Equinox: Training (for Free) on a Custom Inference Accelerator

Mario Drumond*
mario.drumond@codedepot.ch
CodeDepot
Switzerland

Louis Coulon
louis.coulon@epfl.ch
EcoCloud, EPFL
Switzerland

Arash Pourhabibi
arash.pourhabibi@epfl.ch
EcoCloud, EPFL
Switzerland

Ahmet Caner Yüzügüler
ahmet.yuzuguler@epfl.ch
EcoCloud, EPFL
Switzerland

Babak Falsafi
babak.falsafi@epfl.ch
EcoCloud, EPFL
Switzerland

Martin Jaggi
martin.jaggi@epfl.ch
EcoCloud, EPFL
Switzerland

ABSTRACT

DNN inference accelerators executing online services exhibit low average loads because of service demand variability, leading to poor resource utilization. Unfortunately, reclaiming idle inference cycles is difficult as other workloads can not execute on a custom accelerator. With recent proposals for the use of fixed-point arithmetic in training, there are opportunities for training services to piggyback on inference accelerators. We make the observation that a key challenge in doing so is maintaining service-level latency constraints for inference. We show that relaxing latency constraints in an inference accelerator with ALU arrays that are batching-optimized achieves near-optimal throughput for a given area and power envelope while maintaining inference services' tail latency goals.

We present *Equinox*, a custom inference accelerator designed to piggyback training. Equinox employs a uniform arithmetic encoding to accommodate inference and training and a priority hardware scheduler with adaptive batching that interleaves training during idle inference cycles. For a 500μs inference service time constraint, Equinox achieves 6.67× higher throughput than a latency-optimal inference accelerator. Despite not being optimized for training services, Equinox achieves up to 78% of the throughput of a dedicated training accelerator that saturates the available compute resources and DRAM bandwidth. Finally, Equinox's controller logic incurs less than 1% power and area overhead, while the uniform encoding (to enable training) incurs 13% power and 4% area overhead compared to a fixed-point inference accelerator.

CCS CONCEPTS

• **Computer systems organization** → *Systolic arrays*; **Neural networks**; Cloud computing.

KEYWORDS

DNN accelerators, DNN inference, systolic arrays

*This work was done while the author was at EPFL.

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480057

ACM Reference Format:
Mario Drumond, Louis Coulon, Arash Pourhabibi, Ahmet Caner Yüzügüler, Babak Falsafi, and Martin Jaggi. 2021. Equinox: Training (for Free) on a Custom Inference Accelerator. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480057

1 INTRODUCTION

DNN infrastructure has observed an explosion in investment due to the increasing popularity of DNNs in online services [16, 18, 22]. 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 [7]. In general-purpose platforms, idle cycles are reclaimed by co-locating best-effort workloads with latency-critical ones [11]. As no other workload can execute on custom inference accelerators, those idle cycles stay unclaimed, leading to a considerable waste of resources.

While DNN training workloads can be used as best-effort tasks to reclaim inference idle cycles, the divergence in the requirements of inference and training workloads poses a significant challenge. Inference accelerators execute algorithms that can tolerate narrow fixed-point arithmetic and have small memory footprints which can be served directly from on-chip memory to achieve high throughput. Modern inference accelerators such as Graphcore [3] or Brainwave [16] have on-chip memory sizes ranging from tens up to hundreds of megabytes, which is sufficient to accommodate the memory footprint of state-of-the-art DNN models. Furthermore, because of tight latency constraints, these accelerators avoid techniques that may delay individual requests, such as batching [1, 16].

In contrast, training accelerators require floating-point arithmetic, exhibit memory footprints in the range of a few GBs [36] which cannot be easily accommodated on chip, and have no online latency constraints. As a result, training accelerators employ ALUs with reduced power efficiency, operate on DRAM-resident data, and use batching to minimize data movement and maximize throughput.

Modern inference accelerators—such as GPUs [1], TPUs [2, 5, 21], and Graphcore [3]—can often perform both inference and training. Unfortunately, they cannot efficiently train while meeting the tight latency constraints of inference services. These accelerators lack the capabilities to quickly switch between inference and training requests during inference load spikes and sacrifice inference throughput to accommodate floating-point arithmetic in the ALUs to support training.

Fortunately, the emergence of novel arithmetic encodings for DNN training [14, 24, 35] presents an opportunity to piggyback training on online inference services. Such encodings offer denser arithmetic without implications on the accuracy, resulting in orders of magnitude improvement in energy efficiency and computational density. These encodings lead to ALUs that consume much less power than the buffers feeding them. Hence, accelerators that employ them are data movement bound.

The fundamental challenge beyond arithmetic is designing an inference accelerator that piggybacks training while meeting service-level latency constraints. We make the observation that inference's sensitivity to service-level tail latency creates a non-linear relationship between latency and throughput. Designers rely on batching [16] to enable weight reuse, minimizing data movement bandwidth and power, thereby increasing the power provisioned for ALUs and throughput under a fixed power budget. When latency requirements are lax with higher degrees of batching, the on-chip data movement power is much lower, freeing up power for more ALUs and throughput, and consequently creating bigger opportunities to piggyback training on inference accelerators. Although such an accelerator cannot match the performance of a custom training accelerator, reclaiming idle cycles comes at a low cost, exposing cheap compute resources to datacenter operators. Finally, with priority scheduling and adaptive batching, an inference accelerator can seamlessly maintain service-level latency guarantees for inference requests while interleaving training requests.

We use text, image, and speech processing models together with analytical modeling, detailed cycle-accurate simulation, and synthesis tools for the TSMC 28nm technology to make the following contributions:

- We quantify the non-linear relationship between throughput and latency with batching in custom inference accelerators and show that optimized ALU arrays for a latency window of $50\mu s$ and $500\mu s$ can increase throughput by 5.53× and 6.67×, respectively, compared to latency-optimal arrays.
- We present Equinox, an inference accelerator designed to piggyback training to reclaim idle cycles, featuring a uniform encoding datapath and a priority scheduler that maintains service-level latency guarantees for inference requests while interleaving training requests.
- We present cycle-accurate simulation results making the case that Equinox can achieve up to 78% of a dedicated training accelerator's throughput that saturates the available compute resources and DRAM bandwidth. Equinox achieves this throughput while maintaining a 99th% latency goal for inference services that is within 10× of the mean service time.
- We present synthesis results indicating that Equinox's controller logic incurs < 1% power and area overheads, and its numeric encoding incurs a 13% power and 4% area overheads, respectively, compared to a fixed-point accelerator.

The rest of the paper is organized as follows. We first present the requirements for piggybacking training on inference accelerators (§2). Then, we present Equinox (§3) and explore its Pareto-optimal space to trade off latency for throughput and allow training (§4). Next, we present the methodology (§5) and detailed evaluation for a family of Equinox accelerators (§6). Finally, we discuss related work (§7) and conclude (§8).

2 PIGGYBACKING ON INFERENCE

In theory, with a uniform arithmetic encoding, it is possible to run inference and training services arbitrarily together in a single custom accelerator. In practice, custom inference and training accelerators, however, have diverse objectives that require conflicting resource provisioning. Because inference accelerators usually execute online services, they primarily operate with on-chip memory, and provision power and organize array processing elements for lower degrees of parallelism and batching to abide by tight latency constraints.

In contrast, training accelerators are optimized for maximum throughput without latency constraints and provision power to arithmetic density, parallelism with a high degree of batching, and DRAM bandwidth. In this paper, we focus on custom inference accelerators that can use their available idle resources for training without sacrificing latency. A study of how custom training accelerators can be used for inference services with a high tolerance for latency is beyond the scope of this paper.

To understand the challenges in piggybacking training on online inference services, we first consider the key objectives in designing a custom inference accelerator. We then present the requirements to enable using idle cycles in inference and accommodate training tasks without violating the inference accelerator's design objectives.

2.1 Design Objectives for Inference

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 a narrow fixed-point numeric encoding [16, 22]. Such an encoding results in up to an order of magnitude improvement in ALU silicon density (relative to floating point) [10], in memory capacity [16], and data movement bandwidth and power [32]. For example, Microsoft's Brainwave uses a variation of block floating point to process CNNs, RNNs and Transformers [16, 30], and TPUv1 uses 8- and 16-bit fixed point [22], allowing both to achieve superior throughput and lower latency than the state of the art at the time they were introduced.

Another key characteristic of inference workloads is their small memory footprint enabling designers to provision a larger fraction of overall power for ALUs and higher throughput. Indeed, in inference, the footprint is dominated by model weights, which are often small enough (a few KBs to 100s MBs) to fit entirely on chip [3, 16] with off-chip memory only present to accommodate the less frequent case of larger models (e.g., Brainwave [16], NVIDIA's T4 [1]). Consequently, data movement in inference accounts for a small fraction of the overall accelerator power budget as on-chip memory accesses consume little power (up to three orders of magnitude less, relative to off-chip accesses [10]). Both Graphcore [3] and Brainwave [16] 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 is usually part of a multi-tiered service where a user query triggers a sequence of sub-query fan-outs, spanning hundreds or thousands of servers [8, 18]. That fan-out effect places tight bounds on the tail response latency of each tier [22, 29].

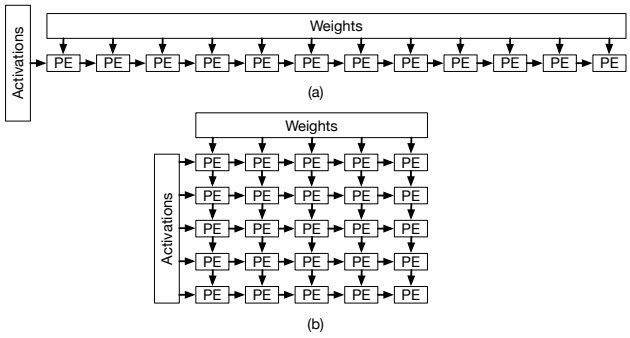


Figure 1: Two ALU arrays with the same power budget: (a) a systolic array with minimal latency similar to a vector processor (e.g., Brainwave [16]), and (b) a systolic array for batch jobs which trades off latency for throughput (e.g., TPU [22]).

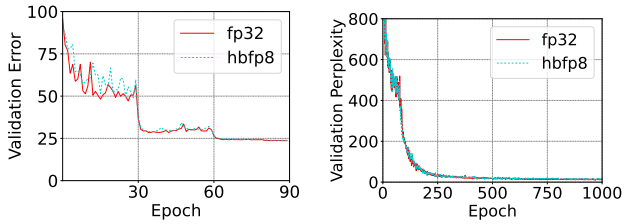
Inference's sensitivity to service-level tail latency creates a non-intuitive relationship between latency and throughput and a dilemma for designers. To increase effective throughput given a fixed power budget, designers improve weight reuse by batching [16], minimizing data movement bandwidth and power, 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, on-chip data movement power becomes negligible, freeing up power for ALUs and throughput.

Figure 1 illustrates this non-linear relationship between latency and throughput. The accelerator on the top with the highest latency constraints incorporates only a single row of ALUs without batching but requires a wide path to on-chip memory. In contrast, the accelerator on the bottom incorporates multiple rows of narrower ALUs 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 relaxing the constraints on tail latency a bit can lead to large increases in throughput and consequently, opportunities to piggyback training.

2.2 Accommodating Training

Inference accelerators are deployed at scale to accommodate online services but face around 30% average load because of service demand variability [7]. Unlike general-purpose servers, custom accelerators cannot execute other tasks while idle. The objective is to identify opportunities to recover these wasted cycles, evaluate the bounds of throughput achieved for training using these cycles, and present a design to do so. We now focus on the key challenges in accommodating training on custom inference accelerators.

A key challenge is that training algorithms, the most prominent of which is stochastic gradient descent (SGD), are sensitive to numeric encoding. Early attempts to use narrow fixed-point encodings in training have resulted in reduced accuracy or slower convergence [17]. Recent custom accelerators have adopted bfloat16 [2, 5] as an encoding, which is an improvement over both single- and



(a) Validation error of Restnet50 trained on the ImageNet dataset

(b) Validation perplexity of BERT trained on the Wikipedia dataset

Figure 2: The convergence rate and final error rate of hbfp8 is similar to single-precision floating point (fp32).

half-precision floating point, but it is still less efficient than fixed point [10].

Fortunately, recent work [14, 24, 30, 35] offers arithmetic encoding for training that is nearly as dense as fixed point while maintaining accuracy and convergence. In this paper, without loss of generality, we use hybrid block floating point (HBFP) [14], which employs fixed-point arithmetic (e.g., 8-bit mantissa) for all matrix operations while matching the accuracy and convergence rate of single-precision floating point (i.e., fp32). Figure 2a depicts the convergence rate of hbfp8 (i.e., HBFP using 8-bit mantissa) and fp32 presented in [14] for Resnet50 [27] on the ImageNet dataset [31]. Figure 2b presents the perplexity of a Bert-base [12] model trained with a dataset consisting of text scraped off of Wikipedia using both hbfp8 and fp32. Finally, we evaluate bfloat16, which is the state of the art in custom accelerators, as a reference encoding.

To piggyback training, the accelerator 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 [36]) 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. We make the observation that training is fundamentally bound by off-chip memory bandwidth and, to offset the bandwidth limitation, it requires a prohibitively high degree of batching well beyond what is affordable by custom accelerators. Therefore, with practical degrees of batching in inference accelerators, we expect contention for on-chip resources between inference and training to be relatively low. We show, in §4, how optimal batching and sizing of array dimensions can help a custom inference accelerator achieve high throughput with a minimal impact on latency.

Finally, 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

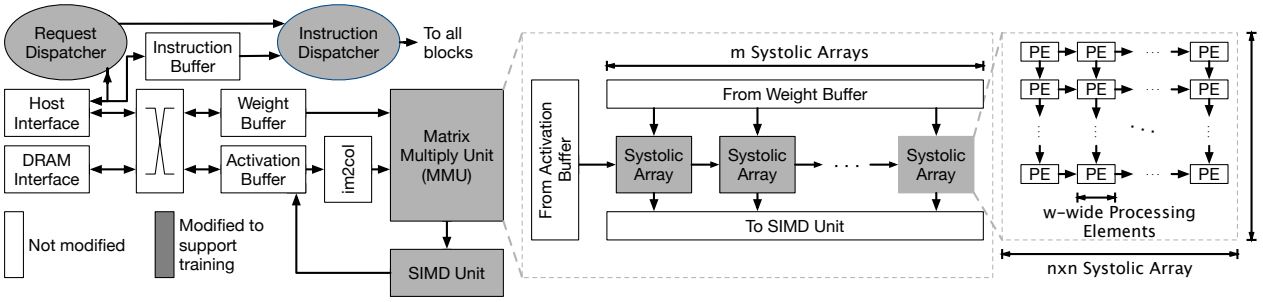


Figure 3: The anatomy of an Equinox accelerator. Shaded blocks indicate components enhanced to support training.

queuing delays due to the longer turnaround time in scheduling and instead present a hardware priority scheduler implemented with minimal logic.

3 EQUINOX

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

3.1 Baseline Inference Accelerator

Figure 3 depicts the anatomy of our baseline inference accelerator (e.g., TPU [22]). 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) described in detail in [13]. 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.

The accelerator's datapath (Figure 3) 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 interface. The MMU consists of a row of m systolic arrays, each with $n \times n$ processing elements (PEs) processing w values connected to the weight and activation buffers. The PEs operate on 2-D *tiles*, whose size is a function of the dimensions of the matrix multiply unit in the datapath. We later show in §4 how the three parameters, n, w, and m, collectively, enable systolic arrays to balance their latency and throughput. The im2col unit lowers convolutions to matrix multiplication.

Figure 4 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 weight matrix are divided into x tiles with $n * w$ side each. As shown in the Figure, each instruction addresses a single activation tile and m weight tiles, producing m tiles. To produce an output tile, the we use x instructions, each of which processes an entire activation tile row and an m weight tile rows. We also use x instructions to add the intermediate output tiles, producing the final tiles.

The SIMD unit performs vector-vector operations similar to the activation unit in TPU [22] 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 3), 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 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 dummy requests whose results are disposed. We compare adaptive batching with a static batching policy in §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 at the bottom of Figure 5, 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.

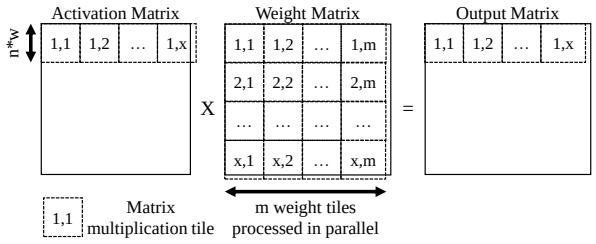


Figure 4: Division of a matrix multiplication into tiles.

3.2 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 3 and 5 indicate the mechanisms that are enhanced to support training.

The first enhancement is in the accelerator's ISA and datapath. We overload the SIMD unit's instructions to add support for derivative and loss calculations required by training. We also add arithmetic support for training to the datapath. Finally, we evaluate two versions of the MMU, one using bfloat16 [2, 5] as a state-of-the-art reference for custom accelerators and one using hbfp8 [14], which offers dramatically higher density. Because the area and power of the SIMD unit is 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 [2, 4, 5] 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 mantissa sharing a single 12-bit exponent. As such, all buffers are modified to store, read and write a block as an operand.

In block floating point, matrix multiplication can be implemented as a fixed-point multiplication of the tiles and an addition of the two exponents. To implement this in the systolic array we use 8-bit multipliers and 25-bit accumulators, both operating in fixed point. Each systolic array also has an adder and a FIFO buffer to compute, 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, which consists of a request queue and an instruction counter. Contexts are only visible to the accelerator's controllers, leaving the datapath oblivious to the interleaving of services. Each context has exclusive space in the buffers, allocated at installation time. Training requests also arrive in batches and therefore bypass batch formation in the front-end.

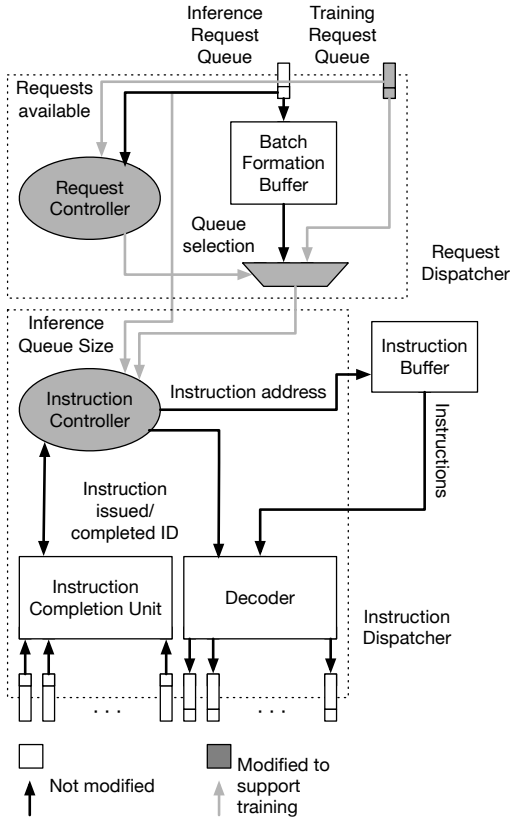


Figure 5: Equinox's front-end consisting of a request dispatcher (top) and an instruction dispatcher (bottom). Shaded blocks indicate components enhanced to support training.

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.

4 OPTIMAL ARRAY DESIGN

In this section, 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. Once we narrow down the design space to a few Pareto-optimal points, we evaluate them using detailed cycle-accurate simulation and synthesis in §6.

DNN workloads exhibit a varying degree of data reuse along an accelerator's dimensions. This difference in behavior affects the

required degree of batching and the relationship between throughput and latency. The matrix multiply unit (MMU) operates in two modes, depending on the dimensions of the input matrices. In the first mode, used for activation matrices with a small height relative to their length, the MMU broadcasts activations and unicasts weights across systolic arrays. As such, the MMU reuses activations both across and within systolic arrays (i.e., across m and n dimensions) but reuses weights only within systolic arrays (i.e., across the n dimension). The MMU uses this mode to process models based on vector-matrix multiplications, which dominate datacenter DNN workloads—such as RNNs and MLPs [18, 22]. To fully utilize the MMU, these models must use a minimum batch size of n.

In the second mode, used for activation matrices that have a large height relative to their length, such as those that appear in lowered convolutions, the MMU broadcasts weights across systolic arrays and unicasts activations. As a result, these matrices exhibit plenty of reuse and rarely present a bottleneck from a utilization perspective. Therefore, we focus our analysis on vector-matrix multiplications models from this point on.

Much like GPUs, DNN accelerators exhibit high degrees of power density which may result in dark silicon [3]. 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.

4.1 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 to 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.

We sweep the design space by varying n and the design frequency. For a given n and frequency, we find the largest values of m and w that are still below the area and power envelopes. Next, we calculate each design's throughput and latency with n, m, w, and the frequency. We also calculate area and power to guarantee that designs are under the 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.[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 [1, 3, 22]; 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 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 [26] 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 [15]. We assume a capacity of $75MB$, which is large enough to accommodate the majority of models used in datacenter services [16, 22]. To model the DRAM interface area, A_{dram}, we extract estimates from [33]. Equation 1 calculates the total area of the accelerator.

$$A = mn^2wa_{alu} + A_{sram} + A_{dram} \quad (1)$$

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 [22], and eliminate all candidate designs that exceed this power constraint. We model static power, P_{static}, only for 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 [28]. 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 [33] for DRAM accesses, P_{dram}. Equation 2 calculates the accelerator's total power consumption.

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

Performance Modeling. Equation 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 [27].

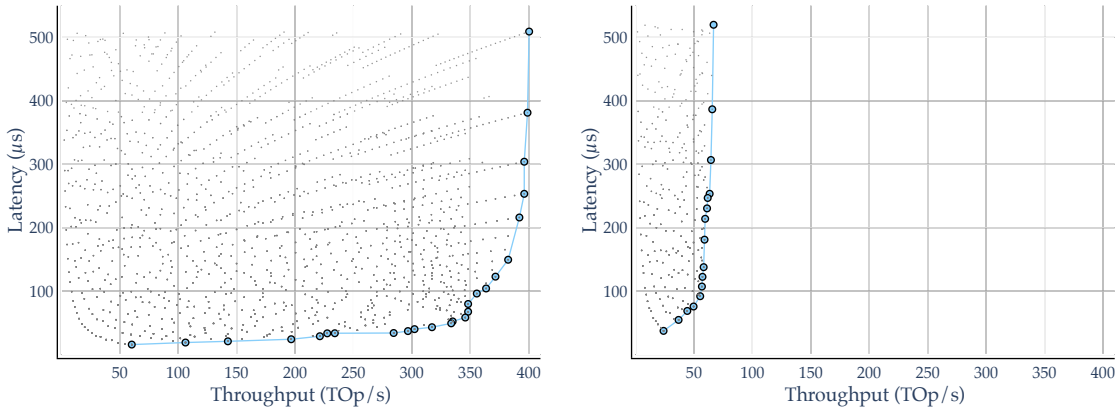
$$T = 2mn^2wf \quad (3)$$

4.2 Analysis

Figure 6 plots the latency against throughput for the modeled accelerators in the design space using (a) hbfp8 and (b) bfloat16 encodings. The designs on the Pareto-optimal frontier appear as large blue dots and the rest as small dots. Table 1 presents the value of n, frequency, latency, and throughput for the design points on the Pareto frontier for four design configurations based on latency constraints. The value of n is related to the batch size in models that use vector-matrix multiplications. To fully utilize the MMU PE array, these models must use a batch size of at least n. We use these configurations in §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

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



(a) hbfp8 (b) bfloat16

Figure 6: Latency vs. throughput for the modeled accelerators using (a) hbfp8 and (b) bfloat16 encodings.

reaching the knee past $350TOp/s$. 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 1(a)). Reducing this array dimension shifts a large fraction of the power for data movement from memory to ALUs in the second dimension with point-to-point links (Figure 1(b)), increasing 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 (n) lead to reductions in m and w. The reductions in m and w lead, in turn, to ALU arrays with larger columns but rows with similar size, 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 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 1 also shows that while today's custom accelerators either select designs without batching ($n = 1$) [16] or those with high batching degrees ($n >> 100$) [22], 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 accelerators are mostly power-limited. Designs with the highest latency constraints favor the lowest frequency ($532MHz$) because they spend most of their power on data movement.

5 METHODOLOGY

Simulation. We use an in-house detailed cycle-accurate simulator, written in Python, to evaluate Equinox's performance, taking into account the effects not included in our analytic models in §4. These include buffer and DRAM access delays, buffer port contention, operation inter-dependence delays, and queuing. We validate the simulator's results against RTL traces. All the blocks shown in Figure 3 are implemented both in RTL and in the simulator, with both implementations exhibiting identical timing properties. For DRAM and host interfaces, we compared the performance of throughput- and latency-limited models against DRAMSim [34] and verified that our analytic models estimate latency and throughput with high accuracy for 512-bit blocks.

Configurations. We use the notation $Equinox_c$ to refer to the optimal configuration with a latency constraint of c. Therefore, the four configurations in Table 1 are $Equinox_{min}$ for the latency-optimal configuration, $Equinox_{50\mu s}$ and $Equinox_{500\mu s}$ for the accelerators with a $50\mu s$ and $500\mu s$ latency constraint respectively, and $Equinox_{none}$ for an accelerator with no latency constraint. In all configurations, we divide the accelerator's SRAM among the activation, weight, 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.

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 [27]. The third workload is a CNN model using Resnet50 [19]. 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 main workload to evaluate training and inference performance and use the other two only to do a sensitivity analysis of 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, and 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, which

Table 1: Pareto-optimal designs under various latency constraints.

Latency constraint	bfloat16 n	bfloat16 Freq. (MHz)	bfloat16 Service Time (μs)	bfloat16 Throughput (TOp/s)	hbfp8 n	hbfp8 Freq. (MHz)	hbfp8 Service Time (μs)	hbfp8 Throughput (TOp/s)
Min. latency	1	532	37.3	23.9	1	532	15.6	60.2
Latency < $50\mu s$					16	532	49.2	333
Latency < $500\mu s$	29	610	386	63.3	143	610	381	390
No constraint	39	610	510	66.7	191	610	509	400

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 [9, 29].

Area and Power. In contrast to the methodology used in §4, in our detailed evaluation, we estimate the area and power of all the blocks, except for the host interface. We synthesize the compute units and controllers of $Equinox_{500\mu s}$ using the Synopsys Design Compiler and TSMC 28nm technology (with the TCBN28HPMBWP35 Core library and Vdd of 0.9V) and add the power and area for the SRAM structures using CACTI 6.5 [26]. Because CACTI does not support 28nm, we scale down the values from 32nm using the methodology found in [15]. We extract area and power of the DRAM interface from [33].

6 EVALUATION

We now proceed to evaluate Equinox. We first corroborate the conclusions of §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 and adaptive batching capabilities.

Inference Performance. Figure 7a shows the inference latency and throughput for our hbfp8 Equinox configurations. These results corroborate our analytical model conclusions (§4) with designs targeting relaxed latency constraints achieving up to 6× higher throughput compared to minimum latency designs. $Equinox_{min}$ has the smallest batch size, reaching the lowest 99th% latency at low load. However, it can only accommodate a limited number of requests due to the limited parallelism exposed by the latency-optimized systolic array. $Equinox_{50\mu s}$ relaxes the latency constraints used in $Equinox_{min}$, showing similar characteristics but reaches a 5× larger throughput.

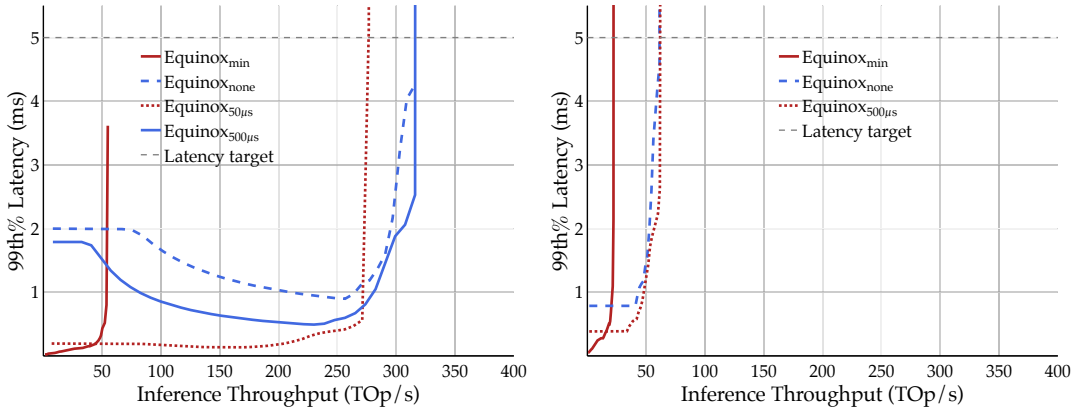
$Equinox_{500\mu s}$ relaxes the latency constraints further with larger batches to fully utilize the larger systolic array. At low load, because the scheduler waits for multiple requests to form complete batches (or extend smaller batches with dummy requests), the 99th% latency is bound by the scheduler's waiting time. However, at a higher load, the scheduler no longer waits and takes full advantage of the higher degree of parallelism and reaches the highest achievable throughput: up to 6x higher than the latency-optimized array.

Finally, comparing Figure 7a to Figure 7b, we also corroborate our analytical model analysis by observing that hbfp8 achieves up to 5.15× higher throughput compared to bfloat16 under the same target latency. These results are from timing simulation accounting for all component latencies, pipeline hazards, and queuing effects (e.g., buffer port contention and dependence stalls), which the analytical model does not consider.

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 8 shows a breakdown of all MMU cycles into four categories: working cycles, cycles spent on computing dummy requests added to fill incomplete batches, idle cycles, and other wasted cycles caused by buffer port contention, dependence stalls, and stalls caused by the mismatch between the dimensions of ALU arrays and the matrix multiplications executed.

At 5% load, not surprisingly, almost 50% of the cycles are idle, and nearly 40% are wasted computing dummy requests. When training is added, most of the idle cycles are reclaimed, as the second bar shows. Unfortunately, the training requests hit bottlenecks unrelated to Equinox, preventing them from reclaiming all the idle cycles. The cycles spent on dummy requests are, however, unavoidable at such a low load, as requests do not arrive fast enough to form a full batch. At 50% load, adding training pushes the number of working cycles up to around 80%, almost saturating the accelerator. At 95% load, the accelerator is saturated, and training requests are not scheduled at all. Finally, the other stalls are also unavoidable and remain even as we approach the saturation of the accelerator resources.

Training Throughput. Figure 9 shows the training throughput of our hbfp8 Equinox configurations. $Equinox_{none}$, the configuration with no latency constraint, achieves the highest throughput. In fact, $Equinox_{none}$ saturates the HBM bandwidth when 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. $Equinox_{500\mu s}$, $Equinox_{50\mu s}$, 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.



(a) hbfp8 (b) bfloat16

Figure 7: Equinox's inference tail latency as a function of its throughput for (a) hbfp8 and (b) bfloat16.

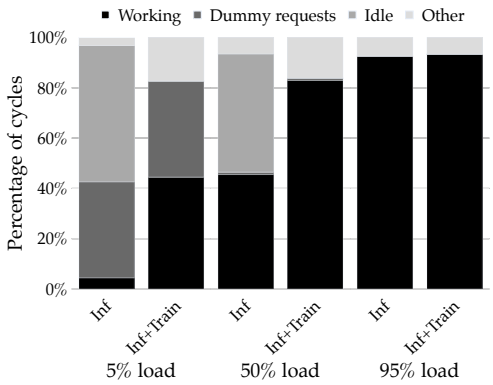


Figure 8: Cycle usage breakdown of $Equinox_{500\mu s}$ at various loads. *Inf* and *Inf+Train* configurations without and with training respectively.

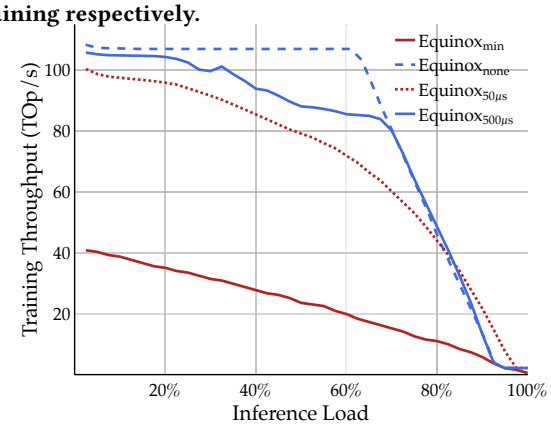


Figure 9: Training throughput vs. inference load.

Workload Sensitivity Analysis. Table 2 shows $Equinox_{500\mu s}$ performance when executing various DNN models. First, we observe that Equinox delivers 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.

Table 2: Training and inference performance for various DNN models. Training throughput is measured with an inference load of 60%. Inference throughput refers to maximum throughput.

Model	Training Throughput (TOp/s)	Inference Throughput (TOp/s)	Inference latency (ms)
LSTM	83.4	319	0.5
GRU	83.4	319	36.6
Resnet50	18	67	1.32

Similarly, Equinox delivers the same training throughput for both LSTM and GRU (83.4TOP/s), showing that Equinox can expose training cycles to workloads with a variety of training execution times. The third row of the table shows the throughput and latency of Resnet50. In Resnet50, Equinox operates at a fraction of its maximum inference and training throughputs because Resnet50 features matrix multiplications that do not map well to the large MMU used in Equinox. This bottleneck has been observed in other accelerator designs with large MMUs [22], 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.

Synthesis Results. Table 3 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 (§4). We also confirm our assumption that the MMU, DRAM interface, and buffers dominate the area and power consumption, taking nearly 95% and 82% of the total area and power of the chip, respectively. Additionally, 13% of the power and 4% of the area are 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.

Table 3: Area and power for $Equinox_{500\mu s}$.

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

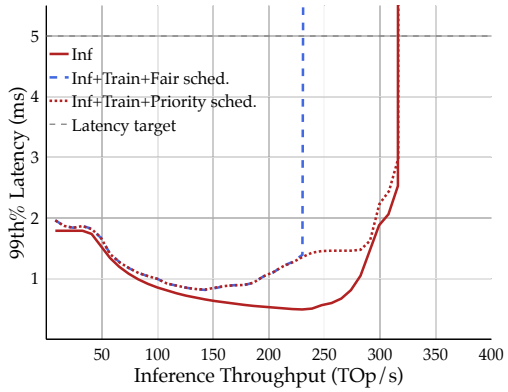


Figure 10: Equinox's inference tail latency against its throughput. *Inf*, *Inf+Train+Fair sched.*, and *Inf+Train+Priority sched.* indicate configurations without training, with training and a fair-share scheduler, and with training and a priority scheduler respectively.

Finally, we show that the request and instruction dispatchers consume less than 1% of the area and power, confirming that Equinox's scheduling mechanisms incur negligible overhead.

Scheduling. 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 10 shows Equinox's 99% latency against its throughput while performing inference under two scheduling policies: fair-share scheduling (Inf+Train+Fair sched.) and a policy that at high loads schedules inference requests only (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 requests. 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. Due to the high rate of instruction issue in Equinox, a software scheduler has to operate at a batch granularity, which 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 to maintain the latency target, preventing Equinox from serving training requests altogether.

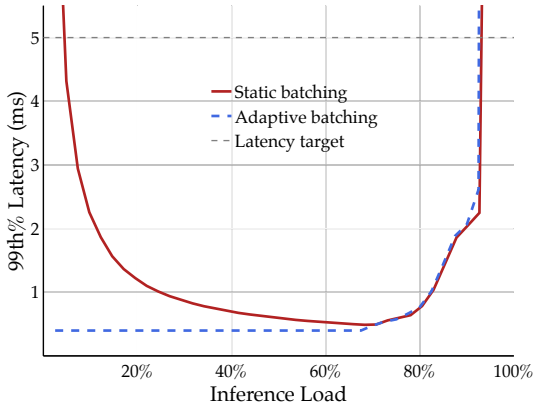
Adaptive Batching. To quantify adaptive batching's impact on the tail latency of inference requests, we compare the 99th% latency of $Equinox_{500\mu s}$ serving inference requests with static and adaptive batching policies at various loads. Figure 11a shows that static batching performs poorly at low 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 11b shows the effect of this threshold over inference latency. We vary the threshold values from 2× to 10× the service time. Increasing the threshold forces the accelerator to wait for longer to form batches, leading to higher 99th% latency, as shown in the figure. We also observe that long waits are infrequent because with even relatively high thresholds (i.e., 10×), we observe that less than 1% of the issued batches are incomplete.

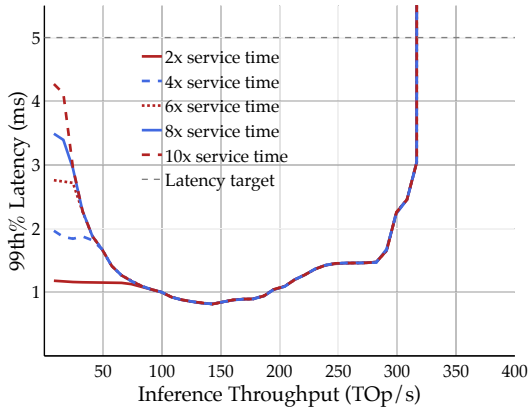
Finally, Figure 11c shows the training throughput obtained while varying the adaptive batching threshold. Increasing the threshold beyond 2× the service time leads to a modest increase in training throughput without violating the latency goals. However, as we increase the threshold, batch scheduling incurs 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 the latency goals, we picked this threshold for all experiments and workloads.

7 RELATED WORK

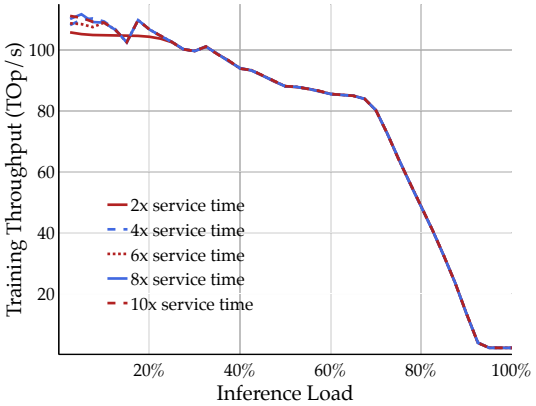
DNN Accelerators are built for low latency inference services and employ low precision arithmetic for efficiency. However, their arithmetic encoding prevents them from being used for training services. Microsoft's Brainwave [16] relies on FPGAs to accelerate DNNs at low latency. Although we argue that 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 to 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 [22], in contrast, is an ASIC design that



(a) Tail latency vs. load with static and adaptive batching



(b) Tail latency vs. inference throughput



(c) Training throughput as a function of inference load.

Figure 11: Equinox's adaptive batching policy impact on latency and training throughput. "$X\times$ service time" indicates that the batching mechanism waits for $X\times$ the workload service time before issuing an incomplete batch.

leverages batching to cope with the memory bottleneck. Neither Brainwave nor TPU have any mechanisms to guarantee latency SLOs in 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 [4] and Google's TPUv2 [2] and TPUv3 [5]. All these accelerators feature high bandwidth memory.

Training Arithmetic. While inference has been known to tolerate low precision fixed-point arithmetic for a long time, training algorithms started adopting more efficient arithmetic encodings only recently. While early works [23] used single-precision floating point, DNN users quickly learned that DNNs are tolerant to low precision arithmetic. As such, most recent accelerators use narrower arithmetic, like bfloat16 [6]. We argue that this trend will continue, with recent work introducing several options for even more efficient training arithmetic [14, 24, 35]. These innovations allow for training arithmetic with efficiency in par with inference arithmetic, acting as an enabler for Equinox. More efficient and dense arithmetic also leads to a smaller fraction of the accelerator power being spent on ALUs, emphasizing of accelerator design on minimizing data movement.

Software Schedulers. 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 a more autonomous DNN accelerator, capable of running entire requests instead of individual instructions. This model enables a Brainwave-like accelerator, reducing latency further.

Task Co-location. Prior work [11, 25] co-locates latency-critical and best-effort tasks on online servers. Additionally, Facebook [18] co-locates training and inference services on general-purpose CPUs during periods of low load. We also leverage a best-effort service to improve the utilization of compute resources executing a latency-critical service. However, the challenges in co-locating inference and training in accelerators are fundamentally different from the challenges of co-locating in online servers. The biggest challenge

in online servers is identifying the resources contended by each workload as 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 a latency-aware mechanism to specialized accelerators.

Finally, EDGE [20] introduces an event-driven GPU execution model that provides the basic mechanisms needed for simultaneous execution of inference and training requests, providing the first step towards a GPU capable of meeting the latency constraints of inference services while performing training. Using EDGE, a GPU is capable of quickly preempting training requests during inference load spikes. The next step would be to provide DNN-aware resource scheduling capabilities in GPUs.

8 CONCLUSION

DNN inference accelerators face a low average load due to service demand variability. Unfortunately, traditional inference accelerators do not have the mechanisms to expose inference idle cycles to training workloads. In this paper, we introduce Equinox, an inference accelerator that piggybacks training services to reclaim inference idle cycles. Equinox reconciles the conflicting requirements of training and inference and achieves near-optimal inference throughput while exposing inference idle cycles to training services. We show that, with a 500μs inference service time constraint, Equinox achieves 6.67× higher throughput than a latency-optimal accelerator. Equinox also achieves a training throughput that is 78% of the throughput of a dedicated training accelerator. Finally, we show that the mechanisms introduced by Equinox lead to an area and power overheads of less than 1%, while its numeric encoding incurs 13% power and 4% area overhead compared to a fixed-point accelerator.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers and the members of PARSA at EPFL for their precious comments and feedback. We would also like to thank Simla Harma for conducting HBFP convergence experiments and sharing the results with us. This work has been partially funded by a Microsoft JRC Research Grant, a Qualcomm PhD Fellowship, the *CE-EuroLab-4-HPC* project, and the following grants: "Memory-Centric Server Architecture for Datacenters" and "Hardware/Software Co-Design for In-Memory Services" from the Swiss National Science Foundation (SNSF).

REFERENCES

[1] 2010. NVIDIA T4 Tensor Core GPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/t4-tensor-core-datasheet.pdf. Accessed: 2019-01-07.
[2] 2017. Cloud TPU. https://cloud.google.com/tpu. Accessed: 2018-01-31.
[3] 2017. Introduction to the IPU architecture. https://www.graphcore.ai/nips2017_ presentations. Accessed: 2019-08-06.
[4] 2018. NVIDIA Volta V100 GPU Accelerator. https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf. Accessed: 2018-01-31.
[5] 2018. Tearing Apart Google's TPU 3.0 AI coprocessor. https://www.nextplatform.com/2018/05/10/tearing-apart-googles-tpu-3-0-ai-coprocessor. Accessed: 2018-05-15.
[6] Martn Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*. 265–283.
[7] Luiz Andr Barroso, Urs Hlzle, and Parthasarathy Ranganathan. 2018. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Morgan & Claypool Publishers.
[8] Daniel Crankshaw, Xin Wang 0066, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*. 613–627.
[9] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. 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)*. 35–48.
[10] William Dally. 2015. High Performance Hardware for Machine Learning. https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf. Accessed: 2018-01-31.
[11] Christina Delimitrou and Christos Kozyrakis. 2014. 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)*. 127–144.
[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019*. 4171–4186.
[13] Mario Drumond. 2020. *ColTraIn: Co-located DNN training and inference*. Ph.D. Dissertation. EPFL.
[14] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. 2018. Training DNNs with Hybrid Block Floating Point. In *Proceedings of the Thirty-second Conference on Neural Information Processing Systems (NeurIPS)*. 451–461.
[15] Hadi Esmaeilzadeh, Emily R. Blem, Rene St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*. 365–376.
[16] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*. 1–14.
[17] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the Thirty-second International Conference on Machine Learning (ICML)*. 1737–1746.
[18] Kim M. Hazelwood, Sarah Bird, David M. Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *Proceedings of the 24th IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 620–629.
[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.
[20] Tayler Hicklin Hetherington, Maria Lubeznov, Deval Shah, and Tor M. Aamodt. 2019. EDGE: Event-Driven GPU Execution. In *Proceedings of the 28th International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 337–353.
[21] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM* (2020), 67–78.
[22] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*. 1–12.
[23] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. *Technical report, University of Toronto* (2009).

[24] Urs Kster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William Constable, Oguz Elibol, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, and Naveen Rao. 2017. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In *Proceedings of the Thirty-first Conference on Neural Information Processing Systems (NIPS)*. 1742–1752.

[25] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*. 450–462.

[26] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2007. 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)*. 3–14.

[27] Sharan Narang and Greg Diamos. 2017. Baidu DeepBench. https://github.com/baidu-research/DeepBench.

[28] Ali Pahlevan, Javier Picorel, Arash Pourhabibi Zarandi, Davide Rossi, Marina Zapater, Andrea Bartolini, Pablo Garca Del Valle, David Atienza, Luca Benini, and Babak Falsafi. 2016. Towards near-threshold server processors. In *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 7–12.

[29] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. 325–341.

[30] Bita Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, Alessandro Forin, Haishan Zhu, Taesik Na, Prerak Patel, Shuai Che, Lok Chand Koppaka, Xia Song, Subhojit Som, Kaustav Das, Saurabh Tiwary, Steve Reinhardt, Sitaram Lanka, Eric Chung, and Doug Burger. 2020. Pushing the Limits of Narrow Precision Inferencing at Cloud Scale with Microsoft Floating Point . *NeurIPS 2020* vol. 33, no. 4, pp. 100-107. (2020).

[31] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. 2014. ImageNet Large Scale Visual Recognition Challenge. *CoRR* abs/1409.0575 (2014).

[32] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[33] Kevin Tran. 2016. Start Your HBM/2.5 D Design Today.

[34] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Katie Baynes, Aamer Jaleel, and Bruce Jacob. 2005. DRAMsim: A memory-system simulator. *Computer Architecture News* vol. 33, no. 4, pp. 100-107. (2005). https://doi.org/10.1145/1105734.1105748

[35] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. 2018. Training Deep Neural Networks with 8-bit Floating Point Numbers. In *Proceedings of the Thirty-second Conference on Neural Information Processing Systems (NeurIPS)*. 7686–7695.

[36] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. 2018. Benchmarking and Analyzing Deep Neural Network Training. In *Proceedings of the 2018 IEEE International Symposium on Workload Characterization*. 88–100.