Thèse n° 10 265

ColTraIn: Co-located DNN training and inference

Présentée le 25 septembre 2020

à la Faculté informatique et communications
Laboratoire d'architecture de systèmes parallèles
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Mario Paulo DRUMOND LAGES DE OLIVEIRA

Acceptée sur proposition du jury

Prof. C. Koch, président du jury
Prof. B. Falsafi, Prof. M. Jaggi, directeurs de thèse
Prof. A. Moshovos, rapporteur
Dr M. Papamichael, rapporteur
Prof. J. Larus, rapporteur

École
polytechnique
fédérale
de Lausanne

2020

Acknowledgements

Doing a PhD was the hardest thing I have ever done in my entire life, by far. I would never have made it without extensive support from my mentors, who taught me everything I needed to complete this journey and the people who provided me with the moral support I needed to keep going.

First, I am grateful to my advisors, Babak Falsafi and Martin Jaggi. Babak's presence is legendary. I still remember my first meeting with him, how impressed I was with his self-confidence and his direct communication. I knew within ten seconds of talking to him that I needed someone like him to realize any potential I had. I was right. Babak taught me everything: to write, to speak, to think, to present my work, and to interact with the community. He did not stop there. He also taught me to appreciate food, coffee, beer, and even skiing. At some points, when my willpower faltered, Babak pushed across many finish lines. The training that Babak offers changes every aspect of his students' lives, and I am grateful that I received his mentorship.

Martin entered my life later during my PhD, in my third year. Martin's steady and calming presence provided a good counterbalance to Babak's aggressiveness. Whenever I needed reassurance, Martin was the right person to look for. Martin also guided me through the world of machine learning, a world I knew nothing about before starting my PhD. With his guidance, I was able to identify good work in a somewhat foreign field. I am very grateful that Martin took time from his busy schedule to help me.

Next, I would like the people who directly contributed to this thesis. First, I would like to thank Jim Larus, Michael Papamichael, Andreas Moshovos, and Christoph Koch for serving in my PhD thesis committee. Their guidance made my thesis better. I would also like to thank Hadi Esmaeilzadeh for participating in my thesis proposal. His relentless grilling helped me solidify my thesis. I would also like to thank Jim Larus (again) and Paolo Ienne, who served in my candidacy exam committee and provided me with instrumental advice early in my PhD. My closest collaborators were Tao Lin, Ahmet Yüzügüller, and Arash Pourhabibi. I was very fortunate to have them. Without them, I would not have a thesis.

I must also thank the people who contributed to my formation. Karu Sankaralingam, from the University of Wisconsin, was the person who put me on the map. Karu took me into his lab, gave me a very interesting project, MIAOW, and, when it was time for me to apply for grad school, guided me through the process. Without Karu, I would have never made it into EPFL. I also want to thank Alexandros Daglis. Alex was a senior student when I joined the lab. Early in my PhD, he helped me formulate my ideas and with my writing. I also learned a lot from just observing Alex working. On top of everything, Alex is also a very close friend of mine.

My lab mates were instrumental in giving me emotional support and valuable technical feedback. I was fortunate to end up in PARSA, where Babak encourages students to form a strong lab culture. Javier Picorel was the most senior student when I joined the lab. I have never met anyone with such an extensive knowledge of everything systems-related. Javier is also one of the funniest people I have ever met. Arash Pourhabibi was a collaborator and a very close friend. We even lived together for a few years. Those were happy days. Arash's is probably the most patient person I have ever known. Siddharth Gupta and Rishabh Iyer joined the lab later as interns but became my very close friends and my ski buddies. Sid is a doer, the most efficient person I have ever met. Although I cannot forgive Rishabh for defecting from the lab, he is still cool. Rafael Pizarro Solar is probably the most intelligent person I have ever met. Also, the craziest. Working with him was a very joyful and exciting journey. Dmitrii Ustiugov is the most persistent person I have ever met, whenever I think about quitting something, I think about Dmitrii never quitting anything. Nooshin Mirzadeh is a great friend. She really helped me feel at home at PARSA. Hussein Kassir was my FPGA buddy in the lab. We also tavelled a lot together, I keep good memories of those trips. Unfortunately, I could never hang out with Zilu Tian, Ognjen Glamocanin, Simla Harma, Dina Mahmoud, and Pradeep Kathirgamaraja, as much as I wanted, as they arrived as I was leaving. However, their company was still appreciated in the short time we had together. I would also like to thank my senior

students, Cansu Kaynak, Djordje Jevdjic, Stavros Volos, and Onur Kocberber, who I did not get to hang out with as much as I wished, as they were leaving when I arrived at PARSA. Finally, I would like to thank Stephanie Baillargues. Stephanie helped us in everything related to EPFL, going above and beyond to fix our messes. Several times I came to Stephanie in desperation, and she quickly solved all my problems.

I would not have made it without the moral support of my family and friends. My mom and sister, Hilda and Daina, are an inspiration and a reason for me to go on. Without them, I would not have gone to university nor have left Brazil. Being away from my family is painful, but I hope I am making them proud. I love them both very much. Another source of moral support was my wife, Juliette, who had to also endure my grumpiness when things did not go well. She always cheers me up, and her support is the main reason I never quit my PhD. I love her more than anything. My parents-in-law, Maria and Jorge, and my sisters-in-law, Sarah and Stéphanie, gave me a family and a home away from home. I am really thankful for that.

Finally, I would like to thank the Swiss people for accepting me in their great country and supporting me through their taxes and EPFL. Switzerland is truly the best place to live in the world, and I was very fortunate to do my PhD here. I hope I get to live the rest of my life here.

Abstract

Deep neural network inference accelerators are deployed at scale to accommodate online services, but face low average load because of service demand variability, leading to poor resource utilization. Unfortunately, reclaiming inference idle cycles is difficult, as no other workload can execute on such custom accelerators. DNN training services offer opportunities to reclaim inference accelerator idle cycles. However, the inference services' tight latency constraints and the training algorithms' dependence on floating-point arithmetic limit the opportunities for piggybacking training services on inference accelerators.

In this thesis, we tackle the challenges that prevent inference DNN accelerators from exposing their idle cycles to training services. We first develop an efficient numeric representation that enables DNN training with accuracy similar to single-precision floating point and energy efficiency similar to 8-bit fixed point. Then, we explore the inference accelerator design space to show that, unlike in current latency-optimal platforms, relaxing latency constraints with ALU arrays that are batching-optimized achieves near-optimal throughput for a given area and power envelope. High throughput inference accelerators maximize the opportunities to piggyback training. Finally, we present Equinox, a family of inference accelerators designed to piggyback training. Equinox employs a uniform encoding and a priority hardware scheduler that processes training requests during inference idle cycles without affecting inference tail latency. Overall, we show that exposing accelerator idle cycles to training services uncovers significant computing power for training services with a small overhead for inference accelerators, improving overall datacenter efficiency.

Keywords: datacenters, deep neural network accelerators, online services, systolic array, arithmetic representation, block floating point

Résumé

Les accélérateurs d'inférence des réseaux neuronaux profonds qui exécutent des services en ligne sont confrontés à de faibles charges moyennes en raison de la variabilité de la demande de services, ce qui entraîne une utilisation réduite des ressources. Malheureusement, il est difficile de récupérer les cycles d'inactivité d'inférence, car aucune autre charge de travail ne peut être exécutée sur de tels accélérateurs personnalisés. Les services d'entraînement des réseaux neuronaux profonds offrent la possibilité de récupérer les cycles d'inactivité des accélérateurs d'inférence. Cependant, les contraintes de latence strictes des services d'inférence et la dépendance des algorithmes d'entraînement à l'arithmétique en virgule flottante limitent les possibilités d'utiliser des services d'entraînement pour récupérer les cycles d'inactivité des accélérateurs d'inférence.

Dans cette thèse, nous abordons les défis qui empêchent les accélérateurs d'inférence des réseaux neuronaux profonds d'exposer leurs cycles inactifs aux services d'entraînement. Nous développons d'abord une représentation numérique efficace qui permet l'entraînement des réseaux neuronaux profonds avec une exactitude similaire à la virgule flottante à simple précision et une efficacité énergétique similaire à la virgule fixe à 8 bits. Ensuite, nous explorons l'espace de conception des accélérateurs d'inférence pour montrer que, contrairement aux plates-formes actuelles de latence optimale, l'assouplissement des contraintes de latence avec des réseaux d'unité arithmétique et logique optimisés par lots permet d'obtenir un débit quasi optimal pour une enveloppe d'aire et de puissance. Les accélérateurs d'inférence à haut débit maximisent les possibilités d'entraînement pour récupérer les cycles d'inactivité des accélérateurs d'inférence. Enfin, nous présentons Equinox, une famille d'accélérateurs d'inférence conçus pour exposer les cycles d'inactivité aux services d'entraînement. Equinox utilise un codage uniforme et un ordonnanceur prioritaire de matériel qui entrelacent l’entraînement pendant les cycles d’inactivité d’inférence sans affecter la latence de la queue d’inférence. Dans l’ensemble, nous montrons que l’exposition des cycles d’inactivité des accélérateurs d’inférence aux services d’entraînement révèle une importante puissance de calcul pour des services d’entraînement avec une faible surcharge pour les accélérateurs d’inférence, ce qui améliore l’efficacité globale du centre de données.

Mots clés : centre de données, accélérateurs des réseaux neuronaux profonds, services en ligne, systolic array, représentation numérique, virgule flottante en bloc

Contents

List of Figures

List of Tables

1 Introduction

Deep neural network (DNN) infrastructure has observed an explosion in investment due to the large popularity of DNNs in online services [18, 27, 32]. Unfortunately, a significant fraction of this investment goes to waste, as custom inference accelerators face an average request load of around 30% because of service demand variability [3]. In general-purpose platforms, idle cycles are reclaimed by co-locating best-effort workloads with latency-critical ones [15]. As no other workload can execute on custom inference accelerators, those idle cycles stay unclaimed, leading to a considerable waste of resources.

Most attempts to reclaim inference accelerators' idle cycles involve using general-purpose platforms. Facebook uses CPUs for DNN inference, consolidating inference services with other best-effort workloads to improve utilization [27]. Microsoft employs FPGAs in their datacenters and leverages reconfiguration to reclaim FPGA idle cycles when load is low [5, 18]. Unfortunately, general-purpose platforms like CPUs and FPGAs are at least an order of magnitude less efficient than ASICs. As such, even when well utilized, general-purpose platforms still exhibit lower overall area and power efficiency for DNN services.

While DNN training workloads can be used as best-effort tasks to reclaim ASIC accelerators idle cycles, the divergence in inference and training workload requirements poses a significant challenge. On the one hand, inference accelerators execute algorithms that can tolerate narrow fixed-point arithmetic and have small memory footprints. As such, these algorithms are served directly out of on-chip memory to achieve high throughput [18, 24]. Because of tight latency constraints, these accelerators avoid techniques that may delay individual requests, such as batching [18, 51]. On the other hand, training accelerators require floating-point arithmetic, exhibit memory footprints in the range of a few GBs [74], which cannot be easily accommodated on chip, and have no online latency constraints. As a result, training accelerators employ arithmetic logic units (ALUs) with reduced power efficiency, operate on DRAM-resident data, and use batching to minimize data movement and maximize throughput.

In this thesis, we design an inference accelerator that enables training services to piggyback on inference idle cycles without affecting inference latency and energy efficiency.

1.1 Challenges in Piggybacking Training

There are several challenges related to designing an accelerator that piggybacks training services while maintaining inference's energy efficiency and arithmetic density. First, training algorithms require floating point arithmetic, which leads to accelerators with less dense and less efficient ALUs. Without a novel arithmetic representation for training, inference accelerators do not provide many opportunities for training services to achieve significant throughput when the load is low.

DNN accelerators also use batching to reduce data movement and improve energy efficiency. Unfortunately, batching also harms individual inference requests latency, with latency scaling as batch sizes increase. Although batch sizes are selected at compile time, the batch size that achieves the maximum energy efficiency is a property of the ALU array dimensions. As such, the ALU array dimensions define both the accelerator latency and throughput. If latency constraints are too tight, inference accelerators present low throughput, minimizing the opportunities for exposing idle cycles to training.

The final challenge is to maintain the inference service latency guarantees despite the resource sharing that occurs in inference accelerators, which also execute training services. This challenge requires in-accelerator scheduling support to prioritize inference requests when load surges occur.

1.2 Thesis Goals

This thesis aims to improve the energy efficiency of DNN accelerators by enabling the co-location of inference and training services. We do so by addressing the three showstoppers for piggybacking training on inference accelerators. First, we address the divergence between inference and training arithmetic. We aim to develop an arithmetic representation for DNN training that does not hurt accuracy or convergence rate, with ALU density and energy efficiency similar to representations used in inference accelerators. Second, we aim to find the ALU array dimensions that lead to efficient inference at low latency. Doing so enables us to find optimal designs that maximize the opportunities to expose idle cycles for training. Third, we aim to develop an in-accelerator scheduler that prioritizes latency-sensitive inference over best-effort training requests when needed. The scheduler must quickly identify and react to load surges. Finally, we aim to design an accelerator to show that piggybacking training on inference accelerators is feasible and more energy-efficient than using separate fabrics for DNN inference and training.

Thesis statement:

Exposing DNN inference accelerators' idle cycles to training uncovers vast computational resources for training at negligible overhead for inference services.

1.3 Thesis Contributions

This thesis introduces the basic features of an inference accelerator that exposes idle cycles for training. We will refer to such an architecture as ColTraIn (Co-located DNN TRAining and Inference). We then introduce Equinox, a family of ColTraIn accelerators. Finally, we evaluate a few instances of Equinox.

First, we introduce the hybrid block floating point (HBFP) [16] DNN training framework, which maximizes the use of narrow fixed-point arithmetic while preserving convergence. We also introduce wide weight storage to HBFP, to improve HBFP's precision with modest area and memory bandwidth overhead. We explore the HBFP design space to show that DNNs trained on HBFP with 12- and 8-bit mantissas match FP32 accuracy, serving as a drop-in replacement for this representation.

Second, we study the design space of inference accelerator ALU arrays. We quantify the non-linear relationship between throughput and latency that appears in inference accelerators when batching is involved. We find out that latency-optimized designs exhibit low throughput while throughput-optimized designs exhibit high latency. We conclude that optimizing for one metric alone is not an effective approach. We show that designing ALU arrays with a specific latency goal leads to superior throughput and energy efficiency, maximizing the opportunities for training services to execute on inference accelerators.

Finally, we introduce the Equinox family of accelerators and evaluate a few instances of it. Equinox introduces multiple contexts to DNN accelerators to enable inference and training services to share resources and uses the same numeric representation for training and inference. Our evaluation shows that HBFP is an efficient numeric representation, with area and power consumption similar to fixed-point representations. We also show that the combination of HBFP, latency-constrained design, and in-accelerator scheduling leads to efficient ColTraIn accelerators, with throughput and latency similar to inference accelerators.

1.4 Thesis Scope

In this thesis, we focus on systolic-array-based, single-node ColTraIn accelerators. This class of accelerators represents a significant fraction of the DNN accelerators' design space, including Brainwave [18] and TPU [32]. However, the DNN accelerator design space is much broader, featuring a wide variety of ALU arrays and distributed accelerator fabrics. While we do not evaluate the entire design space, the contributions of ColTraIn generally apply, with a few caveats. In this section, we describe how each contribution of this thesis applies to the design space of ColTraIn accelerators.

First, the numeric representation we introduce, HBFP, is independent of ALU array organization, and it applies to both single node and distributed training. The only constraint imposed by HBFP is the physical separation between dot-products and other computations. Fortunately, most accelerators separate dot products from other operations [18, 32, 51] to improve energy efficiency.

HBFP is applicable to a wide variety of image and text processing models, as shown in the evaluation, with convergence rates similar to floating-point representations. While we cannot guarantee the same convergence rates on models that were not evaluated, we believe that it is applicable in general. As we show, all individual operations performed using HBFP have accuracy similar to single-precision floating point.

Our second contribution, the design space exploration of systolic-array-based accelerators, is not directly applicable to accelerators with other ALU array organizations. The conclusions of our study, however, are general. We show how batching affects throughput and latency in all models that employ vector-matrix multiplications. Our analysis also shows that batching effects are much stronger in accelerators with ALUs that are data movement bound. Even in accelerators like GPUs, which are fundamentally different from systolic arrays, larger batches mitigate data movement bottlenecks but negatively affect latency. Finally, models based on matrix-matrix multiplications, which are not the focus of this thesis, naturally exhibit weight reuse and are not affected as much by batching.

Our third contribution, Equinox, introduces a systolic-array-based inference accelerator capable of piggybacking training services. We introduce mechanisms and policies to share accelerator resources between inference and training services and to schedule operations for both services. The mechanisms introduced are specific to systolic-array-based accelerators, but the policies apply to a wide variety of accelerators. For instance, in NVIDIA GPUs, the policies introduced by Equinox can be implemented in the CTA and warp schedulers.

We evaluate Equinox on a single node setting for both inference and training, showing that it exposes variable throughput to training services, depending on the inference load. As such, in distributed training settings — where higher-level load balancers manage the nodes — Equinox introduces significant scheduling challenges. If load balancers cannot guarantee uniform load across all nodes, then training services observe irregular performance. For training services that employ synchronous distributed algorithms, the overall throughput is capped by the slowest node (i.e., the node in which inference load is highest). Training services can issue hedged or tied requests [13], collecting gradients from whichever nodes finish batch processing first to mitigate load imbalance issues while still providing synchronous guarantees to training services.

We observe that Equinox guarantees synchronous execution in multiple nodes, albeit at reduced throughput. Additionally, if training services tolerate asynchrony, then Equinox clusters are much more effective, as nodes make progress at their own pace. Our goal is not to match the throughput of optimized training accelerators but to exploit the idle cycles already present in inference services to perform training for free.

Finally, we argue that ColTraIn accelerators introduce an efficient way of exploiting inference services' idle cycles. The alternative approach — piggybacking inference on optimized training accelerators — complicates resource provisioning for several reasons. First, training accelerators aim to provide maximum throughput, employing ALU arrays that require large batches to achieve high utilization, negatively affecting latency. Second, training accelerators maximize DRAM and network bandwidth provisioning in detriment of ALU arrays. The introduction of inference services on training accelerators leads to DRAM and network resources' underutilization and introduces contention for ALU arrays. Finally, training services are best-effort, long-running services, without as much load variance as inference accelerators and without as many idle cycles. Without idle cycles, the introduction of inference services on training accelerators would force designers to increase accelerator fabric sizes to accommodate more services.

Nevertheless, the contributions of this thesis still apply to DNN training accelerators. The HBFP encoding improves training throughput by reducing data movement, power, and area of ALU arrays. The scheduling mechanisms introduced by Equinox also apply to training accelerators that piggyback inference services, as they too must prioritize inference services when load is high.

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 provides background on DNN accelerator design, describing the design space and providing an in-depth explanation for the diversion between inference and training accelerators. Chapters 3, 4, 5, and 6 present the contributions of this thesis.

- Chapter 3 introduces the HBFP arithmetic encoding.
- Chapter 4 introduces our methodology to size ALU arrays for latency-constrained DNN accelerators, mapping the latency-throughput trade-off for inference accelerators.
- Chapters 5 and 6 introduce the design and implementation of Equinox, a family of ColTraIn accelerators.

Finally, we present related work in Chapter 7 and conclude the thesis in Chapter 8.

2 Background

DNN workloads are fundamentally different from traditional datacenter services for various reasons. First, DNNs must be trained before they are deployed for inference. Second, both DNN inference and training require algorithms that are computationally and data intensive. Third, DNNs have broad applicability, being used in services ranging from image analysis to text and speech processing. These requirements have forced datacenter operators to adopt accelerators for both inference and training. This chapter provides an overview of DNN workloads and the accelerators that execute them.

2.1 DNN Workloads

DNN workloads are fundamentally different from traditional workloads due to their high computational demands and data intensity. For instance, a single Resnet50 [28] request requires around 4 billion instructions that process a model of around 100MB, requiring several orders of magnitude more data and computation than a single 128-byte memcached [40] request, which requires 175 thousand instructions. Additionally, DNNs can be applied to solve more problems than most other algorithms, both in datacenters and at the edge [27, 32], finding application in image processing, web search, text processing, speech processing, and more.

The broad applicability of DNNs leads to their application in a wide variety of computing systems. In the cloud, online services combine DNNs with traditional datacenter workloads to provide a customized experience for users [18, 27, 32]. The high volume of DNNs services has lead datacenter operators to adopt specialized accelerators that are designed to fit under traditional servers' power budgets [18, 27, 32]. These accelerators are part of traditional datacenter servers, usually not optimized to host DNN services, and are accessed through PCIe or network interfaces. As such, datacenter DNN services are subject to the same constraints of online services, often facing contention for resources [73]. In this thesis, we focus on datacenter accelerators due to their large scale and many opportunities for exposing inference idle cycles to training workloads.

DNNs are also executed on dedicated clusters designed with high per-server power budgets, often featuring several accelerator instances per server board [23, 22, 52]. These clusters often feature servers specialized for DNN services using high-performance networks. These clusters execute both latency-constrained and best-effort DNN services.

Finally, DNNs are also used at the edge [66], where they face tight power constraints, which lead to the use of efficient numeric representations, even if it affects accuracy. The biggest challenges of edge DNN services are the diversity of both the compute fabrics, where some nodes are equipped with DNN accelerators and others are not, and the workloads they execute. Additionally, the requirements of edge DNN services vary widely [44], from real-time (i.e., self-driving cars) to best-effort (i.e., training of face recognition models).

DNNs also differ from traditional workloads because they have to be trained before they are deployed for inference. While the development of traditional workloads requires relatively low computational resources, compared to their actual execution, DNN training is a computational and data-intensive process. DNNs' training algorithms often require millions of iterations [61, 74], which process large datasets. Additionally, DNN training involves exploring the design space, requiring the training of a large number of different variations of a particular model to reach optimized results. Although the training process has different requirements from DNN

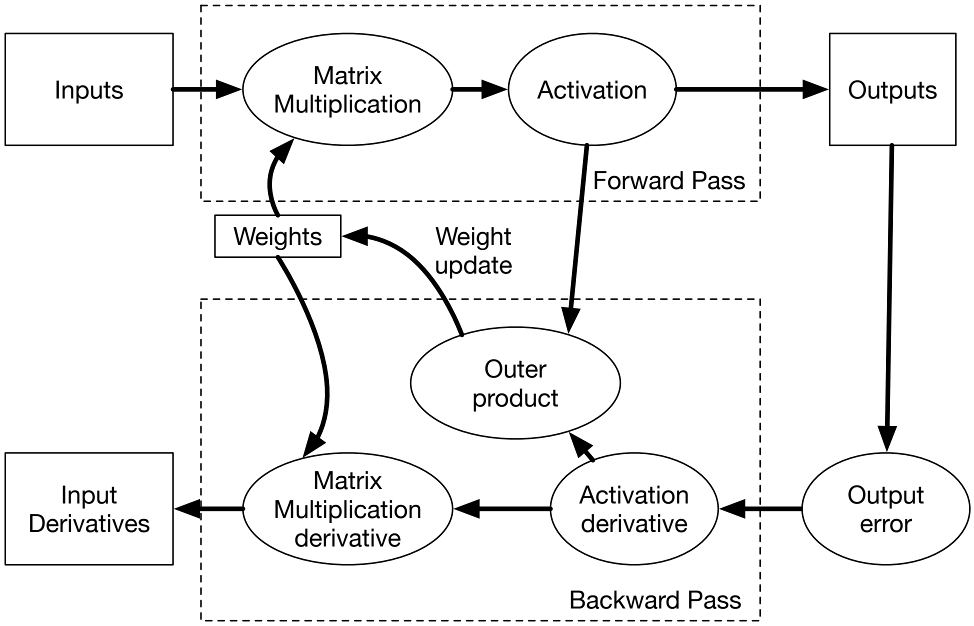


Figure 2.1 – DNN operations.

inference, it is composed of similar operations.

DNN workloads are composed of chains of large matrix operations, which dominate execution time, followed by large vectorial operations. Figure 2.1 shows the operations of a single layer DNN. Squares indicate inputs, outputs, and weights, while ellipses indicate operations. The arrows indicate the operands that flow between operations, also known as tensors. The top part of the figure shows a forward pass. First, groups of inputs are organized into tensors that are then multiplied by the weight matrix. The results then go through a non-linear activation function. For DNNs with multiple layers, this process is done multiple times. The bottom part of the figure shows the backward pass. It starts by computing the output error, which is then processed by derivatives of the activation functions to compute the activation derivatives. The activation derivatives are then fed to the matrix multiplication derivatives, which are sent to previous layers. Activation derivatives are also used to compute the weight updates.

One critical issue in performing the matrix operations present in DNNs is the high relative power consumption of moving data, compared to the power consumed by ALU operations. Accelerators have to exploit reuse in DNN processing to mitigate this issue. DNN services are diverse, with different models leading to various matrix operations dimensions, with varying levels of reuse. Traditional multi-level perceptrons (MLP) models and recurrent neural networks like long short-term memory networks (LSTM) and gated recurrent unit networks (GRU) are based on vector-matrix multiplications. These models have a lot of inherent activation reuse, with each activation value reused up to a few thousand times [47], but no weight reuse. As such, datacenter operators are forced to batch requests together to increase reuse. Other models, like convolutional neural networks (CNN) [28] and Transformer-based [63] models, are based on matrix-matrix multiplications, offering a lot of activation and weight reuse, leading to more efficient accelerators. Additionally, the relative sizes of activation and weight matrices affect efficiency. In MLPs, RNNs, and Transformer-based models, most of the memory accesses are to model weights [74] while in CNNs, memory accesses are more equally divided between weights and activations [74].

2.2 Physical Constraints in DNN Accelerators

The DNN workloads' high computational density leads to increased pressure on the accelerators' power and area envelopes. In this section, we describe how these constraints affect accelerator design. We first explain how the power constraints lead to data movement bound accelerators, and then we explain how the area constraints affect accelerator design.

The end of Dennard's Scaling led to power-constrained silicon devices [17], also affecting DNN accelerators. Accelerator ALUs, the high-throughput buffers that feed them, and off-chip memory interfaces consume most of the accelerator power [6, 24], limiting the maximum accelerator throughput. Additionally, ALUs and buffers interfaces are power dense [11, 24] and cannot be used to fully occupy dies. To cope with the power density, accelerator designers reduce the operating frequency and increase the number of ALUs, to exploit the parallelism inherent to DNN workloads.

The numeric representations used in accelerators' ALUs also affects their power efficiency. Compared to fixed point, floating-point representations cover a vast range of values but incur high area and power for hardware implementations. Floating point numbers are represented by a mantissa and an exponent, in contrast to fixed-point numbers represented only by a mantissa. As such, a number in floating point is represented by the form $mantissa \times 2^{exponent}$. The exponent dynamically adjusts the range of values represented by the mantissa. However, compared to fixed point, the addition of an exponent requires hardware implementations to employ complex circuitry to manage the exponents and to normalize mantissas.

Fixed point and narrow representations reduce the power and area spent per operation [11] by similar amounts, keeping the ALU power density constant. However, taking into account data movement power, efficient ALUs mitigate the power density issues. For instance, 8-bit fixed-point ALUs consume up to one order of magnitude less power and area than bfloat16 ALUs. However, they reduce data movement power by $2\times$. Assuming that the buffer area remains the same, the power density of 8-bit ALUs and buffers that feed them is $2\times$ lower than bfloat16's.

Using efficient ALUs increases the fraction of power dedicated to data movement in accelerators. Floating-point ALUs consume more energy to perform operations than to access operands. However, as ALUs become more efficient, this trend changes. The energy required to access 8-bit operands from a 32kb SRAM is $6.3\times$ larger than the energy needed to perform an 8-bit fixed-point multiply and accumulate operation [11]. If data is accessed from an off-chip buffer, the difference is three orders of magnitude. As such, accelerators that do not exploit reuse spend a significant fraction of their power in data movement, limiting the power dedicated to ALUs and accelerator throughput.

DNN accelerators exploit reuse in various ways. On the one hand, accelerators with many SIMD-like processors offer more flexibility in processing different matrix operations. These processors store data in register files and scratchpad memories for reuse. Unfortunately, this organization also incurs high synchronization and communication overheads. Traditional GPUs [50, 51], (left-hand side of Figure 2.2) are highly threaded, distributing threads across processors, which communicate through the memory system. Inter-thread communication is

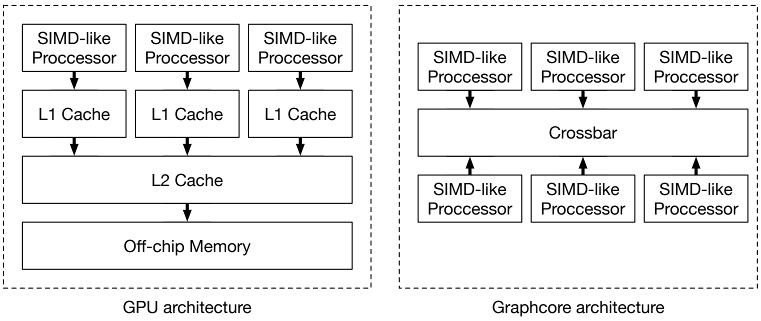


Figure 2.2 – Accelerators employing fine-grained ALU arrays.

often a bottleneck in GPUs [74]. Graphcore [24] (right-hand side of Figure 2.2) uses a similar architecture, but, instead of communicating through the memory system, the processors communicate directly through a large on-chip crossbar. Graphcore divides execution into super steps composed of computation and communication phases, with all processors turned off during the communication phase.

There are also accelerators with large monolithic matrix processing units, featuring ALU arrays connected through a network that is specialized to implement matrix operations. These units access shared on-chip buffers, which are often partitioned to improve their efficiency. Large monolithic units incur fewer communication overheads, because of the specialized network, but are less flexible in the operations they can implement. Microsoft's Brainwave [18] uses large SIMD units specialized for dot products connected through an accumulator tree, like the ones depicted on the left-hand side of Figure 2.3, while Google's TPU [32] uses systolic arrays like the ones pictured on the right-hand side of Figure 2.3. In this thesis, we focus on monolithic processing units due to their high efficiency. We also observe that they exhibit an explicit relationship between the ALU array dimensions and reuse, facilitating the exploration of the accelerator design space.

Area constraints affect DNN accelerators by limiting the number of ALUs and the amount of on-chip buffer available. As such, the slowdown in Moore's Law has reduced the opportunities for accelerators to exploit the vast parallelism in DNNs and to increase on-chip capacity. On-chip capacity is crucial on DNNs because on-chip buffers provide at least an order-of-

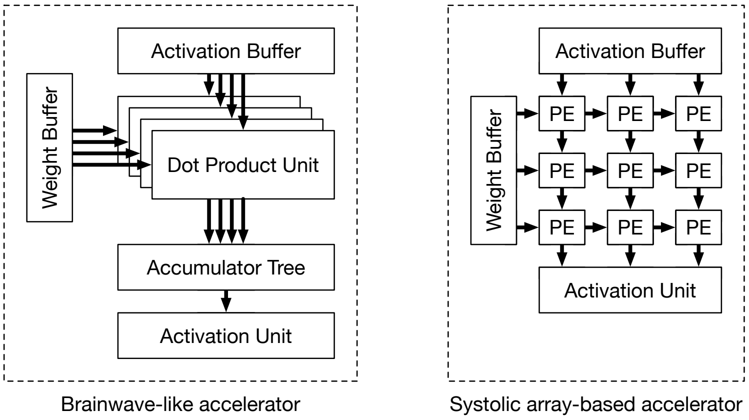


Figure 2.3 – Accelerators employing monolithic matrix multiplication units.

magnitude higher bandwidth than off-chip buffers [18, 24]. As such, if on-chip buffers are large enough to accommodate all the DNN service memory footprint, DNN accelerators can achieve superior throughput without relying so much on reuse.

2.3 DNN Inference Accelerator Design

Inference accelerators are conventionally designed to maximize throughput while honoring tight latency requirements for online services. A salient characteristic of inference workloads is their tolerance to narrow fixed-point numeric encoding [18, 32]. Such an encoding results in up to an order of magnitude improvement in ALU silicon density (relative to floating point) [11], in memory capacity [18], and data movement bandwidth and power [61]. For example, Microsoft's Brainwave uses block floating point to process both LSTMs and CNNs [18], and TPUv1 uses 8- and 16-bit fixed point [32]. Using fixed-point-like representations allows both Brainwave and TPUv1 to achieve superior throughput and lower latency than the state of the art when introduced.

Another critical characteristic of inference workloads is their small memory footprint enabling designers to provision a more significant fraction of overall power for ALUs and higher throughput. In inference, the footprint is dominated by model weights, which are often small enough (a few KBs to 100s MBs [74]) to fit entirely on chip [18, 24] with off-chip memory only present to accommodate the less frequent case of larger models (e.g., Brainwave [18], NVIDIA's T4 [51]). Consequently, data movement in inference accounts for a small fraction of the overall accelerator power budget. On-chip memory accesses consume little power (up to three orders of magnitude less, relative to off-chip accesses [11]). Both Graphcore [24] and Brainwave [18] are designs that exploit large on-chip memories with 10s-100s MBs worth of capacity for inference models.

While inference services lend themselves well to designs with high computational density, they are often online and have tight latency constraints. Inference services are usually part of multi-tiered services where a user query triggers a sequence of sub-query fan-outs, spanning hundreds or thousands of servers [8, 27]. That fan-out effect places tight bounds on the tail response latency of each tier [32, 55]. In DNN services, each tier is comprised of one or more accelerators. DNN accelerators lack all the sources of latency variance present in traditional servers, leaving only service time and queuing delays as sources of latency.

Inference's sensitivity to service-level tail latency creates a non-intuitive relationship between latency and throughput and a dilemma for designers. To increase effective throughput given a fixed power budget, designers rely on batching [18] to increase weight reuse, minimizing data movement bandwidth and power, thereby increasing the power provisioned for ALUs and throughput. At one end of the design space, when latency requirements are tight, lower degrees of batching result in a cap on the accelerator throughput because of the power provisioned to on-chip memory. At the other end, when latency requirements are lax with higher degrees of batching, the on-chip data movement power becomes negligible, freeing up power for ALUs and throughput.

Finally, inference services face low average loads, of around 30% [3], due to service demand variations. Traditional general-purpose servers use best-effort services to improve utilization when the online load is low [15, 39]. In general-purpose servers, the biggest challenge is the complex interactions between the various resources in CPUs, which causes best-effort services to interfere with online services in non-intuitive ways. In the case of DNN accelerators, partitioning resources is trivial, as compilers can precisely calculate resource utilization. As such, the biggest challenge is that inference accelerators cannot execute other tasks when service load is low, leading to low utilization.

2.4 DNN Training Accelerator Design

A fundamental difference between training and inference stems from the prominent algorithm, stochastic gradient descent (SGD), which poses several limitations on accelerator design. First, SGD requires floating point to converge, leading to ALUs with up to an order of magnitude more area and power consumption than the fixed-point ALUs used in inference services. SGD requires floating point because it operates on values with a wide range. SGD trains neural networks by computing the derivatives of a loss function, which represents how well the DNN performs, with regards to weights and activations. The weights and activation values are much larger than their derivatives, which are based on small differences between the results predicted by the network and the target results. Floating-point represents numbers with a mantissa and an exponent, with the exponent being used to adjust the representation range. Unfortunately, the exponent also introduces high area and power overheads, which affect overall ALU efficiency.

Prior work has attempted to train DNNs with fixed-point arithmetic [36, 64, 71], which represents numbers with a mantissa. While it has been shown that it is possible to achieve training convergence using fixed point, the limited range of fixed-point representations imposes algorithmic restrictions. These restrictions force algorithm designers to impose disruptive modifications to SGD, limiting its applicability and increasing the training process's complexity. The state-of-the-art in DNN training has evolved towards using narrower floating point representations (i.e., bfloat16 [14]), which are more efficient than single-precision floating point, but still less efficient than fixed point.

Second, SGD employs backpropagation, an algorithmic stage with long-term dependencies requiring a memory footprint in the order of up to a few GBs [74]. Such capacity can only be fulfilled by off-chip DRAM, a memory technology that consumes at least an order of magnitude more energy per access than SRAM. The DRAM's high energy consumption limits the maximum bandwidth, forcing DNN accelerators to exploit more reuse from DNN workloads to achieve high throughput. Unfortunately, SGD also imposes a limitation on the maximum batch size used [38], limiting the maximum weight reuse on DNNs. Batch size limitations, in addition to DRAM's limited bandwidth, leads to accelerators with lower throughput compared to inference accelerators.

Training is often performed in a distributed fashion due to its high computational demands. In distributed training, each the DNN inputs or models are divided across many workers, which compute partial weight updates. Then, the partial weight updates are accumulated across workers, and the full updates are sent back to workers. Distributed training incurs high network bandwidth demands due to the exchange of weight updates.

3 Hybrid Block Floating Point

DNN training algorithms require floating point to converge, in contrast to inference, which tolerates fixed-point arithmetic without any accuracy loss [21]. The algorithmic tolerance to fixed-point arithmetic enables inference accelerators to use ALUs with up to an order of magnitude higher silicon density and energy efficiency than the ALUs used by training accelerators. In this chapter, we design an efficient arithmetic representation for training by exploiting dot product operations' properties, which dominate DNN processing.

Numeric representations have two key properties—precision and range— which affect training convergence in different ways. Precise numbers represent values with high resolution within a scale. For instance, an 8-bit fixed-point number can represent 256 values, while a 16-bit fixed-point can represent around 65 thousand values, leading to higher precision. Single-precision floating point numbers have a 24-bit mantissa and an 8-bit exponent, meaning that, for a given exponent, the floating point can represent almost 16 million values. DNNs are sensitive to numeric precision, but only up to a certain point. Table 3.1 shows the validation error obtained when training ResNet-20 [28] models on CIFAR10 [35] using floating point-multipliers with various mantissas and exponent widths. We observe convergence without loss of accuracy with 8-bit mantissas, convergence with a small loss of accuracy with 4-bit mantissas, and divergence only when using 2-bit mantissas. As such, the 24-bit mantissa used in single-precision floating point is overkill. For this reason, machine learning developers

Table 3.1 – Validation test error of ResNet-20 on CIFAR-10 as a function of the mantissa bit width.

2	4	8	24
N/A	9.77%	8.05%	8.42%

Table 3.2 – Validation test error of ResNet-20 on CIFAR-10 as a function of the exponent bit width.

2	6	8
N/A	14.67%	8.42%

started using bfloat16 [14], which employs an 8-bit mantissa and an 8-bit exponent [23, 22].

DNNs are, however, sensitive to the range of numeric representations. As such, the exponent width cannot be reduced because of its impact over the numeric range. As Table 3.2 shows, validation accuracy decreases as we reduce the exponent width from 8 to 6 bits, and training diverges altogether when using 2-bit exponents. Additionally, the use of half-precision floating points, which use 5-bit exponents, affect training convergence [12]. This reliance on representation range prevents training algorithms from converging using fixed-point representations since they are the equivalent of a floating point with a 0-bit exponent.

While floating-point is a more appropriate representation for training due to its wide range, floating-point ALUs have low silicon density and energy efficiency compared to their fixed-point counterparts. The small silicon density and energy efficiency of single-precision floating point (fp32) has led developers to adopt half-precision floating point (fp16) and bfloat16 as the state-of-the-art in representations for accelerators. However, the logic overhead of lower precision floating-point representations is still high compared to that of fixed point. For instance, although the area of an fp16 multiplier is 4.7× smaller than that of an fp32 multiplier, it is still 5.8× larger than its 8-bit fixed-point counterpart [11], with similar trends for energy consumption. This significant overhead is due to the additional hardware necessary to manage the exponent and the mantissa alignment in floating-point representations.

Block floating point (BFP) is a compromise between fixed and floating point. Like floating point, BFP represents numbers with mantissas and exponent and therefore exhibits a wide dynamic range. Unlike floating point, BFP shares a single exponent across a block of values. As such, within a block, BFP enforces a fixed-point-like range, but different blocks can have different exponents, leading to a wide range of representation. This property enables dot products to be computed with high precision but prevents arbitrary operations from being computed precisely. Additionally, BFP ALUs achieve higher silicon density and energy efficiency because they amortize the exponent management overhead over many values.

In this chapter, we show that using BFP for dot product operations and floating point for other operations leads to a training accelerator with silicon density and energy efficiency comparable to fixed-point accelerators and training convergence rate similar to floating-point accelerators.

3.1 Dot Products with BFP Arithmetic

Equation (3.1) computes the real value a_i of an element i of a BFP tensor a with mantissa m_i^a and exponent e_a.

$$a_i = m_i^a \times 2^{e_a} \tag{3.1}$$

In this example, BFP can only represent a accurately if the value distribution of a is not too wide to be captured by m^a, and the exponent e_a is representative of said value distribution. If e_a is too large, then small values are lost, and the most significant bits of the mantissas are wasted. If e_a is too small, then the larger values in a will be saturated, leading to data loss.

Equation (3.2) calculates the dot product between BFP tensors a and b, each with N elements.

$$a \cdot b = \sum_{i=1}^{N} \left((m_i^a \times 2^{e_a}) \times (m_i^b \times 2^{e_b}) \right) = 2^{e_a + e_b} \times (m^a \cdot m^b) \tag{3.2}$$

The dot product $m^a \cdot m^b$ is computed entirely in fixed-point arithmetic, without the alignment of intermediate values, since all elements m_i^a and m_i^b are fixed point. In a matrix multiplication $A \times B$, it is enough for A to have one exponent per row, and B to have one exponent per column. In DNNs, activations or activation derivatives would map to the A matrix, and weight matrices would map to B. During training, input matrices are multiplied by both the weight matrix (during forward pass) and their transposed forms (in backpropagation). As such, using one exponent per column in weight matrices would lead to a dot product with a different exponent per weight value, which would degenerate to a floating-point-like dot product. We address this issue by using matrix tiles as BFP blocks. With tiled matrices, tile multiplications are performed using fixed point, and the resulting tiles are accumulated using floating-point arithmetic, requiring mantissa realignment.

3.2 Requirements for Numeric Representation

This section describes the challenges and opportunities presented by each operation in DNNs to the arithmetic representations. We start by explaining how different operations present different challenges for numeric representations, making a case for using different representations for different operations in DNN processing.

Dot products — the operations that dominate DNN processing — feature long reductions, which introduce error in hardware that uses floating-point. To calculate dot products, we first compute the element-wise product of the two input tensors, and then reduce the results. In floating-point additions, the mantissa's inputs are first aligned, to normalize their exponents. In this process, the value with the smallest exponent has its mantissas shifted to the right. If the exponent difference is too large, the mantissa of the smallest value can be partially or entirely lost, minimizing that value's contribution to the addition. Over the several additions that compose a reduction, this process is repeated, shifting away the smaller values' mantissa bits, leading to accuracy reduction. The convergence of DNN training with floating-point indicates that the training algorithm is robust enough to tolerate this loss of accuracy.

When narrow floating-point numbers are used, however, the loss of accuracy starts affecting DNN accuracy [42], forcing accelerators to employ mixed-precision arithmetic. These accelerators use narrow representations, like bfloat16 or half-precision floating point, for dot products inputs and single-precision floating point for accumulators. Single-precision floating point accumulators are useful because of their 24-bit mantissas, which enable values with large differences between exponents to be aligned without loss of precision. However, the single-precision floating point adders used in these accelerators consume significant area and power.

We observe that block floating point (BFP) is a promising numeric representation for dot products because it behaves similarly to floating points. When blocks of values are converted to BFP, the largest value's exponent is used as the exponent of the entire block, and all the other values are aligned. As such, the smallest numbers of the block lose precision, like in floating-point reductions. Additionally, BFPs can implement precise floating-point dot products when accumulators are wide enough. Using wide accumulators in BFP dot products can be done at low overhead when compared to floating-point dot products because fixed-point adders have low area and power compared to fixed-point multipliers.

BFP, however, may negatively affect accuracy when blocks are large. Large blocks force more values to share exponents, requiring a larger number of numbers to fit into the fixed-point range imposed by the mantissa. While this is a problem in arbitrary operations, dot products do not suffer from this issue. When dot products are large, they exhibit an averaging effect, reducing the impact of small individual values over the result.

In activations and batch normalizations, larger input values may not dominate the outputs as they do in dot products. As such, the use of BFP might lead to loss of accuracy. BFP-based accelerators convert dot product results from their wide accumulators back to narrower mantissas by identifying the largest exponents in the block, aligning, and truncating all of the accumulators, reducing the precision of smaller values in the process. Some activation functions, however, may amplify smaller values, amplifying the accuracy reduction. Additionally, in the backward pass, the derivative of some activation functions have their active region around zero, meaning that only values around zero are propagated to the previous layers. Again, if only BFP is used, these values would be lost. Finally, using BFP in activations and batch normalizations have reduced benefits due to the extra logic used to manipulate and align mantissas.

As such, we show in this chapter that using floating-point ALUs for activations and batch normalizations enables training convergence with minimal impact on energy efficiency. The overhead of using floating-point ALUs for arbitrary operations is small because each floating-point ALU is fed by a dot product unit featuring hundreds of multiply and accumulate (MAC) units. To draw an upper bound on the overhead of the floating-point ALUs, we assume that each floating-point ALU is 10× larger than a fixed-point MAC unit and is fed by a dot product unit with 100 MACs, the overhead of the floating-point ALUs is 10%. We expect this number to be smaller, given that the average dot product unit in accelerators is wider than 100 MACs [18, 32].

3.2.1 Minimizing BFP Data Loss

To minimize data loss in model weights that last through the training process, we store weights with wider mantissas. All operations are executed using the original mantissa. Only weight updates use the wider mantissa. Therefore, we still reduce the memory bandwidth requirements for forward and backward passes, during which only the most significant bits of the weights are accessed. Weight updates are the only operations that access the least significant bits of the weight values.

3.3 BFP Accuracy Analysis

In this section, we show that BFP incurs low accuracy degradation for dot product operations but can incur high accuracy degradation for arbitrary operations. We do so by emulating BFP dot products and BFP sigmoid derivative operations on synthetic data while varying the key parameters of the design space: mantissa bit width, accumulator bit width (for dot products), block size, and range of input values.

We measure the relative root mean square error (RRMSE) between BFP and single-precision floating points. The RRMSE value indicates a ratio between the average error and the total value of the floating-point baseline. As such, an RRMSE value of zero indicates that BFP and floating point are identical, and an RRMSE of 0.1 indicates that the error incurred by BFP is, on average, 10% of the floating-point value.

3.3.1 BFP Dot Product Accuracy

We evaluate the accuracy of BFP dot products by simulating fully connected layers. We compare the result of matrix multiplications of synthetic matrices, generated randomly using the distribution of values observed in original activation and weight matrices.

For each experiment, we varied one of the parameters that affect BFP accuracy. Unless otherwise stated, we use mantissa and accumulator bit widths of 8- and 24-bits, respectively. We multiplied weight and activation matrices with 100 × 100 elements. Weight matrices have a normal distribution with zero average and a range of -4 to 4, while activation matrices have a range of -4 to 4. These value distributions were observed when training LSTMs with floating point.

The mantissa bit width is the most important parameter of BFP, as it defines the precision and the range of values that can be represented within a tensor. Figure 3.1a shows the RRMSE incurred by BFP with various mantissa bit widths while keeping the other parameters constant. As expected, the Figure shows that the error exponentially decreases as we increase the bit width, settling down at 0.01%. As DNNs operate on noisy data, the error incurred by 8-bit BFP (around 2%) does not disturb the training process. For models more sensitive to arithmetic precision, 16-bit BFP introduces almost no error. Finally, we execute each experiment a thousand times and report the distribution of the RRMSE values in box plots.

The width of the accumulator used is also an important parameter of BFP dot products, defining accuracy. Figure 3.1b shows the RRMSE of BFP dot products with various accumulator bit widths. As the Figure shows, accumulators with less than 16-bits result in an unacceptable error. Dot product accumulators add values resulting from the multiplication of two mantissas. As such, accumulators store values that require twice the mantissa width to be represented, incurring high error when accumulators are narrower than that. We observe little benefits as the accumulator width increases over twice the mantissa width.

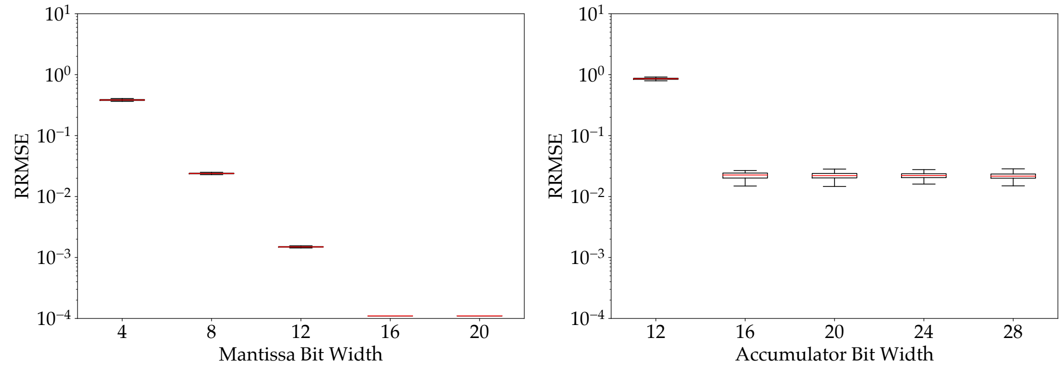
We also evaluate the influence of input weight matrix range on BFP dot product accuracy, as Figure 3.1c shows. BFP dot products are very robust to input range variations due to the reduction step, which leads to large values dominating the sums. As such, we show that the precision of dot products is virtually the same independent of the range used. We also varied the range of the activations (not shown in the Figure), showing that dot products are insensitive to variations in the range of either operand.

Finally, we evaluate whether the size of the block used in BFP dot products affects the output accuracy. We do so by varying input matrices' sizes, as shown in Figure 3.1d. Precision slightly improves as we increase the matrix size, forcing larger matrices to share exponents. Smaller dot products result in more noisy results, which are amplified by BFP. As we increase the matrix sizes, the long reductions have an averaging effect on the results, reducing the noise.

3.3.2 BFP Sigmoid Derivative Accuracy

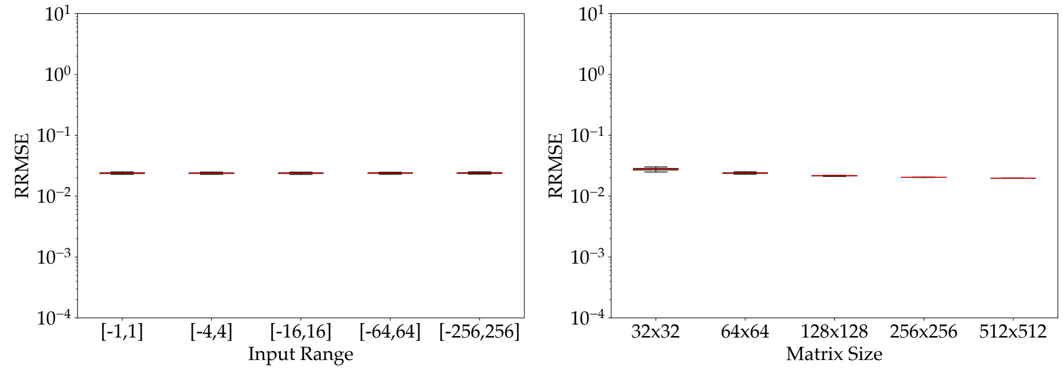
To show a case when BFP incurs high error, we evaluate the performance of BFP on the sigmoid derivatives, which appear on LSTM training. BFP reduces the accuracy of such operations because they amplify the inputs' small values, which are shifted away during conversions. We compare the result of activations applied to synthetic matrices generated randomly using the distribution of values observed in the original activations. Unless stated otherwise, all results use mantissa bit widths of 8-bits. We used activation matrices with 100 × 100 elements. Activation matrices have a normal distribution with zero average and a range of -4 to 4. These value distributions were observed when training LSTMs using floating point.

We show that the mantissa width of BFP has a similar effect on sigmoid derivatives and matrix multiplications, as shown in Figure 3.2a. As expected, the Figure shows the same trends as in dot products, with error exponentially decreasing as we increase the bit width, settling down at 0.01%.



(a) BFP RRMSE of dot product for different mantissa widths.

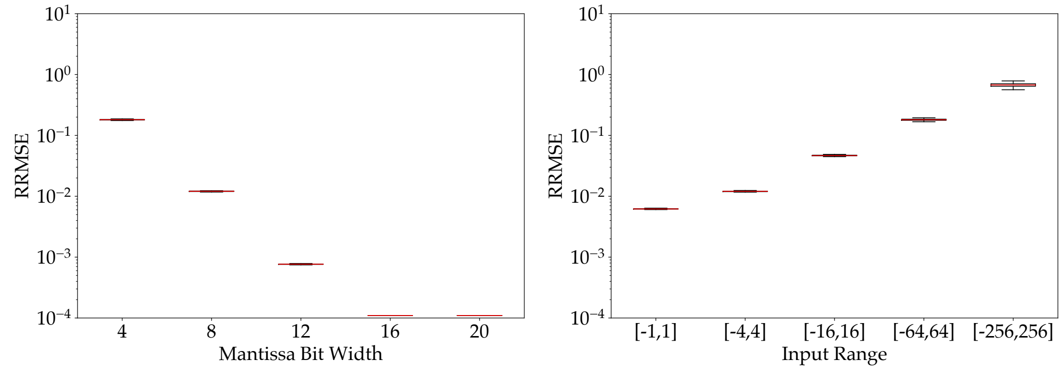
(b) BFP RRMSE of dot product for different accumulator widths.



(c) BFP RRMSE of dot product for different input ranges.

(d) BFP RRMSE of dot product for different tile sizes.

Figure 3.1 – Relative root mean square error (RRMSE) incurred by the use of BFP dot products, compared to single-precision floating point results.



(a) BFP RRMSE of sigmoid derivative for different BFP mantissa widths.

(b) BFP RRMSE of sigmoid derivative for different input ranges.

Figure 3.2 – Relative root mean square error (RRMSE) incurred by the use of BFP sigmoid derivatives, compared to single-precision floating point results.

Finally, we evaluate the effect of input range on BFP sigmoid derivatives. We show that error increases as the input range increases. The error increase is the main reason why BFPs are not suitable to process arbitrary operations. Although value ranges larger than [-4,4] are rare in the input matrices of derivative operations, they affect results significantly when they appear, disrupting the training process.

3.4 Methodology

3.4.1 HBFP Simulation on GPU

We train DNNs with the proposed HBFP approach, using BFP in the compute-intensive operations (matrix multiplications, convolutions, and their backward passes) and FP32 in the other operations. We simulate BFP dot products in GPUs by modifying PyTorch's [54] linear and convolution layers to reproduce the behavior of BFP matrix multipliers. We redefined PyTorch's convolution and linear modules using its *autograd.function* feature to create new modules that process the inputs and outputs of both the forward and backward passes to simulate BFP. In the forward pass, we convert the activations to BFP, giving the x tensor one exponent per training input. Then we execute the target operation in native floating-point arithmetic. In the backward pass, we perform the same pre-/post-processing of the inputs/outputs of the x derivative.

We handle the weights in the optimizer. We created a shell optimizer that takes the original optimizer, performs its update function in fp32, and converts the weights to two BFP formats: one with wide and another with narrow mantissas. The former is used in future weight updates while the latter is used in forward and backward passes.

3.4.2 Evaluation Setup

Baseline. We use models trained with single-precision floating point (fp32) as a baseline. fp32 is the most accurate representation used in DNN training. As such, showing that HBFP presents accuracy comparable to fp32 also indicates that it is as or more accurate than other floating point alternatives, like half-precision floating point and bfloat16. Additionally, to make the case that HBFP is a drop-in replacement for fp32, we trained both HBFP and fp32 models with the same hyper-parameters.

Datasets. We experiment with a set of popular image classification tasks with the CIFAR-100 [35], SVHN [48], and ImageNet [59] datasets. We used standard data augmentation [28, 29] for CIFAR-100 and no augmentation for SVHN. We also evaluate LSTM language modeling tasks with the Penn Tree Bank(PTB) dataset [43]. Finally, we evaluate the BERT [20] language modeling model with the English Wikipedia dataset [65].

Evaluation metrics. To evaluate the impact of HBFP and explore the design space of different BFP implementations, we tune the models using fp32 and then train the same models from scratch with the same hyperparameters in HBFP. For the image classification experiments, we report training loss and validation top-1 error. For the language modeling models, we report training loss and validation perplexity.

Training. We use a WideResNet [69] trained on CIFAR-100 to explore the BFP design space, evaluating models trained with various mantissa widths. To show that HBFP is a viable alternative to FP32, we train a wide range of models using various datasets. We train ResNet [28], WideResNet [69], and DenseNet [30] models on the CIFAR-100 and SVHN datasets; a ResNet model on ImageNet and the LSTM from [41] on PTB. We also trained BERT-Tiny from [20]. We trained all models using the same hyperparameters reported in their respective original papers.

BFP configuration. Both the BFP emulation framework and our prototype use accumulators large enough to accommodate the results of the dot products without saturation. For instance, for a 32-wide BFP dot product with 8-bit mantissas, we use 21-bit accumulators. We found out that the overhead of the wide accumulators is small.

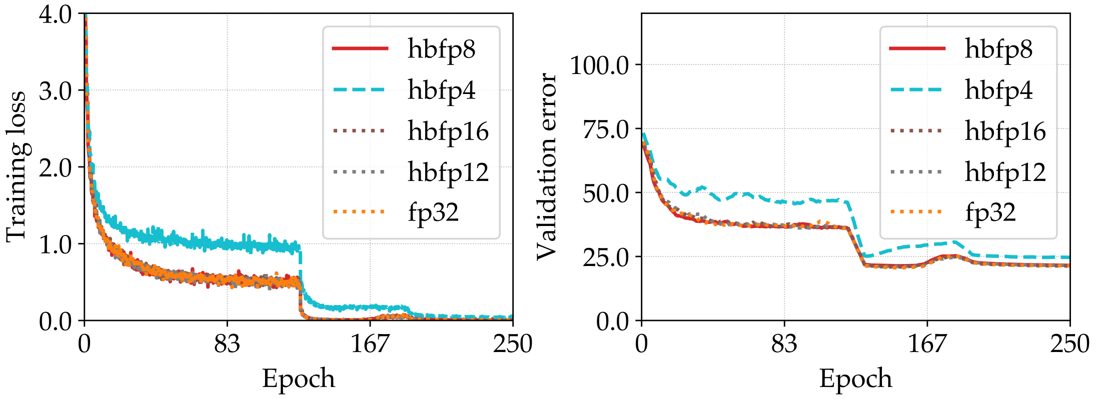


Figure 3.3 – HBFP design space for various mantissa widths. *hbfpX_Y* indicates an experiment with with X-bit mantissas and Y-bit weight storage. All dot product operations are performed with X-bit arithmetic.

3.5 Evaluation

We now evaluate DNN training with HBFP. We explore the design space of HBFP, varying the mantissa width to find the best-performing configurations. Then, we move on to evaluate HBFP on various datasets and tasks, to show that HBFP is indeed a drop-in replacement for fp32.

3.5.1 HBFP Design Space

To explore the HBFP design space, we train WideResNet-28-10 models on CIFAR-100 using various HBFP configurations, showing the results in Figure 3.3. We train models with 4-, 8-, 12- and 16-bit wide mantissas. All models with mantissas wider than 8 bits result in final validation error within 1% of the fp32 baseline, with only 4-bit mantissas showing a large accuracy gap, with 4.1% larger error. We also evaluate models with 8- and 12-bit mantissas paired with 16-bit weight storage. We observe small accuracy improvements of 0.21% and 0.43% over their counterparts with narrow weight storage. We observe similar trends in other models.

The sweet spot in the design space is HBFP with 8- to 12-bit mantissa, 16-bit weight storage. This configuration matches fp32 accuracy while improving arithmetic density and reducing

Table 3.3 – Test error of image classification models. RN, WRN and DN indicate ResNet, WideResNet and DenseNet, respectively. *hbfpX_Y* indicates an experiment with X-bit mantissas and Y-bit weight storage. All dot product operations are performed in X-bit arithmetic.

	CIFAR 100			SVHN			ImageNet
	RN-50	WRN-28-10	DN-40	RN-50	WRN-16-8	DN-40	RN-50
fp32	26.07%	20.35%	26.03%	1.89%	2.00%	1.80%	23.64%
hbfp8_16	25.12%	20.78%	26.27%	1.98%	1.98%	1.79%	23.88%
hbfp12_16	25.10%	20.78%	25.82%	1.96%	1.94%	1.85%	23.58%

Table 3.4 – Perplexity of language modeling models. *hbfpX_Y* indicates an experiment with X-bit mantissas and Y-bit weight storage. All dot product operations are performed in X-bit arithmetic.

	LSTM-PTB	BERT-Tiny
fp32	61.3	60.9
hbfp8_16	61.9	62.9
hbfp12_16	61.3	60.9

memory bandwidth requirements. Using 8-bit mantissas reduces the memory bandwidth requirements of the forward and backward passes by up to 4× compared to fp32. HBFP stores activations in floating-point format. While doing so may increase bandwidth requirements, we observe that these activations can be stored in narrow floating-point representations or even in summarized formats (e.g., for ReLU, only a single bit per value needs to be saved for the backward pass). Furthermore, activations account for a small fraction of the memory traffic when training DNNs. While activation traffic is dwarfed by weight traffic in fully connected layers, in convolutional layers, the computation-to-communication ratio is so high that the memory traffic incurred by activations is not a significant throughput factor.

3.5.2 HBFP vs. fp32

Table 3.3 reports the validation error for all the image classification models and Table 3.4 reports the validation perplexity of the language modeling models. Using hbfp12 in all models leads to validation errors and perplexity values similar to the fp32. Similarly, for all image classification and LSTM models, we observed similar validation errors and perplexity values for hbfp8 and fp32. For BERT-Tiny, however, using hbfp8 leads to a small degradation in perplexity compared to fp32. We attribute that to the attention layers which feature several small matrix multiplications followed by the grouping of results from different attention heads. Adjusting the attention layer design for HBFP might eliminate this perplexity gap.

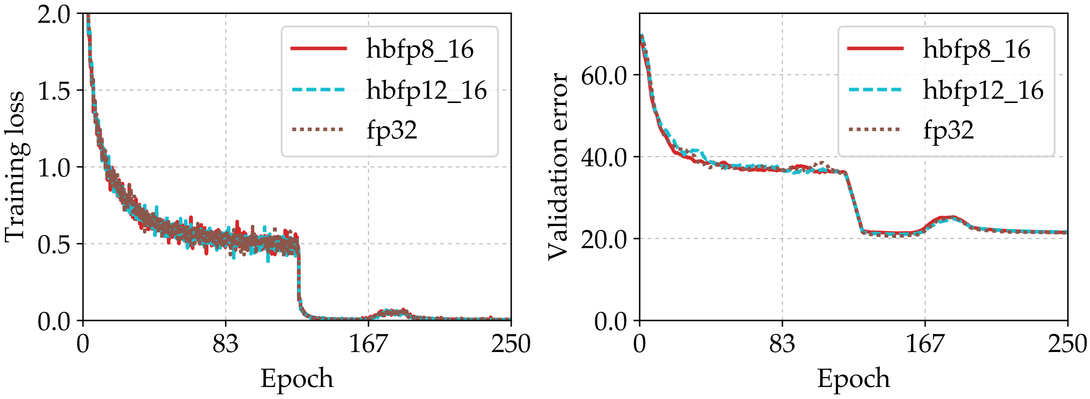
In addition, Figure 3.4 illustrates the training process for three of the evaluated models: a WideResNet28-10 trained with CIFAR-100, a ResNet-50 trained with ImageNet, and an LSTM trained with PTB. HBFP matches the performance of fp32 in all the models and datasets tested. In all models tested, HBFP led to convergent rates similar to fp32, using the same hyper-parameters.

Additionally, in the ImageNet experiments, we observed that the loss evolution of HBFP was noisier than fp32. We attribute the noisier loss evolution due to quantization issues in weights. Loss values are calculated using the most significant bits of the weight values, with weight variations leading to more significant changes in the loss. Fortunately, the backpropagation process quickly readjusts the weights, and the loss variations do not affect the convergence rate. In the same experiment, the validation error evolution is different at the beginning of the training process but converge later, as we adjust the learning rate. We hypothesize that this is due to differences in the weight initialization, and the differences disappear as training evolves.

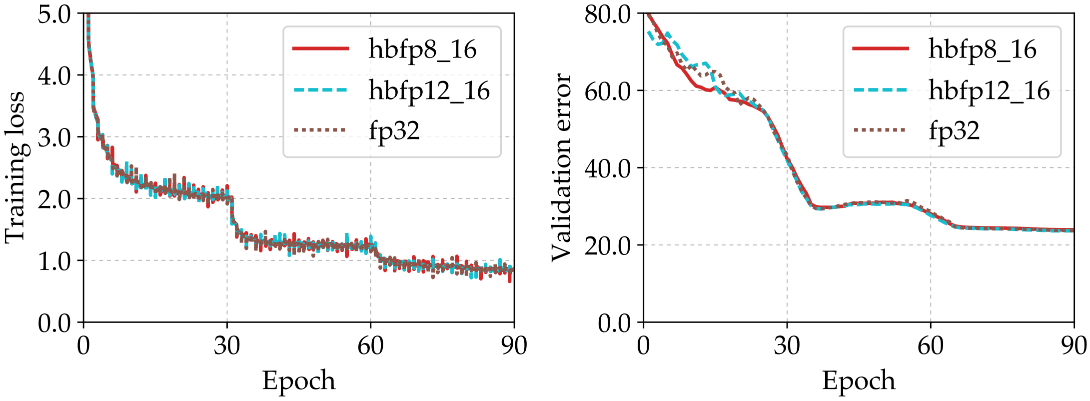
We conclude that HBFP is indeed a drop-in replacement for fp32 for a wide set of tasks, leading to models that are more compact and enabling HW accelerators that use fixed-point arithmetic for most of the DNNs computations.

3.6 Chapter Summary

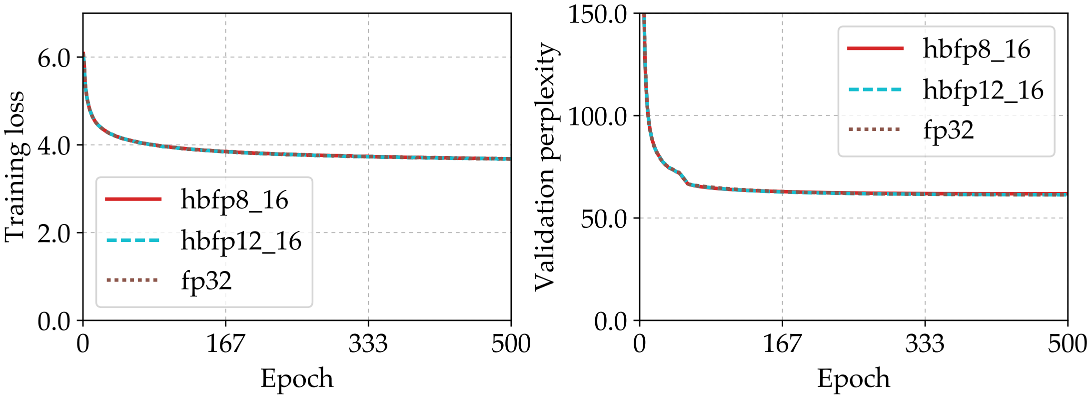
DNNs have become ubiquitous in datacenter settings, forcing operators to adopt specialized hardware to execute and train them. However, DNN training still depends on floating-point representations for convergence, severely limiting the efficiency of accelerators. In this chapter, we propose HBFP, a hybrid BFP-FP number representation for DNN training. We show that the HBFP leads to efficient hardware, with the bulk of the silicon real-estate spent on efficient fixed-point logic. Finally, we evaluate HBFP and show that, for all models evaluated, BFP-FP



(a) WideResNet-28-10 trained on CIFAR-100 for 250 epochs.



(b) ResNet-50 trained on ImageNet for 90 epochs.



(c) LSTM trained on PTB for 500 epochs.

Figure 3.4 – Comparison between HBFP and FP32. *hbfpX_Y* indicates an experiment with with X-bit mantissas and Y-bit weight storage. All dot product operations are performed with X-bit arithmetic.

training matches their fp32 counterparts while resulting in 4× more compact models. HBFP also uses area- and power-efficient ALUs, increasing the throughput of accelerators. Higher throughput leads to faster and more energy-efficient DNN training/inference and model compression leads to lower bandwidth requirements for off-chip memory, lower capacity requirements for on-chip memory, and lower communication bandwidth requirements for distributed training.

4 Modeling Accelerator Performance

Inference accelerators must maximize throughput to maximize the opportunities for piggybacking training services. Unfortunately, maximizing throughput is often at odds with the inference tight latency goals. On the one hand, techniques like batching increase the data reuse in DNN inference workloads, leading to higher throughput. On the other hand, batching harms inference latency because it delays individual requests. Traditional DNN accelerators take opposing approaches with regard to batching. Brainwave [18] does not use batching at all, to minimize inference latency while Google's TPU [32] uses large batches to maximize throughput.

We argue that selecting designs in the edges of the latency versus throughput trade-off is a misguided goal because of the non-linear relationship between latency and throughput. Furthermore, the choice of numeric encoding used in the ALU arrays also affects the non-linearity of the relationship between latency and throughput. As we show in this chapter, with HBFP, optimizing for either metric leads to diminishing returns and sub-optimal accelerators, allowing accelerator designers to trade-off small increases in latency for substantial throughput gains. Floating-point representations, however, exhibit a more linear relationship between latency and throughput, reducing the opportunities for large performance gains.

In this chapter, we first identify the key parameters impacting the accelerator's design space, then present a preliminary design space exploration revealing a Pareto-optimal frontier of throughput against latency using analytical models for area, power and performance. We show that the Pareto-optimal frontier exists for both HBFP- and bfloat16-based accelerators. Once we narrow down the design space to a few Pareto-optimal points, we evaluate them using cycle-accurate simulation and synthesis in Chapter 6.

4.1 Optimizing Latency Constrained DNN Accelerators

We optimize the throughput of latency constrained inference accelerators to maximize the opportunities to piggyback training. Maximizing throughput in inference accelerators requires minimizing the fraction of power that is spent on data movement. Unfortunately, reducing data movement is tightly related to inference. Additionally, the numeric encoding used in ALU arrays also dramatically affects the breakdown between power spent on ALUs and in data movement. In this section, we explain how data reuse and numeric encoding affect both accelerator throughput and latency.

To explore the relationship between inference latency and throughput, we use the ALU array shown in Figure 4.1, which consists of a row of m systolic arrays, each with $n \times n$ w-wide processing elements (PEs) connected to the weight and activation buffers. The w dimension determines the length of the dot products performed by the ALU array, affecting both throughput and data movement equally. The m and n dimensions are used to exploit the varying degree of data reuse present in DNN workloads. Activations are reused both across and within systolic arrays (i.e., across both m and n dimensions), but weights are only reused within systolic arrays (i.e., across the n dimension). While activation reuse is abundant in most DNNs, weight reuse is non-existent in models that dominate datacenter DNN workloads—such as RNNs and MLPs [27, 32]—necessitating designs that batch requests to enforce weight reuse across the n dimension.

Figure 4.2 illustrates this non-linear relationship that appear between latency and throughput when n is varied. The accelerator on the top with the highest latency constraints incorporates

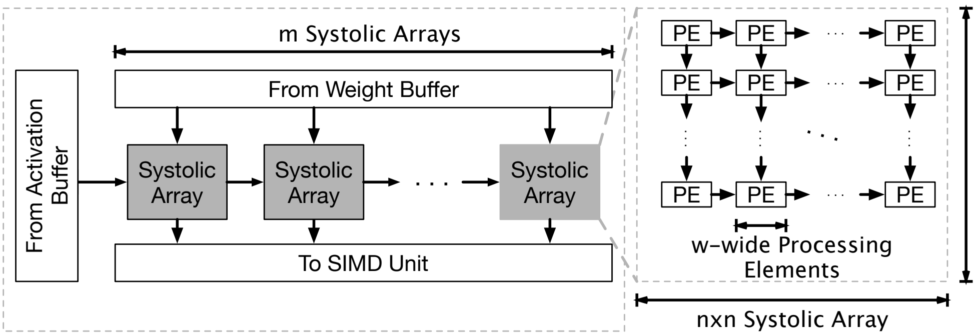


Figure 4.1 – Equinox matrix multiplication unit.

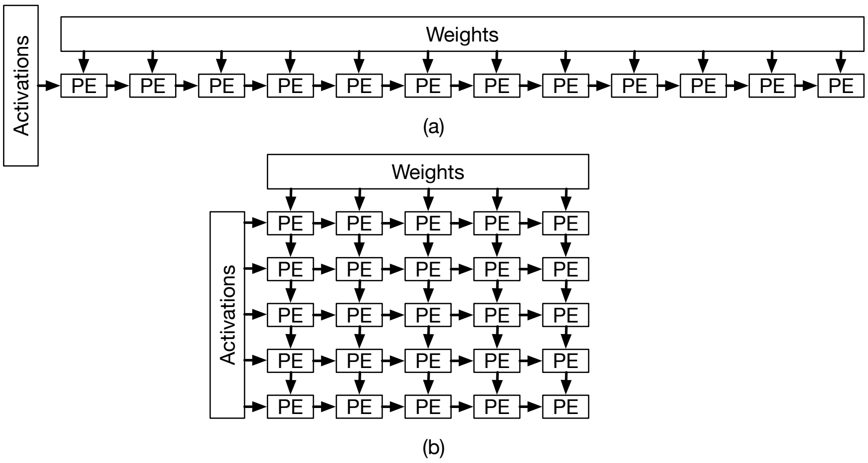


Figure 4.2 – Comparison between two ALU arrays both consuming roughly the same power. The array on the bottom trades off latency to improve throughput by requiring batching.

only a single row of ALUs (i.e., $n == 1$, large m, and large w) without batching, but requires a wide path to on-chip memory. In contrast, the accelerator on the bottom incorporates multiple rows of narrower ALUs (i.e., $n > 1$, smaller m, and smaller w) with batching to trade off a bit of increase in latency (the height of the array) for a much larger increase in the total number of ALUs and throughput. The latter means that increasing n and relaxing the constraints on tail latency a bit can lead to large increases in throughput and, consequently, opportunities to piggyback training.

Another important factor in maximizing inference throughput is the numeric encoding used in accelerators. Numeric representations that are data movement bound—i.e., lead to ALUs which spend more power accessing operands than performing operations—benefit from large n dimensions more, providing better opportunities for piggybacking. We observe that fixed-point-based representations minimize the power spent on ALUs and, as such, are data movement bound. Floating-point representations, in contrast, are computation bound.

Finally, much like GPUs, DNN accelerators exhibit high degrees of power density that, if not mitigated, results in dark silicon [24]. To ameliorate the power density, designers scale down the accelerator's operating voltage together with frequency to provision power for larger ALU arrays. Therefore, frequency (coupled with voltage) is also an important parameter dictating the overall attainable throughput.

4.2 Analytical Models

Complexity and runtime requirements make it impractical to rely on cycle-accurate simulation for a large-scale design space exploration. Instead, we use first-order analytical models of dominant accelerator components, relating the effects of physical constraints on performance. With latency and throughput as performance metrics, our objective is to jointly optimize the accelerator's dimensions and frequency to find the best performing designs under power and area constraints.

Besides the ALU arrays and their associated buffers which naturally account for much of an accelerator's area and power, the only remaining dominant component in the first-order models is the DRAM interface [6, 32][1] For the DRAM interface, we reserve enough power to accommodate an HBM stack with $1TB/s$ bandwidth (the largest HBM bandwidth commercially available), and enough area to accommodate the HBM interface.

Area modeling. We model a $300mm^2$ die, which is in line with reported die areas of DNN accelerators [24, 51, 32]; candidate designs that exceed this die area constraint are not considered. To estimate the required aggregate ALU area, we first synthesize a set of matrix multiply units (MMUs) with various dimensions using the Synopsys Design Compiler and TSMC 28nm

---

[1] Our cycle-accurate models in Chapter 6 capture all accelerator components.

technology (with the TCBN28HPMBWP35 Core library and Vdd of 0.9V). We then calculate an ALU's average area, a_{alu}, for each of bfloat16 and hbfp8 and scale it to match the MMU dimensions of the modeled design.

To estimate the required SRAM area, A_{sram}, we scale the per-byte area reported by CACTI 6.5 [45] to the accelerator's aggregate SRAM capacity. Because CACTI does not support 28nm, we scale down the area values from 32nm using the methodology found in [17]. We assume a capacity of $75MB$, which is large enough to accommodate the majority of models used in datacenter services [32, 18]. To model the DRAM interface area, A_{dram}, we extract estimates from [62]. Equation 4.1 calculates the total area of the accelerator.

$$A = mn^2 w a_{alu} + A_{sram} + A_{dram} \tag{4.1}$$

Power modeling. We use a first-order model to calculate the accelerator's total power by summing the dynamic and static power of the ALUs, the SRAM buffers, and the DRAM interface. We assume a $75W$ power envelope, which is in line with reported power budgets of DNN accelerators [32], eliminating all candidate designs that exceed this power constraint. We model static power, P_{static}, only in the SRAM buffers because its contribution from ALUs is negligible.

To model dynamic power, we first estimate the energy consumption in the ALUs and buffers at a fixed frequency point, and then adjust the base energy values according to the design's frequency, f, ranging from $532MHz$ to $2.4GHz$ using values extracted from [53]. An ALU's average energy consumption, e_{alu}, is derived from the same area synthesis methodology (above) and is scaled to match the modeled MMU dimensions. Similarly, we scale the average per-byte energy consumption in buffers, e_{sram}, for various block sizes reported by CACTI to match the accelerator's dimensions. Finally, we extract power estimates from [62] for DRAM accesses, P_{dram}. Equation 4.2 calculates the accelerator's total power consumption.

$$P = f \times (mn^2 w e_{alu} + e_{sram} \times (wn + mwn + mn)) + P_{dram} + P_{static} \tag{4.2}$$

Performance modeling. Equation 4.3 estimates the maximum inference throughput as a function of the accelerator's operating frequency and dimensions. Each ALU performs two operations (i.e., multiply and accumulate) per cycle. To estimate latency, we measure service time while processing a batch of n inference requests of an LSTM model with 2048 hidden units and 25 steps from DeepBench [47].

$$T = 2mn^2 wf \tag{4.3}$$

4.3 Exploring the Design Space

Figures 4.3 and 4.4 show the design space. In each figure, the first five plots show a heatmap with a particular design metric, as a function of n (x-axis) and frequency-voltage point, referred by the frequency value for simplicity (y-axis). A colored point in the plot indicates a valid design. Points with a dark color indicate a low value for the depicted metric, while points with a lighter color indicate a higher value for the metric. In each plot, a legend in the right shows the value range for the depicted metric. We adjust each value range to ease visualization. Regions in the plot without color represent points where it was impossible to fit a design under area or power constraints. We present data movement power percentage, power, area, average latency, and throughput for the design space.

The first interesting trend is the discontinuities in the plots. For instance in Figure 4.3a, we observe sudden decreases in power values (i.e., bright points with dark points at their right) at certain values of n. These discontinuities are caused by the discrete parameters used in the model. For models close to the maximum power constraint, increasing the value of n from x to $x+1$ requires smaller m and w. These designs end up not utilizing all the power envelope, leading to the discontinuity. This pattern is observed across the design space, becoming more frequent when n is smaller.

We also observe that bfloat16 designs violate power and area constraints at a much smaller n than hbfp8. Figures 4.1 show that the maximum n for bfloat16 is around 220 while hbfp8 has valid designs with n as large as 400. This difference is due to the high area and power consumption of bfloat16. The design space limitations of bfloat16 ultimately limit the maximum throughput and minimum latency of such designs.

Power is the biggest constraint for designs with higher frequencies, due to the increased power density of these designs. In Figures 4.3a and 4.4a, designs with higher frequencies are limited to low values of n because, as we increase n, we quickly reach the point where no design fits the power constraints, leading to invalid designs. As the frequency decreases, larger designs, with high values of n, fit under the power constraints, due to their lower power density.

In Figures 4.3b and 4.4b we observe that higher frequencies leads to designs with lower areas. As explained before, these designs are power limited. The figure also shows that, with most frequencies, even the larger designs have a low area, indicating a poor usage of the area envelope. As we reduce the frequency, we also reduce the power density of the ALU array. Designs with lower power density utilize more of the area envelope, up until we reach the point of optimality where both the area and the power envelope are well utilized. Both bfloat16 and HBFP have balanced area and power consumption for designs with the lowest frequencies.

Figures 4.3c and 4.4c show the effect of n and frequency over the power spent on data movement. Because n dictates the batch size, it affects the amount of weight reuse. As such, increasing n reduces the amount of power spent in data movement, allowing for higher overall throughput, as shown in Figures 4.3d and 4.4d. These designs are data movement bound.

However, bfloat16 and hbfp8 differ on how data movement bound they are. Figures 4.3c and 4.4c show the key difference between bfloat16 and HBFP. The plots show that the maximum percentage of power spent on data movement for bfloat16 is around 75%. These values are usually achieved on designs that do not use the power envelope well, exhibiting low power consumption (shown in Figure 4.3a). More interestingly, designs with small n spend a smaller fraction of their power on data movement than hbfp8 designs under the same circumstances. In the hbfp8 design space, designs with small n spend up to 90% of their power on data movement (left part of Figure 4.4c, with data movement power reducing as n increases. These differences are due to bfloat16 ALUs spending almost an order of magnitude more area and power than hbfp8, but moving only 2× more data. As such, bfloat16 accelerators are less data movement bound than hbfp8.

Figures 4.3d and 4.4d show the maximum throughput for bfloat16 and hbfp8 designs. First, we observe that, as expected, throughput increases as n increases for both representations. Unfortunately, high values of n require large batch sizes to utilize the systolic arrays fully. We also observe that there are designs with near-optimal throughput, even with smaller values of n. These designs can accommodate high compute power with smaller n by increasing the value of m and w. We also observe that the maximum throughput of bfloat16 is significantly lower than hbfp8.

The positive latency effects of designs with lower n are shown in Figures 4.3e and 4.4e. Designs with lower n but high throughput achieve lower latencies, with minimal latency achieved at $n == 1$ for both bfloat16 and hbfp8. As n increases, latency suffers, with even designs with optimal throughput achieving high latency.

Finally, figures Figure 4.3f and 4.4f depict the latency and throughput of the modeled accelerators in the design space for bfloat16 and hbfp8. The designs on the Pareto-optimal frontier appear as large blue dots and the rest as small dots. Tables 4.1 and 4.2 present the batching degree, frequency, latency and throughput for design points on the Pareto frontier for four design configurations based on latency constraints. We use these configurations in Chapter 6 to evaluate Equinox's performance.

The Pareto frontier for hbfp8 follows a *sub-linear* relationship between latency and throughput for latencies below $50\mu s$ with over a 5× increase in throughput against a gradual increase in latency reaching the knee past $350 TOp/s$. Designs before the knee spend most of their power in on-chip buffers and data movement. This large fraction of on-chip buffer power is due to the skew in the array dimension with high-bandwidth connectivity to memory (Figure 4.2(a)). Reducing this array dimension shifts a large fraction of power for data movement from memory to ALUs in the second dimension with point-to-point links (Figure 4.2(b)) increasing

Table 4.1 – Maximum throughput for bfloat16 designs under various latency constraints.

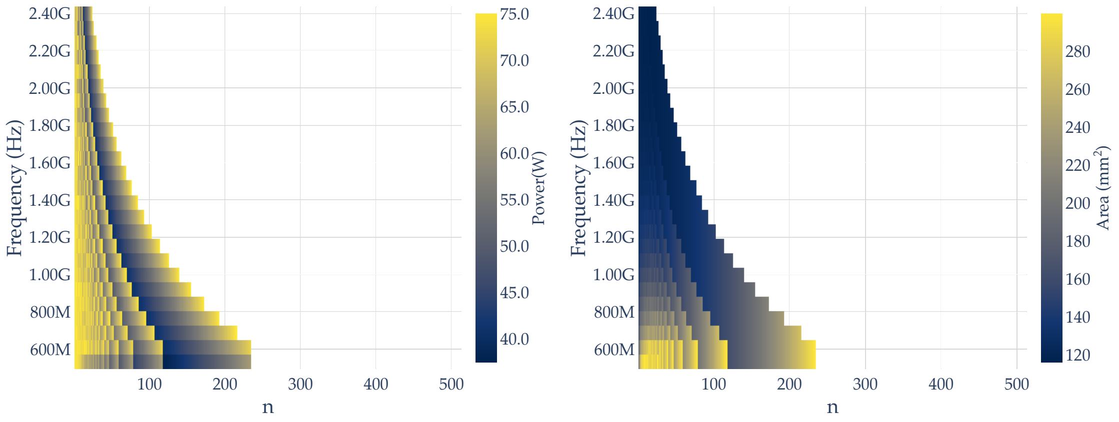
Latency constraint	Batch size	Frequency (MHz)	Average Latency (s)	Throughput (TOp/s)
Min. latency	1	532	$37.3\mu s$	23.9
Latency < $500\mu s$	29	610	$386\mu s$	63.3
Max. Throughput	39	610	$510\mu s$	66.7

throughput linearly with batch size with tiny increases in latency.

Once the curve reaches the knee, the fraction of power dedicated to data movement is low, and as such increases in batch size shift ALU's from one dimension (rows) to other (columns) resulting in little improvement in throughput while greatly hurting latency. Because the designs at the knee are optimal in offering throughput at constrained latencies, these designs are great candidates for Equinox to exploit idle cycles for training services.

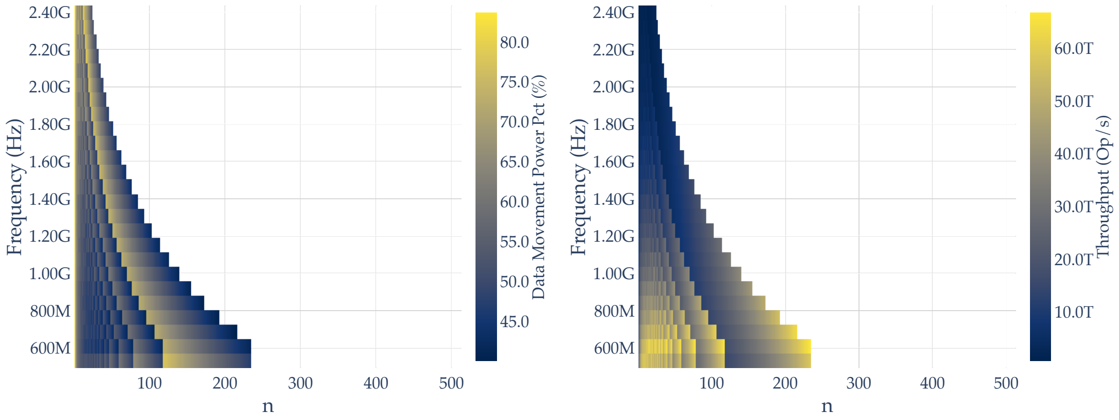
In contrast to hbfp8, bfloat16 exhibits high sensitivity to latency from the start with a linear rather than a sub-linear relationship between latency and throughput, reaching the knee almost immediately. Because bfloat16 provisions an order of magnitude more power in floating-point ALUs, there is little power to be shifted from on-chip memory to increase batching and throughput. Therefore, bfloat16 designs can not support batching for latencies below $50\mu s$ and with higher batching degrees mostly shift ALU power from one dimension to another with no increase in throughput.

Table 4.2 also shows that while today's custom accelerators either select designs without batching ($n = 1$) [18] or those with high batching degrees ($n >> 100$) [32], many designs with moderate batching degrees ($n < 100$) offer near-optimal throughput at a sub-millisecond latency, achieving the best of both worlds. Finally, we observe that optimal designs have relatively low frequencies, showing that DNN designs are mostly power-limited. Designs with the highest latency constraints favor the lowest frequency ($532 MHz$) because they spend most of their power on data movement.



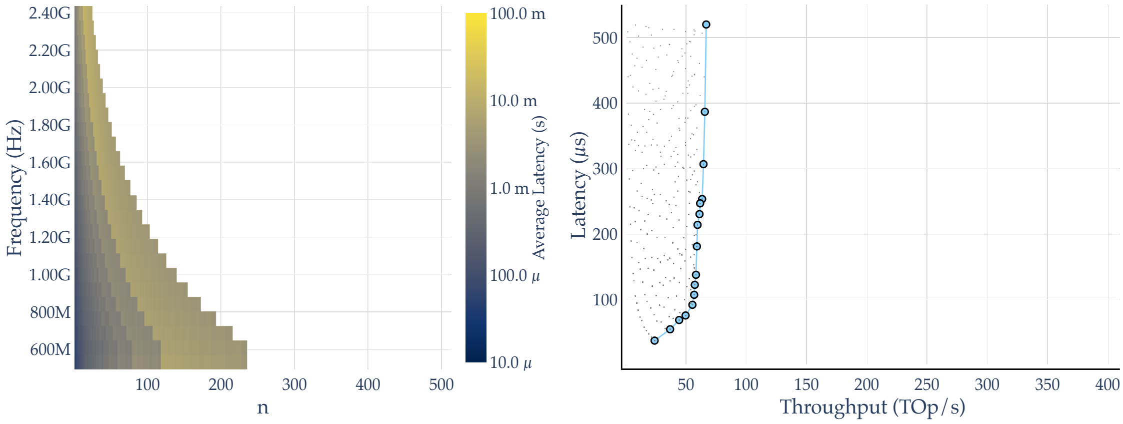
(a) bfloat16 power (W) design space.

(b) bfloat16 area (mm^2) design space.



(c) bfloat16 percentage of power spent on data movement (%) design space.

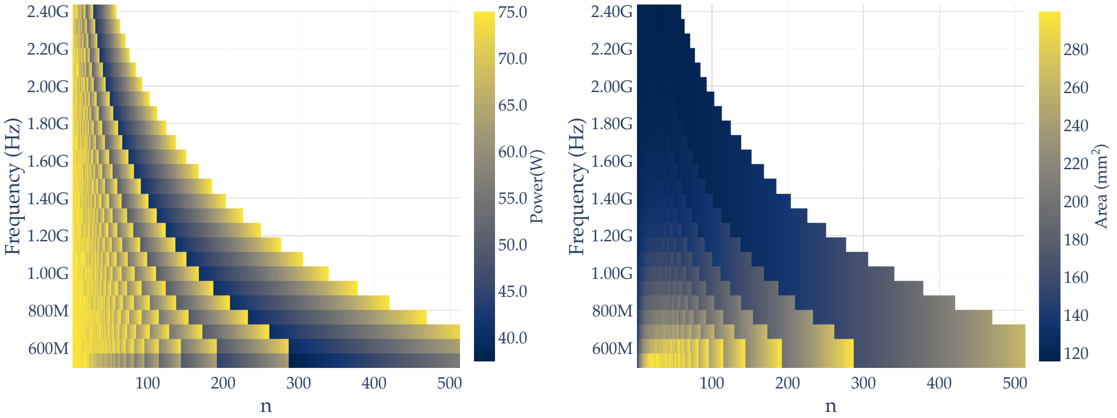
(d) bfloat16 throughput (Op/s) design space.



(e) bfloat16 average latency (s) design space.

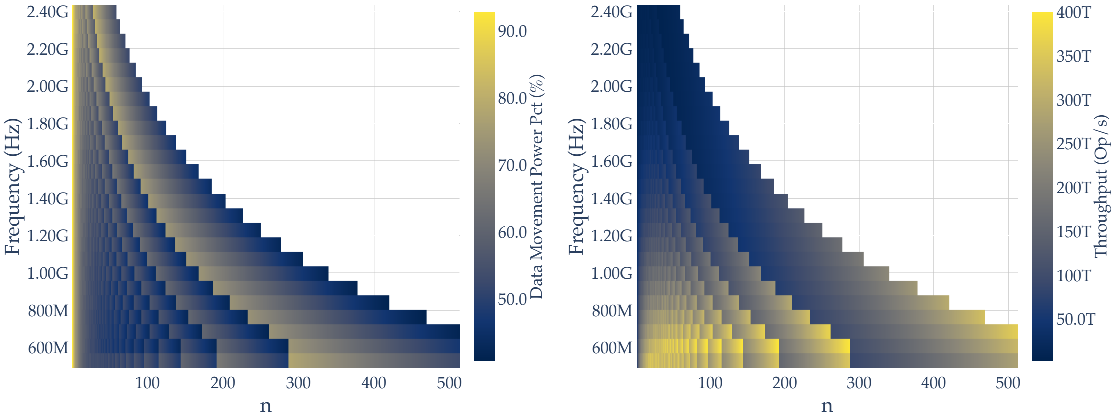
(f) bfloat16 Designs with optimal latency and throughput. The plot shows average latency (s) vs. throughput (Op/s).

Figure 4.3 – bfloat16 accelerator design space.



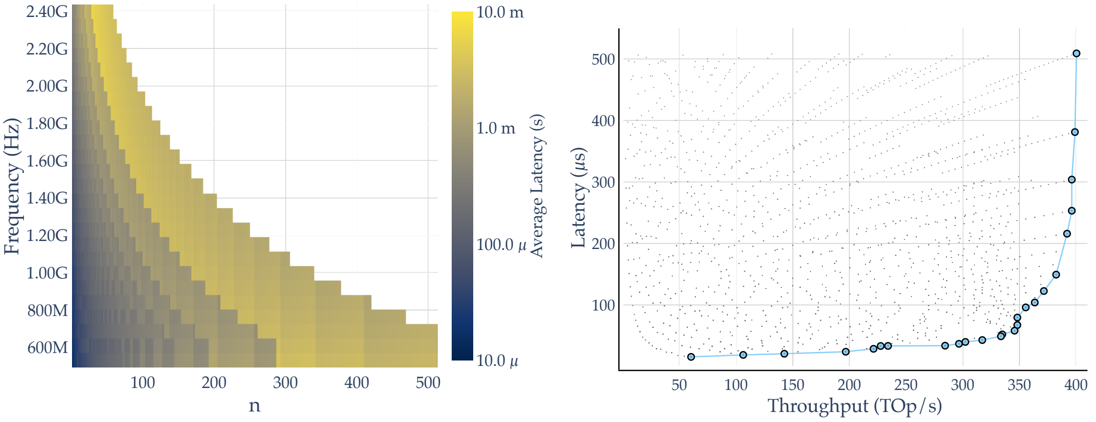
(a) HBFP power (W) design space.

(b) HBFP area (mm^2) design space.



(c) HBFP percentage of power spent on data movement (%) design space.

(d) HBFP throughput (Op/s) design space.



(e) HBFP average latency (s) design space.

(f) HBFP designs with optimal latency and throughput. The plot shows average latency (s) vs. throughput (Op/s).

Figure 4.4 – HBFP accelerator design space.

Table 4.2 – Maximum throughput for HBFP designs under various latency constraints.

Latency constraint	Batch size	Frequency (MHz)	Average Latency (s)	Throughput (TOp/s)
Min. latency	1	532	15.6μs	60.2
Latency < 50μs	16	532	49.2μs	333
Latency < 500μs	143	610	381μs	390
Max. Throughput	191	610	509μs	400

4.4 Chapter Conclusion

In this chapter, we showed that the use of batching in inference accelerators leads to a non-linear relationship between throughput and latency for ALU arrays which employ hbfp8. Moreover, we show that this non-linear relationship leads to a sweet spot in the design space enabling accelerator designers to achieve near-optimal throughput with sub-millisecond service times. Designs in the sweet spot expand the opportunities for training services to execute when the inference load is low.

5 The Equinox Family of Accelerators

Inference accelerators are deployed in large fabrics to serve online services under tight latency constraints. Unfortunately, online services face variable service demand, leading to accelerators with a low average load of around 30% [3]. Because inference accelerators cannot execute other workloads, ALUs go idle when the load is low. While training services can reclaim these idle cycles, doing so requires a numeric representation that guarantees training convergence. While floating-point satisfies this requirement, it leads to lower inference throughput, reducing the number of idle cycles available for training. We identify HBFP as a promising numeric representation for an inference accelerator that exposes idle cycles for training. Chapter 3 shows that HBFP satisfies the convergence requirement for training and Chapter 4 shows that HBFP maximizes inference throughput, when latency requirements are relaxed enough.

However, even if idle cycles are plenty, adding training may also lead to unacceptable latency for inference services. Some inference accelerators [23, 22, 24] are capable of performing both inference and training, but are not designed to share resources across multiple services. As such, resource sharing has to be managed by external software controllers, in a coarse-grained way, leading to long scheduling delays. We observe that introducing contexts for inference and training services in accelerators, as well as scheduling mechanisms to manage both contexts, can be done with a low area and power overhead, enabling fine-grained resource scheduling. A simple priority scheduling mechanism can prioritize inference requests' instructions when the load is high, bounding the maximum latency observed by inference services.

Using priority scheduling and efficient encoding, an inference accelerator — even not being optimized for training — can expose enough idle cycles to training services to saturate off-chip memory bandwidth. In this chapter, we introduce Equinox, a family of ColTraIn accelerators — inference accelerators designed to piggyback training to reclaim inference idle cycles. Equinox accelerators feature a uniform encoding datapath and a priority scheduler that maintains service-level latency guarantees for inference requests while interleaving training requests. In chapter 6, we evaluate Equinox showing that optimized instances of Equinox achieve a training throughput that is close to the throughput of an optimized training accelerator without affecting inference throughput.

5.1 Piggybacking on Inference

Piggybacking training services on inference accelerators faces three essential requirements. First, inference accelerators must expose enough idle cycles for training; second, the inference accelerator must support the execution of multiple contexts; and third, the accelerator must maintain the latency guarantees of traditional inference accelerators, even in the presence of training services. We now focus on these three key challenges.

As shown in chapter 4, the ALU arrays' dimensions and numeric encoding are a crucial design point in maximizing inference throughput when under latency constraints. High throughput accelerators provide more opportunities for Equinox accelerators to expose idle cycles for training services. Accelerators which employ ALU arrays with efficient numeric representations — such as HBFP — maximize the opportunities for an accelerator to expose enough idle cycles for training services to saturate off-chip memory bandwidth, maximizing training throughput. HBFP, however, requires modifications to the accelerator datapath, to accommodate both BFP matrix multiplications and bfloat16 activations.

Even when inference throughput is maximized, accelerators that piggyback training must also host inference and training services simultaneously, which may lead to resource contention. As in multithreaded CPUs, the accelerator requires space sharing in the buffers and time sharing in the execution units for the two services. Because training relies on fetching data from off-chip memory, due to its large memory footprint (e.g., a few GBs [74]) and long-distance dependencies in SGD's backpropagation, on-chip buffers are used only to stage operands right before computation. As such, training's staging buffers require only a small fraction (i.e., less than 2%) of the on-chip buffer space.

Similarly, as in multithreaded CPUs, resource contention in the execution units could impact inference's service-level latency constraints. DRAM latency is orders of magnitude longer than ALU array latencies. Therefore it is relatively easy to schedule idle slots in the array for training. To minimize the impact on inference service times, a custom accelerator can incorporate a priority scheduler that monitors incoming requests and schedules DRAM accesses and array idle slots with priority given to inference requests. We show that scheduling requests in software may negatively impact queuing delays due to the longer turnaround time in scheduling and, instead, present a hardware priority scheduler implemented with minimal logic.

One final issue that arises from piggybacking training is managing inference batching in a way that does not hurt the throughput observed by training. When inference requests are batched, batch formation times — the delay between the moment a batch's first request arrives and the moment a batch starts executing — can be arbitrarily large when load is low, due to long request inter-arrival times. To mitigate this issue, we employ adaptive batching, a technique that issues batches that are smaller than the batch size specified by the service, when batch formation takes too much time. Adaptive batching leads to reduced ALU utilization. The ALU cycles wasted by adaptive batching cannot be reclaimed by training services, and as such, must be minimized.

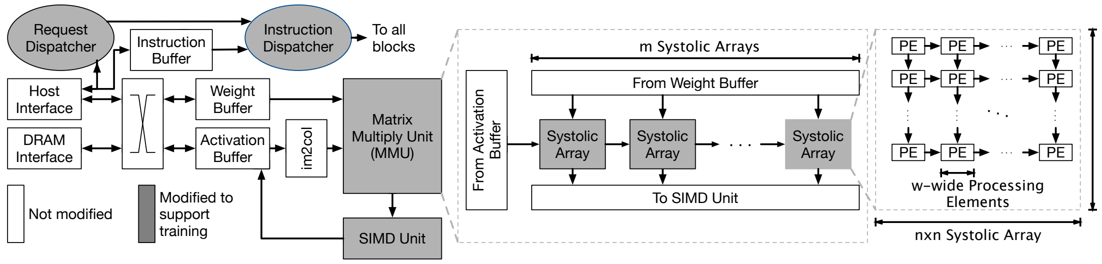


Figure 5.1 – Block diagram of Equinox.

5.2 Accelerator Design

In this section, we present the design of Equinox accelerators. We first describe a baseline inference accelerator and then present the enhancements to the baseline design to support training.

5.2.1 Baseline Inference Accelerator

Figure 5.1 depicts the anatomy of our baseline inference accelerator (e.g., TPU [32]). Much like other discrete accelerators, a host interface connects the accelerator to the host and network/storage peripherals through a standard I/O fabric (e.g., PCIe). The host interface enables both service installation and client request/response for installed services. Service installation consists of loading the code and model and launching the accelerator, after which the accelerator operates autonomously.

The service specifies the model through a custom instruction set architecture (ISA) shown in Table 5.1. The accelerator implements all instructions necessary for popular inference services (e.g., RNN, MLP, and CNN), including matrix-vector multiplication, convolution, vector-vector operations, activation, batch normalization, and pooling. The ISA also includes instructions to move data among the DRAM, the network buffers, and the accelerator's datapath.

Instructions operate on 2-D *tiles*, whose size is a function of the dimensions of the matrix multiply unit in the datapath. Figure 5.2 shows how a matrix multiplication is divided into tiles. In the Figure, the first $n * w$ rows of the activation matrix and the first $n * w$ columns of the

Table 5.1 – Accelerator ISA.

Instruction	Operands	Description
BARRIER	–	Pause instruction issue until this instruction retires
DATA_TRANSFER	destInterface, destTensorId, srcTensorId, srcTensorId	Transfer a tile between on-chip buffers and external interfaces
MM_CONV	convDims, weightTensorId, actTensorId, outTensorId	Convolution
MM_MULT	weightTensorId, actTensorId, outTensorId	Matrix multiplication
VECTOR_OP	inTensor1ID, inTensor2ID, outTensorID	Vector-vector operation
POOL	poolingDims, inTensorID, outTensorID	Pooling operation

weight matrix are divided into x tiles with $n * w$ side each. Each instruction addresses a single activation tile and m weight tiles, as shown in the Figure, producing m tiles. To produce an output tile, the compiler generates x instructions, each of which processes an entire activation tile row and an m weight tile rows. The compiler also generates x instructions to add the intermediate output tiles, producing the final tiles.

The accelerator's datapath (Figure 5.1) is composed of a matrix multiply unit (MMU), a SIMD unit for vector-vector operations, an im2col unit, activation and weight buffers, and a DRAM

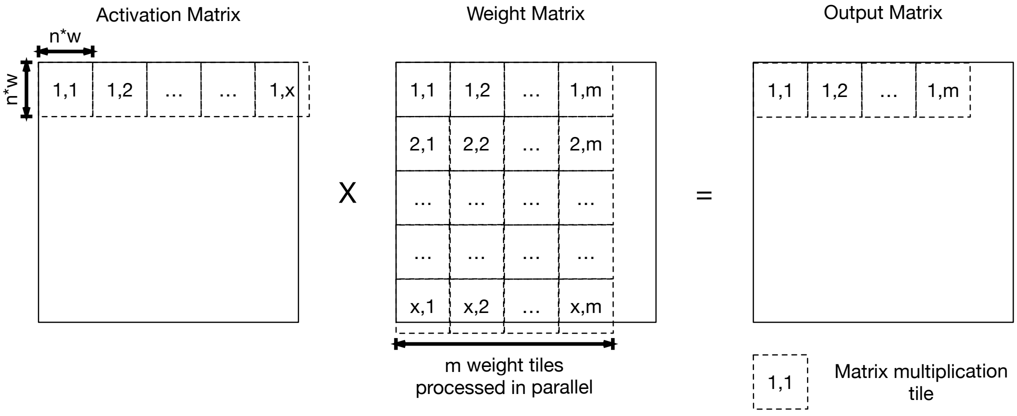


Figure 5.2 – Division of a matrix multiplication into tiles.

interface. The im2col unit lowers convolutions to matrix multiplication. The MMU consists of a row of m systolic arrays, each with $n \times n$ w-wide processing elements (PEs) connected to the weight and activation buffers. The PEs operate on 2-D tiles with sides of $n \times w$ elements. We show in chapter 4 how the three parameters, n, w, and m, collectively, enable systolic arrays to balance their latency and throughput. The SIMD unit performs vector-vector operations similar to the activation unit in TPU [32] and includes a register file to store intermediate vectors and accumulated values. It fetches operands from either the register file or the MMU's output and writes its results into the activation buffer.

The activation and weight buffers are organized into banks. The weight buffers have direct connectivity between each bank and a corresponding systolic array. The activation buffer banks have broadcast connectivity to all arrays (Figure 5.1), implemented through a ring. The activation buffer banks each have a read port facing the systolic arrays, a read-write port facing the DRAM and host interfaces, and a write port facing the SIMD unit. The weight buffer banks each have a read port facing the systolic arrays and a read-write port shared by the DRAM and host interfaces.

Figure 5.3 depicts the anatomy of the accelerator's front-end. Upon service installation, the request dispatcher copies the weights and instructions into their respective buffers. Upon service launch, the request dispatcher monitors the request queue and forwards arriving requests to the instruction dispatcher. Much like TPU, our baseline accelerator supports batching to reduce data movement. The request dispatcher gathers arriving requests in a batch formation buffer and notifies the instruction dispatcher upon a full batch formation.

To reduce the impact of batch formation on latency, we implement adaptive batching. The request controller issues incomplete batches when batch formation time exceeds a threshold (defined at installation time) by padding the input arrays [8] with null requests whose results are discarded. We compare adaptive batching with a static batching policy in Chapter 6, and show how adaptive batching minimizes batch formation's impact on latency when the load is low at the cost of wasting execution resources.

The instruction dispatcher, shown on the bottom part of Figure 5.3, features a controller which keeps track of instruction issue and completion. The controller generates addresses for the instruction buffer, which forwards instructions to the decoder unit. The latter generates control signals for the datapath. Arithmetic instructions are decoded into signals issued to the execution units, and data movement instructions are decoded into control signals issued to the DRAM and host interfaces as well as other blocks in the datapath. The dispatcher has an instruction completion unit to keep track of responses received from the datapath.

5.2.2 Enhancements for Training

We now describe the enhancements that Equinox introduces to the base inference accelerator. These enhancements are not meant to create a full-blown custom training accelerator; in contrast, they are only there to piggyback training on an inference accelerator. The boxes shaded in gray in Figures 5.1 and 5.3 indicate the mechanisms that are enhanced to support training.

The first enhancement is in the accelerator's ISA and datapath. We overload the VECTOR_OP instruction in the SIMD units to add support for derivative and loss calculations required by training. We also add arithmetic support for training to the datapath and evaluate both a bfloat16 [23, 22] version as a state-of-art reference for custom accelerators and an hbfp8 [16] version which offers dramatically higher density. Because the area and power of the SIMD unit are relatively small compared to the rest of the datapath and the unit's density is not critical, we use bfloat16 for the SIMD unit in both versions.

The MMU and the buffers, however, are fundamentally different across the two datapath versions. The bfloat16 version uses the 16-bit numeric encoding in all buffers and systolic array multipliers, and single-precision floating point for the accumulators, which is common in DNN accelerators [23, 22, 50] to maintain high accuracy. The hbfp8 version, in contrast, uses block floating point in the systolic arrays and buffers, where each operand consists of a block of 8-bit mantissas sharing a single 8-bit exponent. As such, all buffers are modified to

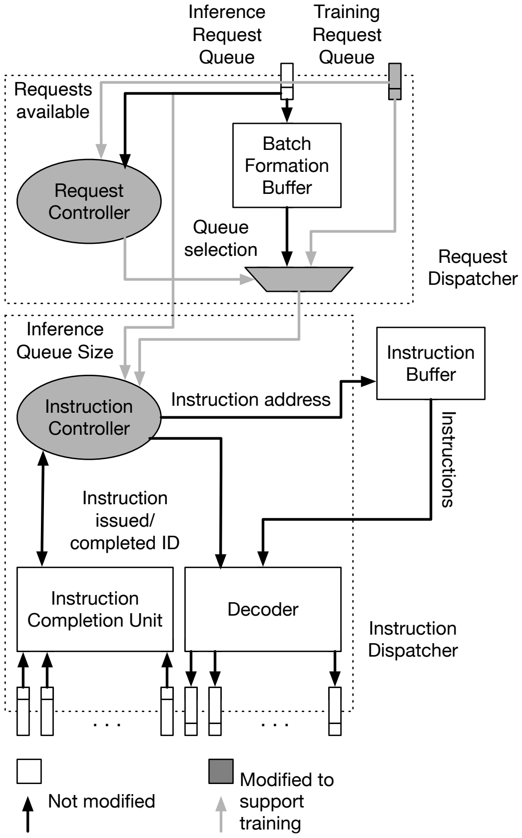


Figure 5.3 – Equinox two-level controller.

store, read, and write a block as an operand. Figure 5.4 shows the modifications we made to the datapath to support BFP matrix multiplications, with the modifications shaded in grey.

In block floating point, matrix multiplication can be implemented as a fixed-point multiplication of the tiles and addition of the two exponents. To implement this in the systolic array, we use 8-bit multipliers and 24-bit accumulators, both operating in fixed point. Each systolic array also has an adder and a FIFO buffer to compute (show in Figure 5.4), store, and synchronize the exponents of the operands. Upon completion of the multiplication, the block floating point values are converted to bfloat16 for use by the SIMD unit. The SIMD results are finally converted back to block floating point and written back to the activation buffer.

The next enhancement is in the request dispatcher to support hosting inference and training services simultaneously. Equinox keeps dedicated hardware contexts for each service, with a

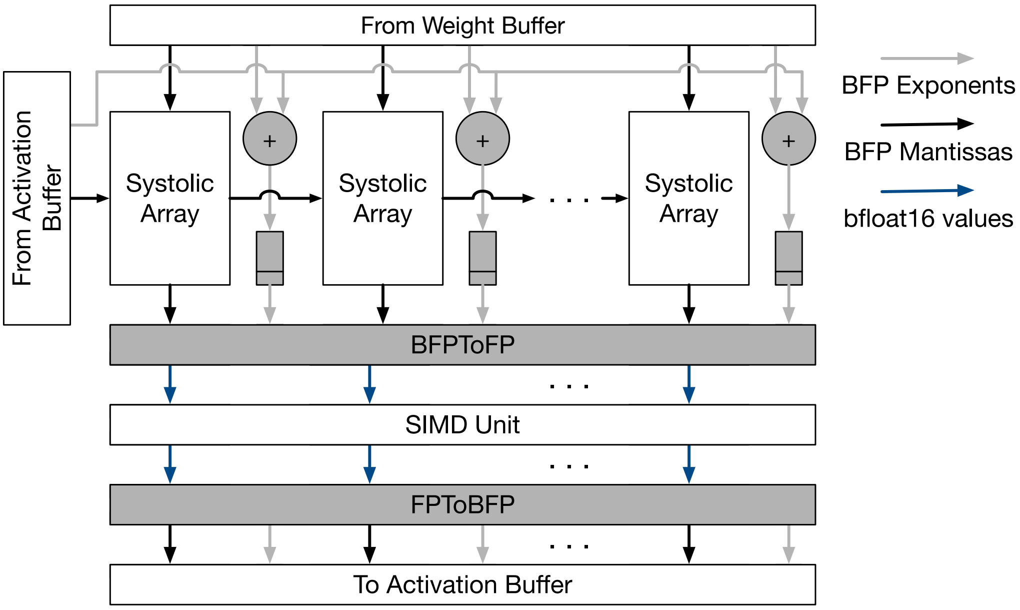


Figure 5.4 – Datapath modifications for HBFP.

context consisting of a request queue and an instruction counter. Contexts are only visible to the front-end of the accelerator, leaving the datapath oblivious to the interleaving of services. Each context has private space in the buffers, which is allocated at installation time. Training requests also arrive in batches and therefore bypass batch formation in the front-end.

The instruction controller is also modified to maintain inference latency guarantees in the presence of training services. The controller schedules instructions from both inference and training services, giving priority to inference requests, by following a round-robin policy only when inference queuing is low. To bound queuing delays to conform to service-level latency constraints, the controller monitors the incoming inference load for spikes by comparing the queue size against a maximum threshold defined at the installation time. When the inference load surpasses this threshold, the controller stops servicing training requests, dedicating all of the accelerator's execution resources to inference requests. The round-robin scheduling resumes when the inference load spike subsides.

5.3 Chapter Conclusion

DNN inference accelerators face a low average load due to service demand variability. Unfortunately, traditional inference accelerators do not have the mechanisms needed to expose inference idle cycles to training workloads. In this chapter, we introduce Equinox, a family of inference accelerators that piggybacks training services to reclaim inference idle cycles. Equinox accelerators reconcile the conflicting requirements of training and inference, achieving near-optimal inference throughput while exposing inference idle cycles to training services.

6 Evaluating Equinox Accelerators

Chapter 5 introduces Equinox, a family of inference accelerators that piggybacks training services. In this chapter, we evaluate Equinox using the Pareto-optimal configurations identified in Chapter 4. We start by introducing the methodology and proceed to evaluate the accelerator on various DNN workloads.

6.1 Methodology

6.1.1 Area and Power

In this chapter, we estimate the Equinox's area and power. We model all the blocks shown in Figure 6.1, except for the host interface. We estimate the area and power of the matrix multiplier, SIMD, dispatchers, and on-chip networks by implementing them in RTL and synthesizing them using the Synopsys Design Compiler and TSMC 28nm technology (with the TCBN28HPMBWP35 Core library and Vdd of 0.9V), using a wire load model. We model power with the Power Compiler, using a statically assigned toggle rate of 1. We model the area and power of the design's large SRAM structures with CACTI 6.5 [45]. Because CACTI does not support 28nm, we scale down the values from 32nm using the methodology found in [17]. Finally, we extract the area and power of the DRAM interface from [62, 33].

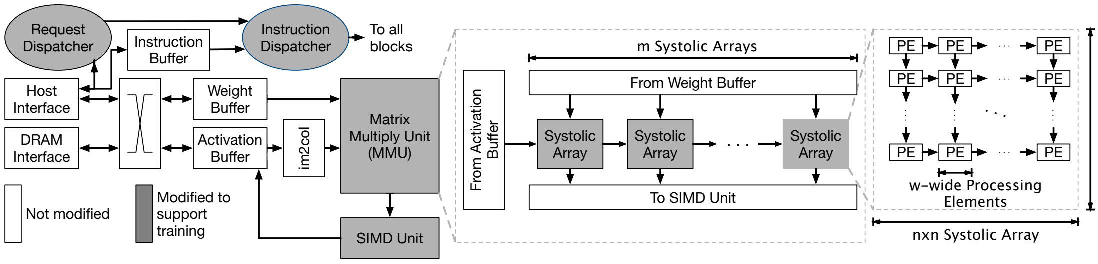


Figure 6.1 – Block diagram of Equinox's RTL model.

All the RTL modules are implemented using the Chisel HDL [2]. We choose Chisel over traditional HDLs because it facilitates the design of parameterized modules as it is embedded in a modern programming language — Scala. We also use Scala-based test benches to unit test all the Chisel modules. Unfortunately, Chisel simulation is much slower than Verilog RTL. As such, to perform longer simulations and integration tests, we generate Verilog and simulate it using traces generated by Scala test benches. Finally, we use the same Verilog code from the tests to synthesize the design.

Both dispatchers, shown on the top of Figure 6.1 are controlled by relatively simple finite state machines, amounting to approximately 500 lines of Chisel RTL and another 500 lines of Scala testbench code each. The request dispatcher state machine has a front end, which reads requests from the host interface, identifies the service requested, and queues them in the appropriate queue. The back end of the dispatcher keeps track of inference batch formation, issuing inference batches whenever there is no inference batch executing. Whenever the first request of a batch arrives, we reset a timer to measure the time this request waits for batch formation. If this timer value goes over a threshold specified by the user, the dispatcher issues incomplete batches. We keep a timer for each batch that fits into the request waiting queue.

The instruction dispatcher issues instructions whenever inference or training services are running and the downstream command queues are not full. The dispatcher selects a service to issue from using two policies. First, it reads the number of inference requests waiting in the request dispatcher. If this number is larger than the threshold specified by the service, the dispatcher always issues inference instructions. If the number of requests waiting is smaller than the threshold, it uses a round-robin policy, alternating between inference and training services. The instruction dispatcher ignores dependencies and only stops issuing instructions when the downstream command queues are full.

We also implemented all the computational units and on-chip buffers in Chisel RTL. The Matrix Multiplication Unit is the largest but less complex module, composed of a 1D array of systolic arrays with simple processing elements (PE). The Matrix Multiplication Unit has low complexity because it has a very regular pattern of replicated and connected PEs. Using Chisel dramatically facilitates the development of such modules. Additionally, the systolic array and PEs feature minimal control logic to command decoding and backpressure logic. Finally, each PE has minimum buffering, with enough space for a single weight and a single accumulated value. Overall, the systolic array code amounts to less than 100 lines of RTL and around 300 lines of testing code.

The SIMD unit is a simplified core featuring commands to execute arithmetic operations but without branches or control commands. We implement vector-vector operations like multiplication, addition, and activation functions. We implement complex activation functions (sigmoid, hyperbolic tangent, and their derivatives) using piecewise linear approximation [46] with a 128-wide look-up table. We add several ALUs per SIMD lane to enable the pipelining of vector-vector operations when many appear in a row.

Finally, we use a banked on-chip buffer architecture to support multiple ports in both the activation and weight buffers. The weight buffer has two read ports and one write port, while the activation buffer has two read and two write ports. Both the weight and activation buffers are divided into eight single-ported banks. We use a crossbar to route data and addresses between the banks and the ports. When bank conflicts occur, we prioritize first the ports facing the matrix multiplication units, second the ones facing SIMD units, and last the ports facing external interfaces. In the activation buffer, we also perform dependency tracking, preventing reads from accessing data from pending writes. This dependency tracking mechanism is sufficient because all the dependencies in a DNN request occur in activations.

6.1.2 Simulation

We evaluate Equinox's performance using a cycle-accurate simulator written from scratch in Python, using a module-oriented architecture, inspired by Flexus [26]. Python introduces an order of magnitude slowdown over a C++ based simulator like Flexus but dramatically facilitates prototyping. We can tolerate the slowdown for several reasons. DNN accelerators feature relatively simple memory and control systems, without any speculation or prediction. The number of modules simulated is relatively small compared to multi-core CPUs, which feature few modules for each core simulated, leading to simulators with hundreds of modules. The accelerator, in contrast, requires only a few modules. The largest component in the accelerator is the Matrix Multiplication Unit, which may have hundreds of PEs. However, because its performance is easily predictable, we can model it without losing precision with a single module. Finally, to reduce simulation time, we do not compute results since they do not affect the accelerator's timing.

We validate the simulator's results against RTL traces. All the blocks shown in Figure 6.1, except for the DRAM and host interfaces, are implemented both in RTL and in Python modules, with both implementations exhibiting identical timing properties. We do not implement the detailed DRAM controllers in detail as they do not strongly affect performance. DNN accelerators exhibit streaming memory access patterns, which minimize the effect of DRAM controllers over performance. As such, a detailed controller model greatly increases the complexity of simulation without increasing the results' accuracy by much. We compared the performance of a simple throughput- and latency-limited model against DRAMSim [58] and observe that, as long as data is accessed in 512-bit blocks, the simple model exhibits the same latency and throughput as the detailed controller. We also evaluate the performance of the host interfaces using a similar simple model.

Table 6.1 – bfloat16 Equinox accelerators evaluated.

Latency constraint	Batch size	Frequency (MHz)	Average Latency (s)	Throughput (TOp/s)
Min. latency	1	532	$37.3\mu s$	23.9
Latency $< 500\mu s$	29	610	$386\mu s$	63.3
Max. Throughput	39	610	$510\mu s$	66.7

Table 6.2 – HBFP Equinox accelerators evaluated.

Latency constraint	Batch size	Frequency (MHz)	Average Latency (s)	Throughput (TOp/s)
Min. latency	1	532	$15.6\mu s$	60.2
Latency $< 50\mu s$	16	532	$49.2\mu s$	333
Latency $< 500\mu s$	143	610	$381\mu s$	390
Max. Throughput	191	610	$509\mu s$	400

6.1.3 Equinox Configurations

To aid the explanation of Equinox's possible configurations, we introduce the following notation: $Equinox_c$ refers to the Equinox configuration with a latency constraint of c. We use the four optimal Equinox configurations introduced in Tables 6.1 and 6.2 and name them $Equinox_{min}$, $Equinox_{50\mu s}$, $Equinox_{500\mu s}$, and $Equinox_{none}$ accordingly. In all configurations, we divide the accelerator's SRAM among the activation, weight and instruction buffers, and SIMD register files, allocating $20MB$, $50MB$, $32KB$ and $5MB$ to each, respectively. All configurations use adaptive batching and hardware priority scheduling unless stated otherwise.

6.1.4 Workloads

We use three workloads to evaluate Equinox's performance. The first two are taken from DeepBench and represent a machine translation LSTM with 2048 hidden units and 25 steps, and a speech recognition GRU with 2816 hidden units and 1500 time-steps [47]. The third workload is a CNN model using Resnet50 [28]. The three models cover a wide range of inference service times. LSTM has a sub-millisecond service time, while Resnet50 has a service time of a few milliseconds, and GRU has a service time of tens of milliseconds. We use LSTM as our primary workload to evaluate both training and inference performance of Equinox, and use the other two only to do a sensitivity analysis of Equinox's performance against various models. For experiments in which Equinox hosts both training and inference services, we use two independent instances of the LSTM model.

For inference services, we set the batch size large enough to fully utilize Equinox's resources. Moreover, for training services, we assume a batch size of 128 when modeling the forward and backward passes. We assume that distributed training uses a parameter server that receives gradients, aggregates them, generates an updated model, and transfers it to Equinox for the next iteration of training. We simulate synchronous training.

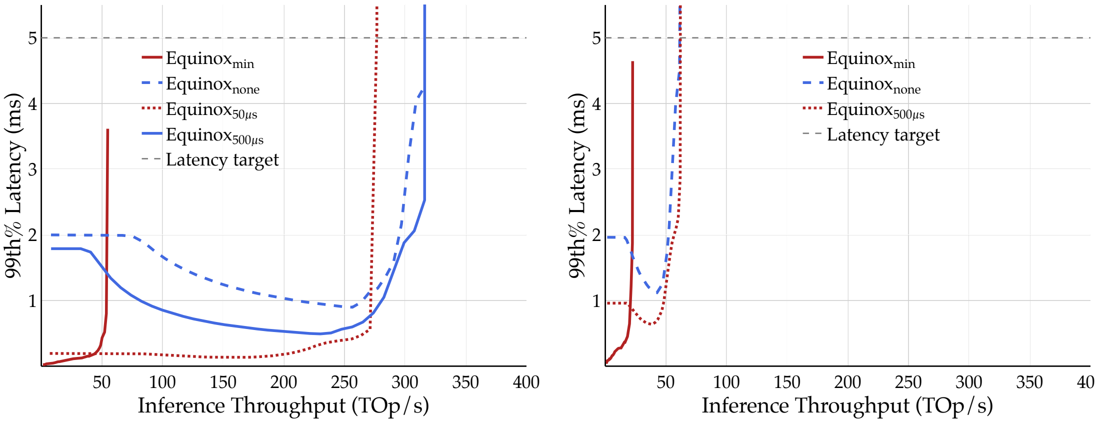
To model the incoming request traffic, we use a load generator that creates inference requests following Poisson arrival rates, while assuming there are always training requests to be processed. We set the 99th% latency target of inference services at 10× their mean service time when being processed by $Equinox_{500\mu s}$, which is in line with prior work [55, 10].

6.2 Evaluation

We now proceed to evaluate Equinox. We first corroborate the conclusions of Chapter 4. We then evaluate Equinox's effectiveness in exposing idle cycles to training services. Next, we show that Equinox's performance is insensitive to the type of DNN workload it executes. We then present synthesis results, followed by an evaluation of Equinox's scheduling capabilities and its adaptive batching mechanism.

6.2.1 hbfp8 vs. bfloat16

Figure 6.2 compares the inference latency and throughput of our Equinox configurations with the hbfp8 (left) and bfloat16 (right) variants. These results corroborate our analytical model conclusions (Chapter 4) with hbfp8 achieving up to 5.15× higher throughput compared to bfloat16 under the same target latency. Not surprisingly, we also observe that our designs reach their saturation point at a lower throughput than the maximum value predicted in Tables 6.1 and 6.2. These results are from timing simulation accounting for all component



(a) hbfp8

(b) bfloat16

Figure 6.2 – Equinox inference tail latency as a function of its throughput.

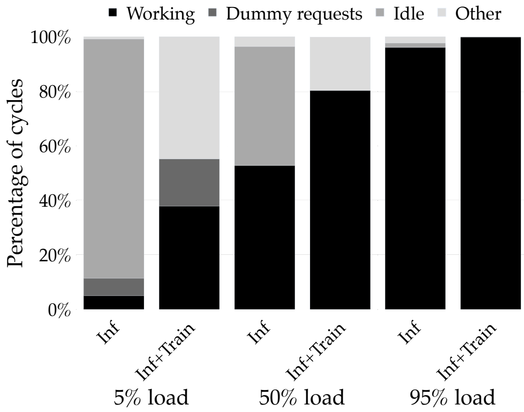
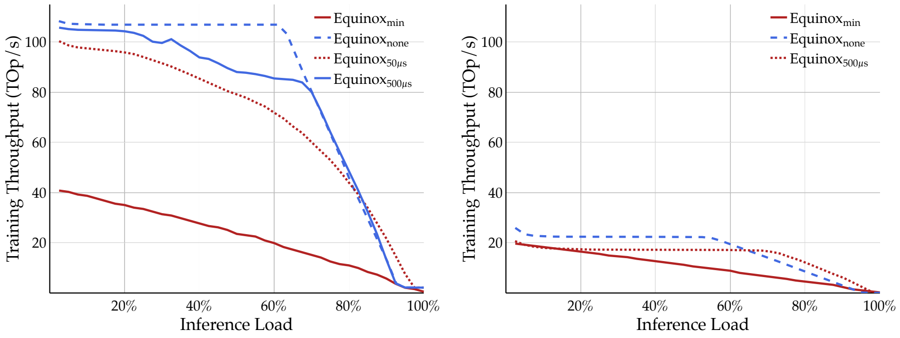


Figure 6.3 – Cycle usage breakdown of $Equinox_{500\mu s}$ at various loads, with and without hosting training. *Inf* indicates the accelerator hosts inference only, and *Inf+Train* indicates that the accelerator hosts inference and training.

latencies, pipeline hazards, and queuing effects (e.g., buffer port contention and dependence stalls), which are not taken into account by the analytical model.

6.2.2 Equinox Cycle Breakdown

To show how Equinox leverages training workloads to turn idle cycles into useful cycles, we plot the cycle usage breakdown of $Equinox_{500\mu s}$ when serving inference at various loads, both in isolation and when piggybacking training services. Figure 6.3 shows a breakdown of all MMU cycles into four categories: working cycles, cycles spent on computing dummy requests



(a) hbfp8

(b) bfloat16

Figure 6.4 – Equinox training throughput as a function of the inference load.

added to fill incomplete batches, idle cycles and other wasted cycles caused by buffer port contention, dependence stalls and stalls caused by a mismatch between the dimensions of ALU arrays and the matrix multiplications executed.

At 5% load, not surprisingly, over 80% of the cycles are idle, and up to 5% are wasted computing dummy requests. When training is added, most of the idle cycles are reclaimed, as the second bar shows. The second bar also shows that 50% of the cycles are wasted for "other" reasons. When we add training, we claim a few of the cycles shown as idle in the first bar, but we also turn many of the idle cycles into dependency stalls, when the accelerator is waiting for training operands to be fetched from DRAM. If the inference load was higher, we would be able to schedule inference requests during these cycles. Additionally, the cycles spent on dummy requests increase. When training is added, the waiting time for inference requests naturally increases, which causes the requests scheduler to issue more incomplete batches. At 50% load, adding training pushes the number of working cycles to around 80%, almost saturating the accelerator. At 95% load, the accelerator is saturated, and training requests are scheduled rarely. Finally, the other stalls are also unavoidable and remain even as we approach the saturation of the accelerator resources.

6.2.3 Training Throughput

Figure 6.4 compares the training throughput of our Equinox configurations with the hbfp8 (left) and bfloat16 (right) variants. We first observe that hbfp8 reaches a much higher throughput than bfloat16, showing once again that hbfp8 is a better fit for Equinox than bfloat16. Second, we see that $Equinox_{none}$, the configuration with no latency constraint, achieves the highest throughput. $Equinox_{none}$ saturates the HBM bandwidth when the inference load is below 60%, reaching the maximum achievable throughput for the LSTM training workload. Other configurations reach lower throughput values as tighter latency constraints shorten the window in which inference requests can be interleaved with training. At 60% load, when using hbfp8, $Equinox_{500\mu s}$, $Equinox_{50\mu s}$, and $Equinox_{min}$ reach 78%, 66% and 19% of the maximum training throughput, respectively. We conclude that designs with more relaxed latency constraints are a better fit for Equinox, reaching close to the maximum available training throughput.

6.2.4 Workload Sensitivity Analysis

Table 6.3 shows $Equinox_{500\mu s}$ performance when executing various DNN models. First, we observe that Equinox delivers significantly higher throughput when using hbfp8, for both inference and training. This trend is observed across workloads. Additionally, for all models except LSTMs, hbfp8 delivers lower latency.

We also observe that, despite the numeric encoding used, Equinox accelerators deliver the same inference throughput for LSTM and GRU, showing that it is insensitive to the two orders of magnitude difference between the latency constraints of the two models. $Equinox_{500\mu s}$ delivers the same training throughput for both LSTM and GRU ($83.4 TOP/s$ for hbfp8 and $17 TOP/s$ for bfloat16), showing that Equinox is capable of exposing training cycles to workloads with a variety of training execution times. The third and sixth rows of the table show the throughput and latency of Resnet50. In the case of Resnet50, $Equinox_{500\mu s}$ operates at a fraction of its maximum inference and training throughputs because Resnet50 features matrix

	Model	Training Throughput (TOp/s)	Inference Throughput (TOp/s)	Inference service time (ms)
hbfp8	LSTM	83.4	319	0.5
	GRU	83.4	319	36.6
	Resnet50	18.2	67.5	1.32
bfloat16	LSTM	17.2	61.3	0.5
	GRU	16.8	55.3	46
	Resnet50	4.2	18.2	4

Table 6.3 – Training and inference performance for various DNN models. Training throughput is measured with an inference load of 60%. Inference throughput refers to the maximum throughput achieved while maintaining the service latency target.

multiplications that do not map well to the large MMU used in $Equinox_{500\mu s}$. This bottleneck has been observed in other accelerator designs with large MMUs [32], which also exhibit low throughput for CNNs. Therefore, Equinox behaves like a typical inference DNN accelerator while also exposing training throughput to a wide variety of models.

6.2.5 Synthesis Results

Table 6.4 shows the area and power of the various components of $Equinox_{500\mu s}$. The values closely match the ones modeled in our design space exploration (Chapter 4). We also confirm our assumption that the MMU, DRAM interface, and buffers together dominate the area and power consumption, taking nearly 95% and 82% of total area and power of the chip, respectively. Additionally, 13% of total power and 4% of the total area is consumed by the SIMD unit, which contains a large register file, and a large number of bfloat16 ALUs. The bfloat16 ALUs are introduced because of HBFP. As such, we consider the area and power of the SIMD units as an overhead compared to an inference accelerator that employs fixed point only. Finally, we show that both request and instruction dispatchers consume an insignificant amount of area and power (less than 1%), confirming that the mechanisms added to handle multiple contexts and scheduling instructions incur a low area and power overhead.

Component	Area (mm^2)	Power (W)
MMU	185.60	36.84
DRAM Interface	46.90	28.60
SIMD Unit	13.43	10.97
Weight Buffer	45.96	4.28
Activation Buffer	18.27	1.07
Request Dispatcher	0.79	0.20
Instruction Dispatcher	0.49	0.14
Others	6.39	3.77
Total	313.85	85.91

Table 6.4 – $Equinox_{500\mu s}$ area and power

We also measured the timing properties of $Equinox_{500\mu s}$. The design achieves a maximum frequency of 1GHz, with the critical path is in the systolic arrays PEs. The PEs' ALUs perform a multiply and accumulate operation in a single cycle, and cannot be pipelined because any delay in the PEs leads to an increase in the systolic arrays' latency. Any increase in the systolic arrays latency increases the stalls caused by data dependencies, hurting the performance of models with small matrices. The other accelerator components are pipelined, as both the buffers and dispatchers can tolerate a few extra cycles of delay without overall performance degradation. As such, we were able to eliminate the critical paths in these components.

6.2.6 Scheduling

Equinox hosts training and inference services simultaneously. To quantify the impact of hosting training services on inference's latency and throughput, we compare Equinox's inference performance with and without training. Figure 6.5 shows Equinox's 99% latency against its throughput while performing inference under two scheduling policies: with fair-share scheduling of training and inference services (Inf+Train+Fair sched), and with a policy that only schedules inference requests at high loads (Inf+Train+Priority sched). As the figure shows, training introduces a latency overhead even at low loads. Both scheduling policies behave similarly at low load, equally dividing the accelerator's execution resources between training and inference requests, leading to an increase in the service time observed by inference

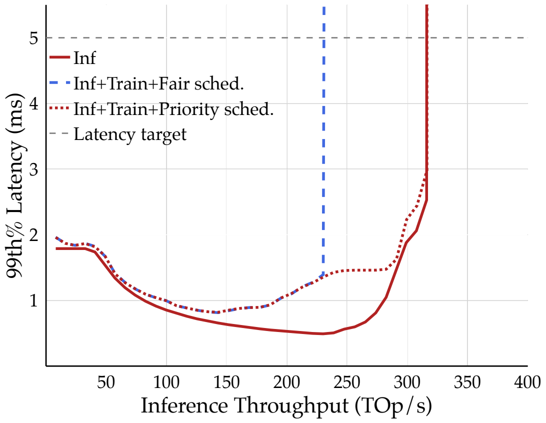


Figure 6.5 – Equinox inference tail latency against its throughput. *Inf* indicates a configuration without training, *Inf+Train+Fair sched.* indicates a configuration with inference, training, and a fair-share scheduler, and *Inf+Train+Priority sched.* indicates inference, training, and a priority scheduler.

requests, compared to the inference-only design. However, as inference load increases, the design with priority scheduling dedicates more ALU time to inference requests, outperforming the design with fair scheduling by 1.3× in terms of throughput under latency constraints and matching the throughput of the inference-only design. Equinox can host training services while delivering the same inference throughput as the baseline inference-only accelerator under the same service-level latency goals.

We also ran experiments to evaluate how Equinox behaves with software scheduling. We observe that, due to the high rate of instruction issue in Equinox, a software scheduler has to operate at a batch granularity. Scheduling at a batch granularity leads to inference requests being queued for a long time, violating the latency target when training batches are running. Hence, the software scheduler ends up not scheduling training batches in order to maintain the latency target, preventing Equinox from serving training requests altogether.

6.2.7 Adaptive Batching

To quantify adaptive batching's impact on the tail latency of inference requests, we compare the 99th% latency of $Equinox_{500\mu s}$ serving inference requests with static and adaptive batching policies at various loads. Figure 6.6 shows that static batching performs poorly at low

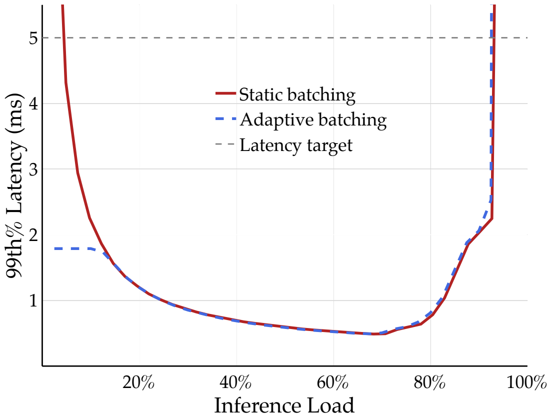
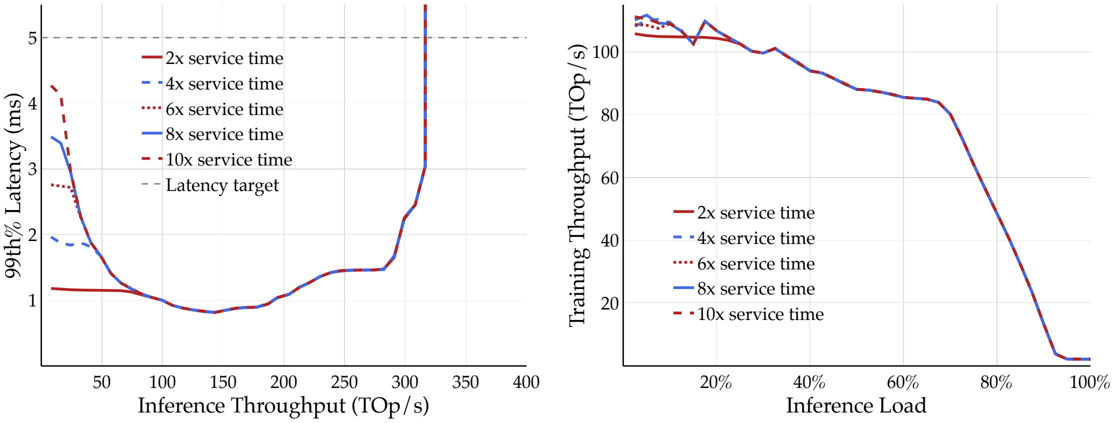


Figure 6.6 – Equinox's tail latency at various loads with static and adaptive batching policies.



(a) Tail latency as a function of inference throughput. (b) Training throughput as a function of inference load.

Figure 6.7 – Equinox's sensitivity to the adaptive batching threshold. "$X\times$ service time" indicates that the adaptive batching mechanism waits for $X\times$ the workload service time before issuing an incomplete batch.

loads, leading to latencies of more than 10× accelerator's service time, hence violating the service-level latency target. The inter-arrival time of requests is so high at low loads that batch formation dominates the execution time. The design with adaptive batching, however, bounds batch formation time leading to a 99th% latency that is close to the accelerator's service time when the load is low. Both designs exhibit the same trend in the presence of higher loads, as requests do not have to wait much longer for batches to form.

Adaptive batching uses a threshold value to decide how much time to wait before issuing an incomplete batch. Figure 6.7a shows the effect of this threshold over inference latency. We vary the threshold values from 2× the service time to 10× the service time. While increasing the threshold leads to higher 99th% latency, but interestingly, even waiting for longer than 10× the service time —i.e., waiting for longer than the 99th% latency goal—still does not violates the 99th% latency goal. The waiting time threshold rarely expires, not reaching more than 1% of the requests.

Finally, Figure 6.7b shows the training throughput obtained when the adaptive batching threshold is varied. Increasing the threshold beyond 2× the service time leads to a modest increase in training throughput, without violations of latency goals. However, as we increase the threshold, batch scheduling mechanisms incur long waiting times, with variation in training throughput. As using a threshold of 2× the service time leads to near-maximum and stable training throughput without violating latency goals, we pick this threshold for all experiments and workloads.

6.3 Chapter Conclusion

In this chapter, we evaluate the Equinox family of accelerators, showing $Equinox_{500\mu s}$ achieves 6.67× higher throughput than a latency-optimal accelerator. Equinox accelerators also achieve a training throughput that is 78% of the throughput of a training accelerator. Finally, we show that the mechanisms introduced by Equinox lead to an area and power overhead of less than 1%. At the same time, its numeric encoding incurs 13% power and 4% area overhead compared to a fixed-point accelerator.

7 Related work

In this chapter, we discuss prior work related to this thesis. Section 7.1 discusses work related to numeric representations in DNN inference and training, Section 7.2 discusses work related to DNN accelerators, and Section 7.3 presents work related to the co-location of online services with best-effort workloads.

7.1 DNN Numeric Representations

7.1.1 Hybrid Accelerators

The separation between dot products and other operations already exists in commodity hardware in NVIDIA Volta's FP16 Tensor Cores [50], in Microsoft's Brainwave [18], and in Google's Tensor Processing Unit [32]. Separating matrix multiplications from activations and other operations allow accelerator designs to provide optimized units for matrix multiplications, like systolic arrays. Like in HBFP, Brainwave also employs different numeric representations for matrix multiplications and activation operations. HBFP takes a step further and uses different numeric representations for these different operations in training, enabling training with dense fixed-point arithmetic for the vast majority of the operations used.

7.1.2 Inference with Reduced Precision

Quantization [21] is a widely used technique for DNN inference. BFP [60] has also been proposed for inference. These techniques quantize the weights of DNNs trained with single-precision floating point to use fixed-point logic during inference. HBFP targets both inference and the training process, serving as an enabler for accelerators that piggyback training with inference.

Posit [4], is a new numeric representation introduced for DNNs, which contains mantissa, exponent, and regime fields. The regime field enables posits to adjust the number of mantissa and exponent bits used to represent numbers, depending on the value of the number. posit-based accelerators using 8-bit operands can achieve the same accuracy as single-precision floating points. Unfortunately, posit operands require additional circuitry to implement multiply-and-accumulate operations in comparison to fixed points, leading to higher area and power.

7.1.3 Training with End-to-end Low Precision

Single-precision floating point ALUs have low energy efficiency and silicon density, leading DNN accelerator designers to adopt bfloat16 [14] and half-precision floating point [50] for the multiplications present in DNNs. These accelerators employ single-precision floating point for matrix multiplication accumulators to minimize the encoding's accuracy impact.

ZipML [70], DoReFa [71], and Flexpoint [36] train DNNs with end-to-end low precision. They use fixed-point arithmetic to represent weights and activations during the forward and backward passes and introduce various algorithms and restrictions to control the numeric range of activations or select quantization points for the fixed-point representations.

DoReFa [71] requires techniques to control the activations' magnitudes and is unable to quantize the first and last layers of networks. Others [1, 70] take a more theoretical approach to find the optimal quantization points for each dataset, performing both computations and communication using fixed-point arithmetic. We use BFP instead, effectively computing quantization points by choosing exponents at a finer granularity, before every dot product.

Flexpoint [36] performs all computations in fixed point. It uses the Autoflex algorithm twice per minibatch to predict the occurrence of overflows and adjust the tensor exponents accordingly. They leverage the slowly changing aspect of gradient exponents to minimize the number of exponent updates. However, to minimize overflows, they end up requiring conservatively large exponents, leaving the higher bits of mantissas unused and increasing mantissa width. Furthermore, Autoflex adds an artificial dependency between computations when it collects tensor value stats, making it unsuitable for DNNs that employ dynamic dataflow and limiting training scalability since it restricts the way DNNs can be sliced for distributed training. Our approach computes exponents more frequently, and it does so in-device, without requiring any additional stat collection and accommodating dynamic dataflows naturally. We observe that, as long as dot product calculations' intermediate values remain in fixed-point-like representations, conversions are infrequent enough that the conversion hardware area accounts for an insignificant fraction of the total accelerator area.

Finally, WAGE [67] and WAGEUBN [68] train DNNs using variable precision integers. They use different precisions for weights, activations, gradients, errors, updates (WAGEUBN only), and batch normalization (WAGEUBN only). Each type of value is quantized differently, and they use various batch-normalization-like transformations to keep values within the ranges that can be represented by integers. WAGE is evaluated only with AlexNet and achieves lower accuracy than floating-point-based models. WAGEUBN is evaluated with Resnet models and achieves accuracies below the state-of-the-art.

7.1.4 Binarized and Ternary Neural Networks

Binarized [31] and Ternary [37, 72] neural networks are another way to compress models. Although these networks enable inference with hardware that is orders of magnitude more efficient than floating-point hardware, they are trained like traditional neural networks, with both activations and parameters represented with floating point. Therefore, these approaches are orthogonal to BFP-based training. Other work [7, 57] uses binary operations for forward and backward passes but not for weight gradient calculation and accumulation. The new training algorithm is not transparent to users, requiring redesign of networks with numeric representation in mind. In contrast, our approach is backward compatible with FP32 models.

7.2 DNN Accelerator Design

7.2.1 Inference Accelerators

DNN accelerators are built for low latency inference services and employ low precision arithmetic for efficiency. However, their arithmetic encoding prevents them from being used for training services. Microsoft's Brainwave [18] is an inference-only accelerator which relies on FPGAs to accelerate DNNs at low latency. Although we argue that latency minimization is a misguided goal for ASIC accelerators, FPGA provisioning leads to a different conclusion. In FPGAs, ALU and data movement resources are not provisioned by the accelerator designer, but by the FPGA. As such, data movement resources are overprovisioned in FPGAs, to cater for general-purpose applications. For instance, one of the FPGAs featured in Brainwave, an Intel Stratix V D5, features 2014 blocks of M20K SRAM, with a width of up to 40 bits each. The same FPGA features 3180 18x18 multipliers, resulting in roughly 1.4 bits of SRAM bandwidth for each multiplier bit.

Google's TPU [32], in contrast, is an ASIC design that uses large systolic arrays and batching to cope with the memory bottleneck. The first version of TPU [32] uses 8- and 16-bit fixed point and does not support training. TPUv2 [23] and TPUv3 [22] employ bfloat16, giving up throughput to support training. The TPU designs do not have any mechanisms to support resource sharing between inference and training services, guaranteeing latency SLOs. Additionally, TPU designs offload all control to software. TPUv1 [32] receives instructions directly from the host through a PCIe interface. As such, the TPUv1 is incapable of implementing techniques such as dynamic batching.

The TPU and Brainwave designs are designed to provide specific latency guarantees to inference services. Brainwave [18] run DNN services without batching to minimize latency, leading to latencies from tens of microseconds to few milliseconds depending on the models. TPU executes services with higher latencies due to large batches, achieving few milliseconds for small MLPs.

Graphcore [24] uses a novel architecture to process DNNs. Instead of employing specialized matrix multiplication units like [18, 32], Graphcore uses a large number of SIMD-like processing units connected through a crossbar. Graphcore's programming model is composed of chains of computation phases, when each processing unit processes on private data, and communication phases, when processing units exchange intermediate results. Graphcore is optimized for small batches, getting around data movement bottlenecks by completely separating data movement and computation phases.

Finally, inference-optimized GPUs [51] use fixed-point arithmetic to maximize throughput. Compared to half-precision floating point, NVIDIA's T4 GPU throughput is 2× and 4× higher when using INT8 and INT4, respectively. Moreover, to avoid off-chip memory bottlenecks, GPU developers keep model weights in the register file [34] across kernel calls.

7.2.2 Training Accelerators

Training accelerators are optimized to provide high throughput to reduce the long execution time of training services. Due to the accuracy requirements of training algorithms, they employ floating-point arithmetic, resulting in lower energy efficiency and computational density. Some examples of training accelerators are NVIDIA's Volta [50] and Google's TPUv2 and TPUv3 [23, 22]. All of these accelerators feature high bandwidth memory.

Training accelerators are deployed in dedicated clusters [23, 27], featuring specialized network fabrics. The dedicated training clusters introduce datacenter heterogeneity, resulting in high management and maintenance costs [56], and leading to reduced resource utilization.

7.2.3 Data Movement in DNNs

Data movement is a well-understood bottleneck in DNNs [6, 19]. Prior work in CNN optimization [6] shows that accelerators that employ dataflow that maximize on-chip reuse achieve up to an order of magnitude higher energy efficiency. Interstellar [19] shows that optimal resource allocation in accelerators improves accelerator energy efficiency independent of the dataflow used. We study the relationship between data movement and latency, showing that accelerator latency constraints affect energy efficiency.

Batching is another technique used to minimize data movement in DNNs. Batching is especially effective for models that are based on vector-matrix multiplications. TPU [32] requires batching to achieve high utilization in its systolic arrays. Batching is also used in GPUs [9] to improve ALU utilization. GPUs require large matrices to utilize all its processing units. Ebird [9] introduces elastic batching in GPUs. They employ several workers, each using a different batch size. Their scheduler assigns requests to workers with smaller batches when the load is low, to minimize batch formation time. When the load is high, it assigns requests to workers with larger batches, to take advantage of reuse in GPUs. Elastic batching takes advantage of the non-linear relationship between latency and throughput in inference accelerators to manage request latency dynamically. It also implements the equivalent of adaptive batching when the load is low.

7.2.4 Scheduling Inference Services

Clipper [8] introduces a low-latency prediction serving system, with a control plane for accelerated inference serving. Clipper assumes a less autonomous DNN accelerator than Equinox, handling all the control logic in software. We argue that the control plane needed to enable low latency inference to execute with training can be implemented in hardware with low area and power overhead and low complexity. Equinox also enables for a more autonomous DNN accelerator, capable of running entire requests instead of running individual instructions. This model enables Brainwave-like accelerator, reducing latency further.

The NIVIDIA's Ampere [49] architecture introduces virtual GPUs, a mechanism that enables the user to statically divide GPU resources between various services. Virtual GPUs can be used to implement ColTraIn accelerators, as users can slice a physical GPU in two virtual GPUs, dedicating one slice for inference and another for training, shifting to using both slices for inference when the load is high. However, this division of resources is too coarse-grained and leads to underutilization of the various GPU resources. Virtual GPUs divide both DRAM bandwidth, register files, and ALUs across slices. As we have shown, inference and training services have different resource demands, and each slice might end up underutilizing different resources. We believe that a better implementation of ColTraIn on GPUs would introduce latency aware scheduling in both GPU schedulers.

7.3 Co-location of Latency Critical and Best-effort Tasks

Prior work [15, 39] co-locates latency-critical and best-effort tasks on online servers. We also leverage a best-effort service to improve the utilization of computing resources executing latency-critical services. However, the challenges in co-locating inference and training in accelerators are fundamentally different from the challenges of co-locating in online servers. In online servers, the biggest challenge is identifying the resources that are contended by each workload; and the mechanisms necessary to enable co-location are already present in the form of OS/cluster schedulers. Identifying contended resources in DNN accelerators is trivial. Instead, we tackled the challenges in provisioning accelerators silicon resources and introducing latency aware mechanism to specialized accelerators.

Additionally, Facebook [27] co-locates training and inference services on general-purpose CPUs during periods of low load. Unfortunately, general-purpose processors deliver an order of magnitude less energy efficiency than ASIC accelerators. As such, even though they reach superior utilization, they still impose higher maintenance costs than underutilized inference accelerators.

8 Concluding Remarks

DNNs have seen increased popularity as datacenter workloads, leading to an explosion in investment on DNN accelerator infrastructure. DNN accelerators, however, are part of online services that are subject to service demand variations, leading to low average load. As such, a large fraction of the resources invested in DNN inference accelerators goes to waste, as they are poorly utilized. In this thesis, we show that — much like in general-purpose computing platform — best-effort tasks, like DNN training, can be used to reclaim inference accelerator idle cycles. Inference idle cycles expose enormous computing resources for training, reducing the need for external training clusters, and improving datacenter homogeneity and energy efficiency.

The first challenge we tackle is the different numeric representations used in DNN inference and training. Traditionally, DNN training workloads require floating point to converge. Unfortunately, floating-point ALUs are up to an order of magnitude less energy-efficient and dense than the fixed-point encodings used in inference accelerators. The difference in encoding forces inference accelerators to give up energy efficiency to accommodate training services, forcing datacenter operators to pick between utilization and efficiency. We introduce Hybrid Block Floating Point (HBFP), a numeric encoding that combines block floating point with floating point to enable efficient DNN training with ALUs that are energy efficient and dense. HBFP uses block floating point for the large matrix multiplications that appear in DNN processing and floating point for activations, which account for less than 1% of the operations. HBFP reaches accuracy comparable to single-precision floating point, enabling inference accelerators to efficiently expose their idle cycles to training services.

The second challenge is to maximize inference throughput under latency constraints. Inference accelerators face a fundamental trade-off between reducing latency and maximizing throughput caused by batching. Datacenter operators are forced to use batching to improve weight reuse in DNN, to reduce data movement, and to increase throughput. Unfortunately, batching also increases the latency of individual requests. We show that, for numeric representations that lead to data-movement-bound ALUs, moderate batching leads to high throughput while maintaining reasonable latency constraints. As batch sizes increase beyond the sweet spot, throughput slowly increases while latency worsens quickly. We also show that batching is less effective for numeric representations that lead to compute-bound ALUs. We conclude that, with appropriate numeric encoding and latency constraints, ALU arrays achieve high throughput, providing opportunities for inference accelerators to expose more idle cycles to training services.

Finally, we introduce Equinox, a family of DNN inference accelerators that is capable of exposing idle cycles to training services. We enhance an inference accelerator to support inference and training contexts. We also add scheduling capabilities to execute training without affecting inference latency. We also design an HBFP datapath for Equinox, to execute inference and training efficiently. We evaluate Equinox to validate the findings of the analytical model. We also show that Equinox exposes inference idle cycles to training services without affecting inference maximum throughput or violating latency constraints. With HBFP-based ALU arrays designed with latency constraints that are relaxed enough, Equinox exposes enough idle cycles to training to saturate off-chip memory throughput, resulting in training throughput comparable to optimized training accelerators.

8.1 Future Directions

8.1.1 Improving DNN Efficiency in Datacenter

While the popularity of DNNs in datacenters has increased, the technological scaling trends that enabled the rise of modern DNNs have since stopped, leaving datacenter operators without an answer for the surge in computing requirements triggered by DNNs. Even assuming fully utilized fixed-point-based DNN accelerators, performance has reached a plateau, and only hardware-software codesign will bring the next order-of-magnitude improvement on power or area consumption for DNN accelerators.

A key challenge in DNN processing is data movement. Data movement has become such a significant constraint that even if ALUs executed computations for free (i.e., without consuming any power), accelerator power would not improve by another order-of-magnitude. As such, we need techniques — such as model and gradient compression — to minimize data movement. Unfortunately, operating on compressed data incurs area and power overheads, which are not well understood. We need to understand the effects of data compression on accelerator power, throughput, and latency to develop better compression mechanisms.

The data compression problem can also be studied from the lenses of numeric encodings. Numeric encodings are optimized for single ALUs operating on uncompressed data. These assumptions need to be revisited. In HBFP, we optimized the numeric encoding for the ALUs that execute dot products. The next natural step is to optimize model and gradient compression for dot product operations. For instance, while model compression leads to sparsity, a large fraction of accelerator power is spent on aligning the operators to perform dot products. Dot product properties can be leveraged to avoid this alignment, for instance, ignoring small operands.

Finally, assuming that the next order-of-magnitude improvement arrives for DNN accelerators, we need to understand the effects of large ALU arrays over DNN latency. Large ALU arrays usually achieve reduced utilization when executing DNNs with small dimensions. One option similar to batching is to partition ALU arrays to execute multiple requests in parallel, potentially improving ALU utilization. This approach, however, negatively affects the latency of individual requests. We hypothesize that this added dimension introduces an effect similar to batching.

8.1.2 Expanding the Scope of ColTraIn

In this thesis, we show that an inference accelerator can efficiently expose idle cycles to training services. A natural step forward would be to investigate how the addition of training services affects large accelerator fabrics. Adding training to inference accelerators puts extra pressure on load balancers and networks, increasing datacenter operating costs. The amount of training throughput exposed by Equinox far outweighs any extra operating costs incurred when using efficient arithmetic representations and relaxed latency constraints. However, when conditions are not as favorable, these operating costs might introduce an interesting trade-off.

Additionally, with the emergence of sophisticated machine learning pipelines involving DNNs [25], inference accelerators require expansion in their capabilities. Workloads like natural language processing and recommender systems introduce DNN operations that are tightly coupled with irregular operations that resemble traditional datacenter workloads. We expect some of these computations to migrate to DNN accelerators, making accelerators resources more heterogeneous. This trend introduces a challenge and an opportunity for Equinox-like accelerators. Sharing more heterogeneous resources requires expanded scheduling capabilities, increasing Equinox's complexity. However, accelerators with heterogeneous resources might be able to perform other tasks when the load is low.

Finally, as datacenter operators adopt accelerators for workloads other than DNNs, the issue of idle cycles will appear for each workload. In DNNs, we leverage training tasks to improve accelerator utilization. Additionally, DNN accelerator resources do not feature complex interactions that complicate scheduling. Other accelerators might not have such an obvious best-effort workload or might feature complex resources interactions. One example is GPUs, which have plenty of best-effort workloads that are suitable for utilizing idle cycles. However, managing resource utilization in GPUs is not trivial. GPUs feature several different on-chip resources like register files, shared memory space, thread slots. Additionally, GPUs depend on off-chip memory, requiring complex scheduling to maximize off-chip bandwidth utilization. As such, a GPU version of ColTraIn would require efficient numeric encodings and forgiving latency constraints to be effective.

Bibliography

[1] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding." in *Proceedings of the Thirty-first Conference on Neural Information Processing Systems (NeurIPS)*, 2017, pp. 1709–1720.

[2] J. Bachrach, H. Vo, B. C. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: constructing hardware in a Scala embedded language." in *Proceedings of the 49th Annual Design Automation Conference DAC12*, 2012, pp. 1216–1225.

[3] L. A. Barroso, U. Hlzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018.

[4] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Deep Positron: A Deep Neural Network Using the Posit Number System." in *Proceedings Design, Automation and Test in Europe Conference and Exhibition (DATE19)*, 2019, pp. 1421–1426.

[5] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture." in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 7:1–7:13.

[6] Y.-H. Chen, J. S. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks." in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367–379.

[7] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations." in *Proceedings of the Twenty-ninth Conference on Neural Information Processing Systems (NeurIPS)*, 2015, pp. 3123–3131.

[8] D. Crankshaw, X. W. 0066, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency Online Prediction Serving System." in *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 613–627.

[9] W. Cui, M. Wei, Q. Chen, X. Tang, J. Leng, L. Li, and M. Guo, "Ebird: Elastic Batch for Improving Responsiveness and Throughput of Deep Learning Services." in *Proceedings of the 37th International IEEE Conference on Computer Design (ICCD)*, 2019, pp. 497–505.

[10] A. Daglis, M. Sutherland, and B. Falsafi, "RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs." in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, 2019, pp. 35–48.

[11] W. Dally, "High Performance Hardware for Machine Learning." https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf, 2015, accessed: 2018-01-31.

[12] D. Das, N. Mellempudi, D. Mudigere, D. 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. O. Pirogov, "Mixed Precision Training of Convolutional Neural Networks using Integer Operations." *CoRR*, vol. abs/1802.00930, 2018.

[13] J. Dean and L. A. Barroso, "The tail at scale." *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[14] J. Dean, G. Corrado, R. Monga, K. C. 0010, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng, "Large Scale Distributed Deep Networks." in

Proceedings of the Twenty-sixth Conference on Neural Information Processing Systems (NeurIPS), 2012, pp. 1232–1240.

[15] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and QoS-aware cluster management." in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014, pp. 127–144.

[16] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi, "Training DNNs with Hybrid Block Floating Point." in *Proceedings of the Thirty-second Conference on Neural Information Processing Systems (NeurIPS)*, 2018, pp. 451–461.

[17] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling." in *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365–376.

[18] 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 *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*, 2018, pp. 1–14.

[19] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory." in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*, 2017, pp. 751–764.

[20] Google, "BERT," https://github.com/google-research/bert, accessed: 2020-03-24.

[21] Google, "How to Quantize Neural Networks with TensorFlow." https://www.tensorflow.org/performance/quantization, 2017, accessed: 2018-01-31.

[22] Google, "Tearing Apart Google's TPU 3.0 AI coprocessor." https://www.nextplatform.com/2018/05/10/tearing-apart-googles-tpu-3-0-ai-coprocessor, 2018, accessed: 2018-05-15.

[23] Google, “Cloud TPU,” https://cloud.google.com/tpu, 2020, accessed: 2020-03-24.

[24] Graphcore, “Introduction to the IPU architecture.” https://www.graphcore.ai/nips2017_presentations, 2017, accessed: 2019-08-06.

[25] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. M. Hazelwood, M. Hempstead, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, “The Architectural Implications of Facebook’s DNN-Based Personalized Recommendation.” in *Proceedings of the 26th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2020, pp. 488–501.

[26] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. Nowatzyk, “SimFlex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture.” *SIGMETRICS Perform. Evaluation Rev.*, vol. 31, no. 4, pp. 31–34, 2004.

[27] K. M. Hazelwood, S. Bird, D. M. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, “Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective.” in *Proceedings of the 24th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2018, pp. 620–629.

[28] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition.” in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[29] G. Huang, Y. S. 0020, Z. Liu, D. Sedra, and K. Q. Weinberger, “Deep Networks with Stochastic Depth.” in *Proceedings of the 13rd European Conference on Computer Vision(ECCV)*, 2016, pp. 646–661.

[30] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, “Densely Connected Convolutional Networks.” in *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2261–2269.

[31] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized Neural Networks.” in *Proceedings of the Thirtieth Conference on Neural Information Processing Systems (NeurIPS)*, 2016, pp. 4107–4115.

[32] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-Datacenter Performance Analysis of a Tensor Processing Unit.” in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[33] H. Jun, “Hbm (high bandwidth memory) for 2.5d,” 2015.

[34] F. Khorasani, H. A. Esfeden, N. B. Abu-Ghazaleh, and V. Sarkar, “In-Register Parameter Caching for Dynamic Neural Nets with Virtual Persistent Processor Specialization.” in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 377–389.

[35] A. Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” *Technical report, University of Toronto*, 2009.

[36] U. Kster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Hall, L. Hornof, A. Khosrowshahi, C. Kloss, R. J. Pai, and N. Rao, “Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks.” in *Proceedings of the Thirty-first Conference on Neural Information Processing Systems (NeurIPS)*, 2017, pp. 1742–1752.

[37] F. Li and B. Liu, “Ternary Weight Networks.” *CoRR*, vol. abs/1605.04711, 2016.

[38] T. Lin, S. U. Stich, and M. Jaggi, “Don’t Use Large Mini-Batches, Use Local SGD.” *CoRR*, vol. abs/1808.07217, 2018.

[39] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, “Heracles: improving resource efficiency at scale.” in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 450–462.

[40] Memcached, “Memcached: High-performance, distributed memory object caching system,” http://memcached.org/, accessed: 2020-03-24.

[41] S. Merity, N. S. Keskar, and R. Socher, “Regularizing and Optimizing LSTM Language Models.” *CoRR*, vol. abs/1708.02182, 2017.

[42] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. Garca, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, “Mixed Precision Training.” *CoRR*, vol. abs/1710.03740, 2017.

[43] T. Mikolov, M. Karafit, L. Burget, J. Cernock, and S. Khudanpur, “Recurrent neural network based language model.” in *Proceedings of the 11st Annual Conference of the International Speech Communication Association (INTERSPEECH)*, 2010, pp. 1045–1048.

[44] M. Mohammadi, A. I. Al-Fuqaha, S. Sorour, and M. Guizani, “Deep Learning for IoT Big Data and Streaming Analytics: A Survey.” *IEEE Commun. Surv. Tutorials*, vol. 20, no. 4, pp. 2923–2960, 2018.

[45] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, “Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0.” in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007, pp. 3–14.

[46] A. H. Namin, K. Leboeuf, R. Muscedere, H. Wu, and M. Ahmadi, “Efficient Hardware Implementation of the Hyperbolic Tangent Sigmoid Function.” in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS 2009)*, 2009, pp. 2117–2120.

[47] S. Narang and G. Diamos, "Baidu deepbench," https://github.com/baidu-research/DeepBench, 2017.

[48] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning." *Deep Learning and Unsupervised Feature Learning Workshop*, 2011.

[49] NVIDIA, "NVIDIA A100 Tensor Core GPU Architecture." https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, accessed: 2020-05-25.

[50] ——, "NVIDIA Volta V100 GPU Accelerator." https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf, 2018, accessed: 2018-01-31.

[51] ——, "NVIDIA T4 Tensor Core GPU." https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/t4-tensor-core-datasheet.pdf, 2019, accessed: 2019-01-07.

[52] ——, "NVIDIA DGX Essential Instrument of AI Research." https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-1-print-infographic-738238-nvidia-web.pdf, 2020, accessed: 2020-03-24.

[53] A. Pahlevan, J. Picorel, A. P. Zarandi, D. Rossi, M. Zapater, A. Bartolini, P. G. D. Valle, D. Atienza, L. Benini, and B. Falsafi, "Towards near-threshold server processors." in *Proceedings Design, Automation and Test in Europe Conference and Exhibition (DATE16)*, 2016, pp. 7–12.

[54] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch." *NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques*, 2017.

[55] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks." in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 325–341.

[56] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. R. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services." in *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 13–24.

[57] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks." in *Proceedings of the 13rd European Conference on Computer Vision(ECCV)*, 2016, pp. 525–542.

[58] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator." *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 16–19, 2011.

[59] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F.-F. Li, "ImageNet Large Scale Visual Recognition Challenge." *CoRR*, vol. abs/1409.0575, 2014.

[60] Z. Song, Z. Liu, and D. Wang, "Computation Error Analysis of Block Floating Point Arithmetic Oriented Convolution Neural Network Accelerator Design." in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018, pp. 816–823.

[61] V. Sze, Y.-H. Chen, T.-J. 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.

[62] K. Tran, "Start your hbm/2.5 d design today," 2016.

[63] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is All you Need." in *Proceedings of the Thirty-first Conference on Neural Information Processing Systems (NeurIPS)*, 2017, pp. 5998–6008.

[64] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, “Training Deep Neural Networks with 8-bit Floating Point Numbers.” in *Proceedings of the Thirty-second Conference on Neural Information Processing Systems (NeurIPS)*, 2018, pp. 7686–7695.

[65] Wikipedia, “English wikipedia,” https://en.wikipedia.org/wiki/English_Wikipedia, accessed: 2020-03-24.

[66] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. M. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. W. 0020, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang, “Machine Learning at Facebook: Understanding Inference at the Edge.” in *Proceedings of the 25th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2019, pp. 331–344.

[67] S. Wu, G. Li, F. Chen, and L. Shi, “Training and Inference with Integers in Deep Neural Networks.” in *Proceedings of the Sixth International Conference on Learning Representations (ICLR 2018)*, 2018.

[68] Y. Yang, L. Deng, S. Wu, T. Yan, Y. Xie, and G. Li, “Training high-performance and large-scale deep neural networks with full 8-bit integers.” *Neural Networks*, vol. 125, pp. 70–82, 2020.

[69] S. Zagoruyko and N. Komodakis, “Wide Residual Networks.” in *Proceedings of the 2016 British Machine Vision Conference (BMVC)*, 2016.

[70] H. Zhang, J. Li, K. Kara, D. Alistarh, J. Liu, and C. Zhang, “ZipML: Training Linear Models with End-to-End Low Precision, and a Little Bit of Deep Learning.” in *Proceedings of the Thirty-fourth International Conference on Machine Learning (ICML)*, 2017, pp. 4035–4043.

[71] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, “DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients.” *CoRR*, vol. abs/1606.06160, 2016.

[72] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained Ternary Quantization." *CoRR*, vol. abs/1612.01064, 2016.

[73] H. Zhu, D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and M. Erez, "Kelp: QoS for Accelerated Machine Learning Systems." in *Proceedings of the 25th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2019, pp. 172–184.

[74] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko, "Benchmarking and Analyzing Deep Neural Network Training." in *Proceedings of the 2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 88–100.

MARIO PAULO DRUMOND

+41 21 69 31379 ⋄ mario.drumond@epfl.ch
EPFL IC IINFCOM PARSA INJ 238 ⋄ Lausanne, VD 1015

EDUCATION

École Polytechnique Fédérale de Lausanne, EPFL, Switzerland *September 2014 - Ongoing*
Ph.D. Candidate in Computer Sciences

University of Wisconsin-Madison, United States *September 2012 - May 2013*
Exchange Program

Universidade Federal de Minas Gerais, UFMG, Brazil *January 2008 - December 2013*
B.S. in Electrical Engineering
Minor in Computer Systems

RESEARCH INTERESTS

Computer architecture, reconfigurable systems, datacenter architecture, machine learning

PUBLICATIONS

Optimus Prime: Accelerating Data Transformation in Servers. Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, *Mario Paulo Drumond*, Babak Falsafi, Christoph Koch. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020), 2020

Analog Neural Networks With Deep-Submicrometer Nonlinear Synapses Ahmet. C. Yzgler, Firat Celik, *Mario Drumond*, Babak Falsafi and Pascal Frossard. in IEEE Micro, vol. 39, 2019.

Training DNNs with Hybrid Block Floating Point. *Mario Drumond*, Tao Lin, Martin Jaggi, Babak Falsafi. In Proceedings of Neural Information Processing Systems. (NeurIPS 2018), 2018.

Algorithm/Architecture Co-Design for Near-Memory Processing. *Mario Drumond*, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, Dionisios Pnevmatikatos. SIGOPS Oper. Syst. Rev. 52, 1, 2018.

LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching. Mohammad Sadrosadati, Amirhossein Mirhosseini, Seyed Borna Ehsani, Hamid Sarbazi-Azad, *Mario Drumond*, Babak Falsafi, Rachata Ausavarungnirun, and Onur Mutlu. In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2018), 2018

The Mondrian Data Engine. *Mario Drumond*, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, Dionisios Pnevmatikatos. In Proceedings of 44th International Symposium on Computer Architecture (ISCA 2017), 2017.

MIAOW: An open source GPGPU. Vinay Gangadhar, Raghu Balasubramanian, *Mario Paulo Drumond*, Ziliang Guo, Jai Menon, Cherin Joseph, Robin Prakash, Sharath Prasad, Pradip Vallathol, Karu Sankaralingam. In IEEE Hot Chips 27 Symposium (HCS), 2015.

EXPERIENCE

Microsoft Research Summer 2017
Research Intern *Cambridge, UK*

- Worked on the Project Brainwave.
- Investigated neural network training on Microsoft's FPGA-accelerated deep learning infrastructure.

Microsoft Research Summer 2015
Research Intern *Redmond, Washington, United States*

- Worked on the Catapult project.
- Implemented direct communication between Catapult FPGAs and SSD storage devices

Invent Vision July 2013 - August 2014
System Analyst *Belo Horizonte, MG, Brazil*

- Design, implementation, and verification of FPGA-based smart cameras
- Mapping of image processing algorithms to FPGAs

Scanitec Equipamentos Automotivos April 2008 - July 2012
Intern/System Analyst *Belo Horizonte, MG, Brazil*

- Embedded systems programming with C and assembly.
- Embedded systems hardware design.
- CAN and LIN network application development.

RESEARCH PROJECTS

ColTraIn - ColTraIn: Co-locating DNN Training and Inference *June 2016 - Ongoing*
EPFL, Switzerland

The ColTraIn project seeks to build homogeneous datacenters for deep learning services, unifying training and inference algorithms on the same accelerator infrastructure.

We are co-designing DNN training algorithms and hardware to accelerate training with cheap, fixed-point-like arithmetic, matching the arithmetic used in inference accelerators. We presented Hybrid Block Floating Point, our arithmetic representation specially tailored for efficient co-located DNN training at NeurIPS'18.

The Mondrian Data Engine *July 2016 - Ongoing*
EPFL, Switzerland

I led the micro-architectural design of The Mondrian Data Engine. The Mondrian project co-designs algorithm and hardware to fully utilize memory bandwidth in near-memory processors.

Mondrian redesigns data-analytics algorithms to perform streaming accesses and avoid random memory access patterns. With streaming, we can use simple hardware to fully utilize the NMP's large internal bandwidth without modifying the internal structure of memory arrays, preserving DRAM cost advantages. We show that, even though streaming access algorithms may incur higher computational complexity, DRAM's internal bandwidth is so high that we can still observe substantial performance and efficiency gains. We presented Mondrian at ISCA 2017.

MIAOW: An open source GPGPU. *September 2012 - August 2014*
University of Wisconsin-Madison, United States

MIAOW is the first open-source GPGPU implementation, and one of the three academic presentations at Hot Chips 27. MIAOW is an RTL implementation of AMD's Southern Islands ISA, with an ASIC floor plan and a FPGA prototype.

I was one of the original designers of MIAOW, contributing to the high-level design of MIAOW's compute unit, in an effort led by Prof. Karu Sankaralingam. Later, I was in charge of the development and implementation of the two scheduling units in MIAOW: the wavefront scheduler and the ultra-threaded dispatcher.

AWARDS & HONORS

2019-2020	Qualcomm Innovation Fellowship
2017-2018	NVIDIA Graduate Fellowship Finalist
2014-2015	EPFL EDIC Fellowship

TECHNICAL SKILLS

Computer programming languages	Python, C++, C, Scala
Hardware description languages	Chisel, SystemVerilog, VHDL
OS	Unix-like OS's, Windows

PROFESSIONAL SERVICE

2018	ISCA Submissions co-chair

LANGUAGES

Portuguese	Mother tongue
English	Fluent
French	Intermediate