2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines

FPGA-accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-off

Kaan Kara*, Dan Alistarh*†, Gustavo Alonso*, Onur Mutlu*, Ce Zhang*

* Systems Group, Department of Computer Science
ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

† Distributed Algorithms and Systems
IST, Austria

***Abstract*—Stochastic gradient descent (SGD) is a commonly used algorithm for training linear machine learning models. Based on vector algebra, it benefits from the inherent parallelism available in an FPGA. In this paper, we first present a single-precision floating-point SGD implementation on an FPGA that provides similar performance as a 10-core CPU. We then adapt the design to make it capable of processing low-precision data. The low-precision data is obtained from a novel compression scheme—called stochastic quantization, specifically designed for machine learning applications. We test both full-precision and low-precision designs on various regression and classification data sets. We achieve up to an order of magnitude training speedup when using low-precision data compared to a full-precision SGD on the same FPGA and a state-of-the-art multi-core solution, while maintaining the quality of training. We open source the designs presented in this paper.**

I. Introduction

FPGAs have become increasingly popular hardware accelerators for machine learning due to the inherent parallelism and the deeply-pipelined computation they offer. Recently, FPGAs have been used to accelerate both training and inference for a range of machine learning models (e.g., generalized linear models [2], [12], [16] and deep learning [7], [8]), using various optimization algorithms (e.g., conjugate gradient [9], [21] and stochastic gradient descent [4], [14]). One prominent feature of machine learning algorithms, especially those trained with stochastic gradient descent, is that they can tolerate certain types of noise and errors incurred during execution and still return statistically the same answer. This observation has enabled a range of system optimizations such as lock-free [20] and asynchronous execution [22]. Recently, one emerging line of research has focused on developing machine learning algorithms that can tolerate a different type of noise—low-precision data representation and/or computation [8], [10], [13], [19]. Using low-precision data for training on FPGAs has been considered before, using nearest-value (naive) rounding to reduce data precision [2], [13], [17].

In this paper, we focus on the question: *"What is impact of using low precision data for FPGA-based stochastic gradient descent on (1) performance and (2) result quality?"* To address this question, we explore a novel way of reducing the precision of data—stochastic quantization, which has been shown by recent machine learning studies [1], [6], [23] to produce high quality and unbiased results for training dense linear models.

Machine Learning Scope. We focus on training one of the simplest class of machine learning models—dense linear models. Despite their simplicity, dense linear models are fundamental for applications such as regression and classification, compressive sensing, and image reconstruction. For applications such as human-in-the-loop analytics and feature selection, one often needs to train hundreds or thousands of models. Thus, the training speed is important.

Performance Objective. SGD is an iterative algorithm that performs multiple passes over the data (so called *epochs*). There are two decoupled metrics to assess the performance of SGD: (1) *statistical efficiency*, the number of epochs (N_{epochs}) the algorithm needs to converge, and (2) *hardware efficiency*, the time the algorithm requires to execute each epoch (T_{epoch}). Our objective is to increase the hardware efficiency, by lowering T_{epoch}, and maintain statistical efficiency, by keeping N_{epochs} the same. We show that this is possible on an FPGA when using stochastically quantized data.

Design Space. The reason quantized data leads to better hardware efficiency is simple: The FPGA needs to read less volume of data per epoch. However, the parallelism of the design has to be increased to be able to process quantized data. Furthermore, stochastic quantization has been shown to maintain statistical efficiency [23]. However, the quantization level (precision) needs to be an adjustable parameter, since its effects on statistical efficiency highly depend on the data set characteristics, among other SGD related configurations. Thus, the design space required for complete control over both hardware and statistical efficiency of SGD contains the following parameters: 1) design decisions for the implemented circuit, 2) data precision, and 3) SGD algorithm configuration parameters (learning rate, mini-batch size).

Contribution 1. We design flexible prototypes to explore different points in the design space. We first present an FPGA-based SGD implementation working on 32-bit floating-point data (`floatFSGD`). Apart from being scalable (handling high dimensionality) and resource-efficient, `floatFSGD`'s performance is on par with a 10-core CPU, despite being bound on the available memory bandwidth on our current platform. Then, we increase `floatFSGD`'s internal parallelism, so that it can process quantized data—`qFSGD` is up to 11× faster than `floatFSGD` and up to 10.6× faster than the fastest 10-threaded CPU version of SGD we have access to.

978-1-5386-4037-1/17 $31.00 © 2017 IEEE
DOI 10.1109/FCCM.2017.39

Contribution 2. We reveal a new trade-off space that helps us to determine the most efficient way to do SGD on an FPGA: (I) We explore different circuit designs for `qFSGD` to achieve better scalability in order to process lower precision data. We show that with 1-bit quantization linear scaling does not work, leading to a compute bound design.
(II) As we vary the precision and the quantization strategies (stochastic quantization vs. naive rounding), we find that:

1) The lower the precision, the better the hardware efficiency, because of higher bandwidth utilization (less volume of data being read).
2) The lower the precision, the worse the statistical efficiency, because of increased variance in the data.
3) The optimal precision regarding statistical and hardware efficiency depends on the data set, especially its number of features.
4) Stochastic quantization leads to an unbiased convergence of SGD, compared to biased naive rounding, while the latter does not require any pre-processing of input data.

(III) We experiment with various data sets and algorithmic configurations to explore the training quality vs. performance trade-off. Among other aspects, we look into the number of quantized samples needed, before `qFSGD` can be used. Our key conclusion is that the FPGA-based SGD on stochastically quantized data is a very efficient and well performing way of training dense linear machine learning models.

II. Preliminaries

A. Stochastic Gradient Descent (SGD)

We consider the following problem: Given a dataset $(a_i)_{i=1,N}$ of D-dimensional data points, each with its own label $(b_i)_{i=1,N}$, we wish to identify the dense linear model $\mathbf{x}$ which minimizes the classification loss over this dataset:

$$\arg\min_x Q(x) = \frac{1}{N}\sum_{i=1}^{N} \text{loss}_i(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} \frac{(\mathbf{a_i}\mathbf{x} - b_i)^2}{2} \tag{1}$$

- $\mathbf{a_i} \in \mathbb{R}^{1\times D}$, a single sample of data set $\mathbf{a} \in \mathbb{R}^{N\times D}$
- $b_i \in \mathbb{R}$, the corresponding true inference value to $\mathbf{a_i}$
- $\mathbf{x} \in \mathbb{R}^{D\times 1}$, the model to be trained and used for inference
- $N \in \mathbb{N}$, the number of samples
- $D \in \mathbb{N}$, the number of features per sample

A standard tool for solving this problem is stochastic gradient descent (SGD), consisting of the iterative process in Algorithm 1. For a small enough step size γ, SGD will converge to the *optimal* solution [3].

Data: dataset $a_1, a_2, \ldots, a_N$ of D-dimensional data
Result: optimal value of the model $\mathbf{x}$
Initially, $\mathbf{x}$ is zero;
while *not converged* **do**
  for *i from* 1 *to* N **do**
    $\mathbf{g_i} = \frac{\partial \text{loss}_i(\mathbf{x})}{\partial \mathbf{x}} = (\mathbf{a_i}\cdot\mathbf{x} - b_i)\mathbf{a_i}$;
    $\mathbf{x} \leftarrow \mathbf{x} - \gamma\mathbf{g_i}$;
  end
end

Algorithm 1: Stochastic Gradient Descent

B. Stochastic Rounding (Quantization)

Recent work by Zhang et al. [23] shows that SGD convergence can be guaranteed even if the data undergoes a compression process called *stochastic quantization* before it is used in the gradient update. We now provide a brief explanation of this procedure. Assume that the data consist of floating-point values contained in an interval $[L, U]$. We *quantize* each data point $a_{i,j}$ to one of s levels, as follows. First, we split the interval $[L, U]$ into $s-1$ intervals of equal length $\Delta = (U-L)/(s-1)$. Then, each datapoint is rounded stochastically to one of the endpoints of its interval:

$$Q_s^{L,U}(a_{i,j}) = \begin{cases} (\left\lfloor \frac{a_{i,j}}{\Delta} \right\rfloor + 1)\Delta & \text{with prob. } a_{i,j} - \left\lfloor \frac{a_{i,j}}{\Delta} \right\rfloor \Delta \\ \left\lfloor \frac{a_{i,j}}{\Delta} \right\rfloor \Delta & \text{otherwise.} \end{cases} \tag{2}$$

Example: Quantize value 0.7 between [0,1] with 2 levels. $\Delta = 0.\bar{3}$

$$Q_2^{0,1}(0.7) = \begin{cases} 1 & \text{with prob. } 0.7 \\ 0 & \text{with prob. } 0.3 \end{cases}$$

This quantization procedure is chosen so that the *expected* quantized value returned equals the value itself, that is:

$$\mathbf{E}[Q(a_{i,j})] = a_{i,j}. \tag{3}$$

In other words, if we iterate the quantization procedure on a sample, the average of the returned values would converge to the value of the sample. The key observation by Zhang et al. [23] is that SGD still converges even if samples are quantized in this way. However, to preserve correctness, we must take two independent quantizations Q' and Q'' for each sample a_i, and update the gradient value to:

$$\hat{\mathbf{g}}_i = (Q'(\mathbf{a_i})^T\mathbf{x} - b_i)Q''(\mathbf{a_i}) \tag{4}$$

This choice of update ensures that $\mathbf{E}[\hat{\mathbf{g}}_i] = \mathbf{g}_i$., i.e., the update is an *unbiased estimator* of the true gradient, which in turn ensures convergence of SGD.

III. Implementation

A. Target platform: Intel Xeon+FPGA

Our target platform is the Intel Xeon+FPGA [18] (Figure 1), made available through the *Intel-Altera Heterogeneous Architecture Research Platform*.[1] It combines a 10-core CPU (Intel Xeon E5-2680 v2, 2.8 GHz) and an FPGA (Altera Stratix V 5SGXEA) on a 2-socket motherboard. The FPGA has cache-coherent access to the CPU-connected main memory through QPI (Quick Path Interconnect) with a combined read and write bandwidth of around 6.5 GB/s. The FPGA accelerator, implemented with VHDL, is straightforward to use with Intel-provided software libraries, a QPI endpoint, and an FPGA-based page table of our own implementation: at the start of an application, the required amount of shared memory is allocated by software in 4MB pages and the page table on the FPGA is populated with the corresponding addresses.

[1]Following the Intel legal guidelines on publishing performance numbers, we would like to make the reader aware that results in this publication were generated using preproduction hardware and software, and may not reflect the performance of production or future systems.

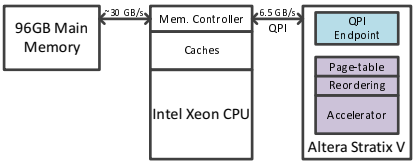


Fig. 1: Intel Xeon+FPGA Architecture

The accelerator can request cache-line wide (64B) reads and writes to the entire memory through the use of the page table and the QPI endpoint, which, in addition to handling the QPI protocol, also implements an FPGA-local cache (128KB two-way associative) using BRAMs. We also added a reorder buffer, since, by default, QPI requests arrive out of order. In addition to an accelerator requesting data from QPI, the software can also issue direct writes, a useful feature for configuring registers containing runtime parameters.

The designs we present in this work are not dependent on the cache-coherency features of the Xeon+FPGA: the processing is done in a streaming fashion, where memory bandwidth, not memory latency, determines performance So, the designs can be integrated into other FPGA platforms (e.g., attached to the network or to storage) with ease.

*B. FPGA-SGD on **float** data (`floatFSGD`)*

We first present an SGD implementation that works on 32-bit floating-point data (Figure 2), a common data representation in machine learning. As the data access width is a 64B cache-line, the circuit is designed to work on that data width. It is able to accept a cache-line at every clock cycle (200 MHz), resulting in an internal processing rate of 12.8 GB/s.

Scale to # of features: The challenging part of the design is to make it capable of handling a number of features D that is larger than 16, which is the default width of the pipeline. This is possible since all vector algebra in Algorithm 1 can be performed iteratively, where each portion contains 16 values. To stay cache-line aligned, we use zero-padding if $D \bmod 16 \neq 0$. Thus, we can calculate how many cache-lines it takes for $\mathbf{a_i}$ (one row in the set) to be completely received:

$$\#\mathbf{a_i} \text{ cache-lines} = \begin{cases} D/16 & \text{if } D \bmod 16 = 0 \\ \frac{D+(16-D \bmod 16)}{16} & \text{if } D \bmod 16 \neq 0 \end{cases}$$

The only parameter determining the scalability of `floatFSGD` is the maximum dimensionality D_{max}, because it determines the amount of BRAM needed for storing the model $\mathbf{x}$. We choose D_{max} to be 8192, which is more than enough for most existing linear dense model training examples. The design can handle any number of samples, *N*, since training is done in a streaming fashion.

Walk-through of computation pipeline: In the following, we explain each stage of the computation pipeline. The first stage is the dot product $\mathbf{a_i} \cdot \mathbf{x} = \sum_{j=1}^{D} a_{i,j} x_j$. When a cache-line containing a part of vector $\mathbf{a_i}$ arrives, it is first multiplied (❶) with the corresponding part of vector $\mathbf{x}$, in floating-point with multiplier IPs,[2] which have 5-cycle latency and throughput of

[2]All floating point IPs are created via Altera Quartus II 13.1.

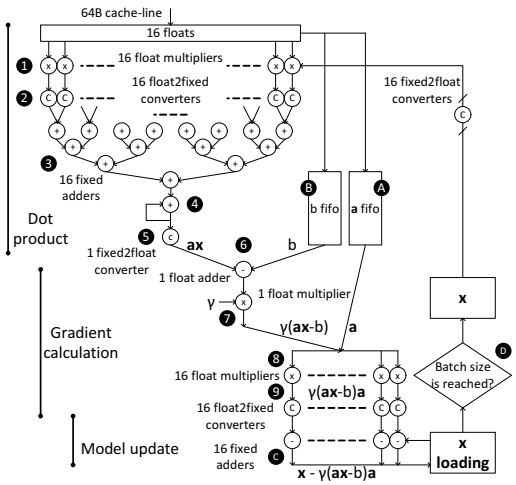


Fig. 2: Computation pipeline for `floatFSGD`, latency: 36 cycles, data width: 64B, processing rate: 64B/cycle.

one result per cycle. Then, the values are converted to a 32-bit integer (❷) by multiplication with a large constant that is configurable during runtime, depending on the value range desired. This *float2fixed* conversion takes 1 cycle. After that, an adder tree (❸) accumulates 16 values, each layer taking 1 cycle. At the output of the adder tree is an accumulator (❹). It accumulates the results coming out of the adder tree for the pre-calculated number $\#\mathbf{a_i}$ cache-lines, building the final value for the dot product. The dot product result is converted back to floating-point (❺), since the next stages of the calculation will be performed with floating-point data. In the next part, the rest of the gradient calculation takes place. First, the scalar value b (the true inference value in the data set), is subtracted from the dot product (❻) using a floating-point adder IP, which has 7-cycle latency and throughput of one result per cycle. The b value is received some cycles before the subtraction takes place and is placed into a FIFO (Ⓑ), waiting there until the dot product result is ready. After the subtraction, a floating-point multiplication takes place with step size γ (❼), which can be configured to any *float* value. At the end of this step, we have a scalar value $\gamma(\mathbf{a_i} \cdot \mathbf{x} - b_i)$, which needs to be multiplied with vector $\mathbf{a_i}$. At this stage, a FIFO (Ⓐ) already contains all parts of vector $\mathbf{a_i}$, because the incoming cache-lines are written to this FIFO simultaneously as they were sent to the dot product calculation. The scalar-vector multiplication (❽) takes place in floating-point, where all parts of $\mathbf{a_i}$ are multiplied with the same scalar value. This gives the gradient $\mathbf{g_i}$ one part at a time, which undergoes *float2fixed* conversion (❾), so that a cycle-by-cycle update of the model $\mathbf{x}$ (Ⓒ) can take place. This would not be possible with a floating-point adder having 7-cycle latency, since the result of the current calculation is needed in the next cycle. The gradient is applied to the corresponding part of the model as it becomes available. After the last part of the gradient is subtracted from the model, the update for $\mathbf{a_i}$ is completed. After all rows go through the same calculation, one epoch is completed (Algorithm 1).

Staleness vs. batch size (as a result of pipelined execution): The model updated and the model read for the dot product are separate (Figure 2). Only when a certain batch size (the number of already processed $\mathbf{a_i}$) is reached, the updated model is carried on to the actual model (❶). The reason is the latency introduced by the computation pipeline: in theory, the whole gradient calculation and the update to the model as in Algorithm 1 should be an atomic operation. However, to exploit deep-pipelining, we don't perform this operation atomically. Instead, we keep the actual model and the updated model separate and carry out the accumulated update only when a certain batch size is reached (called a mini-batch SGD). The batch size is a configurable parameter, which should be set to the latency of the pipeline (36 cycles) to avoid any so-called stale updates.

End-to-end float vs. hybrid computation: We choose a hybrid (*float+fixed*) over end-to-end *float* computation, because a 7-cycle floating-point addition latency leads to: (1) A high latency adder tree (❸) that imposes a larger batch-size to avoid staleness, slowing down the convergence rate, (2) not being able to do a cycle-by-cycle accumulation (❼), since the result of an ongoing addition is required in the next cycle. Thus, to keep the processing rate at 64B/cycle, we choose a hybrid design that eliminates both these disadvantages.

C. FPGA-SGD on ***quantized*** *data (`qFSGD`)*

We explain how we change the `floatFSGD` design to work on quantized data. The main purpose is simple: Instead of reading *float* data (only 16 values in a cache-line), we quantize the data beforehand, so that more than 16 values fit into a cache-line, thus reading less volume of data in total. There is one main challenge in making the FPGA-SGD work on quantized data: scaling out the `floatFSGD` pipeline so that it can work on more than 16 values in parallel. Before we explain how this is achieved, we first review the quantization options we consider and how the data layout looks like.

Quantization for `qFSGD`: Equation (2) shows that, given a non-integer value, the quantization still might produce a non-integer value. However, floating-point arithmetic induced by non-integer values are hard to implement on the FPGA and scaling out such a design would be difficult. We take advantage of the fact that we can select the quantization variables $[L, U]$ and s aptly, so that only integer values are produced. Table I shows our choices for these values.

TABLE I: Choice of quantization levels, lower and upper bounds, so that only integer values are produced.

Levels	Data set positive	Data set negative	Needed bits
s=2	$[L, U] = [0, 1]$	N/A	1, *Q1*
s=3	$[L, U] = [0, 2]$	$[L, U] = [-1, 1]$	2, *Q2*
s=9	$[L, U] = [0, 8]$	$[L, U] = [-4, 4]$	4, *Q4*
s=129	$[L, U] = [0, 128]$	$[L, U] = [-64, 64]$	8, *Q8*

After selecting a quantization precision (one of *Q1*, *Q2*, *Q4* or *Q8*; powers of two to stay cache-line aligned), the data set (the values in matrix $\mathbf{a}$) must be normalized to the selected quantization's corresponding $[L, U]$. At this stage, the sign of the data set is considered for the normalization: we do not normalize a negative data set into a positive interval in order to keep existing zeros (maintain sparsity). Thus, we do not use *Q1* for negative data sets.

The layout of quantized data: As we showed in Section II-B, to calculate the correct gradient, we need 2 quantization samples of the same data point. That is, if we, for example, select *Q8*, a quantized sample has 8 bits and we need 2 of them to calculate the gradient; the actual amount of bits we use is 16. That's why, when we perform quantization on a data set, we always create 2 samples and store them in memory next to each other. Thus, we can calculate how many quantized values can fit into one cache-line, a value we call K, in Table II. The value K dictates the amount of zero-padding we need to perform, in order to be cache-line aligned, similar to as it did for `floatFSGD`. Thus, the number of cache-lines required to receive one quantized row $Q(\mathbf{a_i})$ can be calculated:

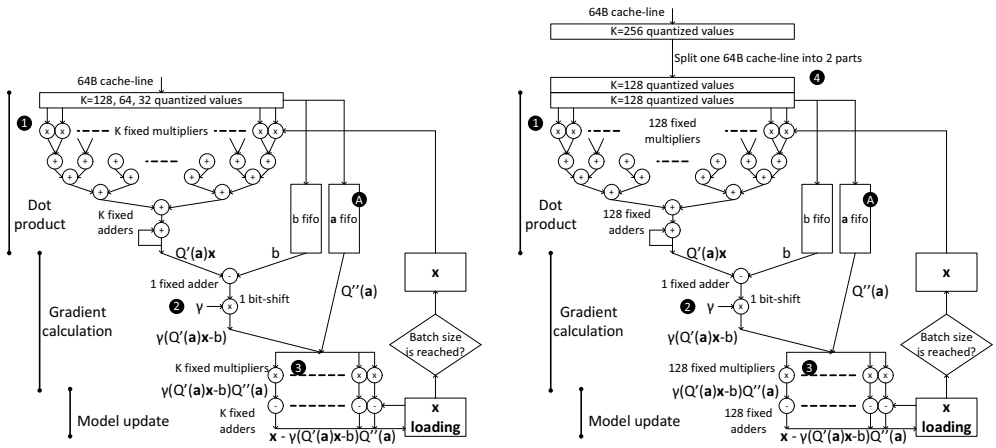
$$\#\mathbf{a_i} \text{ cache-lines}(K) = \begin{cases} D/K & \text{if } D \bmod K = 0 \\ \frac{D+(K-D \bmod K)}{K} & \text{if } D \bmod K \neq 0 \end{cases} \quad (5)$$

TABLE II: Number of received values in a single cache-line.

Data type	*Q1*	*Q2*	*Q4*	*Q8*	*float*
# of values in a cache-line, K	256	128	64	32	16
Processing rate (GB/S), PR	6.4	12.8	12.8	12.8	12.8

When and where does quantization happen?: Unfortunately, there is no way to perform stochastic quantization on the fly and gain the same performance benefits on the target platform. Some naive ideas prove to be useless in this regard: (1) the 10-core CPU can't create samples at the rate of FPGA's memory bandwidth, (2) naively rounding the data once, and then creating stochastically quantized samples on the FPGA is not possible, since quantization depends on the full-precision value itself. Thus, in the current system, the quantization happens as a pre-calculation step on the CPU, before running `qFSGD`. To achieve perfect statistical soundness, we have to create as many quantized samples (so called indexes) of the same data set as the number of epochs. However, creating a separate index for each epoch might be undesirable, because of space or time overheads (exact memory space needed: # of indexes $\times N_{CL}$, from Equation 6). We consider the possibility of reusing indexes and its effect on statistical efficiency in Section IV.

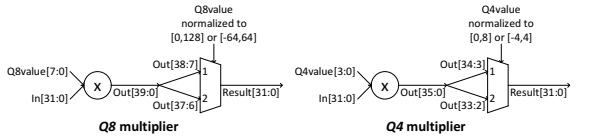
Computation pipeline for quantized data (Figure 3): The selection of *Qx*, which determines the width of the pipeline is a generic parameter that can be set before synthesis. Thus, each *Qx* results in a different bitstream. The explanation here only focuses on the differences from `floatFSGD` and how the pipeline is scaled out. The first thing to note is that *Qx* pipelines work only on integer data, so there is no need for converters. This is because the arriving quantized data $Q(\mathbf{a_i})$ is already in integer form, as explained previously, and the inference values b are also converted to integer by multiplication with a large constant. Another difference here is that for a given value in vector $\mathbf{a_i}$, 2 quantized samples



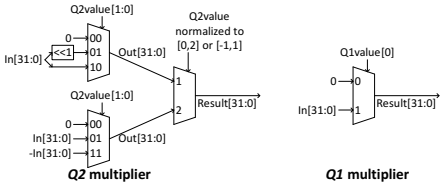
(a) *Q2*, *Q4* and *Q8* qFSGD,latency: $log(K)$+5 cycles, data width: 64B, processing rate: 64B/cycle.

(b) *Q1* qFSGD, latency: 12 cycles, data width: 32B, processing rate: 32B/cycle.

Fig. 3: Computation pipelines for all quantizations. Although for *Q2*, *Q4* and *Q8*, the pipeline width scales out and maintains 64B width, for *Q1* it does not scale out and the pipeline width needs to be halved, making *Q1* qFSGD compute bound.



(a) Signed integer multiplication for *Q8* and *Q4* data, implemented with DSPs.



(b) Signed integer multiplication for *Q2* and *Q1* data, implemented with combinational logic.

Fig. 4: Multiplication implementations depending on the quantization type.

arrive due to the double sampling method. The first sample is given to the dot product calculation (❶) and the second sample is put into a FIFO (Ⓐ), where it is kept until the dot product result is ready, as depicted in Figure 3. The last difference is applying the step size γ (❷), which is actually a division. Since we are dealing with integer data here, we choose to apply γ as a bit-shift operation. By how many bits the value is shifted to the right is a runtime configurable parameter, allowing adjustments according to the data set characteristics.

Scaling out for quantized data and trade-offs: Scaling out the pipeline for *Q8* and *Q4* is straightforward using conventional signed multipliers (❸) implemented by DSP resources, followed by a bit-shift, to keep the data width at 32-bit (Figure 4a). However, for *Q2* and *Q1*, we can do a more efficient multiplication using multiplexers (Figure 4b), since one of the multiplicands is only 2-bit and 1-bit, respectively. Doing this efficient multiplication allows the *Q2* pipeline to scale to 128-value parallelism, which would have otherwise required

TABLE III: Resource consumption for computation pipelines.

Data type	Logic (ALMs)	DSP	BRAM (bits)
float	38% (89194)	12% (33)	7% (3.471K)
Q8	35% (82152)	25% (64)	6% (3.145K)
Q4	36% (84500)	50% (128)	6% (3.145K)
Q2, *Q1*	43% (100930)	1% (2)	6% (3.145K)

a 100% usage of the available DSP resources on the target FPGA (see Table III). However, the pipeline shown in Figure 3a does not scale to 256-value parallelism (we can't meet timing with the target frequency of 200 MHz), even though the *Q1* multiplier is just one multiplexer. The main issue here is (1) the bus for propagating the model from the BRAM to compute units becomes too wide (8192 signals), (2) the adder tree becomes too wide and deep. Note that, we still have to perform addition in full-precision, because we can't simplify the addition as we have done with the multiplication. Using compressor trees [11] instead of standard adder trees also did not help in meeting timing. For this reason, we decided to halve the qFSGD pipeline to process *Q1* data (Figure 3b). To do so, we split an arriving cache-line into 2 parts (❹), which are processed sequentially by the pipeline shown in Figure 3b. The processing rate of this pipeline is 32B/cycle, or 6.4 GB/s, which is slightly less than the memory bandwidth available in our platform.

Model for predicting the speedup with quantized data (hardware efficiency): We can now create a simple model to predict if using quantization will provide any speedup, depending on the dimensionality of the data set, the selected precision, and memory bandwidth. The total number of cache-lines floatFSGD or qFSGD needs to read is:

$$\#\text{ of cache-lines } N_{CL}(K) = N \cdot \#\mathbf{a_i} \text{ cache-lines}(K) \quad (6)$$

The time for each SGD epoch with the processing rate of the circuit (PR) and available platform bandwidth B is thus:

$$T_{epoch} = \begin{cases} N_{CL}(K)/B & \text{if } PR > B \\ N_{CL}(K)/PR & \text{else} \end{cases} \quad (7)$$

The model shows that as long as quantization leads to a reduction of the data size, T_{epoch} can be reduced. An exception to this occurs between *Q1* and *Q2*, when the bandwidth available is larger than 12.8 GB/s. There, *Q1* is not faster than *Q2* (both are compute bound): even when the data size is halved with *Q1*, *Q2* can process twice the data twice as fast. However, *Q1* can still be interesting in platforms with higher memory bandwidth, because one can put multiple `qFSGD` instances, multiplying the bandwidth requirements, making *Q1* still more attractive compared to *Q2*. As for the decision between *Q2*, *Q4*, *Q8*, and *float*, their processing rates are the same, so lower-precision quantization always leads to speedup regardless of bandwidth, unless cache-aligned zero padding causes $\#\mathbf{a_i}$ cache-lines(K) to be the same.

IV. Experimental Evaluation

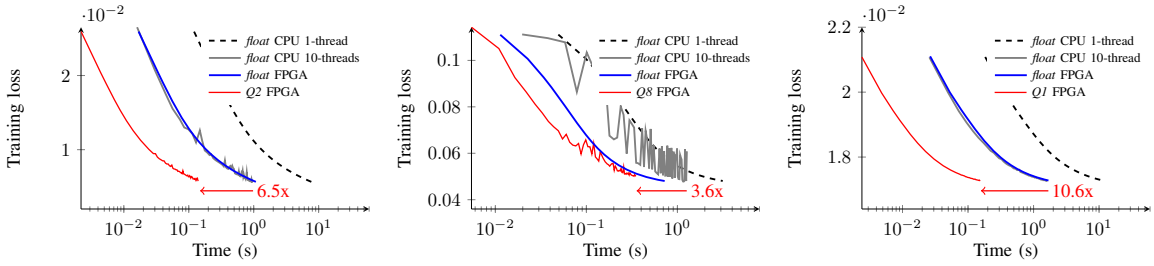
The main hypothesis that we would like to experimentally validate is: (1) low-precision data representation created via stochastic quantization can be used for training dense linear models while maintaining quality, and (2) since the processor doing the training needs to read less data per epoch, using quantized data provides speedup. To validate this, we run our FPGA-SGD on various data sets having different characteristics (see Table IV). As the CPU baseline, we use both a single-threaded SGD doing exactly the same calculation as `floatFSGD` and a high performance parallel library called "Hogwild!" [20] working on *float* data, with a mini-batch size of 36 (equivalent to `floatFSGD`). The multi-thread parallelism in Hogwild! is achieved through asynchronous updates: each thread works on a separate portion of the data and applies gradient updates to a common model without any synchronization. The asynchrony might reduce the statistical efficiency, especially if the data set is dense. Both CPU baselines make use of vectorized instructions and are compiled with GCC 4.8.4, with -O3 enabled.

TABLE IV: Data sets used in experimental evaluation.

Name	Training size	Testing size	# Features	# Classes
cadata	20,640		8	regression
music	463,715		90	regression
synthetic100	10,000		100	regression
synthetic1000	10,000		1000	regression
mnist	60,000	10,000	780	10
gisette	6000	1000	5000	2
epsilon	10,000	10,000	2000	2

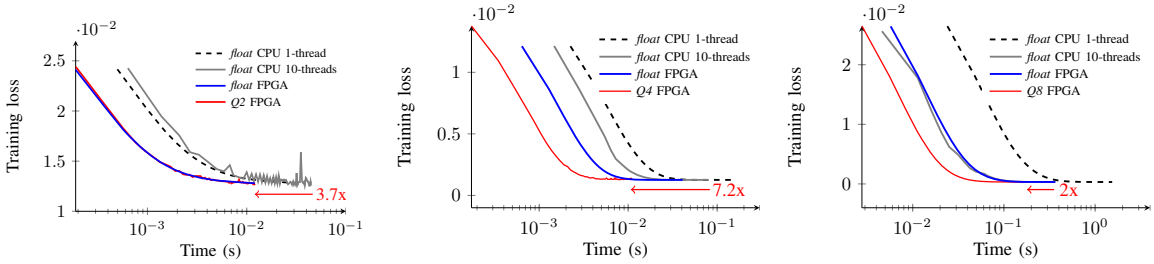
Methodology: Since we apply the step size as a bit-shift operation, we choose one of the following step sizes, which results in the smallest loss for the full-precision data after 64 epochs: $(1/2^6, 1/2^9, 1/2^{12}, 1/2^{15})$. With a given constant step size, we run FPGA-SGD on all the precision variations that we have implemented (*Q1*-only for classification data-, *Q2*, *Q4*, *Q8*, *float*). For each data set, we present the loss function over time in Figures 5 and 6, showing 4 curves: single-threaded and a 10-threaded CPU-SGD for *float*, `floatFSGD`, and `qFSGD` for the **smallest** precision data that **has converged within 1% of the original loss**. We would like to show the difference in time for all implementations to converge to the same loss, emphasizing the speedup we can achieve with `qFSGD` compared to full-precision variants.

Main results: In Figure 5 we observe that for all classification data sets, `qFSGD` achieves a speedup while maintaining convergence quality. For *gisette* in Figure 5a, *Q2* reaches the same loss 6.9x faster than Hogwild!. Due to the high variance data in *epsilon*, both Hogwild! and *Qx* curves seem to be unstable (Figure 5b). We can see that `floatFSGD` in this case behaves well, providing both 1.8x speedup over Hogwild!



(a) *gisette*. $\gamma = 1/2^{12}$. (b) *epsilon*. $\gamma = 1/2^{12}$. (c) *mnist*, trained for digit 7. $\gamma = 1/2^{15}$.

Fig. 5: SGD on classification data. All curves represent 64 SGD epochs. Speedup shown for *Qx* FPGA vs. *float* CPU 10-threads.



(a) *cadata*. $\gamma = 1/2^{12}$. (b) *synthetic100*. $\gamma = 1/2^{9}$. (c) *synthetic1000*. $\gamma = 1/2^{9}$.

Fig. 6: SGD on regression data. All curves represent 64 SGD epochs. Speedup shown for *Qx* FPGA vs. *float* CPU 10-threads.

and better convergence quality. The results for the *music* data set are very similar to *epsilon*, so we omit them for space reasons. On the *mnist* data set (Figure 5c), we can even use *Q1* without losing any convergence quality, showing that the characteristics of the data set heavily effect the choice of quantization precision, which justifies having multiple *Qx* implementations. For *mnist*, *Q1* `qFSGD` provides 10.6x speedup over Hogwild! and 11x speedup over `floatFSGD`. In Figure 6a, `floatFSGD` converges as fast as *Q2* for *cadata*, because they read the same number of cache-lines. The reason is the cache aligned zero padding (set $D = 8$ and respective Ks for *Q2* and *float* into Equation (5)). For *synthetic100* and *synthetic1000*, we need to use *Q4* and *Q8*, respectively, to achieve the same convergence quality as *float* within the same number of epochs. In Figure 6c, Hogwild! convergence is slightly faster than that of `floatFSGD`, and *Q8* provides 2x speedup over that. Hogwild! becomes faster with higher dimensionality (compare Figures 6b and 6c), because with lower dimensionality cache pollution occurs more frequently [20].

The outcome of the main results: We can converge to the same loss using quantized data within the same number of epochs as with full-precision data; thereby achieving better hardware efficiency while maintaining statistical efficiency. To achieve this, the precision has to be selected carefully (which we did empirically). Predicting the ideal precision for a given data set is out of the scope of this work.

Effects of quantized SGD parameters: We now study the effects of various parameters on the convergence quality. We have performed the same experiments on all our data sets and observed similar results. Here we present a subset (due to space constraints) of our experiments.

1) Precision: Previously, we observed that, for some data sets, at least *Q8* precision is needed to be within 1% of the original loss given the same number of epochs. Figure 7a shows how the convergence curves look like, if we insist on using lower precision data on *epsilon*. The inherently higher variance of *epsilon* is amplified by having quantized data, and so statistical efficiency drops. This can be fixed by applying a smaller step size, as we discuss next.

2) Step Size: A higher step size causes higher variance during quantized SGD, since the error introduced by quantization has a greater effect during each gradient update. In the first part of the experimental section, we chose step sizes optimized for *float* precision data. This is why very low precision data does not converge to the original loss for some data sets: the variance caused by *float*-optimized step size is too high. In Figure 7b, SGD for all data sets can converge even with *Q2* data if the step size is chosen to be small enough. The downside of a smaller step size is the slower convergence rate.

3) Dimensionality: Higher dimensional data causes higher variance in `qFSGD`, since the summed-up quantization error for each data sample is greater. We see this effect when comparing Figures 6b and 6c: while for *synthetic100*, *Q4* provides high-quality convergence, for *synthetic1000*, we need at least *Q8*.

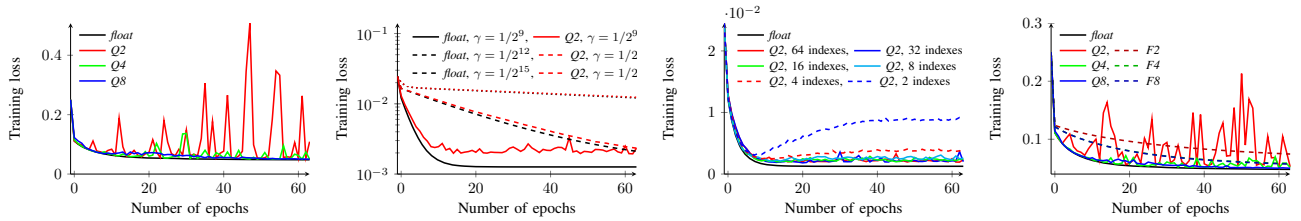
4) Reusing indexes: In all previous experiments, the number of indexes for an SGD run is chosen to be equivalent to the number of epochs for keeping statistical soundness, as explained in Section III-C. Here, we discuss the possibility of reusing indexes, and how it affects SGD convergence quality. Theoretically, reusing indexes of quantized data introduces bias to the gradient, since not every quantized sample is statistically independent. Figure 7c shows the effects of reusing indexes, thereby causing samples not to be statistically independent. We empirically conclude that using more than 16 indexes is enough to get the same convergence quality, but using fewer than 16 causes the convergence curve to be biased.

5) Naive Rounding vs. Quantization: Figure 7d shows the difference between having stochastically quantized data (*Qx*) vs. naively rounded (to the nearest integer) data (*Fx*). If the data is naively rounded, the original optimization problem changes; the global minimum the algorithm converges to is a different one than the original one, showing itself as a bias in the convergence curve.

Classification accuracy: We discuss the accuracy for *mnist*, for which 10 separate models (for each digit) are trained. This can be parallelized with the CPU very well since each model can be trained completely independently. The CPU reaches a processing rate of 10 GB/s for this test, the highest we observe for the CPU. On the FPGA, we need to train 10 models one after the other, since there is only one SGD instance. In Table V, *Q1* achieves 8x speedup against the highest performing CPU implementation we have, while maintaining the same multi-classification accuracy.

V. Related Work

FPGAs have been extensively used for accelerating and prototyping machine learning algorithms. Most of the existing



(a) Effect of different precision quantization on *epsilon*. $\gamma = 1/2^{12}$.

(b) Effect of decreasing the step size on *Q2* with *synthetic100*.

(c) Effect of reusing indexes with *synthetic100*.

(d) Naive rounding vs. stochastic quantization

Fig. 7: SGD on various data sets, showing the effects of data precision, step size γ, index reuse, and naive rounding.

TABLE V: Multi-class classification on *mnist*. Run times are for training 10 models with 100 iterations and $\gamma = 1/2^{15}$.

Precision	Accuracy for 10 digits	Training time (s)
float CPU-SGD	85.82%	19.0354s
Q1 `qFSGD`	85.87%	2.4083s

work focuses on classification with a pre-trained model [12], [13], [15] since classification is thought to be the more frequent operation, often with real-time constraints. As the demand for machine learning applications and data sizes keep growing, accelerating training also becomes highly important. Contrary to common belief, training is an operation that has to be run many times, often as a human-in-the-loop process, until an optimal model is produced.

There is a large body of work considering FPGA as the target processor for linear model and neural network training. Mahajan et al. [14] present Tabla, a framework for automatically generating SGD solvers with different loss functions. The architecture of the generated circuit resembles a MIMD processor with static scheduling, making the approach general purpose. The highest dimensional data set tested is *mnist*. No absolute performance numbers are presented. Kesler et al. [9] focus on accelerating linear algebra operations and present simulation results of a CPU-integrated accelerator architecture. A maximum matrix size of 150 is presented. Roldao et al. [21] consider accelerating a conjugate gradient algorithm, which takes symmetric matrices of a maximum size 58 as input. Cadambi et al. [4] consider using low precision (naive rounding) arithmetic, achieving a maximum of 256-value parallelism with 4-bit precision, for support vector machine (SVM, also a linear classifier) training. Their FPGA accelerator is implemented as a coprocessor for accelerating the multiply-accumulate part of the algorithm. Bin Rabieah et al. [2] implement SVM training with custom precision data (4-bit and 8-bit, via naive rounding). The parallelism on the FPGA is exploited by partitioning the data, and then training multiple models with each partition and, at the end, aggregating the models, similarly to the asynchronous update method used for achieving parallelism for SGD on a CPU. Majumdar et al. [16] present a framework—MAPLE, not necessarily designed for FPGAs, but using an FPGA as a prototyping platform. Its architecture is based on a many core with large on-chip memory, enabling highly parallel computation. The FPGA-SGD we presented achieves the highest dimensional scalability among existing work, considering reported test data sets and workloads.

To our knowledge, the only efforts considering stochastic quantization as a viable data representation for deep neural network training and combining this with an FPGA accelerator are presented by Gupta et al. [8] and Courbariaux et al. [5]. Gupta et al. [8] use the FPGA as a matrix multiplication accelerator working on quantized data. Courbariaux et al. [5] focus on reducing multiplication precision on FPGAs, not necessarily on stochastically quantized data. In contrast to these, our study focuses on the detailed analysis of statistical vs. hardware efficiency trade-offs when using stochastically quantized data for dense linear model training on FPGAs.

VI. Conclusion

We present various highly scalable and parameterizable FPGA-based stochastic gradient descent implementations for performing linear model training. We open source our designs for the community.[3] Our key takeaways are: (1) The most efficient way of training linear models on an FPGA is through the usage of low-precision data obtained with stochastic quantization. (2) Doing so opens up a trade-off space: precision vs. end-to-end runtime, convergence quality, design complexity, data and system properties. (3) The multivariate trade-off space motivates new research focusing on how to optimize a linear model training system on FPGAs and heterogeneous systems.

Acknowledgements: We would like to thank Ji Liu and Hantian Zhang for their contributions. The HARP v1 prototype used in the paper has been generously donated by Intel.

References

[1] D. Alistarh, J. Li, R. Tomioka, et al. QSGD: Randomized Quantization for Communication-Optimal Stochastic Gradient Descent. *arXiv'16*.
[2] M. Bin Rabieah and C.-S. Bouganis. FPGASVM: A Framework for Accelerating Kernelized Support Vector Machine. In *BigMine'16*.
[3] L. Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In *COMPSTAT'10*.
[4] S. Cadambi, I. Durdanovic, V. Jakkula, et al. A Massively Parallel FPGA-Based Coprocessor for Support Vector Machines. In *FCCM'09*.
[5] M. Courbariaux, J.-P. David, and Y. Bengio. Training Deep Neural Networks with Low Precision Multiplications. *arXiv'14*.
[6] C. M. De Sa, C. Zhang, K. Olukotun, and C. Ré. Taming the Wild: A Unified Analysis of Hogwild!-style Algorithms. In *NIPS'15*.
[7] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, et al. Large-scale FPGA-based Convolutional Networks. *Scaling up Machine Learning: Parallel and Distributed Approaches*, 2011.
[8] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep Learning with Limited Numerical Precision. *CoRR'15*.
[9] D. Kesler, B. Deka, and R. Kumar. A Hardware Acceleration Technique for Gradient Descent and Conjugate Gradient. In *SASP'11*.
[10] J. K. Kim et al. Hardware Acceleration of Iterative Image Reconstruction for X-ray Computed Tomography. In *ICASSP'11*.
[11] M. Kumm and P. Zipf. Pipelined Compressor Tree Optimization Using Integer Linear Programming. In *FPL'14*.
[12] C. Kyrkou et al. Embedded Hardware-Efficient Real-Time Classification with Cascade Support Vector Machines. *NNLS'16*.
[13] B. Lesser, M. Mücke, and W. N. Gansterer. Effects of Reduced Precision on Floating-Point SVM Classification Accuracy. *Proc. Comp. Sc.'11*.
[14] D. Mahajan, J. Park, E. Amaro, et al. TABLA: A Unified Template-based Framework for Accelerating Statistical Machine Learning. In *HPCA'16*.
[15] D. Mahmoodi, A. Soleimani, H. Khosravi, M. Taghizadeh, et al. FPGA Simulation of Linear and Nonlinear Support Vector Machine. *JSEA'11*.
[16] A. Majumdar, S. Cadambi, M. Becchi, S. T. Chakradhar, and H. P. Graf. A Massively Parallel, Energy Efficient Programmable Accelerator for Learning and Classification. *TACO'12*.
[17] M. Mücke et al. Peak Performance Model for a Custom Precision Floating-Point Dot Product on FPGAs. In *Euro-Par'10*.
[18] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, et al. A Reconfigurable Computing System Based on a Cache-Coherent Fabric. In *ReConFig'11*.
[19] A. Pérez-García et al. Multilayer Perceptron Network with Integrated Training Algorithm in FPGA. In *CCE'14*.
[20] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS'11*.
[21] A. Roldao et al. A High Throughput FPGA-based Floating Point Conjugate Gradient Implementation for Dense Matrices. *TRETS'10*.
[22] Y. You, X. Lian, J. Liu, H.-F. Yu, et al. Asynchronous Parallel Greedy Coordinate Descent. In *NIPS'16*.
[23] H. Zhang, K. Kara, J. Li, D. Alistarh, et al. ZipML: An End-to-end Bitwise Framework for Dense Generalized Linear Models. *arXiv'16*.

[3] http://www.systems.ethz.ch/fpga/ZipML_SGD