Accelerating Generalized Linear Models with MLWeaving: A One-Size-Fits-All System for Any-Precision Learning (Technical Report)

Zeke Wang, Kaan Kara, Hantian Zhang, Gustavo Alonso, Onur Mutlu, Ce Zhang
Systems Group, Department of Computer Science
ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

ABSTRACT

Learning from the data stored in a database is an important function increasingly available in relational engines. Methods using lower precision input data are of special interest given their overall higher efficiency. However, in databases, these methods have a hidden cost: the quantization of the real value into a smaller number is an expensive step. To address this issue, we present MLWeaving, a data structure and hardware acceleration technique intended to speed up learning of generalized linear models over low precision data. MLWeaving provides a compact in-memory representation that enables the retrieval of data at any level of precision. MLWeaving also provides a highly efficient implementation of stochastic gradient descent on FPGAs and enables the dynamic tuning of precision, instead of using a fixed precision level during learning. Experimental results show that MLWeaving converges up to 16× faster than low-precision implementations of first-order methods on CPUs.

PVLDB Reference Format:
Zeke Wang, Kaan Kara, Hantian Zhang, Gustavo Alonso, Onur Mutlu, Ce Zhang. Accelerating Generalized Linear Models with MLWeaving: A One-Size-Fits-All System for Any-Precision Learning. *PVLDB*, 12(7): 807-821, 2019.
DOI: https://doi.org/10.14778/3317315.3317322

1. INTRODUCTION

Database engines have started to provide support for training machine learning (ML) models over relational data (e.g., MADlib [32]). Generalized linear models, such as support vector machines and logistic regression solved using stochastic gradient descent (SGD), are among the most common approaches used within databases. Although useful in many applications, these models are expensive to compute and, thus, there is a great deal of activity exploring ways to reduce the overhead of training. One promising approach is quantization applied to either the values in the model [15, 31, 92] or, our focus in this paper, lower precision input data. Using lower precision on the input data reduces the amount of data accessed during training, thereby shortening training times [107].

Proceedings of the VLDB Endowment, Vol. 12, No. 7
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3317315.3317322

From a database perspective, these approaches are interesting as they alleviate the memory bottleneck. However, existing approaches quantizing the input data implicitly assume that either the data is always available in the correct precision (e.g., for 32-bit fixed-point numbers, 32 copies of the original data with precision ranging from 1 to 32 bits are required) or the data is pre-quantized at its source (e.g., memory or disk) [17, 45]. A further challenge is that the correct precision needed by each application is not always known in advance and varies depending on the statistical characteristics of the data. The overhead for machine learning that uses quantized datasets is even bigger when hardware accelerators are used, since existing solutions require 1) a different microarchitecture for each precision level and 2) a separate copy of the quantized data at the right level of precision [45].

To address these issues, we have designed MLWeaving, a novel *end-to-end* system enabling *any-precision* learning of generalized linear models in database engines. We implement a prototype of the MLWeaving system in DoppioDB, a column store database augmented with FPGA acceleration [87], as a first step to explore how database storage formats can be combined with the requirements of machine learning algorithms. MLWeaving has two key innovations that make the learning process on FPGAs quantization-friendly (**C1**) and synchronous (**C2**).

C1: Flexible Memory Layout and Hardware Implementation. MLWeaving combines 1) a memory layout supporting the efficient retrieval of the input data at *any level of precision* and 2) an FPGA-based design providing hardware acceleration to speed up SGD regardless of the precision used. The key idea behind MLWeaving is a transposed memory layout where different bits of a given value are stored separately. SGD is evaluated *sample-at-a-time*, i.e., all the features of the sample are read before the gradient is computed. At full precision, reading a row corresponding to a data point accesses all the features for that sample. MLWeaving vertically partitions each data point (a row in a table) at the bit level so that the first bit of all features of a data point are stored consecutively, then the second bit, etc. This provides two benefits. First, the number of memory accesses needed to read a value is proportional to the precision used. Lower precision leads to fewer memory accesses. Second, the format allows the serialization of the data into a hardware accelerator in the form of a bit stream, improving the memory bandwidth utilization.

We show that the bit stream format can be used to compute the gradient using bit-serial multipliers (i.e., process all the first bits in the first cycle, all the second bits in the second cycle, etc. [33, 43, 66, 88, 93, 101, 104]) a more efficient approach than that used in existing systems [17, 45]. The resulting design employs only one third of the resources used in previous solutions and allows higher frequency, doubling the rate at which data can be processed on an

FPGA accelerator. Furthermore, our approach supports dynamic selection of the level of precision by simply reading more or fewer bits when processing the data in each epoch of the SGD algorithm.

C2: Efficient Synchronous Execution. Due to the sequential nature of the SGD algorithm, a common approach to parallelizing it on modern CPUs/GPUs is to perform asynchronous updates to the model. By doing so, different cores do not need to acquire expensive locks for every gradient calculation [71, 108] (thus better hardware efficiency). This approach is guaranteed to converge under certain (often mild) conditions, but it could converge slower than the synchronous approach (thus worse statistical efficiency). In MLWeaving, we show how synchronous SGD on an FPGA can be made almost as efficient, in terms of hardware efficiency, as asynchronous SGD on an FPGA. Meanwhile, MLWeaving often requires fewer epochs to converge than its asynchronous *CPU* counterpart. As a result, MLWeaving converges faster, in terms of end-to-end performance, than its asynchronous CPU counterpart.

We have implemented MLWeaving on an Intel Arria 10 FPGA using Intel's Xeon+FPGA platform (HARP) [28] where the FPGA is integrated in the same package as a Xeon multicore CPU. Experimental results show that MLWeaving achieves up to 16× performance improvement over the state-of-the-art low-precision first-order CPU implementation (Table 8).

2. BACKGROUND

MLWeaving combines ideas from several areas: databases, machine learning, and computer architecture. In this section we introduce the necessary background to understand the overall design. Table 1 summarizes the notation used throughout the paper.

Table 1: Notation used in the paper

Term	Definition	Range
N	Number of samples for training	Input
M	Number of features in the sample	Input
B	Mini batch size	Hyper-param
λ	Learning rate	Hyper-param
s	Number of bits used at runtime	Input
$\vec{x}$	Model, i.e., a vector of parameters	Output
S	Maximum number of bits of the quantized value	32
T_s	s-bit fixed-point table (i.e., training dataset)	$1 \leq s \leq S$
$a^{[i]}$	i-th bit of fixed-point value a, $1 \leq i \leq S$	0 or 1
$Q_s(A)$	s-bit quantized value of full-precision A	$1 \leq s \leq S$
$\#CL$	Number of bits of a cache line	512 bits
$\#Freq$	Frequency of the SGD hardware design	400 MHz
$\#M_{max}$	Maximum dimensions of supported model	32K
$\#Bank$	Number of banks implemented in hardware	8

2.1 Normalization and Quantization

For all ML algorithms, raw data must be converted into a suitable format before the learning process. In MLWeaving, data is stored in the format resulting from a normalization and a quantization step.

Normalization reduces the range of values without affecting the overall result of the training. Without loss of generality, MLWeaving uses the scaled range [0, 1]. Accordingly, for each column, the original value f is normalized to the value $\tilde{f}$ (Equation 1):

$$\tilde{f} = \frac{f - f_{min}}{f_{max} - f_{min}}, \tag{1}$$

where f_{min} and f_{max} are the maximum and minimum values in the column, respectively. An advantage of learning inside the database engine is that normalization can be accomplished using either the meta-data available on a relational table or computed using standard SQL (min, max).

Quantization happens over the normalized dataset. It involves converting a full-precision floating-point value $\tilde{f}$ into a lower-precision fixed-point value. Specifically, our goal is to construct a function $Q_s : \mathbb{R}^d \mapsto \mathbb{F}^d$, a *deterministic quantization function* that maps a floating-point value to an s-bit fixed-point representation $Q_s(a)$.The quantization process is conducted in two steps.

1, Floating Point to Fixed Point Conversion. The first step is to generate a *full-precision fixed-point table* T_S, where S is the maximum number of bits of the quantized value in the table. The new fixed-point value ($\tilde{a}$) in T_S is calculated to be $\tilde{f}$ multiplied by $2^S - 1$ (instead of 2^S),[1] since the range of each floating-point value in the normalized table is between 0 and 1. Therefore, the larger the S value, the higher the precision. In our FPGA design, the fixed-point value $\tilde{a}$ is interpreted as:

$$\tilde{a} = \sum_{i=1}^{S} \tilde{a}^{[i]} \times 2^{-i}, \tag{2}$$

where $\tilde{a}^{[i]}$ represents the i-th bit of $\tilde{a}$, 0 or 1. The first bit (i = 1) is the most significant bit. Figure 1(a) shows an example of a full-precision fixed-point table T_S where $S = 4$. The quantized value ($\tilde{a} = 1010_2$) of the seventh column and the first row[2] represents the value of $\tilde{f}$: $1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} = 0.625$.

2, Fixed Point Quantization. The second step of Q_s is to quantize the fixed point table T_S into the desired level of precision. This step is performed by keeping the s most significant bits of T_S. Figures 1(b-d) illustrate the quantized 3-bit, 2-bit and 1-bit fixed-point tables for the original data shown in Figure 1(a). For example, the full-precision fixed point representation 1010_2 in T_4 is 101_2, 10_2, and 1_2 for 3, 2, and 1 bit precision, respectively.

1st row	ABCD	EFGH	IJKL	MNOP	RSTU	VXYZ	1010	0101
2nd row	abcd	efgh	ijkl	mnop	rstu	vxyz	1100	0011

(a) Full-precision fixed-point table T_S = T_4

1st row	ABC	EFG	IJK	MN	RST	VXY	101	010
2nd row	abc	efg	ijk	mno	rst	vxy	110	001

(b) 3-bit fixed-point table T_3

1st row	AB	EF	IJ	M	RS	VX	10	01
2nd row	ab	ef	ij	mn	rs	vx	11	00

(c) 2-bit fixed-point table T_2

A	E	I	M	R	V	1	0
a	e	i	m	r	v	1	0

(d) 1-bit fixed-point table T_1

Figure 1: Four fixed-point quantized tables (T_4, T_3, T_2, T_1) containing 2 data points with 8 features. Each table has two rows and eight columns, and each element has four bits, where the symbol (e.g., A-Z, a-z) is binary, 0 or 1.

2.2 Low-precision SGD

SGD is a popular algorithm to train generalized linear models. Given a relation A, the full precision SGD solves the following optimization problem:

$$\underset{\vec{x}}{\text{minimize}} : \frac{1}{N} \sum_{i=1}^{N} f(\vec{x} \cdot \vec{a_i}, b_i),$$

where $\vec{a_i}$ is one row in the input relation, b_i is the corresponding training label, $\vec{x}$ is the model, and $f(-)$ is a loss function. SGD solves this problem by iteratively scanning the input relation A – for each row $\vec{a_i}$, it calculates the gradient with respect to $\vec{x}$, and updates the model. Each pass over the input data is called an *epoch*. SGD usually runs for multiple epochs until convergence.

[1] $\tilde{f} = 1.0$ leads to $\tilde{a} = 2^S - 1$.

[2] We use three pairs of words (table, training dataset), (row, sample) and (column, feature) interchangeably within each pair.

Training SGD over low precision representation of the input data using a generalized linear model targets the following function:

$$\underset{\vec{x}}{\text{minimize}} : \frac{1}{N}\sum_{i=1}^{N} f(\vec{x} \cdot Q_s(\vec{a_i}), b_i),$$

where the i-th sample consists of a vector of quantized s-bit values ($Q_s(\vec{a_i}) \in \mathbb{F}^{1\times M}$).

Low-Precision Mini-batch SGD on FPGAs. One variant of SGD that is popularly implemented in many systems is *mini-batch SGD* [57] – instead of calculating the gradient using a single sample, mini-batch SGD uses multiple samples, called a "mini-batch", to calculate the *average* gradient. Mini-batch SGD can be easier to accelerate because all samples in a mini-batch share the same model and thus can be processed independently in parallel. For applications such as distributed deep learning for image classification, mini-batch SGD, instead of the standard SGD, is the *de facto* training algorithm. When deploying SGDs on FPGAs, we also use mini-batch SGD [45].[3]

Algorithm 1 illustrates the flow of low-precision mini-batch SGD, which is iteratively evaluated in E epochs (Line 1). In each epoch, the entire low-precision training dataset is scanned, one mini-batch of B samples per iteration (Line 2). Inside a mini-batch, we initialize the average gradient ($\vec{g}$) to zero at the beginning (Line 3), and compute the average gradient of this mini-batch (Lines 4-13).[4] Each sample is processed as follows. First, we compute the dot product of two vectors: the i-th low-precision sample $Q_s(\vec{a_i})$ and the full-precision model[5] $\vec{x}$ (Line 8). Its output a_dot_x is a full-precision scalar value. Second, we compute the scaling value $scale$, based on the derivative (i.e., df) of the given loss function (Line 9). For different learning algorithms, we only need to modify this function to compute the scaling value, while keeping the other parts unchanged. Third, the gradient ($g_{\vec{j+k}}$) of this sample is computed (Line 10). Fourth, $g_{\vec{j+k}}$ is accumulated into the gradient of this mini-batch (Line 11). Fifth, the model ($\vec{x}$) is updated with the average gradient $\vec{g}$ (Line 14).

Performance Metrics. The end-to-end performance of SGD can be measured as the time that it takes to achieve the target loss. This can be further decomposed into two metrics [106] that we will use throughout this paper. *Hardware efficiency* measures the time that SGD requires to finish one epoch. *Statistical efficiency* represents the number of epochs that SGD requires to converge.

2.3 Hardware Acceleration

Most ML algorithms are known to be compute- and data-intensive. Not surprisingly, in recent years, we have seen a significant increase in the number of specialized hardware solutions for ML, from GPUs to FPGAs to specialized processors such as TPUs [41]. In this paper, we focus on FPGAs since they provide a higher degree of versatility in exploring possible algorithms and designs, something important in an area that is evolving as quickly as ML. Since the early pioneer work exploring the use of FPGAs on databases [68, 69, 70], a growing number of database operations accelerated with FPGAs have been proposed [36, 37, 40, 47, 86, 95, 96, 97, 103]. FPGAs are also increasingly available, specially in cloud platforms such as Microsoft's Catapult [8] and Brainwave [15, 26] projects, or Amazon F1 instances, making them a suitable target for hardware acceleration.

[3]We explain more in Section 4.

[4]In this context, we assume that $Bank$ is 1 for ease of understanding such that one sample is processed at a time. $Bank$ is 8 in MLWeaving that addresses the constraints encountered when deploying any-precision SGD on an FPGA (Subsection 4.3).

[5]In this paper, we do not consider the quantization of the model.

Algorithm 1: LOW-PRECISION MINI-BATCH SGD

Input : N: number of samples,
E: number of epochs,
γ: learning rate,
$Q_s(\vec{a_i})$: s-bit quantized data set of the i-th sample, $\in \mathbb{F}^{1\times M}$,
b_i: label value of the n-th sample, $b_i \in \mathbb{R}^1$.
Output : $\vec{x}$: model with a set of parameters.

```
/* Evaluate the e-th epoch.                                   */
1  for e = 1 to E do
       /* Mini-batch processing                                */
2      for (i = 0; i < N; i+ = B) do
           /* Zero the gradient of this mini-batch             */
3          g⃗ = 0;
4          for (j = 0; j < B; j+ = #Bank) do
               /* Multi-bank processing                        */
5              #pragma parallel in hardware
6              for (k = 0; k < #Bank; k + +) do
7                  int32 t = i + j + k;
                   /* Dot product                              */
8                  int32 a_dot_x = Q_s(a_t⃗) · x⃗;
                   /* Serial part                              */
9                  int32 scale = γ × df(a_dot_x, b_{i+j+k});
                   /* Gradient computation                     */
10                 g_{j+k}⃗ = scale × Q_s(a_t⃗);
                   /* Gradient accumulation                    */
11                 g⃗ = g⃗ + g_{j+k}⃗;
12             end
13         end
           /* Model update                                     */
14         x⃗ = x⃗ − g⃗/B;
15     end
16 end
```

The degree of microarchitectural freedom in designing an algorithm is much higher on an FPGA than in software in general as it is possible to design specialized compute units from scratch and tailor the hardware to the application at hand [91]. The limiting constraints in FPGA designs are *meeting timing* (how fast the hardware design can be clocked) and *resource usage* as the FPGA is a physical device and, at a certain point, there are no more gates available to implement additional function. Often, designs have to balance one against the other and strive for an efficient trade-off between parallelism on the one hand and FPGA resource usage on the other hand.

3. SYSTEM OVERVIEW

Target Platform. The target platform (Figure 2) for this work is the 2nd generation Intel Xeon+FPGA [28], combining a Broadwell 14-core CPU E5-2680v4 with an Arria 10 FPGA. The FPGA has cache-coherent access to the main memory (DDR4: 64GB) of the CPU via 1 QPI link and 2 PCIe links, resulting in around 20 GB/s combined read and write bandwidth. We perform all our experiments on this machine.

Database Integration. We integrate our FPGA-accelerated training designs into DoppioDB [87], an open source solution for FPGA-accelerated databases, based on MonetDB [35]. DoppioDB

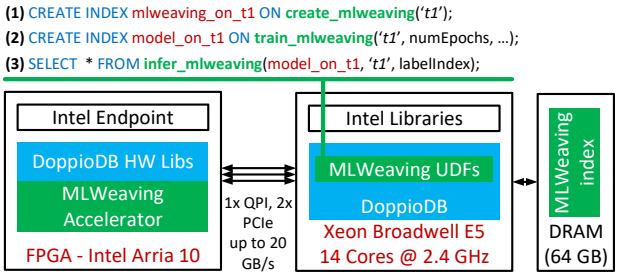


Figure 2: Target platform (Intel Xeon+FPGA Gen2), overview of DoppioDB on this platform and sample queries with MLWeaving UDFs in DoppioDB.

enables easy integration of FPGA-based accelerators through a set of software and hardware libraries [73]. The hardware libraries expose a native memory interface, via which the FPGA accelerators use to access the main memory of the host CPU. On the software side, FPGA accelerators can be started and monitored as separate threads. These FPGA threads run in parallel to software threads.

We implement three User Defined Functions (UDFs) in DoppioDB, as shown in Figure 2. (1) The first UDF is to initiate the creation of the *MLWeaving index* that is associated with a certain table and kept internal in the database. (2) The second UDF is to initiate training using MLWeaving. Under the hood, DoppioDB will look for the MLWeaving index that belongs to the table given by this UDF. If found, the FPGA thread is started. It uses the MLWeaving index to train a model which then gets transferred to the main memory. (3) The third UDF is to perform inference on tuples using the model that has been trained before. Similar to the MLWeaving index, the trained model is also associated with a certain table and will check during query execution if inference can be performed, conditioned on a model having been trained before.

MLWeaving combined with DoppioDB offers users a SQL front-end for learning models from relational data in a seamless manner as the data selection and transformation needed for the learning can be done using SQL and without incurring expensive data transfers in and out of the database.

4. MLWEAVING DESIGN

MLWeaving involves many aspects that interact in a tight manner. To make it easier to understand how it works, we develop MLWeaving in three stages: "Quantized" (Subsection 4.1), "BWeaving" (Subsection 4.2) and "MLWeaving" (Subsection 4.3). Table 2 summarizes the comparison results.

Table 2: Comparison of three approaches. $\#CL$ is 512. "n" represents any positive integer.

Hardware Metrics	Quantized	BWeaving	MLWeaving
Supported precision levels	1	32	32
Required memory layouts	Per any s-bit	1	1
Bitwidth (bits) of model $\vec{x}$	$\lfloor \frac{512}{s} \rfloor \times 32$	16K	2K
Number of banks, $\#Bank$	1	1	8
Mini-batch size, B	n	n	8*n

4.1 Quantized

Understanding of MLWeaving requires understanding the interplay between the data representation and the processing required by the accelerator. In the case of SGD, the most important components in the design are the *computing circuits*, *how the input data is transferred from the external memory*, and *how the model is accessed on the FPGA's local memory*. In the following, we discuss the hardware properties of Quantized [45].

Computing Circuits. Quantized employs a fixed circuit for each precision level. It, thus, requires the data to be available at the corresponding precision. 'Quantized" processes features using fixed-point bit-parallel multipliers and can only support one precision level per hardware design.

Memory Layouts. In the Xeon+FPGA platform [73] used to implement MLWeaving, the FPGA has cache coherent access to the main memory of the CPU. The data arrives to the FPGA in the form of cache lines, *#CL* (512 bits per cache line). Efficiency is achieved by processing in parallel as many elements within that cache line as possible. How many elements are within a cache line depends on how the data is stored in memory. Quantized stores each data point at a given precision in a consecutive manner. If the precision is 16 bits, every cache line brings in 512/16 data points (i.e., features or columns) to be processed. Because the value for each data point arrives as a whole, the multipliers needed for the gradient computation are based on the 16-bits used to represent a value. Such 16-bit wide multipliers are complex and take up significant hardware space. Besides, Quantized requires one memory layout for each precision level.

Accessing the Model on FPGAs. The SGD hardware reads not only the input data from the external memory but also the corresponding entry in the model on the FPGA. In particular, the SGD hardware that operates on each dimension needs to read a value from the model as a whole. Since the SGD hardware processes a great number of dimensions concurrently, an additional design factor we should consider is how to efficiently access the corresponding values of the model. This is referred to the *bitwidth of the model*. The bitwidth is a very important parameter in an FPGA design because it affects the complexity of the interconnects and the way data has to move from the model storage to the computing units. The higher the bitwidth, the more complex the design and, thus, the higher the probability that it will not meet timing at higher clock rates. Quantized processes data points as a whole, so its model bitwidth is $\lfloor \frac{512}{s} \rfloor \times 32$ bits, where the precision of the model is 32 bits and s is the precision level of input data.

4.2 BWeaving

BWeaving extends Quantized [45] such that both memory access and computation time linearly decrease with a lower number of bits used for ML training. To do so, we propose a bit-serial memory layout (Subsection 4.2.1) and a specialized hardware design powered by bit-serial multipliers (Subsection 4.2.2).

4.2.1 BWeaving Memory Layout (Software)

The BWeaving memory layout is based on ideas proposed by BitWeaving [59,60]. In a nutshell, we transpose the training dataset to allow the runtime selection of the precision level and to reduce memory traffic for lower precision levels. Contrary to BitWeaving that performs the weaving on one column, BWeaving transposes each row (a sample) to preserve access locality, since the training dataset is accessed by SGD in a *sample-at-a-time* manner.

The transposition of the data used in BWeaving is shown in Figure 3. M S-bit features of a sample are transposed into S M-bit words (where M is 8 and S is 4 in the example shown). Inside the first row, eight 4-bit features are transposed into four 8-bit words. The first bits (i.e., AEIMRV10) of the first sample are stored in an 8-bit word. The second word BFJNSX01, which contains the second bits, is stored next to the first word, and so on. Between rows, the second row is stored consecutively to the first row.

Under the BWeaving memory layout, data at any level of precision can be retrieved by following a different access pattern over the same data structure. In Figure 3(b), we show the accesses needed to retrieve the data set at a precision of 3 bits. The first bits in the first row are accessed, followed by the second and third bits in the first row. Then, the access jumps to the first bits of the second row, skipping the fourth bits of each row.

Instantiation of Memory Layout. Table 3 shows an example of a BWeaving memory layout with $\#CL$ = 512 and M (number of features) = 2048. For instance, the first memory slot (with index = 0) is populated with the first bits of the first 512 features of the first sample, while the second memory slot (with index = 1) is populated with the second bits of the first 512 features of the first sample. If M is not a multiple of 512, we use padding to fit a 512-bit boundary such that we can easily retrieve bits.

4.2.2 BWeaving Arithmetic (Hardware)

Using a bit-serial multiplier operating on the data one bit per cycle [33,43,66,88,93,101,104], we design the BWeaving arithmetic

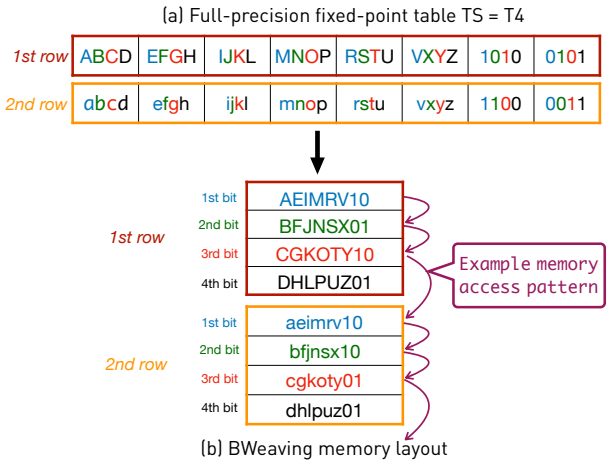


Figure 3: Full-precision fixed-point memory layout (a) converted into the BWeaving memory layout (b). Each symbol (e.g., A-Z, a-z) in the table is binary, 0 or 1. The BWeaving memory layout enables the flexible selection of precision in memory. As an example, we show the "memory access pattern" with a 3-bit precision (s = 3).

Table 3: BWeaving memory layout: $x\_y{:}z\_w$ denotes the w-th bits of 512 features (from z-th to y-th) in the x-th sample.

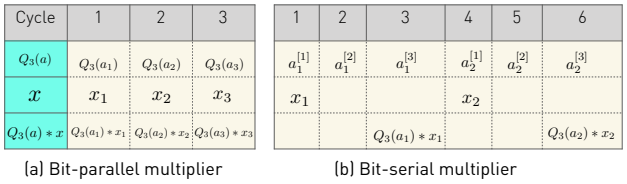
Description	Index	Content ($\#CL$=512 bits)
First 512 features of the first sample	0	0_511:0_0
	1	0_511:0_1
	2	0_511:0_2
	3	0_511:0_3
	…	…
	31	0_511:0_31
Second 512 features of the first sample	32	0_1023:512_0
	33	0_1023:512_1
	34	0_1023:512_2
	…	…
…	…	…
First 512 features of the second sample	128	1_511:0_0
	129	1_511:0_1
	…	…

that provides bit-level flexibility (i.e., supporting any precision with a single hardware design) while maintaining a processing rate of a cache line per clock cycle. We now present the difference between bit-serial and bit-parallel multipliers, followed by the SGD hardware design powered by a bit-serial multiplier.

Bit-serial Multiplier vs. Bit-parallel Multiplier. Figure 4 illustrates the difference between a bit-parallel multiplier and a bit-serial multiplier with one example multiplying 3-bit $Q_3(a)$ (low-precision) by 4-bit x (full-precision).

In a bit-parallel multiplier, each clock cycle can enable the multiplication of two numbers, in this case the quantized input $Q_3(a)$ and the corresponding value from the model x (Figure 4a). This is the type of multiplier used in conventional CPUs and also previous specialized hardware solutions [45].

In a bit-serial multiplier, one multiplication result is produced every three cycles, a bit of $Q_3(a)$ per cycle. After $Q_3(a)$ is replaced with $\sum_{i=1}^{3} a^{[i]} \times 2^{-i}$ (Equation 2), the product $Q_3(a) \times x$ (Equation 3) is computed to be the sum of the product of $a^{[i]}$ and $(x \ggg i)$, where the binary value $a^{[i]}$ represents the i-th bit of a (0 or 1), $\ggg$ means signed right shift, and i is from 1 to 3. In terms of cycles, the product $Q_3(a) \times x$ is set to be $a^{[1]} \times (x \ggg 1)$ in the first cycle, $a^{[2]} \times (x \ggg 2)$ is added to the product in the second cycle, and $a^{[3]} \times (x \ggg 3)$ is added in the third cycle. The advantage of bit-serial multiplier is that shift-and-add operations are enough



(a) Bit-parallel multiplier (b) Bit-serial multiplier

Figure 4: Multipliers: bit-parallel (a) vs. bit-serial (b). Bit-parallel multiplier produces one multiplication result per cycle, while bit-serial multiplier produces every three cycles, e.g., on cycle 3 or 6.

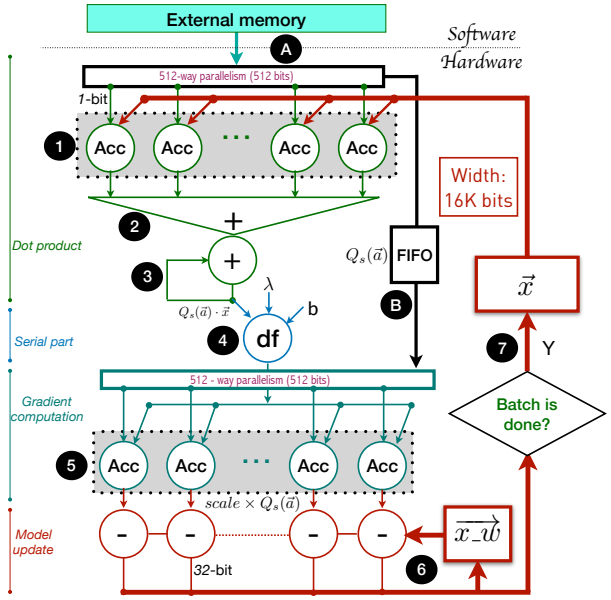


Figure 5: Fully pipelined BWeaving hardware design with 1 bank ($\#Bank$ = 1), according to Algorithm 1.

for the calculation, considerably simplifying the design. The disadvantage is that multiple cycles are needed to complete an operation due to its inherent bit-serial nature.

$$Q_3(a) \times x = x \times \sum_{i=1}^{3} a^{[i]} \times 2^{-i} = \sum_{i=1}^{3} a^{[i]} \times (x \ggg i) \quad (3)$$

Hardware Design of BWeaving. The goal of BWeaving arithmetic is to consume 512-bit data per cycle from the BWeaving memory layout described above. We implement the fully-pipelined hardware design for the BWeaving arithmetic in Figure 5, according to Algorithm 1 with $\#Bank$ = 1. The design has four stages, each of which occupies unique hardware resources of an FPGA. In the following, we discuss the detailed implementation for each stage.

In the "dot product" stage, 512 bit-serial multipliers are instantiated (❶) to consume the 512-bit data stream (Ⓐ) from the BWeaving memory layout per cycle, where each bit-serial multiplier can handle a bit from a feature per cycle. At the same time, the data stream is also fed to the "FIFO" (Ⓑ). In the first cycle, the first bit of each of the first 512 features from a sample enter the pipeline in a lock-step manner, and 512 values are read from the *architectural model* (labelled $\vec{x}$ in Figure 5), where the architectural model stores the committed state of the model $\vec{x}$ to preserve the semantics of mini-batch SGD. After processing s bits of the first 512 features (where s is the precision level used in the execution), 512 multiplication results are passed to the fully-pipelined adder tree (❷), whose depth is $log_2(512)$. The output of the adder tree is connected to an accumulator (❸), which aggregates $\lceil \frac{M}{512} \rceil$ valid results from

the adder tree and then computes the final result ($Q_s(\vec{a}) \cdot \vec{x}$) of the dot product,[6] where M is number of features in the sample.

In the "serial part" stage, the scaling value *scale* (❹) is computed to be $\lambda \times df(Q_s(\vec{a}) \cdot \vec{x}, b)$, where λ is the learning rate (parameterizable at runtime), df is the derivative of the given loss function, and b is the label[7] of the sample.

In the "gradient computation" stage, we again instantiate 512 bit-serial multipliers (❺) to compute the full-precision gradient with comparable throughput to the "dot product" stage. The scaling value *scale* is broadcast to each bit-serial multiplier as its bit-parallel input, while the bit-serial input $Q_s(\vec{a})$ can be read from the "FIFO" (Ⓑ). The bit-serial multipliers in this stage require exactly the same bit stream order of $Q_s(\vec{a})$ as that required by the "dot product" stage.

In the "model update" stage, the part of the gradient from the first 512 features is computed after s cycles, and then used to update the *working model* ($\overrightarrow{x\_w}$). The working model keeps the temporary model updated after each data sample is processed (❻).[8] Later, $\overrightarrow{x\_w}$ is updated with the part of gradient from the second 512 features after next s cycles, and so on. $\overrightarrow{x\_w}$ is updated at the rate of every sample, while the architectural model ($\vec{x}$) is updated only after a mini batch (B) of samples (❼), where B is the mini batch size (parameterizable at runtime). Therefore, the semantics of mini-batch SGD is preserved.

Instantiation of Hardware Design. "BWeaving" has a really high model bitwidth of $16K$ (32*512), making it unsuitable in real FPGA implementations.[9]

4.3 MLWeaving

MLWeaving develops the ideas behind BWeaving so as to make them implementable. In particular, MLWeaving changes both the memory layout and the design on the FPGA to dramatically reduce the required bitwidth of the model. Inspired by the mini-batch SGD that uses the same model to process one mini batch of B samples, we can process multiple samples in the same mini-batch simultaneously such that we can reduce the model bitwidth without compromising on throughput: 512 bits per cycle. As shown in Algorithm 1, MLWeaving instantiates 8 physical banks to accommodate 8 samples ($\#Bank$ = 8) in the same mini-batch simultaneously (Lines 5-6) such that 8 samples reading the same portion of the model are processed in a lock-step manner. This is why B must be a multiple of 8.[10] At the same time, we adjust the related memory layout such that the data stream from the memory flows into the MLWeaving hardware without any transposition overhead.

[6]In the actual implementation, we adopt the distributed arithmetic computation approach [43, 101] to compute the dot product, since it produces the same result with fewer hardware resources. Here, we use the basic bit-serial multiplier for ease of understanding.

[7]In our actual implementation, we also load the label b from memory. However, we omit the related data path in Figure 5 for clarity.

[8]The pair (architectural model, working model) is only used in the hardware design (Figures 5, 7), analogous to the pair (architectural register, physical register) in computer architecture. The architectural register indicates the register specified by instruction set architecture (ISA), visible to the programmer. The physical register is used to store temporary results, invisible to the programmer.

[9]In Intel Arria 10 FPGA, the model is implemented with 20-Kb memory blocks (i.e., M20Ks). Each M20K can provide a 32-bitwidth with ECC (or 40 without ECC) for the model, so a 16K-bitwidth requires 512 M20Ks with ECC (or 410 without ECC). It is extremely difficult to access 410 M20Ks in a lock-step manner (e.g., sharing the same read/write address) while maintaining high frequency, as M20Ks are uniformly distributed inside an FPGA.

[10]The larger $\#Bank$ leads to less complexity of hardware design but more limitation on mini-batch size.

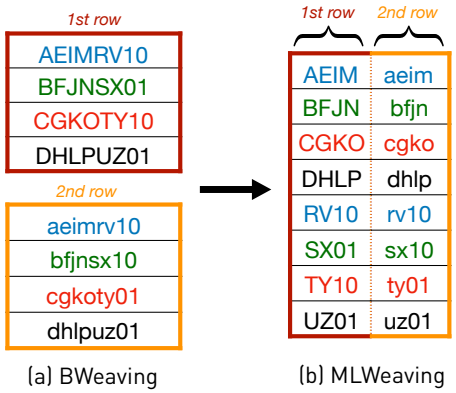


Figure 6: BWeaving memory layout (a) converted to MLWeaving memory layout (b): M = 8, $\#CL$ = 8 and $\#Bank$ = 2.

Table 4: Instantiation of MLWeaving memory layout. $x\_y{:}z\_w$ denotes the w-th bits of 64 features (from z-th to y-th) in the x-th sample and $y - z = 63$. A row contains a cache line (512 bits).

Description	Index	Bank 7	...	Bank 1	Bank 0
First 64 features of the first 8 samples	0	7_63:0_0	...	1_63:0_0	0_63:0_0
	1	7_63:0_1	...	1_63:0_1	0_63:0_1
	2	7_63:0_2	...	1_63:0_2	0_63:0_2
	...	...	...	...	...
	31	7_63:0_31	...	1_63:0_31	0_63:0_31
Second 64 features of the first 8 samples	32	7_127:64_0	...	1_127:64_0	0_127:64_0
	33	7_127:64_1	...	1_127:64_1	0_127:64_1
	...	...	...	...	...
...	...	...	...	...	...
First 64 features of the second 8 samples	1024	15_63:0_0	...	9_63:0_0	8_63:0_0
	1025	15_63:0_1	...	9_63:0_1	8_63:0_1
	...	...	...	...	...

4.3.1 *MLWeaving Memory Layout (Software)*

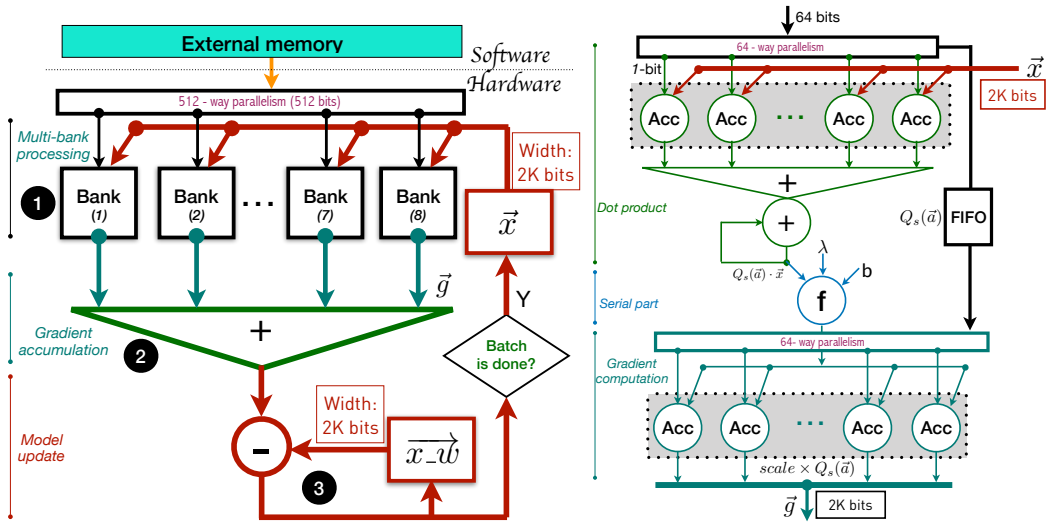
Starting from the BWeaving memory layout in Figure 3, we show the memory layout transition to MLWeaving in Figure 6. BWeaving populates each memory transaction with eight bits from the same row, e.g., AEIMRV10 of the first row. In contrast, the first/second row contributes four bits for each memory transaction under MLWeaving. For instance, the first bits of the first four features of two rows assemble into the first memory transaction, e.g., AEIMaeim. Since the bits AEIM and aeim share the same weights, the model only needs to provide four weights within a cycle.

Instantiation of Memory Layout. We instantiate the MLWeaving memory layout by setting $\#CL$ (or $\#Bank$) to be 512 (or 8). Now we use the case with M = 2048 as an example in Table 4. For instance, the first memory transaction is populated with the first bits of the first 64 features of the first eight samples. If M is not a multiple of 64, we use padding to align it to 64 bits. Thus, the memory traffic MT (in terms of bits) for each sample consists of two parts, as shown in Equation 4. The first equality shows the memory traffic from all the features, evaluated to be the precision level s multiplied by the value that rounds up M to the nearest multiple of 64. The second equality (32 bits) comes from the label of each sample.

$$MT = s \times \lceil \frac{M}{64} \rceil \times 64 + 32 \quad (4)$$

4.3.2 *MLWeaving Arithmetic (Hardware)*

Following Algorithm 1, we present the fully-pipelined hardware design of MLWeaving arithmetic in Figure 7. The targeted throughput of MLWeaving arithmetic is 512 bits per cycle. It consists of



(a) Overall architecture **(b)** Detailed design of each bank

Figure 7: Fully pipelined MLWeaving hardware with 8 banks ($\#Bank = 8$), according to Algorithm 1. Its throughput is 512 bits per cycle.

three main pipeline stages. In the following, we explain the design details of each pipeline stage.

In the "multi-bank processing" stage, the 512-bit computing pipeline is divided into 8 banks, each of which consumes 64 bits from one sample (❶), as shown in Figure 7a. The most important property of this stage is that 8 banks read the same portion of the model at a given time. Each bank behaves the same as that in the BWeaving arithmetic in Figure 5, except that each bank instantiates 64 bit-serial multipliers in the "dot product" and "gradient computation" stages (Figure 7b). Since we instantiate 8 banks, the total throughput of the MLWeaving arithmetic is still 512 bits per cycle.

In the "gradient accumulation" stage, a portion of the gradient (64 32-bit elements) from 8 banks is fed to 64 element-wise adder trees, each of which generates one element of the average gradient within a cycle (❷).

In the "model update" stage, the average gradient is used to update the working model ($\overrightarrow{x\_w}$) (❸) for every 8 samples. The architectural model ($\vec{x}$) is updated only after a mini batch (B) of samples is finished.

Instantiation of Hardware Design. Table 5 shows the resource consumption in our FPGA when we implement the computing pipeline of MLWeaving. MLWeaving achieves high clock frequency (400MHz) while requiring a reasonable amount of FPGA resources. This is because 1) the proposed multi-bank architecture of MLWeaving leads to fewer BRAMs (on-chip memory blocks on FPGAs) for the architectural and working models in Figure 7, and 2) the memory layout allows MLWeaving to directly consume data from memory, without auxiliary hardware modules for transposition. The theoretical throughput of MLWeaving's hardware design is roughly 25.6GB/s (400M * 512 bits per cycle), much larger than the available memory read bandwidth: 15GB/s.

Table 5: FPGA resource consumption

Name	Logic (ALMs)	DSPs	BRAMs	Frequency
BWeaving	N.A	N.A	N.A	N.A
MLWeaving	35670 (8.4%)	0 (0%)	3.25Mb (6.1%)	400 MHz

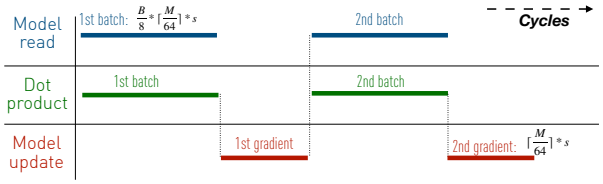
5. PRESERVATION OF PRECEDENCE

We propose a simple yet efficient scheme to keep SGD hardware design synchronous, without compromising processing speed. Our scheme is orthogonal to the BWeaving and MLWeaving designs, so it can be applied to both.[11] Our aim is three-fold. **G1**: it lets SGD read the up-to-date model. **G2**: it supports various batch sizes. **G3**: it exploits the greatest possible overlap between computation (i.e., dot product) and communication (i.e., model update). Next, we identify the performance issue of synchronous SGD, followed by our mechanisms to achieve three goals.

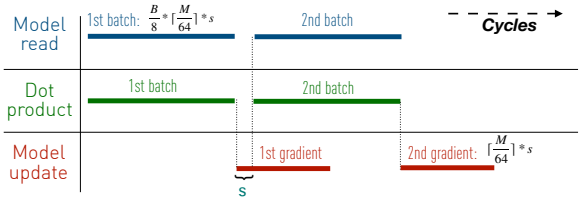
Performance Issue of Synchronous SGD. According to Algorithm 1, model reading (Line 8) and model update (Line 14) has a Read After Write (RAW) dependency, due to the inherently sequential nature of synchronous SGD. For example, the model read by the second batch (B samples) should be up-to-date such that the gradient from the first batch has already been accumulated into the model, as illustrated in Figure 8a. In other words, the second batch has to wait until the model is updated. The performance issue is not trivial, as it takes $\lceil M/64 \rceil * s$ cycles to update the model for each batch, where M is the number of features, 64 is the number of elements written to the model within a cycle, and s is the number of cycles to do one multiplication with a bit-serial multiplier.

Basic Mechanism. The goal of the basic mechanism is to preserve the precedence for synchronous SGD (**G1** and **G2**). In particular, our mini-batch SGD reads the up-to-date model. The key idea of the basic mechanism is to record the read/write operations performed on the model $\vec{x}$ such that MLWeaving will not read the out-of-date model and wait until the model becomes up-to-date. To do so, we introduce two 16-bit registers: *wr_counter* and *rd_counter*. Table 6 illustrate how to manipulate such two counters to preserve the dependency. The *wr_counter* records the times of writing operations performed on the model $\vec{x}$ in the "model update" stage. Its initialization value is B, where B is the input batch size. It means that B credits are provided at the beginning for B samples to read $\vec{x}$ in the "dot product" stage. *wr_counter* is incremented by B when the model $\vec{x}$ is updated, indicating that $\vec{x}$ is updated by the average gradient from every B samples in Algorithm 1. We can update the *wr_counter* once the average gradient is ready, since there is no WAR or WAW dependency. The *rd_counter* records the times of reading operations performed on the model $\vec{x}$ in the "dot product"

[11] In the following, we describe our scheme for MLWeaving. The scheme can easily generalize to BWeaving.



(a) Basic mechanism: without chaining



(b) Chaining-enhanced mechanism

Figure 8: MLWeaving with and without chaining.

Table 6: Basic mechanism to preserve RAW dependency

	Init Value	Step Size	Updating Condition
wr_counter	B	B	Always ok
rd_counter	0	8	*rd_counter* != *wr_counter*

stage. Its initialization value is 0, indicating that $\vec{x}$ has not been read yet. It is incremented by 8 when the model is read, since MLWeaving processes 8 samples concurrently. Its updating condition is that *rd_counter* is not equal to *wr_counter*, indicating there are still enough credits for samples to read $\vec{x}$. This is the critical step to preserve the dependency.

Chaining-Enhanced Mechanism. To increase the overlap between computation and model update (**G3**), we propose the chaining enhanced mechanism for synchronous SGD. The key idea is to allow computation when updating the model, while preserving the dependency. To do so, we treat the model $\vec{x}$ as a vector register that requires multiple cycles to read/write and follows a sequential access pattern. Then, we use the chaining technique [23, 51, 52, 80] of vector processing to maximally overlap computation and model update. In particular, the model updated with the gradient from the first batch can be forwarded to the second batch before the entire updating operation completes, as shown in Figure 8b. The second batch can begin to read after s cycles, where s is the precision level. We observe that it is safe for the second batch to read the model after the first part of the model (e.g., 64 values) is updated into the model. The data dependency is preserved since the model updating speed is not slower than the model reading speed.

6. PER-EPOCH TUNING OF PRECISION

MLWeaving introduces one tuning knob for the user: *the precision level to be used for training*. Most existing work assumes that it is the user's responsibility to set the right precision level. This is understandable, as the right level depends on both the data and the error tolerance and as no tight theory can map error tolerance back to the right precision level.

In this paper, we do not address the problem of determining the right precision level for the user, as it goes well beyond the scope of the work. Instead, we provide a simple, dynamic schedule of precision that harvests the potential of MLweaving. Such a schedule works robustly on all data sets we have.

Dynamic Precision Schedule. Our schedule is based on a very simple observation: at the beginning of the training, the system is less sensitive to the error introduced by low precision data representation; at the end of the training, the system often requires more bits to converge. MLWeaving allows us to dynamically change the number of bits to use for each epoch. We exploit this flexibility and build a simple dynamic precision schedule: use 2 bits for the 1st-4th epochs, 3 bits for the 5th-8th epochs, 4 bits for the 9th-16th epochs, 5 bits for the 17th-32nd epochs, and so on. That is, the number of bits grows over time until we reach the targeted loss.

This simple schedule is inspired by the following theoretical observation: for SGD to converge with $O(1/\sqrt{K})$ rate, at each iteration K, it only requires that the bias introduced by the low precision representation decreases faster than $O(1/K)$. The above schedule is one example that satisfies this property, as the bias introduced by low precision is halved when one more bit is used.

Remarks. Note that the above schedule is far from optimal and perfect. One can design more adaptive schedules by, for example, monitoring the speed of the decrease of loss and dynamically choosing when to switch to the next level of precision. MLWeaving allows this possibility by providing an end-to-end solution that enables dynamic precision schedule. We will explore more sophisticated precision schedule schemes as well as the problem of mapping precision and error levels as part of future work.

7. EXPERIMENTAL EVALUATION

7.1 Experimental Setup

Workloads. We carry out our experiments with the five data sets shown in Table 7. For the multi-class dataset TL [46], we use 10 classes of ImageNet [53]. Instead of directly using the images in ImageNet, we use 2048 features per sample extracted by a neural network (InceptionV3 [90]) that can be used for transfer learning, and we train binary classifiers using the one-vs-one strategy [78]. For the dataset Madelon, the training samples are duplicated 10 times so that the size exceeds the capacity of the last level cache in the CPU. For the dataset KDD, since its original dataset is highly skewed, i.e., 93% samples are labelled 0, we uniformly delete the samples labelled 0 so that the final dataset is balanced.

Table 7: Evaluated datasets.

Dataset	Features	Training samples	Testing samples	Classes
Gisette [9]	5000	6000	1000	2
TL [46]	2048	26,000	5200	10
Epsilon [9]	2000	40,000	10,000	2
KDD [63]	2399	40,000	44772	2
Madelon [9]	500	20,000	600	2

FPGA Implementations. There are two FPGA implementations. First, MLWeaving (sync) represents the MLWeaving hardware that satisfies the RAW dependency. Second, MLWeaving (async) represents the MLWeaving hardware that violates the RAW dependency such that it directly reads the model even when the model is still out-of-date.[12]

CPU Baselines. Since SGD is inherently sequential, keeping consistency when running SGD leads to no parallelism among cores. Two first-order variants of SGD ("Hogwild" and "ModelAverage") have been proposed to parallelize SGD on modern CPUs and we use both as baselines, employing existing optimization methods on CPUs: multi-core (14 cores), low-precision (8-bit) and AVX2 instruction (256-bit).

"Hogwild" [71, 106] allows each core to compute the gradient from its own portion of dataset and then to perform asynchronous update on a single copy of the model without any synchronization. Therefore, the parallelism among cores is exploited at the cost of low statistical efficiency due to asynchronous updates. Even though no synchronization is required, Hogwild still suffers from cache coherence overhead, i.e., invalidating the model copies in the private

[12] In the following experiments, by default, MLWeaving means MLWeaving (sync) on FPGAs and with chaining enabled.

caches of other cores before the real write operation. The cache coherence overhead is so severe on a multi-core CPU that Hogwild cannot benefit from using low-precision (Figure 14b).

"ModelAverage" [108] allows each core to have its own copy of the model so that no costly invalidation among cores occurs. It averages the models at the end of each epoch and then broadcasts the aggregated model to each core at the beginning of the next epoch. Therefore, its multi-core implementation can saturate the maximum memory bandwidth of the CPU (leading to high hardware efficiency). However, ModelAverage has relatively lower statistical efficiency since each worker uses its local (not global) model to compute the gradient. Since its 32-bit floating-point implementation is memory-bound, ModelAverage can significantly benefit from a low-precision dataset that requires less memory traffic. In our experiment, we choose 8-bit precision because 1) the smallest bank width of a SIMD register is 8 bits and 2) ModelAverage becomes compute-bound when the dataset is quantized to 8 bits, indicating lower performance for a lower-than-8-bit precision.

Learning Rate Schedules. During training, the learning rate decays based on a pre-defined schedule to achieve a higher convergence rate. We describe our concrete learning rate schedules on FPGAs and CPUs. On CPUs, the schedule determines the learning rate (λ_e) of the e-th epoch to be $\lambda \times \beta^{\sqrt{e}}$, where λ is the initial learning rate while β is the decay factor. In our experiment, a more aggressive decay policy, e.g., $\lambda/\sqrt{e}$ or λ/e, slows down the convergence rate. We find that a constant learning rate leads to the best performance on several datasets. On FPGAs, we employ a relatively simple schedule, as shown in Equation 5, where α is the threshold to decay the learning rate, because 1) on FPGAs, we apply the learning rate (λ_e) as a right-shift operator and thus $\lambda_e = 2^{-j}$, where j is an integer; and 2) MLWeaving needs a lower number of epochs to converge to the same training loss versus its CPU rivals that are not synchronous.

$$\lambda_e = \begin{cases} \lambda, & e \leq \alpha \\ \lambda * 0.5, & e > \alpha \end{cases} \quad (5)$$

Comparison Methodology. Our evaluations mainly validate three hypotheses. First, MLWeaving can achieve linear speedup when a smaller number of bits is used in the training (Subsection 7.2). Second, MLWeaving converges faster than its first-order counterparts on CPUs (Subsection 7.4). Third, using a dynamic precision schedule further accelerates the convergence process (Subsection 7.5).

7.2 Hardware Efficiency: Throughput

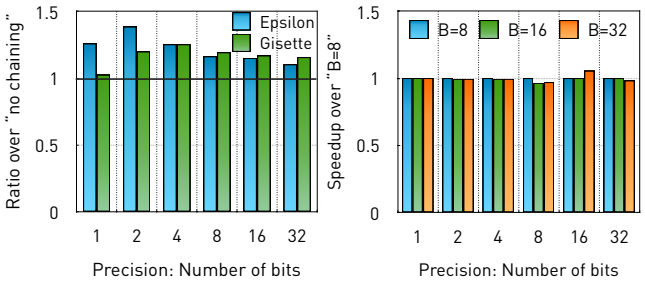
In this subsection, we demonstrate the hardware efficiency of MLWeaving, i.e., elapsed time for each epoch.[13] Our objective is two-fold. First, we analyze the performance characteristics of MLWeaving on FPGAs. Second, we compare MLWeaving with the state-of-art implementations on CPUs.

7.2.1 Hardware Characteristics of MLWeaving

We analyze five different hardware properties of MLWeaving. In our analysis, we typically run 50 epochs and get the average time for each epoch.

Effect of Chaining. We examine the effect of our chaining technique that relaxes the unnecessary RAW dependency for MLWeaving (Section 5). Figure 9a decipts the speedup of "chaining" over "no chaining" for two datasets. The batch size is 8. The x-axis depicts the precision s. We observe that with chaining, we can achieve up to 1.4X speedup over no chaining for different combinations of precision level (s) and number of features, since chaining can fully

[13]The elapsed time for each epoch is inversely proportional to the throughput that each implementation can achieve. In the following, we use both terms interchangeably to illustrate hardware efficiency.



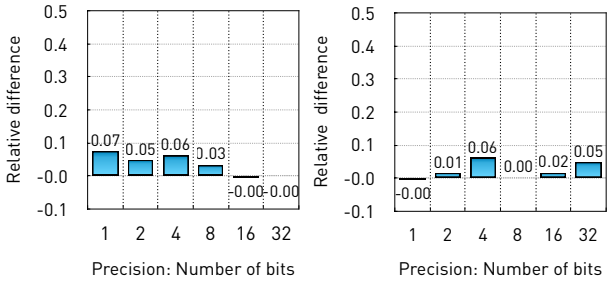
(a) Effect of chaining **(b)** Effect of batch size on Gisette

Figure 9: Hardware characteristics on hardware efficiency.

overlap computation and memory access. We conclude that chaining significantly increases the hardware efficiency. In order to fully understand the trend, we develop an analytical cost model to predict the performance for both chaining and no chaining settings, as shown in Appendix.

Effect of Mini Batch Size. We examine the effect of mini batch size (B) on hardware efficiency. Figure 9b illustrates the speedup of various batch sizes over "B=8" on the dataset Gisette. We observe that performance is roughly stable for different batch sizes, since MLWeaving is able to maximally overlap computation and memory access, regardless of mini batch size. We conclude that the batch size has a negligible effect on hardware efficiency.

MLWeaving (sync) vs. MLWeaving (async). We examine the effect of the RAW dependency on hardware efficiency. Intuitively, MLWeaving (async) would achieve more throughput than MLWeaving (sync), since MLWeaving (async) does not need to wait the RAW dependency to be resolved. However, since both approaches are memory-bound on the targeted FPGA, their relative performance difference, i.e., (MLWeaving (sync)- MLWeaving (async))/MLWeaving (sync), is small, as shown in Figure 10. We conclude that MLWeaving (sync), which preserves the RAW dependency, has little effect on hardware efficiency, with the help of our chaining technique.

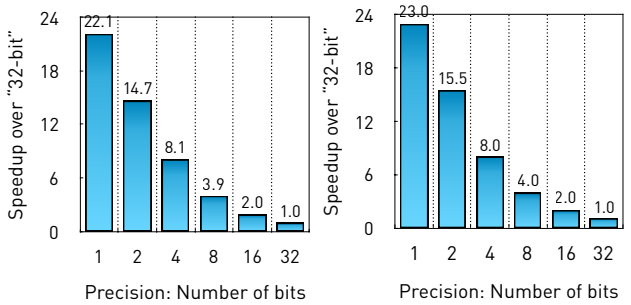


(a) Gisette **(b)** Epsilon

Figure 10: Relative performance difference between MLWeaving (sync) and MLWeaving (async). B is 16. s is 8.

Effect of Precision Level on Execution Time. Figure 11 provides the performance improvement of various precision levels over the full-precision implementation "32-bit" under MLWeaving for two datasets. We make two observations. First, the performance of MLWeaving improves roughly linearly as bit precision reduces, especially when s is larger than 4. Second, when s is less than 4, the speedup is sub-linear since the benefit from our chaining technique cannot fully amortize the negative impact of the inherent pipeline latency. We conclude that MLWeaving significantly reduces the elapsed time when using a smaller number of bits.

Effect of Precision Level on Memory Traffic. Figure 12 illustrates the memory traffic required by each sample for two datasets,



(a) Gisette (5000 features) **(b)** Epsilon (2000 features)

Figure 11: Relative performance improved with various precision levels over "32-bit", where "32-bit" is the case with $s = 32$.

as s is varied. We observe that the required memory traffic almost increases linearly as the precision level increases, consistent with the trend demonstrated in Equation 4. We conclude that the MLWeaving memory layout enables efficient data retrieval from external memory for any precision level at runtime.

7.2.2 Comparison with CPU Implementations

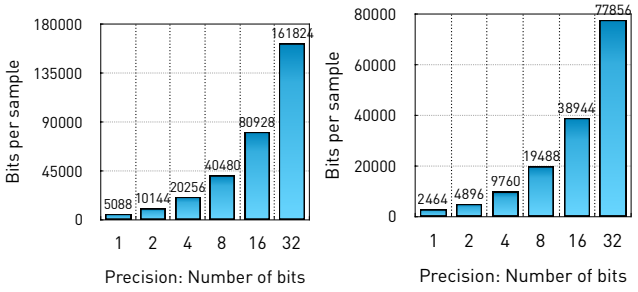
We compare the throughput of MLWeaving with two state-of-art CPU algorithms: Hogwild and ModelAverage. Figure 13 shows the comparison result. Both CPU algorithms are fully optimized. They use a low-precision dataset, employ all the 14 cores and are AVX2-enhanced. "x-FP" (or "x-char") indicates "x" with floating-point (or char) dataset, where "x" is Hogwild or ModelAverage. We use two metrics for comparison: time and memory traffic. "MLWeaving-32bit" (or "MLWeaving-8bit") means MLWeaving with $s = 32$ (or 8) on FPGAs.

Time. Figure 13a illustrates the normalized throughput of two CPU approaches and MLWeaving. We make two major observations. First, ModelAverage-FP achieves roughly 4 times more throughput than MLWeaving-32bit, since both are memory-bound and the achievable memory read bandwidth of the CPU (i.e., 60GB/s) is roughly four times as much as that of the FPGA (i.e., 15GB/s). ModelAverage-FP and MLWeaving-8bit roughly have the same throughput, while ModelAverage-char (which is compute-bound) achieves obviously more throughput than MLWeaving-8bit. Second, even though Hogwild-FP achieves more throughput than MLWeaving-32bit, it is still compute-bound since it suffers from severe cache coherence overhead due to the fact that multiple cores try to update the same memory address. When the model dimension becomes smaller, the overhead becomes larger as cache invalidation occurs more frequently among cores. [14]

Memory Traffic. Figure 13b illustrates the normalized memory traffic, where the memory traffic on the CPU is collected using the Intel Performance Counter Monitor [102]. We observe that both Hogwild-FP and ModelAverage-FP require roughly the same memory traffic as MLWeaving-32bit, since their datasets are all full-precision (32-bit), while Hogwild-char and ModelAverage-char require roughly the same amount of memory traffic as MLWeaving-8bit. We conclude that low-precision datasets causes less memory traffic on both CPUs and FPGAs.

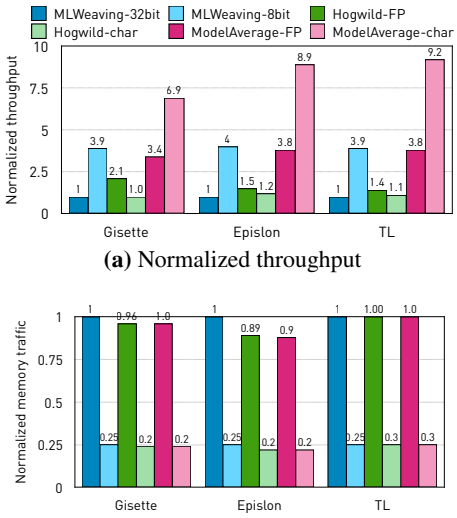
To sum up, due to the small amount of memory bandwidth available on FPGAs, MLWeaving has raw throughput advantage over CPU approaches only when the chosen precision level is relatively low, e.g., less than 4.

[14]In Appendix, we add the experiments about MLWeaving on CPUs, where the lack of hardware instructions makes it hard to exploit MLWeaving on the CPU.



(a) Gisette (5000 features) **(b)** KDD (2399 features)

Figure 12: Memory traffic (bits) per sample as the precision varies.



(a) Normalized throughput

(b) Normalized memory traffic

Figure 13: Hardware efficiency: MLWeaving vs. CPU rivals.

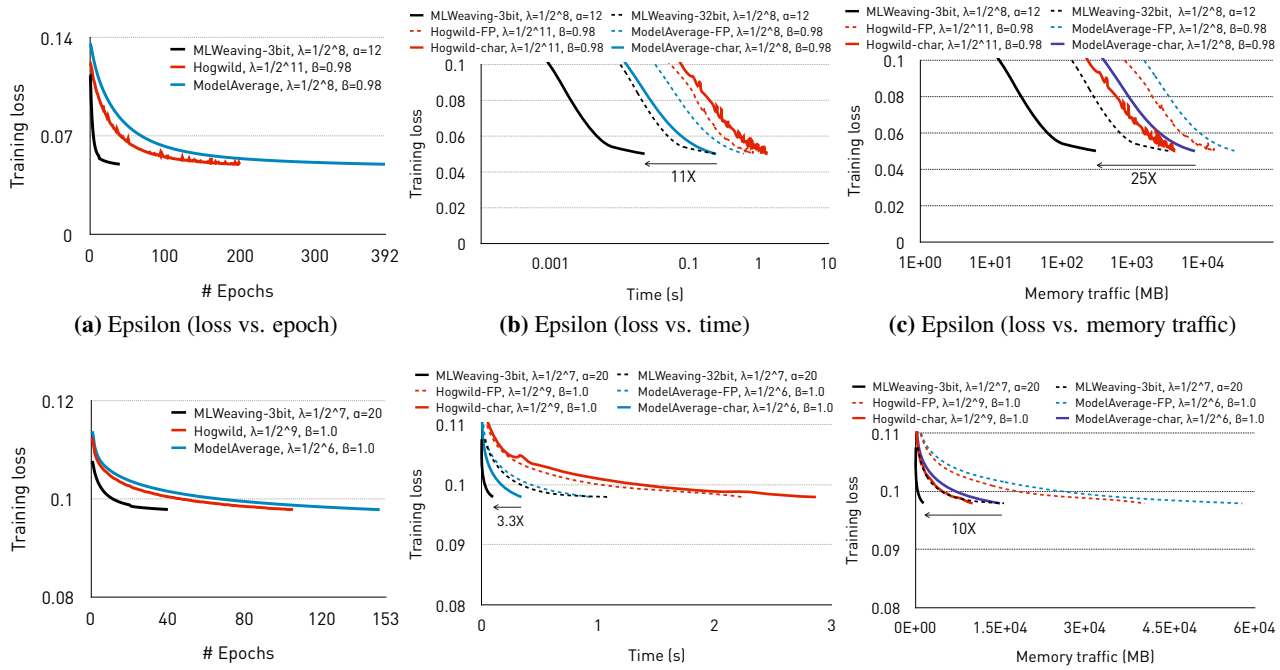
7.3 Statistical Efficiency: Loss vs. Epochs

We analyze the statistical efficiency of MLWeaving. We mainly validate that MLWeaving requires significantly fewer number of epochs to converge than its state-of-the-art CPU rivals.

MLWeaving vs. CPUs. We compare the statistical efficiency between MLWeaving and two full-precision CPU approaches for two datasets Epsilon and KDD[15] in Figures 14a,14d. We observe that MLWeaving requires a smaller number of epochs to converge than its CPU counterparts, even though MLWeaving uses 3-bit precision while CPU approaches use full precision. For example, MLWeaving requires only 40 epochs to converge for the dataset Epsilon, while ModelAverage (or Hogwild) needs 392 (or 199) epochs to converge to the same loss in Figure 14a. The underlying reason is that MLWeaving is always working on the up-to-date model while ModelAverage and Hogwild are not.

Impact of Mini Batch Size. We examine the impact of batch size B on statistical efficiency under MLWeaving. Figure 15 compares the convergence trend with different batch sizes for the datasets Gisette and KDD. We run 40 epochs and observe the training loss for each epoch. We find that a larger batch size leads to a slightly slower convergence speed. Thus, we prefer to use a small mini-batch size to train with MLWeaving.

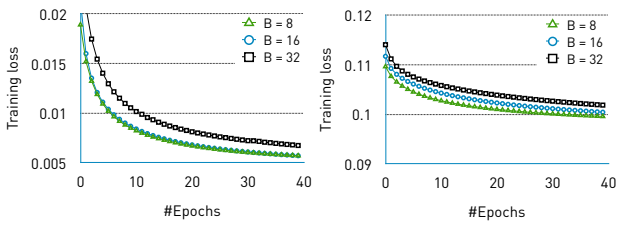
[15]We do not examine lower-precision (i.e., less than 8 bits) CPU approaches, as a lower precision always leads to a slightly worse statistical efficiency.



(a) Epsilon (loss vs. epoch) **(b)** Epsilon (loss vs. time) **(c)** Epsilon (loss vs. memory traffic)

(d) KDD (loss vs. epoch) **(e)** KDD (loss vs. time) **(f)** KDD (loss vs. memory traffic)

Figure 14: Convergence comparison: training loss vs. epoch/time/memory traffic. The batch size is 8. Speedup indicates MLWeaving versus the fastest 14-core AVX2-enhanced low-precision CPU approach, in terms of time and memory traffic.



(a) Gisette, $\lambda = 1/2^7$ **(b)** KDD, $\lambda = 1/2^8$

Figure 15: Effect of batch size on statistical efficiency. s is 8.

7.4 End-to-End Comparison: Loss vs. Time

In this subsection, we validate that MLWeaving outperforms its CPU rivals in terms of end-to-end performance (training loss vs. time), even though the evaluated CPU has 4 times more achievable memory bandwidth than the targeted FPGA. We employ two datasets Epsilon and KDD to demonstrate the comparison result, as shown in Figure 14.[16] We make three observations.

First, MLWeaving uses a low-precision dataset to converge to the same training loss as ModelAverage or Hogwild, each of which works on the full-precision dataset. For example, 3-bit precision is good enough for MLWeaving to train the dataset Epsilon, indicating great potential for low-precision training. Second, low-precision Hogwild slightly slows down the training. Since Hogwild is bounded by cache coherence overhead, low precision, which potentially speeds up other parts of SGD, causes more cache coherence traffic among cores and then slows down the convergence speed, as shown in Figure 14b. Third, MLWeaving requires up to 25X less data movement to converge to the same loss, compared to its low-precision CPU counterpart, as illustrated in Figures 14c. Thus, MLWeaving could provide 25X speedup if the FPGA had comparable memory bandwidth. It means that MLWeaving does significantly more energy-efficient training than its low-precision CPU counterparts. We conclude that MLWeaving, with its low-precision and synchronous training on FPGAs, greatly outperforms the state-of-the-art low-precision and asynchronous first-order training on CPUs for training linear models.

7.5 Effect of Flexible Precision Schedule

In this subsection, we examine the effect of our flexible precision schedule (in Section 6) that increases the level of precision with a simple fixed schedule during training. Table 8 illustrates the speedup that MLWeaving with dynamic precision schedule achieves over MLWeaving with fixed precision schedule and over the fastest CPU method with fixed precision. The CPU approach is fully optimized. We make two observations.

First, the dynamic precision schedule (i.e., "adaptive" approach) reduces end-to-end training time on average by 1.19x, compared with the "non-adaptive" approach that uses fixed precision under MLWeaving, indicating great potential of dynamic precision schedule.[17] Second, the dynamic precision schedule sometimes can lead to slowdowns, for instance for Epsilon. The reason is that Epsilon needs only 40 epochs to train with a low-precision (i.e., 3-bit) dataset to converge to the targeted loss, while our dynamic precision schedule (in Section 6) uses the high precision (above 3 bits) after the first 8 epochs. The training that uses the high precision (above 3 bits) needs the same number of epochs to converge as using the 3-bit precision, while the throughput of using the 3-bit precision is much higher. Therefore, we need a sophisticated precision schedule to fully utilize MLWeaving. We conclude that even a simple dynamic precision schedule can significantly reduce end-to-end training time.

[16] We only put the results of the two datasets with the largest and smallest speedups in Figure 14, while the results of all the other datasets are in Figure 19 of Appendix.

[17] The exact per-epoch tuning process for the dataset Gisette is shown in Appendix.

Table 8: End-to-end speedup due to flexible precision schedule.

Adaptive Approach	TL	Gisette	KDD	Madelon	Epsilon
Vs. non-adaptive	1.26×	1.5×	0.92×	1.35×	0.9×
Vs. fastest CPU	10.7×	16.5×	3×	6.1×	9.9×

8. RELATED WORK

To our knowledge, MLWeaving is the first novel solution for data representation and hardware acceleration for ML in database engines. MLWeaving builds on previous work from multiple communities: databases, machine learning, and computer architecture.

Bulk bitwise operations. Bulk bitwise operations [1, 25, 49, 58, 59, 60, 72, 79, 82, 83, 98] have been used in a variety of applications, including database scans [25, 59, 60, 83]. Among them, MLWeaving is inspired by BitWeaving [59], a transposed columnar storage layout designed for predicate evaluation on a per-column basis. BitWeaving is very efficient when answering a subfamily of relational queries. The memory layout used in MLWeaving is different from BitWeaving to accommodate the access pattern of batch SGD and the hardware implementation on an FPGA.

FPGA-accelerated ML/DL. MLWeaving builds on a growing line of research accelerating machine learning with FPGAs [5, 6, 8, 14, 15, 17, 26, 45, 61, 64, 65, 67, 84]. Closest to MLWeaving is the work by De Sa et al. [17] and Kara et al. [45] using FPGAs to implement low-precision generalized linear models, and they are asynchronous. These previous methods achieve good performance using individual circuits for each level of precision. Microsoft Brainwave [15] leverages FPGAs to implement low-precision programmable Neural Processing Unit for DNN inference at scale. It can support four low precision levels: 8/16-bit int and ms-fp9/8. In contrast, MLWeaving provides a single, flexible hardware design for all precision levels. We believe that MLWeaving has great potential to be used in Brainwave to enable any-precision inference.

Low-Precision DNNs. One specific application whose low-precision implementation on hardware has been intensively studied is Deep Neural Networks. Prior efforts [3, 18, 19, 21, 42, 43, 50, 85] explore the fine-grained variation in bit-level precision for DNN inference, so their computation time is proportional to the bitwidth used. Since the computation time dominates the overall performance for their DNN inference, the overall performance scales linearly with the bitwidth. Other research [2, 4, 7, 10, 11, 12, 16, 20, 29, 30, 38, 39, 41, 41, 54, 62, 77, 105] focuses on using a fixed-point, low-precision data representation and arithmetic to accelerate DNNs, instead of using full-precision. For example, Google's TPU [41] features 64K 8-bit fixed-point MACs to accelerate neural network inference. More information about DNNs can be found in the survey [89]. Compared with these efforts, MLWeaving focuses on a different workload, i.e., generalized linear models. We do not focus on the fixed quantization of the input data, but on flexible data retrieval. As part of future work we would like to investigate whether the design in MLWeaving can also be used to reduce memory traffic for DNNs.

Compression on DB/ML. Previous work [13, 24, 27, 31, 34, 55, 72, 74, 75, 76, 79, 81, 100] employs compression techniques, e.g., dictionary encoding, to compress the data such that the further memory traffic can be significantly reduced at the cost of lightweight decompression overhead. Previous work [22, 48] directly performs compressed operations on sparse data representations to accelerate linear algebra. In contrast, MLWeaving exploits the low precision of dataset to accelerate machine learning training.

9. CONCLUSION

MLWeaving is an innovative solution for embedding machine learning in relational engines and taking advantage of modern hardware. It consists of an in-memory data storage layout that allows efficient retrieval of quantized data at *any* level of precision, an efficient implementation of SGD on an FPGA, and an adaptive algorithm to learn a model using lower precision without having to determine the level of precision in advance. MLWeaving achieves linear speedup as precision level is decreased, and provides up to a 16X performance improvement compared to the state-of-the-art low-precision first-order CPU implementation. We make the MLWeaving design open-source.[18]

Future Directions and Limitations. The current prototype of MLWeaving has a number of limitations that will require additional work to make learning of generalized linear models possible in all cases. Some of these limitations are methodological and affect the ML algorithms used. Others are a question of exploring the design space enabled by MLWeaving in more detail, which cannot be done in this paper due to page limit. A first methodological limitation of MLWeaving is that, as is, it only supports dense, numerical data. This is fine in *some* applications as it is not uncommon for sparse and categorical values (e.g., YouTube video IDs) to be mapped to a dense embedding. However, there are cases where training directly on sparse or categorical data is still necessary. To support the latter case, MLWeaving would need to be extended by potentially combining it not only with other lossless compression strategies but also with ML techniques such as feature hashing [99] and weight sharing [31]. A very interesting future research direction MLWeaving opens is the development of a unified data structure supporting dense, sparse, categorical, or numerical data while providing similar level of flexibility as MLWeaving.

Another methodological limitation of MLWeaving is that it currently only supports generalized linear models. However, as long as the loss function is Lipschitz continuous over the input data and the data access pattern is row-wise, it is likely that similar techniques could still be applied with some adjustments. Thus, another interesting future direction is to adapt MLWeaving to problems such as matrix factorization [56] or clustering using K-Means [44].

An area where MLWeaving needs more work is dynamic precision scheduling. The current strategy is based on a simple intuition just to illustrate the benefit provided by MLWeaving. More sophisticated dynamic schedules will require a systematic analysis of the convergence properties with the dynamic precision changes. For example, one could use a technique similar to how AdaComm [94] dynamically adjusts communication frequencies. In particular, we can determine the convergence upper bound with respect to the precision schedule and choose the schedule that minimizes the convergence upper bound. We leave this direction to future work.

Limitations of MLWeaving caused by the current implementation and hardware can be solved by using different platforms or more complex designs. For example, MLWeaving currently supports models up to 32K dimensions due to the available on-chip memory capacity. It is reasonable to expect that the capacity will increase in future FPGAs. Also, the learning rate supported by MLWeaving can only be 2^{-j}, where j is an integer, since MLWeaving uses a right-shift operator to control the learning rate. It is not clear how limiting this is in practice given the current results, but we will explore different hardware designs to provide more flexibility as part of future work.

Acknowledgments. Some experiments in the paper were obtained through the Intel Hardware Accelerator Research Program (HARP2) at the Paderborn Center for Parallel Computing (PC^2). We thank Intel for their donation of the HARP2 machine.

[18] Github: https://github.com/fpgasystems/MLWeaving

10. REFERENCES

[1] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute Caches. In *HPCA*, 2017.

[2] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, H. Esmaeilzadeh, and R. Gupta. SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks. In *ISCA*, 2018.

[3] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos. Bit-pragmatic Deep Neural Network Computing. In *MICRO*, 2017.

[4] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. Cnvlutin: Ineffectual-neuron-free Deep Neural Network Computing. In *ISCA*, 2016.

[5] A. Boutros, S. Yazdanshenas, and V. Betz. Embracing Diversity: Enhanced DSP Blocks for Low-Precision Deep Learning on FPGAs. In *FPL*, 2018.

[6] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. P. Graf. A Massively Parallel FPGA-Based Coprocessor for Support Vector Machines. In *FCCM*, 2009.

[7] R. Cai, A. Ren, N. Liu, C. Ding, L. Wang, X. Qian, M. Pedram, and Y. Wang. VIBNN: Hardware Acceleration of Bayesian Neural Networks. In *ASPLOS*, 2018.

[8] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A Cloud-Scale Acceleration Architecture. In *MICRO*, 2016.

[9] C. Chang and C. Lin. LIBSVM: A Library for Support Vector Machines. *TIST*, 2(3):27:1–27:27, 2011.

[10] Y. Chen, J. Emer, and V. Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ISCA*, 2016.

[11] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *JSSC*, 2017.

[12] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. DaDianNao: A Machine-Learning Supercomputer. In *MICRO*, 2014.

[13] Z. Chen, J. Gehrke, and F. Korn. Query Optimization in Compressed Database Systems. In *SIGMOD*, 2001.

[14] G. R. Chiu, A. C. Ling, D. Capalija, A. Bitar, and M. S. Abdelfattah. Flexibility: FPGAs and CAD in Deep Learning Acceleration. In *ISPD*, 2018.

[15] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. E. Husseini, T. Juhasz, K. Kagi, R. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Y. Xiao, D. Zhang, R. Zhao, and D. Burger. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[16] M. Courbariaux, Y. Bengio, and J. David. Low Precision Arithmetic For Deep Learning. *CoRR*, abs/1412.7024, 2014.

[17] C. De Sa, M. Feldman, C. Ré, and K. Olukotun. Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent. In *ISCA*, 2017.

[18] C. De Sa, M. Leszczynski, J. Zhang, A. Marzoev, C. R. Aberger, K. Olukotun, and C. Ré. High-Accuracy Low-Precision Training. *ArXiv*, 2018.

[19] A. Delmas, S. Sharify, P. Judd, and A. Moshovos. Tartan: Accelerating Fully-Connected and Convolutional Layers in Deep Learning Networks by Exploiting Numerical Precision Variability. *CoRR*, abs/1707.09068, 2017.

[20] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *ISCA*, 2015.

[21] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das. Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. In *ISCA*, 2018.

[22] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. Compressed Linear Algebra for Large-scale Machine Learning. *PVLDB*, 9(12):960–971, 2016.

[23] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: a vector extension to the alpha architecture. In *ISCA*, 2002.

[24] F. Farber, N. May, W. Lehner, I. Muller, H. Rauhe, J. Dees, and S. Ag. The SAP HANA Database: An Architecture Overview. In *IEEE Data Eng. Bull.*, 2012.

[25] Z. Feng, E. Lo, B. Kao, and W. Xu. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *SIGMOD*, 2015.

[26] 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 *ISCA*, 2018.

[27] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.

[28] P. Gupta. Accelerating Datacenter Workloads. *FPL*, 2016.

[29] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep Learning with Limited Numerical Precision. In *ICML*, 2015.

[30] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ISCA*, 2016.

[31] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *ICLR*, 2015.

[32] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[33] W. Hillis. *The Connection Machine*. MIT Press, 1986.

[34] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans. In *SIGMOD*, 2007.

[35] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. MonetDB: Two Decades of Research in Column-oriented Database. *IEEE Data Engineering Bulletin*, 2012.

[36] Z. Istvan, D. Sidler, and G. Alonso. Runtime

Parameterizable Regular Expression Operators for Databases. In *FCCM*, 2016.

[37] Z. Istvan, L. Woods, and G. Alonso. Histograms As a Side Effect of Data Movement for Big Data. In *SIGMOD*, 2014.

[38] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko. Gist: Efficient Data Encoding for Deep Neural Network Training. In *ISCA*, 2018.

[39] Y. Ji, Y. Zhang, W. Chen, and Y. Xie. Bridge the Gap Between Neural Networks and Neuromorphic Hardware with a Neural Network Compiler. In *ASPLOS*, 2018.

[40] R. Johnson and I. Pandis. The Bionic DBMS is Coming, but What Will It Look Like? In *CIDR*, 2013.

[41] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. 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 *ISCA*, 2017.

[42] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. Moshovos. Proteus: Exploiting Numerical Precision Variability in Deep Neural Networks. In *ICS*, 2016.

[43] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. In *MICRO*, 2016.

[44] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An Efficient k-Means Clustering Algorithm: Analysis and Implementation. *TPAMI*, 24:881–892, 2002.

[45] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang. FPGA-Accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-Off. In *FCCM*, 2017.

[46] K. Kara, K. Eguro, C. Zhang, and G. Alonso. ColumnML: Column-Store Machine Learning with On-the-Fly Data Transformation. *PVLDB*, 12(4):348–361, 2018.

[47] K. Kara, J. Giceva, and G. Alonso. Fpga-Based Data Partitioning. In *SIGMOD*, 2017.

[48] V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris. An Extended Compression Format for the Optimization of Sparse Matrix-Vector Multiplication. *TPDS*, 24(10):1930–1940, 2013.

[49] J. Kim, M. Sullivan, E. Choukse, and M. Erez. Bit-Plane Compression: Transforming Data for Better Compression in Many-Core Architectures. In *ISCA*, 2016.

[50] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray, 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 *NIPS*. 2017.

[51] C. Kozyrakis and D. Patterson. Vector vs. Superscalar and VLIW architectures For Embedded Multimedia Benchmarks. In *MICRO*, 2002.

[52] C. E. Kozyrakis and D. A. Patterson. Scalable, Vector Processors For Embedded Systems. *IEEE Micro*, 2003.

[53] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*. 2012.

[54] H. Kwon, A. Samajdar, and T. Krishna. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *ASPLOS*, 2018.

[55] L. Lamport. Multiple Byte Processing with Full-word Instructions. *Commun. ACM*, 18(8):471–475, 1975.

[56] D. Lee and S. Sebastian. Algorithms for Non-negative Matrix Factorization. In *NIPS*. 2001.

[57] M. Li, T. Zhang, Y. Chen, and A. J. Smola. Efficient Mini-batch Training for Stochastic Optimization. In *SIGKDD*, 2014.

[58] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie. Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories. In *DAC*, 2016.

[59] Y. Li and J. M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *SIGMOD*, 2013.

[60] Y. Li and J. M. Patel. WideTable: An Accelerator for Analytical Data Processing. *PVLDB*, 7(10):907–918, 2014.

[61] Z. Li, C. Ding, S. Wang, W. Wen, Y. Zhuo, C. Liu, Q. Qiu, W. Xu, X. Lin, X. Qian, and Y. Wang. E-RNN: Design Optimization for Efficient Recurrent Neural Networksin FPGAs. In *HPCA*, 2019.

[62] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen. PuDianNao: A Polyvalent Machine Learning Accelerator. In *ASPLOS*, 2015.

[63] Y. Liu, H. Zhang, L. Zeng, W. Wu, and C. Zhang. MLbench: Benchmarking Machine Learning Services Against Human Experts. *PVLDB*, 11(10):1220–1232, 2018.

[64] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh. In-RDBMS Hardware Acceleration of Advanced Analytics. *PVLDB*, 11(11):1317–1331, 2018.

[65] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh. TABLA: A Unified Template-based Framework For Accelerating Statistical Machine Learning. In *HPCA*, 2016.

[66] B. Moons and M. Verhelst. A 0.3 x2013;2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets. In *VLSI-Circuits*, 2016.

[67] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration. In *HPCA*, 2015.

[68] R. Mueller, J. Teubner, and G. Alonso. Data Processing on FPGAs. *PVLDB*, 2(1):910–921, 2009.

[69] R. Mueller, J. Teubner, and G. Alonso. Streams on Wires: A Query Compiler for FPGAs. *PVLDB*, 2(1):229–240, 2009.

[70] R. Mueller, J. Teubner, and G. Alonso. Glacier: A Query-to-hardware Compiler. In *SIGMOD*, 2010.

[71] F. Niu, B. Recht, C. Re, and S. Wright. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*. 2011.

[72] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *SIGMOD*, pages 38–49, 1997.

[73] M. Owaida, D. Sidler, K. Kara, and G. Alonso. Centaur: A Framework for Hybrid CPU-FPGA Databases. In *FCCM*,

2017.

[74] G. Pekhimnko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework. In *MICRO*, 2013.

[75] O. Polychroniou and K. A. Ross. Efficient Lightweight Compression Alongside Fast Scans. In *DaMoN*, 2015.

[76] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *ICDE*, 2008.

[77] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks. Minerva: Enabling Low-power, Highly-accurate Deep Neural Network Accelerators. In *ISCA*, 2016.

[78] R. Rifkin and A. Klautau. In Defense of One-Vs-All Classification. In *JMLR*. 2004.

[79] D. Rinfret, P. O'Neil, and E. O'Neil. Bit-Sliced Index Arithmetic. In *SIGMOD*, 2001.

[80] R. M. Russell. The CRAY-1 Computer System. *Commun. ACM*, 21(1):63–72, 1978.

[81] B. Schlegel, R. Gemulla, and W. Lehner. Fast Integer Compression Using SIMD Instructions. In *DaMoN*, 2010.

[82] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Fast Bulk Bitwise AND and OR in DRAM. *IEEE CAL*, 2015.

[83] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Ambit: In-memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *MICRO*, 2017.

[84] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From High-Level Deep Neural Models to FPGAs. In *MICRO*, 2016.

[85] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. Kim, V. Chandra, and H. Esmaeilzadeh. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks. In *ISCA*, 2018.

[86] D. Sidler, Z. István, M. Owaida, and G. Alonso. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *SIGMOD*, 2017.

[87] D. Sidler, Z. István, M. Owaida, K. Kara, and G. Alonso. doppioDB: A Hardware Accelerated Database. In *SIGMOD*, 2017.

[88] A. Sinha and A. P. Chandrakasan. Energy Efficient Filtering Using Adaptive Precision and Variable Voltage. In *IEEE International ASIC/SOC Conference*, 1999.

[89] 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*, 2017.

[90] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception Architecture For Computer Vision. In *CVPR*. 2016.

[91] J. Teubner and L. Woods. *Data Processing on FPGAs Synthesis Lectures on Data Management*. 2013.

[92] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. Finn: A framework For Fast, Scalable Binarized Neural Network Inference. In *FPGA*, 2017.

[93] Y. Umuroglu, L. Rasnayake, and M. Sjlander. BISMO: A Scalable Bit-Serial Matrix Multiplication Overlay for Reconfigurable Computing. In *FPL*, 2018.

[94] J. Wang and G. Joshi. Adaptive Communication Strategies to Achieve the Best Error-Runtime Trade-off in Local-Update SGD. *CoRR*, abs/1810.08313, 2018.

[95] Z. Wang, B. He, and W. Zhang. A Study of Data Partitioning on OpenCL-based FPGAs. In *FPL*, 2015.

[96] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang. Relational Query Processing on OpenCL-based FPGAs. In *FPL*, 2016.

[97] Z. Wang, J. Paul, B. He, and W. Zhang. Multikernel Data Partitioning With Channel on OpenCL-Based FPGAs. *TVLSI*, 25(6):1906–1918, 2017.

[98] Z. Wang, K. Zhang, H. Zhou, X. Liu, and B. He. Hebe: An Order-Oblivious and High-Performance Execution Scheme for Conjunctive Predicates. In *ICDE*, 2018.

[99] K. Q. Weinberger, A. Dasgupta, J. Attenberg, J. Langford, and A. J. Smola. Feature Hashing for Large Scale Multitask Learning. *CoRR*, 2009.

[100] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Rec.*, 29(3):55–67, 2000.

[101] S. White. Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review. *IEEE ASSP Magazine*, 6(3):4–19, 1989.

[102] T. Willhalm, R. Dementiev, and P. Fay. Intel Performance Counter Monitor - A better way to measure CPU utilization, https://software.intel.com/en-us/articles/intel-performance-counter-monitor,2016.

[103] L. Woods, Z. István, and G. Alonso. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading. *VLDB*, 7(11):963–974, 2014.

[104] T. Xanthopoulos and A. Chandrakasan. A Low-power DCT Core Using Adaptive Bitwidth and Arithmetic Activity Exploiting Signal Correlations and Quantization. In *Symposium on VLSI Circuits*, 1999.

[105] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In *ISCA*, 2017.

[106] C. Zhang and C. Ré. DimmWitted: A Study of Main-memory Statistical Analytics. *PVLDB*, 7(12):1283–1294, 2014.

[107] 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 *ICML*, volume 70, pages 4035–4043, 2017.

[108] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized Stochastic Gradient Descent. In *NIPS*. 2010.

11. APPENDIX

11.1 Cost Model

In this section, we propose a hybrid analytical/empirical cost model to predict the performance of MLWeaving on FPGAs. We predict the throughput Th to be the minimum of the computing throughput Th_{comp} and memory throughput TH_{mem}, as shown in Equation 6.

$$Th = Min(Th_{comp}, Th_{mem}) \quad (6)$$

11.1.1 Evaluating Computing Throughput

In this subsection, we present an analytical model to evaluate the computing throughput for "chaining" and "no chaining", under the assumption that the memory subsystem can provide the data as soon as the computing logic requires. The MLWeaving hardware design can consume 512 bits per cycle, so its theoretical computing throughput is 512 bits*400MHz=25.6GB/s. However, the practical computing throughput is lower, since MLWeaving has to guarantee the RAW dependency in the SGD model. One unavoidable factor is the pipeline latency L, which is the latency between the dot product module and the model update module. The value of L is $40 + 2s$ cycles, where s is the precision level.

"Chaining". We evaluate the computing throughput of "chaining", as shown in Equation 7.

$$Th_{comp} = \frac{(B/8) \times \lceil M/64 \rceil \times s}{(B/8) \times \lceil M/64 \rceil \times s + L} \times 25.6GB/s, \quad (7)$$

where the right part is the theoretical computing throughput and the left part is the utilization of the computing logic (i.e., the dot product module). Since chaining is enabled, the overhead of updating the model can be removed. However, the pipeline latency L cannot be ignored, especially when M and s are small.

"No chaining". We evaluate the computing throughput of "no chaining", as shown in Equation 8.

$$Th_{comp} = \frac{(B/8) \times \lceil M/64 \rceil \times s}{(1 + B/8) \times \lceil M/64 \rceil \times s + L} \times 25.6GB/s, \quad (8)$$

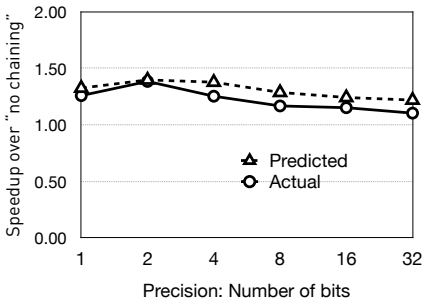
where the overhead of updating the model is $\lceil M/64 \rceil \times s$. The main difference from "chaining" is that the dot product module is idle when the model update module is active.

11.1.2 Evaluating Memory Throughput

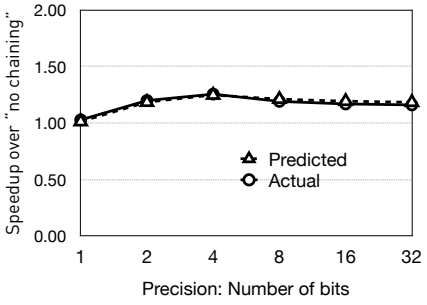
MLWeaving accesses the host memory via two PCIe and one QPI links, so the memory throughput is affected by both the memory subsystem and the PCIe/QPI links. This makes the explicit prediction (in terms of the analytical cost model) of memory throughput extremely difficult, especially when the memory-related IP core on the FPGA side is encrypted. Therefore, we present an empirical cost model to evaluate the memory throughput Th_{mem}. In order to predict the memory throughput of MLWeaving, we analyze the memory access pattern of MLWeaving and measure the memory throughput. Note, both "chaining" and "no chaining" use the same memory subsystem.

Memory Access Pattern. The memory access pattern of MLWeaving is quite fixed. Essentially, for the precision level s, MLWeaving sequentially fetches s cache lines (512 bits each) every 32 cache lines, regardless of the number of features. We conclude that MLWeaving's memory bandwidth is quite fixed.

Benchmarking Memory Throughput. Based on the memory access pattern, we can benchmark the memory throughput for each precision level s under our framework Centaur [73]. We can get the empirical memory throughput, as illustrated in Equation 9. We can observe that when s is small (e.g., < 4), the memory throughput becomes noticeably lower due to its low utilization of row buffer contents. When s is larger than 4, the throughput of the PCIe/QPI links becomes the new bottleneck, so the achievable throughput stays constant.



(a) Epsilon (2000 features)



(b) Gisette (5000 features)

Figure 16: Speedup of "chaining" over "no chaining.

$$Th_{mem} = \begin{cases} 10.2GB/s & s = 1 \\ 13.3GB/s & s = 2 \\ 13.8GB/s & s = 3 \\ 14.8GB/s & s \geq 4 \end{cases} \quad (9)$$

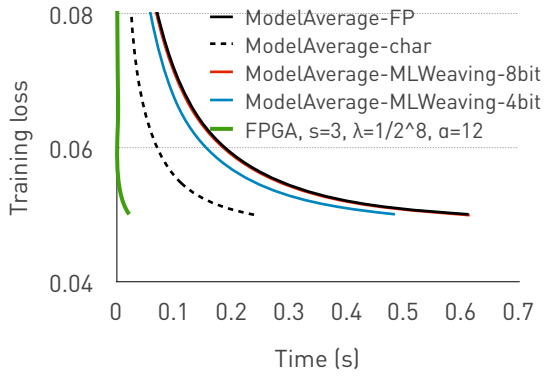
11.1.3 Predicting MLWeaving Performance via the Cost Model

In this subsection, we leverage our cost model to predict the speedup of chaining over no chaining to demonstrate the effect of chaining. Actually, the intuition behind this speedup is not straightforward. In order to understand this speedup, we leverage our cost model to predict the throughput of chaining and no chaining individually. Then, we can calculate the speedup to be the throughput (Th) of chaining divided by the throughput (Th) of no chaining with different numbers of features (M) and different precision level (s). Figure 16 shows the different peak speed up for different datasets. "Actual" indicates the real speedup measured via our experiments, while "predicted" shows the estimated speedup using our cost model. We make two observations.

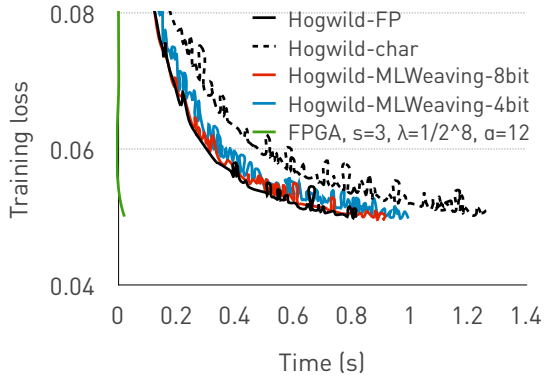
First, our cost model can roughly predict the speedup for different precision levels. Second, our cost model can predict the peak speedup for different numbers of features (M) and different precision levels (s). In particular, the peak speedup is for s=4 for Epsilon (2K features), while the peak is at s=2 for Gisette (5k features). We conclude that our cost model can guide us to find the peak speedup with different precision levels and different numbers of features.

11.2 MLWeaving on Modern CPUs

We examine the performance of MLWeaving on CPUs. Since modern CPUs do not yet have any custom instructions to efficiently



(a) ModelAverage, $\lambda = 1/2^8, \beta = 0.98$



(b) Hogwild, $\lambda = 1/2^{11}, \alpha = 12$

Figure 17: MLWeaving on CPU with the dataset of Epsilon.

consume the bit stream from MLWeaving layout, we have to employ regular instructions to retrieve each data element for further computing. For instance, an 8-bit element contains bits from eight different memory locations, leading to a very significant memory lookup overhead. We make our best effort that our implementation is able to retrieve 32 elements (i.e., 32-way parallelism) in a single AVX2 instruction. Figure 17 illustrates the performance of MLWeaving on CPUs. We make three observations.

First, MLWeaving can converge faster than the floating-point implementation under ModelAverage, when the precision level is less than 8 bits, as shown in Figure 17a. This means that MLWeaving can also bring reasonable performance benefits on CPUs. Second, MLWeaving makes the low-precision implementation slower under ModelAverage. In particular, "ModelAverage-MLWeaving-8-bit" is much slower than "ModelAverage-char", even though both have the same precision of 8 bits. It means that the memory lookup overhead of MLWeaving cannot be fully amortized when deploying it on CPUs. Third, the low-precision implementation makes HogWild slower, as HogWild is always bounded by the cache coherence overhead. This means that the low-precision implementation, which can generate more shared model update operations in a fixed period, incurs more pressure to cache coherence module inside a CPU.

11.3 Flexible Precision Schedule

Figure 18 shows the per-epoch tuning process for the Gisette dataset. The x-axis is the elapsed time and the y-axis is the training loss. The baseline here is the 4-bit precision, denoted by "Non-adaptive, 4-bit", which can converge to the same training loss as full precision does. The proposed adaptive approach increases the level of precision during the training. In particular, it starts with 2-bit precision (denoted by "2-bit") for the first four epochs, followed by 3-bit precision (denoted by "3-bit") for the next four epochs. Then, 4-bit precision is used for the next eight epochs. Finally, ten epochs are employed under 5-bit precision to reach the target loss. We make two observations.

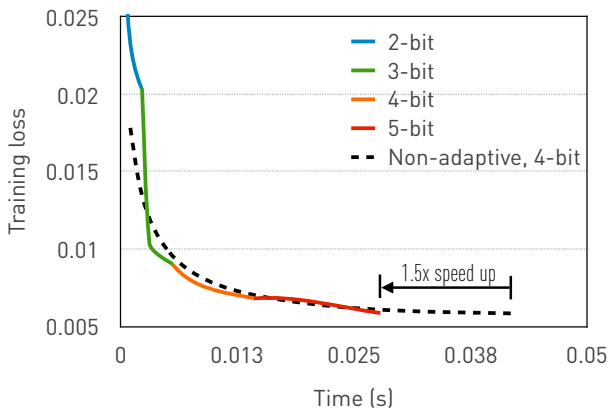
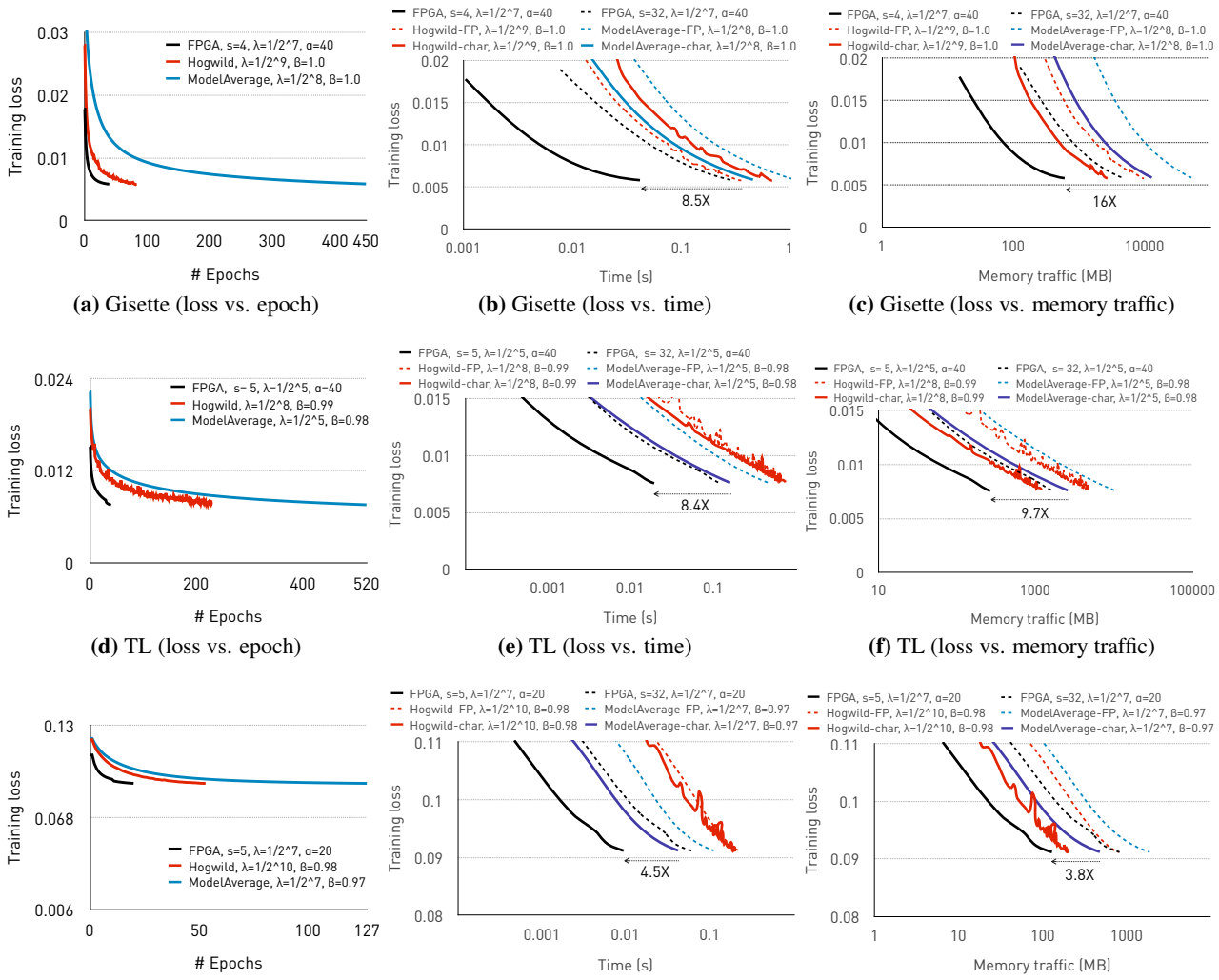


Figure 18: Effect of flexible precision schedule.

First, the proposed flexible precision schedule can reach the same training loss, while achieving 1.5X performance improvement over the baseline ("Non-adaptive, 4-bit"), even though the schedule is preliminary, indicating a great potential for using per-epoch precision schedules. Second, each precision level transition (e.g., 2-bit to 3-bit) brings a significant reduction in loss, coinciding with our theory that tuning the precision level higher (e.g., 2-bit to 3-bit) at runtime can converge to the same loss as a fixed 3-bit precision for all the epochs.

11.4 Convergence Analysis

We add the experimental results for the remaining three datasets, as shown in Figure 19.



(a) Gisette (loss vs. epoch) **(b)** Gisette (loss vs. time) **(c)** Gisette (loss vs. memory traffic)

(d) TL (loss vs. epoch) **(e)** TL (loss vs. time) **(f)** TL (loss vs. memory traffic)

(g) Madelon (loss vs. epoch) **(h)** Madelon (loss vs. time) **(i)** Madelon (loss vs. memory traffic)

Figure 19: Convergence comparison: loss vs. epoch/time/memory traffic. The batch size is 8. Speedup indicates MLWeaving vs. fastest 14-core AVX2-enhanced low-precision CPU approach, in terms of time and memory traffic.