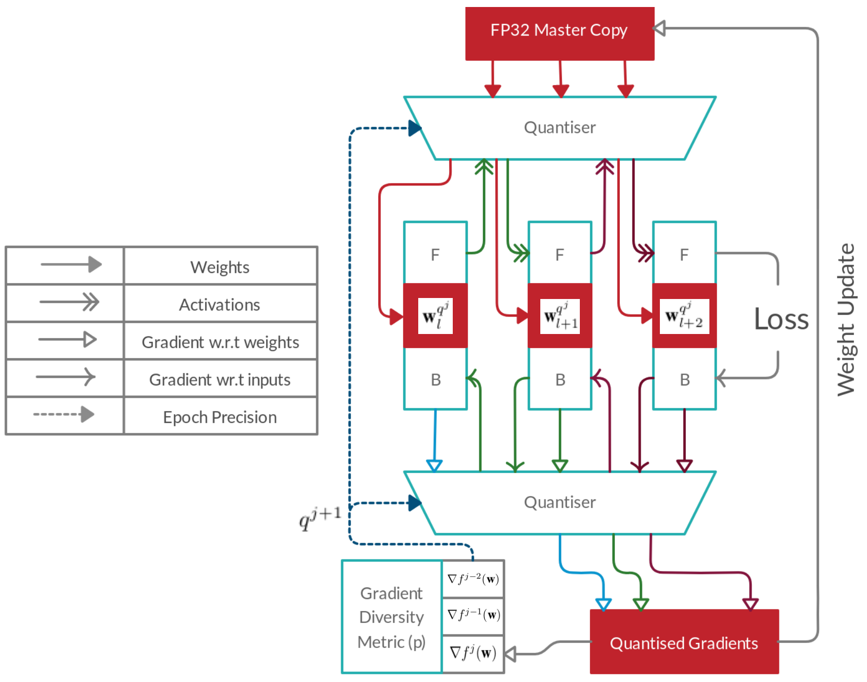


Figure 3.2: Quantized training scheme of MuPPET.

rounding. Therefore, adding 0.5 and -0.5 to the upper and lower bounds respectively ensures maximum utilization of $\mathrm{WL}^{\mathrm{net}}$.

Quantized Execution

For MuPPET to perform multilevel optimization the weights are required to be stored in an FP32 master copy, which are quantized to the desired fixed-point precision (q^j) on-the-fly, where j is the current epoch. The degree of quantization depends on the state of the multilevel optimization that training is in. Further details on multilevel optimization are provided in Section 3.1.3.

Fig. 3.2 depicts the required components for on-the-fly quantized multilevel optimization training utilized in MuPPET. In Fig. 3.2, the $w_l^{q^j}, w_{l+1}^{q^j}, \dots, w_{l+k}^{q^j}$ blocks where $k = |\mathcal{L}|$ represent the quantized weights for a layer. The F and B blocks represent the forward and backward operation respectively, associated with that layer. At epoch j, all Quantizer modules are set to the current quantized precision q^j.

As demonstrated by the 'Weights' arrows, at the beginning of an iteration, the FP32 master copy of the weights is fed to the Quantizer component. After receiving quantized weights, input data is passed through the network. The computations the F and B blocks are performed at the current quantized precision (q^j) as determined by the Quantizer component. The activations and the gradients obtained from each layer are computed in a precision greater than q^j, as mentioned in Section 3.1.2 with regards to the accumulator word length. Rather than just truncating the data, the output of the F and B blocks are requantized by the Quantizer module, adjusting the DFXP scaling factor. The outputs are then either stored or passed on to the next layer. After each iteration of a minibatch, the computed gradients w.r.t. weights computed by the B blocks are passed through a Quantizer component. The full-precision master copy of the weights is updated using the quantized gradient matrices. The same quantized gradient matrices used to update the FP32 master copy are provided to the MuPPET Policy unit (turqoise box bottom left of Fig. 3.2), which determines whether the precision should change. If a change in precision is required, q^{j+1} is passed to the Quantizer components for the next minibatch iteration.

3.1.3 Multilevel Optimization

MuPPET seeks to produce a network with full-precision weights after training. To enable training in reduced precision while retaining an FP32 model, multilevel optimization is implemented. Conventionally, the training process of a CNN can be expressed as in Eq. (3.6).

$$\min_{\mathbf{w}^{(\mathrm{FP32})} \in \mathbb{R}^D} Loss(f(\mathbf{w}^{(\mathrm{FP32})})) \tag{3.6}$$

Given a CNN model f parameterized by a set of weights $\mathbf{w} \in \mathbb{R}^D$, where D is the number of weights of f, training involves a search for weight values that minimize the task-specific empirical loss, $Loss$, on the target dataset. Typically, a predetermined arithmetic precision is employed across the training algorithm. FP32 is currently the *de facto* representation used by the deep learning community for non-distributed training. There are of course other representations such as those described in Sec. 2.5, but these are not the default options used by most. The proposed method follows a different approach by introducing a multilevel optimization scheme [90]. This leverages the performance gains of reduced-precision arithmetic while maintaining an FP32 master copy of the weights.

In [90], the authors only utilize FP16 during training for acceleration. MuPPET aims to take this to a greater extreme by utilizing a multiple precisions. To be able to accelerate training through reduced precision requires the quantization itself to have a low computation load. This excludes many of the highly accurate yet complicated quantization schemes presented in [16, 17, 18, 52, 58, 55, 57, 21]. Training exclusively with a single low precision with a simple quantization will not allow for final FP32 accuracy [88, 23]. The intuition applied here to combat low final accuracy is through simple quantization which is progressively increased over time. A more in-depth analysis of the analysis performed to determine precision throughout training will be discussed in Section 3.2.

In MuPPET, the single optimization problem of Eq. (3.6) is transformed into a series of optimization problems. Each optimization problem employs a uniquely different precision. This means all computations are performed at the specified precision. To enable all computation to be performed at reduced precision requires the weights and inputs for each to be at this precision. Consequently, all the activations and gradients (outputs of the layers) will be in reduced precision too. Nonetheless, the model weights have a master copy stored at FP32 precision to ensure the final result is an FP32 CNN. The weight update step using the reduced precision is appended to the FP32 master copy of the weights. Under this scheme, an N-level formulation comprises N sequential optimization problems to be solved. Each of the N levels corresponds to a progressively higher precision model.

This formulation adds a hierarchical structure to the network training, increasing arithmetic precision across the hierarchy of optimization problems. Starting from the N-th problem, the inputs, weights and activations of the CNN model f are quantized with precision q^N. The N-th problem is the lowest precision during the training process and represents the coarsest version of the model. Each of the N levels progressively employs higher precision until the first level is reached, which corresponds to the original problem of Eq. (3.6).

Formally, at the i-th level, the optimization problem is formulated as

$$\min_{\mathbf{w}^{(q^i)} \in \mathcal{V}} Loss(f(\mathbf{w}^{(q^i)})) \quad \text{s.t.} \mathcal{V} = \left\{ \mathbf{w}^{(q^i)} \in [\mathrm{LB}, \mathrm{UB}]^D \right\} \tag{3.7}$$

where LB and UB are the lower and upper bound in the representational range of precision q^i. In Eq. (3.7) $f(\mathbf{w}^{(q^i)})$ represents the target CNN model with its weights quantized with precision q^i. The master weights remain in FP32 throughout. Hence the solution of this optimization problem can be interpreted as an approximation to the original problem of Eq. (3.6). To transition from one level to the next, the result of each level of optimization is employed as a starting point for the next level. This continues up to the final outermost optimization that is equivalent to Eq. (3.6).

As discussed in Section 3.1.3, the quantization level is gradually increased over the period of the training. The point at which to switch between these optimization levels at the correct times is crucial, as switching must happen as late as possible to achieve the maximum possible acceleration, but on time to ensure the final validation accuracy is not impeded. The proposed precision-switching policy is described in Section 3.2 and determines when it is appropriate to switch precision.

Information Transfer between Levels

The multilevel optimization scheme employed in MuPPET means that at certain points during training the network layers will increase the precision of their computation. This increase requires an appropriate mechanism for transferring information between precision levels. To achieve this, the proposed optimizer maintains a master copy of the weights in full precision (FP32) throughout the entire training process. Similar to mixed-precision training [14], the SGD update step is performed by accumulating a fixed-point gradient value into the FP32 master copy of the weights. Starting from the coarsest quantization level $i = N$, to transfer the solution from level i to level $i-1$, the master copy is quantized using the quantization scheme q^{i-1} rather than q^i. With this approach, the weights are maintained in FP32 and are quantized on-the-fly during runtime to be used throughout training as shown in Eq. (3.8).

$$\mathbf{w}^{(q^{i-1})} \leftarrow \text{Quantise}\left(\mathbf{w}_{\text{master}}^{(\text{FP32})}, q^{i-1}\right) \tag{3.8}$$

When using a heterogeneous system to perform CNN training, a common bottleneck is the movement of data to and from the external processors such as a GPU or an FPGA. For hardware setups with only a single accelerator with enough memory to hold both the quantized and master copy of the weights, the weight update can be performed locally, so the weights do not need to be moved to and from the host CPU. In this scenario the extra memory transfers created by the master copy of the weights is only moved once having a negligible impact on performance. For hardware setups with multiple accelerators training in a distributed fashion or when the accelerator does not have enough memory to hold both the weights and the master weights, the weight update occurs locally. In these training scenarios, weights are being shipped to and from the accelerator anyways, but instead now a weight updates are being returned to the host machine. Although the accumulation happening on the CPU will be slower than when performed on the accelerator, the number of FLOPs for performing the weight updates for the entire model is significantly lower than the execution FLOPS. As a result, although the benefit of parallelism is missed on the CPU, this will not have a significant impact on overall execution time.

3.2 Precision-Switching Policy

The precision switching policy of MuPPET ensures that training remains at the lowest precision possible, for as long as possible, without harming the final inference accuracy. As shown in [89] it is established that 8bit fixed point reduced precision hardware can produce up to $4\times$ lower runtime than FP32. If final inference accuracy can be maintained while benefiting from this acceleration potential, MuPPET can provide acceleration with no negative tradeoff. The policy is based on the first core principle, gradient diversity [33], but modifies it to suit the purpose of tracking learning progress throughout training. As discussed in Section 3.1.1, [33] computes the gradient diversity between gradients across minibatches. Contrarily, MuPPET computes $\Delta_{\mathcal{S}}(\mathbf{w})$ (Eq. (3.1)) between gradients obtained across epochs. In the context of MuPPET, gradient diversity serves as a proxy to measure the information that is obtained during the training process. The intuition is, the lower the diversity between the gradients, the less new information this iteration of the training process at this level of quantization provides towards the training of the model. This is not formally verified due to the complexity of moving in a high dimensional hyperspace. Nonetheless, empirically gradient diversity performs very well as an indicator for the amount of information gained throughout training. Therefore, the proposed method comprises a novel normalized inter-epoch version of the gradient diversity. This novel version of the gradient diversity drives the run-time policy that determines the epochs at which to switch precision.

The following policy is employed to determine when a precision switch is to be performed. For a network with layers $\mathcal{L}$ and a quantization scheme q^i that was switched into at epoch e:

1. For each epoch j and each layer $l \in \mathcal{L}$, the last minibatch's gradient, $\nabla f_l^j(\mathbf{w})$, is stored.

2. After r (resolution) number of epochs, the inter-epoch gradient diversity at epoch j is

$$\Delta_{\mathcal{S}}(\mathbf{w})^j = \frac{\sum_{\forall l \in \mathcal{L}} \frac{\sum_{k=j-r}^{j} ||\nabla f_l^k(\mathbf{w})||_2^2}{||\sum_{k=j-r}^{j} \nabla f_l^k(\mathbf{w})||_2^2}}{|\mathcal{L}|} \tag{3.9}$$

   The gradient diversity is computed across all layers of the network as the policy applies globally across the network.

3. At an epoch j, given a set of gradient diversities

$$\mathcal{S}(j) = \left\{ \Delta_{\mathcal{S}}(\mathbf{w})^i \;\; \forall \; e \leq i < j \right\} \tag{3.10}$$

   the ratio:

$$p = \frac{\max \mathcal{S}(j)}{\Delta_{\mathcal{S}}(\mathbf{w})^j} \tag{3.11}$$

   is calculated.

4. An empirically determined decaying threshold

$$T = \alpha + \beta e^{-\lambda j} \tag{3.12}$$

   is placed on the ratio p.

5. If p violates T more than γ times, a precision switch is triggered and $\mathcal{S}(j) = \emptyset$.

As long as the gradients across epochs remain diverse, $\Delta_{\mathcal{S}}(\mathbf{w})^j$ (Eq. (3.9)) at the denominator of p sustains a high value and the value of p remains low. However, when the gradients across epochs become similar, $\Delta_{\mathcal{S}}(\mathbf{w})^j$ decreases and the value of p becomes larger.

The intuition behind increasing precision gradually is related to how learning rate is used to traverse the learning hyperspace. Early on training must avoid getting stuck in local minima, which is why the learning rate is kept at a high value. This results in the network traversing the hyperspace in large steps in varying directions. As the network starts to settle for a minima, the learning rate is decreased as traversal becomes more nuanced. Early on in training, very high precision is of little importance as the difference in direction is very high. As training progresses, more nuance is required in traversing the space. In the space traversal requiring more nuance, the value of p will increase as at the current precision, the directions seem very similar. The high value of p will violate the threshold and increase the networks operating precision. By increasing the precision at which the network operates, more information about the direction becomes available and p will be low again. The threshold decays over time, to reflect the need to have an increasingly nuanced traversal over time.

As stated at the beginning of this chapter, the generalisability of the policy across epochs, networks and datasets is vital for making MuPPET a versatile tool that can be used in any training scenario. Beyond versatility, generalisability simplifies training for the user as well. By making MuPPET a tool that can generalize across a variety of networks and datasets, the user does not need to perform any hyperparameter tuning. Hyperparameter tuning forces the user to rerun training multiple times drastically increasing training time. Ensuring generalisability means there is no need for multiple training runs at any point in time. This is essential as providing acceleration to training but requiring multiple additional iterations of training would nullify any benefit provided by MuPPET.

Generalisability across epochs is obtained as p accounts for the change in information relative to the maximum information available since the last precision change. Hence, the metric acknowledges the presence of temporal variations in information provided by the gradients. *Generalisability across networks* and *datasets* is maintained as p measures a ratio. Thus the absolute values of gradients which could vary between networks and datasets, matter less. These claims of generalisability investigated and supported in Section 3.3.2.

Overall, MuPPET employs the metric p as a mechanism to trigger a precision switch whenever p violates threshold T more than γ times. The likelihood of observing r gradients across r epochs that have low gradient diversity, especially at early stages of training is low. The intuition applied here is that when this does happen at a given precision, it may be an indication that information is being lost due to quantization. Too much information lost through quantization corresponds to a high p value, which argues moving to a higher bitwidth.

The precision switching policy stays active until the precision is finally switched from 16-bit fixed point to FP32. 16-bit fixed point is chosen as the highest precision fixed point value as the performance benefit between FP32 and 32-bit fixed point is not worth exploring. Once FP32 is reached, the rest of the training is performed at FP32 for a fixed number of cycles until the desired validation accuracy is reached. In these final FP32 training epochs the learning rate schedule normally associated with the network is used to rapidly fine tune the network to FP32 accuracy. These are referred to as the epochs where learning rate is changed. This fixed number of cycles is another MuPPET hyperparameter to be tuned and generalized.

3.2.1 Hyperparameters

For MuPPET there are two sets of hyperparameters. The basic hyperparameters are the ones commonly seen during CNN training such as; learning rate schedule, optimizer, batch size, epochs, momentum, weight decay, dropout ratio. The hyperparameters specific to the

MuPPET algorithm are the following:

1. Values of α, β, and λ that define the decaying threshold from Eq. (3.12)
2. The number of threshold violations, γ allowed before the precision change is triggered (*patience*)
3. The resolution r
4. The set of precisions at which training is performed
5. The epochs at which the learning rate is changed

The values of α, β, λ, r, and γ were set at 1, 1.5, 0.1, 3, and 2 respectively after empirical cross-validation. These were tuned by running training on AlexNet and ResNet20 on the CIFAR-10 dataset.

The empirically-chosen quantized precisions at which training was performed were 8-, 12-, 14- and 16-bit fixed-point. When attempting to start at a precision lower than this, the training did not result in any progress towards convergence for any network. This indicates that starting too aggressively with quantization will mean training struggles to move in the correct direction at all. The reason 10-bit fixed point was not included is because the extra cycles spent getting out of 10-bit precision did not benefit final accuracy and increased the total number of epochs spent training. This negatively affected total runtime. Dropping out 12-bit fixed point was considered, but empirically the performance benefit did not outweigh the impact on final accuracy.

As discussed, MuPPET operates on the intuition that that decaying the learning rate causes a finer exploration of the optimization space as does increasing the quantization level. When attempting to merge learning rates with the quantization schedule, training was no longer stable and reliable. Whenever quantization or learning rate changed too close together there was a chance of training derailing. Therefore, the learning rate was kept constant during quantized training and was decayed only after switching to FP32. The number of cycles spent decaying the learning rate was set to 15 epochs per value in the learning rate schedule. More advanced schedules such as polynomial, exponential, cosineannealing with or without warm restarts and cyclic learning rate schedules were not analyzed for this work [91]. If they were to be used they should only be activated after FP32 is reached, but the quality and generalizability of these schedules was not tested for the results presented.

All MuPPET hyperparameters remain the same regardless of network or dataset. To verify the validity of this assumption, after the initial hyperparameter turning the generalizability of this hyperparameter setup across networks and datasets was validated on AlexNet, ResNet-18 and GoogLeNet on CIFAR-10 and ImageNet. Analysis of generalizability and the training hyperparameters used are presented in Section 3.3.2.

Regarding the standard training hyperparameters (with the exception of learning rate schedule), batch size was increased from 128 to 256 going from CIFAR-10 to ImageNet. The remaining hyperparameters such as loss calculation, having no distribution policy with only a single GPU, amongst others, followed PyTorch defaults. All training hyperparameters, remained constant, following no schedule or change during the quantized portion of training. Once FP32 is achieved, learning rate is no longer required to remain constant.

Overall, MuPPET introduces a policy that decides at run time an appropriate point to switch between quantization levels to achieve maximal acceleration through by keeping training at the lowest precision during training without compromising final inference accuracy.

3.3 Evaluation

MuPPET was developed to be both effective at switching between precisions for acceleration without impacting inference accuracy and generalizing well across networks and datasets (generalisability) without the need of parameter tuning. Precision switching, discussed in detail in Section 3.3.5, is evaluated by exploring how the epoch at which precision switches occur affects the final validation accuracy. Generalisability, discussed in detail in Section 3.3.2, is evaluated by exploring the values taken by the metric p across time (epochs), architectures (networks) and datasets.

3.3.1 Setup

All results presented in the MuPPET evaluation were performed on the same hardware. The baseline FP32 results were also collected on the hardware setup as the MuPPET execution to ensure a fair analysis. The framework that was utilized for training was PyTorch, and all training happened on a heterogeneous system of an NVIDIA RTX 2080Ti GPU with an Intel Xeon Bronze 3106 CPU.

Regarding timing the quantization, the 2080Ti GPU has reduced precision cores, but these were unusable through PyTorch. As a result the convolutions were replicated and executed on the NVIDIA CUTLASS library [92] on the 2080Ti which is based on the Turing architecture with contains reduced precision tensor cores. This profiled time was then combined with a profiled run on the PyTorch training run to emulate how long training would have taken if it was integrated within PyTorch.

The MuPPET specific hyperparameters were manually tested through AlexNet and ResNet20 runs on CIFAR-10. By tuning each parameter, seeing how the system responded and updating the hyperparameters accordingly lead to the values of α, β, λ, r and γ. The validity of assuming these tuned hyperparameters for other networks and datasets will be explored during the evaluation. The learning rate during each network's training run was set to 15 epochs per learning rate value once FP32 was reached based on the learning rate schedule presented for each work. The other training hyperparameters such as momentum, decay and batchsize were kept to those presented by the authors of the networks, with SGD employed as the optimizer. Anything not specified by the authors of the papers were left as PyTorch default values.

The datasets followed the default testing and training split provided by the datasets themselves. Within the training dataset, an 80:20 split was used for the training and validation sets. The 80:20 split is done at random to ensure a fair train:validation split.

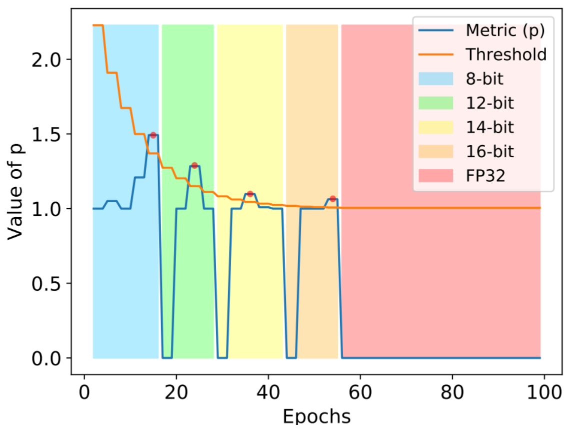
All training run data was collected after running each test 5 times and taking the average result to account for any possible fluctuation due to hardware or effect of the random starting setup used for training. Furthermore, the CUTLASS results were also collected 5 times and averaged to ensure fairness.

The focus networks for MuPPET were AlexNet, ResNet18, ResNet20 and GoogLeNet. At the time of creating this work, these were popular networks (AlexNet) as well as modern networks (ResNet and GoogLeNet) that operated on CIFAR-10, CIFAR-100 and ImageNet. It was necessary to operate on both datasets to underline the generalizability of MuPPET. Furthermore, the sizes of convolutions seen across these networks has the greatest diversity including 1x1, 3x3, 5x5 and 7x7 convolutions for the forward pass. This diversity of convolutions again tests the generalizability of MuPPET to see if it performs as well for varying workloads. GoogLeNet and ResNet also both have a custom convolution block with fire and residual modules. Testing these advanced modules over just straight forward connections like AlexNet were used to again help demonstrate the flexibility and generalizability of the precision switching policy.

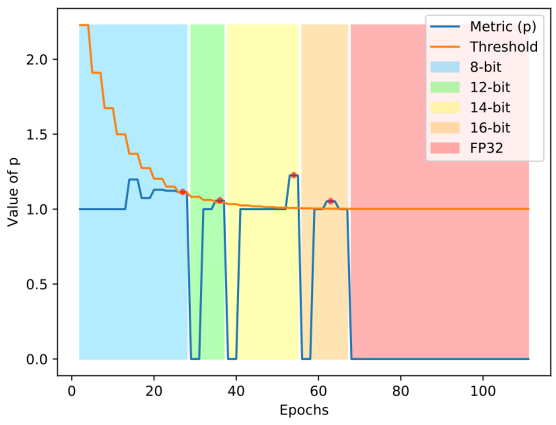
3.3.2 Generalisability

The MuPPET framework was evaluated on its generalisability across epochs, networks and datasets. As mentioned in Sec. 3.2.1, the MuPPET hyperparameters were empirically chosen. To avoid data peeking and ensure correct out-sample testing, the hyperparameters are not altered after the AlexNet and ResNet20 CIFAR-10 testing. This ensures that the analysis performed on any ImageNet networks as well as the other CIFAR-10 networks are evaluated in a fair and objective manner. For AlexNet and ResNet20 CIFAR-10, the results can be considered to have been influenced by the training and testing data, but this does not affect the conclusions presented in this section.

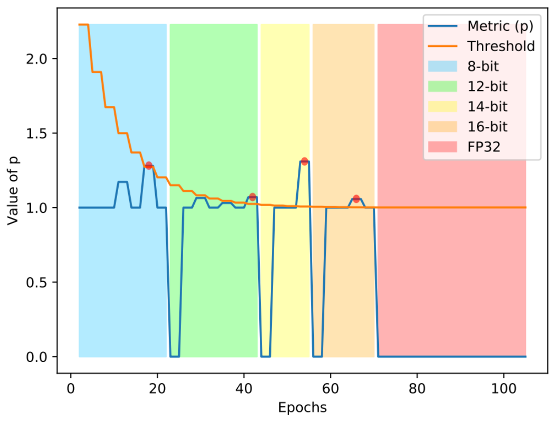
Fig. 3.3 shows the value of the metric p over the epochs in blue, and the decaying threshold described in Eq. (3.12) in orange. The number of epochs for which training in each precision was performed is shown by the various overlay colors. The first violation is denoted by a red dot and the second violation is not seen as it results in violations $= \gamma$ making it the point of switching. The graphs show that across various networks and datasets, the values of p stay relatively similar, validating the choice of a universal decaying factor. Furthermore, empirical



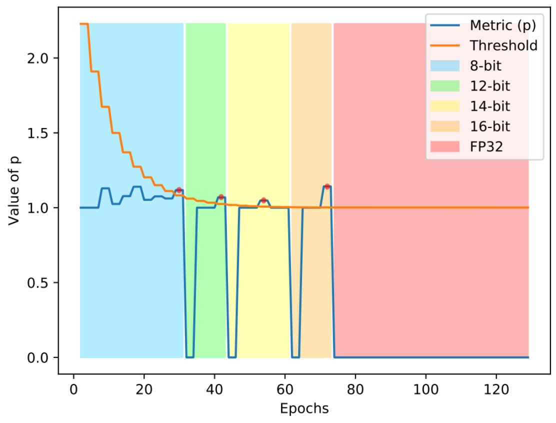
(a) AlexNet - CIFAR-10



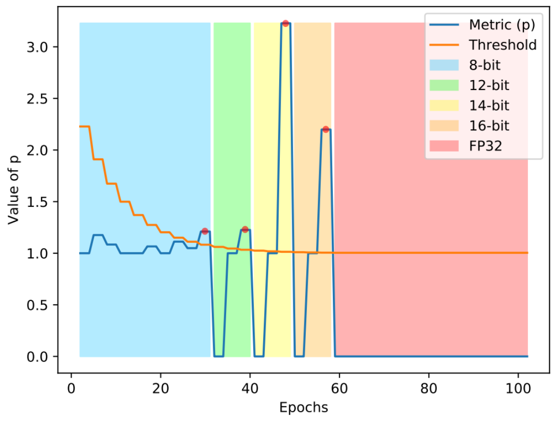
(b) AlexNet - ImageNet



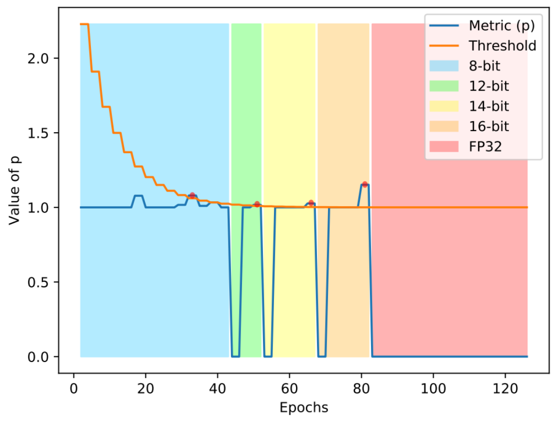
(c) ResNet20 - CIFAR-10



(d) ResNet18 - ImageNet



(e) GoogLeNet - CIFAR-10



(f) GoogLeNet - ImageNet

Figure 3.3: Demonstration of the generalisability of p over networks, datasets and epochs. Figures show the metric p (blue) and the decaying threshold T (orange) over epochs of training.

results for CIFAR-10 indicated that changing from one fixed-point precision to another too early in the training process had a negative impact on the final validation accuracy. Using a decaying threshold ensures that the value of p needs to be much higher in the initial epochs to trigger a precision change. The initially higher threshold is due to the volatility of p in early epochs of training, related to the intuition behind the precision switching policy (Section 3.1.3).

Looking at the GoogLeNet graphs (Fig. 3.3e and Fig. 3.3f), the p value is less representative of the overall trend compared to the other networks. This can be explained due to the fact that the p metric is based on AlexNet and ResNet20 CIFAR-10 training runs. Nonetheless, the p metric still ensures that GoogLeNet is able to achieve comparable accuracy for FP32 training at an accelerated pace (see tab. 3.1). This demonstrates that the P metrics is sufficient for now but can definitely benefit from being tailored to each network for even greater performance benefit. As this was outside of the scope of this work, further analysis was not performed but should be considered for the future.

3.3.3 Network Accuracy

The core aim of MuPPET was to provide acceleration of the CNN training process. Nonetheless, the accuracy of the trained network must be competitive with the state-of-the-art works, or there is no point in accelerating the training process.

The accuracy results presented in this section utilized the previously proposed stochastic quantization strategy. PyTorch was used as the CNN training framework for MuPPET and all results presented henceforth. The PyTorch framework does not natively support low-precision implementations. Consequently, all quantization and computations corresponding to 8-, 12-, 14-, and 16-bit precisions were performed through emulation on floating-point hardware. The code for this emulation is open sourced at: https://github.com/ICIdsl/pytorch_training.git. All hyperparameters not specified below were left as PyTorch defaults. For all networks, an SGD optimizer was used with:

- Batch Size : 128 (CIFAR-10 & CIFAR-100), 256 (ImageNet)

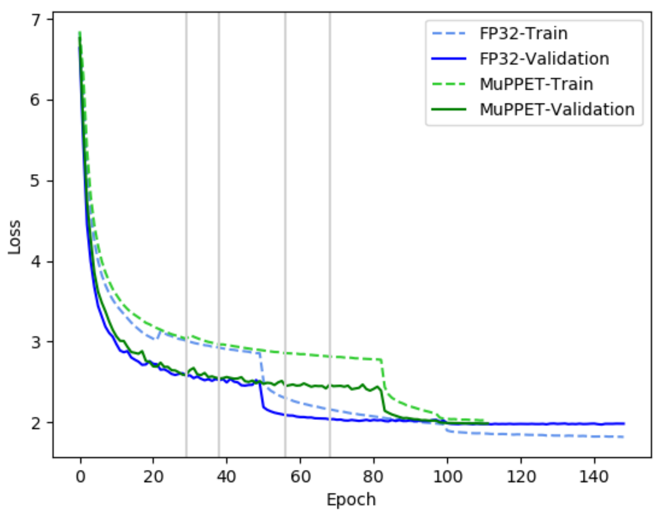
- Momentum : 0.9
- Weight Decay : $1e^{-4}$

As a baseline, an FP32 model with identical hyperparameters (except for learning rate schedule) was trained. The baseline FP32 training was performed by training for 150 epochs and reducing the learning rate by a factor of 10 at epochs 50 and 100. In order to achieve comparable final validation accuracy to the FP32 baseline, once MuPPET triggers a precision change out of 16-bit fixed-point, 45 training epochs at FP32 precision are performed. The learning rate is reduced by a factor of 10 every 15 FP32 training epochs. For AlexNet, ResNet18, ResNet20, and GoogLeNet, the initial learning rate is set to 0.01, 0.1, 0.1, and 0.001 respectively, and for MuPPET remained there until FP32 was reached. The ImageNet training and validation loss curves can be seen in the Fig.3.4. For each of the graphs, the light gray lines indicate the epoch at which precision was switched in the MuPPET run. The green lines follow MuPPET runs and the blue FP32 training. Solid lines show validation loss and dashed training loss.

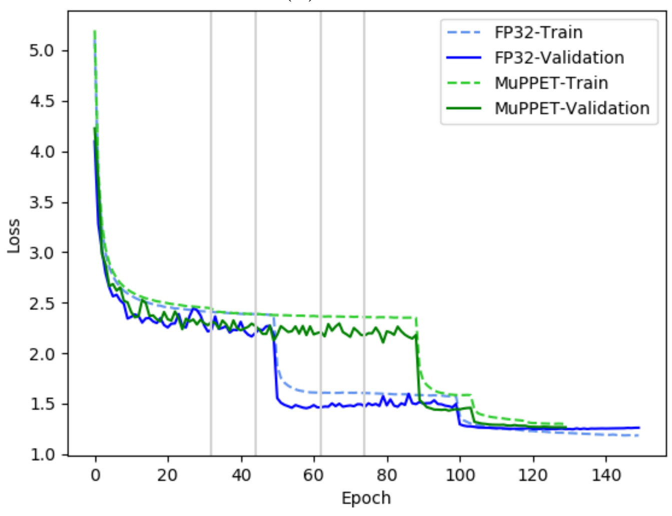
	CIFAR-10			**CIFAR-100**			**ImageNet**		
	FP32	MuPPET	Diff (pp)	FP32	MuPPET	Diff (pp)	FP32	MuPPET	Diff (pp)
AlexNet	75.45	74.49	-0.96	39.20	38.19	-0.99	56.21	55.33	-0.88
ResNet	90.08	90.86	0.78	64.60	65.80	1.20	69.48	69.09	-0.39
GoogLeNet	89.23	89.47	0.24	62.90	65.70	2.80	59.15	63.70	4.55

Table 3.1: Top-1 test accuracy (%) on CIFAR-10/100 and ImageNet (ILSVRC12 Validation Set) for FP32 baseline and MuPPET. Results averaged over 5 runs.

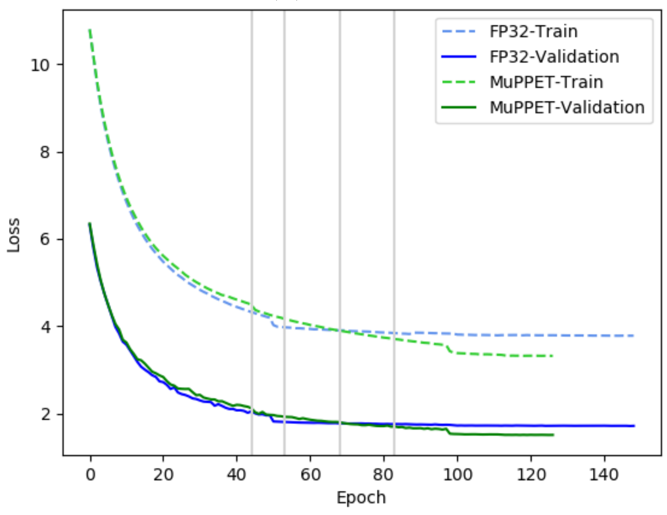
Table 3.1 presents the achieved Top-1 validation accuracy of MuPPET and the FP32 baseline, together with the accuracy difference in percentage points (pp). This is based on the same data seen in Fig. 3.4. As shown on the table, MuPPET is able to provide comparable Top-1 validation accuracy to standard FP32 training across both networks and datasets. Training setup of GoogLeNet on ImageNet was sub-optimal due to not using cosineannealing for the learning rate schedule. As a result the baseline and MuPPET training underperformed compared to reported state-of-the-art works. Although this is a limitations of analysis, the results nevertheless demonstrate the quality of training with MuPPET using identical hyperparameters. MuPPET's performance demonstrates the effectiveness of the precision-switching strategy in



(a) AlexNet



(b) ResNet18



(c) GoogLeNet

Figure 3.4: FP32 vs MuPPET training and validation loss graphs for different networks on the ILSVRC12 dataset.

achieving significant acceleration of training time (Section 3.3.4) at negligible cost in accuracy by running many epochs at lower precision, particularly on very large datasets. Being able to closely match the final accuracy of the FP32 baseline demonstrates that MuPPET is able to match accuracy. This indicates that any runtime improvements achieved do not come at the cost of final inference accuracy. This allows for acceleration without needing to make a sacrifice for accuracy.

Additionally to matching accuracy with the baseline, MuPPET inherently enables a reduced number of epochs as shown in Fig. 3.4. Traditional training approaches force the user to select a number of training epochs at the start, a parameter that is only tuned through multiple runs or experience. MuPPET on the other hand automatically changes precision depending on how training is progressing. In the cases for AlexNet, ResNet18 and GoogLeNet not only were early epochs executed faster, the overall number of epochs required to get to a similar precision was reduced too. This is an additional vector of runtime acceleration available to MuPPET.

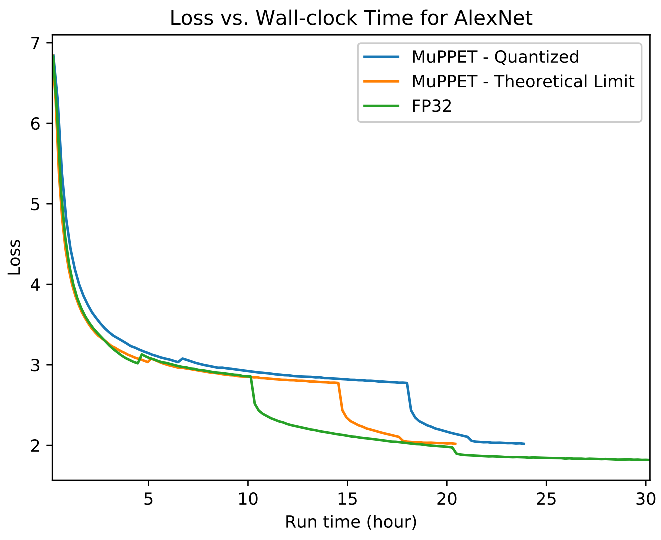
3.3.4 Wall-clock Time Improvements

This section explores the estimated wall-clock time improvement of the current implementation of MuPPET (Current Impl.) with respect to baseline FP32 training. This is presented alongside the performance of Mixed Precision by Micikevicius et al. [14] (multi-level optimization with FP16 computation with FP32 master copy of weights) and MuPPET's ideal implementation. For all performance results, the target platform was an NVIDIA RTX 2080 Ti GPU.

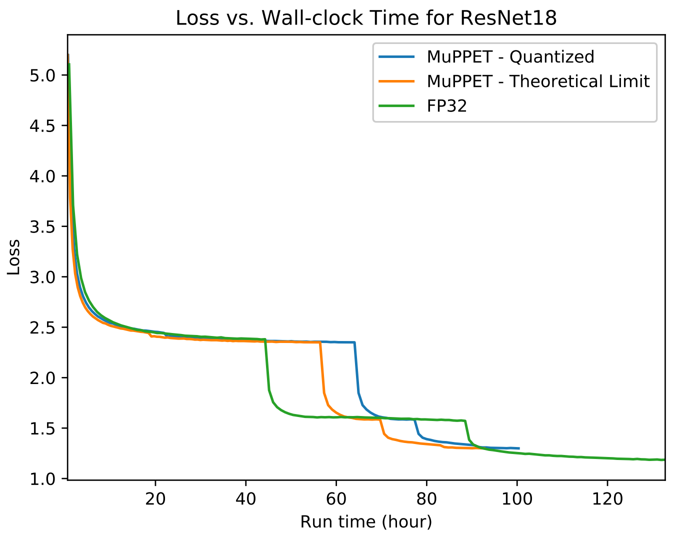
At the moment, deep learning frameworks, such as PyTorch, do not provide native support for reduced-precision hardware except for FP16 used by Micikevicius et al. [14] or emulated quantized training. Utilizing real quantized hardware is currently unavailable in the standard version of PyTorch. Consequently, the wall-clock times provided in Table 3.2 were estimated using a performance model. The performance model was developed using NVIDIA's CUTLASS library [92] for reduced-precision general matrix-multiplication (GEMM) employing Turing architecture GPUs. The entire training process in PyTorch was profiled including the following MuPPET specific operations: quantization of weights, activations and gradients as well as all calculations pertaining to $\Delta_{\mathcal{S}}(\mathbf{w})^j$ (Eq. 3.9). The GEMMs that were accelerated were in the convolutional and fully-connected layers of each network. INT8 hardware was used to profile the 8-bit fixed-point computations, while FP16 hardware was used to profile 12-, 14-, and 16-bit fixed-point computations as well as Mixed Precision [14] wall-clock time. CUTLASS [92] natively implements bit-packing to capitalize on improved memory-bandwidth utilization. To exploit further performance benefits by utilizing 12- and 14-bit fixed-point computations, custom hardware such as FPGAs or ASICs would be required. This thesis does go on to explore potential FPGA designs that can exploit these custom precisions. The model for the current implementation is limited by the fact that current frameworks force quantization of activations in between layers to happen to and from FP32. For the MuPPET (Ideal) scenario and Mixed Precision [14], the model assumes native hardware utilization for all deployed data representations which would eliminate the current overheads.

The orange line in Fig. 3.5 establishes an acceleration bound by ignoring stochastic quantization overheads due to current hardware restrictions. The blue line incorporates these overheads into the performance model.

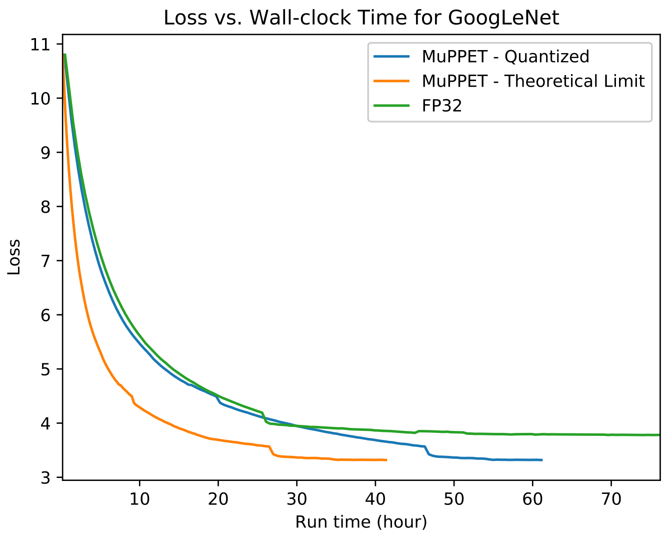
As shown in Table 3.2 and Fig. 3.5, MuPPET consistently achieves 1.25-1.32$\times$ speedup over the FP32 baseline across the networks when targeting ImageNet on the given GPU. In Fig. 3.5 it can be seen that the training loss achieved by the FP32 baseline is better than that of MuPPET. Yet, as demonstrated in Table 3.1 this does not translate equally to Top1 test accuracy, the main method of evaluating the quality of the trained network where MuPPET matches FP32 baseline quality. Fig. 3.5 indicates that the FP32 baseline runs for epochs near the end of training where there is only marginal loss improvement. This is an inherent limitation to the current technique of manually adjusting and tuning the training process. MuPPET is able to produce the presented result in its first iteration whenever applied to a new network or dataset, as was demonstrated in Sec. 3.3.2. It might be possible to reduce the epochs and stop training earlier for the FP32 baseline with a manual tuning technique. Such an approach would require a large amount of training reruns making the effective training time significantly greater than it already is. There is also the consideration that performing more epochs as FP32 for MuPPET could potentially improve the test accuracy even further. Considering the Top-1 test accuracy



(a) AlexNet



(b) ResNet18



(c) GoogLeNet

Figure 3.5: Loss vs estimated GPU wall-clock times for various networks on ImageNet.

	FP32 (Baseline)	**Mixed Prec [14]**	**MuPPET (Current Impl.)**	**MuPPET (Ideal)**
AlexNet	30:13 (1×)	29.20 (1.03×)	23:52 (1.27×)	20:25 (1.48×)
ResNet18	132:46 (1×)	97:25 (1.36×)	100:19 (1.32×)	92:43 (1.43×)
GoogLeNet	152:28 (1×)	122:51 (1.24×)	122:13 (1.25×)	82:38 (1.84×)

Table 3.2: Wall-clock time (GPU hours:mins) & relative acceleration for networks targeting ImageNet.

for MuPPET is already so close to the FP32 baseline and the target is performance benefits, this was not explored any further.

With respect to Mixed Precision, the proposed method outperforms it on AlexNet by 1.23× and delivers comparable performance for ResNet18 and GoogLeNet. Currently, the absence of native quantization support, and hence the necessity to emulate quantization and the associated overheads, is the limiting factor for MuPPET to achieve higher processing speed. In this respect, MuPPET runs on native hardware would yield 1.05× and 1.48× speedup for ResNet18 and GoogLeNet respectively compared to Mixed Precision. As a result, MuPPET demonstrates consistently faster time-to-accuracy [93] compared to Mixed Precision across the benchmarks. Additionally, while Mixed Precision has already reached its limit by using FP16 on FP16-native GPUs, the 8-, 12-, 14- and 16-bit fixed-point computations enabled by MuPPET leave space for further potential speedup when targeting next- and current-generation [12] precision-optimized fixed-point platforms. Similar to the analysis presented in Section 3.3.3, Micikevicius et al. [14] and Wang et al. [23] compare their respective schemes to baseline FP32 training performed by them. The reported results demonstrate that their methods achieve similar accuracy results to our method by lying close to the respective FP32 training accuracy. The work presented by Wang et al. [23] does not provide any results for gains in wall-clock times and provide only emulated results from an in-house (not open source) machine learning framework. Furthermore they propose a custom FP8 data type and computation and could build only a subset of their proposed ideas on an ASIC. For these reasons their work could not be directly compared to our method.

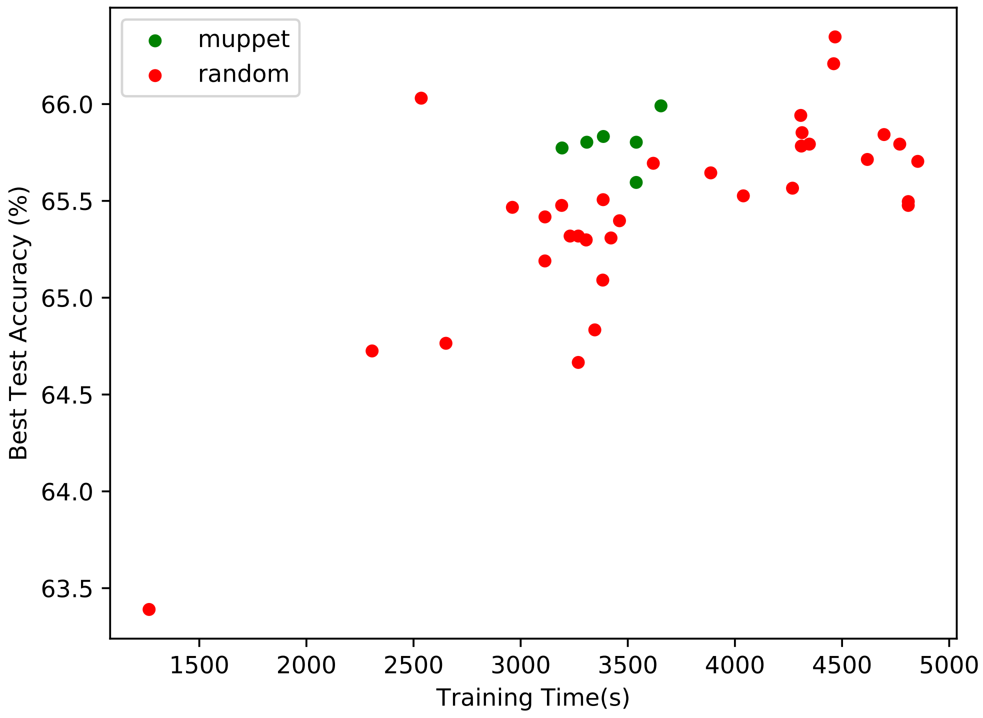


Figure 3.6: Test accuracy vs time trade-off for ResNet20 MuPPET runs on CIFAR-100.

3.3.5 Precision Switching

To evaluate the ability of MuPPET to effectively choose an epoch to switch precision at, AlexNet and ResNet20 were first trained using MuPPET on the CIFAR-100 dataset and the epochs at which the learning switched precision was recorded. The hyperparameters for MuPPET were kept the same across all runs.

Following this, AlexNet and ResNet20 were trained using only 8-bit fixed-point computations updating an FP32 master copy of weights with learning rate changes performed at the same epochs as MuPPET, and the overall number of epochs identical MuPPET. This way, the only hyperparameter that differed between these runs was the precision at which computations were performed. Using checkpoints from this 8-bit training run, the precision was switched from 8-bit to FP32 at epochs 10, 15, 25, 50, and 75. The zero column remains in 8-bit fixed-point for the entire duration of training. The best validation accuracy for each of these runs is shown in Table 3.3. The value in brackets next to the validation accuracy of the MuPPET run is the epoch at which MuPPET switched out of 8-bit into 12-bit precision. The range of optimal switching epochs differ between AlexNet (10-15) and ResNet20 (25-50). For ResNet20,

	Switching Epoch (8-bit to FP32)						
	0	10	15	25	50	75	MuPPET
AlexNet	36.6	39.0	37.8	37.9	37.6	37.3	38.0 (19)
ResNet20	64.5	64.4	65.1	65.5	65.4	65.3	65.8 (37)

Table 3.3: Top-1 validation accuracy (%) on CIFAR-100 switching from 8-bit fixed-point to FP32 at epochs 10, 15, 25, 50 and 75

	ResNet20		**GoogLeNet**	
	CIFAR-10	CIFAR-100	CIFAR-10	CIFAR-100
ResNet20	65.01	**65.80**	-	65.0
GoogLeNet	-	64.00	64.70	**65.70**

Table 3.4: Evaluation of MuPPET's ability to tailor to dataset and network combination using Top-1 test accuracy on CIFAR-100.

MuPPET performs as expected and switches out of 8-bit precision at epoch 37 which falls well within the range of optimal switching epochs. For AlexNet MuPPET switches outside of the optimal range at epoch 19, not between 10-15. In doing so, MuPPET manages to exploit computational acceleration from reduced-precision arithmetic to a greater extent by staying at reduced precision for longer, while still limiting damage to the validation accuracy. From the results note that training at reduced precision and not switching at all (column 0 in Table 3.3) causes a drop in validation accuracy of 1.4 and 1.3 percentage points for AlexNet and ResNet20 respectively. This drop demonstrates the need to switch precisions when training at bitwidths as low as 8-bit fixed-point.

To demonstrate the benefits of a precision-switching methodology, two further sets of experiments were conducted on ResNet20 using CIFAR-100. First, 34 training runs were performed (34 red dots in Fig. 3.6), where for each run four epochs along the standard training duration were randomly selected and used as the switching points. Fig. 3.6 shows the best test accuracy achieved by each of the runs and the training time as estimated by our performance model described in Section 3.3.4. It shows that for a given time-budget, MuPPET runs (6 green dots) outperform on average all other experiment sets, demonstrating the need for a precision-switching policy that is real-time in order to achieve a good accuracy-to-training-time trade-off. The accuracy and time have mean ± standard deviation of (65.54%±0.06, 3429.19s±46.1) and

(65.44%±0.09, 3679.15s±141.1) for the green and red points respectively.

To demonstrate that MuPPET tailors the training process to the network-dataset pair while using a policy that is agnostic of the network-dataset combination, the experiments in Table 3.4 were performed. The switching strategy generated by MuPPET for both ResNet20 and GoogLeNet on CIFAR-10 and CIFAR-100 were recorded. The experiments shown in Table 3.4 used the CIFAR-100 dataset. The bold values show the accuracy achieved by the schedule proposed by MuPPET for the corresponding network-dataset pair. For the ResNet20 row, column 1 shows the accuracy achieved if the precision schedule proposed by MuPPET on the CIFAR-10 dataset for ResNet20 was applied to CIFAR-100 training (dataset tailoring). In the same row, column 4 shows the accuracy obtained when CIFAR-100 training was performed on ResNet20 using GoogLeNet's CIFAR-100 MuPPET schedule (network tailoring). Similar results are shown for GoogLeNet. As all experiments underperform compared to the MuPPET training process, Table 3.4 demonstrates the ability of MuPPET to tailor the process to a network-dataset combination.

3.4 Conclusion

This chapter introduced and analyzed MuPPET; a novel low-precision CNN training scheme that combines the use of fixed-point and floating-point representations to accelerate the production of a network trained for FP32 inference. MuPPET introduces a precision-switching mechanism that decides at run time an appropriate transition point between different precision regimes. It is demonstrated that the precision switching policy selects highly appropriates times to switch for the best accuracy-acceleration tradeoff, which cannot be matched through random selection. The proposed framework achieves Top-1 validation accuracy comparable to that achieved by state-of-the-art FP32 training regimes while delivering significant speedup in terms of training time. Quantitative evaluation demonstrates that MuPPET's training strategy generalizes across CNN architectures and datasets by adapting the training process to the target CNN-dataset pair during run time.

3.4.1 Limitations

Overall, MuPPET enables the utilization of the low-precision hardware units available on modern specialized processors for machine learning, such as next-generation GPUs, FPGAs and TPUs, to yield improvements in training time and energy efficiency without impacting the resulting accuracy. Unfortunately the hardware utilized for this acceleration does not properly support the precisions which MuPPET relies on. This diminished the acceleration provided by MuPPET and ensured it potential was not fully exploited. Furthermore, time was lost performing the multitude of quantizations. A properly utilized custom hardware design would be able to quantize on-the-fly during runtime to mitigate this cost. Finally, native integration of the quantized layer execution into PyTorch would allow for direct experimentation rather than having to execute on CUTLASS separately and then partially model the final results.

The other works within this thesis explores addressing the limitations presented. Custom FPGA kernels are presented to efficiently execute the majority of the training workload in the desired custom precisions. That system can narrowly tailor its architecture to the desired precision, network and dataset combination. This provides the opportunity to yield greater performance and better utilization of benefits of reduced precision. Just as MuPPET focused on integrating into the popular ML training framework PyTorch, these custom accelerators need to also align with a machine learning framework so that they can and will be easily adopted by users. Moreover, native integration into PyTorch is presented alongside the custom FPGA hardware architecture. Finally, improved performance analysis and control over the quantizations available when using an FPGA are demonstrated to allow for additional criteria to be considered when selection precision. This elevates MuPPET to expand its acceleration potential using all information available to it. The next two chapters of this thesis will explore the requirements to enable such a hardware accelerator to exist.

Chapter 4

Caffe Barista

Brewing Caffe with FPGAs in the Training Loop

Chapter 3 addressed the rising complexity of CNNs in pursuit of ever more accurate networks and the detrimental effect this has on training times. CNN training is based on the backpropagation algorithm, consisting of two distinct phases: the forward and backward pass. In the forward pass, an input is passed through each layer of the CNN and the average loss across a batch of inputs is computed. In the backward pass, the computed loss is fed back through the network using the backpropagation algorithm which calculates the gradient w.r.t inputs and gradient w.r.t weights for each layer. The gradients w.r.t. weights are then utilized in updating the parameters of the model.

Out of all types of layers, the convolution layer dominates the computational load in both phases taking up to 90% [35, 36], when the training is executing on a CPU. As a result, most research on accelerating CNN inference and training focuses mainly on the convolution layer [94, 35, 36, 95, 64]. Towards the acceleration of the computation of the convolution layer, the strategies for inference and training differ in one key way. During inference, the sizes of the convolution kernel are limited to a specific subset of usually 1x1 and 3x3 convolutions, with occasionally 5x5 and 7x7 convolutions. For training on the other hand, the backward pass contains convolutions of any size, primarily due to the gradient with respect of weights computation. Therefore, if a single type of convolution is to be selected the convolution is usually expressed as a general matrix multiply (GEMM). Additionally, there is a large availability of high performance libraries and high-performance hardware for matrix multiplication making GEMMs a very attractive option for convolution. This is also why they are always supported by every existing ML training framework.

Currently, the most popular device for performing a general matrix multiply (GEMM) is a GPU. Although server-grade GPUs are well suited for training CNNs when convolutions are expressed as GEMMs, they consume significant power. As greenhouse gas emissions and energy efficiency becoming an ever growing problem, the world as a whole favors energy efficient technological solutions. Both of these factors fuel research into architectures and devices with better performance per watt [63, 73]. This fact has led industrial players to equip their servers with custom ASICs, such as Google TPU [9], Graphcore IPU [10] and Amazon Inferentia [96]. Nevertheless, the long development time and time-to-market together with their fixed functionality limit the ASICs' ability to exploit model-specific optimizations and support the latest fast-paced algorithmic advances.

To combat both the long development time as well as the high power usage, reconfigurable computing is considered a promising solution. Especially in the form of FPGAs, reconfigurable computing has attracted attention from the research community for addressing the above problem. FPGAs possess the ability to tune the system architecture to the targeted workload characteristics, promising better performance per watt ratios compared to a GPU based solution [73, 63, 42]. At the time of publishing, FPGA-based CNN training has only slightly been explored [62, 97, 98]. Currently, although the field has expanded, FPGAs are really shining in edge-based inference with most of the research focusing on that field. This is largely due to the lack of tools to easily prototype and deploy various hardware and/or algorithmic techniques for efficient CNN training.

This work this chapter presents in Caffe Barista. Barista provides an FPGA architecture for performing convolutions as a GEMM with Caffe integration. The hardware architecture leverages a systolic array based architecture to perform the GEMMs required for the convolutions.

Additionally Barista consists of an open-source automated toolflow[1] that provides seamless integration of FPGAs into the training of CNNs within the popular deep learning framework Caffe. Seamless integration enables the rapid prototyping and deployment of FPGA-based kernels for CNN training. Additionally, the work provides a memory-aware model for the execution of an FPGA-based general matrix multiply (GEMM) kernel along with an initial HLS implementation of this kernel. In this manner, *Barista* allows both hardware researchers and machine learning experts to explore novel hardware and algorithmic techniques respectively for power-efficient training. Unfortunately, since the creation of this work, Caffe has been bought by PyTorch. Although the interface to Caffe still exists, the ability to integrate with PyTorch would be far more beneficial and is explored in the next chapter.

4.1 System Design

This section describes the overview of all the components that Barista provides. First an overall overview of all the components of Barista are shown demonstrating how they interact. After this each individual component is described in detail.

4.1.1 Overall Design

The proposed toolflow, Barista, integrates with the Caffe framework and targets systems with PCIe-based FPGA accelerators. Barista assumes the target hardware is a heterogeneous platform with a server-grade FPGA accelerator and a host CPU. To this end, Barista consists of three components:

1. FPGA-based hardware accelerator with an accompanying performance model.
2. Software integration layer that enables the seamless integration of the FPGA accelerator with Caffe.

[1]https://github.com/ICIdsl/caffe_fpga.git

3. OpenCL runtime integration that orchestrates the CNN execution between a host CPU and the FPGA.

The first component is the hardware accelerator architecture itself. This architecture is a systolic array that performs matrix multiplications. Barista's hardware architecture aims to be a flexible architecture that can adapt depending on the workload that will be processed. To enable this flexibility, Vivado HLS was used as the design language for this architecture. To decide on the best size of the architecture, a performance model is created to accompany this design. This model will report the performance of various architecture configurations sizes for various workloads.

Once an architecture configuration is decided by the user, the design is compiled into a bitstream. Caffe handles the execution of the model, loading in data, setting up training parameters and executing training. At the convolution level Caffe calls the custom Barista functionality to execute the GEMM on the FPGA. Barista relies on an OpenCL setup to deploy and run the compiled bitstream as well as on- and offload the data from the FPGA. Upon deployment, the FPGA device runs the compiled bitstream with the proposed systolic array architecture performing matrix multiplications. These matrix multiplications are involved in the forward and backward pass of the CONV layers throughout CNN training. The CPU executes all other Caffe operations required for the training process.

As this is not an end-to-end toolflow, the user cannot provide a description of the network and get a functional training system in return. The user must understand how to utilize the performance model, configure and compile the architecture and run Caffe for training.

4.1.2 Caffe GEMM Execution Flow

This section provides a description of Caffe's native execution flow for CONV layers. Initially, Caffe selects which platform (CPU/GPU/FPGA) to execute on. Input data is provided in predetermined batchsizes which move through the network. For each convolution layer as a batch is provided to the layer, the Caffe framework batch-level GEMM function is called. As the convolution operations is executed as a GEMM, the inputs and weights must be transformed into a matrix.

At this point, the implementations for forward and backward pass start to deviate. The forward pass is considered the most basic operation. As described in Section 2.3.3, the forward pass convolution is the simplest. All that is required is the `im2col` procedure performed on all the inputs to convert them from N_{INP} 3 dimensional tensors to N_{INP} 2 dimensional matrices. Furthermore the weights are flattened along their inner 3 dimensions to also become 2 dimensional matrices.

For the backward pass, gradients w.r.t. the weight are calculated first. Following the computation of the gradient w.r.t weight, Caffe calculates the gradient w.r.t. to the input. There are specific details related to converting the data structures so that they can be processed as GEMMs, consisting of a set of dilations, rotations of the input and permutations of both the weight and input data. Once the dilations, rotations and permutations are dealt with, the input and weight data are treated exactly like in the forward pass. As Barista only focuses on replacing the GEMM part of the convolution. Further details of the im2col process, the rotations, dilations and permutations are left out for simplicity as they have no impact on the design of the architecture. The only relevant information is that they affect each convolution and are essential to performing a convolution as a GEMM. The specific details regarding this transformations can be found in the background chapter in Sec. 2.3.1.

Once all the data is converted into matrices, each provided matrix is split into tiles (Section 4.1.3) and then fed to the FPGA GEMM kernel (Section 4.1.4). In the Caffe framework at the level of calling the GEMM function, there are custom GEMM execution calls each optimized for a specific target hardware. Barista extends the native Caffe framework by adding FPGA tailored GEMM functionality. The FPGA functionality can be called with identical inputs to the GPU and CPU alternatives. Barista's FPGA architecture accepts incoming data in a post-processed manner to ensure seamless integration into the Caffe framework. This avoids any additional overhead unrelated to the convolution itself. By integrating at this level, all optimizations performed by Caffe do not interfere with the FPGA and can be fully exploited independently of the hardware implementation. The OpenCL runtime functionality (Section 4.1.4) takes care of all operations between being provided with matrix inputs and providing the convolved output as a return.

4.1.3 Accelerator Architecture for CNN Training

The developed hardware architecture is designed to be used for all CNN convolution training workloads, being able to adapt to any size encountered during the forward or backward pass. Therefore a blocked GEMM strategy is implemented to adapt to any size matrices. This blocked GEMM architecture is reused across both the forward and backward passes of the CONV layers. The core compute block comprises of a parameterized systolic array architecture for the execution of the blocked GEMM. The architecture is parameterized to scale based on the resource budget of the target platform. Compile time parameterization is applied to attempt to fit the expected size of the input matrices while keeping the design resource consumption within the target hardware limitations. The hardware restrictions refer to the number of processing elements and buffer sizes (Section 4.1.3). By adopting a blocked variant of GEMM with tunable tile sizes (Section 4.1.3), it is possible breakdown the convolution is the required hardware exceeds what the FPGA can provide.

Blocked GEMM

The primary operation that is to be accelerated under the proposed system for neural network training is matrix multiplication (i.e. $\mathbf{C} = \mathbf{AB}$). As shown in Fig. 4.1, in the operation ($\mathbf{C}$=$\mathbf{AB}$), matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are partitioned into tiles. Matrix $\mathbf{A}$ is partitioned into $\left\lceil \frac{R}{T_r} \right\rceil \cdot \left\lceil \frac{P}{T_p} \right\rceil$ tiles of size $T_r \times T_p$, matrix $\mathbf{B}$ is partitioned into $\left\lceil \frac{C}{T_c} \right\rceil \cdot \left\lceil \frac{P}{T_p} \right\rceil$ tiles of size $T_p \times T_c$ and the output matrix $\mathbf{C}$ is partitioned into $\left\lceil \frac{R}{T_r} \right\rceil \cdot \left\lceil \frac{C}{T_c} \right\rceil$ tiles of size $T_r \times T_c$. If $\frac{R}{T_r} \notin \mathbb{Z}$, then zeros are added to dimension R until $\frac{R}{T_r} \in \mathbb{Z}$. The same applies to both P and C. This process will be referred to as *Tiling* henceforth.

From an operational perspective, the accelerator computes one tile of the output matrix $\mathbf{C}$ at

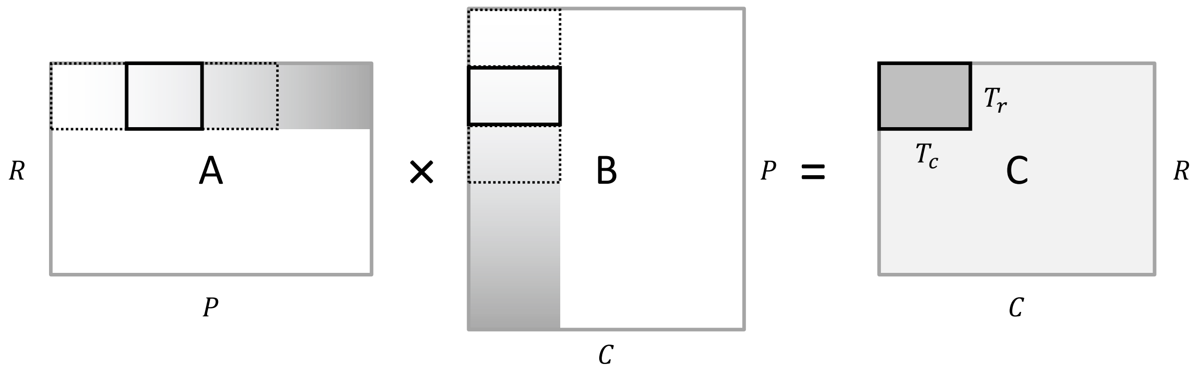


Figure 4.1: Overview of the adopted blocked GEMM strategy.

a time. For the computation of a single output (**C**) tile, $\left\lceil \frac{P}{T_p} \right\rceil$ tiles from matrix **A** and **B** are multiplied together. In the implemented tiling strategy, an output tile is cached in the on-chip memory of the accelerator until it has been fully formed. Consequently, the intermediate results of the tile are reused $\left\lceil \frac{P}{T_p} \right\rceil$ times before they are written back to the off-chip memory. This data reuse relaxes the bandwidth requirements of the accelerator. All outputs are stored on the off-chip memory until the convolution is complete and the result is returned to the CPU. Finally, the output tiles are then reassembled on the CPU until all output tiles are computed and the entire output has been constructed.

Hardware Architecture

Fig. 4.2 shows the adopted hardware architecture for accelerating the GEMM algorithm. As the tiling happens outside the FPGA, the architecture on the FPGA has no need to consider which tile/block is being processed. The core of the design is a mesh of processing elements (PEs) together which form the systolic array. Each PE has a initiation interval equal to one accepting a new input and weight value each cycle as long as the necessary data is available in buffer A and B. The dimensions of the mesh are compile-time configurable with a total of $T_r \times T_c$ PEs.

Beside the compute infrastructure is the memory infrastructure. Two on-chip memory arrays are implemented as memory buffers for matrices **A** and **B**. As soon as the execution of the kernel is triggered, the input tiles $T_r \times T_p$ and $T_p \times T_c$ are burst-read from the off-chip memory

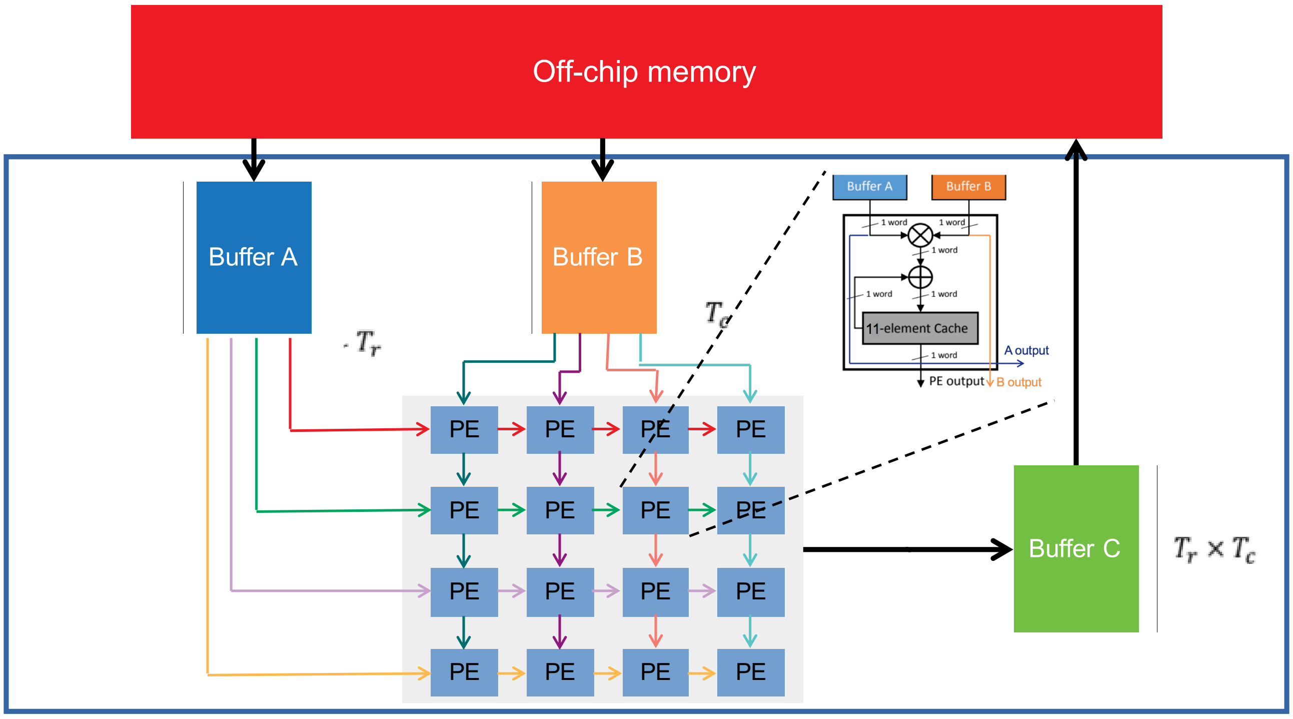


Figure 4.2: Diagram of the systolic-array design.

into buffers A and B. These on-chip memories are the size of a respective tile; $T_r \times T_p$ and $T_p \times T_c$ for **A** and **B** respectively. Both of these memories are double buffered to allow for reading data from off-chip while performing a matrix multiplication.

After two tiles have been multiplied together the result is moved to Buffer C. As mentioned, $\left\lceil \frac{P}{T_p} \right\rceil$ tiles need to be computed and accumulated before the related output tile is complete. Therefore the accumulation of the intermediate results remains in Buffer C. After all $\left\lceil \frac{P}{T_p} \right\rceil$ tiles have been computed and accumulated, the tile in Buffer C is written to off-chip memory. All tiles for matrix **C** are stored on off-chip memory until the convolution is complete.

All of the inputs to the GEMM are preprocessed by the CPU into a tiled layout that is sequentially stored in memory. As will be demonstrated in the evaluation (Sec. 4.3), this imposes a very significant overhead. When the FPGA is large enough to hold a significant portion of the GEMM, the overhead decreases, but as demonstrated in Sec. 4.3 it is still significant.

When providing the tiled data to the FPGA, the data must be ordered properly. The fastest and most efficient way for an FPGA to read off-chip memory is through a process called burst-reading, which requires sequential data access. Consequently the data must be pre-

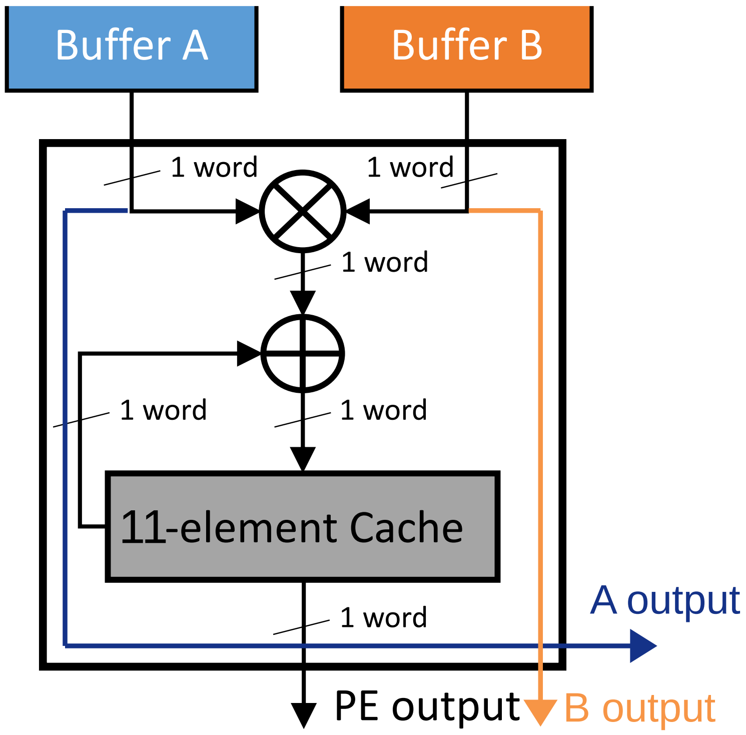


Figure 4.3: Processing element overview.

ordered to allow for burst-reading of the input matrices' tiles into memory buffers A and B. Furthermore, data is passed form the host CPU to the FPGA off-chip memory through a PCIe port. To minimize the overhead of transferring data through PCIe and to maximize the PCIe bandwidth, the two matrices to be multiplied are sent from the host CPU machine to the off-chip memory on the FPGA board in one transaction. Server grade FPGAs tend to have large off-chip memories that can support this amount of memory quite easily. For smaller FPGAs, multiple calls through the PCIe process need to be made to ensure all the data is accessible by the FPGA. As the process is tiled already, the only cost here is the data transfer overhead.

Processing Element

Fig. 4.3 shows the internal design of each processing element (PE). Each PE is responsible for computing one element of a tile of the output matrix. From a hardware perspective, each PE contains a single multiply-accumulate unit and a local cache for storing the intermediate results of the output. This cache holds the MAC result until the final result is ready and written out to external memory.

Inputs are fed into each PE from the two input buffers in a pipelined manner. Data comes either directly from the memory buffer or through a neighboring PE depending on its location in the processor as shown in Fig. 4.2. Intermediate results due to interleaving are cached within the output cache of the PE. They are only written out to the external memory once the final result is ready. As can be seen from the blue and orange lines (Fig. 4.3), the original value from buffer A is passed on to the PE on the right and the original value from buffer B is passed on to the PE below. The PE functions as both a store for the intermediate calculations as well as a mechanism of passing the values from buffer A and B to the subsequent PEs. This dataflow depicted in Fig. 4.2 enabled by the PEs variety of functionality enables efficient data passing between PEs in a pipelined fashion, saving routing resources and improving the scalability potential of the design.

To provide maximum performance, each PE must have an initiation interval (II) of 1. In other words this mean that each PE must be able to accept a new set of values every cycle. In this case that means taking one weight and input value every cycle. Additionally because of the pipelined structure, the PE must pass on one weight and input value each cycle as well to ensure a constant flow of data.

The use of a systolic array itself is not unique for performing GEMMs on an FPGA, and what is presented here is referred to as a semi-systolic array. Usually, all the from Buffer A would be provided to every element in a row, rather than letting it propagate through. The same applies to the data from Buffer B being provided to every element in a column. Connecting the memory to every PE imposes a large routing burden as the fabric needs to be able to pass the data to every PE. At a small scale this is not very impactful, but if the design is scaled, this quickly becomes an issue. Large routing tracks can limit the clock speed, reduce resource efficiency and potentially make the design unroutable if too many PEs are required. By propagating the values, all the routing pressure is removed as connecting neighboring PEs is a simple connection that does not change regardless of the number of PEs desired.

Precision-aware interleaving

As mentioned each PE performs a multiply-accumulate with a desired II = 1. This means that every cycle, the PE must accept an input and a weight value, perform a multiplication and then add the result to a locally stored intermediate sum. Depending on the adopted precision and target device, the latency of a multiplier and the adder are P and S cycles respectively.

Any sequence of independent multiplications, like the ones used in the proposed architecture, can be pipelined at the cost of more hardware. Accumulations however, cannot inherently be pipelined. This is because the result of the previous accumulation needs to complete before the next accumulation can start as its being directly utilized for next accumulation. Therefore, in a naive implementation, each accumulation will be S cycles apart. In the proposed architecture, a new value arrives every cycle to comply with the $(II) = 1$. Having to wait S cycles between accumulation would violate this requirement if $S > 1$. To alleviate this, when $S > 1$, we employ an interleaving technique. This technique stores $S + 1$ independent intermediate results in the PE's cache as shown in Fig. 4.3. Each entry in the cache stores an intermediate accumulated value, that is accumulated upon every $S + 1$ cycles. After $S + 1$ products have been computed, the cache will be loaded with values. Every cycle afterwards, a product is accumulated in one slot of the cache, for example slot 1. The next cycle, slot 1 is still busy accumulating, so the product accumulates in slot 2. This repeats itself going along each of the cache slots. After S cycles, slot 1 will be done accumulating, and the product computed in cycle $S + 1$ can be assigned to slot 1 again. This design allows the entire PE to operate in a pipelined fashion at a throughput of 1 without being forced to wait on any single accumulation to complete. It does require a final accumulation of all the intermediate values in the PE cache to produce the final result.

In an ideal case scenario a floating point accumulation will take 4 cycles. This would mean that the size of the cache in the PE would be 5 elements. Yet Fig. 4.3 has an 11 element cache instead. This is due to the fact the proposed design is implemented in Vivado HLS (see Sec. 4.1.1. This HLS compiler has inherent limitations which setups up an accumulation to be 10 cycles, forcing the intermediate value cache to be 11 elements.

4.1.4 OpenCL Setup

The CPU-FPGA interactions and the FPGA execution are orchestrated by *Barista*'s OpenCL module. As Caffe executes the network, when a convolution is encountered initially basic data processing occurs. For the forward pass this is minimal, but for the backward pass various permutations and rotations are performed. Once this is complete, depending on the target hardware, the im2col process is performed on either the CPU (if the target is CPU or FPGA) or the GPU (if the target is a GPU). After this, the data is provided to a GEMM unit. This unit decides how the GEMM is performed, taking a seperate path for each type of target hardware. For this work, a new path through an FPGA focused module is introduced, handling the execution of the GEMM using the proposed architecture. Prior to performing the GEMM operation, this module is responsible for configuring the GEMM unit for the next matrix multiplication. This requires computing tiling size, accounting for all memory and data ordering operations and allocating the memory on the FPGA off-chip memory. Furthermore, this module allocates the necessary memory for the matrices as well as transforming and tiling the input matrices given the selected tile sizes. All allocated memory must be properly aligned in the FPGA off-chip memory, ensuring the tiles are all stored as a float `aligned_allocator` vector. Furthermore, beyond just transforming and tiling the input matrices, this module also untiles the produced output from the GEMM. Untiling the data makes sure that the output is recreated correctly to resemble the desired output configuration.

For the GEMM operation itself, the OpenCL module coordinates all CPU-FPGA data transfers across the PCIe channel and launches the GEMM execution. All operations outside of the GEMM itself presented in this module are executed on the host CPU.

4.2 Performance Model

To select the dimensions of the mesh that would yield the lowest latency for the overall convolution, a performance model was built which estimates the attainable execution time of the hardware design. The performance model consists of two components:

1. The estimated execution time for the processing of the entire matrix multiplication using the proposed architecture (Eq. (4.6))

2. The estimated memory transfer time for transferring the matrices **A**, **B** and **C** between the host and the FPGA's off-chip memory.

Additionally a resource usage model for both on-chip memory and DSP utilization is included. This ensures the desired design is able to fit on the target hardware. Other resource constraints such as LUTs and flip flops and were not considered as they were less limiting to the possible designs and therefore of no interest. The resource model is used to set hard limits on the maximum values of T_r, T_c, and T_p.

4.2.1 Off-chip memory transfer time

To compute the off-chip memory transfer time, first the total amount of required data must be known. The design requires $T_r + T_c$ inputs per cycle to keep the systolic array operating at a throughput of 1. For each tile in matrix **A** and **B**, there are P cycles along the common dimension. This produces a $T_r + T_c$ sized tile for matrix **C**. $\text{Data}_{\text{tile}}$ represents the total memory used for a single tile:

$$\text{Data}_{\text{tile}} = WL \cdot ((T_r + T_c) \cdot P + T_c \cdot T_r) \tag{4.1}$$

where WL denotes the wordlength (bits) of an element (*i.e.* 32 for a single-precision float variable).

There are a total of $\left\lceil \frac{R}{T_r} \right\rceil$ tiles of matrix **A** and $\left\lceil \frac{C}{T_c} \right\rceil$ tiles of matrix **B**. Each tile of matrix **A** must be multiplied with each tile of **B** to produce the final output. Data_{mem} represents the data that needs to be accessed from off-chip memory:

$$\text{Data}_{\text{mem}} = WL \cdot \underbrace{\left\lceil \frac{R}{T_r} \right\rceil \cdot \left\lceil \frac{C}{T_c} \right\rceil}_{\text{All tiles}} \cdot \left(\underbrace{(T_r + T_c) \cdot P}_{\text{input data}} + \underbrace{T_c \cdot T_r}_{\text{output data}} \right) \tag{4.2}$$

Overall, given a memory bandwidth of B_{mem}, the minimum latency for accessing the off-chip memory per matrix multiplication is:

$$\text{Latency}_{\text{mem}} = \frac{\text{Data}_{\text{mem}}}{B_{\text{mem}}} \tag{4.3}$$

4.2.2 Compute time

As mentioned in The design of the architecture 4.1.3 each PE is designed to have an II = 1. As a result the number of compute clock cycles is equal to the number of operations performed by the PEs.

Due to the specific systolic array design data goes through the SA in a pipelined fashion. Therefore in the first cycle, only the first column and first row have any data, and only the first PE of the SA has enough data to be executed. It is only after $T_r + T_c - 2$ cycles that the last PE will receive its first value. This will be referred to as “flush” time. In the matrix multiplication of the tiles, there are T_p values along the common dimension. This is repeated for all blocks of T_p. Finally, the cost of the interleaving must be considered requiring $(S+1)^2$ cycles per tile calculation. Therefore the nubmer of cycles required to complete a single tile are:

$$\text{Cycles}_{\text{tile}} = \left\lceil \frac{P}{T_p} \right\rceil \cdot (T_p + T_c + T_r - 2) + (S+1)^2 \tag{4.4}$$

Finally this is repeated for all remaining tile combinations providing the number of clock cycles that is needed for the developed system to process the matrix-multiplication computation:

$$\text{Cycles}_{\text{compute}} = \underbrace{\left\lceil \frac{R}{T_r} \right\rceil \cdot \left\lceil \frac{C}{T_c} \right\rceil}_{\text{All tiles}} \cdot \underbrace{\left(\left\lceil \frac{P}{T_p} \right\rceil \cdot (T_p + T_c + T_r - 2) + (S+1)^2 \right)}_{\text{Cycles}_{\text{tile}}} \tag{4.5}$$

4.2.3 IP execution time

The total GEMM kernel execution latency, when the data is already available in the off-chip memory is:

$$\text{Latency}_{\text{Compute}} = \max\left(\frac{\text{Cycles}_{\text{compute}}}{f_{\text{clk}}}, \text{Latency}_{\text{mem}}\right) \tag{4.6}$$

where f_{clk} denotes the clock frequency of the FPGA device. This equation only accounts for the time spent performing the GEMM calculation, any CPU computation is not considered.

4.2.4 PCIe transfer time

The PCIe transfer time captures the latency for the communication of the data from the CPU to off-chip memory. $\text{Data}_{\text{PCIe}}$ captures the total amount of data transferred for matrices **A** and **B** in and **C** out:

$$\text{Data}_{\text{PCIe}} = WL \cdot \left(\underbrace{R \cdot P}_{\text{Matrix } \mathbf{A}} + \underbrace{C \cdot P}_{\text{Matrix } \mathbf{B}} + \underbrace{R \cdot C}_{\text{Matrix } \mathbf{C}}\right) \tag{4.7}$$

Eq. 4.8 captures the transfer latency given the PCIe bandwidth B_{PCIe}:

$$\text{Latency}_{\text{PCIe}} = \frac{\text{Data}_{\text{PCIe}}}{B_{\text{PCIe}}} \tag{4.8}$$

The overall latency is a combination of moving data to and from the FPGA (Eq. 4.8) and executing the GEMM (Eq. 4.6): The reason these can simply be added together is due to the OpenCL implementation used for this scenario. There are three steps to the OpenCL process. First, the process starts by moving data onto the FPGA off-chip memory. Secondly, once all the data is loaded, the compute process can be started. Finally, after the compute is complete, the FPGA off-chip memory to CPU host memory transfer can be initiated. The first and third steps make up $\text{Data}_{\text{PCIe}}$ and the second step makes up $\text{Latency}_{\text{PCIe}}$. Therefore the overall latency is as follows:

$$\text{Overall latency} = \text{Latency}_{\text{PCIe}} + \text{Latency}_{\text{Compute}} \tag{4.9}$$

4.2.5 Resource Model

A model for estimating resource usage as a function of the configurable parameters T_r, T_c and T_p was developed. Eq. (4.10) and (4.11) model resource usage.

$$\text{Util}_{\text{DSP}} = \underbrace{(T_r \cdot T_c)}_{\#\text{ of PEs}} \cdot \underbrace{V}_{\frac{\text{DSPs}}{\text{PE}}} \tag{4.10}$$

$$\text{Util}_{\text{BRAM}} = WL \cdot \left(\underbrace{T_r \cdot T_p}_{\text{buffer } \mathbf{A}} + \underbrace{T_p \cdot T_c}_{\text{buffer } \mathbf{B}} + \underbrace{T_r \cdot T_c \cdot (S+1)}_{\text{buffer } \mathbf{C}} \right) \tag{4.11}$$

In Eq. 4.10, the estimated number of DSPs includes the DSPs that are consumed for the multiply-accumulate (MAC) unit of each PE, multiplied by the number of PEs in the processor. In Eq. 4.11, each term represents the size of the corresponding on-chip buffer memeory utilization. The multiplicative factor of $S + 1$ for buffer C is due to the interleaving factor. All accumulation of the intermediate interleaving values is performed synchronously with the systolic array outside of the processing elements, requiring extra memory. WL is the wordlength which represents the number of bits that are used for data representation. For floating point operations $V = 9$ which was found empirically. Note that the interleaving cache is not included in the on-chip memory allocation. This is due to the fact they they are very small local memories that are constructed as LUTRAM to reduce the routing pressure of the design.

4.2.6 Overall Model

Thusfar, the performance model computes the latency based on the size of the incoming GEMM, the tiling size accepted by the FPGA and the PCIe bandwidth. The overall goal is to minimize the latency modeled by the performance model. The tiling size accepted by the FPGA based on the constraints of the FPGA itself, which will vary depending on what kind of FPGA is used. Overall, the highest performing configuration of the architecture corresponds to the (T_r, T_c, T_p) combination that maximizes the performance (Eq. 4.6) while complying with the platform-specific resource constraints (Eq. 4.10 and Eq. 4.11). This looks as follows where FPGA_{DSP}

and $\text{FPGA}_{\text{BRAM}}$ refer to the FPGA total available DSPs and BRAM blocks respectively:

$$\arg\min_{T_r,T_c,T_p} \left(\text{Latency}_{\text{PCIe}} + \text{Latency}_{\text{Compute}}\right) \tag{4.12}$$

$$s.t.\ \text{FPGA}_{\text{DSP}} \geq \text{Util}_{\text{DSP}}, \text{FPGA}_{\text{BRAM}} \geq \text{Util}_{\text{BRAM}} \tag{4.13}$$

In the end Eq. 4.13 returns the tiling configuration that provides the lowest latency while still fitting on the target FPGA. The overall model only accounts for the GEMM execution itself, not any of the additional data manipulation performed on the CPU to prepare the data for execution. The reason this was not modeled is because the amount of work taken to perform the tiling is roughly equal regardless of the size of the tiling that occurs, as all the elements are accessed once per tiling round. The effect of the tiling is considered in any performance analyses as demonstrated in Sec. 4.3.1 and Sec. 4.3.2.

4.3 Evaluation

The performance of the tool was evaluated on the Xilinx Virtex UltraScale+ XCVU9P FPGA hosted on the F1 instances on Amazon Web Services (AWS) [99]. This device has 2586k logic cells, 6840 DSPs and 75.9Mb BRAM. When operating in FP32, $S = 10$ cycles and $V = 9$ DSPs are used per FP32 MAC unit. For any fixed point operations, $S = 1$ and $V = 3$. This was investigated as the move to fixed point from floating point is quite simple. It only requires changing the declared data types and removing the interleaving setup. The reason interleaving is no longer required is due to the fact that fixed point values accumulate in one cycle. In MuPPET (Chapter 3) we demonstrated the effect that reduced precision fixed point can have on training. Following on from this, we demonstrate how the architecture for Barista can support floating and fixed point computation and the performance of both design choices. The fixed point design discussed here is not fully optimized for various fixed point precisions and does not exploit potential benefits from PCIe bandwidth optimization, LUTRAM storage or LUT computation. As a result, the value for S and V stay the same for 8, 12, 14 and 16-bit fixed point values. Therefore, 8-bit fixed point is used as a representative for all fixed point computations.

The proposed FP32 GEMM accelerator (Section 4.1.3) is developed using Vivado HLS and synthesized using SDAccel 2018.2. The design is clocked at 250 MHz and the accelerator is configured with $\langle T_r, T_c, T_p \rangle$ set to $\langle 16, 16, 64 \rangle$ using the performance model presented in Section 4.2. The size selected was a size that compiled with a low overall execution latency considering every convolution seen during the training of the ResNet20 architecture on the CIFAR10 dataset. This was the highest performing design that would route with the current HLS implementation within reasonable time limits. Optimization of the verilog produced by the HLS would porbably allow the largers design sizes to be built, but was considered outside of the scope fo this work. The selected sizes of $T_r = 16$, $T_c = 16$ and $T_p = 64$ used 18.8%, 10.8%, 8.8% and 14.1% of the available DSPs, LUTs, FFs and BRAM respectively. The accuracy of the GEMM results of this design were verified locally on a Xilinx Alveo U250 FPGA. Two widely used CNNs, AlexNet [3] and ResNet20 [4] were trained on the CIFAR10 dataset and used as the reference networks for analysis. *Barista* was compared with the CPU and GPU implementation on Caffe [100].

4.3.1 Single Configuration

Figure 4.4 shows the average performance-per-watt (PPW) measured in GOPS/watt across all CONV layers during the training process (i.e. forward and backward passes) of ResNet20 using *Barista* for the FPGA and CPU for various tile sizes. The FPGA design that was built and implemented was an FP32 configuration. For the GPU, the average PPW across all Resnet20 CONV layers was 1.54. AWS power profiling showed that the FPGA used 80W of power when running these designs, compared to the CPU's (Intel Xeon E5-2686v4@2.3GHz) 145W and GPU's (NVIDIA GTX 1080Ti) 279W. All profiling across each type of target hardware was performed using Caffe's internal timers. The accuracy of the Barista design was verified by comparing the FPGA output to the CPU output.

Fig. 4.4 demonstrates that for all sizes of kernel larger than $\langle 8, 8, 32 \rangle$, both the FP32 (blue bars) and INT8 (orange bars) model predictions outperform the CPU (green bars). For architecture

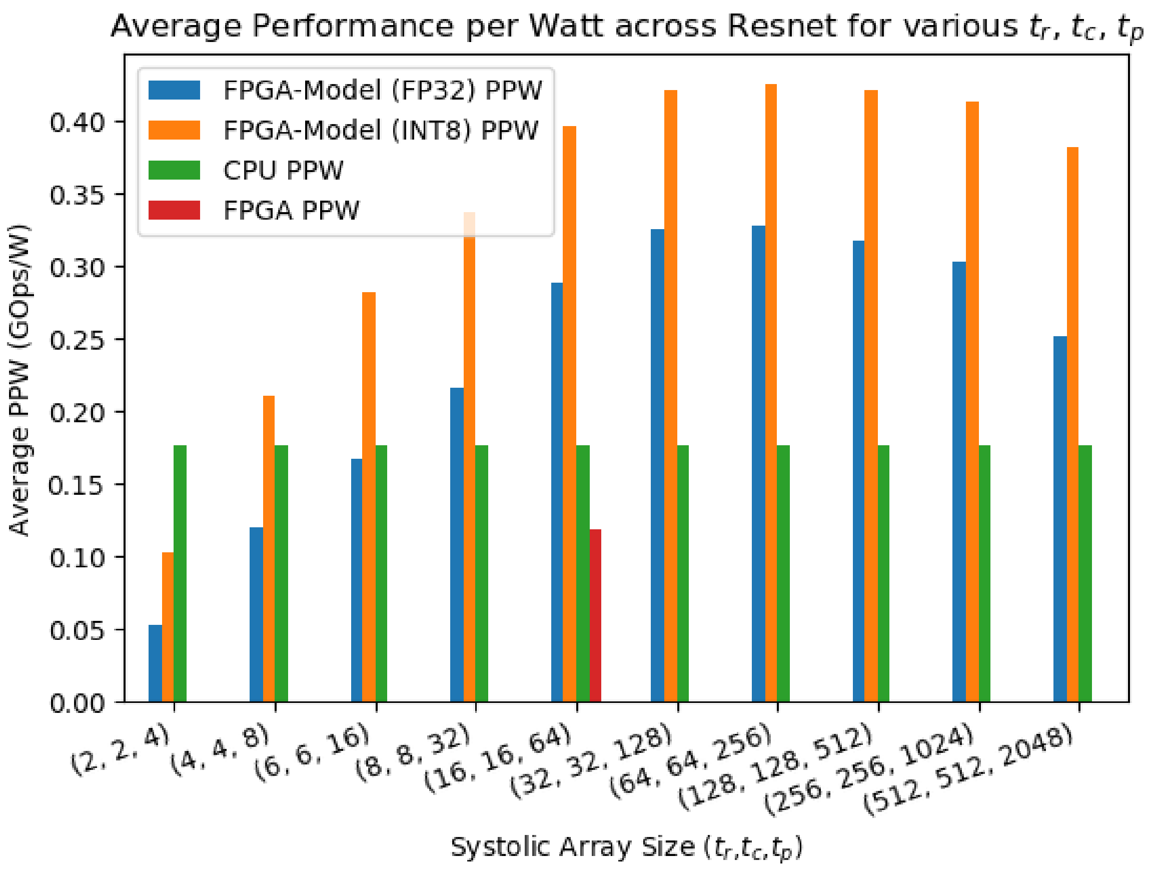


Figure 4.4: Average PPW across ResNet20 for various $\langle T_r, T_c, T_p \rangle$.

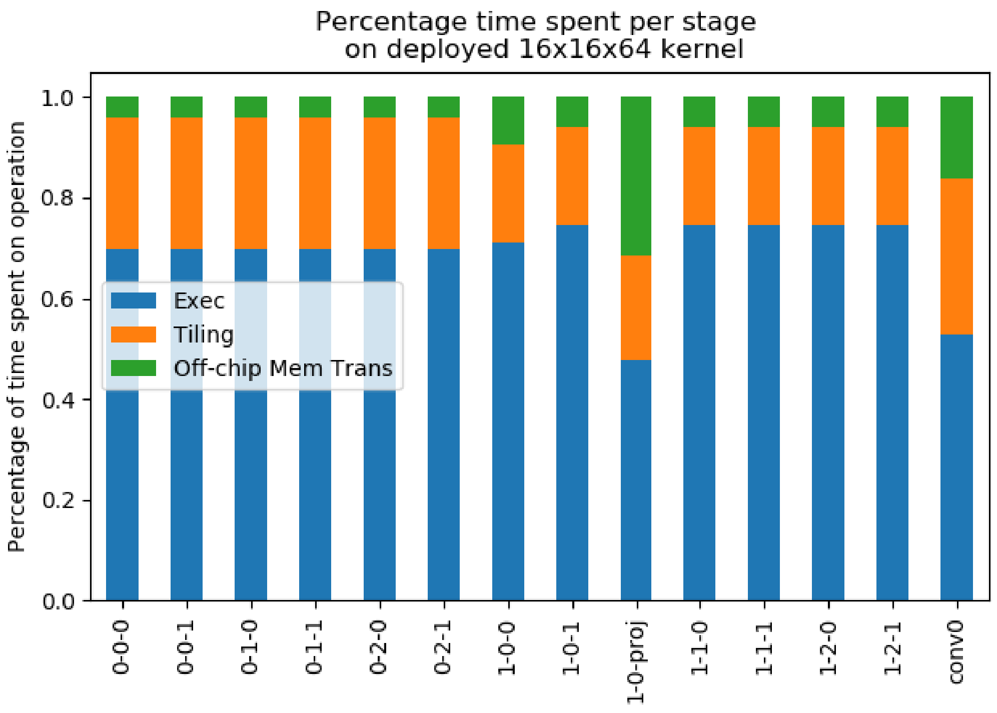
configurations larger than $\langle 64, 64, 256\rangle$ the performance per watt degrades. This is due to the fact that during training there are a large range of convolution sizes. For large tile sizes, if the GEMM associated with a convolution consists of a large number of small matrix multiplications, the area of the tile not covered is filled with zero ops. The GEMM for a convolution therefore will not utilize all the allocated resources when the tile size starts to significantly exceed the sizes of the input matrices. Performing a large number of zero ops due to tiling (Section 4.1) leads to an increased inefficiency, leading to a lower performance efficiency.

4.3.2 Model Analysis

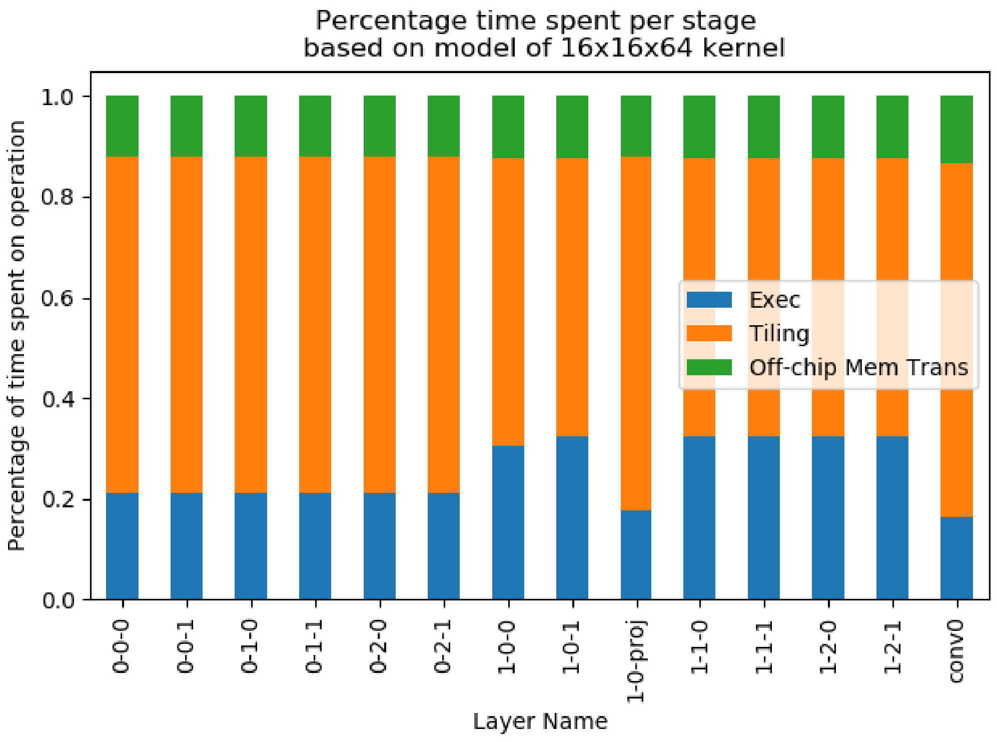
When looking at the compiled implementation in Fig. 4.4 of a $\langle 16, 16, 64\rangle$ kernel (red bar) it does not outperform the CPU (green bars). This discrepancy between expected and achieved performance demonstrates a large inaccuracy in the performance model. To identify the reasons behind the difference in expected and achieved performance, further profiling was performed using OpenCL.

Fig. 4.5a breaks down the relative time spent on each stage of GEMM execution for the implemented compiled bitstream using profiled data. The kernel execution (blue), which includes off-chip to FPGA memory transfers, is seen to be the biggest bottleneck at the moment taking more than 50% of the time in all CONV layers. Kernel execution (blue) profiled using the Xilinx Vitis profiler showed that memory bandwidth utilization for kernel to off-chip memory transfers was in the range of about 10%. This indicates a large amount of the potential off-chip memory transfer bandwidth was not being used. Nevertheless, compute unit utilization rates are at least 70% indicating the system can be further improved by exploiting memory optimizations. Reducing the DDR4 bandwidth assumption to 3Gbps (10%) in the model predicts a performance close to that achieved by the implemented kernel, supporting the bottleneck analysis from Xilinx's Vitis tool.

Fig. 4.5b shows the same breakdown assuming a design where the memory bandwidth is being fully utilized at 30Gbps. To achieve this, the data read in and read out components need to operate at a speed of 500MHz. But now using data from the model for estimates of kernel



(a) Profiled Implementation



(b) Model of Implementation

Figure 4.5: Relative time spent on each stage for various ResNet20 layers. Layer names have format (group-residual block-conv).

CONV Layer	conv1	conv2	conv3	conv4	conv5
$\langle \mathbf{T_r}, \mathbf{T_c}, \mathbf{T_p} \rangle$	$\langle 32, 32, 74 \rangle$	$\langle 32, 32, 64 \rangle$	$\langle 36, 36, 64 \rangle$	$\langle 32, 32, 64 \rangle$	$\langle 32, 32, 64 \rangle$
FPGA PPW (GOPS/watt)	0.59	0.29	0.078	0.076	0.073
CPU PPW (GOPS/watt)	0.35	0.24	0.089	0.13	0.11
GPU PPW (GOPS/watt)	0.13	0.58	0.43	0.50	0.28

Table 4.1: AlexNet predicted best FPGA, CPU and GPU PPW (GOPS/watt)

execution time (blue) and host to off-chip memory transfer time (green). Profiled time was used for tiling (orange), which is performed on the CPU. The model assumes full utilization of the DDR4 bandwidth (30Gbps) between off-chip memory and the kernel. Fig. 4.5b demonstrates that with full bandwidth utilization, the bottleneck is shifted from the FPGA kernel execution (blue) to tiling on the CPU (orange).

The size of the BRAM memory utilization is presented as a total number bits. What this fails to recognize is that FPGA memory is broken into memory blocks, the size of which can vary between FPGAs. Depending on how the memory is structured and partitioned, certain parts of the data cannot be in the same memory block. This results in only partially filled memory blocks, making the effective memory footprint larger than just the total number of bits of data. Failing to account for this makes it possible for the design to assume a design size will fit on an FPGA as the total number of bits is less than the capacity, but in reality will not fit when attempting to compile.

4.3.3 Multiple Bitstreams

A further experiment on tailoring the kernel architecture to each convolution workload seen throughout training was conducted using the presented models. Each convolution layer is provided with an individual kernel that provides optimal performance within the allocated resource constraints of the XCVU9P FPGA. A grid-search was performed across various values of T_r,T_c and T_p for designs that are expected to fit on the chosen board based on the resource model. For ResNet20, the kernel which is predicted to have the highest PPW on average across all layers of the network is $\langle 36, 36, 72 \rangle$ with a performance of 0.33 GOp/s/W compared to the

CPU's 0.18 (+83%). Layer-wise tuning showed that although different-sized kernel performed better on different layers, there is no overall difference in achieved PPW compared to using a single $\langle 36, 36, 72\rangle$ kernel for all layers. For AlexNet, however, this exploration showed that tailoring the kernel to the layer can provide overall PPW benefits. Table 4.1 describes the performance of the best kernels per layer and shows that for some layers a CPU performs better than an FPGA for FP32 computations. By selectively performing FPGA-based GEMM for conv1 and 2, otherwise using the CPU, the overall achieved PPW is 0.24 compared to the CPUs 0.18 (+33%) and 0.22 (+10%) achieved if all layers use one $\langle 32, 32, 64\rangle$ kernel.

4.4 Conclusion and Limitations

Caffe *Barista* enables hardware designers to rapidly prototype novel custom accelerators by seamlessly replacing the provided kernel with one that implements the same interface. The model suggests that up to 83% higher PPW compared to a CPU can be achieved for a lower absolute power consumption using custom precision arithmetic and/or increasing memory bandwidth utilization through batching and on-chip tiling. From the perspective of a deep learning researcher, *Barista* allows running any combination of optimizers (*e.g.* SGD, RMSProp, AdaGrad), learning rate schedules and a variety of other training-related parameters or algorithms that are natively supported by or can be implemented in Caffe. At the time of publication, to the best of our knowledge, *Barista* was the first open-source tool that allows for such versatile and rapid deployment of hardware and algorithms related to the training of CNNs on FPGAs.

4.4.1 Limitations

This work presented an initial approach for accelerating convolutions, but was unable to fully capitalize on the benefit of the architecture. As described, this is largely due to the inefficiencies in preparing the data on CPU, such as the tiling, untiling and im2col processes. Performing the various transformations, the im2col procedure and the tiling are computationally costly and are tasks that can benefit from parallelization. Additionally, the memory structure used in this design required the convolution to be broken into pieces and assembled afterwards. The im2col procedure increases the total amount of memory required to perform the convolution exclusively through data duplication. Beyond im2col, other preparation procedures will cause data duplication as well. If all of these operations were performed on-the-fly on the FPGA, the memory footprint as well as CPU computation burden would drop leading to improved performance.

Furthermore, at the time of publication Caffe was one of the popular frameworks, but overtime PyTorch and TensorFlow have taken over as the leading frameworks. Native integration into these frameworks would allow for a greater uptake by the research and development communities. Finally, this is not an end-to-end tool. This makes it hard for users who have no FPGA experience to use this tool and get started on FPGAs. Knowledge of the tools as well as the training problem are required to utilize the model and configure and compile the FPGA bitstream.

The memory and performance models are another limitations of this work. To create a full end-to-end system that can provide the best performing setup, highly accurate models are required to accurately reflect the architecture. Although the weakness of the models has been evaluated, these models will not be usable to accurately predict performance or resource utilization. This applies to the performance model so a reliable performance prediction can be provided, as well as a reliable resource model to ensure the proposed design will actually compile. This enables the ability to perform a complete design space exploration where the space of possible configurations can be explored and the best architecture configuration can be proposed.

The next chapter works on improving most of these limitations by unburdening the CPU. This includes onboarding the im2col, permutation and rotations processes as well as improving memory structure and efficiency. Furthermore it addresses the fact Barista is not easy to use for those not experienced with FPGAs by providing a full end-to-end ecosystem. This ecosystem accepts a network description and using highly accurate models produces compiled bitstreams that natively integrated into the PyTorch ML framework without any requirement of the user to understand the proposed architecture or have a deep understanding of FPGAs.

Chapter 5

FPGPT: FPGA-enabled PyTorch Training

Acceleration of DNN Training through a Custom Memory System

In Chapter 3 MuPPET demonstrated the benefit of utilizing reduced precisions throughout the training process of CNNs. One of the stated limitations was that the hardware utilized to perform the quantized training was not tailored for quantized training. When operating at precisions between 8 and 16bit fixed point (FXP), 16bit hardware is utilized. This limits the implementable potential acceleration that MuPPET can provide. Furthermore, there was no native integration of the reduced precision hardware into PyTorch, limiting the realistic usability of MuPPET.

Such limitations led to the inspiration for the content of this chapter and its accompanying toolflow FPGPT. FPGPT provides a design that allows for flexible creation of hardware that can execute a convolution at any precision for both the forward and backward pass. Whether the goal is optimized hardware for maximal performance, reusable convolution kernels to simplify execution or a middle ground between the two, FPGPT can adapt to whatever is required. Along with the hardware comes a highly accurate model to allow for analysis and insight into performance benefits. This informs the user of the performance difference between the various approaches would provide the user. Finally to maximize accessibility and usability, FPGPT integrated itself natively into one of the most popular machine learning frameworks; PyTorch. This integration allows for easy adoption by the machine learning community to allow anyone to accelerate CNN training.

Similarly to the work presented in Chapter 4, the work in this chapter aims to accelerate the training process of a CNN by targeting an FPGA-based system. As mentioned in the limitations sections of the previous chapter, there were clear shortcomings and bottlenecks which limited the efficacy of the hardware design. FPGPT addresses some of these limitations to provide a competitive hardware design with a complete ecosystem. The base idea of having a semi-systolic array persists, but FPGPT onboards the expensive data manipulation that Barista performs on the CPU, all tiling is removed and the amount of host-to-FPGA data movement is notably reduced. Furthermore, the resource and performance models for FPGPT provide greater detail and are significantly more accurate, never overestimating performance. Additionally, FPGPT presents an ecosystem that consists of novel architecture that adapts to the CNN's topology with a design space exploration tool that adapts the architecture to the expected CNN workload and the device characteristics which at the same time easily integrates into modern ML frameworks . Native integration with an ML framework ensures it will benefit from current and future improvements on those frameworks.

The requirement for seamless integration with existing ML frameworks sets constraints on how data become available to the FPGA accelerator as well as what functionality of the training can be offloaded into the FPGA. Additionally, there is a requirement for high precision modeling of the FPGA architecture to allow for automated design space exploration for configuring the architecture to the expected workload. These requirements were set to allow for easy adoption of FPGA training by non-experts. By automating exploration of possible designs, automatic generation of the FPGA bitstream and the native integration of the FPGA bitstream into an ML training framework means the user has to put in minimal effort and minimal FPGA knowledge to train CNNs on FPGAs. Easy uptake and utilization of FPGAs for training CNNs is difficult to come by deterring potential researchers, an issue FPGPT directly addresses. Both are addressed by the work presented in this chapter. The primary contributions of this work are as follows:

1. A tightly coupled memory subsystem: a compute unit integration maximizing system performance for a specific workload
2. A validated performance and resource model with accompanying DSE for identifying the highest performance architecture configuration for a given CNN and FPGA device.
3. A seamless integration with the PyTorch CNN training framework [75], allowing the users to utilize the available features of this ML framework.

5.1 Architecture Motivation

Within a CNN, the characteristics and potential parallelism in the computation of the convolution layers changes widely. Currently processors such as GPUs, TPUs and IPUs have a fixed degree of hardware parallelism regardless of the incoming workload's computational and memory requirement [9, 10]. Consequently, software techniques are employed to achieve more efficient utilization of the target hardware. An FPGA on the other hand can be reconfigured to adapt to the incoming workload enabling efficient hardware utilization. Such flexible adaptation is associated with a corresponding reconfiguration cost. To be able to utilize the flexible adaptation of FPGAs, the design of the proposed hardware kernel is highly parameterized. This allows it to most effectively adapt to the model requirements as well as user choices such as target hardware and the performance to compilation time tradeoff. The aim of adaptation can vary from optimally performing for the incoming workload to allowing multiple convolutions to run on the same kernel at a potential performance penalty or any point in-between.

Adapting to an incoming workload through custom tailored FPGA hardware kernels allows for improved computational efficiency. However, the performance of training CNNs on FPGAs is often limited by the available memory bandwidth. To combat this, [62, 67, 64, 66, 65] impose constraints on the possible target networks, runtime precision and datasets. For example, some works such as [62, 67, 66, 65, 1] only train on CIFAR-10 or smaller networks, only support networks like AlexNet, VGG16 or LeNet5. Additionally, works like [1, 15] do not perform computations at floating point precision to avoid the hardware complexity this introduces. This often results in systems that can only cope with running smaller networks or smaller datasets. Accepting these reduced network and dataset options imposes constraints on the quality of the result and limits the trained solution's applicability to real world problems. Even for the network-dataset pairs that can be addressed by those works, they are often limited by memory bandwidth limitations. Overcoming these limitations often requires specific architectural changes to achieve competitive performance results.

Reducing the amount of data sent across the memory channel can help alleviate the memory bottleneck. An example is moving the im2col operation onto the FPGA and performing it on the fly. Offloading im2col can reduce the amount of input data transferred and thereby reducing onboard input memory utilization. Onboard memory can be reduced up to 25x such as in GoogLeNet. These benefits are only realized if the im2col is performed on-the-fly. The reason for this is if the im2col operation is performed, the results saved and then the GEMM executes, the design still saves the full expanded im2col processed matrix. Performing the im2col operation only when needed (on-the-fly) will ensure any data duplication is never saved to memory, allowing the design to benefit from the potential 25x memory decrease.

Before performing the im2col, there are three other transformations that need to occur to transform the tensors into matrices so that the convolution can be executed as a GEMM. These consist of dilation, rotation and permutation of the tensors. To place this into context and as a clarification recap, there are three convolutions that occur for each layer as shown in Eq. 5.1, Eq. 5.2 and Eq. 5.3:

$$Fwd := \underbrace{OFM}_{\text{OUT}} = \underbrace{(IFM)}_{\text{INP}} * dilate \underbrace{(weight)}_{KERN} \tag{5.1}$$

$$Grad_{weight} := \underbrace{\frac{\delta loss}{\delta weight}}_{\text{OUT}} = perm\left(perm \underbrace{(IFM)}_{\text{INP}} * perm \left(dilate \underbrace{\left(\frac{\delta loss}{\delta OFM} \right)}_{\text{KERN}} \right) \right) \tag{5.2}$$

$$Grad_{IFM} := \underbrace{\frac{\delta loss}{\delta IFM}}_{\text{OUT}} = dilate \underbrace{\left(\frac{\delta loss}{\delta OFM} \right)}_{\text{INP}} * perm \left(rot180 \underbrace{(weight)}_{\text{KERN}} \right) \tag{5.3}$$

In this setting and throughout the chapter, INP, KERN and OUT refer to the convolution parameters themselves. INP represents the input data, KERN represents the convolution kernel and OUT represents the result of the convolution. IFM, $weight$, OFM, Fwd, $Grad_{weight}$ and $Grad_{IFM}$ represent the convolution layer tensors in the network. IFM is the data brought into a convolution on the forward pass, $weight$ represents the convolution layer weights and OFM represents the activation values after the forward convolution. $Grad_{weight}$ is the computed result after a gradient w.r.t. weight convolution in the backward pass, and $Grad_{IFM}$ is the computed result after a gradient w.r.t. IFM convolution in the backward pass. Additionally, $Grad_{OFM} := \frac{\delta loss}{\delta OFM}$, which represents the gradient w.r.t. OFM. For a complete description on the exact details of how these apply to each convolution for the forward and backward convolutions encountered during training are discussed in full detail in the background in Sec. 2.3.1.

Beyond the reduced memory provided by performing the im2col on the FPGA, another reason for onboarding the im2col operation is that this will offload the burden of performing the im2col on the CPU. Additionally, the rotations, dilations and permutations required for the forward and backward convolutions are also onboarded further offload computation from the CPU. The CPU executes the im2col and the tensor transformations sequentially whereas a hardware approach can exploit inherent parallelism for further acceleration. Simultaneously, performing the im2col on the FPGA reduces the memory burden on the target hardware. By performing the im2col on-the-fly allows for all the benefits of onboarding this process to occur while not penalizing the throughput of the FPGA design. The onboarding of the im2col and matrix transformations for overall performance benefits while simultaneously reducing the memory footprint to allow for larger datasets like ImageNet all while in FP32, makes this architecture novel to all of its predecessors.

The fact that CuDNN convolution implementations already adopt this approach [84] validates the real world efficacy of onboarding these transformations, namely the im2col process. Please note that the GPU architecture is not tailored to characteristics of the network's layers. GPU architectures exhibit a fixed data flow implementation, a key point of departure from the proposed architecture of this work. The difference between the two approaches is elaborated upon in Sec. A.1.4.

The proposed design architecture is expected to adapt to the various requirements of a CNN and is required to target modern networks on large datasets i.e. GoogLeNet and SqueezeNet on ImageNet. The ImageNet dataset contains large inputs of 224x224, compared to 32x32 for the CIFAR datasets, resulting in a highly demanding workload when executed on these networks. To adapt to such large data and workload demands, an on-the-fly im2col module is proposed to be instantiated in hardware. Additionally the design must be runtime parameterizable, providing two key benefits. First, it will be able to support different convolutions without having to compile a kernel for each convolution configuration. Second, there will be no need to reconfigure the device if the consecutive convolutions share a hardware kernel. Both of these features will provide flexibility to the user and minimize the number of compiled kernels needed to train the target network. However the ability to execute multiple different convolutions comes at a cost to performance. Understanding of the trade-off between performance and the desired number of compiled kernels is of key importance to most users of FPGPT and is explored throughout this work.

5.2 Related Works

Similarly to Caffe Barista, FPGPT solely looks at accelerating CNN training through the computation of convolutions as GEMMs. This provides it two areas of comparable works. There are the state-of-the-art (SOTA) works in performing convolutions as GEMMs and the works that enable CNN training on FPGAs. For evaluation, only the latter will be considered as those share the same application space as FPGPT. Both of these areas are analyzed in Sec. 2.6 to show the reader what solutions currently exist for performing GEMMs on an FPGA.

As a refresher to the reader, the main works that are relevant to this chapter are summarized here to ensure proper context is provided for the rest of the chapter. FeCaffe [74] is currently the SotA in framework integration as well as allowing for FP32 training of modern networks on relevant datasets. From a hardware design perspective, EF-train [73] is the only work to

	Networks	**Dataset**	**Representation**	**FPGAs**
FeCaffe [74]	AlexNet, VGG16, SqueezeNet, GoogLeNet	ImageNet	FP32	1× Stratix 10
EF-Train [73]	AlexNet, VGG16	ImageNet	FP32	1× PYNQ-Z1, 1× ZCU102
FPDeep [1]	AlexNet, VGG16	CIFAR-10	FXP16	15-100× XC7VX690T

Table 5.1: SotA FPGA Convolution Breakdown Summary

create an FPGA architecture specifically for convolution able to provide high accuracy results due to FP32 training and high performance, but lacking integration characteristics. From a peak performance perspective, FPDeep [1] is currently the SotA for achievable performance obtaining the highest GOPS design, but at FXP16 on the CIFAR-10 dataset and using 15 server-grade FPGAs. Tab. 5.1 summarizes the key characteristics of these three SotA works.

Departing from the previous approaches, the FPGA based works in this thesis aim to be able to compete with the three SotA works by being the first unified toolchain providing a parameterizable architecture of an FPGA-based system for training CNNs at FP32 that does not impose any restriction on the characteristics of the input topologies (size or type of layer), or target dataset, and at the same time is integrated to an existing widely used ML framework: PyTorch [75].

5.3 Hardware Architecture

The hardware architecture of the design presented in this section has been created with the explicit purpose of performing the convolutions that calculate the Forward (Fwd), $Grad_{IFM}$ (GI) and $Grad_{weights}$ (GW) tensors as GEMMs operations as shown in Eq. 5.1, Eq. 5.3 and Eq. 5.2 respectively. These convolutions are the three most costly operations in CNN training, taking up between 60-90% of all compute time. Each of these convolutions has unique requirements, requiring various forms of data manipulation in order to be executed as a GEMM. To address the individual needs of each type of convolution, the design has been split into four keys components:

1. INP data handling

Forward	**Grad**$_{IFM}$	**Grad**$_{weights}$
$INP = IFM$	$INP = Grad_{OFM}$	$INP = IFM$
$KERN = weights$	$KERN = weights$	$KERN = Grad_{OFM}$
$OUT = OFM$	$OUT = Grad_{IFM}$	$OUT = Grad_{weights}$

Table 5.2: Mapping of each convolution parameter to the corresponding CNN tensor

2. KERN data handling,
3. GEMM computation kernel (Systolic Array)
4. OUT data handling

For each of the four sections of the architecture, a high level description is provided, followed by an in-depth breakdown of the core components of each section. Utilizing the presented breakdown, the formulaic basis for the performance model is also derived. As a result, this section both breaks down the FPGPT hardware architecture and uses this breakdown to explain how the performance model was derived. The next section discusses how the derived performance model is used to conduct design space exploration for identifying the best performing kernels under the target workload and device.

Within the hardware architecture, INP, KERN and OUT represent the three tensors involved in the convolution. Tab. 5.2 shows how for each type of convolution, the CNN data matrices (IFM, weights, OFM, $Grad_{OFM}$, $Grad_{IFM}$ and $Grad_{weights}$) map to the memory subsystems that handle INP, KERN and OUT. For example, for $Grad_{IFM}$, $Grad_{OFM}$ is assigned as the INP tensor for the convolution, $weights$ as the KERN tensor and $Grad_{IFM}$ as the OUT tensor. Assigning the relevant tensor to the convolution parameters INP, KERN and OUT separates describing the functionality of the overall architecture and memory subsystems from the details of the operation of the convolutions within CNN training.

Each of the three tensors required for a convolution seen during CNN training has at least four dimensions. Only the convolution for $Grad_{weight}$ has a fifth dimension, which will be explained in Sec. 5.3.1. Each dimension of the tensors is assigned a name. Going from the inner to outer most dimensions, the are W, H, C, N and A. Such a system allows the tensors INP, $KERN$

	Forward				**Grad$_{IFM}$**				**Grad$_{weights}$**				
	N	C	H	W	N	C	H	W	A	N	C	H	W
INP	B_{size}	Ch	H_{IFM}	W_{IFM}	B_{size}	D	H_{OFM}	W_{OFM}	B_{size}	Ch	1	H_{IFM}	W_{IFM}
KERN	D	Ch	H_{weight}	W_{weight}	Ch	D	H_{weight}	W_{weight}	B_{size}	D	1	H_{OFM}	W_{OFM}
OUT	B_{size}	D	H_{OFM}	W_{OFM}	B_{size}	Ch	H_{IFM}	W_{IFM}	N/A	D	Ch	H_{weight}	W_{weight}

Table 5.3: Architecture variables for each type of convolution

and OUT as well as they dimensions A, N, C, H, and W to be parameters of the hardware architecture and be completely agnostic of which convolution is being performed and which stage training is in. Tab. 5.3 shows how the CNN tensors map onto the hardware architecture tensors. For clarity, B_{size} is batch size, Ch is channels, H and W represent the height and width of the IFM, weight or OFM.

For the convolutions performed for the Forward and $Grad_{IFM}$, N_{INP} becomes N_{OUT} and N_{KERN} becomes C_{OUT}. This would mean that the OUT for the $Grad_{weights}$ convolution would be expected to be of shape $Ch \times D \times H_{weight} \times W_{weight}$. Yet the expected shape is $D \times Ch \times H_{weight} \times W_{weight}$, corresponding to the shape of the weights. Conversion from $Ch \times D \times H_{weight} \times W_{weight}$ to $D \times Ch \times H_{weight} \times W_{weight}$ is performed by the outermost permutation as seen in Eq. 5.2. Details on permutation and the A column in the $Grad_{weights}$ grouping as well as how that applies to the $Grad_{weights}$ computation are provided in the following section (Sec. 5.3.1).

5.3.1 Gradient w.r.t. Weight

With the aim to optimize the design, the calculation of Eq. 5.2 must be refactored resulting in the removal of the need to perform permutations on the INP tensor. Performing a permutation on the INP tensor significantly increases the complexity of the hardware resulting in a lower performance design. No such transformation is required for the Fwd or $Grad_{IFM}$ computation as they do not perform any permutations on the INP tensor.

In Sec. 5.1 and in Sec. 2.3.1 the $Grad_{weight}$ is calculated using Eq. 5.2. For the $Grad_{weight}$ convolution, the INP and KERN correspond to the IFM and $Grad_{OFM}$ respectively (see Table 5.3). Both are permuted before the convolution begins. A final permutation on the OUT structure is performed at the end of the computation. To improve the performance on the computation of $Grad_{weight}$, we propose to eliminate the need for INP permutation. Rewriting the convolution with an accumulation across B_{size} replaces the INP and KERN permutations as shown:

$$\frac{\delta loss}{\delta weight} = \sum_{n=0}^{B_{size}} perm\left(IFM'[n] * dilate\left(Grad'_{OFM}[n]\right)\right) \tag{5.4}$$

where $IFM' = transform(IFM)$ and $Grad'_{OFM} = transform\left(\frac{\delta loss}{\delta OFM}\right)$. The transform function takes a 4D tensor of $N \times C \times H \times W$ and creates a 5D tensor $N \times C \times 1 \times H \times W$, inserting an additional dimension at dimension 3. This transformation is shown in Tab. 5.3 where the $Grad_{weights}$ group of columns has 5 dimensions rather than 4. To ensure the design can refer to the same variables, in the 5D tensor, the format is referred to as $A \times N \times C \times H \times W$, so if the tensors starts off as $N_{original} \times C_{original} \times H_{original} \times W_{original}$ after the transform $A = N_{original}$, $N = C_{original}$, $C = 1$, $H = H_{original}$ and $W = W_{original}$.

A notable implication is how dimensions of INP and KERN map to OUT. For the forward and $Grad_{input}$ convolutions, $C_{OUT} = N_{KERN}$ and $N_{OUT} = N_{INP}$. For the $Grad_{weight}$ convolutions, $C_{OUT} = N_{INP}$ and $N_{OUT} = N_{KERN}$. The hardware architecture for $Grad_{weight}$ differs from Forward and $Grad_{IFM}$ convolution hardware. This is due to the differences in how C_{OUT} and N_{OUT} relate to N_{INP} and N_{KERN} and the need for accumulation of the A dimension (i.e. introduced dimension) that are produced in accordance with Eq. 5.4.

Tab. 5.3 shows the transformed IFM and $Grad_{OFM}$ as INP and KERN respectively as well as the accumulate variable in the $Grad_{weight}$ section. The architectural implications will be discussed in Sec. 5.3.7 and 5.3.8.

5.3.2 Architecture Overview

All the work presented here assumes a heterogeneous system of a non-SOC FPGA board with a PCI express connection to the host CPU, one of the most common high-end FPGA setups. It should be noted that the design can be adapted to function on all types of FPGA setups.

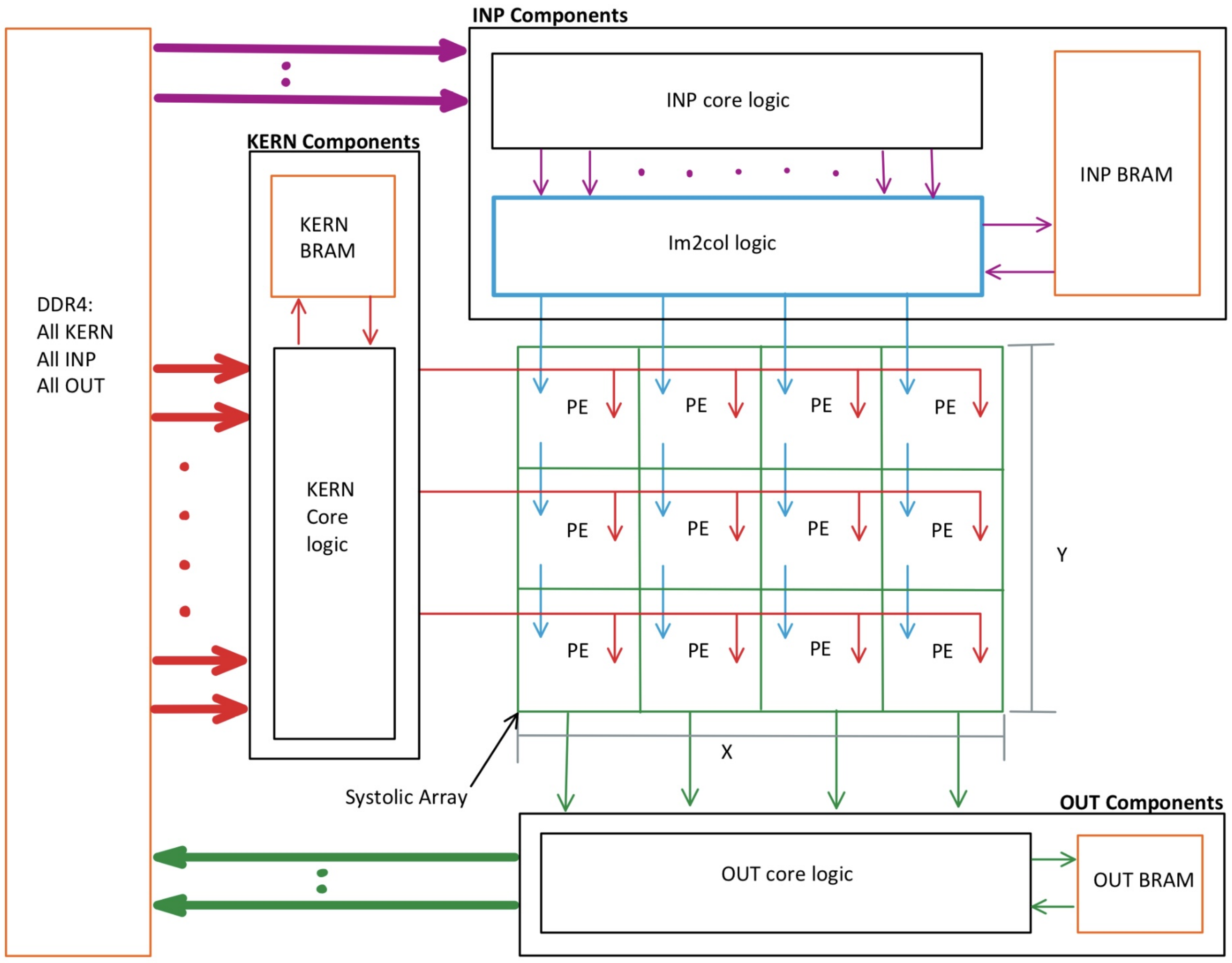


Figure 5.1: Hardware Architecture Red=KERN, Purple=INP, Blue=I2C, Green=OUT

The top-level design of the proposed architecture is shown in Fig. 5.1. The design consists of two key parts:

1. The memory subsystem: responsible for reading and manipulating the convolution input and output data
2. The systolic array (SA): responsible for data processing

The systolic array (green box) is designed to perform the GEMM operation. The internal black and blue boxes represent the data handling components. Connected to the data handling components are the orange boxes representing the on-chip memory. Each of the three memory subsystems has an independent on-chip memory. Combining the data handling component and the on-chip memory represent the overall memory subsystem.

The architecture's execution time is affected by both compile time parameters and runtime parameters. All compile time parameters are static during runtime and are defined when the hardware kernel is compiled. The runtime parameters are flexible and determined during execution when performing the convolution. Runtime flexibility is essential to allowing the system to adapt to the target workload on-the-fly. There are two sets of parameters that define all the configurable aspects of the architecture; α and β. The α set comprises of all static compile-time parameters and the β set comprises of all the runtime parameters. As various architecture design elements are introduced, the corresponding contents of α will be introduced accordingly.

$$\begin{aligned}\beta =\{&A, N_{(INP,KERN,OUT)}, C_{(INP,KERN,OUT)}, H_{(INP,KERN,OUT)}, W_{(INP,KERN,OUT)},\\ &Z_{channel}, KRange, stride, pad, dilation\}\end{aligned} \tag{5.5}$$

The parameters in the β set are provided to the architecture at runtime and vary depending on the target layer and convolution type. These parameters are in place to allow the design to adapt to the target workload during runtime to allow for more efficient, tailored execution.

5.3.3 Systolic Array Architecture

The systolic array (SA), represented by the green grid in Fig. 5.1 is an array of multiply and accumulate (MAC) processing elements (PE). The SA performs a GEMM based on the data provided by the memory subsystem as shown in Fig. 5.1 by the thin red and blue arrows. The SA has a degree of parallelism of $X \times Y$, which can be adjusted at compilation time to scale resources and performance. As X and Y are compile time parameters, $X, Y \in \alpha$, they cannot be adjusted at runtime. Although N_{KERN} and N_{INP} relate to different dimensions of OUT for $Grad_{weight}$ compared for Fwd and $Grad_{IFM}$, as discussed in Section 5.3.1, the systolic array architecture is no different for $Grad_{weight}$.

Referring back to Sec. 2.3.3, to produce a 2D OUT matrix, $Matrix_{INP}$ of dimensions $O \times M$ is multiplied by $Matrix_{KERN}$ of dimensions $M \times P$ where M refers to the common dimension of the GEMM, and O and P are the size of the remaining dimension of $Matrix_{INP}$ and

$Matrix_{KERN}$ respectively. Each processing element (PE) multiplies a KERN value by an INP value and accumulates the computed product each cycle. As a result, it takes M multiply and accumulate operations for one PE to compute a single element of a 2D OUT tile.

In the systolic array, each processing element corresponds to an element of the 2D OUT tile presented in Sec. 2.3.3. The columns of the SA corresponds to the O dimension of $Matrix_{INP}$, enabling X elements of a 2D OUT tile to be computed simultaneously. If $X < O$, $Matrix_{INP}$ must be broken into $\lceil \frac{O}{X} \rceil = \lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \rceil$ blocks, where each block is of dimensions $X \times M$. To complete an entire 2D OUT tile, $\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \rceil$ iterations of M MACs occur. Please not that if $X > H_{OUT} \cdot W_{OUT}$ the excess columns will not be utilized during computation, wasting resources. The aim should therefore be to have X divide into $H_{OUT} \cdot W_{OUT}$ in such a way to maximize performance and minimize moot computation.

As shown in Sec. 2.3.3, a 3D KERN tensor is transformed to a flat 2D matrix $Matrix_{KERN}$ of $M \times 1$. Each of these 2D matrices is provided simultaneously to every element in a single PE row, as indicated by the thin red arrows in Fig. 5.1. There are Y rows, resulting in Y $Matrix_{KERN}$ that are computed in parallel. Correspondingly Y out of the total N_{KERN} 3D KERN tensors are loaded in parallel. Just as X is related to $H_{OUT} \cdot W_{OUT}$, Y is related to N_{KERN}. If $Y > N_{KERN}$ there will be $Y - N_{KERN}$ wasted rows of computation.

To achieve maximum performance, the architecture is designed to achieve an iteration interval (II) of 1 for each PE. This requires each PE to receive a KERN and an INP value every cycle to perform 1 MAC every cycle. Given the above architecture and assuming this is a forward or $Grad_{input}$ convolution, Y of the C_{OUT} OUT channels are being computed in parallel. They each take $M \cdot \lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \rceil$ cycles (assuming $II = 1$) to compute Y complete 2D OUT matrices. The above process is repeated for all the channels to produce a full 3D OUT tensor of size $C_{OUT} \times H_{OUT} \times W_{OUT}$. This will be referred to as Completing N_{KERN} for the rest of the chapter.

The first 3D OUT tensor corresponds to the convolution of the first 3D INP tensor and the entirety of the weights. As there are N_{INP} 3D INP tensors, the entire process thus far is repeated N_{INP} times, leading to $N_{INP} = N_{OUT}$. This also leads to the overall SA runtime for a standard convolution executed as a GEMM to be:

$$\Gamma_{SA}(\alpha,\beta) = \underbrace{M \cdot \left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{Y \text{ 2D OUT matrices}} \cdot \underbrace{\left\lceil \frac{N_{KERN}}{Y} \right\rceil}_{\text{Completing } N_{KERN}} \cdot \underbrace{N_{INP}}_{\text{Completing } N_{INP}} \tag{5.6}$$

This will be referred to as Completing N_{INP} for the rest of the chapter.

In each clock cycle, I2C data (blue) is provided to the first row of the SA and then gets propagated to the next row. This propagation scheme ensures the on-chip memory ports are only connected to the first row of PEs, reducing the routing pressure of the design. On the first cycle, all X columns in the first row SA receive a value, leaving $Y-1$ rows with no I2C data. This creates a *shifting delay* of Y cycles before the processing elements in the last row of the SA receive their first value. This delay is incurred for all $A \cdot N_{INP} \cdot \lceil \frac{N_{KERN}}{Y} \rceil$ computations of a Y 2D OUT matrix. Ideally this would be amortized away through constant data feeding, but restrictions in the implementation language (Vitis HLS) prevented this from being possible. Accounting for the shifting delay the overall performance becomes:

$$\Gamma_{SA}(\alpha,\beta) = \left(\underbrace{M \cdot \left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{Y \text{ 2D OUT matrices}} + \underbrace{Y+8}_{\text{shift delay + acc}} \right) \cdot \underbrace{\frac{N_{KERN}}{Y}}_{\text{Completing } N_{KERN}} \cdot \underbrace{N_{INP}}_{\text{Completing } N_{INP}} \tag{5.7}$$

Increasing parallelism through Y or increasing N_{INP} both increase the impact of this penalty.

Finally, if the convolution is a $Grad_{weights}$ convolution, the accumulation must be accounted for. This means A 4D OUT tensors need to be produced by the SA (accumulation happens in the OUT memory subsystem and is addressed in Sec. 5.3.8). For $Grad_{weights}$ the execution cycle total becomes:

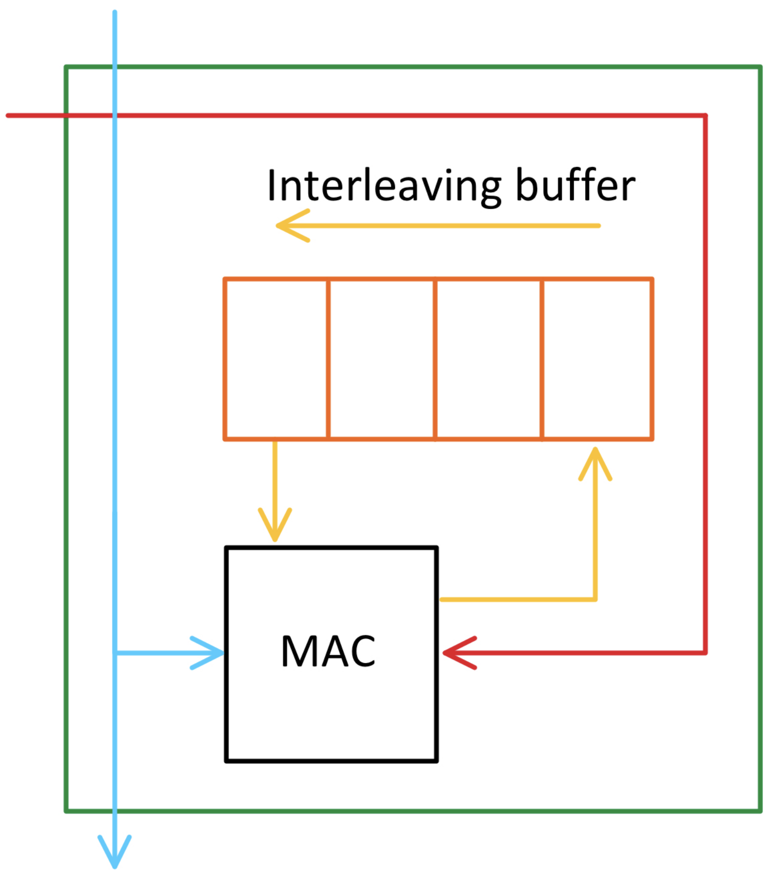
$$\begin{aligned}&\Gamma_{SA}(\alpha,\beta) = \\ &\left(\underbrace{M \cdot \left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{Y \text{ 2D OUT matrices}} + \underbrace{Y+8}_{\text{shift delay + acc}} \right) \cdot \underbrace{\left\lceil \frac{N_{KERN}}{Y} \right\rceil}_{\text{Completing } N_{KERN}} \cdot \underbrace{N_{INP}}_{\text{Completing } N_{INP}} \cdot \underbrace{A}_{\text{Completing } A}\end{aligned} \tag{5.8}$$

For the Forward and $Grad_{IFM}$ convolutions, $A = 1$. Every component in the architecture that provides or consumes data to another will be tightly coupled to each other. Consequently because the SA operates with $II = 1$, every component operates at an $II = 1$ as well to ensure the SA does not stall.

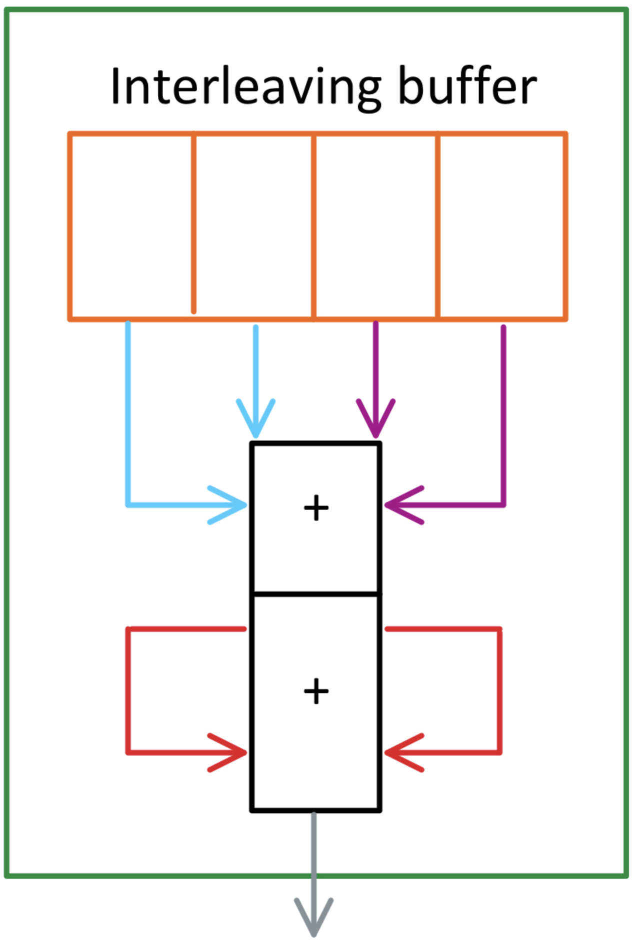
Processing Element Details

A GEMM operation consists of a sequence of multiply and accumulate operations (MACs). In the proposed SA design each PE performs 1 MAC per cycle. As this is a floating-point hardware design, maintaining an II=1 with a MAC requires using an interleaving buffer to store partial results and a partial results adder to compute the final result.

The design of the interleaving design can be seen in Fig. 5.2. The interleaving floating point PE design consists 2 phases. Phase 1 performs multiplication of the INP value (blue) with the KERN value (red). The product is accumulated into one of the 4 intermediate accumulation variables. There need to be at least 4 intermediate variables as it takes 4 cycles to fulfill any dependencies for a floating point MAC. The DSPs on the FPGA are able to pipeline the multiplication operations as long as the input variables to the DSP are not required in the next iteration. This phase is identical to the design presented in the previous chapter for Caffe Barista. Unlike the previous chapter, in this design, there 11-element cache limitation has been fixed through improved HLS coding techniques. As soon as all the MACs for a M computations have been completed, phase 2 starts. Phase 2 consists of performing the accumulation of the intermediate values. The design of this phase is where the FPGPT design differs from the Barista design. In the Barista design, no MACs are occurring as the accumulation is happening. If the FPGPT design, each PE in actuality contains 2 interleaving buffers so that Phase 2 of one OUT element can happen synchronously with Phase 1 of the next OUT element. This ensures the PE maintains its II=1 requirement. To save on hardware resources, all three additions (represented by different arrow colors) in this section are performed on the same adder, resulting in an 8 cycle delay in performing the addition. Furthermore, the double buffered interleaving means that the *shifting delay* is incurred less frequently.



(a) Phase 1: MAC with interleaving buffer with incoming KERN value (red), incoming I2C value (blue) and interleaved accumulation value (yellow).



(b) Phase 2: Progressive accumulation of interleaving buffer

Figure 5.2: Both phases of floating-point MAC design to enable II=1

5.3.4 Memory Subsystem Architecture

The memory subsystem is responsible for providing the SA with data from the off-chip memory and writing the results of the SA back to the off-chip memory. It provides each PE with the correct data per cycle and consumes the completed calculation values and prepares them for writing out. The INP and KERN memory subsystems perform standard data manipulation like dilation and reordering. They also perform the required data manipulations stated in Eq. 5.2 and Eq. 2.3, such as permutation and rotation of the data that feeds the SA. Uniquely, the INP memory subsystem performs the on-the-fly *im2col* procedure. The OUT memory subsystem takes in the computed SA values and performs reordering to ensure it is in the format accepted by the ML framework. Which operations are performed and how they are performed varies depending on the convolution type at hand. Both the KERN and OUT memory subsystems have separate functionality and therefore different designs for $Grad_{weights}$ (GW) convolutions. This unique functionality and design will be addressed separately from the Forward and $Grad_{IFM}$ (FGI) convolution data handling.

5.3.5 DDR4 interaction

The DDR4 off-chip memory hosts the INP, KERN and OUT data. When the computation convolution starts INP and KERN data is read from the DDR4 off-chip memory into the respective memory subsystems. As soon as the OUT memory subsystem has received and handled its provided data, the DDR4 off-chip memory hosts the ordered data before it is transferred back to the host to continue execution. The memory subsystems perform all data handling, so the host CPU can directly pass data to and from the ML framework without any manipulation. All off-chip memory interactions burst read on a $BAND \cdot$ word length bit channel. Additionally the memory interfaces of both INP and KERN are doubled. The burst reading and additional interfaces amortize the AXI-interconnect 35 cycle latency [101], leading to highly efficient memory transfers $BAND$ refers to the maximum port bandwidth in words of the target hardware. The details of how the off-chip memory is handled by the individual memory subsystems will be discussed in their respective sections; Sec. 5.3.6, Sec. 5.3.7 and

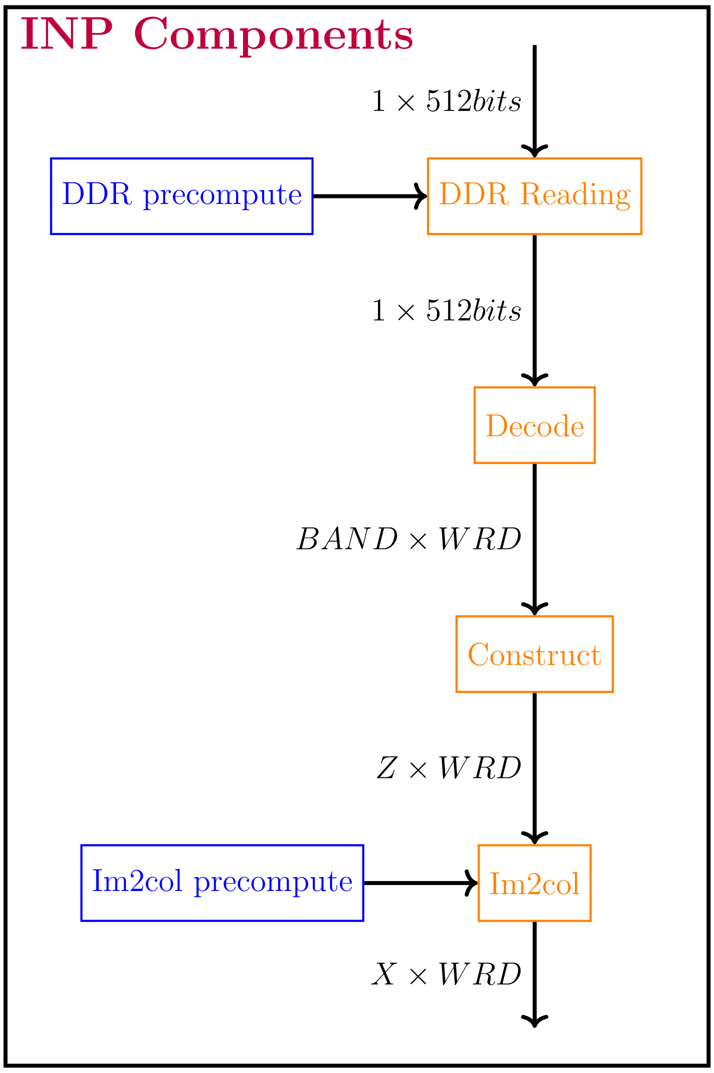


Figure 5.3: INP functional components

Sec. 5.3.8.

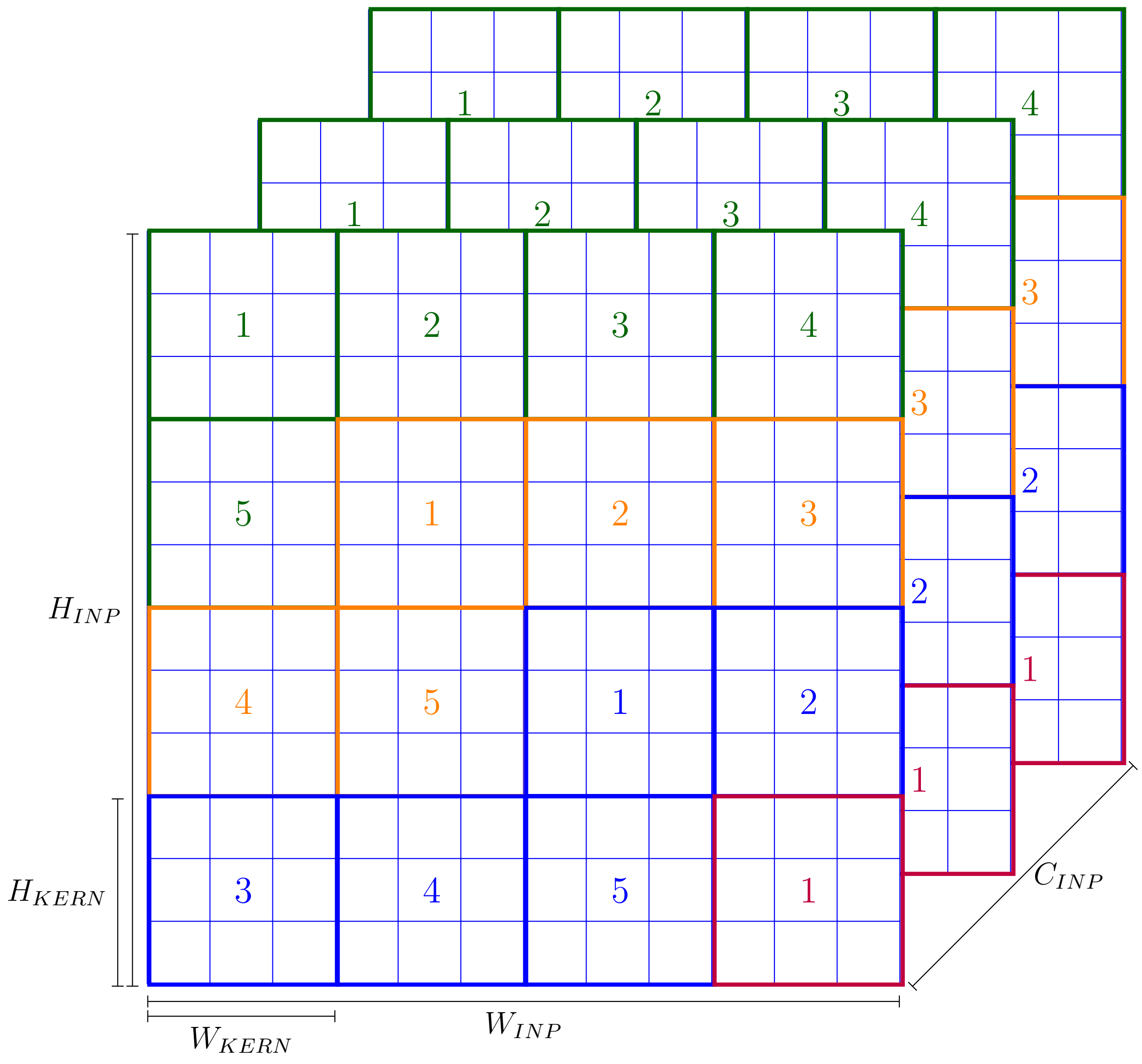
5.3.6 INP Memory Subsystem: Design and Performance Model

In this subsection, the INP memory subsystem is introduced with a high level description of the subsystems operation, concluded with an overall performance model of the INP memory subsystem. For a detailed breakdown of each component that creates the entire memory subsystem alongside the description of each components functionality as an equation for that components contribution to the performance model, see Appendix A.1. For each memory subsystem, the notation of Γ_x is maintained for every memory subsystem and always represents the cycles required for each memory subsystem component listed as x. For details on each of the individual

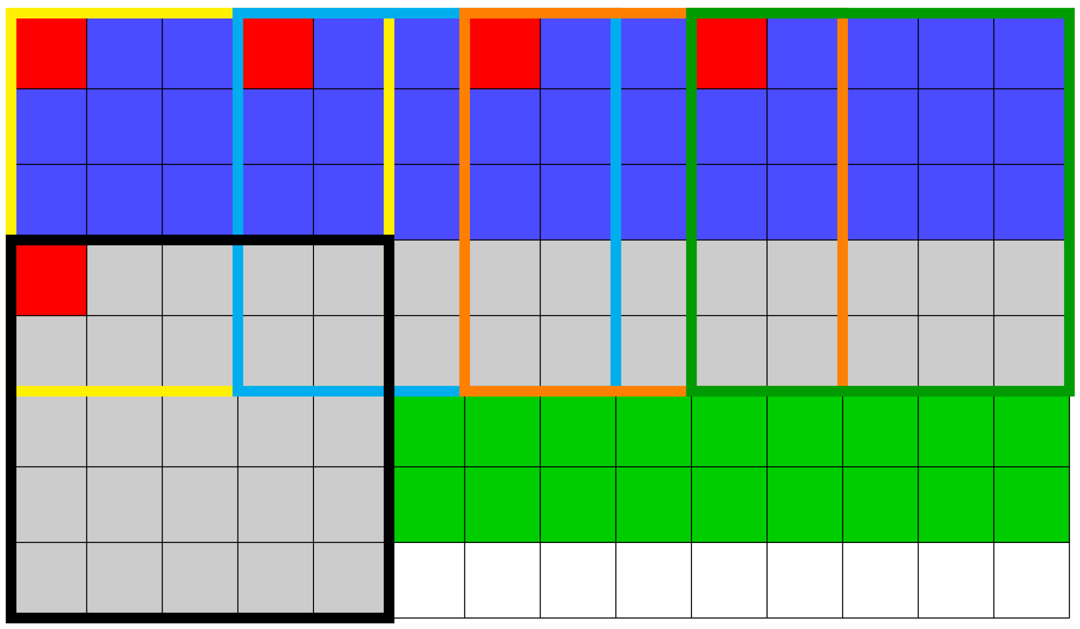
Γ values, please refer to Appendix A.

The INP memory subsystem consists of 4 data handling components, INP DDR reading, INP decode, INP construct and INP im2col. These components read in and manipulate the data. The two precompute components aid in achieving an II=1. Fig. 5.3 shows how the various functional components (orange) and precompute components (blue) are connected. In the overall setting of Fig. 5.1, Fig. 5.3 represents a functional breakdown of the INP components block in Fig. 5.1. Depending on the dimensions of the workload that is being computed and the dimensions of X, Z, $BAND$ and WRD different components could be the cause of a bottleneck. Considering each component is tightly coupled to its subsequent component, a bottleneck early on in the process can cause all subsequent components to consistently stall. To minimize the chance of components stalling and the according performance penalties, all components must have an $II = 1$. The connection arrows in Fig. 5.3 shows the amount and width ($amount \times width$) of the FIFOs containing data that connect the components. WRD refers to the number of bits of a word as required by the precision of the design. The design also contains control FIFOs that are responsible for controlling dynamic iteration spaces throughout training. Additionally, there are precompute FIFOs that separate precomputation of memory/buffer locations from execution. This could be merged, but implementation limitations due to HLS have resulted in separating precomputation. These control and precompute FIFOs are left out of Fig. 5.3 for clarity.

Most ML frameworks provide data in an $N_{INP} \times C_{INP} \times H_{INP} \times W_{INP}$ fashion on the off-chip memory. As mentioned in Sec. 5.3.3, in M cycles the SA computes X 2D OUT tile elements per cycle across Y OUT channels. The INP memory subsystem must provide the SA with the relevant X values each cycle to maximize the SA utilization. To visualize how the tiling affects the reading of INP data, Fig. 5.4a shows the scenario where $X = 5$, $H_{KERN} = W_{KERN} = 3$, $C_{INP} = 3$, $H_{INP} = W_{INP} = 12$, $stride = 3$, $H_{OUT} = W_{OUT} = 4$ and $O = 9$, for a 3D INP Tensor. $stride = W_{KERN}$ avoids overlap of data to make the explanation clearer to follow. Furthermore, KERN and INP dilation are left out for the same reason. Each colored block is a location that a 2D KERN tile would occupy on a 2D INP tile. Because $X < O$, $Matrix_{INP}$ must be broken in into $\lceil \frac{O}{X} \rceil = \lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \rceil$ blocks of dimensions $X \times M$ as discussed in Section



(a) 3D INP tensor read for with $X = 5$, $H_{KERN} = W_{KERN} = 3$, $C_{INP} = 3$, $H_{INP} = W_{INP} = 12$, $stride = 3$, $H_{OUT} = W_{OUT} = 4$ and $O = 9$



(b) Extended Line-buffer demonstration with X target values (red), Z (blue), $Z_{channel}$ (blue, green & gray)

Figure 5.4: INP Memory subsystem data loading

5.3.3. The data required to construct the first block of $Matrix_{INP}$ is all the green squares in Fig. 5.4a. The number in the square corresponds to which of the X SA columns that data belongs. Because the block is of dimension M, it requires data from each C_{INP} channel. The results in the data in the orange, blue and purple squares in each channel of the 3D INP tensor to be skipped until all green box values are read. This scenario of skipping of data will be referred to as Channel Jumping from here on.

After all the green squares are read across all channels of the 3D INP tensor, the orange squares are read across all channels, followed by blue and then purple. This process of reading in sets of X squares will be referred to as X Set Jumping for the rest of the chapter. The sequential nature of the data from the top left of square 1 until the bottom right of square 5 form a predictable partially-sequential access pattern. By being partially sequential and predictable, all this data can be burst read to improve access efficiency.

Providing the SA with X values in the first cycle requires more than X values to be present at that point in time. Fig. 5.4b portrays a convolution where $X < O$ so $Matrix_{INP}$ is constructed in blocks of $X \times M$. In Fig. 5.4b the background grid represents one channel of the 3D INP tensor. The outlined boxes represent kernel locations and the red squares correspond to the X values required to construct the first column of $Matrix_{INP}$. As can be seen, the red square values are not consecutive. Data can only be efficiently provided when burst read from memory which requires sequential data. Furthermore, with exception of the first red tile, all data is involved in the convolution for multiple different kernel positions and are reused multiple times. The distance from the first to the last red square is represented by Z. Z is the degree of parallelism required by the INP memory subsystem because at any point, X of these Z values might be required by the SA.

A line-buffer is implemented to provide all the relevant values and avoid rereading values from memory multiple times. Ideally the line-buffer receives Z values every cycle. The size of the required line-buffer is represented by $Z_{channel}$, which is all the values represented by the blue, gray and light green squares in Fig. 5.4b. This is the amount of data required to perform all the MACs for X columns of the SA for a single channel of the 3D INP tensor. When all the data from one channel is loaded into its line-buffer, the INP memory subsystem immediately starts reading the data for the next channel.

Z is a compile time parameter and therefore $Z \in \alpha$. $Z_{channel}$ is both a runtime and compile time parameter. $Z_{channel,\alpha}$ and $Z_{channel,\beta}$ represent the compile time and the runtime version respectively. The compile and runtime variables do not need to be equal as long as $Z_{channel,\alpha} \geq Z_{channel,\beta}$. Furthermore, $Z_{channel,\alpha} \in \alpha$ and $Z_{channel,\beta} \in \beta$.

INP Overall Performance Model

The overall performance of the INP memory subsystem is represented by:

$$\Gamma_{INP}(\alpha,\beta) = \max\left(\Gamma_{INP-DDR\ Read}(\alpha,\beta), \Gamma_{INP-Constr}(\beta), \Gamma_{INP-I2C}(\alpha,\beta)\right) \tag{5.9}$$

where Γ_x is the number of cycles required for that unit. $\Gamma_{INP-Decode}$ is not included due to the guarantee of it never being a bottleneck, as presented in Eq. A.9.

The reason the overall performance is the max across all the components is twofold. First off, all the memory subsystems are tightly coupled receiving data from their predecessors through streams, starting execution as soon as data arrives. As a result, the amount of cycles spent stalling before executing is unimpactfully small and not included in the performance calculations. Secondly, due to the fact each component is required to operate at an $II = 1$, the highest cycle count component will be the bottleneck for the entire memory subsystem.

5.3.7 KERN Memory Subsystem: Design and Performance Model

Just like in Sec. 5.3.6, this section provides an overview of the overall memory subsystem with an accompanying performance model. The detailed breakdown of the operation and performance model of each component can be found in Appendix A.2. The KERN memory subsystem has two distinct versions. One version is for the forward and gradient w.r.t. IFM convolutions (FGI). The other version is for the gradient w.r.t. weight convolutions (GW). The amount

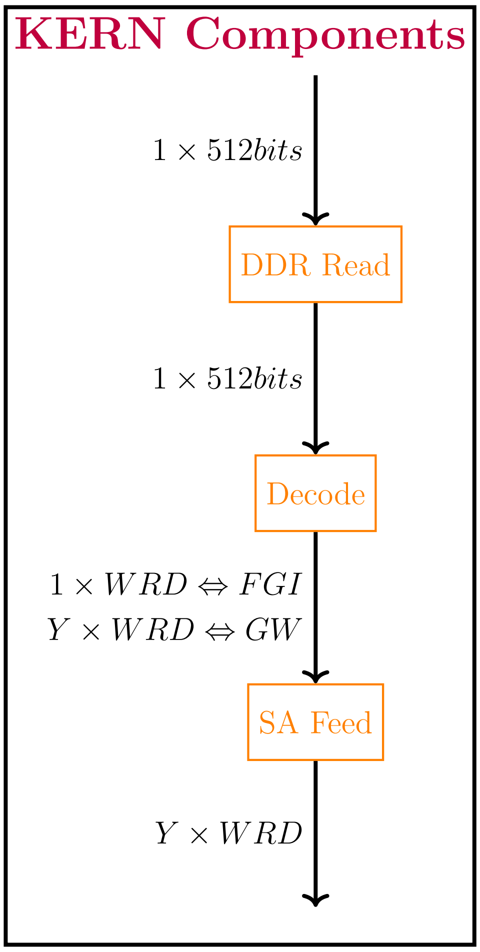


Figure 5.5: KERN functional components

and type of functional components are the same between both scenarios, just the method of operation differs. A final, single, equation is created as the overall performance model.

The KERN memory subsystem must flatten all the 3D KERN tensors as discussed in Sec. 2.3.3. For clarification, Fig. 5.6 demonstrates the flattening process which transforms the KERN tensor from a 3D tensor to a 2D matrix of dimensions $M \times P = C_{KERN} \cdot H_{KERN} \cdot W_{KERN} \times P$. When the 3D KERN tensors are stored in memory and read sequentially, this automatically flattens the weight tensor to resemble $Matrix_{KERN}$. In Sec. 5.3.3, it was stated that the KERN memory subsystem provides data from Y different $Matrix_{KERN}$. As a result, the data required in the first cycle in each subsequent row is $C_{KERN} \cdot H_{KERN} \cdot W_{KERN}$ words apart in memory. The way this is dealt with depends on whether it is an FGI or GW convolution. Each of these convolution types is described and modeled separately as they diverge for each component, with details described in Appendix A.2. Nonetheless, they both contain only the components

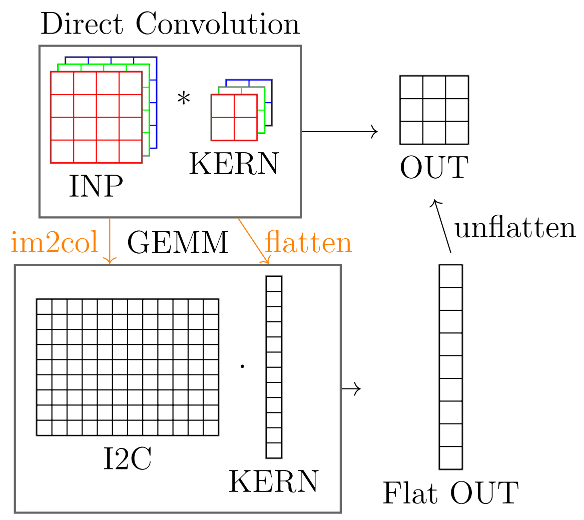


Figure 5.6: Im2col and Flattening recap where $N_{KERN} = 1$, $C_{KERN} = 3$, $H_{KERN} = 2$, $W_{KERN} = 2$, $M = 12$, $P = 1$

as described in Fig. 5.5. Similarly to the INP components, Fig. 5.5 represents a functional breakdown of the KERN Components block in Fig. 5.1.

KERN overall

The overall performance for FGI has a clear bottleneck regardless of target workload. In the conclusion from the KERN FGI Decode it was demonstrated that $\Gamma_{KERN-Decode}(\beta) > \Gamma_{KERN-DDR}(\beta)$ (Eq. A.16) . Additionally in the section describing the KERN FGI Systolic Array Feeder $\Gamma_{KERN-Feed_{Phase1}}(\beta) = \Gamma_{KERN-Decode}(\beta)$ (Eq. A.18). Combining these conclusions:

$$\begin{gathered}\Gamma_{KERN-DDR}(\beta) < \Gamma_{KERN-decode}(\beta) = \Gamma_{KERN-Feed_{Phase1}}(\beta) < \Gamma_{KERN-Feed}(\alpha, \beta) \\ \therefore \Gamma_{KERN_{FGI}}(\alpha, \beta) = \Gamma_{KERN-Feed}(\alpha, \beta)\end{gathered} \tag{5.10}$$

This shows no matter what the FGI implementation has a single component dictating its performance.

Unlike for the FGI scenario, for the $Grad_{weight}$ scenario there is no way to guarantee which component will dictate the performance of the KERN components. As a result, the overall

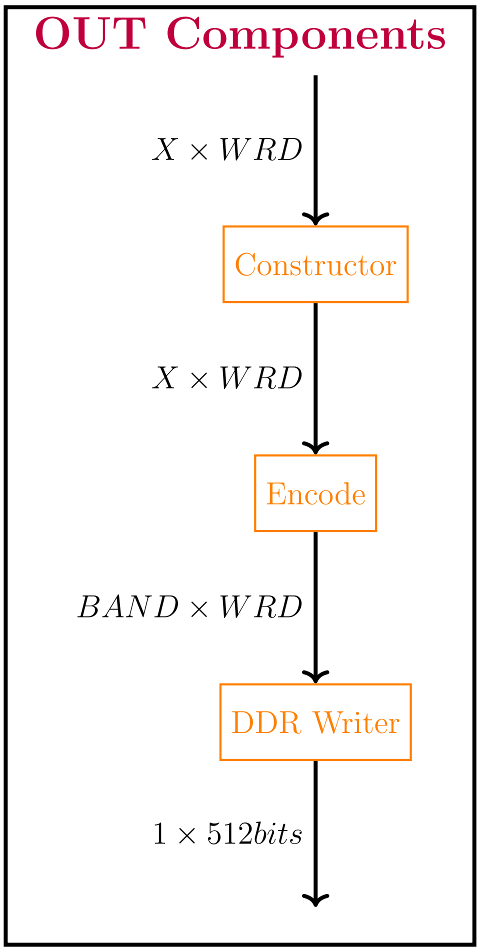


Figure 5.7: OUT functional components

performance is calculated as:

$$\Gamma_{KERN_{GW}}(\alpha, \beta) = \max(\Gamma_{KERN-DDR}(\beta), \Gamma_{KERN-decode}(\beta), \Gamma_{KERN-Feed}(\alpha, \beta)) \tag{5.11}$$

5.3.8 OUT Memory Subsystem: Design and Performance Model

Fig. 5.7 represents a functional breakdown of the OUT components block in Fig. 5.1.

The OUT memory subsystem consists of three key components. The first receives the completed $X \times Y$ values from the SA and ensure all the data is stored in the correct order. The next groups the into sets of $BAND$ words. The final component burst writes the grouped data to the off-chip memory. These units can be viewed as the inverse of components introduced for INP and KERN.

As mentioned in Sec. 5.3.3, OUT is computed as Y 2D OUT tiles in the SA, where each tile contains X values. Due to the *shifting delay*, each row of PEs of the SA completes its MACs 1 cycle apart. Therefore the OUT constructor receives X incoming values per cycle when the MACs have completed. Fig. 5.8 depicts how data arrives and is then sorted into the final OUT array. The part of the diagram above the arrows shows across multiple time stamps, represented by t, how data arrives from the systolic array. Channel in the diagram refers to C_{KERN} and C_{INP} as $C_{KERN} = C_{INP}$. As discussed, each channel within a timestamp will arrive one cycle apart. As can be seen, it is not until the first cycle of $t = 4$ for the first channel to be completed. As demonstrated in Fig. 5.8, it is only after $\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \rceil$ tiles that Y complete 2D OUT matrices have been formed.

Similarly to the KERN components, a distinction is made between FGI and $Grad_{weight}$ convolutions. Considering the permutation and accumulation in $Grad_{weight}$ convolutions, the constructor component in Fig. 5.7 has FGI and $Grad_{weight}$ variations. The outputs from either version of the constructor component provide data in order of the desired result; $N_{INP} \times N_{KERN} \times H_{OUT} \times W_{OUT}$ and $N_{KERN} \times N_{INP} \times H_{OUT} \times W_{OUT}$ for FGI and $Grad_{weight}$ respectively.

So far it has been established that Y is used to provide multiple tensors across the N_{KERN} dimension when providing data to the SA. For FGI, N_{KERN} is related to C_{OUT} whereas for $Grad_{weight}$ N_{KERN} is related to N_{OUT}. Therefore in the FGI scenario, after $\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \rceil$ tiles there are 2D OUT matrices across Y consecutive tensors of the C_{OUT} dimension.

The $Grad_{weight}$ convolutions require accumulation and permutation (see Eq. 5.4) which must be performed before passing data to subsequent components. After $\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \rceil$ tiles there are 2D OUT matrices across Y consecutive tensors of the N_{OUT} dimension, each with only one element across the C_{OUT} dimension. Writing to off-chip memory cannot be initiated until the tiles for the final C_{OUT} channel is received from the SA. Due to the order of data arriving, an entire 4D OUT tensor worth of data is produced before any accumulation happens. Therefore the on-chip memory must support holding an entire 4D OUT tensor.

Additionally the constructor component filters the excess computed values caused due to $Y\%N_{KERN} \neq 0$ and $X\%\left(H_{OUT} \cdot W_{OUT}\right) \neq 0$ on-the-fly. This excess data is represented by

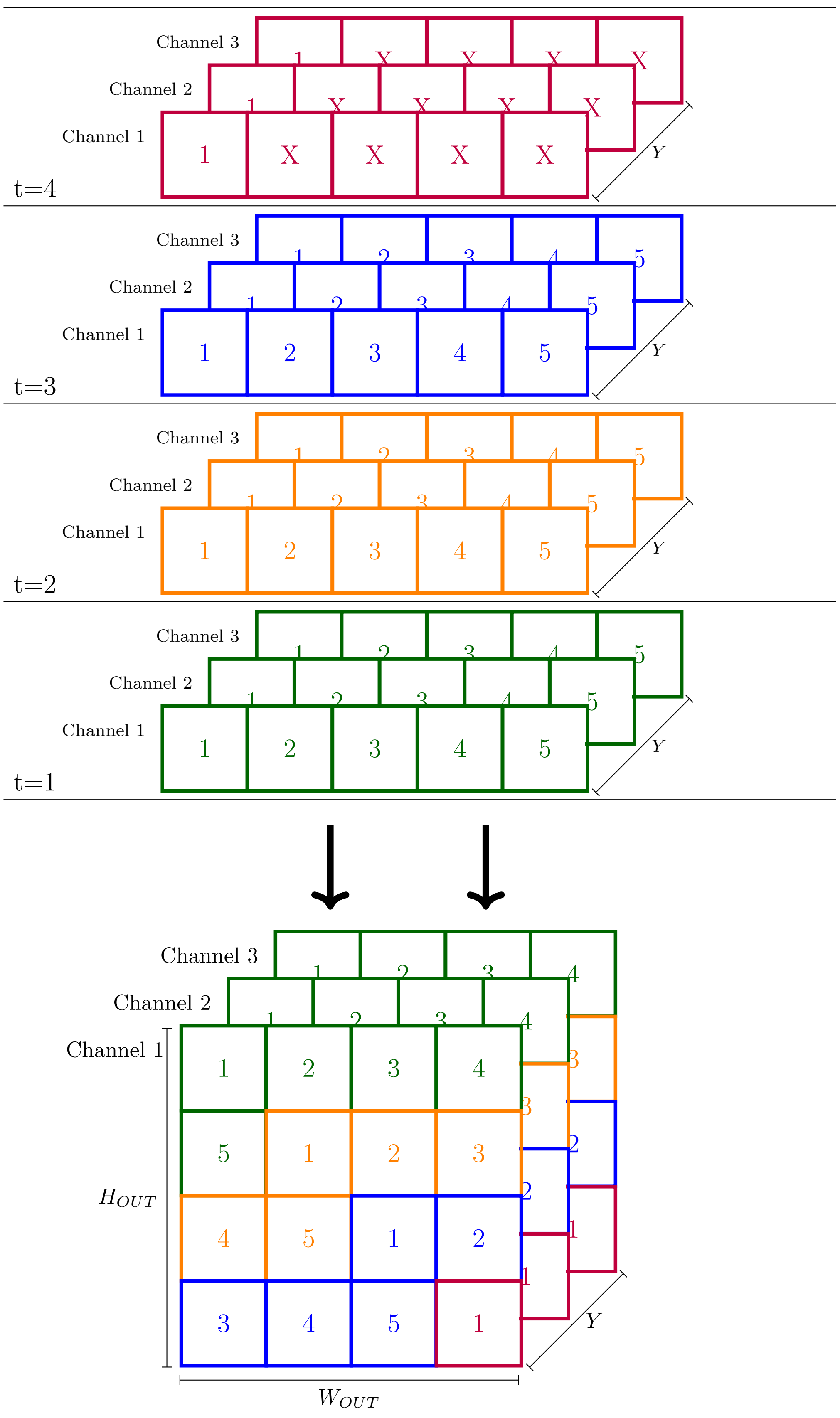


Figure 5.8: OUT data arrival where $X = 5$, $Y = 3$, $H_{OUT} = W_{OUT} = 4$

the 'X' in the purple tiles in Fig. 5.8. Please note this is not all excess data that can be produced. Producing the desired layout allows the remaining components to be identical regardless of FGI or $Grad_{weight}$ convolutions. The KERN memory subsystems introduces compile-time differences for FGI and $Grad_{weight}$. As a result, adding compile-time differences for the OUT memory subsystem imposes no notable extra burden on compilation.

All the data written to off-chip memory must be sequential to allow for burst reading to maximize performance. For FGI convolutions the OUT memory subsystem waits until complete 2D OUT matrices ($M \cdot \lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \rceil$ cycles, see Sec. 5.3.3) are present before handing them to the subsequent component. Consequently, Y complete 2D OUT matrices are stored on on-chip memory before being passed on. Similar to the FGI variation of the KERN SA Feeder, the limited memory footprint of the data makes storing on on-chip memory possible for most mid to large size FPGAs. For example, as most layers of GoogLeNet are 1x1 and 3x3 convolutions (in the forward pass), this means that the $Grad_{weight}$ OUT results are also 1x1s and 3x3s. Of course the FPGA on-chip memory must be large enough to store the required number of channels and filters. If this is too many, then both the FGI KERN and $Grad_{weight}$ OUT will not fit. To accommodate for this, as the filters are independent, the problem can be broken up along this parameter, but that functionality is not supported yet. For large scale FPGAs like the Xilinx U50, U250, U280 modern networks like GoogLeNet and SqueezeNet fit for all convolution layers.

OUT overall

The final OUT performance model is derived just like the INP performance model where the slowest component dominates the overall memory subsystem performance, due to the tight coupling and II=1:

$$\Gamma_{OUT} = \max\left(\Gamma_{OUT_{DDR}}, \Gamma_{OUT_{Encode}}, \Gamma_{OUT_{Constructor}}\right) \tag{5.12}$$

5.3.9 Total Performance

All components are run concurrently on the FPGA and designed to pass on any information they hold as soon as it becomes available. Furthermore, the entire design has every component tightly coupled to the next through FIFOs. This enables the design to move data through the stages of the architecture with minimal delay. Due to this design, the overall number of cycles of execution can be determined by the component in the hardware architecture that runs for the greatest number of cycles.

$$\Gamma_{Total} = \max(\Gamma_{INP}, \Gamma_{KERN}, \Gamma_{SA}, \Gamma_{OUT}) \tag{5.13}$$

5.3.10 Resource Constraints

This section discusses the resource modeling of the system. The resource model combines with the performance model to guide the design space exploration method. Throughout the design space exploration, the resource model identifies which X and Y values are viable for each layer. Additionally, it analyzes which sets of layer combinations can run on the same set of compile time parameters and still fit on the FPGA memory. The execution of these layers will only be differentiated by runtime parameters. The performance model then proceeds to determine which of the feasible configurations and layer combinations provide the best performance. Overall this allows the design space exploration to create a high performing and valid design.

Constraint 1: Memory requirements rsc_{mem}

The first constraint captures the architecture's on-chip memory requirements. It ensures that the proposed architecture for a chosen parameter configuration does not exceed the target device's Block RAM (BRAM) and Ultra RAM (URAM) allowance. Following up from the previous analysis, for each of the INP, KERN and OUT memory subsystems, only one component has any interaction with on-chip memory. This component is always the one connected to the systolic array. As the memories need to be instantiated during compilation, the sizes presented here will be the minimum memory required for a single layer. Over predicting the memory requirement has no negative effect, yet under predicting will render the entire design unusable and prevent successful compilation.

The analysis for each on-chip memory is split into two parts. The first is computing the blocks required per partition, followed by a computation of how many partitions required and the resultant total number of memory blocks required.

To enable parallel access to memory, the on-chip memory needs to be partitioned. Partitioning on-chip memory creates multiple sets of memory blocks, equal to the partitioning factor. Each set of memory blocks is independent of the others which enables the parallel access. This also implies, that if a memory block is assigned to one of the partition sets, it cannot be used by the other partition sets. In effect this means that even if only 1% of a memory block is used by a partition set, the rest of the block cannot be utilized by any other partition sets. As a result, the number of words per partition set *pmem* is computed first followed by the number of blocks per partition set represented by *util*. This number of blocks must be a whole number rounded up to ensure a block is completely assigned to a partition set.

$$pmem^{INP} = \text{align}(\text{align}(\text{align}(Z_{channel,\alpha}, BAND), Z), 2 \cdot Z) \tag{5.14}$$

$$pmem^{KERN} = \begin{cases} \text{align}(C_{KERN} \cdot H_{KERN} \cdot W_{KERN}, Y) & \text{if } GW \\ \lceil \frac{N_{KERN}}{Y} \rceil \cdot C_{KERN} \cdot H_{KERN} \cdot W_{KERN} & \text{if } GFI \end{cases} \tag{5.15}$$

$$pmem^{OUT} = \begin{cases} N_{OUT} \cdot C_{OUT} \cdot \lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \rceil & \text{if } GW \\ Y \cdot \lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \rceil & \text{if } GFI \end{cases} \tag{5.16}$$

Using the amount of words in each partitioned memory the total number of blocks is computed as follows:

$$util^{d}_{RAM} = \left\lceil \frac{pmem^{d}}{WPB_{RAM}} \right\rceil, d \in \{INP, KERN, OUT\}, RAM \in \{BRAM, URAM\} \tag{5.17}$$

where WPB_{RAM} is words per block for a specific RAM. Note that this value is rounded up because as soon as a block is even partially filled, it must fully dedicated to a partition set.

Using this information the total amount of blocks of RAM is as follows.

$$mblocks^{INP}_{RAM} = util^{INP}_{RAM} \cdot 2, RAM \in \{BRAM, URAM\} \tag{5.18}$$

$$mblocks^{KERN}_{RAM} = \begin{cases} util^{KERN}_{RAM} \cdot 2 \cdot Y^2 & \text{if } GW \\ util^{KERN}_{RAM} \cdot Y & \text{if } GFI \end{cases}, RAM \in \{BRAM, URAM\} \tag{5.19}$$

$$mblocks^{OUT}_{RAM} = \begin{cases} util^{OUT}_{RAM} \cdot X & \text{if } GW \\ util^{OUT}_{RAM} \cdot 2 \cdot X & \text{if } GFI \end{cases}, RAM \in \{BRAM, URAM\} \tag{5.20}$$

The number of blocks for INP is the computed number of blocks ($util^{INP}_{RAM}$), but needs to be double to account for double buffering. For KERN in the $Grad_{weight}$ scenario, data is accessed in Y values at a time, but as discussed, data is accessed in a different order than how it is brought in, requiring the Y^2 partitioning. Furthermore, it is double buffered, explaining the $\times 2$. For KERN in the FGI scenario, data is read in once and remains in memory throughout execution, being read out Y values at a time, requiring Y partitioning. For OUT in the $Grad_{weight}$ scenario the explanation, is very similar to KERN for FGI. This is reflected in the discussion of the OUT Constructor component. For OUT in the FGI scenario, it is again double buffered and accessed X values a time, resulting in X partitioning.

Apart from the storage capacity, to support the degree of parallelism required by the design, the on-chip RAM must provide Z ports for INP, Y ports for KERN and X for OUT. This is inherently covered by the number of partitions each design has. Furthermore, each block in RAM has a specific number of ports, and therefore the port requirement will create a block requirement as show in Eq. 5.21, where PPB is the ports per block.

$$pblocks^{d}_{RAM} = \left\lceil \frac{ports^{d}_{RAM}}{PPB_{RAM}} \right\rceil, d \in \{INP, KERN, OUT\} \tag{5.21}$$

The constraint Eq. 5.22 specifies the absolute limit on the size of the data that can be stored.

$$\sum_{d \in DA} \max\left(mblocks^d_{RAM}, pblocks^d_{RAM}\right) \cdot stored^d_{RAM} \leq totalBlock_{RAM},\ \ d \in \{INP, KERN, OUT\} \tag{5.22}$$

The *stored* variable refers to whether that data is stored on that specific RAM. The memory blocks required due to either ports or memory requirement must be less than the RAM blocks on the board, making memory consumption a hard constraint. X, Y and Z alter memory consumption and configuration, molding the design to fit in on-chip memory.

Constraint 2: rsc_{PE}

The second constraint addresses the maximum degree of parallelism that can be achieved, which is bounded by the number of FPGA DSP blocks. Even though each PE should utilize 3 DSP for multiplication and 1 for accumulation, empirical analysis shows on average 9 DSPs per PE are used. Due to HLS having the freedom to allocate/reallocate resources, it is hard to exactly breakdown where the extra 6 DSPs occur from. As a result, X, Y is limited to $9 \cdot (X \cdot Y) \leq DSP_{avail}$. We found that this limit varies between different versions of HLS, the problem size and the target FPGA. The impact of the MUX units and routing, possibly lowering the X, Y upper bound, is empirical and hard to predict and hence not included in this constraint.

5.4 Design Space Exploration

To meaningfully track the effect of various different configurations of the α parameter set, an accurate performance and resource model are required. Furthermore the entire architecture is designed for various convolutions to run on the same compiled design. The only requirement is that the dimensions of the target convolution fit within the compiled design. This allows for compiling designs with various target objectives such as: optimal performance (i.e. one design per convolution type per layer), minimal compilation effort (i.e. a single dsign for all convolution types and layers) or a trade-off between the two.

5.4.1 Optimal Performance per Convolution

One possible approach is ignoring compilation effort and building design implementations (α) that achieve the best possible performance for the target workload (β). The proposed parameterized architecture along with the resource and performance models identify optimal configurations for a target device and target workload. Given a network topology with convolution layers $\mathcal{L}$ which contain the total convolutions $\mathcal{C}$, a design space exploration (DSE) is performed on each convolution $c \in \mathcal{C}$. Each convolution's parameters are represented by β_c. The DSE uses β_c to find the α_c that produces the configuration with the lowest cycle count within the rsc_{mem} and rsc_{PE} resource constraints. Eq. 5.23 formalizes this layer specific DSE approach.

$$arg \min_{\alpha_c} \Gamma_{Total}(\alpha_c, \beta_c) \text{ s.t. } rsc_{mem}, rsc_{PE}, \forall c \in \mathcal{C} \tag{5.23}$$

For larger networks this does lead to an extremely large amount of compiled kernels. For example for GoogLeNet this would be 170 kernels, where each design's compilation times are estimated to be between 1-4 days per kernel. This long compilation time is due to the complexity of the design it self and the compilation process. The compilation process has two main stages, HLS to Verilog and Verilog to hardware. Although, the second phase is known to be computationally demanding, in our scenario, the first stage plays a significant role. The Vitis HLS compiler is still under development making it very demanding to compile such a large design that has added complexity to get HLS to produce an II=1 result. Additionally, all designs were compiled on a Intel Xeon Bronze 3104 CPU, which is a standard-tier CPU, especially for such a demanding compilation. This approach to generating kernel configurations can be implemented as a brute force approach as it is quite a simple and straightforward problem.

5.4.2 Limited Number of Kernels

Compiling new FPGA kernels is a slow process and a design of this level of complexity can require multiple days to compile. This amount of time provides incentive for convolutions to share a compiled kernel. To address this the long compile time, the proposed DSE supports an approach which reduces the number of kernels required to run the network. This is why the hardware architecture is designed with compile time and runtime parameters. If multiple convolutions can fit on the FPGA with the same compile time parameters, then runtime parameters can be used to still ensure workload specific execution. This allows multiple convolutions to share a single compiled bitstream while still adapting performance to the incoming workload.

The DSE aims to producing the minimal number of kernels for a network within a certain relative performance penalty with respect to individual kernel compilation. The relative penalty is set as a percentage deviation from the optimal performance:

$$penalty_{relative} = \frac{\Gamma_{Total}(\alpha_{group}, \beta_{group}) - \Gamma_{Total}(\alpha_{opt}, \beta_{opt})}{\Gamma_{Total}(\alpha_{opt}, \beta_{opt})} \tag{5.24}$$

where $\alpha_{opt}, \beta_{opt}$ are the optimal configurations for the target workload and $\alpha_{group}, \beta_{group}$ are the configurations for the target group the relative error is being computed towards.

Overall the DSE attempts to merge new convolutions $\mathcal{N}$ into the group of convolutions $\mathcal{G}$. The DSE process attempts to merge as many convolution groups while maintaining $penalty_{limit} <$ threshold. Here the threshold presents the user defined acceptable relative performance penalty per convolution. Alg. 1 breaks down the overall DSE process. The DSE creates a group for each convolution set to its optimal parameters ($kernels$). All groups are then sorted according to absolute performance. The slowest group becomes $\mathcal{N}$ and its elements merge with the group that causes the lowest change in $penalty_{relative}$ (lines 9-17). If multiple groups are possible, the group with the highest performance is selected (line 18-19). The resulting groups are sorted based on the number of elements in each group line line(20). The smallest group becomes $\mathcal{N}$ and its elements merge with the group that causes the lowest change in $penalty_{relative}$. Groups are split and distributed until the remaining groups can no longer be merged. Please note that

Algorithm 1: DSE

```
Input: kernels /* List of groups containing 1 convolution per group set to
    their optimal parameters */
1  $kernels = performance\_sort(kernels)$;  /* sort from worst to best performance */
2  $combine = True$;
3  while $combine$ do
4      $\mathcal{N} = kernels.pop()$;  /* take out the slowest/smallest group of convolutions */
5      $min\_set = None$;
6      $min\_n = None$;
7      $min\_idx = None$;
8      $combine = False$;
9      for $count, \mathcal{G}$ in $enumerate(kernels)$ do
10         $\mathcal{G}_{new}, \mathcal{N}_{new} = merge(\mathcal{G}, \mathcal{N})$;
11         if $\mathcal{G}_{new}.perf < min\_set.perf$ then
12             $min\_set = \mathcal{G}_{new}$;
13             $min\_n = \mathcal{N}_{new}$;
14             $min\_idx = count$;
15             $combine = True$;
16         end
17     end
18     $kernels[min\_idx] = min\_set$;
19     $kernels.append(min\_n)$;
20     $kernels = size\_sort(kernels)$;  /* sort from smallest to largest group */
21 end
22 return $kernels$
```

the above heuristic is not guaranteed to be optimal. This sacrifice of optimality is necessary due to the rapid growth of the exploration space and the non convex nature of the problem.

Algorithm 2: Merging convolution groups

```
Data: $\alpha$ /* List of compile time parameters */
Input: $\mathcal{G}$ /* List of existing convolution group to merge into */
Input: $\mathcal{N}$ /* List of new convolutions to merge */
1  $\mathcal{N}_{out} = []$;
2  for $n$ in $\mathcal{N}$ do
3      $\mathcal{G}_{tmp} = \mathcal{G}.clone()$;
4      for parameter in $\alpha$ do
5          $\mathcal{G}_{tmp}.parameters(parameter) = \max(n.parameters.(parameter), G.parameters.(parameter))$;
6      end
7      if test_fit($\mathcal{G}_{tmp}$) & test_penalty($\mathcal{G}_{tmp}$) then
8          $\mathcal{G}_{tmp}.convolutions.append(n)$;
9          $\mathcal{G} = \mathcal{G}_{tmp}.clone()$;
10     else
11         $\mathcal{N}_{out}.convolutions.append(n)$;
12     end
13 end
14 return $\mathcal{G}, \mathcal{N}_{out}$
```

Regarding the merge function on line 10, the pseudocode in Alg. 2 breaks down how the merging algorithm operates. To merge a new $n \in \mathcal{N}$ into $\mathcal{G}$, a new $\mathcal{G}_{tmp}$ is created where $elem_{\mathcal{G}_{tmp}} = \max(elem_{\mathcal{G}}, elem_n), \forall elem \in \alpha$ (lines 4-6). Additionally, $rsc_{mem}^{\mathcal{G}_{tmp}}$ and $rsc_{PE}^{\mathcal{G}_{tmp}}$ cannot violate their margins and $\max\limits_{\forall g \in \mathcal{G}_{tmp}} (penalty_{relative}^{g}) < penalty_{limit}$ (line 7). If these conditions are met $\mathcal{G} \leftarrow \mathcal{G}_{tmp}$. Finally after all elements of $\mathcal{N}$ have been tested, the remaining elements of $\mathcal{N}$ and the newly formed $\mathcal{G}$ are returned.

5.5 Overall Toolflow

The entire FPGPT ecosystem comes with many components and as a result works as an overall toolchain to implement CNN training on an FPGA. Fig. 5.9 shows how the DSE is provided with the user input including the network to target, and whether to target all or some of its layers. Furthermore the DSE requires the target FPGA device, which is used to infer the parallelism and memory according to the device limitations. Finally the user must decide to

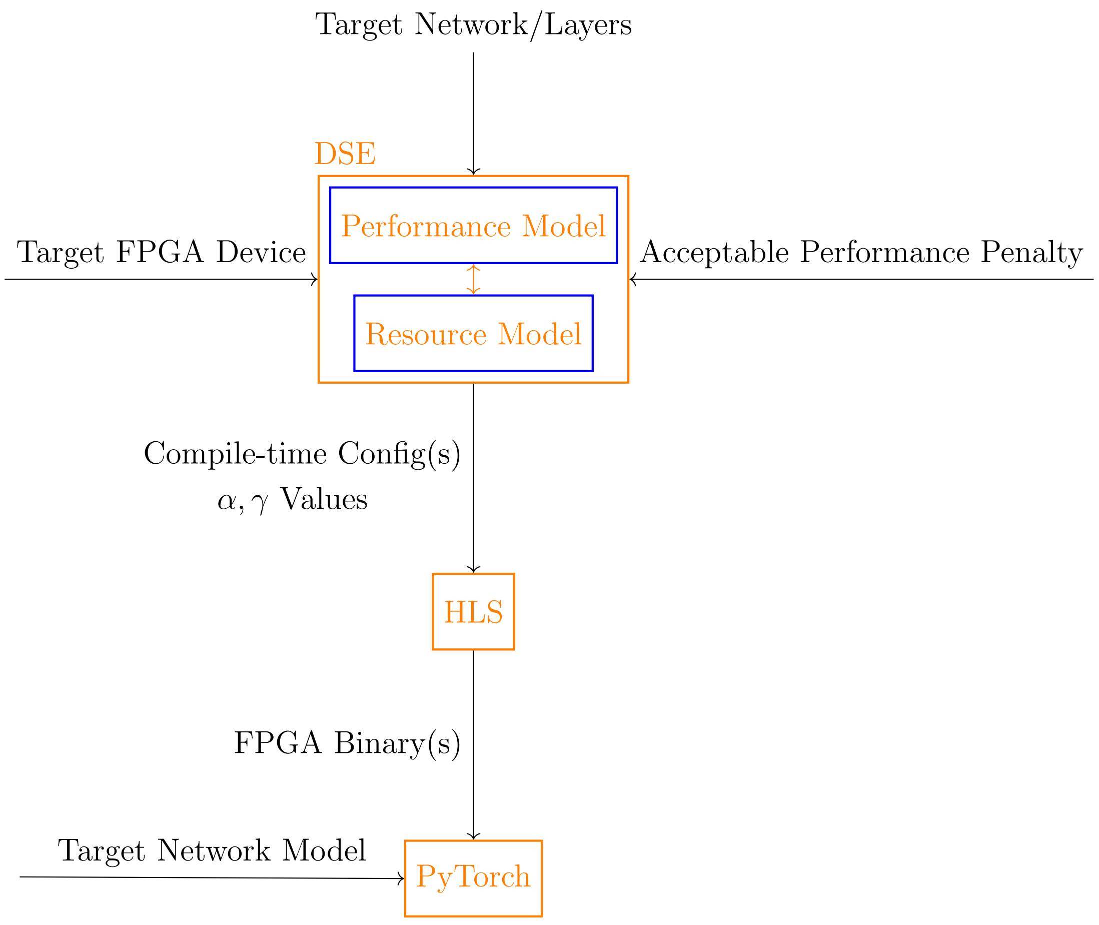


Figure 5.9: FPGPT Toolchain Information Flow

what degree they are willing to incur a performance penalty to reduce the number of designs to compile. The DSE then produces the compile-time config(s) that hold all α values which are used by Vitis HLS and a series of make files to generate the target hardware binaries. These binaries are then located in a directory for PyTorch to utilize. PyTorch is also provided with a modified network model, indicating the layers to be performed on the FPGA. Due to the integration into PyTorch, this takes minimal effort, as will be described in Sec. 5.5.1. From here on, PyTorch is able to run and produce a CNN trained on an FPGA.

5.5.1 Pytorch Framework Integration

PyTorch, and more specifically LibTorch (PyTorch's C++ API) has been selected as the framework for FPGPT. LibTorch allows for the integration of custom C++ units into the existing framework. Unlike other frameworks, LibTorch does not require any changes to its autograd source code to be able to run custom C++ layers and provides the ability to fully customize all aspects of the forward and backward pass. FPGPT has a custom convolution layer that replaces the LibTorch Conv2d class providing FPGA execution functionality. Each custom layer is called identically to the Conv2d module used by most SotA networks for CNN convolution layers. The native integration of the backward pass with the LibTorch autograd ensures seamless integration with all other layers and the LibTorch execution graph. For efficiency, the runtime variables are computed only once during model setup as they will not change during execution. During runtime, the custom C++ layer analyzes the current and required bitstream on the FPGA and reprograms only when necessary. Data is passed to the FPGA, the convolution is executed, the result is returned back to LibTorch and processing continues. Please note that this is directly compatible with the rest of the standard LibTorch library (PyTorch C++ API) and the custom nn::Conv2d as well as a functional execution template can be found at (https://github.com/DiederikVink/pytorch_cxx). Furthermore, these layers were written that they can also be integrated with the original PyTorch library which has a more mainstream appeal. For this work LibTorch was used as this made development easier and provided higher performance as everything operated in C++.

5.6 Evaluation

The performance of the proposed FPGPT framework, the quality of the performance and resources models, and the performance of the produced hardware are assessed in this section. All results are compared to state of the art (SotA) works in the field. Initially, the necessity and benefits obtained through the DSE process are demonstrated. This is followed by a comparison between the performance obtained by the hardware implementations produced by FPGPT to SotA FPGA works. Finally FPGPT is compared to a high-end GPU implementation.

5.6.1 Implementation Details

The target FPGA platform is a single Xilinx Alveo U250 Data Center Accelerator Card. The build configurations for the FPGA architecture are obtained through the DSE described in Sec. 5.4 for each convolution of the targeted network. The host machine for the FPGA contains an Intel Xeon Bronze 3104 CPU that runs the non-convolution layers of the CNN. The GPU results are collected on an NVIDIA RTX 2080Ti GPU running CuDNN 7.6. The batch sizes used for the evaluation on the FPGA are set to 256, while the GPU runs are fixed to 128 except for GoogLeNet which was fixed at 64, due to GPU memory limitations. Please note that this design can support larger batchsizes as well, limited only by the size of the available off-chip memory. Hardware compilation is performed through Vitis HLS 2022.1 with Vitis 2022.1, setting the target clock frequency to 100MHz. Depending on design configurations, the critical path will change for each design.

The bitstreams generated by FPGPT integrate with the PyTorch C++ library: LibTorch (see Sec. 5.5.1). Normally, to execute the training of a CNN on an FPGA, this requires either constantly reconfiguring the FPGA or distributing the compute across multiple FPGAs [1]. Constantly reconfiguring would result in an exceptionally large time penalty quickly negating any benefit provided by the proposed architecture. Furthermore moving data around in a naive implementation would definitely be suboptimal compared to existing frameworks. The alternative is distributed hardware, but this requires expensive hardware infrastructure that is prohibitive for most researchers. To mitigate these issues, FPGPT directly interfaces with PyTorch data structures as they appear in standard training. This ensure efficient data management, amortized reconfiguration costs (during non-convolution layers) and can operate efficiently on only one FPGA. Although the primary target if PyTorch, FPGPT should seamlessly integrate with TensorFlow and Caffe as well. The justification behind this is that as FPGPT replaces a Cuda kernel, and this same kernel is used in TensorFlow and Caffe, no integration problems should occur.

Please note the large compilation time of any FPGA design before training CNNs. Uniquely for FPGPT the cost creating a hardware kernel is a one-off. Due to the hardware kernel's runtime programmability it can be employed in multiple training scenarios. As the FPGPT architecture is used more, a library of kernels can be accumulated reducing the need to compile as the chance of having an appropriate kernel is increasingly great.

In the rest of this section, all graphs are presented for GoogLeNet run on ImageNet to provide concise and coherent analysis of the proposed architecture. The conclusions drawn from this data generalize across all layers of GoogLeNet and all the networks presented in Tab. 5.8. Unless stated otherwise, all results originate from the performance and resource models to provide a more extensive and detailed analysis of performance. To ensure all conclusions about FPGPT are valid, the models have been validated as described in Sec. 5.6.2.

5.6.2 Model Analysis

The performance model described in Sec. 5.3 is used for most of the analysis to come. Both the performance and resource models are analyzed for their validity. This is crucial to ensuring the validity of any ensuing analysis of the performance of FPGPT. To make the comparison to other works as direct as possible, the model includes the time taken to move data between the host and the FPGA. The time taken to move the data is based on a multiple profiled runs to increase accuracy.

For both the performance and resource models, beyond just accuracy, they must guarantee to

Convolution	Mean	St.dev	Max	Min
Forward	8.19%	6.21%	15.0%	0.229%
$Grad_{IFM}$	9.35%	5.80%	15.1%	0.294%
$Grad_{weight}$	5.47%	2.66%	15.8%	0.002%

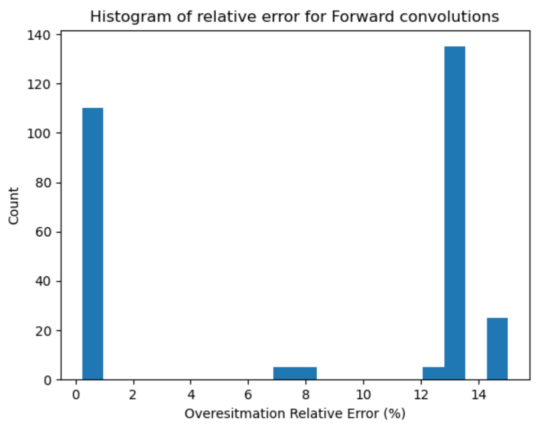
Table 5.4: Performance model runtime underestimation as relative error across all convolutions in GoogLeNet

never underpredict results. For the resource model this is essential a underpredicting resources can lead to suggested designs that will not compile, rending the entire DSE result useless. For the performance model, underpredicting results would invalidate any analysis based on the performance model. To ensure the validity of any claims and conclusions made about the FPGPT design as well as the DSE, the performance and resource models have been analyzed and validated.

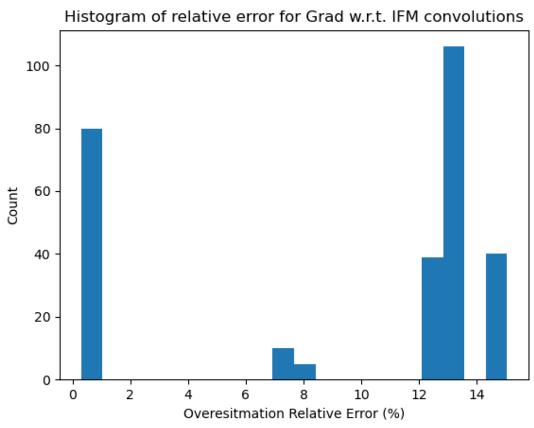
Performance Model Validation

To properly verify the accuracy of the performance model, 9 kernels of various configurations were built and run with real CNN training workloads from the GoogLeNet architecture. A total of 170 convolutions were tested in excess of 5 times across those 9 kernels, with an even balance of Forward, $Grad_{input}$ and $Grad_{weight}$ convolutions. These tests were performed by profiling the hardware as it ran multiple successful training iterations of FPGPT. The β values (see Sec. 5.3) of these 170 convolutions were varied to ensure the model accurately captured the latency changes introduced by varying these parameters. Furthermore, across the 9 configurations, the values of X and Y ranged from 1-11, with the overall degree of parallelism varying around 32 PEs. Higher degrees of parallelism were avoided to keep compilation times realistic, attempting to stay under 16 hours per compilation.

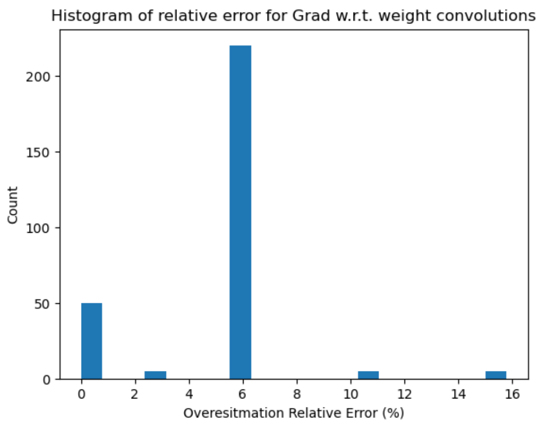
As shown by Tab. 5.4 the model overestimated runtimes up to 15% more than the actual runtime, with an average error of 8.19%, 9.35%, and 5.47% for Forward, $Grad_{input}$ and $Grad_{weight}$ respectively. These values are relative errors computed as $\frac{model-empirical}{empirial}$ where *empirical* is the runtime values and *model* is modeled value. It must be noted that all performance estimates underestimate the actual performance, meaning that all actual runtimes will be lower in reality.



(a) Forward direction



(b) $Grad_{IFM}$ direction



(c) $Grad_{weight}$ direction

Figure 5.10: Histogram of the prediction error of the performance model across all convolutions in GoogLeNet

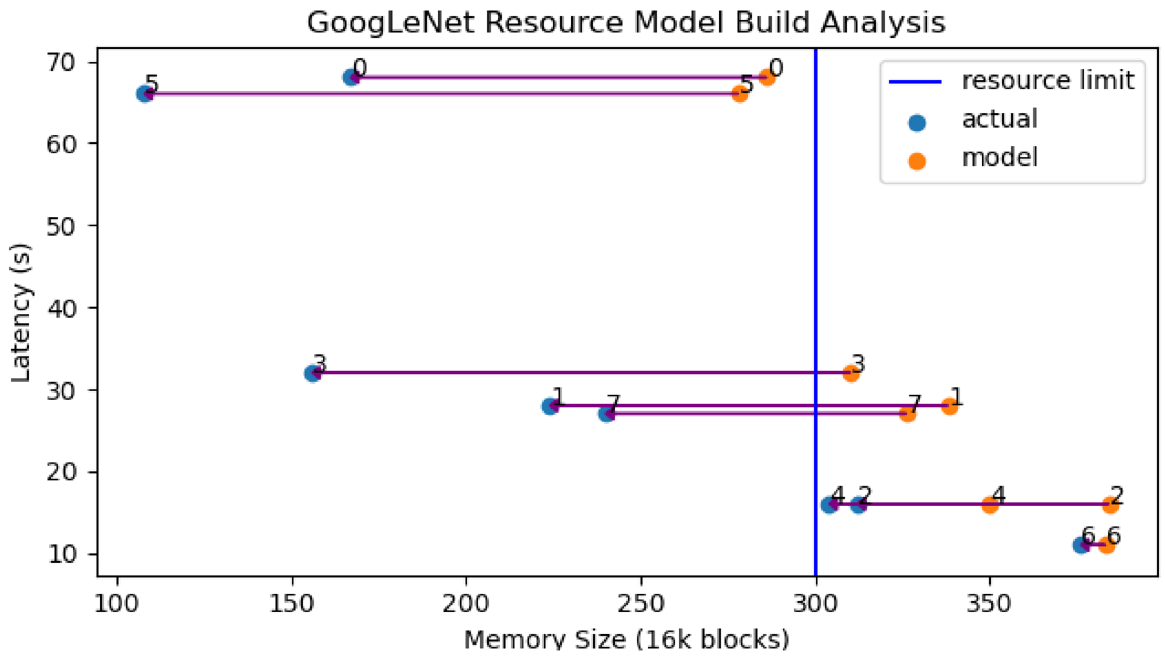


Figure 5.11: Modeled vs Actual memory block allocation relative to a resource threshold for GoogLeNet

The same data represented in Tab. 5.4 is demonstrated in Fig. 5.10. The data is broken up for each type of convolution. Due to the highly similar hardware architecture and workload dimensions for the Forward and $Grad_{IFM}$, the overestimation seems to exhibit itself very similarly for both convolution types. For $Grad_{weight}$, it seems that the error is far more concentrated, as confirmed by the low standard deviation shown in Tab. 5.4. This would be reasonably explained by the entirely different execution structure in the KERN and OUT components, leading to potentially different sources of model error.

Resource Model Validation

In order to assess the accuracy of the resource model, the same 9 kernels were inspected and compared to the resource model. For all cases, the model overestimated the amount of on-chip memory utilized by the actual hardware. The main reason for this is due to the fact that Vitis HLS will attempt to place a memory on the fabric directly if possible as this improves the maximum achievable frequency. As a result, the designs will always build, but due to overestimation of memory consumption certain designs are never considered by the DSE. As

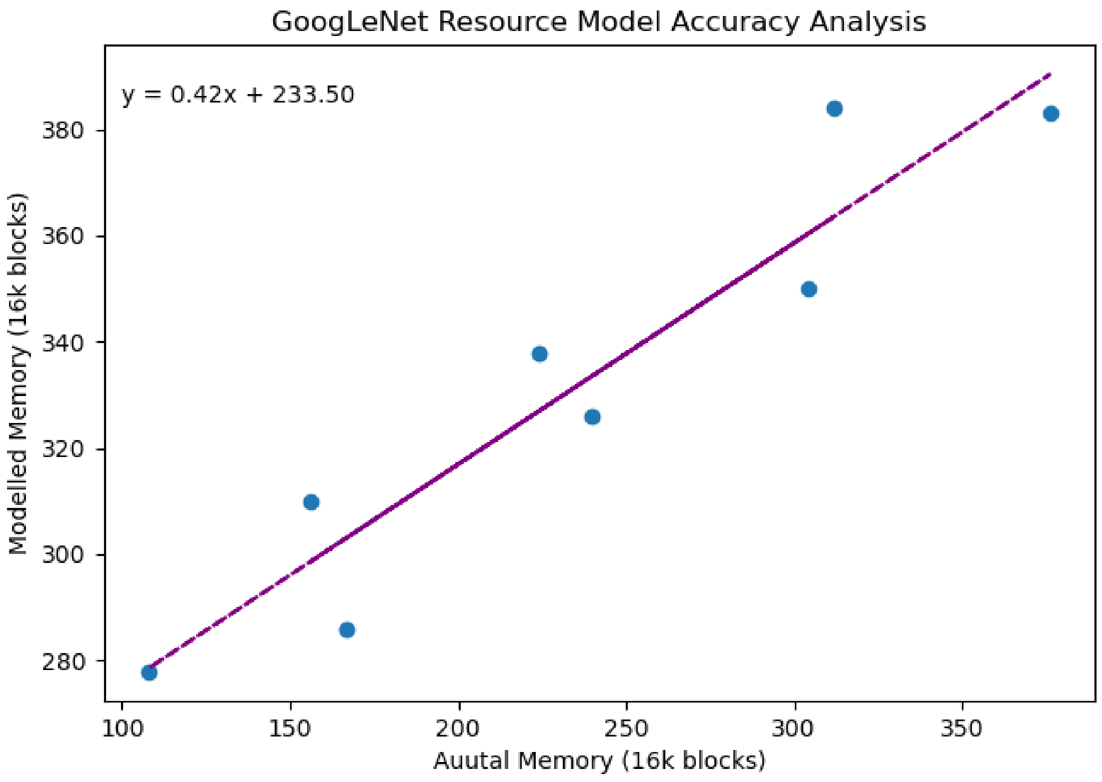


Figure 5.12: Modeled vs Actual memory block allocation for GoogLeNet

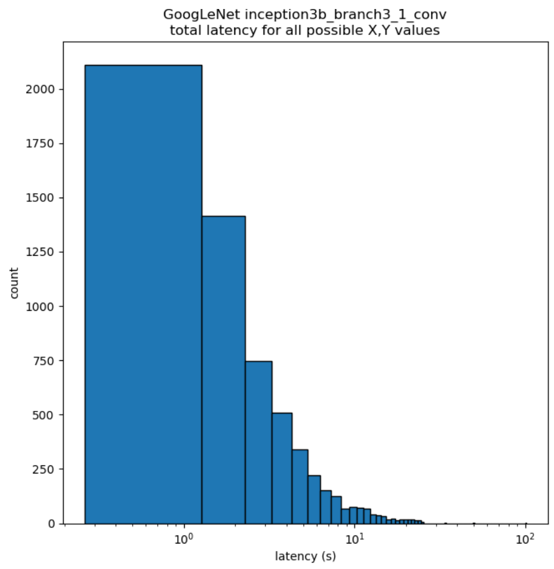
a result, potential higher performance units are never built as demonstrated by Fig. 5.11. In Fig. 5.11, 8 kernels were built and their actual BRAM memory utilization (blue) was compared to the model prediction (orange). Fig. 5.11 assumes a hypothetical scenario with a device that is limited to 300 memory blocks represented by the vertical blue line. In this hypothetical, designs 0,1,3,5 and 7 would be able to fit. Designs 2,4 and 6 were too large to fit. Yet when using the resource model (orange dots) only designs 0 and 5 would have made it past the resource model and be considered as feasible designs whereas 1, 3 and 7 would have been discarded.. Designs 0 and 5 have a more than twice the latency compared 1,3 and 7, demonstrating how high performance kernels are being discarded due to resource modeling inaccuracies. This indicates that to ensure every configuration produced by the DSE is feasible, potentially higher performance configurations are not proposed as they are considered too large to be implemented.

Fig. 5.12 directly compares the prediction of the actual memory block consumption versus the modeled memory block consumption. Clearly there is both a fixed offset as well as a relative error in the model. If more models are built, the degree and type or error for the model could be inferred, and this analysis could be used to improve the quality of the memory model. It is critically important to keep in mind that the memory model may never underestimate the amount of memory required, as this will prevent successful compilation, breaking the flow of the overall FPGPT ecosystem.

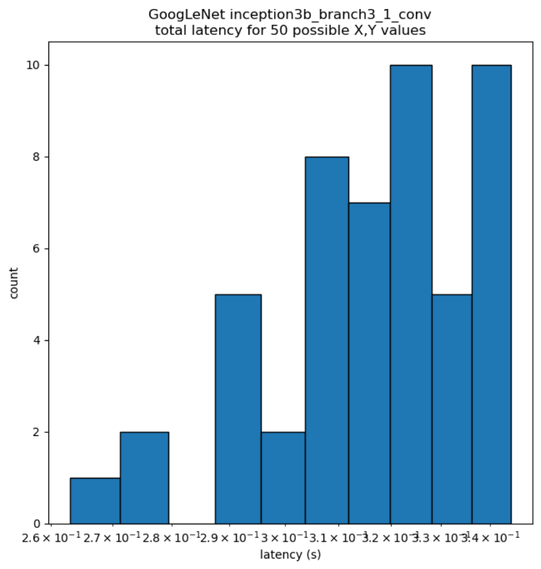
DSE Impact Analysis

Fig. 5.13 depicts the distribution of latencies for all three convolutions seen in the forward and backward pass. These latencies are produced by the performance model on designs with X, Y values verified to acceptable by the resource model. GoogLeNet Inception 3b branch3-1 is used as the target layer and the combinations of X and Y span a range of parallelism between 4-4096 PEs. Fig. 5.13a, 5.13c and 5.13e demonstrate how large the range of latencies across all possible X, Y values are. In total there are 12424 different possible X, Y combinations, with performance ranging from 0.26 seconds to $>$ 50 seconds. These figures demonstrate that selecting the optimal X, Y value cannot just be selected at random due to the range of latencies. Fig. 5.13b, 5.13d and 5.13f show the performance of the top 50 best performing X, Y values. All three figures provide insight into the variation among the highest performing X, Y configurations. In all three figures, it can be seen that even within this highly limited subset, randomly selecting a subset of kernels will most likely not result in an optimal X, Y combination. To identify high performing X, Y configuration the performance model must be employed during the DSE.

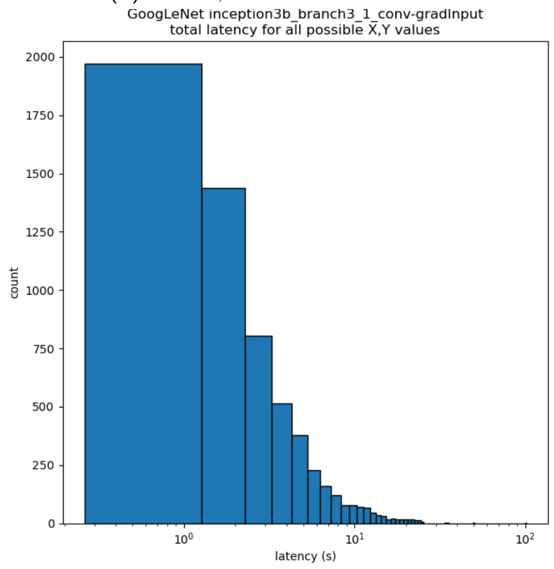
As demonstrated there are a very small number of absolute top performing kernels to get maximal performance. To further underline the need for a DSE, it can be shown merely picking the highest degrees of parallelism does not lead to the best performance. This is enforced by the fact that the top 10 highest performing kernels have a parallelism varying from 530-737 PEs. This implies that there are builds with up to 4096 PEs that have a performance outside of the top 10. This further demonstrates that the highest degree of parallelism does not guarantee top performance. This is because very large degrees of parallelism can result in a large amount



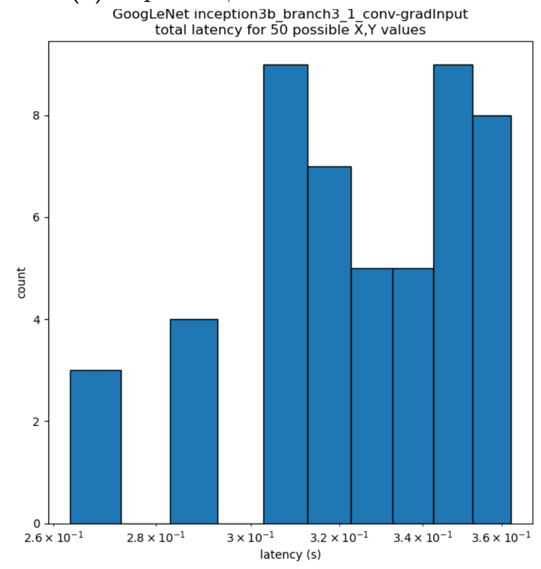
(a) All X, Y values: Forward



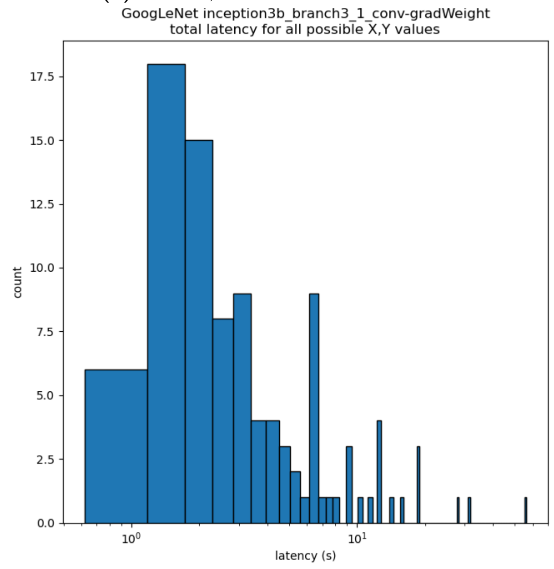
(b) Top 50 X, Y values: Forward



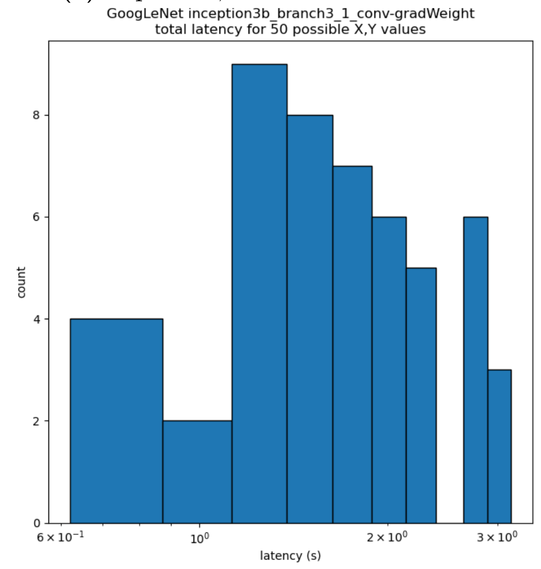
(c) All X, Y values: $Grad_{IFM}$



(d) Top 50 X, Y values: $Grad_{IFM}$



(e) All X, Y values: $Grad_{weight}$



(f) Top 50 X, Y values: $Grad_{weight}$

Figure 5.13: Histogram of latency for X and Y values for all convolutions involved in the inception 3b branch3-1 GoogLeNet layer

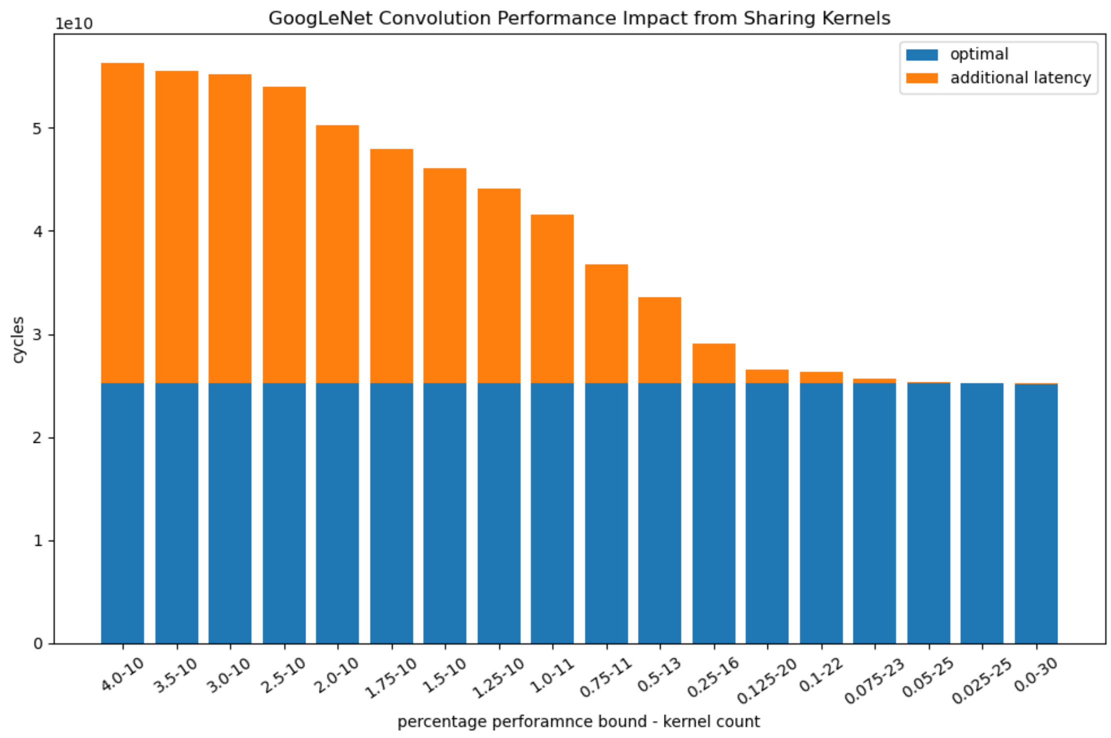


Figure 5.14: GoogLeNet convolution performance for various numbers of kernels

of waste computation which can result in performance degradation. All of this reinforces the need for the DSE to find the optimal X, Y configuration.

Kernel Sharing Analysis

Fig. 5.14 demonstrates the impact of the penalty incurred by having convolutions running on a shared hardware implementation. The x-axis in the graph shows (performance penalty - number of kernels). This represents the performance penalty provided to the DSE and the minimum number of kernels generated by the DSE while adhering to all restrictions. Each bar represents performance of the best X, Y combinations produced by the DSE for this scenario. The blue section of the bar shows the optimal latency. This represents the performance if each convolution is assigned its own kernel, which is why this value is constant throughout. To have one hardware kernel per convolution, 170 compilations are required. The orange part of the bar demonstrates the penalty incurred by running on a suboptimal, multiple convolution kernel.

Fig. 5.14 shows the ability to efficiently run multiple layers of the proposed architecture on the same FPGA. Only 30 hardware configurations are required to incur no performance penalty. Furthermore, if compiled with foresight, these kernels could be used for different networks or datasets. This could lead to building a library of precompiled binaries for a range of kernels, each with different α sets. Using the resource and performance models, viable kernels in terms of meeting performance and resource constraints could be filtered out. These kernels would then allow the user to perform their desired training without needing to compile any kernels. Even if not all kernels can be accommodated through the library, the compilation would still be notably reduced. The benefit of the library is why a large amount of runtime configuration is crucial to the design.

5.6.3 FPGA Performance Comparison

The performance of FPGPT shown here is based entirely on the performance model presented in this chapter (Sec. 5.3.9). The verification of the model, the high degree of accuracy and the fact that training has been successfully performed using FPGPT are the underlying assurances that the model can be used to validly represent the FPGPT performance discussed in the rest of this chapter. As the model only predicts the time spent on convolution, the time spent on other layers has been profiled from real PyTorch runs of the network. The PyTorch autograd profiler was used to perform all PyTorch profiling. All profiling code has been included with the open source version of this project.

For all analyses, the metrics presented are based on those presented in the works themselves. FPGPT is compared on a like-for-like basis to ensure the fairest comparisons. The results are presented through works with similar metrics to ensure clarity.

Custom FPGA Kernels

Tab. 5.5 presents the performance obtained by SotA works for FPGA CNN training. The metrics presented here are representative of the information provided by the works themselves. The values highlighted in bold are the highest performing values. These presented works

	AlexNet CIFAR-10			**AlexNet** ImageNet		
	$GOPS$	$\frac{GOPS}{w}$	$\frac{GOPS}{DSP}$	$GOPS$	$\frac{GOPS}{w}$	$\frac{GOPS}{DSP}$
FPDeep [1]	**1157**	**37.1**	**0.402**	-	-	-
EF-Train [73]	-	-	-	138.08	17.8	0.09
FPGPT	685	22.8	0.351	**752**	**25.1**	**0.245**

Table 5.5: FPGA-based CNN training approaches running on simpler networks. Uses FPGA cluster, data scaled to be per FPGA for FPDeep [1]

specifically focus on creating FPGA-based computation units that allow for CNN training. All performance results for FPGPT are based on employing the best performing implementations for that network, not concerned with the number of kernels required.

Tab. 5.5 analyses three key metrics presented in both FPDeep [1] and EF-Train [73]. The $GOPS$ metric represents the peak performance of the work across the target network in GOPS. The $\frac{GOPS}{w}$ metric represents the efficiency of the design considering highest energy consumption, i.e. peak $GOPS$ per watt. Finally $\frac{GOPS}{DSP}$, just like $\frac{GOPS}{w}$, reports the DSP efficacy based on peak performance. The DSP count utilized for this analysis is based on the resource count of the presented hardware.

Considering CIFAR-10, this is a network that is not commonly supported by modern day ML practitioners. The networks contains smaller 32x32 images, a smaller dataset size of 60,000 images and only 10 classes. The ML community has now shifted its focus to ImageNet. A commonly used variant, and the one used for FPGPT is ILSCVRC2017 which consists of 1.2 million images of size 254x254 with 1000 classes. Clearly ImageNet is a more complex dataset that requires significantly more processing and memory capabilities to execute properly and effectively.

When comparing to FPDeep, only CIFAR-10 can be considered as this is the only dataset that FPDeep [1] supports. This is due to the fact that FPDeep puts the entirety of its training on its distributed FPGA cluster so larger datasets would require more memory than the current system can provide. In this scenario FPGPT is at about 50% peak performance compared to FPDeep [1] and also outperformed in $\frac{GOPS}{w}$. Please note that the FPDeep is designed to perform on a cluster of 15 FPGAs. The performance in GOPS are estimates per FPGA assuming a cluster of 15 FPGAS. FPDeep demonstrates the benefit of putting the entire network on a cluster of FPGAs. This allows for reduced resource constraints optimizing for every layer on an FPGA. It also indicates that the proposed architecture demands at minimum quite an extensive infrastructure to operate which is not commonly available to most users. This removes suffering a performance penalty incurred by running layer(s) on a CPU.

Furthermore, FPDeep [1] adopts a 16-bit fixed point training scheme as opposed to FPGPT which runs 32-bit floating point. Moving to reduced precision demands simpler and less hardware, decreasing the design complexity. Furthermore going to 16-bits lowers the memory requirement of the design. When using standard reduced precision training with no techniques to counteract the error introduce by quantization like MuPPET, the convergence rate and final accuracy of a network are decreased compared to FP32 [102]. Even with this memory saving technique, the ImageNet memory footprint was still too large for FPDeep to host across 15 FGPAs, a limitation FPGPT surpassed. FPGPT can easily be adapted to run at fixed precision as it was written in HLS so the conversion would be trivial. Nonetheless, FPGPT is not optimized for fixed point execution, and a separate, lightweight, optimized design should be created for those training scenarios.

Moving the computation of all layers to the FPGA is an approach adopted by a number of current methodologies [1, 15, 63]. This methodology removes the expensive transfer of data too and from the host CPU. More importantly, it allows for the acceleration of all layers, not just convolutions. However, this requires a significant amount of memory storage as all activations are stored locally to be used for the backward pass. As such large datasets or input size like those seen in the case of ImageNet, cannot be used. The current approaches to address the extra required memory are to limit the size of network and dataset (i.e. using CIFAR-10), reduce the operating precision or make use of a cluster of FPGAs. Both these approaches are undesirable or non-applicable in many cases. Limiting the size of the network and dataset as well as reducing precision decrease the final network accuracy. Installing a cluster of FPGAs is a prohibitive infrastructure and as shown by FPDeep [1] 15 FPGAs is not enough to support ImageNet. FPGPT transfers all the layer results back to the CPU and runs non-convolution layers on the CPU. This lifts the above restriction but incurs the aforementioned data movement penalty in performance. This data movement penalty as well as the large ImageNet memory footprint motivated implementing im2col to decrease the INP memory.

When looking at the dataset more widely accepted by the ML community, ImageNet, FPGPT compares to the state of the art work EF-train [73]. Looking at Tab. 5.5, with regards to peak GOPS, FPGPT outperforms EF-train [73] by > 5.4 times. Furthermore, for all other metrics, FPGPT provides better power and resource efficiency. When looking at the overall of performance of FPGPT, the average GOPS drops to 384. Overall performance refers to the average GOPS of FPGPT across all layers in the network which includes the layers run on the CPU. This demonstrates a more realistic performance metric, not just peak performance at a specific point in training. As the CPU does not execute the layer workload in parallel there is a notable impact on the average GOPS. Considering the whole network training performance, FPGPT still comfortably outperforms the peak performance of EF-train [73]. This shows that when employed in a real world scenario, FPGPT will perform competitively when compared to the state of the art EF-train [73]. A limitation with EF-train [73] is that it does not provide performance results for more modern networks such as GoogLeNet and SqueezeNet.

ML Framework Integration

Section 5.6.3 discussed the state of the art networks that proposed CNN specific kernels for execution. This section takes a look at state of the art works that integrate FPGAs into machine learning frameworks. FPGPT guarantees to natively interact with ML frameworks and benefit from the basic optimizations provided by the framework itself. The focus for FPGPT is not on improving performance through custom ML framework operations, so the integration was solely focused on seamless integration of the kernels into the training process. Due to FPGPT being, to the best of our knowledge, the first work that addresses both facets this analysis has been separated from the architecture focused works.

FeCaffe [74] is a work that optimized the integration of FPGAs into the Caffe machine learning framework. Unlike FPDeep [1] and EF-train [73], FeCaffe is able to train modern complex networks on ImageNet. As only $\frac{Hours}{Epoch}$ were provided by FeCaffe [74], this is used as the reference

	AlexNet ImageNet	**GoogLeNet** ImageNet	**SqueezeNet** ImageNet
	$\frac{Hours}{Epoch}$	$\frac{Hours}{Epoch}$	$\frac{Hours}{Epoch}$
FeCaffe [74]	86.41	291.08	159.62
FPGPT	**38.0**	**212.42**	**143.35**

Table 5.6: FPGA-based CNN training approaches running on complex modern networks

metric in Tab. 5.6. For AlexNet, FPGPT is able to outperform FeCaffe [74]. AlexNet is only a small, 5 layer network and therefore not considered a modern competitive network. For the more important networks, FPGPT is comfortably considered competitive as it is 27% and 10% faster for GoogLeNet and SqueezeNet respectively.

FeCaffe [74] is a framework focused work that optimizes host-side execution of CNN training. The authors did not put any specific work into optimizing the FPGA execution code of any of the layers, directly adopting the existing OpenCL code. FPGPT conversely only optimizes convolution layers. FPGPT has been painstakingly designed to have the ability to operate on any ML frameworks' native data structures. This provides the opportunity for the FPGPT architecture to serve as the convolution layer in FeCaffe [74]. FeCaffe [74] would still offload all other layers as it is currently doing. This combination of works would produce a higher performance hybrid that would outperform either individual work.

5.6.4 GPU Performance Comparison

Currently GPUs are the primary method used for training CNNs. They provide a high degree of available parallelism and are natively integrated into every ML framework. GPUs can be instantly deployed and provide a very high degree of acceleration, especially for large problem sizes. As networks, and therefore the convolutions, get bigger, the advantage provided by GPUs grows accordingly.

Tab. 5.7 compares an NVIDIA RTX 2080Ti GPU to the FPGPT architecture. All GPU results are acquired through the PyTorch autograd profiler or from NVIDIA documentation. Furthermore, in this table, *GOPS* refers to the average performance for all layers throughout the network. *Peak GOPS* refers to the maximum theoretical *GOPS* that can be achieved by the

		GPU	**FPGPT**
AlexNet CIFAR-10	*Peak GOPS*	**13450**	685
	Conv GOPS	**1809**	27.4
	GOPS	**689**	22.3
	$\frac{GOPS}{w}$	0.305	**22.8**
	$\frac{GOPS}{DSP}$	0.022	**0.351**
AlexNet ImageNet	*Peak GOPS*	**13450**	752
	Conv GOPS	**13357**	400
	GOPS	**8798**	384
	$\frac{GOPS}{w}$	7.73	**25.1**
	$\frac{GOPS}{DSP}$	**0.553**	0.245
ResNet-20 CIFAR-10	*Peak GOPS*	**13450**	570
	Conv GOPS	**420**	55.3
	GOPS	**196**	20.9
	$\frac{GOPS}{w}$	0.688	**19.0**
	$\frac{GOPS}{DSP}$	0.05	**0.217**
GoogLeNet ImageNet	*Peak GOPS*	**13450**	643
	Conv GOPS	**2592**	156
	GOPS	**1252**	146
	$\frac{GOPS}{w}$	0.683	**21.4**
	$\frac{GOPS}{DSP}$	0.05	**0.277**
SqueezeNet ImageNet	*Peak GOPS*	**13450**	685
	Conv GOPS	**7454**	110
	GOPS	**3252**	106
	$\frac{GOPS}{w}$	12.0	**22.8**
	$\frac{GOPS}{DSP}$	**0.875**	0.300

Table 5.7: GPU vs. FPGA-based CNN training approaches. Peak GOPS for GPU are theoretical GPU maximum GOPS

	FPGPT (hw kernels)
AlexNet CIFAR-10	7
AlexNet ImageNet	9
ResNet-20 CIFAR-10	17
GoogLeNet ImageNet	30
SquuezeNet ImageNet	22

Table 5.8: Numer of hardware kernels needed per network for produced results.

hardware in question (GPU or FPGA). Tab. 5.7 has the same *Peak GOPS* value for the GPU for all networks at 13.45TOPS is the theoretical peak that can be achieved by the 2080Ti. For GPU runs average *GOPS* includes layers that perform less optimally like Pooling, BatchNorm and ReLU layers. For the FPGPT runs, the average *GOPS* metric includes running all layers that are not convolutions on the CPU. Finally, *Conv GOPS* represents the performance when only considering the convolution layers. This demonstrates the raw potential of FPGPT excluding the external impact of running other layers on the GPU. It becomes clear there is quite a gap between peak and average GOPS as the peak GOPS is not equally representative of the average performance across the networks for either the GPU or FPGPT.

As shown by Tab. 5.7 the GPU performs better in terms of network performance on ImageNet AlexNet than it does on CIFAR-10 AlexNet across all metrics. This is because when the workload consists of smaller problem sizes the GPU is underutilized. This is best demonstrated by looking at the differences between the *Peak GOPS* and *Conv GOPS* metrics. When analyzing AlexNet, the difference between *Peak GOPS* and *Conv GOPS* barely changes for ImageNet, demonstrating that the GPU being utilized almost as effectively as possible. On the other hand, when considering CIFAR-10, the *Conv GOPS* seems to only operate at 10% of *Peak GOPS*. FPGPT similarly performs notably better on large datasets which create larger workloads. This is partially explained through large workloads increasing the opportunity for amortizing delays.

Among the networks that operate on ImageNet, AlexNet-ImageNet is the highest performing network presented for both the GPU and FPGPT. The reason for this is that it contains primarily convolutions which are all very large in workload and consist of a large convolution kernel. Consequently, each individual convolution provides a large opportunity to amortize key delay points. GoogLeNet and SqueezeNet support the inception and fire modules respectively. Both of these modules include a large number of 1x1 convolutions. These 1x1 convolutions require custom handling as all amortization does not take place due to how small the problem is. Consequently, these networks are not able to perform anywhere near the peak performance offered by GPU. Another justification for this is looking at the $\frac{GOPS}{DSP}$ metric. GoogLeNet on the other hand is in the same range of utilization as the CIFAR-10 networks at < 0.05. This demonstrates how detrimental these 1x1 convolutions are for a fixed multi-core processors like a GPU.

FPGPT contrarily is not punished as extremely by these multiple smaller layers. Additionally ,the 1x1 convolutions are less impactful that general small layers, as the GoogLeNet results still indicate higher performance for *Conv GOPS* and *GOPS*. This is because FPGPT utilizes both compile- and runtime configuration to adapt to the convolution at hand and provide the best hardware for the target workload.

It must be noted that for FPGPT the *Peak GOPS* is quite stable. This is for the same reason the impact of very small and 1x1 convolutions are less detrimental to the performance. FPGPTs ability to adapt to the workload at hand makes it more universally applicable and appropriate.

In terms of $\frac{GOPS}{w}$, the power efficiency of the FPGA has allowed FPGPT to outperform the GPU. This is an expected result, as the FPGA design will only utilize the required resources and power those, running at low power. The GPU on the other hand is far less capable of saving power by shutting down non-utilized components, consistently running at higher power. This is further reinforced when looking at the $\frac{GOPS}{w}$ for the low workload CIFAR-10 networks. Here the efficiency is dramatically lower as the workload as drastically decreased, but the GPUs power consumption remains high due to being underutilized. Furthermore, with the exception of AlexNet CIFAR-10, FPGPT has the best power efficiency of any state of the art presented. EF-Train and FPDeep do still outperform the GPU for this metric. This shows that in terms of absolute speed the GPU is unparalleled, but for power efficiency, FPGAs are strong competitors.

Im2col Performance Comparison

Tab. 5.9 includes a 'FPGPT no im2col' column. This column represents what the performance for FPGPT would look like if it was simply designed as standard systolic array convolution. As a result, the im2col is performed on the CPU before moving the data across. The no im2col version of FPGPT runs on the same hardware and therefore has the same restrictions for memory. On the other hand, as the im2col is performed before any data is moved, the

		GPU	**FPGPT**	**FPGPT, no im2col**
AlexNet	*Conv GOPS*	**1809**	27.4	6.63
CIFAR-10	*GOPS*	**689**	22.3	6.28
AlexNet	*Conv GOPS*	**13357**	400	33.1
ImageNet	*GOPS*	**8798**	384	33.0
ResNet-20	*Conv GOPS*	**420**	55.3	5.25
CIFAR-10	*GOPS*	**196**	20.9	4.54
GoogLeNet	*Conv GOPS*	**2592**	156	31.4
ImageNet	*GOPS*	**1252**	146	31.0
SqueezeNet	*Conv GOPS*	**7454**	110	17.4
ImageNet	*GOPS*	**3252**	106	17.3

Table 5.9: FPGA-based CNN training approaches including scenario where the im2col is performed on the CPU. Peak GOPS for GPU are theoretical GPU maximum GOPS

size of the data moved is likely to be larger. The *Conv GOPS* value for the no im2col version considers the time from when the CPU starts converting data to the FPGA producing the final result. As both the FPGA and GPU operate on PCIe protocol for transferring data between the host memory and the off-chip memory, this was not included in the *Conv GOPS* value.

As can be seen, the performance is between 8-28% of standard FPGPT when not performing the im2col on device. This analysis assumes a favorable stance where the host to DDR transfer time is ignored assuming data is provided to the FPGA as soon as it is ready. Clearly, the amount of effort taken to performing the im2col on the CPU is clearly a prohibitive design choice, justify FPGPT's onboarding of the im2col. Furthermore, when the im2col is occurring on the FPGA in GoogLeNet and SqueezeNet for both the forward and gradient w.r.t. weights convolutions, the rate limiting factor is the DDR to fabric. As a result, increasing the amount of data to read will only exacerbate an already existing bottleneck. Both of these factors validate the benefit of moving the im2col process to the FPGA.

Theoretical Ideal FPGA device for CNN training

FPGPT offloads non-convolution layers to the CPU and will therefore always struggle to compete with a full GPU implementation. The analysis presented here investigates the impact of changes to available hardware resources as well as what is required for FPGPT to perform at a GPU level in terms of raw performance. AlexNet will be used as the example, as this was the network that elicited the best performance on the GPU. As demonstrated previously, FPGPT adapts to the workload of other networks better than the GPU. Therefore, if the FPGA can match GPU performance on AlexNet, then it should be able to similarly do so for other networks where the GPU performs worse.

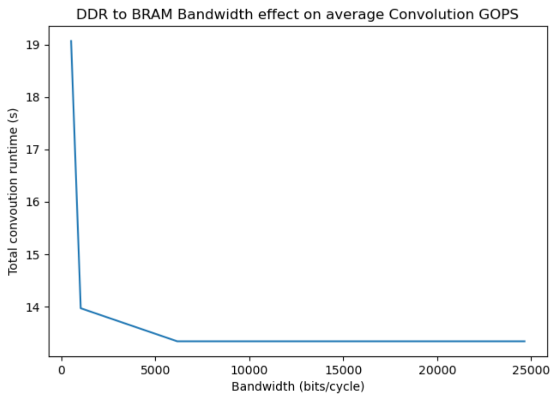
There are three parameters that are tuned. These parameters are key components of the resource model which in turn change the configurations produced by the DSE through the performance model. The parameters are the DDR to BRAM bandwidth, the DSP count and the number of BRAM blocks. In essence this demonstrates that if an FPGA of that size existed, this would be the averages GOPS that would be maximally achievable.

First the impact of changing a single parameter is explored. Fig. 5.15 demonstrates how each parameter individually scales while leaving the others at the Alveo U250 baseline. For all three figures, the Alveo U250 baseline is the first point along the x-axis. For reference, this is a bandwidth of 512 bits per cycle, 3072 dsps and 1344 BRAM blocks. In Fig. 5.15, the y-axis metric is the total time taken to perform all the convolutions or a single iteration. The time taken to perform other operations is left out to emphasize the impact of the convolutions.

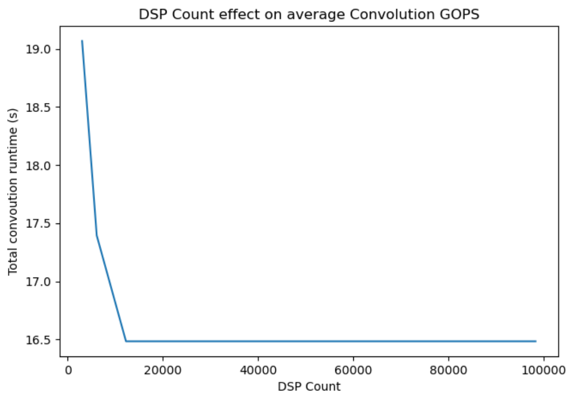
For all three parameters, there is a clear roofline of how much can be achieved solely by tuning a single parameter. As shown in Fig. 5.15b, when tailoring only a single parameter increasing the amount of DSPs provides the greatest benefits. This is as expected because the amount of parallel processing that can be performed scales directly with the DSP count. It also shows an almost linear progression until it is limited by one of the other two parameters.

Finally, looking at how bandwidth and BRAM scale generally with regard to average performance, the gradient tapers off towards the roofline limit. The reason for this is that as these parameters increase, the bottleneck resolved will benefit a decreasing number of kernels as the performance is more consistently limited by another factor.

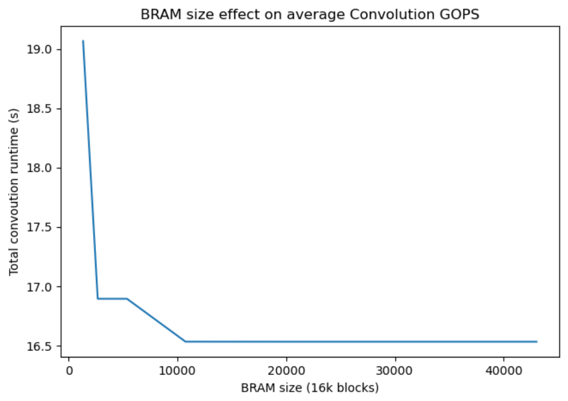
To get a better understanding of how the parameters react with regards to each other, Fig. 5.16 shows how the performance scales with respect to the combination of each parameter.



(a) DDR to BRAM bandwidth

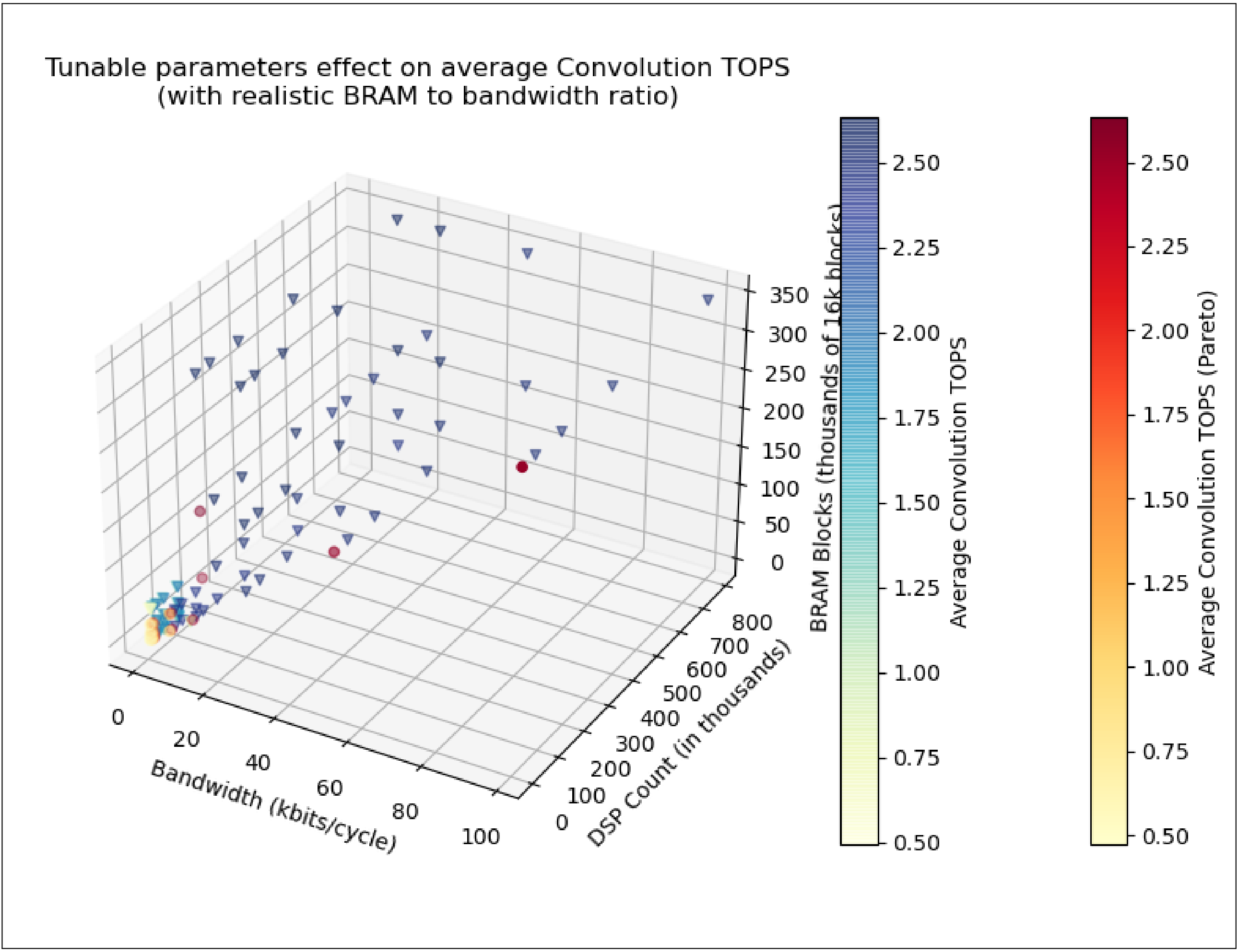


(b) DSP Count

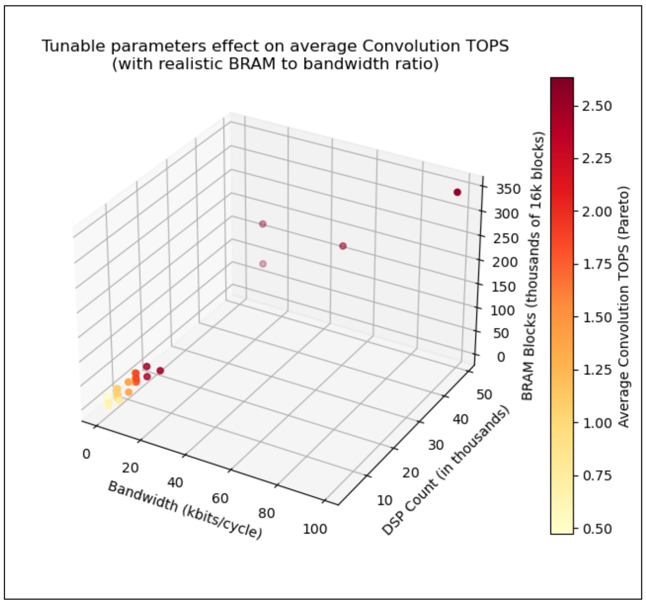


(c) BRAM size as 16k blocks

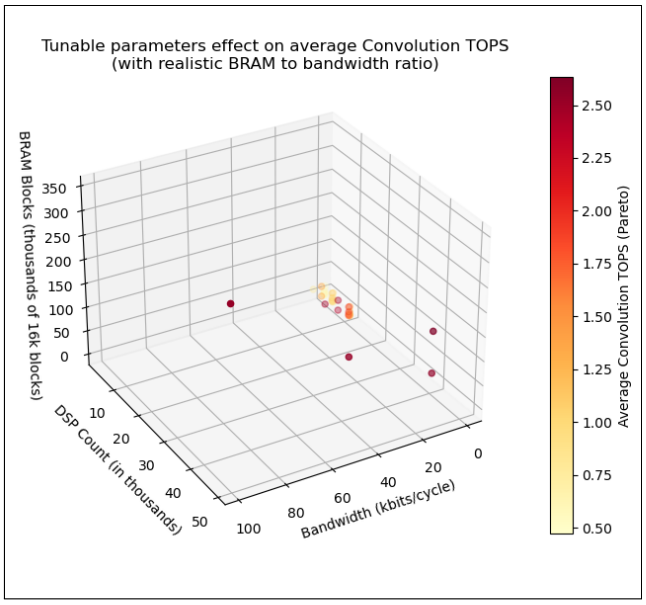
Figure 5.15: Effect of varying a single parameter while keeping all others at the current board values



(a) Effect of scaling tunable parameters. Blue scaled values triangles represent all data, red scaled values are represents pareto data



(b) Effect of scaling tunable parameters with only pareto values for clarity



(c) Effect of scaling tunable parameters with only pareto values for clarity (alternative perspective)

Figure 5.16: Effect of AlexNet average performance (in TOPS) due to scaling tunable parameters while ensuring that the ratio of BRAM to Bandwidth stays within exsiting bounds. The TOPS are calculated using the performance model. Bandwidth stands for DDR-to-BRAM bandwidth

Specifically in these graphs, there is a limitation that the ratio of number of BRAM blocks to bandwidth. The bandwidth in this scenario is the DDR-to-BRAM bandwidth. In this work so far the target FPGA has been a U250, which consists of 4 super logic regions each with an off-chip to fabric bandwidth of 512bits/cycle bandwidth. The FPGPT hardware architecture design assumes that each of the INP, KERN, SA and OUT sit on their own super logic region. INP, KERN and OUT all need to be able to read and write data so the effective usable bandwidth considered per super logic region at 512 bits/cycle. If a memory was created that could interact with the fabric at a far higher rate, it could be assumed a GPU would implement this too, making the analysis invalid. As a result the value of feasible bandwidth is limited to the current ratio of BRAM to bandwidth. This represents increasing the number of super logic regions for each of INP, KERN, SA and OUT have and thereby increasing the effective bandwidth. This is done to keep the assumption of the type of memory fair and consistent with the GPU.

In Fig. 5.16 there are 2 types of points, blue scaled triangular points and red scaled dots. The red scaled dots represent the minimum combination of tunable parameters for each potential TOPS values representing a pareto front. The color scaling of the dots represents the performance in TOPS of that combination of tunable parameters. The considered combination of kernels is the set that allows for the highest possible performance for a theoretical FPGA of that size.

Fig. 5.16a demonstrates the full set of data including the non-pareto values. As can be seen, there is a diminishing return when increasing the dimensions of the FPGA. The largest strides in performance all occur at lower values of all three parameters. For example, having an FPGA with 2x bandwidth, 4x the DSP count, and 4x the BRAM blocks compared to a U250 already produces 1.941 TOPS average performance, compared to 0.472 TOPS, a 4.11x improvement. That being said, to reach the peak performance of 2.633 TOPS average performance, an FPGA with 12x bandwidth, 16x DSP Count, and 128x the BRAM blocks is required. This demonstrates that as the FPGA gets larger, the bottlenecks becoming increasingly difficult to resolve by just blindly adding resources.

To make the analysis clearer, Fig. 5.16b and 5.16c only include the pareto values. These two graphs show exactly the same information, but are rotated 90 degrees to make the 3D representation clearer. Looking at the axes, Bandwidth and BRAM Blocks are the same as before, but the DSP axis is notably smaller. This makes is clear that increasing the DSP count above 50,000 DSP is of no benefit as the performance will not increase above that. As a result, it can be determined that the TOPS limit is caused by the DSP count, which becomes an insurmountable bottleneck.

The TOPS limit seen for these networks caused by the DSPs is caused by an inherent trait of the architecture design. As mentioned in Sec. 5.3 the X variable corresponds to one of the two dimensions of the systolic array. This means that as the DSP count grows, to be able to properly utilize it, both X and Y need to grow. Both of these values though are also directly linked to dimensions of workload. X is directly related $H_{OUT} \cdot W_{out}$ while Y is directly related to N_{KERN}. Consequently this results in DPS Count $< H_{OUT} \cdot W_{OUT} \cdot N_{KERN}$. Not only would this raise the total usable DSP Count, but this would also allow for more balanced values of X and Y even in scenarios where N_{KERN} or $H_{OUT} \cdot W_{OUT}$ are very small values. Changing the architecture to support splitting the problem to utilize large values of X and Y would require notable overhaul of the design and is therefore moved to future work.

Regarding the pareto values, for each performance value there is only a single possible combination of the tunable parameters. Because this data is the average performance across all the convolutions in a network, there is no clear discernible pattern in how the parameters affect performance. The reason for this is that the bottlenecking parameter for one layer is not necessarily the bottlenecking factor for another. If a certain layer (or set of layers) that provides the greatest workload, addressing that bottleneck is the most effective way to increase average performance. Addressing the bottleneck of other layers will still provide benefits, but not as meaningfully as addressing the core issue. An example is that having a bandwidth of 512 $\frac{bits}{cycle}$ can range from 472 - 1323 GOPS, depending on the other variables. Yet having a bandwidth of 6160 can have the performance of 680 - 2439. This is inline with the 2D graphs shown in Fig. 5.15. Tuning a single parameter leads to improvements, but eventually no layers are bottlenecked by that tunable parameter, and the others must be adapted to increase performance. But as soon a this happens, lowering the previously tuned parameter can still lead to an overall greater performance as more influential layers are accelerated more than Although each combination is the smallest combination for each parameter at that GOPS value, the interplay between the variables is crucial to achieving the highest possible performance. The general trend of increasing all three parameters leads to an increased performance, but being able to specifically pinpoint which metric is most appropriate at any point of time requires detailed analysis of the current bottlenecks for each layer.

There is one final parameter that can improve GOPS, which is clock speed. If the clock speed were to be increased to 500MHz for the highest performing configuration of the pareto values, the GOPS would match those achieved by a GPU. This is technically achievable on modern FPGAs, but is the absolute upper limit of clock speeds. Although theoretically possible, this is very hard to achieve. Improving clock speed can be achieved through a variety of methods. The primary method is by creating a more efficient design, for example by creating the design in Verilog. When developing in HLS, the compiler will make very conservative assumptions about data dependencies leading to inefficient pipeline design. The high degree of control provided by Verilog will allow for a greater degree control over the design avoiding the overly conservative data dependency assumptions. Additionally, sacrificing more fabric to perform multiplications removing the need for as many DSPs would enable computations to happen locally in fabric. This would reduced data movement to and from the DSP area of the FPGA further reducing routing distances and increasing the potential clock speed. Furthermore, specific layout handling can further improve the achievable clock speed. For example placing units physically closer together on the FPGA fabric will decrease the routing time and would allow for a higher clock speed as signal travel time can be reduced. This was not pursued in this work but can be implemented for greater performance benefit. This results in taking a design that has 12x greater bandwidth, 16x the DSP Count and 128x the BRAM blocks Xilinx's second largest FPGA in the market and creating a near perfect layout.

More realistically, the approach to match GPU performance would be achieved aiming for a 300MHz design as this is realistically achievable. This would require updating the architecture to be able to exploit excess DSPs more efficiently leading to a whole new set of possible layouts that have been ignored thus far.

5.6.5 Overall Evaluation

FPGPT provides an easy to implement CNN training acceleration ecosystem, making FPGA training accessible to all users. FPGPT has been actually run to on hardware to ensure all results presented are realistically implementable.

It has been demonstrated that for complex networks with large datasets, FPGPT is able to outperform EF-train [73] and compete with FeCaffe [74]. It is interesting to note the performance difference for FPGPT caused solely due to the dataset. Tab. 5.7 demonstrates that the FPGPT design performs better for ImageNet designs than CIFAR-10. FPGPT has been designed with complex networks trained using large datasets in mind. This does mean that at small execution sizes, FPGPT incurs the various penalties discussed in the performance model more often. The majority of the incurred penalties occur at the read in of the INP and KERN as the penalties become relatively greater for a smaller input image. Therefore, FPGPT is not a competitor for small networks and small datasets. The focus for FPGPT was to get high performance at high precision for the ImageNet dataset, which was been successfully achieved. As discussed, this is not a large problem as the ML community does not have a great interest in small networks training using small datasets.

It should be noted that other works like [66, 67] provide results only for MNIST. MNIST is considered completely outdated as a benchmark by the ML community and as such a performance comparison is omitted as it is seen of no real value. Currently in the machine learning community, research presented on anything less than ImageNet will not be accepted as relevant or novel. Furthermore, networks such as AlexNet and VGG (the networks utilized by FPDeep [1] and EF-Train [73]) are not able to compete in accuracy compared to more modern networks. They are also less common in machine learning applications. The results in Tables 5.5 and 5.6 demonstrate that FPGPT is able to perform competitively with all SotA works. Specifically, across all their respective domains for ImageNet training with regards to performance. This results in FPGPT being a highly competitive approach for training ImageNet in terms of overall network performance.

FPGPT sports another facet that is entirely unique; shared kernels. Because the architecture is runtime configurable, FPGPT can provide a single hardware kernel for multiple target layers. This is demonstrated in Section 5.6.2. Beyond sharing across layers within a network, FPGPT can create kernels that function across various networks. As long as the bitstreams have been compiled with a large enough design in mind it can be reused to train any layer across network. Being able to repurpose compiled bitstreams provides the flexibility to alter the target network without performing time consuming FPGA recompilation. Less recompilation through increased flexibility significantly decreases the overall amount of work required to train a network.

FPGPT's kernel sharing design, the ability to operate in FP32, and its inherent ML framework integration ability further enhances the use of FPGAs for CNN training. Improved performance and flexibility will make FPGAs more appealing to researchers. They no longer need to commit to a specific training setup or incur the large recompilation time as they do with the other approaches. They will also have the freedom to experiment with various the networks and datasets without needing to sacrifice performance or accuracy.

5.7 Conclusion, Limitations and Future Works

FPGPT has introduced a high performance FPGA kernel that is competitive with state of the art works while providing the following the following:

- Can benefit from but not exclusively dependent on FPGA clusters like FPDeep [1] for modern networks and datsets like GoogLeNet and SqueezeNet on ImageNet.
- Can be implemented on a variety of network or dataset.
- Runs convolution at FP32 precision while maintaining a high throughput.
- Allows for the flexibility to run different layers and different networks on the same kernel without needing to recompile.

- Is adapted to easily integrate into PyTorch or FeCaffe [74] to make FPGA CNN training more accessible to the ML research community..

FPGPT has demonstrated the ability to out perform current FPGA based CNN training techniques when analyzing peak and average GOPS when scaled to a single FPGA. Additionally, FPGPT can compete in runtime with other FPGA-based ML framework integration works. Furthermore it has outperformed GPUs for power efficiency. This places FPGPT as strong choice when training CNNs with an idea of the importance of training cost and efficiency.

Beyond the efficiency, FPGPT uniquely provides flexibility and adaptability. All other works require expensive recompilation to be applied to other networks. This makes the application easier for the research community as they need the ability to tweak and tune networks, and constant recompilation would make this prohibitively time consuming. FPGPT, with smart building of kernels, is able to reduce or even entirely avoid this recompilation cost.

5.7.1 Limitations

Currently the FPGPT design has certain limitations. The first is that it does not support partial reconfiguration. Partial reconfiguration allows for runtime reconfiguration of part or all of the FPGA. As the focus of FPGPT was to create a scalable, HLS based implementation of the design in Sec. 5.3, the extra Verilog work required to enable partial reconfiguration was not undertaken, but would benefit the FPGPT design. Secondly, the $Grad_{weight}$ kernels of a 1x1 convolution induce quite a large performance penalty as discussed in Sec. 5.6.4. This links to the design restriction discussed in Sec. 5.6.4 when analyzing how performance would scale in a theoretical FPGA. Here it was established that if the parameters of the workload can force X and Y to be small values underutilizing the available DSPs. This would require a redesign of the architecture to be able to breakdown the problem removing the X and Y restrictions. As many networks are moving towards having a large number of 1x1 layers, this would be a worthwhile addition to FPGPT. Finally, networks with limited but very large layers, such as the range of VGG networks, are not supported by FPGPT as their configurations cause memory issues with the on-chip memory making them unsupportable on the Alveo U250, and therefore unsupportable on most FPGAs. This is not considered a large issue as more relevant and powerful networks such as GoogLeNet and SqueezeNet are supported. The design of a higher amount of smaller convolutions is gaining popularity in the machine learning world, meaning this limitation is not of great concern.

There were additionally a range of issues by focusing on using HLS. HLS itself required a tremendous amount of to be able to create the desired design. A lot of the code is written in a non-obvious manner to accommodate the HLS compiler nuances. Also, many of the provided pragmas did not act as specified in the documentation. Furthermore, it placed various restrictions leading to inefficiencies. An example is all the aligning required to the $pmem^{INP}$, and the memory analysis performed in HLS compilations is extremely conservative. Additionally, the inner and outer loops required in the INP and KERN DDR reading components are another symptom of using HLS. Finally, HLS designs are often less optimized and lead to a lower clock speed. The design presented here operated a clock speed of 100MHz. This is only 20% of the board maximum. Moving to Verilog and spending time performing specific optimizations would realistically allow the design to move up to 300MHz (commonly seen speed for reasonably optimized designs). Such a change would provide a 3x speed up without needing to change or improve the architecture's inherent design. Additionally such a change would still provide benefits regardless of other changes implemented, making it a highly worthwhile pursuit.

5.7.2 Future Work

Regarding future work, merging with FeCaffe [74] would provide a higher performance hybrid outperforming either work individually. As stated in Sec. 5.6.2, there is potential for even higher performing kernels if provided with a more intricate resource model. Additionally, partial reconfiguration is definite target alongside the ability for the values of X and Y to not be limited by the potentially small values of $H_{OUT} \cdot W_{OUT}$ and N_{KERN}.

Outside of runtime optimization, an optimized reduced precision kernel can reduce resource consumption per PE allowing for greater parallelism with minimal impact on accuracy if implemented correctly [102]. This would also allow for greater data transfer and burst-reading more words per cycle, currently one of the main bottlenecks in the design. Additionally, there is the introduction of Verilog blocks into the HLS code to circumvent inefficiencies originating from getting the HLS compiler to create the desired design. This would provide a greater opportunity for optimization through greater control of the design. This could result in more controlled resource allocation and well as specific pipeline structures that would aid in the interleaving. Moreover, Verilog blocks would make it possible to significantly improve the overall performance by raising the clock speed of the design if implemented efficiently.

Finally, the layers that are accelerated by hardware can be expanded. As the convolution layers get increasingly accelerated, different layers will become the new bottlenecks. Additionally, other layers can be integrated with the FPGPT hardware architecture. For example, ReLU layers filter out all negative values, an operation that can easily be implemented within the OUT memory subsystem of the FPGPT architecture. Partial reconfiguration can even be combined with the ability to accelerate a variety of layers to improve training overall. The only thing to keep in mind is that all forward activations still need to be passed back to the host to ensure it is present for the backward pass.

Chapter 6

GM: Glocalization of MuPPET

Local & global multi-precision CNN training policy with a custom FPGA convolution accelerator

This chapter introduces the final part of the thesis. So far two aspects of CNN training have been explored. The first was the algorithmic route with MuPPET. Low precision techniques were explored to accelerate CNN training and global training progress as tracked to reduce the effect of quantization error on the final accuracy of the network. The second was the hardware route, initiated by Caffe Barista and then improved by FPGPT. Both works utilized systolic arrays on FPGAs, with FPGPT utilizing the information learned about Barista's bottlenecks to create a flexible, easy to use hardware architecture with SotA performance. Although they operate on different systems, algorithmic versus hardware, they both place their focus on accelerating the computation the most time consuming operation in a CNN: convolutions. Furthermore, FPGPT provided a highly accurate and verified performance model that allows for detailed a analysis of convolution performance.

Another way to view the difference between the MuPPET and FPGPT is the scope they address. MuPPET operates at a macro level, tracking how the network operates as a whole. It treats the entire network as a monolith which operates at a single precision. Attempting to operate at precisions below 8 bits while treating the network as a monolith resulted in an inability achieve any training convergence. MuPPET is only concerned with the training progress over epochs and analyzing progress in large time steps. As a result, MuPPET does not concern itself with what happens to each individual convolution in each individual iteration. FPGPT on the other hand operates on a micro scale. Each individual convolution is examined and optimized for improved individualized performance. Furthermore, as demonstrated in Ch. 5, the computation time for the convolutions encountered in a network can vary. Consequently, the absolute reduction in computation time achieved by computing a convolution at a reduced precision varies according to the dimensions of the convolution. Having a detailed performance model, FPGPT can enable the identification of the most essential convolutions to focus on to accelerate the training process while avoiding the quantization of the rest of the convolutions in the network.

Due to the difference in operating levels, these works are ideal for being superimposed upon each other to cover both micro and macro levels for training acceleration. To the best of our knowledge, the policy presented in this chapter, `GM`, is the first to utilize insights about custom hardware to aid in guiding the acceleration of CNN training by exploiting the number representation utilized by each convolution computation.

The work presented in this chapter is not the complete product expected from `GM`, as there was not enough time to complete it. The underlying design and assumptions are discussed, as well as any results and analysis that have been performed.

6.1 Related works

The comparable works for GM have a large overlap with the those presented for MuPPET in Ch. 3 and FPGPT in Ch. 5. Therefore only works that did not apply to MuPPET and FPGPT are discussed here.

As will be introduced in Sec. 6.3.4, `GM` contains an error metric, in order to quantify the impact or reduced precision per quantization. When looking at state-of-the-art literature, there are two general routes to analyzing error during training and quantization: 1) complex higher-accuracy metrics, 2) simple lower-accuracy metrics. [21, 58, 46, 55] focus on the high complexity, high accuracy metrics, whereas [52, 56] focus on applying simpler to compute metrics. These works include either quantize aware training and post training quantization approaches.

The complex high-accuracy metrics works do not focus on the overheads they introduce, as they aim to produce the most accurate reduced precision final network. AutoQ presented in [21] utilizes deep reinforcement learning schemes to find the optimal quantization level for each layer. The authors of [46] investigate the effect the quantization of an individual layer has on the loss landscape, eventually combining layer-by-layer quantization with a multivariate quadratic optimization that accounts for the impact of the quantization. [58] propose differentiable soft quantization, a methodology that throughout training approximates the best clipping range for the forward layers while adapting the gradients to guide training towards ultra-low precisions networks. EWGS (element-wise gradient scaling) from the authors of [55] focus only on gradient scaling, adaptively scaling each $Grad_{weight}$ element, basing the scaling factor on the second order derivatives of the loss w.r.t. to quantized activations. All of these works are able to produce highly accurate reduced precision networks, but at the cost increasing training time.

On the end of low complexity metrics, these works aim to use the minimal overhead required to be able to produce networks with state-of-the-art accuracy. OCTAV (optimal clipped tesnors and vecotrs), presented in [52], computes scalar clipping on-the-fly using the fast Newton-Raphson method focusing on minimizing MSE. There authors state that MSE between computed data and the post quantization result is one of the best methods of tracking quantization error in CNNs. The authors of [56] present Piecewise Fixed Point (PWF) that accounts for the bias introduced, with a primary focus on $Grad_{weight}$ values to limit the noise during training.

The NITI training framework presented in [70] and the DarkFPGA framework presented in [69] are of particular interest. DarkFPGA places more of a focus on a hardware-software co-design that focuses on using batch-level parallelism to accelerate FPGA-based CNN training. NITI on the other hand focuses on integrating custom reduced precision quantization techniques in their loss and weight update functionality that provide additional benefits on custom hardware. For more details on these works, please refer to Ch. 5 Sec. 5.2. Both of these works present their own frameworks and therefore do require the user to switch to this custom framework. Although

DarkFPGA only presents results for MNIST and CIFAR-10, NITI does provide acceleration potential on ImageNet tests. These tests are exclusively on the GPU, but initial testing for FPGA has been shown, but not in a CNN scenario.

In the space of having an accelerated FPGA training ecosystem that merges with an existing training framework there is not a lot of related work. Looking beyond the scope of FPGAs, Mixed Precision training [14] is still one of the leading works for training full precision networks while utilizing reduced precision during training. The authors of [103] produce a 32bit dynamic fixed point network trained at reduced precisions with comparable final accuracy to FP32 networks. The core principles are based off the Mixed Precision [14] work where they also keep a high precision master copy of the weights while performing computations in a lower precision. But where Mixed Precision use floating point for all their computations, [103] gain additional performance benefits by performing all their computations in dynamic fixed point.

6.2 Hybridization Motivation

At its core, MuPPET is a general yet complex policy that treats the entire network as monolith. This excludes a lot of specific information that could be gained from looking at each convolution individually.

Gradient diversity is considered as a single metric across all convolutions, applying the precision switch to all convolutions. Furthermore, due to the nature of the information collected and the complexity of the metric, MuPPET only applies the precision switch once per epoch. This makes the opportunities to switch a far coarser decision, potentially missing finer but optimal windows. Increasing the switching frequency would increase the amount of times gradient diversity is computed. The increased cost of computation is the reason behind MuPPET's approach to only compute the gradient diversity once per epoch, as this allowed for enough control to lead to a high final accuracy while not overly diminishing the acquired benefit.

As we have seen in Chapter 5, there are a variety of workload characteristics across convolution. This means that having them all run at a reduced precision does not provide an equal amount of absolute acceleration. This begs the question as to whether each convolution should be treated with equal importance seeing as they do not provide equal acceleration benefits.

Finally, there is a limitation to MuPPET that it was unable to properly exploit its own full potential. MuPPET was tested on a GPU, which has limited precision options. The GPU only hosts an 8bit fixed point core and a 32bit (repurposable as 16bit) floating point core. This means that all the specific fixed point precisions between 8bit fixed point and 32bit floating point were not running efficiently. They were all executed on 16bit floating point cores although they were computing in a lower precision fixed point representation.

To address the limitations of MuPPET described above, a low complexity per-convolution tracking policy could be implemented. FPGPT currently creates custom and adaptable convolution units. A simple combination of the two systems where FPGPT acts as the convolution unit for MuPPET, addresses the current inability of MuPPET to benefit from the non-standard precision levels. The FPGPT hardware architecture could be amended with extended functionality to perform additional analysis to drive a fine-grained precision policy. This would additionally entail that MuPPET uses FPGPT as the convolution engine. Both works being natively integrated into PyTorch eases the process of combining the two works.

Beyond allowing MuPPET to overcome the limitation of not exploiting non-standard precisions, the combination of these two works is able to provide benefits to each individual work that is as of yet unexplored.

6.2.1 Additional MuPPET Benefits

Currently, although GPUs have some support for reduced precision computation, this is only for 4- and 8bit fixed point and 16bit floating point. FPGPT on the other hand, can be modified to support any custom precision scheme, as long as its properly defined in Vitis HLS. As a result, FPGPT provides MuPPET with the ability to utilize all the specific quantization levels that are currently supported by hardware on any existing GPUs. Beyond this, FPGPT allows for a fine grained insight into the acceleration benefit, through its performance model that indicates

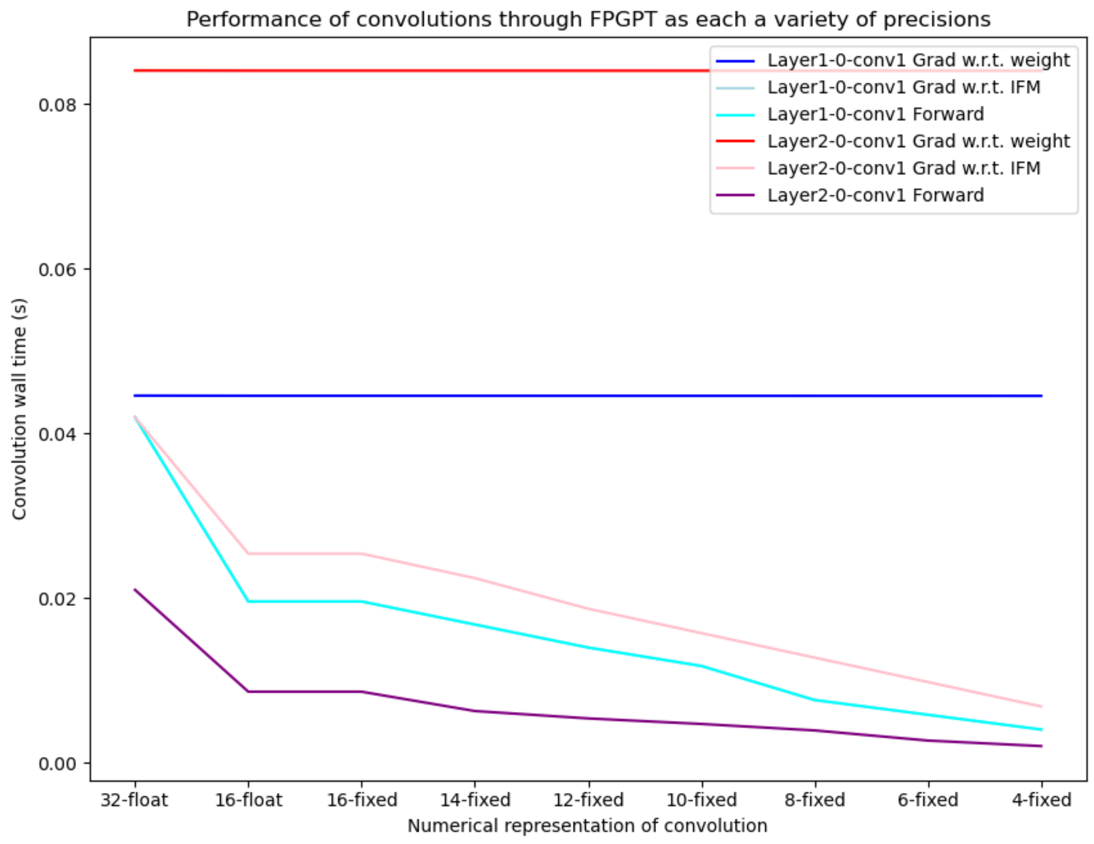


Figure 6.1: Performance of various ResNet20 convolutions at different precision

the acceleration potential of each convolution.

Fig. 6.1 demonstrates the potential benefit provided by all three convolutions of both Layer1-0-conv1 and Layer2-0-conv1 from ResNet20 at various precisions. It is clear that accelerating certain convolutions has a greater impact on performance than others. As a result, reducing precision for those layers would potentially harm the convergence and final accuracy while not providing any acceleration. This demonstrates that MuPPET is hindering its own ability to provide a high accuracy final network and gaining no training acceleration in return. Looking at the $Grad_{weight}$ data, there is only a minimal change in convolution wall-clock time. Upon further inspection of the model, for these layers the convolution the slowest component is the DDR Read component for all precisions. A likely explanation for this is the fact that the small input size (due to CIFAR-100 being the dataset) limits the efficacy of the burst read, stunting performance. Outside of the $Grad_{weight}$, looking specifically at Layer2-0-conv1 forward, it provides a far less acceleration benefit than Layer1-0-conv1 forward. This shows the benefit to prioritizing convolutions that have the greatest acceleration potential. This indicates there are convolutions reducing the ability of MuPPET to train while not providing the same degree of acceleration, potentially negatively impacting the convergence of the training.

Additionally, because the FPGPT bitstreams compute all the convolutions during training, they can be modified to also function as a fine-grained analysis unit. As long as the analysis performed only minimally affects the performance of the convolution, this would provide the potential for further acceleration beyond MuPPET. Furthermore, this analysis is computed at the finest time interval available which is once per training iteration.

Finally, due to the way MuPPET performed the alterations to the network to induce reduced precision, the network operates at a single precision. In FPGPT, each kernel would operate as an individual unit, meaning each kernel can operate at its own precision. This fine grain control would be the factor that enables individualized precision per convolution.

6.2.2 FPGPT Requirements

In Chapter 5, the FP32 version of the FPGPT hardware architecture design is analyzed and described. Specifically for this chapter, the model must be expanded to include a fixed point design as well. The first benefit of moving to reduced precision computation is that lower precisions can improve the utilization of the FPGA's DDR-to-BRAM bandwidth. The bandwidth of the DDR burst-read remains at 512bits per cycle. In the case of the FP32 design, this is 16 words per cycle. For 8bit fixed point (FXP8), the burst-read can now provide 64 words per cycle. Similarly the reduced precision allows for better utilization of the on-chip memory scaling similarly to the DDR benefit. Having a more relaxed memory resource constraint will allow for potentially greater parallelism. Lastly, moving from floating to fixed point simplifies the architecture of the PEs. A large source of complexity was dealing with the interleaving buffer introduced due the accumulation delay caused by floating point addition. Fixed point addition is a 1 cycle process, which results in removing the need for an interleaving buffer. If there is no interleaving buffer, there is no partial results accumulation either, removing the 2nd phase of each PE. This reduction in complexity will allow for more efficient hardware utilization and potentially a faster clock speed ass a result as well.

The DSP resource count drops from 9 DSP per PE to 3 DSP per PE, potentially tripling the number of PEs that can be instantiated on the same FPGA device and therefore the resultant parallelism.

6.3 Updated Precision Policy

When developing MuPPET, it became clear that starting training below 8bit fixed point made convergence challenging. Therefore, a hard bottom line was created for acceptable precision for the MuPPET policy. Fig. 6.1 not only demonstrates how different convolutions are affected by quantization, but also that not every quantization step provides the same level of acceleration. Looking just at Layer1-0-conv1 forward (cyan line), the potential acceleration from 10bit fixed point to 8bit fixed point provides greater acceleration than going from 8bit fixed point to 6bits fixed point. In this scenario, dropping from 8bit fixed point to 6bit fixed point provides less acceleration yet will introduce a greater quantization error to the network. This demonstrates the need for an analysis for each convolution at each precision to determine whether the increased quantization error is worth the acceleration achieved by performing that convolution at a reduced precision.

If during training only the convolutions that accounted for the largest acceleration to runtime were considered, similar acceleration can be achieved while leaving many convolutions at a higher precision. Leaving multiple convolutions at a higher precision decreases the amount of quantization error introduced during training. This leads to the underlying intuition of the expanded precision switching policy, `GM`. If the effect on the overall training process caused by quantizing parts of the network is reduced, lower precisions can be explored. This leads to the core concept behind `GM`; extend the precision switching policy by adding the feature of fine-grained convolution-wise adjustments on top of the MuPPET precision switching policy.

This extra feature will reduce the precision of convolutions that are deemed acceptable, while original MuPPET tracks the overarching training process, raising the global precision whenever is deemed necessary. Layers are deemed acceptable to reduce precision if they limit the amount of quantization error introduced to the system while providing enough acceleration to be deemed useful. The performance and error boundaries are `GM` hyperparameters that will be discussed in Sec. 6.3.3 and Sec. 6.3.4 respectively.

6.3.1 Core Principle

Before going into the details of the performance and error limits, the core principles of the `GM` are described. `GM`, as a new policy, has an unchanged version of the MuPPET policy at its base. The aim is to add fine-grained, convolution-specific adjustment to the training process. MuPPET is restricted to treating the network as a monolith, both due to its implementation and the gradient diversity metric. Furthermore, MuPPET only updates and analyzes its metric following a coarse-grained time scale of once per epoch. Increasing the frequency of use of the metric could result in early switching and potentially even more acceleration, yet doing so would be very computationally expensive. Consequently this additional high complexity analysis could slow down training more than it is accelerated by reducing precision.

To perform fine-grained analysis while not sacrificing acceleration potential, computationally light metrics must be used to analyze each convolution's quantization error. These metrics focus on computationally light quantization error metrics and the effective performance benefit for reducing a convolutions operating precision. Additional hyper-parameters are introduced that control the policy's sensitivity to the performance and error metrics, which can be tuned to improve policy performance. As discussed in Sec. 6.1, attempting to predict the effect that the quantization error of a convolution will have on the overall accuracy is computationally expensive. As the new metrics are meant to be of low complexity to compute, the analysis of predicting the overall impact of a convolution quantization error is not attempted.

Combining a quantization error metric and a performance metric ensures that if a convolution can have its precision reduced, but the performance benefit is negligible, the convolution's precision remains unaltered. This approach allows convolutions that would not provide a lot of acceleration to not additionally compound the existing quantization error. The intuition behind this is that keeping low performance impact convolutions at a high precision will allow the high performance impact convolutions to reduce precision more aggressively while maintaining a strong training convergence.

6.3.2 Overall Policy

First the policy is described including the components that create the overall policy. Afterwards, each metric and hyperparameter are discussed individually for more detail.

The novel addition that `GM` contributes is guided by two new metrics, an error and a performance metric. The performance metric τ captures the acceleration benefits of reducing the operating precision of a convolution. The error metric ϵ ensures that the error that this reduction introduces is not so great that it affects convergence.

Before training starts, a hyperparameter of the performance metric decides which of the total convolutions, $conv_{total}$, are eligible for acceleration, referred to as $conv_{acc}$. For each of these convolutions an error metric history buffer is put in place. This buffer is of a fixed size, η, and stores a running history of the previous results of ϵ.

At the start of training, the current performance of all the convolutions in the network are computed based on the FPGPT performance model. For each element of $conv_{acc}$, the amount of acceleration it will provide to the network if its precision is reduced is calculated. If this is deemed acceptable, based on a hyperparameter, the convolution is added to the set $conv_{\tau}$ and will be considered for reducing its precision.

At every iteration, for all elements of $conv_{\tau}$ after the convolution is computed, ϵ is calculated and a locally held running arithmetic average, ϵ_{avg} is updated. After a set number of iterations, as determined by the frequency hyperparameter ω, a 'tick' occurs. In every 'tick', ϵ_{avg} is compared to the error metric threshold, $\epsilon_{threshold}$ set by the user. If $\epsilon_{avg} \leq \epsilon_{thershold}$, a 1 is placed in the history buffer, otherwise a zero is placed. After the history buffer is updated, ϵ_{avg} is set to 0. After updating the buffer and resetting ϵ_{avg}, if the number of 1s in the history buffer is greater that the pass threshold ψ, the operating precision of that convolution is reduced. The history buffer is then reset for this convolution. As the overall networks performance has changed, the set of $conv_{\tau}$ is recomputed. In essence, this system monitors whether in the recent past the number of times that $\epsilon_{avg} \leq \epsilon_{thershold}$ is greater than ψ, and if so reduces the precision of the convolution.

This process repeats itself throughout training, continuously attempting to lower the precision of all convolutions in $conv_{acc}$ when possible. Nonetheless, throughout all of this, MuPPET still acts as the overruling policy as it ensures the final precision is within acceptable bounds. This entails that when MuPPET induces a precision switch all convolutions in $conv_{acc}$ are set to the same precision, all history buffers are cleared and ψ is set to 0.

The user is able to decide if they want to optimize the final network by allowing `GM` to continue to operate once the MuPPET policy has moved $conv_{acc}$ to 32bit floating point. If `GM` operates past this point, the final network will be a combination of fixed and floating point convolutions, otherwise the result will just be a complete 32bit floating trained network.

The combination of the two metrics and the additional `GM` hyperparameters η, ω, ψ configures the additional `GM` policy. Each of the hyperparameters allows for the user to tune how aggressively and fine-grained the error metric is analyzed and applied. Increased aggression does come with a compute performance trade-off and an increased vulnerability to outliers. The most direct factor for this is the frequency parameter, ω. The history buffer size parameter, η also contributes to increase compute. Pass count ψ and error threshold $\epsilon_{threshold}$ only change how reactive the policy is and exclusively address the vulnerability to outliers.

If the hyper-parameter setup is too vulnerable to outliers, this can lead to going too low in quantization reducing final accuracy values. Fig. 6.2 demonstrates this susceptibility. The only difference between this run and a successfully accelerated run with almost no loss in accuracy compared to MuPPET is the value of η. Reducing η to 10 (see Sec. 6.4.2) instead of 50 allows for successful training, as any the combined effect of the outliers led to too many convolutions having their precision reduced at inappropriate moments, derailing training.

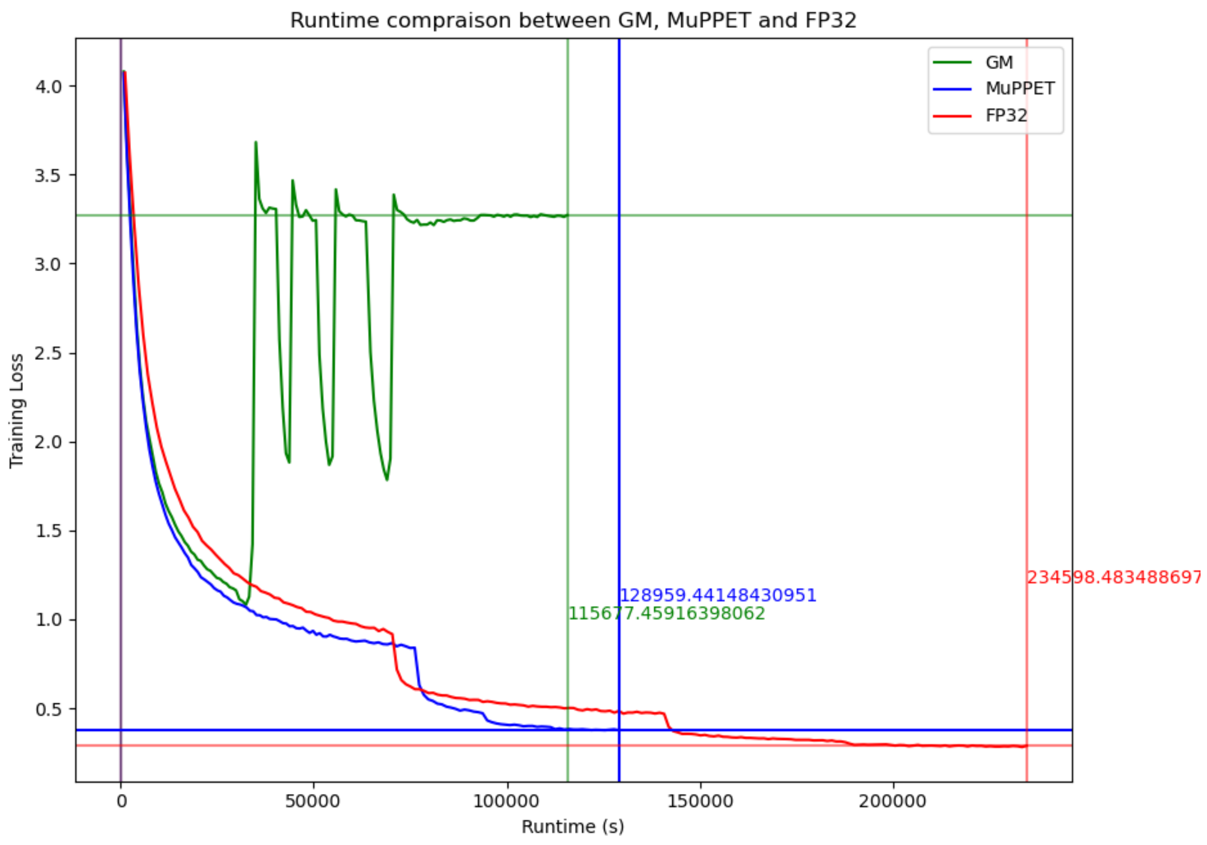


Figure 6.2: Runtime performance vs. training loss of `GM` (green), MuPPET (blue) and FP32 (red) with increased susceptibility to outliers. $\eta = 50$, $\psi = 3$, $\omega = 1$, $\epsilon_{threshold} = 0.05$, $\tau_{total} = 100$, $\tau_{total} = 0.0$

The overall policy for this work aims to reduce the individual precision of each convolution as low as possible without harming the final accuracy. Being able to maintain a balance of pushing the level of quantization while maintaining accuracy is essential to getting desired results. Although MuPPET is in place to track overall training progression, overly aggressive quantization will derail training. Contrarily, being overly conservative when setting the hyper-parameters will lead to just running vanilla MuPPET. Therefore these hyper-parameters require manual tuning to be utilized as effectively as possible.

6.3.3 Performance Metric

In the overall policy, the error metric is only applied to a subset of convolutions. The first round of pruning convolutions is to generate $conv_{acc}$, the second round generates $conv_{\tau}$. The way in which this pruning is performed is based on the performance metric, τ. This metric aims to ensure that only convolutions that truly provide a significant amount acceleration are quantized. What is considered to be significant enough to merit a reduction in precision is determined through hyperparameters. The metric has two tunable hyperparameters; 1) the percentage of the total acceleration potential that is targeted τ_{total} and 2) the required acceleration for a lower precision to be utilized $\tau_{threshold}$. In perspective of the overall policy, τ_{total} is used to generate $conv_{acc}$ and $\tau_{threshold}$ is used to generate $conv_{\tau}$

The first hyperparameter for this metric is τ_{total}, which represents the percentage of total acceleration, allows the user to select how much of the potential speed up they actually want. The lower this percentage, the less convolutions that are subjected to quantization, i.e. less convolutions in $conv_{acc}$ and the higher the network accuracy is likely to be. Setting this hyperparameter to 100% will result in all convolutions being considered for quantization. Setting this parameter to 80% on the other hand, will ensure only the convolutions that account for 80% of the network performance are considered. For example, for ResNet20, with only 21 out of the total 63 convolutions, 80% of the acceleration potential can be achieved. Looking at these convolutions, all of them are forward and $Grad_{IFM}$ convolutions. This is of significance, as quantizing $Grad_{weight}$ is often a notable challenge due to how small the values are, meaning they can vanish during quantization hindering convergence [16]. This is considered a tunable hyperparameter, as finding the percentage of the total acceleration that leads to the best accuracy and performance combination will be network and dataset dependent. Furthermore, operating with $\tau_{total} < 100\%$ will result in less potential acceleration (as less convolutions can reduce their precision) but as a result can result in faster convergence leading to overall faster training.

The second part of the performance metric is the required acceleration to reduce the precision of a *selected convolution*, $\tau_{threshold}$. In this context, a *selected convolution* refers to one of the convolutions selected for quantization by the first performance metric hyperparameter. $\tau_{threshold}$ is a hyperparameter to determine if, for a certain convolution, a reduction in quantization provides enough acceleration to merit the loss in accuracy, during runtime. The acceleration produced by a convolution (τ) represents the performance metric and is computed as the difference in overall performance caused by the precision of this one convolution:

$$\begin{aligned} \tau &= \text{total runtime}_{\text{current setup}} - \text{total runtime}_{\text{convolution reduced in precision}} \\ \text{total runtime} &= \sum_{c \in conv_{total}} performance(c) \end{aligned} \tag{6.1}$$

where $performance$ refers to the modeled performance based on the FPGPT performance model. The two 'total runtime' variables refer to the total convolution runtime with the target convolution at its current precision and the total convolution runtime with the target convolution at its target precision respectively.

If a convolution is deemed to provide enough performance benefit, it becomes eligible to have its precision reduced. Eq. 6.2 formulates the operation of this threshold:

$$\tau_{reduce} = \begin{cases} 1 \text{ if } \tau \geq \tau_{threshold} \\ 0 \text{ if } \tau < \tau_{threshold} \end{cases} \tag{6.2}$$

where τ_{reduce} determines whether the convolution is eligible.

The performance data for each convolution is derived the performance model utilized in the

FPGPT DSE.

As shown in Fig. 6.1, the relative acceleration between one quantization level and the other is not consistent across convolutions. The dimensions of the convolution workload, the dimensions of the compiled bitstream and the current operating precision are all factors that will influence where the current bottleneck is in the convolution architecture. Therefore, changing the precision for a convolution will have varying impacts based on the convolution, the compiled bitstream and the new precision. One such example would be utilizing the memory more efficiently. Imagine for a convolution going from 10 to 8 bit fixed point that is not able to not instantiate partially filled BRAM blocks This would result in more memory being available increasing the Z and X parameters increasing overall parallelism. On the other hand, a convolution that is not instantiating partially filled BRAM blocks at 10bits would not gain the same advantage moving to 8bits.

Once a convolution has $\tau_{reduce} = 1$, the error metric (ϵ) is analyzed to see if the operational precision of the convolution can be reduced.

6.3.4 Error Metric

In addition to the performance metric, `GM` has an error metric ϵ. The underlying principle of the error metric is that if the error introduced by further quantizing a single convolution is large, this will have a notable impact on the overall accuracy of the network. As presented in Sec. 6.1, there are works that actively analyze the impact a certain convolution has on the overall accuracy. Such a metric could be used to predict the impact of the increased error introduced by quantizing a specific convolution. However in many cases this does come at a large computational cost, negating all the acceleration attempts. To avoid the complexity of these advanced analyses, the error metric adopted in this work is kept simple and local. Although this does not as accurately capture the impact on overall accuracy, the underlying principle still holds.

The work presented in OCTAV [52] was used as guiding principle, as it focused on minimizing

MSE. In `GM`, one metric is used for all three convolutions which have a large range of values, particularly $Grad_{weight}$ which often tends to be smaller compared to Forward and $Grad_{IFM}$. As a result, the selected error metric for this work is the difference in the Relative Absolute Error (RAE), as this is based on MSE, but scales relative to the data at hand. Eq. 6.3 shows how the RAE is computed:

$$\text{RAE} = \frac{\sqrt{mean\left((orig - new)^2\right)}}{\sqrt{mean\left(orig^2\right)}} \tag{6.3}$$

where *orig* represents the original tensor and *new* represents the target tensor. As a note, when performing a convolution as a GEMM at a reduced precision, due to the fact there are a potentially large number of MACs, the internal precision is often raised to prevent overflow. In `GM`, the internal precision is 32bit fixed point, so therefore *orig* is a tensor of 32bit values.

When looking at the error metric, the difference in RAE is analyzed. Eq. 6.4 portrays how the difference in RAE is computed:

$$\begin{aligned}\epsilon = \text{RAE}_{difference} &= \frac{\sqrt{mean\left((orig - target)^2\right)}}{\sqrt{(orig^2)}} - \frac{\sqrt{\left((orig - current)^2\right)}}{\sqrt{(orig^2)}} \\ &= \frac{\sqrt{mean\left((orig - target)^2\right)} - \sqrt{\left((orig - current)^2\right)}}{\sqrt{(orig^2)}}\end{aligned} \tag{6.4}$$

where *orig* is the same as in Eq. 6.3, *target* is the tensor at the target precision and *current* is the tensor at the current precision.

With ϵ computed, it is compared to the error metric threshold $\epsilon_{threshold}$ to see if this convolution is allowed to reduce its precision, as demonstrated in Eq. 6.5:

$$\epsilon_{reduce} = \begin{cases} 1 \text{ if } \text{RAE}_{difference} \leq \epsilon_{threshold} \\ 0 \text{ if } \text{RAE}_{difference} > \epsilon_{threshold} \end{cases} \tag{6.5}$$

If both $\tau_{reduce} = 1$ and $\epsilon_{reduce} = 1$ then the precision of the convolution is considered to be a 'pass' (see Sec. 6.3.5 for details). If enough 'passes' occur, then convolution is lowered according

Parameter	Symbol	Function
$\epsilon_{threshold}$	Error Metric threshold	Determine if RAE$_{difference}$ is small enough to allow for a reduction in convolution precision.
$\tau_{threshold}$	Performance Metric threshold	Determine if τ is large enough to allow for a reduction in convolution precision.
τ_{total}	Performance Metric total acceleration percentage	Determine how much of the maximum potential acceleration is exploited.
ω	Frequency	Frequency with which a 'tick' occurs per epoch.
η	History	Size of the history buffer.
ψ	Pass count	Required number of times $\tau_{reduce} = 1$ and $\epsilon_{reduce} = 1$ within the last η 'ticks'.

Table 6.1: Summary of all the hyperparamters

to the precision change scheme and the next iteration it will operate at a new precision.

6.3.5 GM Hyperparameters

A primary contribution of the new policy is the fine-grained analysis. Throughout training, the error metric is measured every iteration with a running average being held in memory. Furthermore, the performance metric is in place to ensure only convolutions that provide enough acceleration are reduced in precision. These metrics introduced their own hyperparameters such as τ_{total}, $\tau_{threshold}$ and $\epsilon_{threshold}$.

In Sec. 6.3.2, three more hyperparameters were introduced. These hyperparameters determine how and when the decision to reduce the precision is made and consist of: frequency (ω), history (η) and pass count (ψ). These hyper-parameters affect the frequency, computational cost and severity of the decision respectively. Tab. 6.1 summarizes all of the hyperparameters introduced for GM.

These only apply to the error metric, not the performance metric. The reason for this because the performance metric is a static precompute known beforehand. As it is static data, it won't change and tracking it in any form will not result in useful analysis.

Frequency: ω

The frequency parameter, ω, provides the user with ability to tune the error metric averaging interval and how often the decision to reduce the precision is considered. Throughout training, a running average of $\text{RAE}_{difference}$ is maintained. In line with ω, this average is committed to memory and a new running average is started. When the average is committed to memory, at the same time the history buffer is analyzed to see if the precision should be changed. This allows for a balance between generalizing behavior and rapidly adapting to training.

If this parameter is set to update very frequently, the system becomes susceptible to outlier minibatches. Furthermore, although the analysis whether to reduce the precision is not expensive, it is not a free computation either. This means that performing this extremely often does come with a computational burden. Contrarily, if this parameter updates too infrequently, a possible precision switching moment can be missed, reducing the acceleration potential. This can lead to longer runtime decreasing the efficacy of the new policy.

Pass Count: ψ

Pass count is in reference to the amount of times the target convolution violates the error threshold. At every 'tick', the convolution is investigated and a pass occurs when $\tau_{reduce} = 1$ and $\epsilon_{reduce} = 1$. If a pass occurs, a 1 is added to the history buffer, otherwise a 0 is added. The total pass count is represented as:

$$pc = \sum_{n \in \text{history buffer}} n \tag{6.6}$$

If enough passes occur in the current history, then the effect of the quantization is considered minimal enough to allow for a reduction in precision:

$$reduce = bool(pc \leq \psi) \tag{6.7}$$

If $reduce = 1$, then the precision of the unit can be lowered. The pass count is only considered for the values inside the history buffer.

Depending on what the frequency parameter is set to, if $reduce = 1$ occurs mid-epoch, the precision will switch then too. This is where the coarse-grained nature of the new policy is demonstrated.

History: η

The history parameter ensures that only recent and relevant information is considered when deciding to reduce the precision. This metric is related to a precision, so if a convolution's precision changes, the history is cleared. This serves a variety of purposes. The first purpose is to make sure that the analysis computation remains computationally efficient. As the history buffer grows, so does the workload of computing the RAE.

Finally, η allows for control over how relevant the data for reducing precision needs to be. If the user wants to only consider whether the precision should be changed based on relevant recent data, the size of the history buffer can be reduced to reflect this. Having too large of an η will increase the impact of outliers on the decision to reduce the precision. This is similar to what frequency was intended to prevent with this being another approach at limiting the impact of outliers.

Error Threshold: $\epsilon_{threshold}$

The error threshold has already been introduced in Sec. 6.3.4. Here additional details on $\epsilon_{threshold}$ are presented.

The user must define what is considered an acceptable error threshold for reducing the precision. This threshold related directly to the $\text{RAE}_{difference}$, putting an upper bound on the acceptable error. Because $\text{RAE}_{difference}$ is a relative metric, the absolute values of the convolution itself are considered irrelevant. Therefore the threshold is static throughout training and regardless of the type of convolution (forward, gradient w.r.t. weights, gardient w.r.t. inputs) that is being performed. There are works that suggest that throughout training and depending on the type of convolution employed the impact of the convolution (and therefore impact of quantzation) will fluctuate. No attempt has been made to account for this as it would either add complexity or require the user to excessively tune hyperparameters. That being said, there is definitely space to explore a more dynamic, adaptive thresholding function. Implementing such a function is definitely possible to implement in the current ecosystem.

6.4 Evaluation

The aim of `GM` was to create an extended policy to synergize with MuPPET. As a result, just like with MuPPET, the focus of `GM` is to evaluate the acceleration provided while maintaining competitive FP32 final accuracy.

The field of quantized training with a full precision target network is a quite a niche field, limiting the amount of direct state-of-the-art competitive works. As a result, MuPPET is used as the primary baseline reference point for `GM`. Although the `GM` can be configured to produce a mixed-precision final network, this is not the primary focus of the evaluation to keep the comparison to MuPPET as direct as possible.

As stated in the introduction of this chapter, `GM` is not yet complete. The hyperparameter tuning is not yet complete and needs more work for a proper evaluation and analysis. Therefore, the results presented will be based on the analysis and data collected so far. The effect of the error metric is at the forefront of the analysis performed. ϵ was prioritized over τ, because out of ϵ and τ, τ's hyperparameters have the potential to limit the performance benefits obtained, but will not be able to derail convergence. The impact of η is evaluated as well, but due to time constraints, no data has been collected for tuning $\tau_{threshold}$, ω and ψ.

6.4.1 Implementation

`GM` is adapted to use intrinsic benefits of the FPGPT convolution kernels. As a result, all the results are based on the FPGPT performance model. Regarding the training of the networks to verify accuracy, all runs have been trained using a heterogeneous CPU-GPU system. The reason everything is executed on the GPU is due to the amount of time it would take to generate all the bitstreams required for this kind of training. To get the best performance no performance penalty will be accepted when merging the convolutions in the FPGPT DSE. Therefore for each convolution group (potentially for each kernel), for each precision a separate bitstream needs to be compiled. As stated in Ch. 5, a compilation on the setup used in that chapter can take between 1-4 days. Without the bitstream library in place, this is a prohibitively long waiting time.

The FPGPT convolution kernels have been tested to ensure they produce the same results as the GPU in question and are able to fully train a network. This ensures the accuracy results produced will be a genuine reflection of what the implemented system would produce. To properly compare to this baseline, all experiments for MuPPET have been recollected to perform a fair comparison.

The assumed hardware for the results presented in this chapter is running this design on an Xilinx Alveo U250 datacenter card. The CPU in this heterogeneous setup is an Intel Xeon Bronze 3106. As FPGPT and MuPPET were designed to natively integrate with PyTorch, `GM` has been created to do so too. As a result, all training work has been done in PyTorch.

6.4.2 Error Metric Analysis

The key focus of `GM` is on the new metrics that were introduced. Between ϵ and τ, τ's hyperparameters have the potential to limit the performance benefits obtained, but will not be able to derail convergence. ϵ's hyperparameters on the other hand, if improperly tuned, can derail convergence entirely.

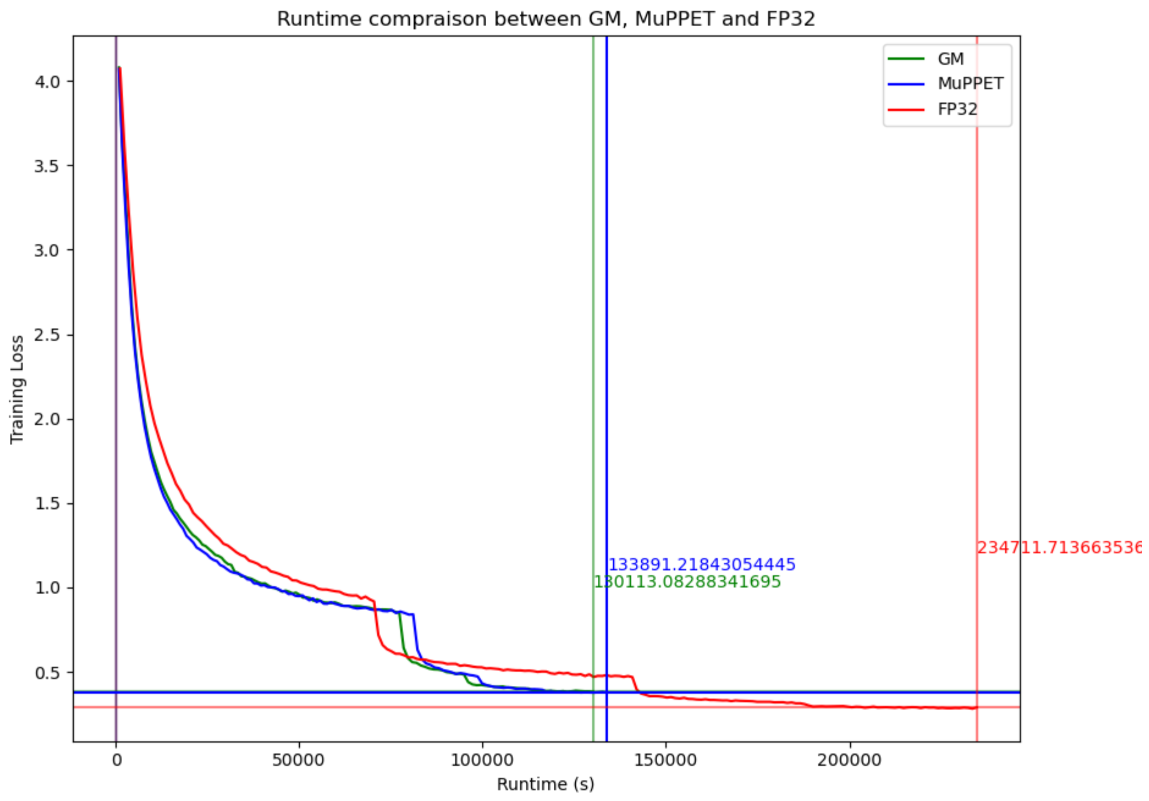
Fig. 6.3 ($\epsilon_{threshold} = 0.0005$) demonstrates the training loss of `GM` in green, MuPPET in blue and a standard FP32 run in red relative to network runtime. For these hyperparameter sets only $\epsilon_{threshold}$ was varied. $\epsilon_{total} = 100\%$ for the tests performed. Additionally, $\omega = 1$, $\psi = 3$, $\eta = 10$ and $\tau_{threshold} = 0.0$. The reason $\tau_{threshold} = 0.0$ is to separate the performance metric from the error metric analysis.

In Fig. 6.3, the loss for MuPPET and `GM` is higher than for FP32 training. As a reminder, this was also seen in Ch. 3, but the Top1 test accuracy was not negatively affected by this same gap in training loss.

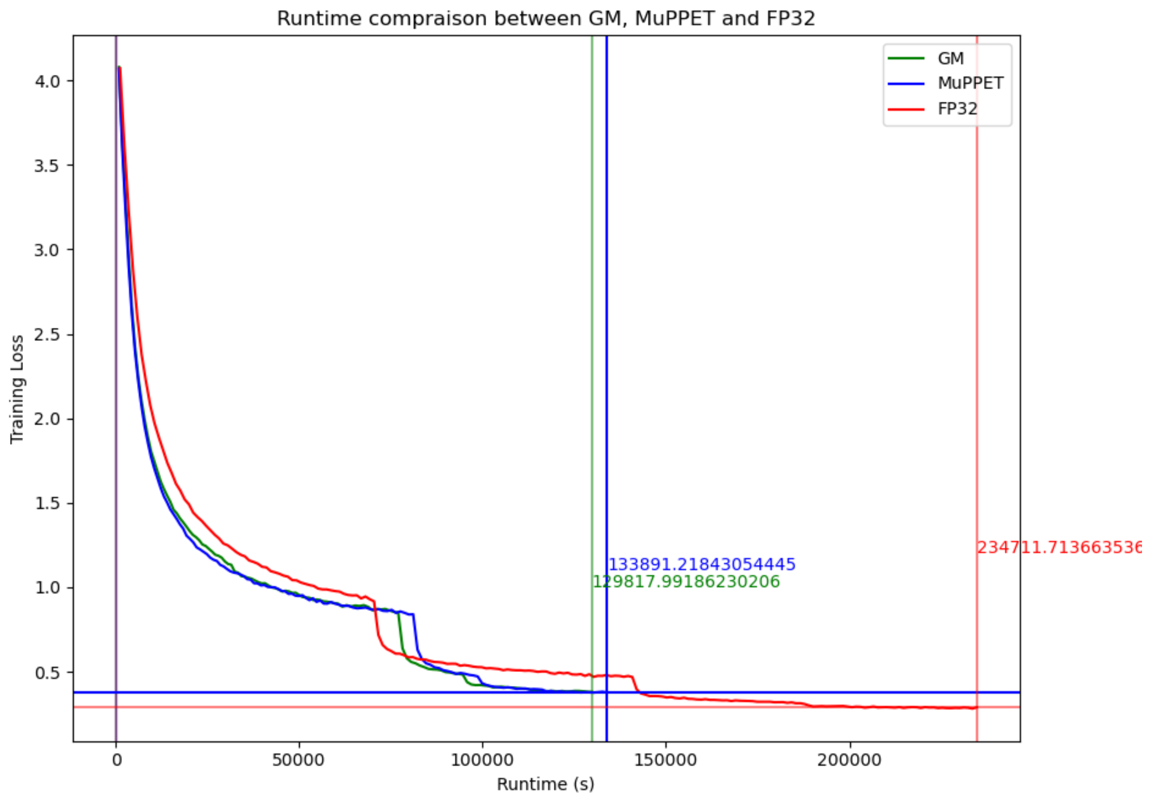
As shown in Fig. 6.3a, there is acceleration potential with a matched training loss to MuPPET, even with the smallest value of $\epsilon_{threshold}$. This ability to accelerate training even with such a small value for $\epsilon_{threshold}$ demonstrates the potential that `GM` has to offer. The performance is exclusively due to lowering the precision of conv1 Forward, as that was the only convolution that had an RAE$_{difference}$ under $\epsilon_{threhsold}$.

Fig. 6.3b ($\epsilon_{threshold} = 0.01$) has slightly better performance compared to Fig. 6.3a. For $\epsilon_{threshold} = 0.01$ as well, only the conv1 Forward convolution has its precision reduced during training, but it happens earlier than for $\epsilon_{threshold} = 0.0005$. This demonstrates that tuning ϵ will enable the ability to switch precision earlier while still maintaining a high accuracy output. Going up even further to Fig. 6.3c ($\epsilon_{threshold} = 0.05$), further performance improvement is demonstrated. In this setting, more convolutions are being quantized outside of conv1 Forward, leading to greater acceleration, while still achieving comparable Top1 test accuracy to both MuPPET and FP32 training.

Due to time constraints, no more data was collected to complete the investigation of $\epsilon_{threshold}$. Further testing needs to be done to investigate the impact of using a too high value for $\epsilon_{threshold}$. The intuition is that this would at the minimum impact the final accuracy and potentially derail training entirely.

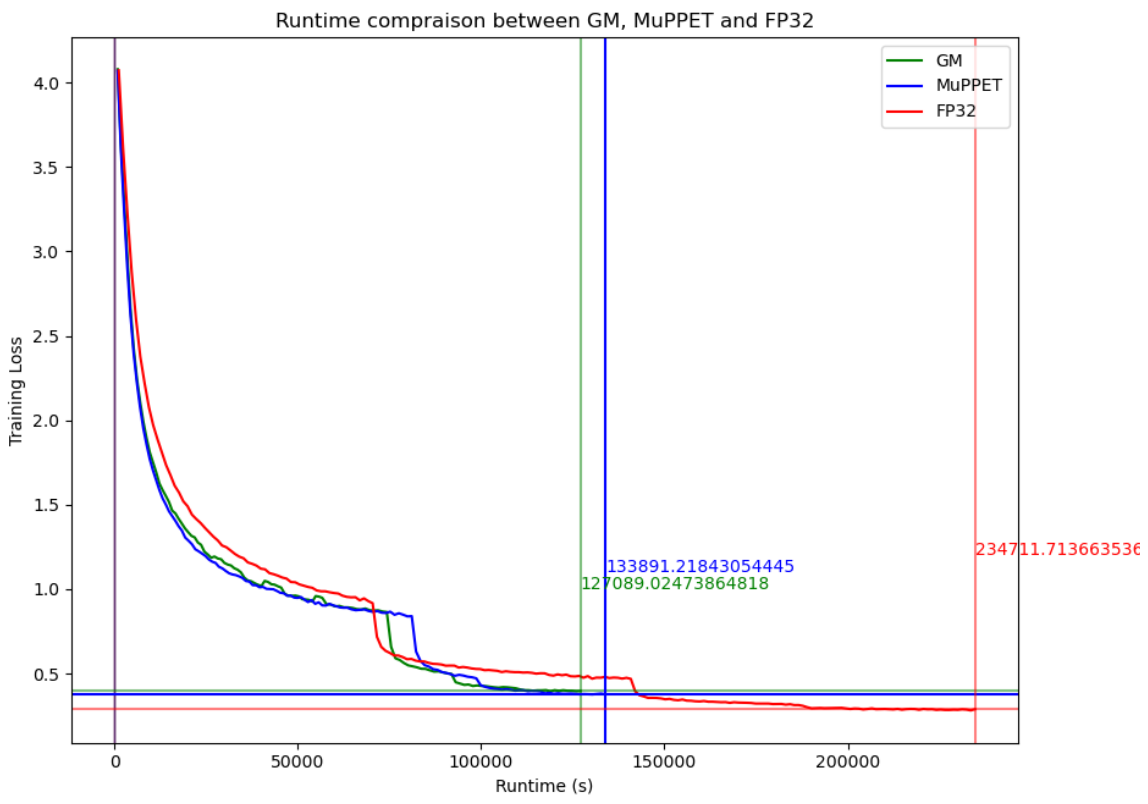


(a) $\epsilon_{threshold} = 0.0005$



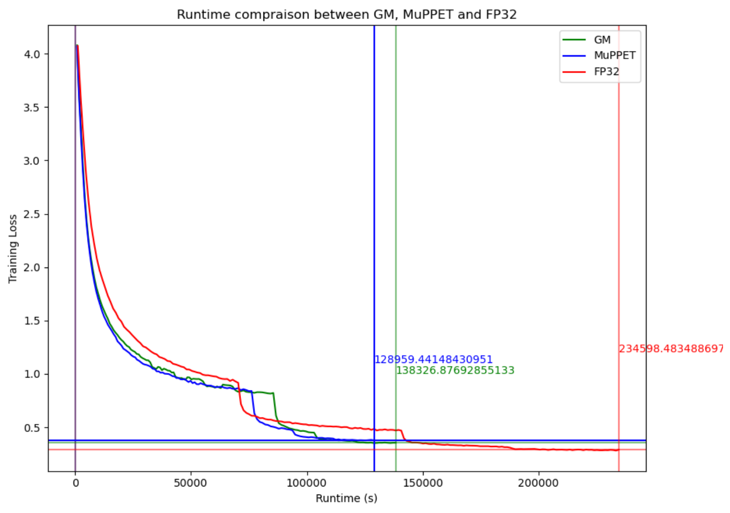
(b) $\epsilon_{threshold} = 0.01$

Figure 6.3: Runtime performance vs. training loss of `GM`, MuPPET and FP32 for various values of $\epsilon_{threshold}$ for ResNet20 on CIFAR-100

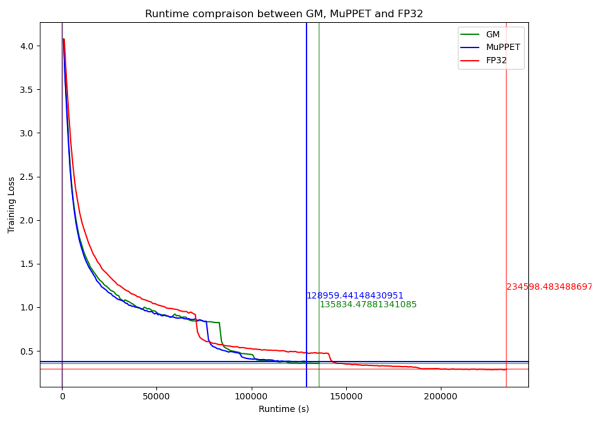


(c) $\epsilon_{threshold} = 0.05$

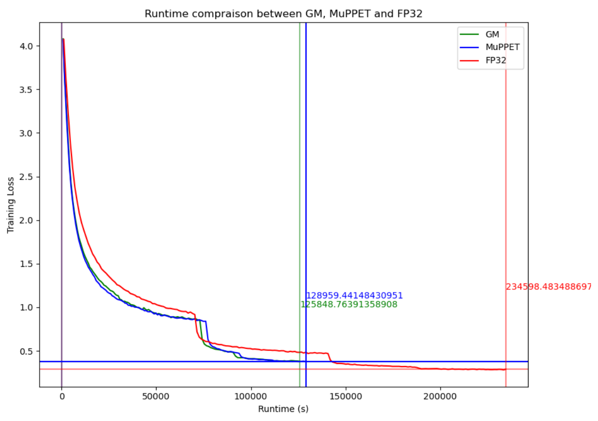
Figure 6.3: Runtime performance vs. training loss of `GM`, MuPPET and FP32 for various values of $\epsilon_{threshold}$ for ResNet20 on CIFAR-100



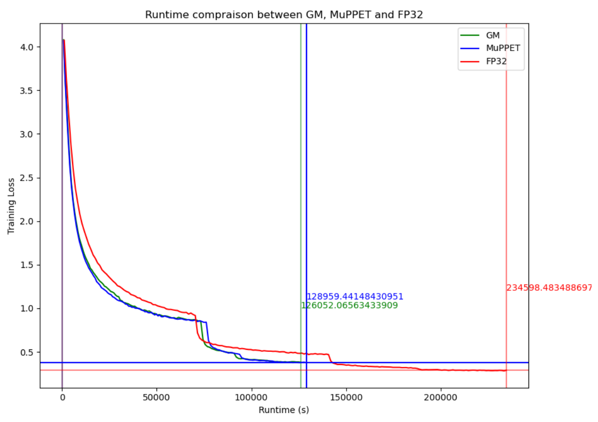
(a) $\tau_{threshold} = 0$



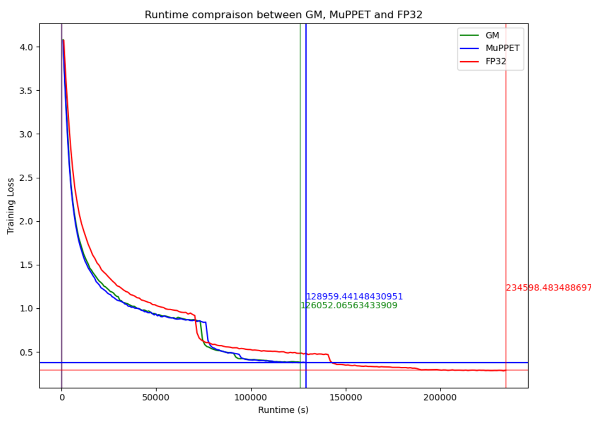
(b) $\tau_{threshold} = 0.0001$



(c) $\tau_{threshold} = 0.001$



(d) $\tau_{threshold} = 0.01$



(e) $\tau_{threshold} = 0.1$

Figure 6.4: Runtime performance vs. training loss of GM, MuPPET and FP32 for various values of $\epsilon_{threshold}$ for ResNet20 on CIFAR-100

6.4.3 Performance Metric Analysis

All graphs shown in this section have a $\epsilon_{threshold} = 0.06$, $\omega = 1$, $\eta = 10$ and $\psi = 3$. $\epsilon_{threshold} = 0.06$ was selected instead of the best value of 0.05 to demonstrate how $\tau_{threshold}$ is able to bring `GM` training back to accelerated times. This is demonstrated comparing times for $0.0001 < \tau_{threshold} \leq 0.1$ to $\tau_{threshold} = 0$ and $\tau_{threshold} = 0.0001$. Fig. 6.4 demonstrates that even if $\epsilon_{threshold}$ is set to cause worse performance, a well tuned $\tau_{threshold}$ can allow `GM` to go back to outperforming MuPPET. That being said, once a reasonable $\tau_{threshold}$ is selected, the exact value seems to have minimal benefit, as long as it is not too strict or too lax.

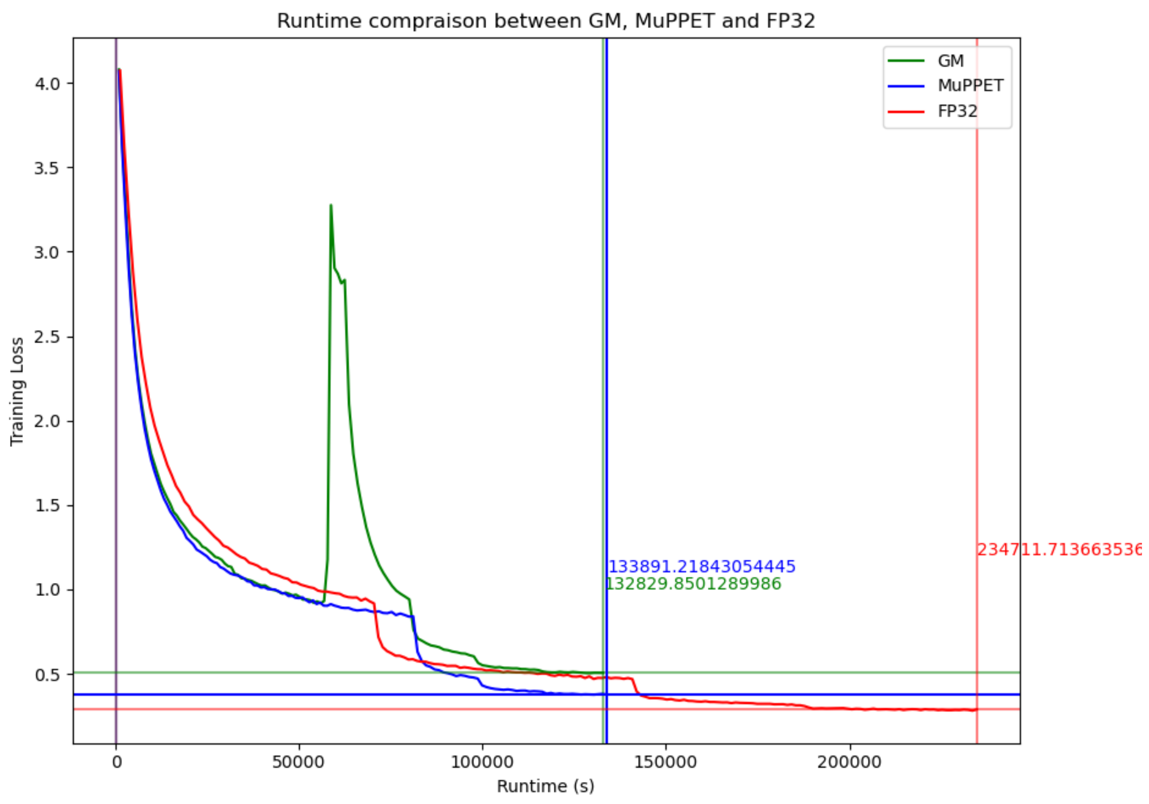
6.4.4 η Analysis

Another hyperparameter other than $\epsilon_{threshold}$ that can impact the final accuracy of the network is η. With a large η, the impact of potential outliers increases. If too many iterations with a minibatch that happens to produce a low $\text{RAE}_{difference}$ occur and the history buffer is large enough to store them all, they might cause a premature or erroneous reduction in precision.

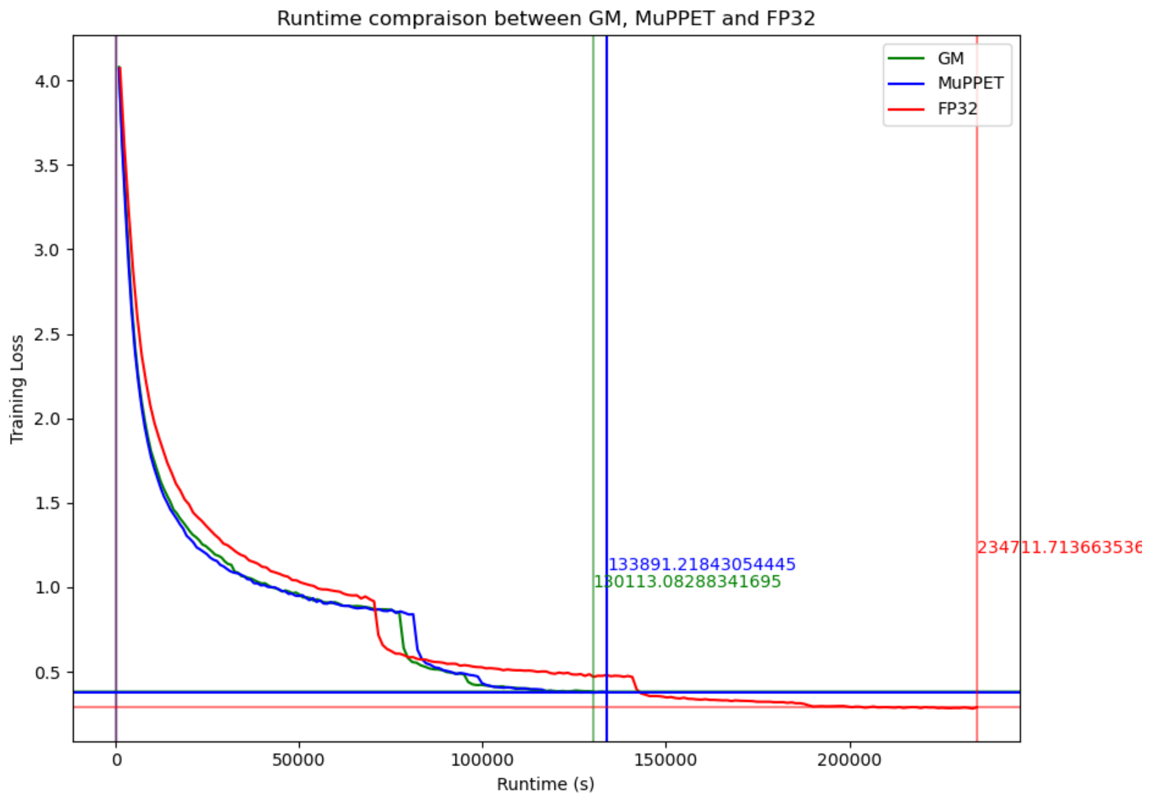
Analyzing the impact of the η hyperparameter was done on a conservative value of $\epsilon_{threshold} = 0.0005$. This was done to make see if impact of η could affect convergence and decrease the final network accuracy even if the quantization error threshold was very low. The other hyper-parameters were set to $\epsilon_{total} = 100\%$, $\omega = 1$, $\psi = 3$, and $\tau_{threshold} = 0.0$.

Selecting an $\eta = 200$ with $\omega = 1$ essentially means that no 'tick' is ever forgotten throughout training. As shown in Fig. 6.5a, convergence is drastically affected around 50,000 seconds. Although the loss value recovers to a certain degree, the training loss is notably higher than for MuPPET along with the Top1 Test accuracy. Additionally, no runtime acceleration is achieved either. Comparing this to Fig. 6.5b, setting $\eta = 10$ results in stable training with the `GM` curve looking like MuPPET with some acceleration.

This demonstrates that tuning this hyperparameter has an impact on convergence and the final accuracy. Additionally it demonstrates that there clearly are outlier minibatches that can

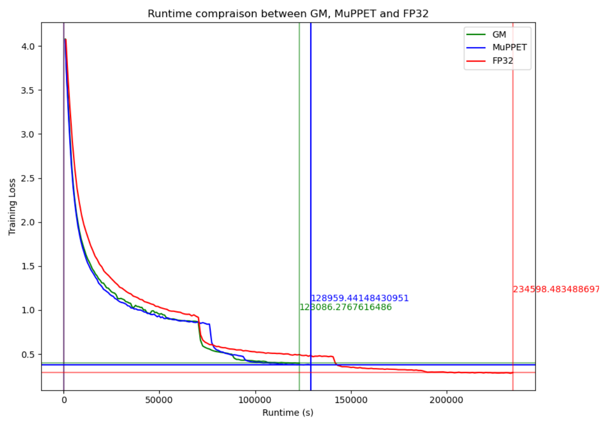


(a) $\eta = 200$

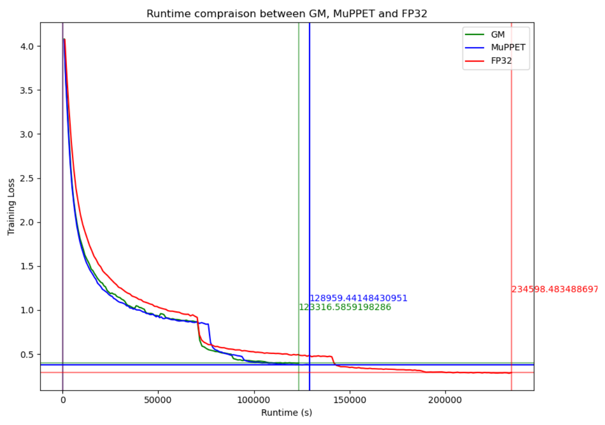


(b) $\eta = 10$

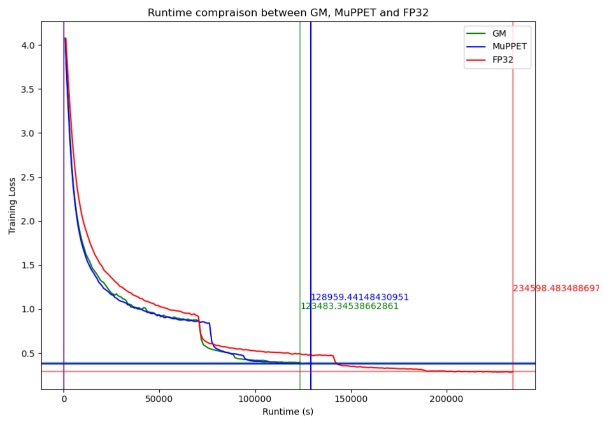
Figure 6.5: Runtime performance vs. training loss of `GM`, MuPPET and FP32 for various values of η for ResNet20 on CIFAR-100



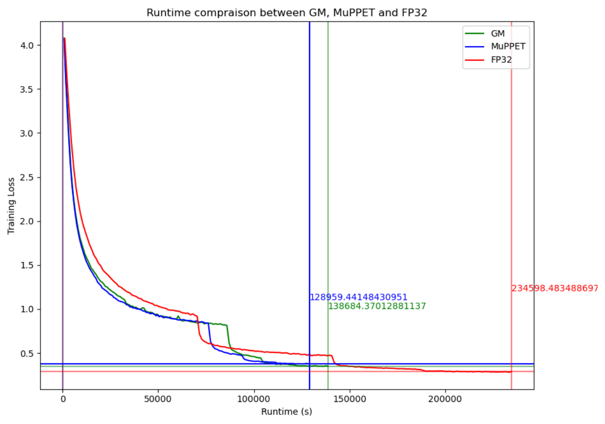
(a) $\psi = 1$



(b) $\psi = 3$



(c) $\psi = 5$



(d) $\psi = 10$

Figure 6.6: Runtime performance vs. training loss of `GM`, MuPPET and FP32 for various values of η for ResNet20 on CIFAR-100

disturb training if too many are considered. What is still to be explored is whether η can have an impact on the runtime of the network by enabling earlier switching, similar to adjusting $\epsilon_{threshold}$.

6.4.5 ψ Analysis

All graphs shown in this section have a $\epsilon_{threshold} = 0.05$, $\omega = 1$ and $\eta = 10$.

Initially looking at Fig. 6.6a, 6.6b and 6.6c, it demonstrates that $\psi = 3$ provides the best results, but even $\psi = 5$ provided acceleration over $\psi = 1$. This demonstrates dropping precision at the first hint of an acceptable error gap might be leading to parts of the training landscape that will require more epochs to train completely. The overall impact is less drastic than with other metrics, but should nonetheless be noted and tuned for optimal training.

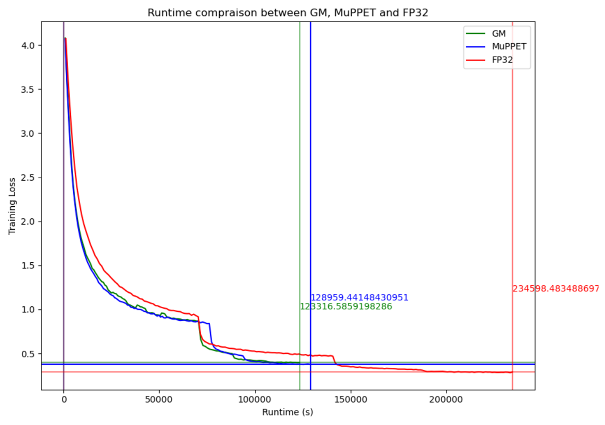
Looking at Fig. 6.6a, 6.6b, 6.6c compared to 6.6d it becomes clear that setting the wrong value for ψ can cause training to derail. In this case setting ψ too high, there's a chance no decision is made at all and training will take longer than compared to MuPPET. One thing to note is that in this scenario, $\psi = \eta$, meaning every encounter in this current history must be flagged. Further experimentation where η is varied w.r.t. ψ to test the effect of the $\eta : \psi$ ratio on the acceleration of training would provide greater insight into the functionality of ψ.

6.4.6 ω Analysis

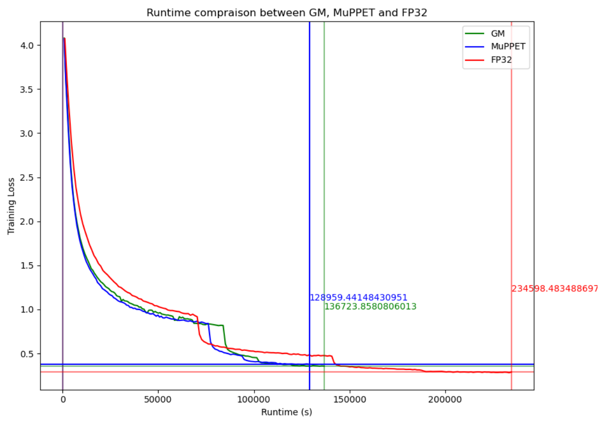
Finally ω is analyzed to see its effect on accelerating training. To avoid what happened for ψ, various values of η are considered to ensure the results are not affected by the ratio between ψ and η. Fig. 6.7 demonstrates the impact of increasing ω to 3. As is clear, in both the scenarios where $\eta = 5$ and $\eta = 15$, setting $\omega = 3$ causes training to become worse than MuPPET. Further analysis demonstrates that this has the most prominent effect on the early epochs, extending training by being unable to switch out of 8bit MuPPET training setting.

Looking at Fig. 6.8 demonstrates that going for a higher value of $\omega = 10$ seems to mitigate the negative effects induced by $\omega = 3$, but still under performs compared to $\omega = 1$. As a result, no broad statements about the value of ω or the ratio of $\omega : \eta$ can be made.

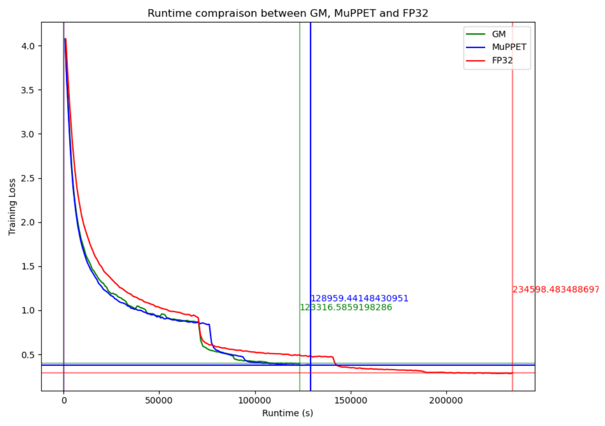
Clearly although intuitively increased ω would lead to an increased flexibility leading to switching at more appropriate point in time, this preliminary data analysis demonstrates this not be the case. There is a argument to be made that increasing ω should go with an according scaling of ψ to switch at roughly the same rate as with $\omega = 1$. Yet when $\omega = 1$ and $\psi = 1$ `GM` training outperforms MuPPET, yet as shown in Fig. 6.7b where $\omega = 3$ and $\psi = 3$ does not manage the same acceleration. As the relation between η and ψ need to be further understood, tuning ω with respect to either of those hyperparameters is only appropriate once the $\eta : \psi$ relationship is better understood.



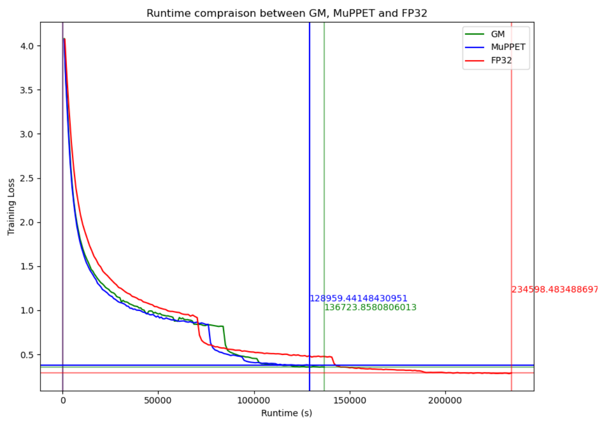
(a) $\eta = 5, \omega = 1$



(b) $\eta = 5, \omega = 3$

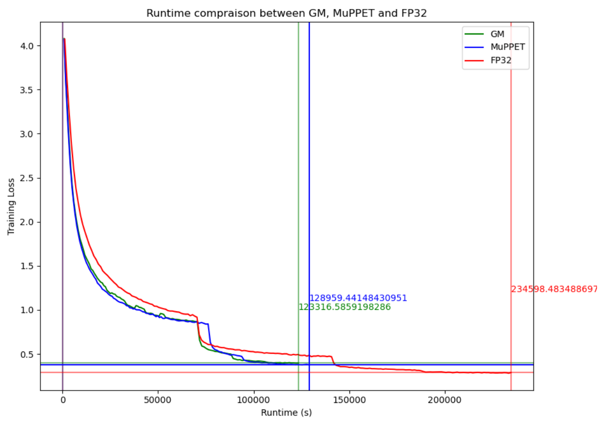


(c) $\eta = 15, \omega = 1$

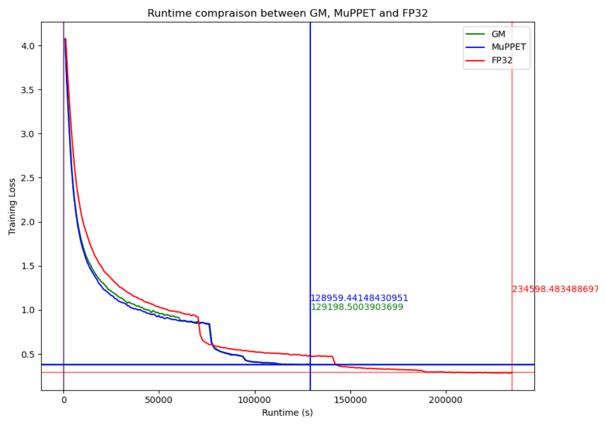


(d) $\eta = 15, \omega = 3$

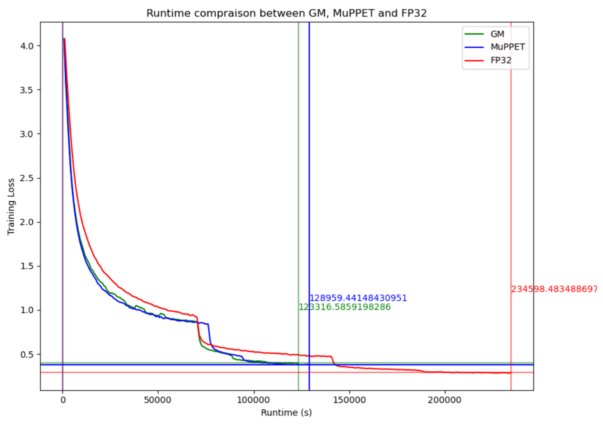
Figure 6.7: Runtime performance vs. training loss of `GM`, MuPPET and FP32 for various values of η for ResNet20 on CIFAR-100



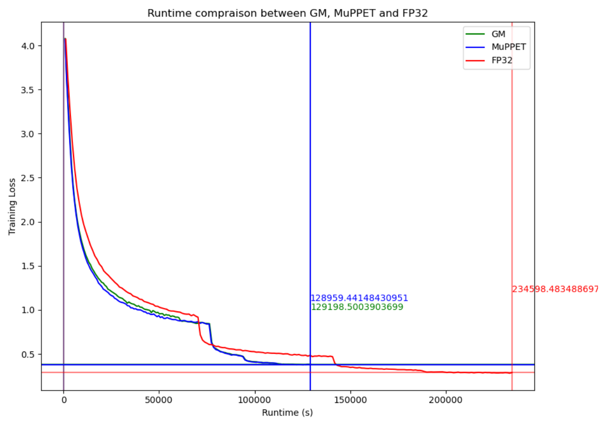
(a) $\eta = 5, \omega = 1$



(b) $\eta = 5, \omega = 10$



(c) $\eta = 50, \omega = 1$



(d) $\eta = 50, \omega = 10$

Figure 6.8: Runtime performance vs. training loss of `GM`, MuPPET and FP32 for various values of η for ResNet20 on CIFAR-100

6.5 Conclusion

This chapter has presented the intuition behind GM and some initial results demonstrating the potential GM has, already improving on MuPPET's acceleration potential. Even with the limited hyperparameter tuning that has been done so far, there is clearly a potential for acceleration by analyzing certain components individually. Although the potential has been demonstrated, further analysis is required to be able to assert the full benefit provided by GM. A complete analysis of each hyperparameter must be performed to demonstrate the impact each hyperparameter has on performance and convergence. As discussed in Sec. 6.4 hyperparameters η, ψ, and ω have a complex relationship that needs to be further investigated. Ideally this will lead to a formulaic approach that is able to tie the hyperparameters together, much like in MuPPET.

Furthermore, an analysis on larger datasets like ImageNet and multiple networks is required as well. Generalisability across networks and datasets was a key factor for MuPPET and will be for GM as well.

Beyond just finishing the analysis, there is a potential for future work regarding GM. Creating a more algorithmic and unifying approach that supersedes tuning of each parameter would be impactful. MuPPET operated on a dynamically adjusting metric with great success, so the same could be applied to GM. By investigating how the various hyperparameters interact with each other and how they might affect convergence at different points throughout training could lead to a more effective way to implement GM. The field of hyperparameter tuning is an established research focus and could be exploited to aid in understand this relationship.

Finally, the precisions used thus far are simply the block-floating point that was used for MuPPET. There are a plethora of newer datatypes such as those presented in [104, 54, 53, 16] could be utilized to explore more extreme quantization techniques.

Chapter 7

Summary and Conclusions

This thesis set out to achieve three sets of goals. They pertain to enabling quantized training, convolution focused hardware architecture and creating an ecosystem to allow for easy adoption and testing of quantized CNN training and training on FPGAs. Through the variety of works presented, PyTorch extensions and ecosystems have been developed that allow users to investigate reduced precision training on GPUs or FPGAs, depending on the infrastructure they have available. All the works presented have the potential combine with other works in the field as they natively integrated into one of the most popular ML training frameworks, PyTorch. With the help of the FPGPT ecosystem all the works presented throughout the thesis are easily accessible and implementable, even if the users are ML and FPGA novices. The works presented here provide a complete infrastructure allowing researchers to easily use and benefit form the acceleration gains presented. The design ensures users' ability to explore new ideas are not impeded by the benefits provided. This of course is only one step out of potentially many. There are limitations to the work presented, but they demonstrate the path forward leading to a multitude of future opportunities.

7.1 Summary of Thesis Achievements

The works presented throughout this thesis address one or more of the thesis goals. As not each work addresses all goals, this thesis concluded with a work bringing all aspects together. Each individual goal is addressed separately for clarity. This servers as a summary of what each work has contributed to the thesis, but from a new perspective of the thesis' goals.

7.1.1 Quantized Training

The specific goal expressed for quantized training was to accelerate training of a CNN that produces a 32bit floating point (FP32) model. The training process must rely on quantization techniques to accelerate the training process, using wall clock time to measure performance. Additionally, there is the push to easy adoption. This entails all works to integrate with an existing ML framework (PyTorch and Caffe specifically) as an extension to the framework. If it operates as an extension to these frameworks, other acceleration techniques are not prevented from being used in tandem. Additionally, users will already be comfortable with the framework, making adoption smoother.

The first work related to quantized training, MuPPET, addresses this goal most directly. MuPPET extends the native PyTorch framework to employ a custom training strategy. At its core it addresses this goal by providing the required metric to track how training is progressing known as gradient diversity [33]. Rooting its intuition in how learning rate affects training, increasing quantization is used as a substitute for decreasing learning rate. As demonstrated by its results, this work provides acceleration to the training process, producing FP32 designs that have highly competitive accuracy. Because the gradient diversity metric is not intrinsically related to quantization, any form of quantization can be applied. Furthermore, as shown in the work, the metric is agnostic to dataset and architecture. This ensures the users have the freedom to utilize any network and dataset while adopting MuPPET. Of course this does come with limitations. The primary limitation is that this works focuses on a variety of precisions between 8bit fixed point (8FXP) to FP32, but does not have hardware to support these intermediate precisions.

This is where the FPGPT work (and by extension Barista) comes into play. FPGPT provides a modular architecture for convolutions on FPGAs. As FPGPT targets FPGAs, it is able to adapt to and execute any custom precision as long as the design is provided. FPGPT specifically focuses on enabling FP32 convolutions, as these are required for MuPPET and entail the most complex design compared to fixed point computation. Nonetheless, this kernel unlocks the ability for MuPPET to utilize all desired precision formats. That being said, FPGPT is limited as it has not optimized for these lower precisions, making the realizable benefit of MuPPET below its potential.

Finally, `GM` is designed to be the convergence point between these two preliminary works. `GM` at its basis combines FPGPT with MuPPET. This is possible as they are both extension of the PyTorch framework, making them intrinsically combinable. The novel contribution of `GM` is exploring the synergy that can be exploited combining FPGPT with MuPPET. As discussed, MuPPET takes a broad scope view analyzing how training progresses as a whole. On the other side, FPGPT is able to collect stats on the individual convolutions as they are happening. This provided the basis for `GM`, to reduce precision based on localized information (FPGPT) while maintaining proper training technique (MuPPET). `GM` represents a complete instance of quantized training with realistic implementation. `GM` does suffer from all the limitations that affect FPGPT and MuPPET. Additionally, the current approach is quite empirically based, and a more generalized approach should be implemented to react to training progress, especially for the precision lowering aspect. As stated in Ch. 6, the analysis of `GM` is yet to be completed, but initial evaluation showed promising results.

Across all of these works, an easy to adopt method to accelerate training through quantized hardware is presented. By implementing a novel training scheme that natively integrates with PyTorch that can utilized both GPUs and FPGAs, acceleration though quantized training is achievable and easily implementable for any user.

7.1.2 Hardware Architecture

The goals of the hardware architecture, were all individually addressed by FPGPT and also attempted by Caffe Barista. Barista fell short on having a proper throughput of one and supporting ImageNet training. This was due to the fact that the memory to fabric interaction was not optimized to utilized burst reading. As a result, the system was constantly stalling, waiting for data to arrive. Additionally, ImageNet was not tested, as at the CPU bottleneck that limited the CIFAR-10 performance would only be exacerbated even further for larger inputs like in ImageNet. Barista was able to serve as a demonstration for the path FPGPT needed to pursue to meet the objectives set out for the desired hardware architecture.

In the process of achieving all the goals set out for the convolution hardware, FPGPT was also able to become a state-of-the-art performing work in its field. It covers a broad base of capabilities being applicable to various network architecture across multiple datasets, adapting to all workloads is encounters. Furthermore, it enabled further research such as `GM`. There were clear limitations with its reduced precision optimization as well as its chosen implementation method.

7.1.3 Ecosystem

The ecosystem requirements vary slightly depending on the target application, but the main objective was easy implementation and adoption as well as accommodation of a variety of networks and datasets.

MuPPET was able to allow for easy integration by presenting itself as a PyTorch extension allowing for the rest of PyTorch to function as before. Furthermore, testing was done to demonstrate it could accommodate a variety of networks and datasets.

FPGPT needed a far more extensive external ecosystem to meet the set out objectives. As FPGPT is effectively the mature version of Barista, the FPGPT ecosystem will be the focus. For FPGPT specifically, there was the extra objective of adapting to the incoming workloads and natively integrating into an ML framework like PyTorch. FPGPT's ecosystem was able to achieve the objectives in a variety of ways. By natively integrating with PyTorch, ease of use and uptake were achieved. The compile time flexible hardware design provided the infrastructure to be able to support a variety of networks and datasets as well as roughly adapt to incoming workloads. The runtime flexibility allowed the FPGA bitstreams to closely adapt to the incoming workload of each convolution. The DSE ensured the user could easily determine which kernel sizes that were required for successful execution. This additionally makes the use of FPGPT easier as the user does not need to understand the design at all, they can just start the DSE and wait. This makes the DSE the key factor to being able to meet all the requirements set for the ecosystem. Only due to the high accuracy of the DSE is FPGPT so easy to use, as it takes away any need for experience with FPGAs.

7.2 Conclusions

The goals set out at the beginning of this thesis have been met, but this does not entirely reflect the impact of the works and the thesis as a whole. There are two key areas in which the impact can be measured. They consist of the algorithmic impact and the hardware impact of the work presented.

7.2.1 Algorithmic

From the algorithmic perspective the impact comes from MuPPET and `GM`. Both works are purpose built to be largely perpendicular to existing research. They also contain a degree of modularity that allows for modification and updating. This refers to the fact that both the tracking metric (graident diversity) and quantization approach (uniform stochastic quantization) are entirely replaceable as research in those fields progresses.

For MuPPET and `GM`, the guiding metric is not directly linked to the method of quantization. This allows for the freedom to explore new quantization techniques while still benefiting from the acceleration provided by both MuPPET and `GM`. Being able to adapt so fluidly regarding guiding metric or quantization technique ensures the impact of both these works is not static. Additionally, MuPPET is network and dataset agnostic, so it can still be applied as new network architectures and datasets are developed and popularized. The ability to grow with new research combined with the seamless connection to PyTorch ensures MuPPET will manage to remain relevant and impactful.

The field of using quantization to accelerate training is one that as a whole shows promise. An increasing number of hardware vendors are generated reduced precision cores. Additionally there is an increasing amount of research regarding the impact that each convolution has on the convergence properties of training CNNs. MuPPET and by extension `GM`, will be able to integrate this new knowledge into their operational metrics. Additionally, with the growing development of custom number representations, the groundwork laid by MuPPET and `GM` can be taken even further. A greater understanding of the impact of a layer on the convergence can be used to better inform the how and when to quantize convolutions in an adaptive manor.

The algorithmic work here has demonstrated that with existing information about CNNs we are already able to make notable performance improvements to CNN training. Looking at the other research occurring in the field of understanding and training CNNs, the potential of MuPPET, `GM` and similar works will only be accentuated. This field has a large potential that is only just being tapped into.

7.2.2 Hardware Architecture

Regarding the hardware architecture, the benefit of onboarding the im2col process along with other tensor transformations to perform a GEMM has been demonstrated. This design enables flexible application of the FPGA supporting complex networks with the ImageNet dataset format supported too. Furthermore, the DSE and native PyTorch integration makes use and implementation of the hardware architecture quite simple. Additionally, the possibility to build a library that sources acceptable kernels that are already prebuilt allows for increasingly easier use of FPGPT. A variety of optimization works such as FeCaffe are even able to work with FPGPT, showing it is not limited to PyTorch providing benefits beyond the integration provided by FPGPT itself.

As FPGA instances are available on cloud computing services like AWS, FPGA hardware is now cheaply and readily available. FPGPT requires no changes to be made to the specific training setup a user might have. If the bitstream library is populated, even the timely compilation stages can be skipped allowing for a direct move into training. All this demonstrates that FPGPT is an initial step to creating an ecosystem that makes FPGA-based CNN training an increasingly viable solution for any user, not just FPGA specialists.

Although the work presented in this thesis for FPGAs has demonstrated that FPGAs can be transformed to be user friendly and easy to adopt, their realistic implementation must be considered. As demonstrated in Sec. 5.6.4, FPGAs need to change significantly to start matching GPU performance. FPGAs have a lot of potential especially due to their ability to adapt to the problem at hand. For training on platforms that have limitations, especially power limitations, FPGAs are definitely the go-to solution. Additionally they boast a larger fault tolerance in harsh environments. For applications like spacecrafts and satellites, power efficiency and harsh environment fault tolerance are critical factors, making FPGAs uniquely equipped to operate where GPUs are currently struggling to perform. But in many modern ML applications, most of the training occurs in massive data centers. In these settings time is prioritized over energy efficiency and as it stand right now, FPGAs will be left by the wayside. The upcoming advances for FPGAs such as the AMD Versal Architecture [105] do boast 7nm technology (versus 16nm on the Xilinx Alveo U250 used in this work) and have a vast array of dedicated IP blocks to accelerate workloads even more. With these advances FPGAs are encroaching on the dominant performance of GPUs. These advances will largely be due to available funding and popularity, which is why accessibility and ease of use, for example through an automated ecosystem, is a key factor in the future of FPGAs in ML training.

Beyond the physical hardware itself, the tools used in this thesis are not appropriate for high performance architecture design. Vitis/Vivado HLS were the tools used to design all the architecture described. The idea behind HLS is to allow users to write code in a high level language, C++ in this case, and that gets compiled to Verilog. The Verilog is then compiled down to hardware. The Verilog to hardware compilers boast strong reliability and performance as they have been used in industry and research for a long time. HLS on the other hand was unable to deliver on its promises. Due to overly conservative memory analysis, poor understanding of scope and overly conservative dependence analysis high performance designs are incredibly difficult to implement. The final HLS code used for FPGPT was structured in a way to produce the desired hardware architecture, but resembled common Verilog structure more than common C++ structure. There is a lot of work that needs to be done before HLS can be considered a viable option for designing complex high performance hardware.

7.3 Limitations and future works

The works here have provided meaningful progression towards promoting quantized training, FPGA-based training and the merger between both of these training applications. Nonetheless, there are limitations to the works presented. These limitations do also introduce the potential routes for future works and improvement on the groundwork already laid down. Each individual work is addressed with regard to its limitations and future work, indicating where they individually fall short and how this can be addressed.

7.3.1 MuPPET

MuPPET is the key algorithmic work in this thesis, with its basis rooted in gradient diversity. At the time this research was performed gradient diversity was a SotA metric for understanding training progression. As the understanding of CNNs is developing it is no longer necessarily the best metric for this purpose. This opens up a path to exploring if new techniques in tracking training progression. As the understanding of idealized training is better understood, accordingly an improved tracking metric could be developed. This improved tracking method would be able to more accurately adjust the network precision during training.

Furthermore, MuPPET currently has a broad focus of treating the entire network as a monolith. `GM` was an initial attempt at introducing convolution specific analysis. Additionally, although MuPPET has been shown to be applicable to a variety networks and datasets, this field is rapidly developing. MuPPET is shown to work across various architectures, but there is no guarantee this will remain. Both of these points indicated the potential utilize localized information from individual or groups of convolutions. This localized analysis can apply to both aid the overall training progress tracking and individual layer development.

As the research in quantized training continues, an increasing amount of number representations are being developed. In many circumstances these all have a specific focus, such as capturing more information with unique scaling methods or counteracting the quantization error by implementing a counter bias. Another potential future work is merging these modern training techniques with MuPPET and `GM`'s adaptive tracking methodology. Based on the assumption that local and global tracking metrics can give insight into training, these insights can be used to select or tailor one of the many novel quantization methods.

Additionally, the focus can be moved beyond just reducing the precision of only convolutions. There are a variety of other layers that can be reduced in precision to provide greater acceleration beyond just convolutions. Especially when convolutions go to very low precisions, the acceleration benefit of other layers becomes relatively more impactful.

Regarding the merger with PyTorch, although existing models and training setups can be left predominantly undisturbed, parts of the training schedule are affected. The total epochs as well as the learning rate are replaced by MuPPET. This makes MuPPET inherently incompatible with all research progress related to learning rate schedules. As learning rate schedules/systems become more advanced, they could drop the training time through epoch reduction potentially as significantly as MuPPET reduces training time through quantized training. This leads to the final avenue of future work; merging the intended learning rate schedule with the level of quantization. This would allow MuPPET to be fully integratable with all aspects of CNN training.

7.3.2 FPGPT

Caffe Barista is not given its own independent section. Many of Barista's limitations were fixed in FPGPT, and the limitations of FPGPT all cover any Barista limitations.

The first limitation of FPGPT has not been largely touched on in the thesis thus far. All of FPGPT and Barista has been written in Vitis HLS. This inherently poses numerous limitations. The first limitation is the ability to develop the hardware design. Although HLS is intended for easily prototyping designs, it is very weak at producing high performance designs. HLS is currently still too new of a tool to effectively implement many of the features it claims to support. This was a large hindrance during development and has definitely lead to an less efficient design compared to using another language to design the hardware. Additionally, using HLS has made creating a resource model for any component other than DSPs extremely challenging. Changing the compilation parameters has the potential to significantly change how the HLS compiler generates the bitstream for the design. Additionally, HLS has struggled with properly implementing guaranteed backwards compatibility, meaning using a new version of HLS could require significant code redesign. This leads to a future work of turning the key components of design into Verilog (or another HDL like Chisel) blocks that are then stitched together using HLS. This would allow for the high degree control and efficiency of an HDL design with the ease of connection afforded by HLS. Lastly, moving away from HLS would result in a greater ability to optimize the design. Such optimizations would enable both a reduced resource requirement and a high chance to have an increased clock speed.

Beyond the coding aspect of FPGPT, the memory model could be improved. As addressed in the chapter itself, the memory model occasionally drastically over-predicts the utilized memory. There is a direct limit under which the compiler will place memory directly on the fabric saving a potentially large amount of memory. The details of this threshold should be attainable from Xilinx allowing for a more accurate memory model. Combining this improved memory model with an improved resource model from having Verilog components would result in a more accurate and therefore more effective DSE. Ideally this more effective DSE would allow for even greater kernel sharing abilities.

Currently FPGPT does not exploit runtime reconfigurability. Runtime reconfigurability would allow individual super logic regions (SLRs) of the FPGA to be reprogrammed while the rest of the FPGA continues computing. Adding this additional level of flexibility would allow kernel sharing to become more elaborate. Bitstreams for each SLR could be shared, requiring only part of the design to be replaced at runtime. This would allow for the "library" to host multiple kernels for each SLR creating even more custom designs for each convolution without impacting the ability to share bitstreams.

FPGPT focused exclusively on generic GEMM computations. This creates two limitations. The first is that very specific, often occurring convolutions like 1x1 and 3x3 convolutions are not as optimized. As these specific convolutions occur so frequently, custom kernels for these specific convolutions could provide even further optimization. The second limitation is that FPGPT only does GEMMs. Sliding window, winograd and FFT convolutions are not included. Depending on the incoming workload, one of these convolution approaches might be more efficient or higher performance than a GEMM. Seeing as there is already a DSE in place to determine kernel construction based on incoming workload, having additional convolution options would only enrich the DSE. This will allow the user to have even more choice between how much time they are willing to invest in compiling and how fast they would like their networks to be trained.

Lastly, the reduced precision implementation of the hardware architecture is not optimized. The current implementation will accelerate the memory utilization and bandwidth as well as more efficiently utilize the onboard DSPs. It does not effectively utilize the FPGA to share the MAC operations between DSPs and the fabric itself. Additionally, the weights and inputs need to be at the same precision according to the current design. This ties in with the future work to have custom Verilog units for optimized performance as well as runtime reconfigurability for each SLR. The custom Verilog units will allow for better utilization of the fabric, while the individual SLR bitstreams can operate at whatever precision best suits the data assigned to that SLR.

7.3.3 GM

As GM is a combination of MuPPET and FPGPT the limitations presented for MuPPET and FPGPT naturally extend to GM. All the future works suggested for both MuPPET and FPGPT will also help improve the quality of GM. This section will therefore only focus on the GM specific limitations and opportunities.

First and foremost, as detailed in Ch. 6, GM must be investigated in greater depth to be able to fully verify it performs as expected. Although primary investigation was promising, it must be investigated further. The rest of the future work is based on the assumption the rest of the GM evaluation turns out as expected.

Currently the extensiveness of how much information can really be extracted by the FPGPT kernels is not explored, and therefore limited. Very rudimentary testing has been done to see if the current information collection would fit. This limits the potential information GM operates on to lower each convolution's precision. This in turn lowers the acceleration potential of GM. Further work on creating efficiently data collection from the FPGA would allow GM to rise above its current capabilities.

This addresses another limitation of GM. GM is based on empirical hyperparameters manually tuned to operate properly. This limits the ability to guarantee GM can adapt well to other datasets or network architectures. Even if it works well across a range of datasets and architectures, additional hyperparameter tuning is realistically required adding additional burdens the user. Seeing as more information could be extracted from each convolution, this extra information could be tracked and analyzed to create a more generic approach. Just as MuPPET formulates its threshold as an adaptive equation, GM should too. This can initially be based on attempting to predict the non-linear quantization error, but by no means needs to be limited there.

This leads to the final future improvement to GM. This is a combination of a future work presented for FPGPT. Having access to runtime reconfiguration of SLRs could allow for GM to narrowly influence every part of the convolution. This would entail that GM can adjust the precision of the inputs, weights, outputs and intermediate computations as well. Of course the impact of each adjustment must be researched and understood, but once this is done, `GM` and the improved FPGPT design would be able to provide this functionality.

Every section of this thesis has shown that CNN training can grow to be a highly dynamic modular system modifying the precision of its operations for acceleration while maintaining high accuracy.

Bibliography

[1] T. Geng, T. Wang, A. Li, X. Jin, and M. Herbordt, "FPDeep: Scalable Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters," *IEEE Transactions on Computers*, pp. 1–1, 2020. arXiv: 1901.01007.

[2] V. Golovko, M. Egor, A. Brich, and A. Sachenko, "A shallow convolutional neural network for accurate handwritten digits classification," vol. 673, pp. 77–85, 02 2017.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, p. 84–90, may 2017.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[5] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size," 2016.

[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," 2014.

[7] OpenAI, "Gpt-4 technical report," 2023.

[8] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, "A survey of large language models," 2023.

[9] N. P. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. Patterson, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," 2023.

[10] Z. Jia, B. Tillman, M. Maggioni, and D. P. Scarpazza, "Dissecting the graphcore IPU architecture via microbenchmarking," *CoRR*, vol. abs/1912.03413, 2019.

[11] Nvidia, "Nvidia a100 tensor core gpu architecture," whitepaper, May 2020.

[12] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *International Symposium on Computer Architecture (ISCA)*, pp. 1–14, 2018.

[13] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. A. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on gpu clusters using megatron-lm," 2021.

[14] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," *CoRR*, vol. abs/1710.03740, 2017.

[15] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Xu, R. Patel, and M. Herbordt, "FPDeep: Acceleration and Load Balancing of CNN Training on FPGA Clusters," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (Boulder, CO), pp. 81–84, IEEE, Apr. 2018.

[16] L. Cambier, A. Bhiwandiwalla, T. Gong, M. Nekuii, O. H. Elibol, and H. Tang, "Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks," 2020.

[17] E. Park, S. Yoo, and P. Vajda, "Value-aware quantization for training and inference of neural networks," 2018.

[18] M. Yu, Z. Lin, K. Narra, S. Li, Y. Li, N. S. Kim, A. Schwing, M. Annavaram, and S. Avestimehr, "Gradiveq: Vector quantization for bandwidth-efficient gradient aggregation in distributed cnn training," 2018.

[19] A. Finkelstein, U. Almog, and M. Grobman, "Fighting quantization bias with bias," 2019.

[20] A. Fan, P. Stock, B. Graham, E. Grave, R. Gribonval, H. Jegou, and A. Joulin, "Training with quantization noise for extreme model compression," 2021.

[21] Q. Lou, F. Guo, L. Liu, M. Kim, and L. Jiang, "Autoq: Automated kernel-wise neural network quantization," 2020.

[22] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision," in *32nd International Conference on Machine Learning (ICML)*, pp. 1737–1746, 2015.

[23] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training Deep Neural Networks with 8-bit Floating Point Numbers," in *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 7675–7684, Curran Associates, Inc., 2018.

[24] Z. Tao and Q. Li, "eSGD: Communication efficient distributed deep learning on the edge," in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, (Boston, MA), USENIX Association, July 2018.

[25] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," 2020.

[26] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: Experiences on accelerating data parallel training," 2020.

[27] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *ACM Comput. Surv.*, vol. 53, mar 2020.

[28] J. Wu, W. Huang, J. Huang, and T. Zhang, "Error compensated quantized SGD and its applications to large-scale distributed optimization," in *Proceedings of the 35th International Conference on Machine Learning* (J. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, pp. 5325–5333, PMLR, 10–15 Jul 2018.

[29] S. Zhao, H. Gao, and W. Li, "Quantized epoch-sgd for communication-efficient distributed learning," *CoRR*, vol. abs/1901.03040, 2019.

[30] K. Zhao, S. Huang, P. Pan, Y. Li, Y. Zhang, Z. Gu, and Y. Xu, "Distribution adaptive INT8 quantization for training CNNs," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 3483–3491, May 2021.

[31] D. Jhunjhunwala, A. Gadhikar, G. Joshi, and Y. C. Eldar, "Adaptive quantization of model updates for communication-efficient federated learning," in *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3110–3114, 2021.

[32] P. Micikevicius, D. Stosic, N. Burgess, M. Cornea, P. Dubey, R. Grisenthwaite, S. Ha, A. Heinecke, P. Judd, J. Kamalu, N. Mellempudi, S. Oberman, M. Shoeybi, M. Siu, and H. Wu, "Fp8 formats for deep learning," 2022.

[33] D. Yin, A. Pananjady, M. Lam, D. S. Papailiopoulos, K. Ramchandran, and P. L. Bartlett, "Gradient diversity empowers distributed learning," *CoRR*, vol. abs/1706.05699, 2017.

[34] W. Luo, Y. Li, R. Urtasun, and R. Zemel, "Understanding the effective receptive field in deep convolutional neural networks," 2017.

[35] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *CoRR*, vol. abs/1608.08710, 2016.

[36] S.-S. Park and K. Chung, "Cenna: Cost-effective neural network accelerator," *Electronics*, vol. 9, p. 134, 01 2020.

[37] D. A. Vink and C.-S. Bouganis, "Fpga-enabled pytroch training (fpgpt): Acceleration of dnn training through a custom memory system," 2023.

[38] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," *Insights Imaging*, vol. 9, pp. 611–629, Aug. 2018.

[39] A. F. Agarap, "Deep learning using rectified linear units (relu)," 2019.

[40] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

[41] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 06 2014.

[42] S. I. Venieris and C. Bouganis, "fpgaconvnet: A framework for mapping convolutional neural networks on fpgas," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 40–47, 2016.

[43] A. Montgomerie-Corcoran, Z. Yu, and C.-S. Bouganis, "Samo: Optimised mapping of convolutional neural networks to streaming architectures," 2022.

[44] P. Toupas, A. Montgomerie-Corcoran, C.-S. Bouganis, and D. Tzovaras, "Harflow3d: A latency-oriented 3d-cnn accelerator toolflow for har on fpga devices," 2023.

[45] A. Zlateski, Z. Jia, K. Li, and F. Durand, "Fft convolutions are faster than winograd on modern cpus, here is why," 2018.

[46] Y. Nahshan, B. Chmiel, C. Baskin, E. Zheltonozhskii, R. Banner, A. M. Bronstein, and A. Mendelson, "Loss aware post-training quantization," 2020.

[47] K. Nakata, D. Miyashita, J. Deguchi, and R. Fujimoto, "Adaptive quantization method for cnn with computational-complexity-aware regularization," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2021.

[48] Z. Yuan, C. Xue, Y. Chen, Q. Wu, and G. Sun, "Ptq4vit: Post-training quantization framework for vision transformers with twin uniform quantization," 2022.

[49] M. Nagel, R. A. Amjad, M. van Baalen, C. Louizos, and T. Blankevoort, "Up or down? adaptive rounding for post-training quantization," 2020.

[50] R. Zhao, Y. Hu, J. Dotzel, C. D. Sa, and Z. Zhang, "Improving neural network quantization without retraining using outlier channel splitting," 2019.

[51] R. Banner, Y. Nahshan, E. Hoffer, and D. Soudry, "Aciq: Analytical clipping for integer quantization of neural networks," *ArXiv*, vol. abs/1810.05723, 2018.

[52] C. Sakr, S. Dai, R. Venkatesan, B. Zimmer, W. J. Dally, and B. Khailany, "Optimal clipping and magnitude-aware differentiation for improved quantization-aware training," 2022.

[53] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A study of bfloat16 for deep learning training," 2019.

[54] S. Fox, S. Rasoulinezhad, J. Faraone, david boland, and P. Leong, "A block minifloat representation for training deep neural networks," in *International Conference on Learning Representations*, 2021.

[55] J. Lee, D. Kim, and B. Ham, "Network quantization with element-wise gradient scaling," 2021.

[56] C. Liu, X. Zhang, R. Zhang, L. Li, S. Zhou, D. Huang, Z. Li, Z. Du, S. Liu, and T. Chen, "Rethinking the importance of quantization bias, toward full low-bit training," *IEEE Transactions on Image Processing*, vol. 31, pp. 7006–7019, 2022.

[57] B. Chmiel, L. Ben-Uri, M. Shkolnik, E. Hoffer, R. Banner, and D. Soudry, "Neural gradients are near-lognormal: improved quantized and sparse training," 2020.

[58] R. Gong, X. Liu, S. Jiang, T. Li, P. Hu, J. Lin, F. Yu, and J. Yan, "Differentiable soft quantization: Bridging full-precision and low-bit neural networks," 2019.

[59] J. de Fine Licht, G. Kwasniewski, and T. Hoefler, “Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis,” in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, (Seaside CA USA), pp. 244–254, ACM, Feb. 2020.

[60] S. Perri, M. Lanuzza, P. Corsonello, and G. Cocorullo, “A high-performance fully reconfigurable FPGA-based 2D convolution processor,” *Microprocessors and Microsystems*, vol. 29, pp. 381–391, Nov. 2005.

[61] K. Mohammad and S. Agaian, “Efficient FPGA implementation of convolution,” in *2009 IEEE International Conference on Systems, Man and Cybernetics*, (San Antonio, TX, USA), pp. 3478–3483, IEEE, Oct. 2009.

[62] Wenlai Zhao, Haohuan Fu, W. Luk, Teng Yu, Shaojun Wang, Bo Feng, Yuchun Ma, and Guangwen Yang, “F-CNN: An FPGA-based framework for training Convolutional Neural Networks,” in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, (London), pp. 107–114, IEEE, July 2016.

[63] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, “Energy-efficient cnn implementation on a deeply pipelined fpga cluster,” pp. 326–331, Institute of Electrical and Electronics Engineers Inc., 8 2016.

[64] D. A. Vink, A. Rajagopal, S. I. Venieris, and C.-S. Bouganis, “Caffe Barista: Brewing Caffe with FPGAs in the Training Loop,” *arXiv:2006.13829 [cs]*, June 2020. arXiv: 2006.13829.

[65] S. Fox, J. Faraone, D. Boland, K. Vissers, and P. H. W. Leong, “Training Deep Neural Networks in Low-Precision with High Accuracy Using FPGAs,” in *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 1–9, Dec. 2019.

[66] Z. Liu, Y. Dou, J. Jiang, Q. Wang, and P. Chow, “An FPGA-based processor for training convolutional neural networks,” in *2017 International Conference on Field Programmable Technology (ICFPT)*, pp. 207–210, Dec. 2017.

[67] S. K. Venkataramanaiah, Y. Ma, S. Yin, E. Nurvithadhi, A. Dasu, Y. Cao, and J.-s. Seo, “Automatic Compiler Based FPGA Accelerator for CNN Training,” *arXiv:1908.06724 [cs, eess]*, Aug. 2019. arXiv: 1908.06724.

[68] S. Fox, J. Faraone, D. Boland, K. Vissers, and P. H. Leong, “Training deep neural networks in low-precision with high accuracy using fpgas,” in *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 1–9, 2019.

[69] C. Luo, M.-K. Sit, H. Fan, S. Liu, W. Luk, and C. Guo, “Towards efficient deep neural network training by fpga-based batch-level parallelism,” Feb 2020.

[70] M. Wang, S. Rasoulinezhad, P. H. W. Leong, and H. K.-H. So, “NITI: Training integer neural networks using integer-only arithmetic,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, pp. 3249–3261, nov 2022.

[71] D. J. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, “A customizable matrix multiplication framework for the intel harpv2 xeon+fpga platform: A deep learning case study,” in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA ’18, (New York, NY, USA), p. 107–116, Association for Computing Machinery, 2018.

[72] S. Wu, G. Li, F. Chen, and L. Shi, “Training and inference with integers in deep neural networks,” 2018.

[73] Y. Tang, X. Zhang, P. Zhou, and J. Hu, “Ef-train: Enable efficient on-device cnn training on fpga through data reshaping for online adaptation or personalization,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 27, jun 2022.

[74] K. He, B. Liu, Y. Zhang, A. Ling, and D. Gu, “Fecaffe: Fpga-enabled caffe with opencl for deep learning training and inference on intel stratix 10,” *CoRR*, vol. abs/1911.08905, 2019.

[75] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison,

A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[76] AISmartz, "Cnn architectures over a timeline (1998-2019)," Oct 2019.

[77] P. Xiyuan, Y. Jinxiang, Y. Bowen, L. Liansheng, and P. Yu, "A review of fpga-based custom computing architecture for convolutional neural network inference," *Chinese Journal of Electronics*, vol. 30, no. 1, pp. 1–17, 2021.

[78] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[dl] a survey of fpga-based neural network inference accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, Mar. 2019.

[79] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo, "Performance modeling for cnn inference accelerators on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 843–856, 2020.

[80] A. Ghaffari and Y. Savaria, "Cnn2gate: An implementation of convolutional neural networks inference on fpgas with automated design space exploration," *Electronics*, vol. 9, p. 2200, 12 2020.

[81] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating cnn inference on fpgas: A survey," 2018.

[82] A. Rajagopal and C.-S. Bouganis, "perf4sight: A toolflow to model cnn training performance on edge gpus," 2021.

[83] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "Imagenet training in minutes," 2018.

[84] NVIDIA, "Nvidia deep learning cudnn documentation," 2023.

[85] A. Stoutchinin, F. Conti, and L. Benini, "Optimally scheduling cnn convolutions for efficient memory access," 2019.

[86] T. Jin and S. Hong, "Split-cnn: Splitting window-based operations in convolutional neural networks for memory system optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), p. 835–847, Association for Computing Machinery, 2019.

[87] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "Optimizing memory efficiency for deep convolutional neural networks on gpus," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 633–644, 2016.

[88] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[89] A. Kouris, S. I. Venieris, and C. Bouganis, "CascadeCNN: Pushing the Performance Limits of Quantisation in Convolutional Neural Networks," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 155–1557, 2018.

[90] A. Migdalas, P. M. Pardalos, and P. Värbrand, *Multilevel Optimization: Algorithms and Applications*, vol. 20. Springer Science & Business Media, 2013.

[91] P. with Code, "Papers with code - an overview of learning rate schedules," 2022.

[92] A. Kerr, D. Merril, J. Demouth, and J. Tran, "CUTLASS: Fast Linear Algebra in CUDA C," Sep 2018.

[93] C. Coleman, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia, "Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark," *SIGOPS Oper. Syst. Rev.*, vol. 53, no. 1, pp. 14–25, 2019.

[94] P. Maji and R. Mullins, "On the reduction of computational complexity of deep convolutional neural networks," *Entropy*, vol. 20, p. 305, 04 2018.

[95] C. Liu, Y. Wu, Y. Lin, and S. Chien, "A kernel redundancy removing policy for convolutional neural network," *CoRR*, vol. abs/1705.10748, 2017.

[96] Amazon, "Amazon Inferentia ML Chip." `https://aws.amazon.com/machine-learning/inferentia/`, 2020. [Retrieved: May 23, 2024].

[97] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Patel, and M. Herbordt, "A Framework for Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters with Work and Weight Load Balancing," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 394–3944, 2018.

[98] H. Zeng and V. Prasanna, "GraphACT: Accelerating GCN Training on CPU-FPGA Heterogeneous Platforms," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, p. 255–265, 2020.

[99] Amazon, "F1 instance in Amazon AWS." `https://aws.amazon.com/ec2/instance-types/f1/`. [Retrieved: May 23, 2024].

[100] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia (MM)*, MM '14, (New York, NY, USA), p. 675–678, Association for Computing Machinery, 2014.

[101] Xilinx, "Optimization techniques in vitis hls," Mar 2021.

[102] A. Rajagopal, D. A. Vink, S. I. Venieris, and C.-S. Bouganis, "Multi-precision policy enforced training (muppet): A precision-switching strategy for quantised fixed-point training of cnns," 2020.

[103] D. Das, N. Mellempudi, D. Mudigere, D. Kalamkar, S. Avancha, K. Banerjee, S. Sridharan, K. Vaidyanathan, B. Kaul, E. Georganas, A. Heinecke, P. Dubey, J. Corbal, N. Shustrov, R. Dubtsov, E. Fomenko, and V. Pirogov, "Mixed precision training of convolutional neural networks using integer operations," 2018.

[104] X. Sun, N. Wang, C.-Y. Chen, J. Ni, A. Agrawal, X. Cui, S. Venkataramani, K. El Maghraoui, V. V. Srinivasan, and K. Gopalakrishnan, "Ultra-low precision 4-bit training of deep neural networks," in *Advances in Neural Information Processing Systems*

(H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), vol. 33, pp. 1796–1807, Curran Associates, Inc., 2020.

[105] AMD, *Versal ACAP: Technical Refernce Manual.* AMD, 1.5 ed., December 2022.

Appendix A

Performance Model

A.1 INP Components

This section describes each component of the INP components shown in Fig. A.1 with a description of their operation and how that leads to constructing the detailed performance model.

A.1.1 INP DDR Reading

The INP DDR reading component is responsible for reading the INP data in the pattern presented in Fig. 5.4a. The INP DDR Precompute component starts alongside the DDR Reading component as the kernel initiates. The precompute component calculates the size of the channel jump required between the last data element of square 5 and the same colored square 1 of the next channel. By knowing the size of the channel jump the DDR Reading component can burst read in all of the data without reading delays. Partially sequential bust-reading requires that the amount of data read in each sequential iteration is known at compilation. Hence $Z_{channel,\alpha}$ is used as the size of the sequential burst-read. The precompute computation takes less time than the burst-read fully amortizing the computation cost.

If $Z_{channel,\alpha} > Z_{channel,\beta}$, excess data is read in and immediately discarded. Discarding excess

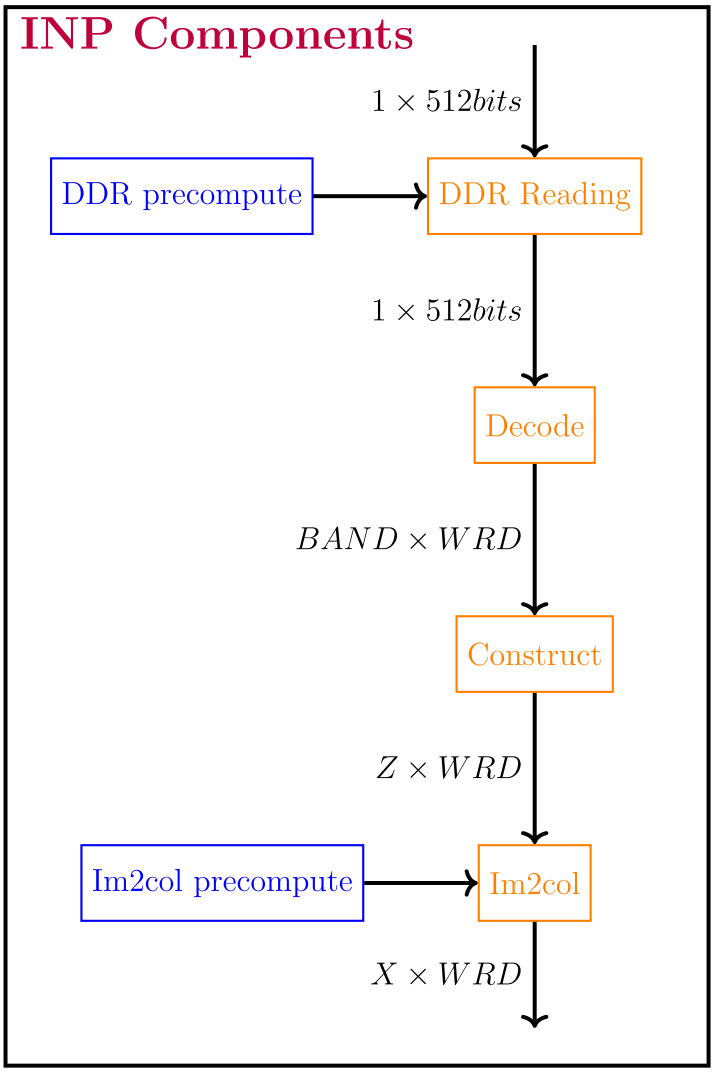


Figure A.1: INP functional components

data allows all the subsequent functional components to run entirely based on layer parameters. Operating exclusively on layer parameters is one of the features that enables this design to adapt its runtime based on the convolution workload..

The cycles taken to read in $Z_{channel,\alpha}$ values is:

$$\Gamma_{Burst}(\alpha) = \text{align}(BurstSize_{Inner}(\alpha), 16) \cdot BurstSize_{Outer}(\alpha) \tag{A.1}$$

where:

$$BurstSize_{Inner}(\alpha) = \min\left(\left\lceil \frac{Z_{channel,\alpha}}{BAND} \right\rceil, 256\right) \tag{A.2}$$

$$BurstSize_{Outer}(\alpha) = \max\left(\left\lceil \frac{BurstSize_{Inner}(\alpha)}{256} \right\rceil, 3\right) \tag{A.3}$$

The align function refers to aligning a value to ensure ϕ is divisible by θ through: align $(\phi, \theta) = \left\lceil \frac{\phi}{\theta} \right\rceil \cdot \theta$. To enable a partially sequential burst read in Vitis HLS, at least two loops must be implemented. The inner loop defines the current burst read (up to 256 reads of 512 bits due to HLS limitations). The surrounding loops are then constructed to reselect where to run the compute from. As a result, the read of each set of $Z_{channel}$ values was split up into an inner loop up to 256 in value and an outer loop. Due to HLS limitations, the size of the outer loop must be a minimum of 3. Furthermore, the total number of reads must be a multiple of 16, as the burst read seems to operate in blocks of 512×16. Consequently reading in 14 blocks takes as many cycles as reading in 16. This is the reason that $\Gamma_{Burst}(\alpha)$ aligns $BurstSize_{Inner}$ with 16.

Once a single $Z_{channel,\alpha}$ has been read in, the channel jump occurs.

$$\Gamma_{\text{Channel Jump}}(\alpha, \beta) = \Gamma_{Burst}(\alpha) \cdot C_{INP} \tag{A.4}$$

After channel jumping for one set of X KERN locations, X Set jumping occurs:

$$\Gamma_{X \text{ Set Jump}}(\alpha, \beta) = \Gamma_{\text{Channel Jump}}(\alpha, \beta) \cdot \left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil \tag{A.5}$$

After $\Gamma_{X \text{ Set Jump}}$ cycles, Y 2D OUT tiles have been computed. Following the same logic as in Sec. 5.3.3, to produce the entire 4D OUT tensor, this is repeated $\lceil \frac{C_{OUT}}{Y} \rceil \cdot N_{INP} \cdot A$ times. This will be referred to as Batch Iteration for the rest of the paper. This leads to a final runtime of:

$$\Gamma_{INP-DDR\ Read}(\alpha, \beta) = \Gamma_{Burst}(\alpha) \cdot \underbrace{C_{INP}}_{\text{Channel Jump}} \cdot \underbrace{\left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{X \text{ Set Jump}} \cdot \underbrace{\left\lceil \frac{N_{KERN}}{Y} \right\rceil}_{\text{Completing } N_{KERN}} \cdot \underbrace{N_{INP} \cdot A}_{\text{Batch Iteration}} \tag{A.6}$$

A.1.2 INP Decode

The decode component receives 512bit values from the INP DDR reading component. This 512bit burst read data is split into $BAND$ FIFO streams. After the split, the data from the streams is passed on to the INP Construction component. The decode stage of INP has a very similar iteration space that of the DDR read. The key difference is that the decode will only operate on valid data, i.e. $Z_{channel,\beta}$ not $Z_{channel,\alpha}$. This is possible due to the fact that the DDR component filters out the unnecessary data and never sends it down (as discussed in Sec. 5.3.6). As a result, it can replace $\Gamma_{INPDDR\ Outer}(\alpha)$ with:

$$\Gamma_{align}(\beta) = \text{align}(Z_{channel,\beta}, BAND) \tag{A.7}$$

The exact same logic as with the DDR reading iteration space is followed with regards to Channel Jumping, X Set Jumping, Completing N_{KERN} and Batch Iteration, resulting in:

$$\Gamma_{INP-Decode}(\beta) = \underbrace{\text{align}(Z_{channel,\beta}, 16)}_{\text{alignment}} \cdot \underbrace{C_{INP}}_{\text{Channel Jump}} \cdot \underbrace{\left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{X \text{ Set Jump}} \cdot \underbrace{\left\lceil \frac{N_{KERN}}{Y} \right\rceil}_{\text{Completing } N_{KERN}} \cdot \underbrace{N_{INP} \cdot A}_{\text{Batch Iteration}} \tag{A.8}$$

It can be deduced that:

$$\begin{gathered} Z_{channel,\beta} \leq Z_{channel,\gamma} \\ \therefore \Gamma_{align}(\alpha) \leq \Gamma_{Outer}(\alpha) \\ \therefore \Gamma_{INP-Decode}(\alpha,\beta) \leq \Gamma_{INP-DDR\ Read}(\alpha,\beta) \end{gathered} \tag{A.9}$$

ensuring that the decode section is never the bottleneck for the INP memory subsystem.

A.1.3 INP Construct

The INP Construct component receives $BAND$ words from the INP Decode component and constructs groups of Z words. Depending on whether or not $Z < BAND$ there are two different construction components (only one will be compiled due to Z and $BAND$ being compile time parameters).

The differentiating component between the two designs is expressed in the amount of cycles it takes to turn $Z_{channel,\beta}$ values into Z values expressed by:

$$\Gamma_{grouping_1}(\beta) = \begin{cases} \left\lceil \frac{Z_{channel,\beta}}{BAND} \right\rceil \cdot BAND & \text{if } Z < BAND \\ \left\lceil \frac{Z_{channel,\beta}}{BAND} \right\rceil \cdot Z & \text{if } Z \geq BAND \end{cases} \tag{A.10}$$

$$\Gamma_{grouping}(\beta) = \begin{cases} \left\lceil \frac{\Gamma_{grouping_1}(\beta)}{Z} \right\rceil & \text{if } Z < BAND \\ \left\lceil \frac{\Gamma_{grouping_1}(\beta)}{BAND} \right\rceil & \text{if } Z \geq BAND \end{cases} \tag{A.11}$$

This is then repeated for the same iteration space as in the INP Decode and INP DDR components resulting in:

$$\Gamma_{INP-Constr}(\beta) = \underbrace{\Gamma_{grouping}(\beta)}_{\text{Grouping}} \cdot \underbrace{C_{INP}}_{\text{Channel Jump}} \cdot \underbrace{\left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{X \text{ Set Jump}} \cdot \underbrace{\left\lceil \frac{N_{KERN}}{Y} \right\rceil}_{\text{Completing } N_{KERN}} \cdot \underbrace{N_{INP} \cdot A}_{\text{Batch Iteration}} \tag{A.12}$$

The Z values constructed per cycle are passed from the construction component to the final *im2col* component. When compiling a kernel for multiple layers, the Z value that is used is the maximum across all target convolutions.

A.1.4 INP im2col

The INP im2col component reads the incoming Z, WRD wide FIFOs from the Construction component. This unit contains the line-buffer mentioned in Sec. 5.3.6. The incoming Z values are stored in a line-buffer memory until all $Z_{channel,\beta}$ values have been provided. Once the entire channel has been provided to the SA the line-buffer is cleared and filled with new oncoming data. Due to the nature of Z, even after the clear there will always be enough data to be able to continue uninterrupted, maintaining and II=1.

At each cycle, the correct data has to be provided to the X FIFOs following Fig. 5.4a and Fig. 5.4b. The line-buffer data stats at the first value of square 1 from Fig. 5.4a and need to provide all the red values form Fig. 5.4b. For each cycle, the hardware computes where each of the required values is relative to the first value in the line buffer. The INP im2col precompute component is instantiated to compute this data.

If the precompute component indicates the value is in a dilated spot (like in Eq. 2.3) or in the padded area of the INP, a zero is passed to the FIFO rather than accessing on-chip memory. This reduces the memory footprint by not storing unnecessary zeros while not penalizing performance. The on-chip memory access precomputation theoretically does not need to occur in a separate component but limitations imposed by the implementation method led to separating the two processes.

Due to the fact that the construction component ensures all data provided to the im2col component is Z grouped, whether $Z < BAND$ is of no concern for this component. The SA assumes data will be available every cycle as every preceding component has $II = 1$. The scenario where this is not the case will be addressed when computing the Γ_{INP}

As the im2col component directly supplies the SA, but is not limited by the 'shifting delay', its performance model is a simpler version of $\Gamma_{SysArray}(\alpha, \beta)$ where $M = C_{KERN} \cdot H_{KERN,dilated} \cdot$

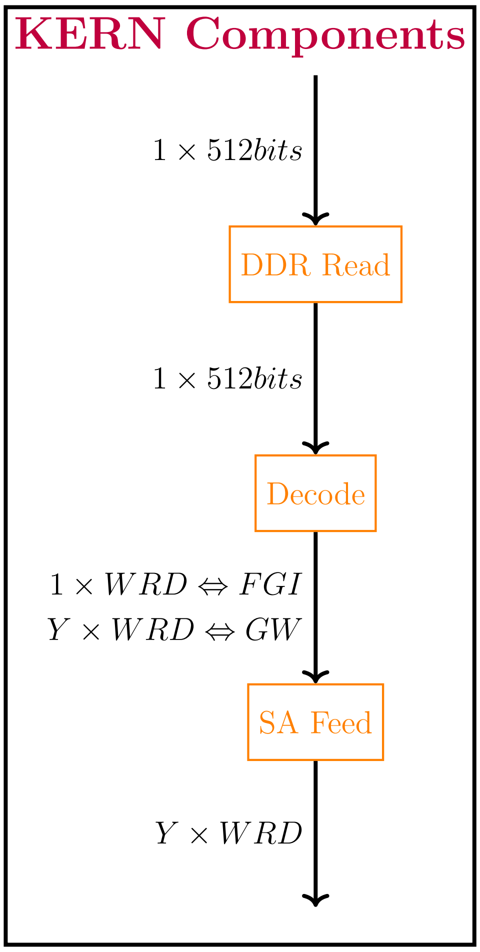


Figure A.2: KERN functional components

$W_{KERN,dilated}$.

$$\Gamma_{INP-I2C}(\alpha,\beta) = M \cdot \underbrace{\left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{X \text{ Set Jump}} \cdot \underbrace{\left\lceil \frac{N_{KERN}}{Y} \right\rceil}_{\text{Completing } N_{KERN}} \cdot \underbrace{N_{INP} \cdot A}_{\text{Batch Iteration}} \tag{A.13}$$

A.2 KERN Components

This section describes each component of the KERN components shown in Fig. A.2 with a description of their operation and how that leads to constructing the detailed performance model.

A.2.1 Forward and $Grad_{IFM}$: KERN Overview

Both forward and $Grad_{IFM}$ convolutions have a significantly smaller KERN size in comparison to either their INP or OUT. This allows for the entire 4D KERN tensor to be stored on on-chip memory. This has been confirmed across a variety of modern ImageNet networks, requiring between circa 20-220 BRAM blocks for GoogLeNet and SqueezeNet. Each of these values are required N_{INP} (Batch Iteration) number of times by the SA. Consequently that memory is maintained throughout the convolution to avoid repeated reading of the DDR. A_{KERN} is not discussed throughout the discussion of FGI convolutions as it is always 1 and will therefore have no impact on the performance model.

A.2.2 Forward and $Grad_{IFM}$: KERN DDR Reading

As the entire set of weights for a layer are stored on-chip, they are read in a single continuous burst read. This takes a total of:

$$\Gamma_{KERN-DDRRead}(\beta) = \left\lceil \frac{N_{KERN} \cdot C_{KERN} \cdot H_{KERN} \cdot W_{KERN}}{BAND} \right\rceil \tag{A.14}$$

cycles. Because this is a completely sequential burst reading, there is no need for compile time parameters to control the size of the inner loops.

A.2.3 Forward and $Grad_{IFM}$: KERN Decode

The data from the DDR Reading component is passed to the decode component. This component combines the functionality of the INP Decode and Construction component. It turns the 512bit value consisting of $BAND$ words into a stream of 1 word per cycle. Because the data arrives in a sequential fashion there is no explicit reason for a line buffer. Such sequential data simplifies the design and enables merging the decode and construction component. Due to the fact that the weights for the FGI scenario are always significantly smaller than the INP data, sequentially reading the data will not bottleneck the overall system. The cycles required to pass the KERN SA Feeder (next component) one value per cycle is:

$$\Gamma_{KERN-Decode}(\beta) = N_{KERN} \cdot C_{KERN} \cdot H_{KERN} \cdot W_{KERN} \tag{A.15}$$

Also it should be noted that:

$$\Gamma_{KERN-Decode}(\beta) > \Gamma_{KERN-DDR}(\beta) \tag{A.16}$$

in all cases as $BAND \geq 1$

A.2.4 Forward and $Grad_{IFM}$: KERN SA Feeder

The data from the decode component are passed to the systolic array feeder component. This component comprises of 2 parts: 1) on-chip memory writer, 2) on-chip memory reader and systolic array feeder. As the entirety of the 4D KERN tensor will be stored on-chip, the memory will be of size $N_{KERN} \times C_{KERN} \times H_{KERN} \times W_{KERN}$ words. The SA does require Y values per cycle, each from a different 3D KERN tensor. To provide this parallelism, the data is stored as Y rows of size $\lceil \frac{N_{KERN}}{Y} \rceil \times C_{KERN} \cdot H_{KERN} \cdot W_{KERN}$ words. The data stored in memory does not include the zeros generated due to dilation and the *shifting delay* in order to minimize the design's memory footprint. These elements only introduce zeros and can be computed during runtime. Not storing these zeros saves:

$$(M - C_{KERN} \cdot H_{KERN} \cdot W_{KERN}) \cdot N_{KERN} \tag{A.17}$$

words.

The $rot180$ referenced in $Grad_{IFM} = dilate\left(\frac{\delta loss}{\delta OFM}\right) * perm\left(rot180\left(weight\right)\right)$ (see Eq. 5.3) occurs by changing the order in which the on-chip memory array is written to. The 180 degree rotation for the $Grad_{IFM}$ scenario is performed on-the-fly. As data comes in one value at a time, there is no need for parallel memory access which allows for a simple hardware implementation.

Based on the rate of data arriving, the array fill happens at pace identical to $\Gamma_{KERN-Decode}(\beta)$:

$$\Gamma_{KERN-Feed_{Phase1}}(\beta) = N_{KERN} \cdot C_{KERN} \cdot H_{KERN} \cdot W_{KERN} = \Gamma_{KERN-Decode}(\beta) \tag{A.18}$$

This sequential read would be very inefficient and a bottleneck if the size of KERN in the Forward and $Grad_{IFM}$ scenario was not as small as it is. All FGI convolutions across all networks presented in this paper were analyzed and the KERN SA Feeder component is never a bottleneck. The potential bottleneck along with the memory issues of storing the entirety of the KERN data are why GW has a separate KERN design.

Once the on-chip memory array has been written to, the second phase initiates, reading from each of the Y rows in parallel. For each write to the output FIFOs a check is performed to see if the current location in processing would be in a dilated space. Additionally the second phase has to adhere to the design of the systolic array execution including the *shifting delay.* If it is a dilated or *shifting delay* location a zero is passed to the FIFO, otherwise on-chip memory is accessed. This component provides Y values per cycle, providing M as in Eq. A.13, accounting for the same X Set Jump to create Y 2D OUT matrices. After Y 2D OUT matrices worth of KERN data have been fed to the SA and before the next set of Y 2D OUT tiles can be addressed, the shifting delay and final accumulation are processed:

$$\begin{aligned}&\Gamma_{KERN-Feed_{Y\text{ 2D OUT matrices}}}(\alpha,\beta) = \\ &\underbrace{C_{KERN} \cdot H_{KERN,dilated} \cdot W_{KERN,dilated}}_{M} \cdot \underbrace{\left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{X\text{ Set Jump}} + \underbrace{Y+8}_{\text{shift delay + acc}}\end{aligned} \tag{A.19}$$

Then following the same reasoning as in Sec. 5.3.3 in Eq. 5.8 complete execution accounts for the batch iteration, the overall execution for phase 2 is:

$$\begin{aligned}&\Gamma_{KERN-Feed_{Phase2}}(\alpha,\beta) = \\ &\left(\underbrace{M \cdot \left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{Y\text{ 2D OUT matrices}} + \underbrace{Y+8}_{\text{shift delay + acc}}\right) \cdot \underbrace{\left\lceil \frac{N_{KERN}}{Y} \right\rceil}_{\text{Completing } N_{KERN}} \cdot \underbrace{N_{INP}}_{\text{Completing } N_{INP}} \cdot \underbrace{A}_{\text{Completing } A}\end{aligned} \tag{A.20}$$

where $M = C_{KERN} \cdot H_{KERN,dilated} \cdot W_{KERN,dilated}$ and $A = 1$. The $N_{INP} \cdot A$ is the same in Sec. 5.3.3 and as the *Batch iteration* in Sec. 5.3.6. Please note that C_{OUT} and N_{KERN} can be used interchangeably when discussing FGI as they are the same for FGI.

Finally, combining phase 1 and 2 together results in:

$$\Gamma_{KERN-Feed}(\alpha, \beta) = \Gamma_{KERN-Decode}(\beta) + \Gamma_{KERN-Feed_{Phase2}}(\alpha, \beta) \tag{A.21}$$

A.2.5 $Grad_{weight}$: KERN Overview

The KERN design for the $Grad_{weight}$ is more complex than for the Forward and $Grad_{IFM}$ (FGI) design. The reason for this is due to the fact that the KERN data is the size of the OFM tensor (see Tab. 5.2 and Tab. 5.3). As a result, the entirety of the KERN data can no longer be stored in the KERN on-chip memory. This makes both the FGI memory requirement and the FGI KERN Decode component's single value passing design unacceptable. To resolve this a dynamic loading design is created with a partially-sequential burst read implementation from off-chip memory.

KERN data is reread from off-chip memory multiple times. The on-chip memory is a double-buffered memory of size $Y \times (C_{KERN} \cdot H_{KERN} \cdot W_{KERN})$. This memory size is large enough to store the KERN tensor data required to compute Y sets of 2D OUT matrices. For the KERN tensor data required to compute the next Y set of 2D OUT matrices, the memory is cleared and replaced. This is repeated *Completing* N_{KERN} times leading to the *Completing* N_{INP} stage (see Sec. 5.3.3). The *Completing* N_{INP} stage expects the KERN data in the exact same order each of the N_{INP} times. Furthermore, for each iteration of the *Completing* A stage, the next set of KERN data is reread N_{INP} times. The added complexity of repeated reading from off-chip memory is why the FGI KERN components store the entirety of the KERN tensor data on-chip, as it decreases off-chip memory reading requirement.

A.2.6 $Grad_{weight}$: KERN DDR Reading

In the INP memory subsystem, $Z_{channels,\alpha}$ values are read sequentially. In contrast, KERN reads $KRange_\beta = N_{KERN} \cdot C_{KERN} \cdot H_{KERN} \cdot W_{KERN}$ values sequentially. Just like with $Z_{channels}$, $KRange_\alpha \geq KRange_\beta$, and $KRange_\alpha \in \alpha$ and $KRange_\beta \in \beta$. Reading the data produces a execution cycle of:

$$\Gamma_{KERN-Burst}(\alpha) = \text{align}(BurstSize_{Inner}(\alpha), 16) \cdot BurstSize_{Outer}(\alpha) \tag{A.22}$$

where:

$$BurstSize_{Inner}(\alpha) = \min\left(\left\lceil \frac{KRange_\alpha}{BAND} \right\rceil, 256\right) \tag{A.23}$$

$$BurstSize_{Outer}(\alpha) = \max\left(\left\lceil \frac{BurstSize_{Inner}(\alpha)}{256} \right\rceil, 3\right) \tag{A.24}$$

After $KRange_\alpha$ values have been read in from off-chip memory, $KRange_\beta$ of the acquired data is passed on. Just like in INP and KERN FGI, the excess data read in due to $KRange_\alpha \geq KRange_\beta$ is not passed on to the subsequent Decode component. The starting point for next set of $KRange_\beta$ values will be computed by a precompute component. As discussed in Sec. A.2.5, the $KRange_\beta$ data is reread N_{INP} times, for each iteration of A:

$$\Gamma_{KERN-DDR\ Read}(\beta) = \underbrace{\text{align}(BurstSize_{Inner}(\alpha), 16) \cdot BurstSize_{Outer}(\alpha)}_{\text{Sequential burst}} \cdot \underbrace{N_{INP} \cdot A}_{\text{Batch iteration}} \tag{A.25}$$

A.2.7 $Grad_{weight}$: KERN Decode

As demonstrated by Fig. 5.6, data is read by the DDR reading component then passed through a decoding component before being handed over the SA Feeder component. As this is a construction component, the inputs to the component are $BAND$ aligned and the output is Y aligned. As a result, there are two diverging implementations for $Y < BAND$ and $Y \geq BAND$. In the scenario where $Y < BAND$, a single input value is split across at least 2 output values while in the scenario where $Y \geq BAND$ more than one input value will be required to produce an output value. As both Y and $BAND$ are compile time parameters, only one of the two paths will be compiled. The amount of cycles required to process $KRange_\beta$ elements:

$$\Gamma_{KERN-Decode_1} = \begin{cases} \left\lceil \frac{KRange_\beta}{Y} \right\rceil & \text{if } Y < BAND \\ \left\lceil \frac{KRange_\beta}{BAND} \right\rceil & \text{if } Y \geq BAND \end{cases} \tag{A.26}$$

As discussed in the introduction for Sec. A.2.6, $KRange_\beta$ is read $N_{INP} \cdot A$ number of times, leading to the overall result of:

$$\Gamma_{KERN-Decode} = \Gamma_{KERN-Decode_1} \cdot \underbrace{N_{INP} \cdot A}_{\text{Batch iteration}} \tag{A.27}$$

A.3 OUT Components

This section describes each component of the KERN components shown in Fig. A.3 with a description of their operation and how that leads to constructing the detailed performance model.

A.3.1 $Grad_{weight}$: KERN SA Feeder

The KERN SA feeder contains a double-buffered memory of $Y \times (C_{KERN} \cdot H_{KERN} \cdot W_{KERN})$ words each. After the first buffer is filled up, while the second is filling, the first is being used to write to the Y SA FIFOs in parallel. This double buffering will amortize any time related to filling the memory with incoming data.

Just like in Sec. A.2.4, the KERN component needs to feed the systolic array according to the pattern as demonstrated by Eq. A.19. The data arrives in chunks of Y values per cycle (unlike in FGI scenario where it arrives 1 value per cycle). To ensure all the data is read into on-chip memory,

$$\Gamma_{DataRead} = Y \cdot \left\lceil \frac{C_{KERN} \cdot H_{KERN} \cdot W_{KERN}}{Y} \right\rceil \tag{A.28}$$

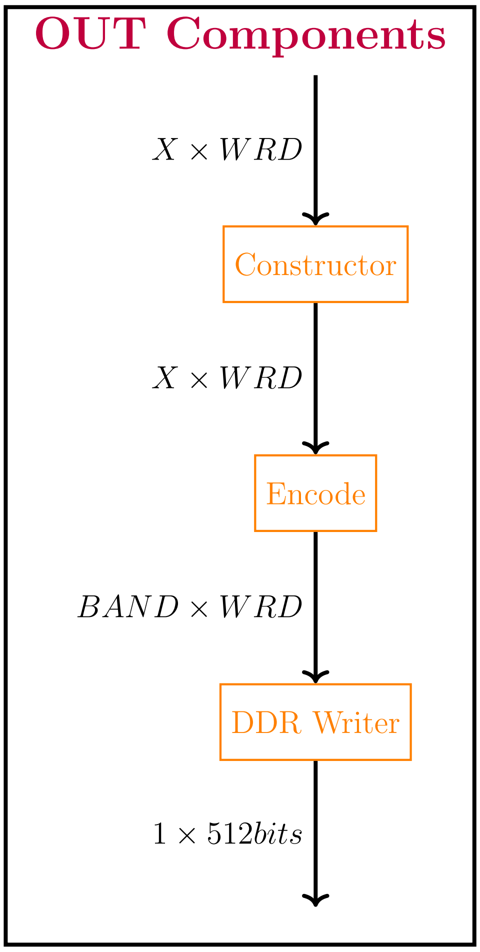


Figure A.3: OUT functional components

cycles are required. This provides the data for M cycles of SA execution. As it is a double-buffered system, filling on-chip memory happens concurrently to writing to the Y SA FIFOs. Writing to the Y SA FIFOs requires

$$\Gamma_{FIFOWrite} = Y \cdot \left\lceil \frac{C_{KERN} \cdot H_{KERN,dilated} \cdot W_{KERN,dilated}}{Y} \right\rceil \tag{A.29}$$

cycles. As $\Gamma_{FIFOWrite} \geq \Gamma_{DataRead}$ and new data is required for the X *Set jump* and *shifting delay*, just as in Sec. A.2.4:

$$\Gamma_{KERN-Feed_1}(\beta) = \underbrace{\left\lceil \frac{C_{KERN} \cdot H_{KERN,dilated} \cdot W_{KERN,dilated}}{Y} \right\rceil \cdot Y \cdot}_{\text{Data loading/ SA feeding}} \underbrace{\left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{X \text{ Set jump}} + \underbrace{Y + 8}_{\text{shift delay + acc}} \tag{A.30}$$

Applying the same reasoning as Sec. 5.3.3 and A.2.4, this results in

$$\Gamma_{KERN-Feed_2}(\alpha, \beta) = \Gamma_{KERN-Feed_1}(\alpha, \beta) \cdot \underbrace{\left\lceil \frac{N_{KERN}}{Y} \right\rceil}_{\text{Completing } N_{KERN}} \cdot \underbrace{N_{INP}}_{\text{Completing } N_{INP}} \cdot \underbrace{A}_{\text{Completing } A} \tag{A.31}$$

An interesting difference with the parallel equation for FGI ($\Gamma_{KERN-Feed_{Phase2}}$), is that although Completing N_{KERN} looks similar, in this scenario, $N_{KERN} = N_{OUT}$, not C_{OUT}.

Finally, before everything can start, the first of the two buffers needs to be filled requiring:

$$\Gamma_{KERN-Feed_{init}}(\alpha, \beta) = Y \cdot \left\lceil \frac{C_{KERN} \cdot H_{KERN} \cdot W_{KERN}}{Y} \right\rceil \tag{A.32}$$

cycles. Note that this does not rely on dilated H_{KERN} and W_{KERN} as the dilation occurs on the fly and storing extra zeros serves no purpose. The resultant performance for the $Grad_{weight}$ Systolic Array Feeder is:

$$\Gamma_{KERN-Feed}(\alpha, \beta) = \Gamma_{KERN-Feed_{init}}(\alpha, \beta) + \Gamma_{KERN-Feed_1}(\alpha, \beta) \tag{A.33}$$

A.3.2 OUT: $Grad_{Weight}$ Constructor

The $Grad_{weight}$ OUT constructor fills an on-chip memory and then accumulates within the on-chip memory. As stated the $Grad_{weight}$ convolution constructor hosts the entire 4D OUT tensor in on-chip memory for accumulation. It has been discussed that the KERN Systolic Array Feeder component for FGI in Sec. A.2.4 is able to store the entirety of the KERN data in on-chip memory. For the same reasons, the entirety of 4D OUT data can be stored in on-chip memory for the $Grad_{weight}$ convolution.

INP and KERN provide the SA with the data that will construct a full $D \times Ch \times H_{weight} \times W_{weight}$ tensor. Accumulation only occurs after a full $D \times Ch \times H_{weight} \times W_{weight}$ tensor has been stored in on-chip memory. The on-chip memory holds the entirety the 4D OUT tensor and once the accumulation starts the amount of memory is static. Once the accumulation completes, the on-chip memory data is passed on to the next OUT component to be grouped and then written to the off-chip memory. This means that the rest of the OUT components initiate after the systolic array is almost complete. Due to the output dimensions being very small in the $Grad_{weight}$ scenario, this initiation delay will have a negligible impact on the overall computation time. The justification for this is exactly the same as in the FGI KERN memory subsystem. The order in which data is passed to the subsequent component performs an on-the-fly permutation.

Once the SA has a valid result, after Y cycles of shifting delay, Y incomplete 2D OUT tiles are stored in memory.

$$\Gamma_{\text{2D OUT Matrix}} = \underbrace{Y}_{\text{shifting delay}} \cdot \underbrace{\left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{X \text{ set jump}} \tag{A.34}$$

Following the SA behavior from Sec. 5.3.3, after producing 2D OUT matrices, *Completing* N_{KERN} occurs which is then repeated for all N_{OUT}, filling in the remainder of the data:

$$\Gamma_{\text{pre-accumulation}} = \underbrace{Y \cdot \left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{\text{Y 2D OUT matrices}} \cdot \underbrace{\left\lceil \frac{C_{OUT}}{Y} \right\rceil}_{\text{Completing } N_{KERN}} \cdot \underbrace{N_{OUT}}_{\text{Complete } N_{OUT}} \tag{A.35}$$

Once this is complete, each element of the resultant 4D OUT tensor is written to the on-chip memory and the accumulation can start. Data is received in the same order as before, but is now accumulated rather than saved to on-chip memory has a cycle count of:

$$\Gamma_{\text{accumulation}} = \underbrace{Y \cdot \left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{\text{Y 2D OUT matrices}} \cdot \underbrace{\left\lceil \frac{C_{OUT}}{Y} \right\rceil}_{\text{Completing } N_{KERN}} \cdot \underbrace{N_{OUT}}_{\text{Complete } N_{OUT}} \underbrace{A}_{\text{accumulation}} \tag{A.36}$$

As discussed no data is passed on the Encode component until after the accumulation has been completed. The cycles taken for the data needs to be passed onto the Encode component a rate of X values per cycle as follows:

$$\Gamma_{write} = \left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil \cdot C_{OUT} \cdot N_{OUT} \tag{A.37}$$

The final time is the combination of the accumulation and the writing stages, resulting in:

$$\Gamma_{OUT_{Constr}} = \underbrace{Y \cdot \left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil \cdot \left\lceil \frac{N_{OUT}}{Y} \right\rceil \cdot C_{OUT} \cdot A}_{\text{read and accumulate}} + \underbrace{\left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil \cdot C_{OUT} \cdot N_{OUT}}_{\text{write out}} \tag{A.38}$$

A.3.3 OUT: Forward and $Grad_{Input}$ Constructor

The inherent design of the Constructor component for FGI has a similar iteration space to $Grad_{weight}$. The key differentiator is that it can pass data on to the next component as soon as data appears. In $Grad_{weight}$ the data arrives filling filters first (as shown in Fig. 5.8). As discussed previously, data arrives in filling Y consecutive tensors of the C_{OUT} dimension before moving on the next tensor in the N_{OUT} dimension. Therefore, no OUT permutation is required. Additionally no accumulation occurs removing the need for on-chip memory to be held long term. Due to these facts, the FGI design processes and then passes on data as soon it is provided by the SA. To ensure continuous execution, double buffering is employed to allow for simultaneous writing to the Encoder component while constructing the next OUT tensor. This double buffering requires $2 \cdot Y \cdot H_{OUT} \cdot W_{OUT} \cdot C_{OUT}$ words of on-chip memory. After constructing the full $C_{OUT} \times H_{OUT} \times W_{OUT}$ 3D OUT tensor, the component passes the tensor on to the encoder which combines the X tensor words into blocks of $BAND$ words.

The 2D OUT tiles arrive in the same way for the FGI and $Grad_{weight}$ designs and therefore a very similar iteration spaces:

$$\Gamma_{\text{2D OUT Matrix}} = \underbrace{Y}_{\text{shifting delay}} \cdot \underbrace{\left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{X \text{ set jump}} \tag{A.39}$$

Data is arriving as Y channels in succession, repeated for all C_{OUT} and then all N_{OUT}

$$\Gamma_{Main} = \underbrace{Y \cdot \left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{\text{Y 2D OUT matrices}} \cdot \underbrace{\left\lceil \frac{N_{KERN}}{Y} \right\rceil}_{\text{Completing } N_{KERN}} \cdot \underbrace{N_{INP}}_{\text{Completing } N_{INP}} \tag{A.40}$$

Being a double buffered design, the on-chip memory filling is one 2D OUT tile ahead of the write out, requiring an additional $\Gamma_{\text{2D OUT Matrix}}$ cycles resulting in an overall cycle count of:

$$\Gamma_{OUT_{Constructor}} = \underbrace{Y \cdot \left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil}_{\text{Initial fill}} + \underbrace{Y \cdot \left\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \right\rceil \cdot \left\lceil \frac{N_{KERN}}{Y} \right\rceil \cdot N_{INP}}_{\text{Main}} \tag{A.41}$$

A.3.4 OUT Encode

After the data is passed down, in order, from the OUT Constructor component, to the encode component. This component converts the incoming data from X grouped to $BAND$ grouped so that it can be burst written by the final component. Excess due to $\lceil \frac{H_{OUT} \cdot W_{OUT}}{X} \rceil \neq \frac{H_{OUT} \cdot W_{OUT}}{X}$ and $\lceil \frac{N_{OUT}}{Y} \rceil \neq \frac{N_{OUT}}{Y}$ is removed by the Constructor component. Everything arrives in groups of X with excess occurring if $\lceil \frac{C_{OUT} \cdot H_{OUT} \cdot W_{OUT}}{X} \rceil \neq \frac{C_{OUT} \cdot H_{OUT} \cdot W_{OUT}}{X}$ which is removed in this component. Any excess data that is not eliminated in the previous step is removed so that only relevant data is sent down. The data is sent to the next component in order to allow for sequential burst writing in the last component.

Both X and $BAND$ are compile time parameters spawning two separate iteration spaces depending on whether $X < BAND$ or not.

$$\Gamma_{OUT\ Encode_1} = \begin{cases} \left\lceil \frac{C_{OUT} \cdot H_{OUT} \cdot W_{OUT}}{X} \right\rceil & \text{if } X < BAND \\ \left\lceil \frac{C_{OUT} \cdot H_{OUT} \cdot W_{OUT}}{BAND} \right\rceil & \text{if } X \geq BAND \end{cases} \tag{A.42}$$

After $C_{OUT} \cdot H_{OUT} \cdot W_{OUT}$ is collected and grouped into $BAND$ components, this is then repeated N_{OUT} times with a final cycle model of:

$$\Gamma_{OUT\ Encode} = \Gamma_{OUT\ Encode_1} \cdot N_{OUT} \tag{A.43}$$

A.3.5 OUT DDR Writing

This is the simplest of the all the components. As all sorting, permuting, ordering and excess data removal has been done, this component gets the already $BAND$ grouped data and burst writes it to the off-chip memory:

$$\Gamma_{OUT_{DDR}} = \left\lceil \frac{N_{OUT} \cdot C_{OUT} \cdot H_{OUT} \cdot W_{OUT}}{BAND} \right\rceil \tag{A.44}$$