HHS Public Access

Author manuscript

Proc Int Symp Comput Archit. Author manuscript; available in PMC 2018 January 30.

Published in final edited form as:

Proc Int Symp Comput Archit. 2017 June ; 2017: 561–574.

Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent

Christopher De Sa, **Matthew Feldman**, **Christopher Ré**, and **Kunle Olukotun**

Departments of Electrical Engineering and Computer Science, Stanford University

Abstract

Stochastic gradient descent (SGD) is one of the most popular numerical algorithms used in machine learning and other domains. Since this is likely to continue for the foreseeable future, it is important to study techniques that can make it run fast on parallel hardware. In this paper, we provide the first analysis of a technique called Buckwild! that uses both asynchronous execution and low-precision computation. We introduce the DMGC model, the first conceptualization of the parameter space that exists when implementing low-precision SGD, and show that it provides a way to both classify these algorithms and model their performance. We leverage this insight to propose and analyze techniques to improve the speed of low-precision SGD. First, we propose software optimizations that can increase throughput on existing CPUs by up to 11×. Second, we propose architectural changes, including a new cache technique we call an obstinate cache, that increase throughput beyond the limits of current-generation hardware. We also implement and analyze low-precision SGD on the FPGA, which is a promising alternative to the CPU for future SGD systems.

Keywords

Stochastic gradient descent; low precision; asynchrony; multicore; FPGA

1 INTRODUCTION

Stochastic gradient descent (SGD) is a ubiquitous optimization algorithm used in a wide variety of applications, notably as part of the famous backpropagation algorithm for training neural networks [4, 6, 42]. SGD and its variants form a critical component of enterprise machine learning systems, such as MLbase [47], Project Adam [7], and Google Brain [24]. Additionally, it is used in finance [13] and other analytics domains, in systems such as GraphLab [30], MadLib [17], which is used by Cloudera Impala and Pivotal, and Vowpal Wabbit [2], which is developed at Microsoft. Since these billion-dollar industries depend on dataflows which rely in part on, and are often bottlenecked [5, 56] by, SGD, it is important for systems designers to study techniques to make SGD run efficiently.

CCS CONCEPTS

• **Computer systems organization→Multicore architectures**; *Reconfigurable computing*; • **Computing methodologies→Machine learning algorithms**; • Mathematics of computing → **Continuous optimization**;

Algorithm 1

Stochastic gradient descent

Require: Initial model $w \in \mathbb{R}^n$, input dataset $x \in \mathbb{R}^{n \times m}$ loss function f, and step size $\eta \in \mathbb{R}$.

1: **for** $k = 1$ **to** NumPasses **do**
2: **for** $i = 1$ **to** m **do**
3: Compute a gradient estimate: $g = \nabla f(w; x_i)$
4: Update the model: $w \leftarrow w - \eta \cdot g$
5: **end for**
6: **end for**
7: **return** w

Concretely, SGD is used for solving optimization problems, wherein the goal is to find a model vector w that minimizes a given loss function. As shown in Algorithm 1[1], it operates by updating the model vector w repeatedly in a sequential loop based on vectors x_i from an input dataset.

In order to enhance the performance of SGD, it is important to consider both the current properties and the design trajectory of hardware systems. Over the past decade, due to the breakdown of Dennard scaling, computer hardware has been trending towards more parallel, specialized architectures [49]. Unfortunately, despite the simplicity of its update rule, Algorithm 1 is a sequential algorithm, so it is unlikely to perform well on this parallel hardware— and generic compiler and architectural techniques cannot fix this problem because they cannot alter the semantics of the algorithm. To address this, it is common to consider variants of SGD which are modified to run in parallel [36].

In this paper, we analyze the performance of a new SGD variant that combines parallel *asynchronous* execution with *low-precision* computation, a technique called Buckwild! [11]. In Buckwild!, multiple worker threads execute the inner loop of Algorithm 1 (lines 2–5) asynchronously *without locking*; this exploits multi-core parallelism. Also, the real numbers in Algorithm 1 are represented by low-precision fixed point numbers[2], which enables higher memory throughput and better utilization of SIMD parallelism.

Despite Buckwild!'s promising benefits in terms of improving the parallelism and memory throughput of SGD, both these techniques cannot be used naively, since they change the semantics of the original algorithm. In order to apply them, we need to be confident that the modified algorithm will still produce a useful answer. There are reasons to think that the modified algorithm will be error-prone: the low-precision computation introduces round-off error and the asynchronous execution may produce race conditions. Fortunately, several recent papers that analyze asynchronous SGD [31, 36] and low-precision SGD [9, 11, 14] show, both empirically and theoretically, that this extra round-off error often does not significantly impact the quality of the output.

[1]Practical SGD applications differ from Algorithm 1 in that the step size η typically decreases over time. Since this and other minor changes do not significantly affect the hardware behavior of SGD, we will not discuss them further in this paper.
[2]Rather than by 32- or 64-bit floating point numbers as is standard.

Unfortunately, just knowing that low-precision SGD is a valid strategy is not enough. There are many choices that must be made when implementing this algorithm and when designing hardware for it. These decisions include setting the precision of the variables, distributing work across parallel resources, and choosing how to perform the rounding when we lower the precision of a number. Changing these implementation details for a Buckwild! SGD algorithm effects a trade-off between the speed at which the hardware can execute an update step and the quality of the resulting solution. We call these metrics *hardware efficiency* and *statistical efficiency*, respectively. [3] While there has been significant theoretical analysis of the statistical efficiency of asynchronous low-precision algorithms, their hardware efficiency has not been explored in depth—this is particularly true for low-precision computation, which has received less attention from SGD researchers and practitioners than asynchronous execution. As we will show, the decisions made when implementing a Buckwild! algorithm can have a significant effect (up to 11×) on its hardware efficiency, and the optimal choices can depend on the structure of the input dataset—for example, the sparsity of the input can affect the optimal design. There has been no principled way of reasoning about these decisions, and past analyses have focused on a particular problem or hardware target in ad hoc ways.

To address this issue, we introduce a principled way of relaxing precision in SGD, called the *DMGC model*. Specifically, "DMGC" is an acronym that identifies four different ways in which arithmetic precision can be reduced: by quantizing the input **d**ataset (x_i), the **m**odel (w), the intermediate **g**radient values, or the interworker **c**ommunications. These ways can be combined arbitrarily in a particular implementation of SGD, and the best-performing system often uses different levels of precision for the different categories. The DMGC model serves as both a taxonomy of existing low-precision implementations, and a way of reasoning about the trade-off space that exists when designing new systems. Additionally, it gives us predictive power, as with a roofline model [53], to estimate the performance of an algorithm by classifying it as being either *bandwidth-bound* or *communication-bound*.

Leveraging insight from the DMGC model, we analyze four software techniques that can be used to produce highly efficient Buckwild! implementations on modern CPUs: (1) hand-optimizing the SIMD code, (2) using fast random number generation, (3) disabling prefetching, and (4) combining multiple iterations into a single *mini-batch* update. To improve the performance of this algorithm beyond what is possible in software, we also suggest two hardware enhancements: introducing new compute instructions, and relaxing cache coherence by randomly ignoring invalidate requests, a strategy we call an *obstinate cache*. To further study how architecture relates to SGD performance, we test Buckwild! on the FPGA, which is a promising alternative to the CPU for next-generation SGD implementations.

In this paper, we study asynchronous, low-precision SGD, making the following contributions:

[3]This nomenclature follows previous work [15, 56] which examined this trade-off in a different setting.

- We introduce the DMGC model, and show how it can be used to estimate the throughput of a Buckwild! implementation with a roofline-like model.
- We describe four software optimizations that can be used to improve the performance of Buckwild! SGD on current-generation CPUs by up to 11×.
- We suggest two hardware enhancements, including a new strategy for cache coherency we call an *obstinate cache*, that can improve the performance of this algorithm beyond what is possible in software. We also illustrate the benefits of low-precision computation on the FPGA, and present useful design techniques.
- We evaluate our methods in several real settings, including deep learning. We show that, with our suggested optimizations, using low-precision can produce near-linear speedups (up to 4×) over full-precision.

2 BACKGROUND AND RELATED WORK

In this section, we will describe asynchronous low-precision SGD in detail, and discuss prior work related to this algorithm. SGD is used for minimizing a function that can be written as a sum of many components, specifically

$$\text{minimize:} \sum_{i=1}^{m} f(w;x_i) \quad \text{subject to: } w \in \mathbb{R}^n. \tag{1}$$

To simplify our analysis throughout this paper, we will focus on a particular version of this problem, *logistic regression* [52]: given data examples $(x_i,y_i) \in \mathbb{R}^n \times \{-1, 1\}$, we want to solve

$$\text{minimize:} \sum_{i=1}^{m} \log(1+e^{-y_i x_i^T w}) \quad \text{over: } w \in \mathbb{R}^n.$$

For this problem, the SGD updates are of the form

$$w \leftarrow \underbrace{w+\eta\, y_i x_i (1+\exp(y_i \overbrace{x_i^T w}^{\text{DOT}}))^{-1}}_{\text{AXPY}}.$$

From a hardware perspective, the cost of this step will be dominated by the two vector operations, a dot product and an AXPY (*a*-*x*-plus-*y* operation); the remainder of the work is in negligible scalar computations. Many other problems can be solved using SGD with a single dot-and-AXPY pair (in addition to negligible scalar computation), including linear regression and support vector machines (SVM). Because of this, SGD on logistic regression has a hardware efficiency that is representative of SGD on any problem in this class. Even

problems with more complicated updates will typically have performance similar to logistic regression, since more complicated SGD steps typically consist of similar linear algebra operations (such as matrix multiply).

The computational structure of SGD can also vary based on whether the input dataset is *dense* or *sparse*. Dense datasets have examples x_i that are represented simply as a array of n numbers, while sparse datasets have examples x_i with mostly zero entries, so it is cheaper to instead store a list of only the nonzero entries. Since dot and AXPY algorithms on dense and sparse vectors differ substantially in terms of their memory access patterns, it is natural that the overall performance of SGD for these two cases will also differ. Throughout this paper we will consider dense and sparse datasets separately.

Next, we will describe asynchronous execution and low-precision computation individually, using a simple implementation of SGD. For dense logistic regression, sequential, full-precision SGD might be implemented as in Figure 1.

Asynchronous execution—Asynchronous execution is a widely-used technique also known as Hogwild! [36] (on which the Buckwild! name was based). Hogwild! SGD could use the same code as in Figure 1; it differs from sequential SGD in that multiple threads each run `sgd_worker`in parallel, sharing a single copy of the model vector `w`. Because the model is accessed without locking, race conditions can occur if one thread writes the model `w`while another thread is computing its own update. On well-behaved problems, Hogwild! is known to both "achieve a nearly optimal rate of convergence" (statistical efficiency) and run "an order of magnitude" faster than methods that use locking (hardware efficiency) [11, 31, 36]. This impressive speedup has inspired a flurry of research into asynchronous SGD across problem domains, including deep learning [37], PageRank approximations [34], and recommender systems [54]. Fast asynchronous variants of other algorithms have also been proposed, such as coordinate descent [27, 28] and Gibbs sampling [10, 19]. Hogwild! has been successfully applied in industry, such as in Microsoft's Project Adam [7].

Low-precision computation—Reduced-precision SGD can be implemented using the code in Figure 1 by simply changing each red **`float`** data type to a low-precision, fixed-point type, such as `int8_t`. Additionally, casts would need to be added to lines 6 and 10 to convert the low-precision numbers safely to and from `float`. Because the conversion in the AXPY operation decreases the number of bits used to represent the numbers, it introduces round-off error, which is especially significant when the precision of the model is small. Additional round-off error can occur implicitly at the start of the algorithm, when the dataset is rounded to a low-precision type. While low-precision SGD has received somewhat less research attention than asynchronous SGD, basic results that characterize its statistical efficiency are still known [11]. Additionally, several systems have been suggested for using low-precision arithmetic for deep learning and other problems [9, 14, 45, 46, 48]. Later, we will examine these systems in more detail in terms of our DMGC model.

Other settings—While we focus here on the performance of SGD on a single CPU or FPGA, much previous work exists that analyzes (full-precision) SGD in other settings. For example, Zhang and Ré [56] analyzed the trade-offs that exist when running asynchronous

SGD on non-uniform memory access (NUMA) machines. Similar work exists for algorithms running on clusters [15] and on GPUs [20, 57]. When designing a system that uses SGD, it is important to understand both how the large-scale structure of the available compute resources affect the performance, as well as how optimizations can improve the performance of individual chips. For this reason, we believe that our contributions in this paper, especially when combined with previous work, will be useful to system designers.

3 THE DMGC MODEL

In this section, we describe our main conceptual contribution, the DMGC model, and describe how low-precision systems described in previous work can be classified thereby. The main idea behind the DMGC model is that the real numbers[4] used by a parallel SGD algorithm can be separated into four distinct groups: numbers used to store the ***d****ataset*, numbers used to represent the ***m****odel*, numbers used as intermediate values while computing the ***g****radients*, and numbers used to ***c****ommunicate* among the several worker threads. This categorization is natural because these numbers are both used differently by the algorithm and stored differently within the memory system, and so making them low-precision will have different effects on performance.

Dataset numbers—Dataset numbers are those used to store the input dataset, the x_i in (1) or the examples ex from Figure 1. As inputs to the algorithm, they are constant, and they compose the vast majority of data in the process's live data at any given time. Since there are so many of them and they are reused only infrequently, dataset numbers are typically stored in DRAM, and we focus our analysis on problems for which this is the case[5]—such as those targeted by popular in-memory ML frameworks, including SciKit Learn [39]. Because dataset numbers are constant inputs, to make them low-precision we need to quantize them only once: either at the beginning of the algorithm, if the input is stored on disk as full-precision floats; or before the algorithm runs, if we are given a low-precision version of the input dataset to load. For some applications, such as recommender systems and compressed sensing where the input dataset is naturally quantized, this can be done without any loss of fidelity; however, in general quantizing the dataset can affect the statistical efficiency of the algorithm. We call the precision of the dataset numbers, measured in bits, the *dataset precision*.

When solving a dense problem, the input dataset consists only of dataset numbers; however, when solving a sparse problem, the dataset also contains values that encode the indexes of the nonzero entries of the example vectors. These integer values also can be made low-precision[6], and since this does not change the semantics of the input dataset, doing so incurs no loss of statistical efficiency. We call the precision of these values the *index precision*.

[4]Throughout this section, we use the word "numbers" to refer specifically to values that represent real numbers in the algorithm, and not to values that represent indexes or counters.
[5]For very small problems, the dataset could be stored in the last-level cache, and for very large problems it would not fit in DRAM and so need to be stored on disk, but since the trade-off space is very different in these rare cases we do not address them here.
[6]For model sizes too large to be indexed by the low-precision type, this can be achieved by storing the *difference* between successive nonzero entries. Since this part of the implementation did not significantly impact throughput in our experiments, we do not discuss it further in this paper.

Using low-precision for the dataset is advantageous from a hardware efficiency perspective. Since most numbers read from DRAM are dataset numbers, representing them in low-precision both decreases the amount of DRAM bandwidth needed to execute the algorithm, and decreases the amount of pressure on the entire cache hierarchy. This will improve performance when SGD is memory bound.

Model numbers—Model numbers are those used to store the model, the w in (1) and Figure 1. In general, model numbers include any mutable state that persists across iterations. Unlike dataset numbers, model numbers are modified continuously throughout the algorithm, and while they make up only a small fraction of the process's live data, they represent a significant part of its working set since every model number is frequently reused. Because of this, being able to effectively cache the model is important for achieving fast execution, and the model numbers are typically all stored in the last-level cache; we focus on problems for which this is possible. We call the precision of the model numbers the *model precision*.

In order to make the model numbers low-precision, it is necessary to quantize by rounding every time the model is written, i.e., every time the AXPY on line 4 of Algorithm 1 is executed. There are two different ways we can do this rounding. The first is standard rounding, also known as nearest-neighbor or biased rounding, which rounds to the closest number that is representable in the low-precision model type. The second, *unbiased* rounding, randomly rounds up or down in such a way that the expected value of the quantized output is equal to the input. Unbiased rounding, which has been used in some [14] previous work on low-precision SGD, must be implemented using a pseudorandom number generator (PRNG), which decreases its hardware efficiency; however, it typically yields more accurate solutions (higher statistical efficiency) than biased rounding. Later, in Section 5.2, we will show how by using an extremely fast PRNG we can make the hardware efficiency cost of unbiased rounding negligible for many applications.

Using a low-precision model has similar advantages to using a low-precision dataset. Having smaller model numbers puts less pressure on the cache hierarchy, and may allow a model to fit in cache when it otherwise would not. Additionally, computing the gradient updates on the CPU can be cheaper with a lower-precision model, since more SIMD parallelism can be extracted for operations producing 8-bit or 16-bit numbers.

Gradient numbers—Gradient numbers are those used as intermediate values while computing the update step, such as `xi_dot_w`and `scale_a`in Figure 1. Unlike with the dataset or the model, which typically have a single precision, it often makes sense to use different precisions for different gradient numbers in an algorithm. Depending on how they are used, making these numbers low-precision may or may not have an effect on statistical efficiency, and their effect on hardware efficiency is similarly context-dependent.

Communication numbers—Communication numbers are those used to communicate among worker threads in a parallel algorithm. Sometimes, this communication is done explicitly, in which case we call its precision the *communication precision*. However, in many implementations, such as in Figure 1 and in standard Hogwild!, communication is not

explicit; instead, the coherence protocol of the CPU cache hierarchy is employed to communicate asynchronously between cores. In this case, there are no communication numbers— and inasmuch as they exist, they will have the same precision as the model, since they are just model numbers communicated by the cache coherence protocol.

DMGC signatures—Using the four classes of numbers outlined above, we can classify a particular implementation of SGD in terms of the precision of its numbers within each class. This classification, which we call a *simplified DMGC signature*, is written as

$$D^{\text{dataset prec}}\left[i^{\text{index prec}}\right]M^{\text{model prec}}G^{\text{gradient prec}}C^{\text{comm prec}}.$$

The i term is included only if the problem is sparse, and the $[i]$ notation means the problem could possibly be sparse. For example, a dense implementation that uses an 8-bit dataset, a 16-bit model, and explicitly computes and communicates with 32-bit floats would have signature $D^8M^{16}G^{32}C^{32}$.

The information in a DMGC signature is enough to model the statistical efficiency of an algorithm from first principles by using techniques from previous work like De Sa et al. [11]. However, as it is a simplified model, this type of signature does not encode everything we want to represent about an algorithm from a hardware perspective. To address this, we augment the simplified signature with rules that capture more information about precision:

- Since floating-point and fixed-point numbers differ, we suffix an f to the size of floating-point numbers.
- When any explicit synchronization is done among workers, we add a s subscript to the C; absence of an s implies asynchronous execution. We can omit the C entirely if, as in Hogwild!, the algorithm relies entirely on the cache hierarchy for implicit communication.
- For simplicity, we omit the G term entirely if the gradient computation is equivalent to using full-precision numbers (i.e. no fidelity is lost in intermediate values). Similarly, we can leave out the D and M terms when the algorithm uses full-precision arithmetic for those numbers.

Using these rules, we can assign any implementation a *DMGC signature*. For example, standard sparse Hogwild! has signature $D^{32f}i^{32}M^{32f}$ and a dense Buckwild! implementation using 8-bits for the dataset and the model and unbiased rounding has signature D^8M^8.

3.1 Classifying previous implementations

In this subsection, we will briefly discuss some low-precision systems implemented in previous work, and how they can be understood under the DMGC model. First, we analyze Seide et al. [46], in which the gradients are “quantized...to but one bit per value” and these gradient values, rather than model values, are used to communicate synchronously among the workers. Since it maintains a full-precision dataset and model, which includes a full-precision representation of the quantization error that is carried forward across iterations,

this algorithm has DMGC signature $G^1C_s^1$. Note that this signature gives us a clearer understanding of the precision used in this algorithm than the title of the paper, which only calls it "1-Bit" SGD—but does not specify which numbers are so quantized.

Another implementation from previous work is SGD using low-precision multiplications, suggested in Courbariaux et al. [9]. The most successful implementation analyzed by the authors uses 10-bit multipliers, but full-precision accumulators; since the inputs and outputs to the multipliers are intermediate numbers, its DMGC signature is just G^{10}.

In Table 1, we list the DMGC signatures of several algorithms from previous work. While most of these papers considered several ways to set the precision, none highlight the full trade-off space described by the DMGC model.

4 MODELING PERFORMANCE

In this section, we describe how the DMGC model can be used to approximate the performance of well-optimized SGD on parallel hardware. Throughout the rest of this paper, we will represent hardware efficiency in terms of the *dataset throughput*, the rate at which data numbers are processed by the algorithm, measured in giga-numbers-per-second (GNPS). For logistic regression where the sizes of the dataset vectors and the model vector are the same, the dataset throughput is equal to the rate at which iterations can be performed multiplied by the model size.

In order to explore the trade-offs generated by varying the precision of SGD, we tested our best general implementations[7], using the precisions listed in Table 2, for both dense and sparse (3% density[8]) artificially-generated datasets[9] of model sizes n (i.e. $w \in \mathbb{R}^n$) ranging from 2^8 to 2^{26}.

Changing the model size has a non-uniform effect on throughput, which we have illustrated in Figure 2. For large models (roughly those larger than 256K in our experiments) changing the model size has little effect on performance. In this regime, the throughput is *bandwidth-bound*, since its performance is limited by the memory bandwidth of the individual cores. On the other hand, for small models, decreasing the model size causes a degradation in performance. In this regime, the throughput is *communication-bound*; its performance is limited by the latency at which updates, which happen more frequently for smaller model sizes, can be sent between the cores.

We can use this intuition to model the throughput of Buckwild! as parameters are changed. Our performance model has the following properties: (1) varying the thread count results in a throughput that follows Amdahl's law [3],

[7]From the optimizations we will discuss in Section 5, we used only hand-optimized SIMD and XORSHIFT rounding; these are the optimizations that are generally applicable, regardless of the problem or model size.
[8]Our choice of density is arbitrary, and similar effects would be observed across a range of densities.
[9]We generated the datasets by sampling from the generative model [35] for logistic regression, using a true model vector w^* and example vectors x_i all sampled uniformly from $[-1, 1]^n$.

$$T_t = \frac{T_1}{((1-p)) + \frac{p}{t}}, \quad (2)$$

where T denotes the throughput, t is the number of threads, and p is the *parallelizable fraction* of the task; (2) the base throughput T_1 is solely a function of the DMGC signature; and (3) the parallelizable fraction p is solely a function of the model size. For the hardware we used, a Xeon® E7-8890 v3 with 18 physical cores running at 2.50 GHz, we found that a good approximation for p was

$$p = \min\left(0.98, 0.15 \cdot \log\left(\frac{\text{model size}}{256}\right)\right). \quad (3)$$

The first term here describes the fixed bandwidth bound, which is independent of the model size. The second term describes the communication bound, which manifests as a decrease in the parallelizable fraction of the algorithm because increasing the thread count makes communication more frequent. This assignment of p, together with the base throughputs T_1 listed as a function of the DMGC signature in Table 2, seems to yield valid predictions for both dense and sparse problems, across all well-optimized SIMD implementations we tried.

Figure 3 compares the measured throughputs of our Buckwild! implementations with the predictions of the performance model, for a selection of thread counts. More broadly, for both dense and sparse datasets, for 90% of the tested algorithm parameters, the prediction was within 50% of the observed throughput. It is perhaps surprising that a model with so fewparameters manages to track the measured performance reasonably accurately. However, this makes sense when we consider that lowering the precision is done with the goal of extracting SIMD parallelism—that is, parallelism within a single core—and so effects that operate across many cores, such as the thread count and the model size (which affects performance primarily through cache coherence effects), should not interact strongly with the precision.

Because of this, we can roughly evaluate the effect of changing the precision, even across a variety of model sizes and thread counts, by just looking at the base throughput number in Table 2. In particular, we can gauge the performance against the best-case theoretical speedup, wherein the throughput is inversely proportional to the number of bits; we call this *linear speedup*. The data in Table 2 show that linear speedup is achieved for dense Buckwild!, and that while sparse SGD shows less than linear speedup as the precision is decreased, $D^8I^8M^8$ Buckwild! is still the fastest scheme. Since these base throughputs are directly proportional to the throughputs predicted by (2), the illustrated speedups are valid across all model sizes.

5 SOFTWARE OPTIMIZATIONS

While there are known techniques for writing efficient Hogwild! implementations [56], there are additional non-obvious optimizations that increase throughput in the low-precision case. In this section, we present two low-precision-specific optimizations that are generally applicable, and which were necessary to achieve the performance described in Section 4. First, we will show that care is needed when vectorizing low-precision SGD, and that hand-vectorized code can be significantly faster than what a compiler would generate. Second, we describe how unbiased rounding can be done with minimal effect on hardware efficiency by using very high-throughput pseudorandom number generators. We also introduce two additional techniques that can further improve the performance when the model size is small (and performance is dominated by cache effects).

5.1 Efficient SIMD computations

A major goal of using low-precision computation is to leverage the ever-widening SIMD capabilities of modern CPUs. In this subsection, we discuss optimizations that improve performance on CPUs that use the AVX2 SIMD instruction set extensions, the newest SIMD extension available on Xeon processors. Unfortunately, on AVX2, a straightforward C++ implementation doesn't fully utilize the capabilities of the processor for lower precisions even when compiled by gcc with `-Ofast`, the highest optimization level. Worse, other compilers (we tested icc and clang) and frameworks (we tested OpenMP) do not seem to significantly improve the quality of the generated code. A hand-optimized implementation that implements the dot and AXPY operations using AVX2 intrinsics—effectively programming in assembly—is necessary to achieve the performance reported in Section 4.

Figure 4a compares the performance of our hand-optimized implementation with GCC's compilation of generic code. As can be seen, GCC consistently underperforms by an significant factor. Since the AVX2 optimizations used in the hand-optimized version don't change the semantics of the algorithm, its speedup is essentially *free*: it doesn't involve any trade-off with statistical efficiency. The DMGC signatures for which it was effective to hand-optimize the implementation are listed in Figure 4c, along with the average (across models and thread counts) speedups that resulted.

To understand this performance gap, we will analyze how the dot operation is implemented in both the GCC and the hand-optimized versions of 8-bit Buckwild! In the hand-optimized version, the numerical computations are done using a *fused multiply-add*, instruction, **`vpmaddubsw`**. This instruction multiplies two pairs of 8-bit numbers, and accumulates the results—with no loss of precision— into a single 16-bit number. The GCC version does not use a fused multiply-add; instead, to dot two 8-bit SIMD vectors, it: (1) converts the 8-bit numbers into 32-bit floats, in the process quadrupling the size of each input and thus expanding it into four vector registers, (2) multiplies the floating point vectors, and (3) sums the resulting floating point numbers. Since each of these steps requires multiple instructions, the GCC version takes almost a dozen instructions to accomplish what the hand-optimized version does in a single instruction. Similar differences in instruction usage occur throughout the code emitted by GCC, which explains the nearly 10× speedup achieved by hand-optimizing the SIMD instructions.

This difference in performance is not simply incidental to the implementation of GCC, but rather can be attributed to the language semantics of C++. This is because in C++, directly multiplying two 8-bit integers (for example) can lead to a loss in fidelity, since it produces an 8-bit result that could possibly overflow. To prevent this, it is necessary to first *cast* the 8-bit numbers to 16-bit numbers, and then do the multiply. This makes it impossible to write a fused multiply-add with basic C++ constructs. Furthermore, GCC does not optimize aggressively enough to transform the code to use the **`vpmaddubsw`** instruction. It would be unreasonable to expect GCC, or a similar general-purpose compiler, to perform this transformation, since sometimes (for example, the small-model-size sparse problems in Figure 4b) it can actually lower the performance of the code. Because of the above concerns, we recommend handwriting the SIMD code of the core operations of any Buckwild! implementation.

5.2 Fast random number generation

In Section 3, we described how choosing between biased and unbiased rounding can trade-off between statistical and hardware efficiency. While biased rounding always maximizes hardware efficiency with no regard for statistical efficiency, unbiased rounding offers additional design decisions that determine how the randomness used for rounding is generated. In this subsection, we discuss these decisions, which allow for finer-grained trade-offs between statistical and hardware efficiency. The simplest way of implementing unbiased rounding is by using the formula

$$Q(x)=\text{to\_ low\_ precision}(\text{floor}(x+\text{rand}())). \quad (4)$$

where x is the full-precision number to round, $Q(x)$ is the low-precision output, floor(z) returns the largest integer smaller than z, and rand() returns an independent random variable uniformly distributed on [0, 1].[10]

The hardware efficiency of an implementation of (4) depends primarily on how the rand function is implemented. The easiest way to implement this in C++ is to use a pseudorandom number generator (PRNG) available in the popular Boost library [1]. In this implementation, a fresh random number is generated by a call to Boost's default PRNG (Mersenne twister [33]) every time we write a model number: n times per iteration, where n is the model size. Even though Mersenne twister is a fast PRNG, if it runs once every write, it dominates the computation cost of the algorithm. Worse, there is no obvious way to transform the Boost implementation of the PRNG into a hand-optimized AVX2 implementation, and, as described in Subsection 5.1, the C++ compiler is unlikely to do it efficiently.

To improve the performance of the quantizer, we used a hand-written AXV2 implementation of XORSHIFT [32], a very fast, but not very statistically reliable [38] PRNG. This very

[10]For simplicity, we are here assuming that we are quantizing to integer precision; rounding to fixed-point numbers with different quanta is a straightforward extension.

lightweight generator has similar statistical efficiency to the Mersenne twister, as shown in Figure 5a, while significantly improving upon its hardware efficiency, as shown in Figure 5b. Unfortunately, since the rest of the computations required by low-precision SGD are so cheap, running even a very lightweight generator like XORSHIFT on every write still makes up a significant fraction of the compute instructions of the algorithm. This means that this strategy still has much lower hardware efficiency than biased rounding.

A third strategy that further improves the performance of the quantizer is to *share randomness* among multiple rounded numbers. In this design, the calls to the rand function in (4) are no longer independent; rather, it will return the same number some number of times before eventually running the XORSHIFT PRNG to generate a fresh number.[11] Despite the lack of independence, the quantized output for each element remains unbiased, and the method has surprisingly good statistical efficiency; as shown in Figure 5a it can be close to the other two strategies. Furthermore, since the PRNG is no longer called at each write, its cost is amortized, allowing us to match the hardware efficiency of the unbiased version in some cases, as shown in Figure 5b. This strategy is used to achieve the performance numbers reported in Section 4. One benefit of this approach is that we can expose a smooth trade-off between statistical and hardware efficiency by changing the frequency at which the PRNG is run.

5.3 Turning off prefetching

So far, the optimizations we have discussed in this section have been focused on improving the memory bandwidth and SIMD parallelism, and thereby the base throughput, of the algorithm. However, as Figure 3 illustrates, when the program is communication-bound, the throughput is almost an order of magnitude less than when the model is large. This decrease in performance is attributable to cache effects: when the model is small, lines in the L2 caches that store model numbers are more frequently invalidated, leading to processor stalls as the cores must wait for data from the shared L3. For the remainder of this section, we will discuss two techniques that can improve the throughput when the algorithm is communication-bound.

One way to improve throughput that requires minimal programmer effort is to simply *turn off the hardware prefetcher*. For processors using recent Intel microarchitectures, this can be achieved by setting bits in the model specific register (MSR) `0x1A4`[50].[12] While this effect may seem surprising, it has been known to happen for some applications [26]. Since the hardware prefetcher typically increases the throughput of the memory subsystem, it is understandable when we consider the facts that: (1) the additional memory operations inserted by the prefetcher consume a significant amount of bandwidth; and (2) the cache lines loaded by the prefetcher are often invalidated before they can be used.

Figures 6a and 6b report the throughput that can be achieved by turning off hardware prefetching for dense and sparse problems, respectively. As can be seen, significant

[11] In our tests, we ran the vectorized XORSHIFT PRNG once every iteration to produce 256 fresh bits of randomness, which we shared for rounding throughout the AXPY operation.

[12] Note that while this MSR provides more fine-grained control of which features of the prefetcher to turn on and off, for all model sizes we tried it was optimal to either turn all the features off (no prefetching at all) or keep them all on (the default setting).

speedups of up to 150% can occur. Furthermore, our experiments showed that turning off the prefetcher does not have a significant effect on statistical efficiency—in fact, the distributions of the quality of the output were indistinguishable from one another. Note that we did not need to change any of our code to do this: they were measured using the *same executable* and differing only in the assigned value of the prefetch control MSR. This means that this technique improves hardware efficiency essentially for free (requiring no programmer effort), and so we recommend that SGD implementers always try disabling the prefetcher when model sizes are small.

5.4 Increase the mini-batch size

Mini-batch stochastic gradient descent is a straightforward variant of SGD in which gradients from multiple dataset examples are summed, resulting in an update rule like

$$w \leftarrow w - \alpha \left(\nabla f(w; x_i) + \nabla f(w; x_{i+1}) + \cdots + \nabla f(w; x_{i+B-1})\right)$$

Here, B, the *mini-batch size*, is a hyperparameter that determines how many examples will be used to compute each model update (for standard SGD, the mini-batch size is just $B = 1$). Since more compute work is done for each time the model is written, increasing the mini-batch size will amortize the cache effects caused by writing to a small model. Specifically, the model is written less frequently, and so L2 cache lines will be invalidated correspondingly less frequently.

Figure 6d illustrates the speedups that can result from using a larger mini-batch size. For very large mini-batch sizes, the throughput for smaller models approaches that of larger models; this scheme effectively increases the parallelizable fraction p of the algorithm.

Unlike some other optimizations, increasing the mini-batch size can effect the statistical efficiency. This relationship is often problem dependent and difficult to capture. For logistic regression, Figure 6e shows the measured statistical efficiency as the mini-batch size is changed. These results suggest that an empirical or theoretical analysis of the accuracy is needed to decide how large the minibatch size can be set before statistical efficiency degrades.

6 HARDWARE OPTIMIZATIONS

In this section, we show how we can improve throughput with two hardware changes that can be used in combination with the software optimization techniques presented in Section 5. The first proposed change affects compute by adding new ALU instructions, while the second affects memory by proposing a new way of relaxing the cache coherence protocol. In contrast to previous work on new ISAs for neural network accelerators [29] and relaxed consistency shared memory [51], our changes are simple and could be added to any existing architecture. It is our hope that these or similar hardware changes may actually be implemented in future CPU generations.

6.1 New vector ALU instructions

The performance improvements from hand-optimized SIMD code depend on the existence of efficient instructions like the fused-multiply add described in Section 5.1. Were this and similar instructions not to exist in AVX2, it would be impossible to improve over the code generated by GCC, which means that fully optimized Buckwild! systems would run significantly slower. In this subsection, we ask the opposite question: can we add compute instructions that will improve the throughput of low-precision SGD?

The most obvious new ALU instructions to add would be ones that allow the inner loops of the dot and AXPY operations to be computed using fewer instructions. Here, we focus on the D^8M^8 case—the one for which instructions are most lacking on current architectures—and propose two specific instructions to do this: one, for dot, which vertically multiplies signed 8-bit integer vectors, producing 16-bit intermediate values, which it then horizontally adds in groups of four to produce 32-bit floating point numbers; and another, for AXPY, which multiplies an 8-bit vector by an 8-bit scalar, producing 16-bit intermediate values, which it then adds to a hardware-generated pseudorandom 8-bit vector, before truncating to produce an 8-bit output. These instructions are sufficient to compute the inner loop bodies of dot and AXPY with one and two instructions, respectively, so they represent an upper bound on the speedup that can result from new ALU instructions.

In order to evaluate these instructions, we ran test programs on our Xeon processor by using an existing ALU instruction (`vpmaddwd`for the new dot instruction, `vpmullw`for the AXPY instruction) as a proxy in place of the new instruction in our code. By doing so, we are supposing that our new ALU instruction will have the same latency as the chosen proxy. If this is the case, then since the proxied instruction only operates on numbers, and does not affect the control flow of the program, the runtime of the proxy program will be exactly the same as the runtime of the program with the new instruction. Thus, while the proxy program produces invalid output, it lets us accurately measure the runtime. In our experiments, these new instructions consistently improved throughput by 5% − 15%.

We can also consider another type of new ALU instruction: those which enable us to run at different precisions than we could otherwise use. Specifically, we are interested in running 4-bit SGD, that is, D^4M^4. This choice is infeasible on current-generation CPUs because AVX2 does not support any kind of 4-bit arithmetic. We used the same methodology as before to test the performance of a hypothetical 4-bit Buckwild! implementation, assuming the existence of 4-bit multiply, add, and fused-multiply-add instructions, all of which have the same latency characteristics as their 8-bit equivalents (which we used as proxies for our experiments). Figure 5c compares the throughput of this dense D^4M^4 implementation to D^8M^8; across most settings, it is about 2× faster (although it often affects statistical efficiency).

6.2 Relaxing coherence: the obstinate cache

In Sections 5.3 and 5.4, we explored software techniques that can address the deleterious cache effects that occur when the algorithm is communication-bound. It is natural to consider hardware changes that can further ameliorate these harmful cache effects. Here, we

propose a simple change that relaxes the coherence of the cache hierarchy—for only those cache lines used to store model numbers[13]—by just *randomly ignoring some fraction of invalidates*. Under this strategy, which we call an *obstinate cache* because it obstinately refuses to respond to invalidate requests, whenever a cache receives a signal that would normally cause it to change a model cache line to the invalid (I) state, with some probability q (the obstinacy parameter), using a hardware PRNG, it instead retains that cache line in the shared (S) state. While this technique makes the caches incoherent (which causes race conditions), we can show that cache incoherence has a negligible effect on statistical efficiency by using the same analysis that shows that the race conditions from asynchronous execution only marginally affect statistical efficiency.

In order to evaluate this technique of relaxed coherence, we ran experiments using ZSim [44], a popular architectural simulator that excels at modeling memory hierarchies. Using ZSim, we simulated an 18-core processor with the same compute characteristics and approximately the same cache characteristics as our 2.5 GHz Xeon processor: a 32 KB 4-cycle latency L1 cache, 256 KB 12-cycle latency L2 cache, and a 45 MB 36-cycle latency shared L3 cache. We used the same code used in Section 4, except that since ZSim does not model a hardware prefetcher, we manually added software prefetching. While the simulation does not model congestion, it does model a coherency protocol (MESI) and so it does exhibit a slowdown caused by invalidates as the model becomes smaller, as shown in Figure 6c. The same figure illustrates how using an obstinate cache can improve throughput: for values of q around 50%, the cost of running with a small model disappears. On real hardware, which may experience additional negative effects from invalidates (such as bus congestion and shared L3 cache bandwidth limitations) that are not modeled by the simulator, we expect the effect of the obstinate cache will be even more dramatic. Furthermore, as shown in Figure 6f, we observed that the obstinate cache has no detectable effect on statistical efficiency, even when q is as high as 95%. These results suggest that hardware that allows for software-controlled relaxation of the cache coherence, even in such a course-grained way as the obstinate cache, could be a useful tool for achieving good performance for low-precision SGD.

7 EVALUATION

In this section we will display the effects of our ideas on some popular problems. First, we will demonstrate that Buckwild! can make deep learning more efficient. Almost all deep learning systems, including CNNs [23] and ResNets [16], are bottlenecked by the training of *convolution layers*; this has been verified experimentally [8]. For this reason, we use the throughput of a convolution layer as a proxy for the hardware efficiency of the system. We measured this throughput for a convolution layer[14] running on images of size 227×227×3 from the ImageNet dataset [43]. We expect that low-precision would yield a linear increase in throughput. Figure 7a shows that this is in fact the case, and that our optimizations are necessary to achieve this speedup.

[13]The obstinate cache behavior could be enabled per-page based on whether the user sets a flag in the page table.
[14]The layer we studied is structured identically to the first convolution layer from Caffe's AlexNet example [18].

Next, we evaluate the effect of low precision on statistical efficiency for neural networks. We study this effect by measuring the test error for LeNet, a successful CNN architecture [25]. To do this, we modified Mocha [55], a deep learning library, to simulate low-precision arithmetic of arbitrary bit widths. Since this simulation was too slow to use ImageNet, we tested on the smaller MNIST [12] and CIFAR10 [22] digit classification tasks. Convolution layers for these datasets have speedups similar to those in Figure 7a: for both MNIST and CIFAR10, we observed $D^{16}M^{16}$ and $D^{8}M^{8}$ having 2.0× and 3.0× speedup, respectively, over full-precision. We expect that that using a 16-bit model (for all the layers) will result in quality indistinguishable from full-precision. Our experiments show that this is the case, and we show in Figure 7b that it is possible to train accurately even below 8-bits, using unbiased rounding. This is a surprising result, as some previous work has suggested that training at 8-bit precision is too inaccurate [9, 14].

One common alternative to deep learning for classification tasks is the kernel support vector machine (SVM). We hypothesized that, as with logistic regression, Buckwild! would have little effect on statistical efficiency in this setting. We evaluated our techniques by running kernel SVMs[15] on MNIST using the random Fourier features technique [41], a standard proxy for Gaussian kernels. To study the statistical efficiency, we measured both the average training loss and the test error when using all our software optimizations (and 18 threads). Our results, which are displayed in Figure 7d (training loss) and Figure 7e (test error), show that 16-bit ($D^{16}M^{16}$) Buckwild! achieves accuracy that essentially matches full-precision computation, and 8-bit ($D^{8}M^{8}$) produces results that are within a percent of full-precision. We also observed runtimes similar to those in Figure 3; compared to the 32-bit floating point version, the 16-bit and 8-bit versions ran 3.3× and 5.9× faster, respectively. This illustrates that Buckwild! has higher throughput than Hogwild! while producing results with similar accuracy.

8 BUCKWILD BEYOND THE CPU

To see how Buckwild! could be implemented if we were free of the architectural constraints of a modern CPU, we studied its performance on an FPGA. On the FPGA, we are able to freely explore the various components of the DMGC model. Specifically, we can: (1) perform arithmetic operations on data types of any precision and reclaim freed logic resources when doing so; (2) operate with SIMD operations that are effectively any length; and (3) compress memory usage directly by using low-precision without incurring overhead for unbiased rounding.

We started by creating a high-level, parameterized description of linear regression SGD (which has the same compute structure as logistic regression), focusing on the case where the model can fit in on-chip block RAM.[16] We then compiled this description down to VHDL using the DHDL framework [21, 40], which uses heuristic search to choose optimal parameters for a particular design.

[15]We ran ten such SVM classifiers, one for each digit, in a standard one-versus-all system.
[16]This is analogous to the model fitting in the L3 cache on the CPU.

In order to design this implementation of the algorithm, we used the concepts presented in previous sections to guide our design. The DMGC model was a useful guide for systematically understanding how statistical efficiency is impacted by precision choices. On the FPGA, we can exploit arbitrarily large SIMD parallelism, as well as local XORSHIFT modules as discussed previously. We can also design modules for performing exactly the same operations that we propose as new ALU instructions. Some of the other optimizations, such as cache considerations, do not apply to the FPGA. With the algorithm and the above concepts in mind, there are still a few new challenges that are unique to the FPGA implementation and had to be explored. First, we had to decide whether to use standard SGD or mini-batch SGD (as in §5.4). In hardware, these two implementations generate very different designs due to the way memory is managed and control signals are generated. In regular SGD, we only need to perform one dot product and one AXPY per model update. This is only acceptable if the model size is large enough to amortize the cost of issuing a new memory command for sequential bursts for every iteration. If the model is small, then we can combine multiple iterations into a single memory request. This means that mini-batch SGD will have more throughput, as an individual worker can work on multiple examples in between each model update. However, this means that each update requires two matrix multiplies, rather than just a dot product and AXPY. We empirically found that for our FPGA, mini-batch SGD has the highest throughput unless a single data vector spans at least 100 DRAM bursts. As on the CPU, though, mini-batch may negatively affect statistical efficiency.

Second, with either of the two implementations, we must match the volume of data being read with the volume of data being processed. Every data element we load from main memory must be read twice per update: once to compute the error of the current model and then again to compute the update. The second step depends on the result of the first step. Therefore, we can either choose to divide the design into two stages, data-load and data-process, where the data-process stage must consume data twice as fast as the off-chip load, or three stages, off-chip-load, error-compute, and update-compute, where the three stages must consume data at the same rate and asynchronously communicate to each other when they are finished. The designs are illustrated in Figure 7c. The three-stage design requires the second stage to copy data from the BRAM it reads from to the BRAM that the third stage reads from so that the third stage can compute the correct update given the error that stage two passes along. Thus, it is a better design when compute logic is scarce but BRAM is abundant. However, since the two-stage design does not need to make a redundant copy of the data, it is a better candidate when BRAM is scarce.

Figure 7f shows that our optimized designs have higher throughput (by up to 2.5×), but use less FPGA resources, as the precision decreases. Similarly, when keeping the model precision fixed, halving the dataset precision improves both throughput and area. This illustrates the benefits of setting precision using the DMGC model on the FPGA. Furthermore, the performance per watt is better for this algorithm on the FPGA. Using an Altera Stratix V GS 5SGSD8, we acheived an average of 0.339 GNPS/watt, while the implementation on a Xeon E7-8890 acheived 0.143 GNPS/watt.

9 CONCLUSION

In this paper, we studied the performance of the asynchronous, low-precision variant of stochastic gradient descent. Understanding this technique is becoming increasingly important for system architects as SGD becomes increasingly dominant within machine learning and other domains. We introduced a new conceptual framework for classifying precision, the DMGC model, and showed how it can be used to both clarify existing techniques, and model the throughput of new implementations. With insight from this model, we proposed several software optimizations and hardware changes (summarized in Table 3) that can improve the performance of a Buckwild! implementation by up to 11×. We also showed that low-precision computation can be useful for SGD beyond the CPU, and described techniques that were useful to achieve good performance on an FPGA.

Acknowledgments

The authors acknowledge the support of: DARPA FA8750-12-2-0335; NSF IIS-1247701; NSF CCF-1111943; DOE 108845; NSF CCF-1337375; DARPA FA8750-13-2-0039; NSF IIS-1353606; ONR N000141210041 and N000141310129; NIH U54EB020405; Oracle; NVIDIA; Huawei; SAP Labs; Sloan Research Fellowship; Moore Foundation; American Family Insurance; Google; and Toshiba.

"The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, AFRL, NSF, ONR, NIH, or the U.S. Government."

References

1. BOOST C++ Libraries. http://www.boost.org
2. Agarwal, Alekh, Chapelle, Olivier, Dudík, Miroslav, Langford, John. A Reliable Effective Terascale Linear Learning System. CoRR. 2011 abs/1110.4198.
3. Amdahl, Gene M. Validity of the single processor approach to achieving large scale computing capabilities. Proceedings of the April 18–20, 1967, spring joint computer conference; ACM; 1967. p. 483-485.
4. Bottou, Léon. Stochastic gradient learning in neural networks. Proceedings of Neuro-Nımes. 1991; 91:8.
5. Bottou, Léon. COMPSTAT' 2010. Springer; 2010. Large-scale machine learning with stochastic gradient descent; p. 177-186.
6. Bottou, Léon. Neural Networks: Tricks of the Trade. Springer; 2012. Stochastic gradient descent tricks; p. 421-436.
7. Chilimbi, Trishul, Suzue, Yutaka, Apacible, Johnson, Kalyanaraman, Karthik. Project Adam: Building an Efficient and Scalable Deep Learning Training System. 11th OSDI; USENIX Association; 2014. p. 571-582.
8. Chintala, Soumith. [Accessed: 2016-11-16] convnet-benchmarks. https://github.com/soumith/convnet-benchmarks. (????)
9. Courbariaux, Matthieu, David, Jean-Pierre, Bengio, Yoshua. Training deep neural networks with low precision multiplications. 2014 arXiv preprint arXiv:1412.7024.
10. De Sa, Christopher, Ré, Christopher, Olukotun, Kunle. Ensuring Rapid Mixing and Low Bias for Asynchronous Gibbs Sampling. ICML; 2016; 2016.
11. De Sa, Christopher, Zhang, Ce, Olukotun, Kunle, Ré, Christopher. Taming the Wild: A Unified Analysis of Hogwild!-Style Algorithms. NIPS. 2015
12. Deng, Li. The MNIST database of handwritten digit images for machine learning research. IEEE Signal Processing Magazine. 2012; 29(6):141–142.
13. Dixon, Matthew, Klabjan, Diego, Bang, Jin Hoon. Classification-based Financial Markets Prediction using Deep Neural Networks. 2016; 2016 arXiv preprint arXiv:1603.08604.

14. Gupta, Suyog, Agrawal, Ankur, Gopalakrishnan, Kailash, Narayanan, Pritish. Deep Learning with Limited Numerical Precision. ICML; 2015; 2015.
15. Hadjis, Stefan, Zhang, Ce, Mitliagkas, Ioannis, Ré, Christopher. Omnivore: An Optimizer for Multi-device Deep Learning on CPUs and GPUs. CoRR. 2016 abs/1606.04487.
16. He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, Sun, Jian. Deep residual learning for image recognition. 2015 arXiv preprint arXiv:1512.03385.
17. Hellerstein, Joseph M., Ré, Christoper, Schoppmann, Florian, Wang, Daisy Zhe, Fratkin, Eugene, Gorajek, Aleksander, Ng, Kee Siong, Welton, Caleb, Feng, Xixuan, Li, Kun, et al. The MADlib analytics library: or MAD skills, the SQL. Proceedings of the VLDB Endowment. 2012; 5(12): 1700–1711.
18. Jia, Yangqing, Shelhamer, Evan, Donahue, Jeff, Karayev, Sergey, Long, Jonathan, Girshick, Ross, Guadarrama, Sergio, Darrell, Trevor. Caffe: Convolutional Architecture for Fast Feature Embedding. 2014 arXiv preprint arXiv:1408.5093.
19. Johnson, Matthew, Saunderson, James, Willsky, Alan. Analyzing hogwild parallel gaussian Gibbs sampling. NIPS; 2013. p. 2715-2723.
20. Kaleem, Rashid, Pai, Sreepathi, Pingali, Keshav. Stochastic gradient descent on GPUs. Proceedings of the 8th Workshop on General Purpose Processing using GPUs; ACM; 2015. p. 81-89.
21. Koeplinger, David, Delimitrou, Christina, Prabhakar, Raghu, Kozyrakis, Christos, Zhang, Yaqi, Olukotun, Kunle. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. ISCA 2016: 43rd International Symposium on Computer Architecture; 2016.
22. Krizhevsky, Alex, Hinton, Geoffrey. Learning multiple layers of features from tiny images. 2009
23. Krizhevsky, Alex, Sutskever, Ilya, Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems. 2012:1097–1105.
24. Le, Quoc V. Building high-level features using large scale unsupervised learning. 2013 IEEE international conference on acoustics, speech and signal processing; IEEE; 2013. p. 8595-8598.
25. LeCun, Yann, Bottou, Léon, Bengio, Yoshua, Haffner, Patrick. Gradient-based learning applied to document recognition. Proc IEEE. 1998; 86(11):2278–2324.
26. Lee, Jaekyu, Kim, Hyesoon, Vuduc, Richard. When prefetching works, when it doesn't, and why. ACM Transactions on Architecture and Code Optimization (TACO). 2012; 9(1):2.
27. Liu, Ji, Wright, Stephen J. Asynchronous Stochastic Coordinate Descent: Parallelism and Convergence Properties. SIOPT. 2015; 25(1):351–376.
28. Liu, Ji, Wright, Stephen J., Ré, Christopher, Bittorf, Victor, Sridhar, Srikrishna. An Asynchronous Parallel Stochastic Coordinate Descent Algorithm. JMLR. 2015; 16:285–322.
29. Liu, Shaoli, Du, Zidong, Tao, Jinhua, Han, Dong, Luo, Tao, Xie, Yuan, Chen, Yunji, Chen, Tianshi. Cambricon: An instruction set architecture for neural networks. Proceedings of the 43rd International Symposium on Computer Architecture; IEEE Press; 2016. p. 393-405.
30. Low, Yucheng, Gonzalez, Joseph E., Kyrola, Aapo, Bickson, Danny, Guestrin, Carlos E., Hellerstein, Joseph. Graphlab: A new framework for parallel machine learning. 2014 arXiv preprint arXiv:1408.2041.
31. Mania, Horia, Pan, Xinghao, Papailiopoulos, Dimitris, Recht, Benjamin, Ramchandran, Kannan, Jordan, Michael I. Perturbed Iterate Analysis for Asynchronous Stochastic Optimization. 2015 arXiv preprint arXiv:1507.06970.
32. Marsaglia, George. Xorshift RNGs. Journal of Statistical Software. 2003; 8:1.
33. Matsumoto, Makoto, Nishimura, Takuji. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation (TOMACS). 1998; 8(1):3–30.
34. Mitliagkas, Ioannis, Borokhovich, Michael, Dimakis, Alexandros G., Caramanis, Constantine. FrogWild!: Fast PageRank Approximations on Graph Engines. PVLDB. 2015
35. Ng, Andrew Y., Jordan, Michael I. On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes. Advances in Neural Information Processing Systems. 2002; 14

36. Niu, Feng, Recht, Benjamin, Re, Christopher, Wright, Stephen. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. NIPS; 2011. p. 693-701.
37. Noel, Cyprien, Osindero, Simon. Dogwild!–Distributed Hogwild for CPU & GPU. 2014
38. Panneton, François, L'ecuyer, Pierre. On the xorshift random number generators. ACM Transactions on Modeling and Computer Simulation (TOMACS). 2005; 15(4):346–361.
39. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research. 2011; 12:2825–2830.
40. Prabhakar, Raghu, Koeplinger, David, Brown, Kevin J., Lee, HyoukJoong, De Sa, Christopher, Kozyrakis, Christos, Olukotun, Kunle. Generating configurable hardware from parallel patterns. Proceedings of 21st ASPLOS; ACM; 2016. p. 651-665.
41. Rahimi, Ali, Recht, Benjamin. Random features for large-scale kernel machines. Advances in neural information processing systems. 2007:1177–1184.
42. Rumelhart, David E., Hinton, Geoffrey E., Williams, Ronald J. Learning representations by back-propagating errors. Nature. 1986
43. Russakovsky, Olga, Deng, Jia, Su, Hao, Krause, Jonathan, Satheesh, Sanjeev, Ma, Sean, Huang, Zhiheng, Karpathy, Andrej, Khosla, Aditya, Bernstein, Michael, Berg, Alexander C., Fei-Fei, Li. ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV). 2015; 115(3):211–252. https://doi.org/10.1007/s11263-015-0816-y.
44. Sanchez, Daniel, Kozyrakis, Christos. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. ACM SIGARCH Computer Architecture News. 2013; 41(3):475–486.
45. Savich, Antony W., Moussa, Medhat. Resource efficient arithmetic effects on rbm neural network solution quality using mnist. 2011 International Conference on Reconfigurable Computing and FPGAs; IEEE; 2011. p. 35-40.
46. Seide, Frank, Fu, Hao, Droppo, Jasha, Li, Gang, Yu, Dong. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. INTERSPEECH; 2014. p. 1058-1062.
47. Sparks, Evan R., Talwalkar, Ameet, Smith, Virginia, Kottalam, Jey, Pan, Xinghao, Gonzalez, Joseph, Franklin, Michael J., Jordan, Michael I., Kraska, Tim. MLI: An API for distributed machine learning. 2013 IEEE 13th International Conference on Data Mining; IEEE; 2013. p. 1187-1192.
48. Strom, Nikko. Scalable Distributed DNN Training Using Commodity GPU Cloud Computing. Sixteenth Annual Conference of the International Speech Communication Association; 2015.
49. Sutter, Herb. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal. 2005; 30:3.
50. Viswanathan, Vish. Disclosure of H/W prefetcher control on some Intel processors. https://software.intel.com/en-us/articles/disclosure-of-hwprefetcher-control-on-some-intel-processors. (????)
51. Vora, Keval, Koduru, Sai Charan, Gupta, Rajiv. ASPIRE: exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM. ACM SIGPLAN Notices. 2014; 49:10, 861–878.
52. Walker, Strother H., Duncan, David B. Estimation of the probability of an event as a function of several independent variables. Biometrika. 1967; 54(1–2):167–179. [PubMed: 6049533]
53. Williams, Samuel, Waterman, Andrew, Patterson, David. Roofline: an insightful visual performance model for multicore architectures. Commun ACM. 2009; 52(4):65–76.
54. Yu, Hsiang-Fu, Hsieh, Cho-Jui, Si, Si, Dhillon, Inderjit S. Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems. ICDM; 2012. p. 765-774.
55. Zhang, Chiyuan. [Accessed: 2016-11-16] Mocha.jl: Deep Learning for Julia. 2016. https://devblogs.nvidia.com/parallelforall/mocha-jl-deep-learning-julia/
56. Zhang, Ce, Ré, Christopher. DimmWitted: A study of main-memory statistical analytics. PVLDB. 2014

57. Zhang, Shanshan, Zhang, Ce, You, Zhao, Zheng, Rong, Xu, Bo. Asynchronous stochastic gradient descent for DNN training. 2013 IEEE International Conference on Acoustics, Speech and Signal Processing; IEEE; 2013. p. 6660-6663.

```
void sgd_worker(float* w, float eta, bool& running) {
  while(running) {
    struct { float* x; float y; } ex=get_example();
    // compute the dot product of ex.x and w
    float xi_dot_w=0.0;
    for(long k=0; k<N; k++) xi_dot_w+=ex.x[k]*w[k];
    // do the logistic regression scalar computation
    float scale_a=eta*ex.y/(1.0+exp(ex.y*xi_dot_w));
    // update the model with an AXPY
    for(long k=0; k<N; k++) w[k]+=ex.x[k]*scale_a;
} }
```

Figure 1.
C++ code for SGD on logistic regression.

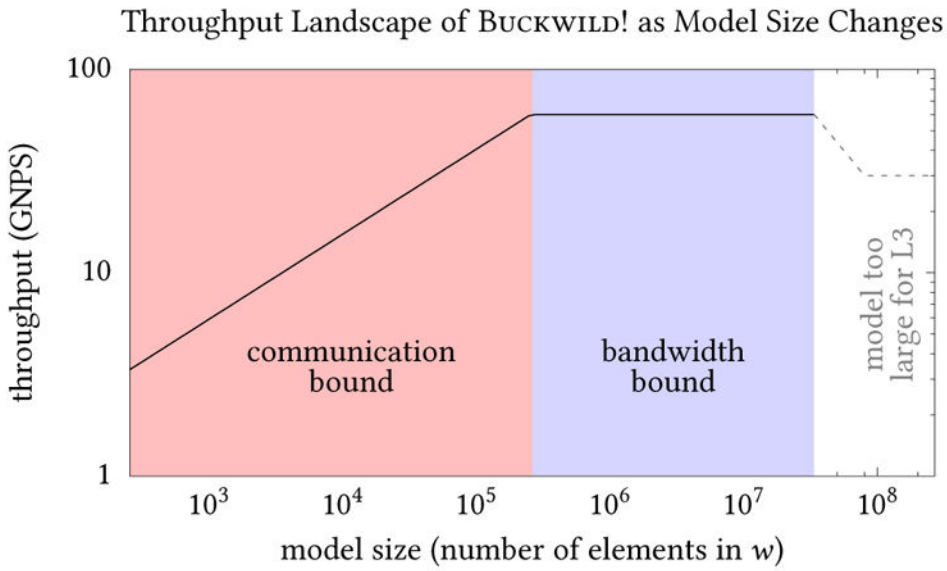


Figure 2.

Bounds for throughput as model size changes. Dashed line represents the setting where the model is too large to fit in the L3 cache.

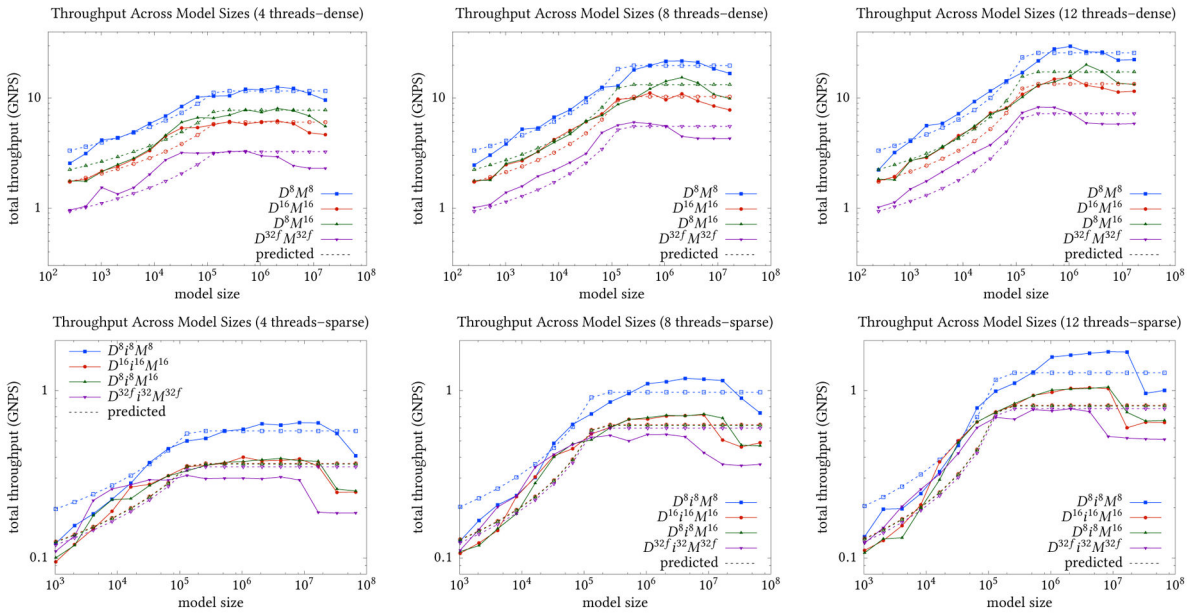
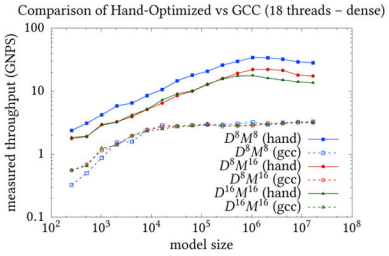
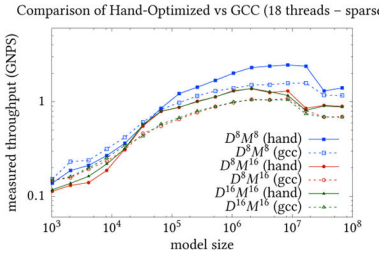


Figure 3.

Comparison of real measured dataset throughput (giga-numbers-per-second) with throughput predicted by the performance model, for multiple threads and precisions, and for sparse and dense data.



(a) Dense problems



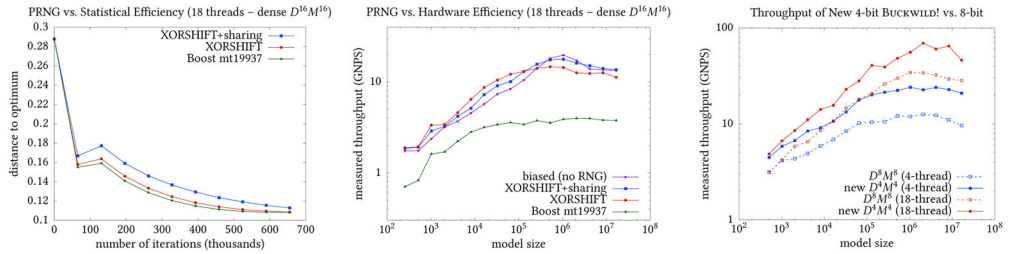
(b) Sparse problems

DMGC Signature	dense	sparse
$D^{16}[i^{16}]M^{16}$	5.43×	1.26×
$D^8[i^8]M^{16}$	6.08×	1.26×
$D^{16}[i^{16}]M^8$	7.97×	1.39×
$D^8[i^8]M^8$	9.72×	1.40×

(c) Average speedups (geometric mean) of hand-optimized AVX2 over GCC -Ofast

Figure 4.

Hand-optimized AVX2 code outperforms GCC-Ofast across multiple precisions by up to 11× (as seen in Figure 4a).



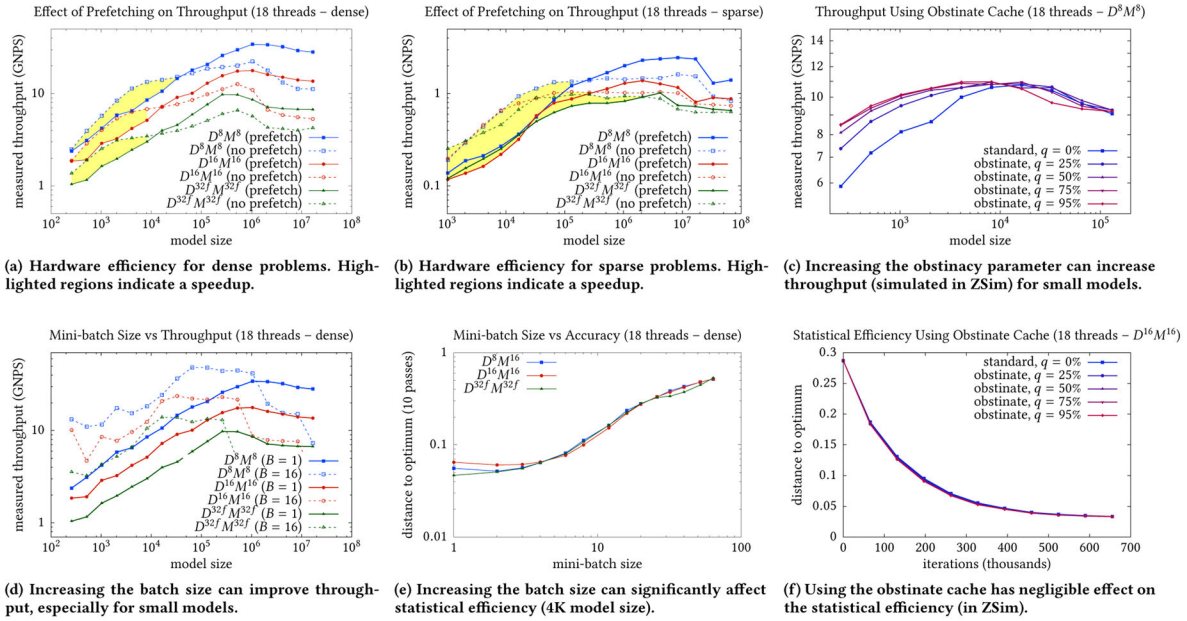
(a) Statistical efficiency comparison of multiple random number generation schemes, each averaged over 5 runs (dense SGD – 16K model).

(b) Hardware efficiency comparison of multiple random number generation schemes (dense SGD).

(c) Adding instructions to enable 4-bit computation improves throughput by up to a factor of 2.

Figure 5.

The effects of random number generation and new 4-bit SGD on hardware and software efficiency.



(a) Hardware efficiency for dense problems. Highlighted regions indicate a speedup.

(b) Hardware efficiency for sparse problems. Highlighted regions indicate a speedup.

(c) Increasing the obstinacy parameter can increase throughput (simulated in ZSim) for small models.

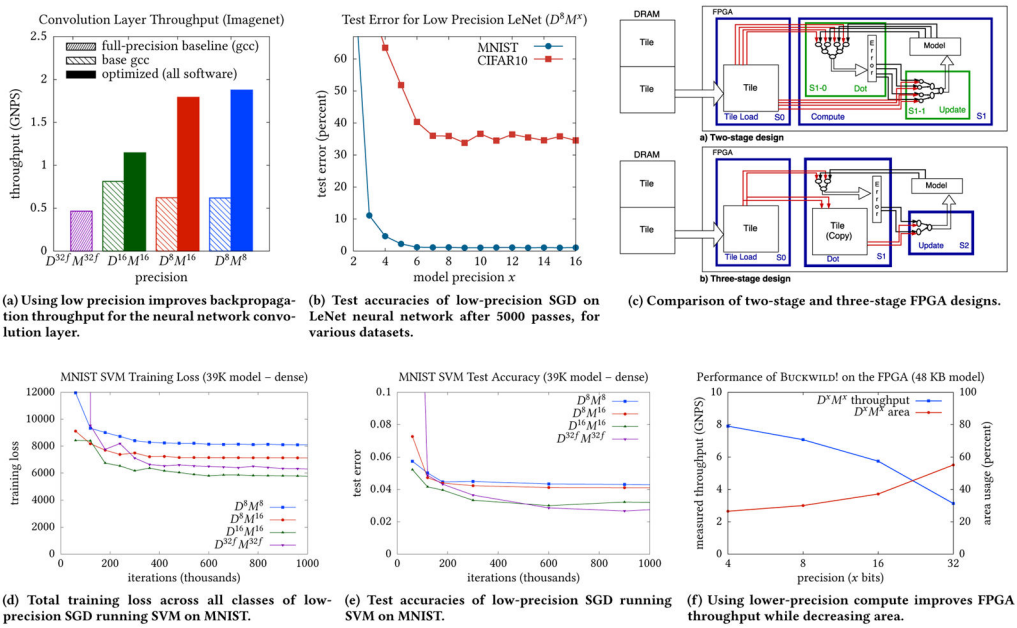
(d) Increasing the batch size can improve throughput, especially for small models.

(e) Increasing the batch size can significantly affect statistical efficiency (4K model size).

(f) Using the obstinate cache has negligible effect on the statistical efficiency (in ZSim).

Figure 6.

Effects of turning off prefetching, changing mini-batch size, using and obstinate cache, and running on an FPGA.



(a) Using low precision improves backpropagation throughput for the neural network convolution layer.

(b) Test accuracies of low-precision SGD on LeNet neural network after 5000 passes, for various datasets.

(c) Comparison of two-stage and three-stage FPGA designs.

(d) Total training loss across all classes of low-precision SGD running SVM on MNIST.

(e) Test accuracies of low-precision SGD running SVM on MNIST.

(f) Using lower-precision compute improves FPGA throughput while decreasing area.

Figure 7.

Effects of running on an FPGA, and validation of approach via alternate applications to convolutional neural network layers (the bottleneck for most systems) and kernels SVM.

Table 1

DMGC signatures of previous algorithms.

Paper	DMGC Signature
Savich and Moussa [45], 18-bit	G^{18}
Seide et al. [46]	$G^1 C_s^1$
Courbariaux et al. [9], 10-bit	G^{10}
Gupta et al. [14]	$D^8 M^{16}$
De Sa et al. [11], 8-bit	$D^8 M^8$

Table 2

Base sequential throughputs used for simplified model, in units of giga-numbers-per-second (GNPS), measured on Xeon E7-8890 (throughputs vary by CPU).

DMGC Signature	dense T_1	sparse T_1
$D^{32f}[i^{32}]M^{8}$	0.203	0.103
$D^{32f}[i^{32}]M^{16}$	0.208	0.080
$D^{32f}[i^{32}]M^{32f}$	0.936	0.101
$D^{8}[i^{8}]M^{32f}$	0.999	0.089
$D^{16}[i^{16}]M^{32f}$	1.183	0.089
$D^{16}[i^{16}]M^{16}$	1.739	0.106
$D^{8}[i^{8}]M^{16}$	2.238	0.105
$D^{16}[i^{16}]M^{8}$	2.526	0.172
$D^{8}[i^{8}]M^{8}$	3.339	0.166

Table 3

Summary of optimizations discussed in this paper.

Optimization	Beneficial when?	Stat. eff. loss
Optimized SIMD	Always	None
Fast PRNG	Using unbiased rounding	Negligible
No prefetching	Communication-bound	Negligible
Mini-batch	Communication-bound	Possible
New instructions	Always	None
Obstinate cache	Communication-bound	Negligible