Fast Quantized Arithmetic on x86: Trading Compute for Data Movement

Alen Stojanov*, Tyler Michael Smith*, Dan Alistarh†, and Markus Püschel*
* Department of Computer Science, ETH Zurich, Switzerland
† IST Austria, Vienna, Austria

***Abstract*—We introduce *Clover*, a new library for efficient computation using low-precision data, providing mathematical routines required by fundamental methods in optimization and sparse recovery. Our library faithfully implements variants of stochastic quantization that guarantee convergence at low precision, and supports data formats from 4-bit quantized to 32-bit IEEE-754 on current Intel processors. In particular, we show that 4-bit can be implemented efficiently using Intel AVX despite the lack of native support for this data format. Experimental results with dot product, matrix-vector multiplication (MVM), gradient descent (GD), and iterative hard thresholding (IHT) demonstrate that the attainable speedups are in many cases close to linear with respect to the reduction of precision due to reduced data movement. Finally, for GD and IHT, we show examples of absolute speedup achieved by 4-bit versus 32-bit, by iterating until a given target error is achieved.**

***Index Terms*—Stochastic quantization, gradient descent, compressive sensing, iterative hard thresholding, SIMD, AVX**

I. Introduction

Recent progress in data processing and machine learning is due not only to the availability of large datasets and the introduction of better algorithms, but also to vastly improved system support for efficient computation. Software frameworks such as Google's TensorFlow [1] leverage multi-core and multi-node parallelism, as well as single-processor pipelining techniques.

Low-precision data representation has emerged as an important design point for speeding up both computation and communication in machine learning applications. For instance, NVIDIA's latest GPU family natively supports computation at 16-bit and 8-bit precisions, while TensorFlow can perform inference over models quantized to as little as 4-bit precision per component [1]. This shift is not entirely surprising: fundamental methods in optimization and sparse recovery are well equipped to dealing with noise in data and measurements; low-precision provides an additional source of *quantization* noise, which such methods may be able to support as well.

Recent theoretical advances [2]–[4] support this intuition: gradient methods can support low-precision data *and* computation and still converge [3]; similar results are known for compressive sensing via specific recovery techniques [4]–[6]. However, these results require *stochastic* quantization (defined in detail later), needing more careful implementation than *deterministic* quantization (e.g., as used in [1]), which is not known to provide convergence guarantees.

Contribution. The fundamental question addressed in this paper is whether applications can take advantage of low precision stochastic quantization on state-of-the-art CPUs. To answer it we provide a library, called *Clover*[1], that supports the basic mathematical routines needed in optimization and sparse recovery with guaranteed convergence. In particular, we provide efficient implementations of the dot product, scale and add, and matrix-vector multiplication in 4-bit quantized, 8-bit quantized, 16-bit half-precision and 32-bit floating point. We also provide an efficient mixed 4,8-bit precision MVM, with the matrix represented in 4-bit quantized and the vectors represented in 8-bit quantized. Our library faithfully implements the variants of stochastic quantization defined in e.g. [3], [4], and therefore enjoys the analytical guarantees provided by these methods.

The main contribution is the efficient implementation of the quantized 4-bit computations on Intel AVX since there is no native support for this data format. (In fact, recent work [7] posited that 4-bit precision could not be supported efficiently on x86 in the absence of native support.)

In a first set of benchmarks we show that our library provides significant speed-ups for the low precision versions in memory bound situations. In particular for MVM we achieve a speedup linear in the precision reduction, e.g., 8x for 4-bit over 32-bit, due to reduced data movement.

In a second set of benchmarks we evaluate our library on two important applications: convex optimization via gradient descent (GD), and reconstruction in compressive sensing via iterated hard thresholding (IHT). First, we show that they inherit the linear (in the precision reduction) speed-up of MVM when run with reduced precision. Then, we show that also an absolute, end-to-end speedup is attainable with precision reduction, i.e., the 4- and 8-bit versions can converge faster in terms of wall clock time to a given target error, since the number of extra iterations needed to convergence is typically smaller than any slowdown due to lowering precision.

II. Background

Quantization. We work with *stochastic quantization* to generate a low-precision version of an arbitrary vector v. In brief, stochastic quantization works as follows. Given the vector v, let $M(v)$ be a scaling factor such that $-1 \leq v/M(v) \leq 1$. Without loss of generality, let $M(v) = \max_i |v_i|$. We partition the interval $[-1, 1]$ using $s+1$ separators: $-1 = l_0 \leq$

[1]Clover is an open source library, publicly accessible via GitHub at https://github.com/astojanov/Clover. Last commit at the time of writing is given with SHA: `80d6cb248737ab786300623a4678b4ab91e78fc2`

$l_1 ... \leq l_s = 1$, which will be mapped to *integer levels* for efficient coding. Each component z in the normalized vector $v/M(v)$, is quantized to one of its two nearest separators: $l_i \leq z \leq l_{i+1}$. More precisely, we map z to l_{i+1} with probability $(z - l_i)/(l_{i+1} - l_i)$, and to l_i with probability $(l_{i+1} - z)/(l_{i+1} - l_i)$.

We denote the *stochastic quantization* function by $Q(v, s)$ and choose the probability of quantizing to different separators such that in expectation values are preserved, i.e., $E[Q(v, s)] = v$. We use $Q(v)$ when s is clear from context.

Compressive sensing (CS) [8]–[10] is a standard technique in sparse signal recovery. It offers a solid mathematical foundation and a range of efficient algorithms for the problem of recovering sparse signals from noisy, high-dimensional, incomplete samples, with a wide range of applications, such as imaging, spectroscopy, and data analysis.

Mathematically, CS assumes a sparse or sparsely-approximable signal $\mathbf{x} \in \mathbb{R}^n$, which is sampled via a linear operator $\mathbf{\Phi}^{m \times n}$ with $m < n$. That is, we are given a vector of measurements $\mathbf{y} \in \mathbb{C}^m$, which can be expressed as

$$\mathbf{y} = \mathbf{\Phi x} + \mathbf{e}, \tag{1}$$

where $\mathbf{e}$ is the observation noise. Under certain assumptions on the matrix $\mathbf{\Phi}$, as in [9], the sparsity constraint imposed on $\mathbf{x}$ can overcome this ill-posed problem, and allows approximate recovery of the original signal.

Iterative hard thresholding. At a high level, recovery algorithms for CS work by iteratively building up a sparse estimate $\mathbf{x}$ such that $\mathbf{\Phi x}$ approximates $\mathbf{y}$, i.e., $\|\mathbf{y} - \mathbf{\Phi x}\|_2$ becomes small. There has been a tremendous amount of work on developing efficient recovery algorithms; for a survey, we refer the reader to [11]. In this paper, we focus on a popular recovery algorithm called *iterative hard thresholding (IHT)* [12], [13].

Given the current iterate $\mathbf{x}^t$, where $\mathbf{x}^0$ is initialized to 0, the IHT update rule is

$$\mathbf{x}^{t+1} = H_s(\mathbf{x}^t + \mu \mathbf{\Phi}^\top (\mathbf{y} - \mathbf{\Phi x}^t)), \tag{2}$$

where $H_s(\mathbf{x}) : \mathbb{C}^n \to \mathbb{C}^n$ is the nonlinear thresholding operator that preserves the largest s (in magnitude) entries of $\mathbf{x}$ and sets the rest to zero, and $\mu > 0$ is a parameter we will call the *step size*. The convergence properties of this method and its variants are well studied [14], and the method can provide state-of-the-art recovery in some settings.

Gradient descent (GD) is a fundamental method for convex optimization problems of the form

$$\underset{\mathbf{x} \in \mathbb{R}^n}{\text{minimize}} \quad f(\mathbf{x}). \tag{3}$$

GD solves problems of the form (3) by forming a sequence of estimators $(\mathbf{x}^k)_{k \in \mathbb{N}}$ such that

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \mu \nabla f(\mathbf{x}^k), \tag{4}$$

given the initial point $\mathbf{x}^0$ and fixed positive step size μ, where $\nabla f(\mathbf{x}^k)$ is the *gradient* of the function f at point $\mathbf{x}^k$. Theoretical results [15] guarantee that GD reaches ϵ accuracy after at most $\mathcal{O}(1/\epsilon)$ iterations if f is convex and differentiable and has a Lipschitz-continuous gradient.

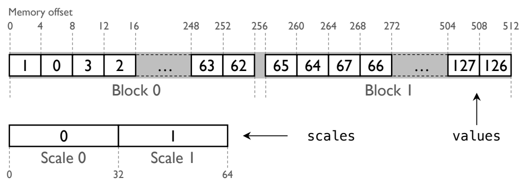


Fig. 1: Illustration of a `CloverVector4` container of 128 4-bit elements.

In this paper, we consider GD for solving least squares problems of the form (3) with $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{Ax} - \mathbf{b}\|^2$, where $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$.

III. Low Precision Arithmetic

In this section we describe the implementation of the low-precision arithmetic library *Clover*. Our library implements the linear algebra operations necessary for GD and IHT in 4-bit quantized, 8-bit quantized, 16-bit half precision, and 32-bit floating point arithmetic. Furthermore, it implements MVM using mixed 4,8-bit quantized arithmetic, where the matrix is represented in 4-bit quantized and the vectors are implemented in 8-bit quantized. We describe the implementation of the 4-bit quantized arithmetic in depth and then briefly describe the differences for the other datatypes.

A. 4-bit Arithmetic

Functionality for quantized 4-bit linear algebra operations is provided by the `CloverVector4` and `CloverMatrix4` containers. `CloverVector4` contains a vector represented in a quantized 4-bit format. The values and the scaling factors are stored in separate contiguous blocks of memory, such that sub-vectors of length 64 share the same scale.

An illustration of the memory layout of a `CloverVector4` container is shown in Fig. 1. Each value in the vector is stored as a two's complement in a 4-bit *nibble*. The x86 architecture uses byte addressing, so addressing the nibbles must be done in software. We accomplish this explicitly in the application logic, storing two consecutive nibbles in a byte. As a result, the values in a `CloverVector4` of length n are represented as an array of $n/2$ bytes.

`CloverMatrix4` contains a matrix represented similarly. The values and scales are stored as separate contiguous blocks of memory, both in row-major order. Sub-matrices of size 64×64 share the same scale.

We now describe the implementation of the operations on the `CloverVector4` and `CloverMatrix4` containers, providing more detail for the dot-product implementation as it is the most important for our applications.

Packing and unpacking of 4-bit values. The x86 architecture does not support 4-bit integer arithmetic. Thus if we have an array of 4-bit values, we must *unpack*, representing them as larger (i.e., 8, 16 or 32-bit) integers for computation.

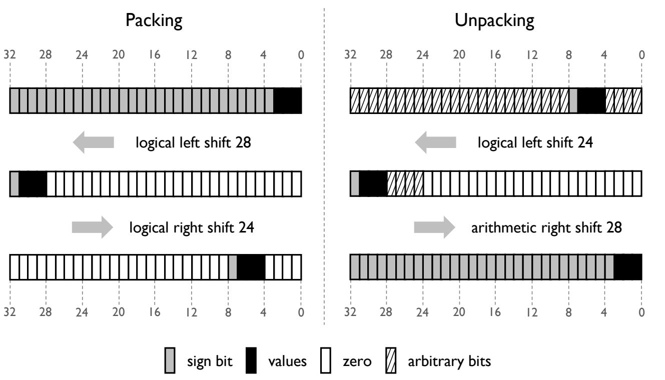


Fig. 2: Illustration of placing a two's complement nibble in the low ordered four bits of a 32-bit integer.

Then, for efficient storage and data movement, we *pack* these larger integers, representing them as 4-bit values stored within an array. We now describe how to efficiently pack and unpack nibbles to and from a 32-bit integer, using the example in Fig. 2. Nibbles can be packed and unpacked into integers of other sizes similarly.

The goal of unpacking from a 32-bit integer (right part of Fig. 2) is as follows: We start with a 32-bit entity that stores eight nibbles. We wish to extract a single 4-bit nibble and represent it as a 32-bit integer. This can be done with two bit shifts: (a) a logical left shift is used to shift the nibble so that it occupies the highest-order 4-bits of the 32-bit entity. (b) an arithmetic right shift is used to shift the nibble to the lowest order 4-bits of the 32-bit entity. The arithmetic right shift has sign extension, filling the high-order 28 bits with the sign bit of the nibble. yielding a 32-bit integer with the same value as the two's complement 4-bit value.

The goal of packing (left part of Fig. 2) is to revert the unpacking operation. Two bit shifts are used to place the lowest order 4 bits of a 32-bit integer anywhere within a 32-bit entity. (a) a logical left shift is used to shift the nibble so that it occupies the highest-order 4-bits of the 32-bit entity. (b) a logical right shift is used to shift the nibble to the desired location within the 32-bit entity. The first sets the bits lower-ordered than the nibble to zero, and the second sets the bits higher-ordered than the nibble to zero. A bitwise `OR` operation is then used to store up to eight nibbles in the 32-bit entity.

Quantization. Quantization converts a vector or matrix of 32-bit floating point values into a `CloverVector4` or `CloverMatrix4`. To achieve fast conversion and stochastic rounding we use the `AVX2` implementation of the XORShift [16] pseudo random number generator (PRNG). For space reasons, we do not describe it in detail here.

Dot product. The dot product performs the operation $\gamma := \mathbf{a}^T\mathbf{b}$, where $\mathbf{a}$ and $\mathbf{b}$ are column vectors. We implement this operation by unpacking nibbles into 8-bit integers first. Then, we use the `vpmaddubsw` instruction to multiply each unsigned 8-bit integer from the first operand pairwise with the corresponding signed 8-bit integer from the second operand, producing intermediate signed 16-bit integers, and storing the sum of adjacent intermediate 16-bit integers in the destination. Finally, we sum the intermediate 16-bit integers, storing the results as 32-bit integers, eventually resulting in a partial sum that is represented as a 32-bit integer, which is then converted to a 32-bit float. The procedure is described as follows:

1) Create a single `__m256` of 8 32-bit floats that accumulates the result and initialize it to zero. Then start traversing the vectors by blocks of 64 elements.
2) For each block, load 64 values of $\mathbf{a}^T$ into one `__m256i` called `A` and 64 values of $\mathbf{b}$ into a `__m256i` called `B`.
3) Unpack `A` into two `__m256i` variables `A1` & `A2`, and `B` into `B1` & `B2`, each storing 32 8-bit integers.
4) Extract the signs from the elements in `A1` and `A2` and use `vpsignb` to transfer them to `B1` and `B2`. Now, `A1` and `A2` store unsigned 8-bit integers, and `B1` and `B2` store signed 8-bit integers.
5) Use `vpmaddubsw` to multiply `A1` and `B1` and to multiply `A2` and `B2`, giving us two `__m256i` variables, each storing 8 signed 16-bit integers.
6) Over several instructions, successively sum the results until we have a single `__m256i` of 32-bit integers.
7) Convert the 32-bit integers into floats and multiply them with the scales, accumulating it in the result variable.
8) Once done iterating over all blocks, perform horizontal sum reduction on the result variable and return.

This dot product routine returns a single 32-bit floating point value, which may be later re-quantized. This re-quantization has some cost, but it is amortized by the rest of the dot product.

Mixed-precision dot product. In the 4-bit dot product, the values in each vector are unpacked to 8-bit before they are used for computation. We use this property to implement a mixed-precision dot product routine where one operand is an 8-bit quantized vector and the other is a 4-bit quantized vector. This is accomplished by unpacking the 4-bit operand into 8-bit integers and loading the 8-bit operand without unpacking. As explained next, the mixed-precision routine offers an advantage in MVM as explained next.

Matrix-vector multiplication (MVM). We implement MVM by traversing the matrix row by row, computing the dot product of each row with the corresponding vector. Once 64-rows are completed, we re-quantize the 64 values into a single 256-bit `AVX` variable, storing it in the resulting vector. This avoids the performance penalty from re-quantization associated with the scale and add routine.

We also implemented a mixed-precision MVM by casting its computation in terms of the mixed-precision dot product routine. In this mixed-precision MVM, the rows of the matrix are kept at 4-bit since they constitute the bulk of data movement. The vector is in 8-bit, since it is only loaded once (and reused) and thus there is little extra data movement but a gain in precision. The overall effect is a mixed-precision 4,8-bit MVM that is nearly as fast as the pure 4-bit MVM.

Scale-and-add. The scale-and-add routine multiplies a `CloverVector4` by a scalar, and adds the result to another `CloverVector4`. For space reasons, we do not describe it in detail. The scale-and-add routine is not as efficient as the dot product routine. This is because for dot product, there is only a single value to re-quantize, and for scale and add, the re-quantization must be performed for every element of the output array.

Parallelization. We use OpenMP for multithreaded execution. The scale-and-add routine is embarrassingly parallel: one loop with independent iterations. For MVM, the outer loop that defines the blocked traversal along the rows of the matrix is parallelized; thus we do not need a parallel dot product.

B. Other datatypes

We now discuss our implementation of the other datatypes.

8-bit arithmetic The implementation of the 8-bit version follows the same design decisions of the 4-bit version, i.e. blocks of 64 elements with a corresponding scale in the vector case, and tile of 64×64 elements with a corresponding scale in the matrix case. While the implementation is almost identical, one important difference is that the 8-bit dot product does not use steps 2) and 3), as the quantized values are already represented as 8-bit integers, and instead loads 64 values into two `__m256i` variables, each containing 32 8-bit integers.

16-bit arithmetic The implementation of the 16-bit version uses the native support for half-precision floats, defined in IEEE 754-2008 as the 16-bit base-2 format. Both the vector container and the matrix container have no notion of blocks; instead they are continuous arrays of type `uint16_t` (C intrinsics functions use `uint16_t` instead of a dedicated half-precision type). We use the `vcvtps2ph` instruction to perform quantiation. When performing scale and add, dot product, and MVM, we use `vcvtph2ps` to convert the packed 16-bit half precision floats into 32-bit single precision floats and perform the computation. Depending on the operation, we usually traverse the arrays in blocks of 16, 32, or 64 elements, such that we load, convert to 32-bit and compute while keeping most computation in registers and maintaining spatial and temporal locality. To benefit from ILP, we unroll the loops and use several accumulators to maximize performance. Our MVM implementation is based on our 16-bit dot product routine, and we use OpenMP to parallelize the outer loops.

32-bit arithmetic The 32-bit version uses single-precision floating point arrays. We use floating-point for this precision because computation with 32-bit floats is much faster than with 32-bit integers in `AVX`.Our implementation of the dot product as well as the scale and add routine is almost identical with the 16-bit version, except that we do not perform conversions. For MVM we use the (parallel) Intel Math Kernel Library (MKL) and use OpenMP to parallelize the other routines.

IV. Evaluation

We evaluated our implementation on an Intel Xeon CPU E3-1285L v3 3.10GHz Haswell with 32GB of RAM and 25.6 GB/s bandwidth to main memory, running Debian GNU/Linux 8 (jessie), kernel 3.16.43-2+deb8u3. We use the Intel icc compiler 17.0.0, Intel IPP 2017.0.0 (r52494), and Intel MKL 2017.0.0 (Build 20160801). We use `RDTSC` to measure the cycle count for each test, performing 15 repetitions with warm cache, and reporting the median result. To avoid the effects of frequency scaling and resource sharing on the measurements, Turbo Boost and Hyper-Threading are disabled.

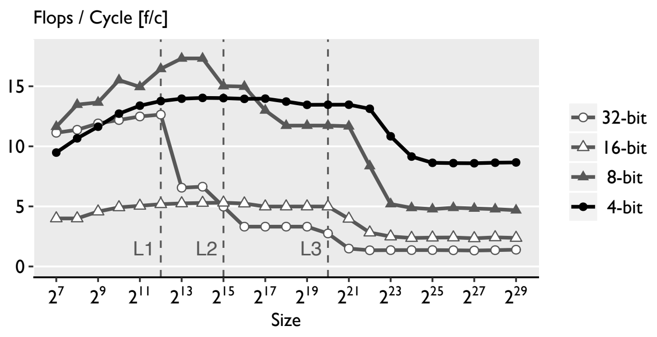


Fig. 3: Performance of single threaded dot product.

A. Performance of Individual Routines

First, we evaluate the performance of the individual routines. For each routine we derive a pseudo flop count, using the number of additions and multiplications required to perform the mathematical operations, and report our results in flops per cycle (f/c). The flop count for dot product and scale-and-add is $2n$ and for MVM it is $2n^2$.

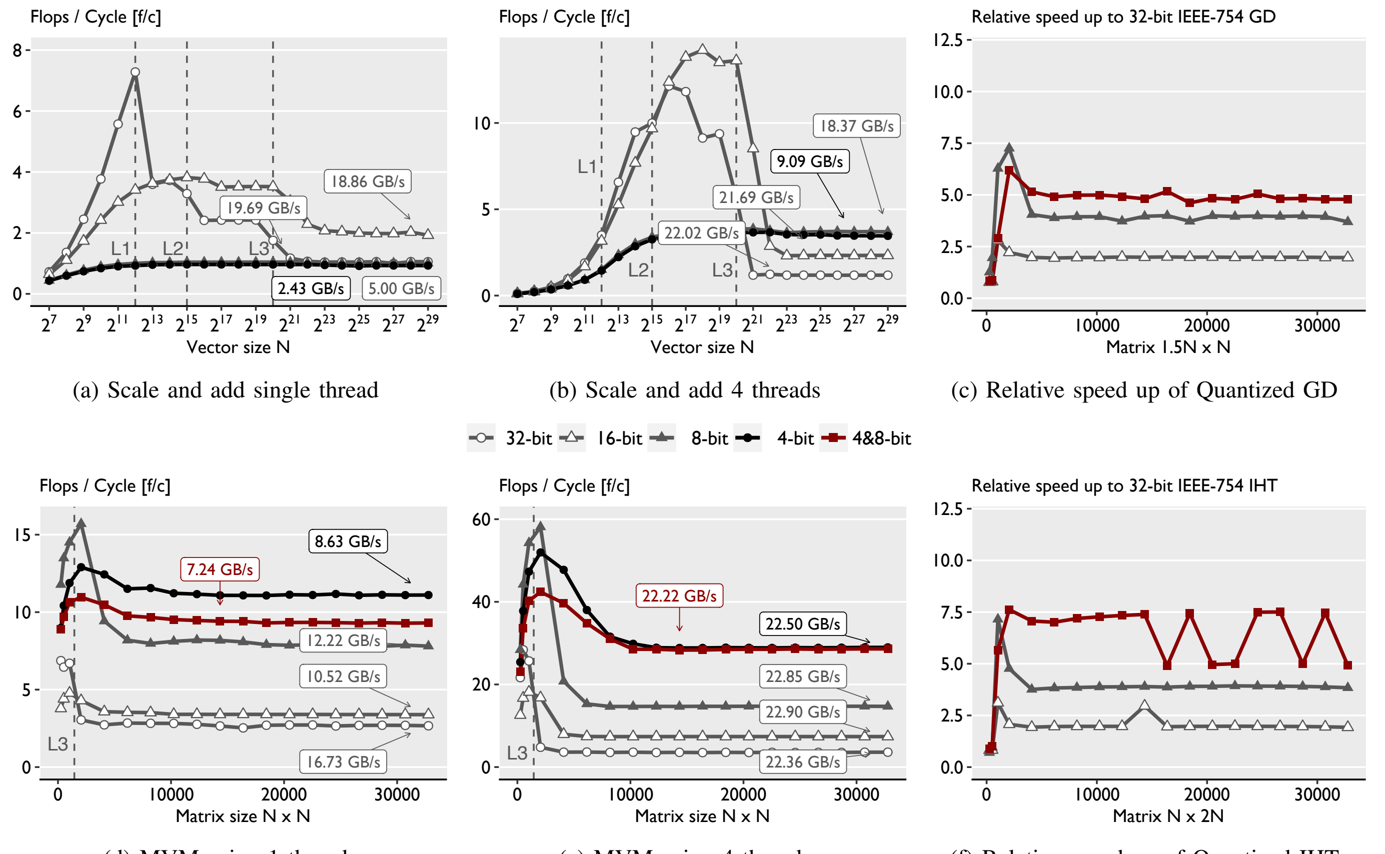
Dot product. Fig. 3 shows the performance profile of the dot product. When the data fits in the L2 cache the 32-bit version is much faster than 4-bit because the entire computation is done using native instructions without unpacking. Once the data exceeds the size of the L3 cache, 4-bit is fastest since the dot product is memory bound: data movement from RAM becomes the bottleneck. The speed-up is up to 6x over the 32-bit version. Since the dot product is only used within (a parallel) MVM, we do not need a parallel version.

Scale-and-add. The results in Fig. 4a show that the 32-bit and 16-bit implementations are faster than 4-bit and 8-bit within cache for the same reasons. However, even outside L3 cache, 4- and 8-bit are no faster than 32-bit due to the overhead of re-quantization. This is reflected in the low bandwidth used. As a result, parallelization yields near-linear speed-up for 4-bit and near none for 32-bit, so parallel 4-bit is about 3x faster overall (Fig. 4b).

MVM. In Figs. 4d and 4e, we compare sequential and parallel implementations of MVM for each datatype, including the mixed 4,8-bit MVM explained in Section III-A. For the sequential case, for problems that do not fit in cache, we see that pure 4-bit is about 4.6x faster than 32-bit but uses only one third of the available bandwidth, and the mixed 4-bit and 8-bit MVM is noticeable slower. However, once parallelized, all version exhaust the available bandwidth and thus reach a speedup linear in the precision reduction. The mixed 4,8-bit MVM is now as fast as the 4-bit version since the bottleneck is loading the matrix.

B. Evaluation of Quantized IHT and Quantized GD

We implemented the IHT and GD algorithms with the routines described in Section III and our basic library functions.



(a) Scale and add single thread (b) Scale and add 4 threads (c) Relative speed up of Quantized GD

(d) MVM using 1 thread (e) MVM using 4 threads (f) Relative speed up of Quantized IHT

Fig. 4: Performance evaluation of quantized linear algebra routines and their application in IHT and GD algorithms.

We show relative performance, comparing the 8-bit, 16-bit, and mixed 4,8-bit versions to the 32-bit implementation of these algorithms. The mixed 4,8-bit version uses 4-bit matrices and 8-bit vectors, using a mixed 4-bit and 8-bit MVM routine and pure 8-bit vector-vector routines.

Experimental setup. In the case of IHT, we set up $\mathbf{\Phi}$ to have twice as many columns as rows ($n = 2m$), with a sparsity of $m/4$. For GD, we set up $\mathbf{A}$ to have 1.5x as many columns as rows ($n = 1.5m$). We define the *recovery error* as the norm of the difference between the recovered x and the original x, divided by the norm of the original x. For both algorithms IHT and GD, we use grid search for the hyperparameter optimization of the μ parameter to decrease recovery error and minimize the number of iterations. We compute $\mathbf{\Phi}^T$ needed in (2) offline and feed it as an argument along with $\mathbf{\Phi}$ to both the IHT and GD routines. We use the parallel version of all operations within the IHT and GD algorithms.

Per-iteration speedup. We observe that the per-iteration speedup of both IHT and GD scales almost perfectly with the reduction in precision, e.g., 4x speedup for 8-bit precision. This speedup is inherited from MVM, which constitutes the bulk of the computation. The precision of the result obtained after a given iteration will of course vary between the data formats. For this reason we investigate absolute speedup next.

Absolute speedup. To compare the end-to-end performance of the different implementations of IHT and GD, we record the time it takes each implementation to reach a given target recovery error. To obtain a target error, we choose for each problem size the best recovery error achievable by the mixed 4,8-bit precision implementation. We then determine the number of iterations that each datatype needs to achieve the target recovery error and record the time needed for each implementation to run for that number of iterations. Results are shown for GD in Fig. 4c and for IHT in Fig. 4f. The first important observation concerns the absolute gain in runtime for the selected target error. For GD, 4-bit is about 5x faster as 32-bit. Since the per-iteration speed-up is about 8x, this means it needs about 3/5 more iterations, typically around 24 versus 15 for 32-bit. Interestingly, 8-bit can offer more than 4x speedup in some cases, meaning that it actually requires fewer iterations than 32-bit. This can happen as stochastic quantization adds noise, which can speed up convergence in some cases, e.g. [3], [17]. Also, the runtime is very sensitive to the value of μ.

For IHT, the runtime behaviour is more erratic, likely due to the dependency on μ and due to the small number of iterations (typically around 3–5). For 4-bit we also observe that usually more iterations are needed but also, in rare cases, fewer, achieving speedups of up to 7.6x. Overall, 4-bit offers an absolute speedup in many cases. The delicate relationship between problem specification, data precision, hyperparameter, and target error invites further investigation.

V. Related Work

There has recently been considerable interest in low-precision computation and communication in the context of machine learning, e.g. [17]–[19]. Due to space constraints, we will focus on work that is immediately related to ours. One of the first to consider low-precision computation with convergence guarantees in the context of (stochastic) gradient descent has been Buckwild! [2]. They provide convergence analysis for the restricted case where only the *gradients* are quantized, and provide CPU implementation results via intrinsics for four and eight bits in the more general case where the gradients are quantized. We note that their implementation departs from the theory, and therefore is not analytically guaranteed to converge. The ZipML project [3] presents an algorithmic framework for end-to-end low-precision stochastic gradient descent (quantized gradients, model, and data), complete with convergence analysis in the convex case including FPGA implementation results for various precisions [20]. We implement their algorithmic framework here in the context of gradient descent. To our knowledge, this is the first efficient such implementation for CPUs. Our framework removes their technical requirement that the data has to be quantized before the execution of the algorithm, as we also implement fast quantization and dequantization. Recent work [7] performed an in-depth study of low-precision and asynchronous SGD computation on CPUs and FPGAs, with an eye towards architectural implications. Due to the lack of native instruction support for 4-bit (or lower) operations in, e.g., AVX2, they focused on 8-bit and higher precisions in their implementation, and obtained 4-bit results by emulating the existence of native instructions in a simulator. Here, we overcome this limitation by replacing native 4-bit instructions through careful use of existing AVX2 intrinsics. This allows us to obtain results for 4-bit instructions on existing CPUs, while at the same time preserving computational efficiency.

Low-precision recovery algorithms for compressive sensing, and their FPGA implementations, have been considered by several references, e.g. [4]–[6]. Our CPU implementation of low-precision IHT is adapted from this last reference. To our knowledge, ours is the first low-precision CPU implementation of this technique.

VI. Conclusion

The main contribution of the paper is to show that reduced precision stochastic quantized arithmetic can be implemented efficiently on modern CPUs. In particular, this holds for 4-bit on Intel AVX even though there is no native support for operations with this data format. This is possible since the extra operations needed are overcompensated by the reduced data movement in the memory-bound situations that are typical in many applications. In these cases a speed-up linear in the precision reduction is possible. As examples where this is the case, we considered gradient descent and iterative hard thresholding, for which we exhibited absolute speed-ups (time to target error) with reduced precision.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: A system for large-scale machine learning,” in *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 16, 2016, pp. 265–283.

[2] C. D. Sa, C. Zhang, K. Olukotun, and C. Ré, “Taming the wild: A unified analysis of hogwild-style algorithms,” in *Advances in Neural Information Processing Systems (NIPS)*, vol. 28, 2015, pp. 2674–2682.

[3] H. Zhang, J. Li, K. Kara, D. Alistarh, J. Liu, and C. Zhang, “ZipML: Training linear models with end-to-end low precision, and a little bit of deep learning,” in *Proc. International Conference on Machine Learning (ICML)*, 2017, pp. 4035–4043.

[4] N. M. Gürel, K. Kara, A. Stojanov, T. Smith, D. Alistarh, M. Püschel, and C. Zhang, “Compressive Sensing with Low Precision Data Representation: Radio Astronomy and Beyond,” *CoRR*, vol. abs/1802.04907v2, 2018.

[5] Y. Plan and R. Vershynin, “One-bit compressed sensing by linear programming,” *Comm. on Pure and Applied Mathematics*, vol. 66, pp. 1275–1297, 2013.

[6] S. Gopi, P. Netrapalli, P. Jain, and A. Nori, “One-bit compressed sensing: Provable support and vector recover,” in *Proc. International Conference on Machine Learning (ICML)*, 2013.

[7] C. D. Sa, M. Feldman, C. Ré, and K. Olukotun, “Understanding and optimizing asynchronous low-precision stochastic gradient descent,” in *Proc. International Symposium on Computer Architecture (ISCA)*, 2017, pp. 561–574.

[8] D. L. Donoho, “Compressed sensing,” *IEEE Trans. Information Theory*, vol. 52, no. 4, pp. 1289–1306, 2006.

[9] E. J. Candes, J. Romberg, and T. Tao, “Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information,” *IEEE Trans. Information Theory*, vol. 52, no. 2, pp. 489–509, 2006.

[10] E. Candes, J. Romberg, and T. Tao, “Stable signal recovery from incomplete and inaccurate measurements,” *Communications on pure and applied mathematics*, vol. 59, no. 8, pp. 1207–1223, 2006.

[11] Y. C. Eldar and G. Kutyniok, *Compressed sensing: theory and applications*. Cambridge University Press, 2012.

[12] T. Blumensath and M. Davies, “Iterative thresholding for sparse approximations,” *The Journal of Fourier Analysis and Applications*, vol. 14, no. 5-6, pp. 629–654, 2008.

[13] T. Blumensath and M. E. Davies, “Iterative thresholding for compressed sensing,” *Applied and Computational Harmonic Analysis*, vol. 27, no. 3, pp. 265–274, 2009.

[14] T. Blumensath, M. Davies, and G. Rilling, “Greedy algorithms for compressed sensing,” in *Compressed Sensing: Theory and Applications*. Cambridge University Press., 2012, pp. 348–393.

[15] Y. Nesterov, *Introductory lectures on convex optimization: A basic course*. Springer Science & Business Media, 2013, vol. 87.

[16] G. Marsaglia, “Xorshift RNGs,” *Journal of Statistical Software, Articles*, vol. 8, no. 14, pp. 1–6, 2003. [Online]. Available: https://www.jstatsoft.org/v008/i14

[17] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, “QSGD: communication-efficient SGD via gradient quantization and encoding,” in *Advances in Neural Information Processing Systems (NIPS)*, vol. 30, 2017, pp. 1707–1718.

[18] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “Xnor-net: Imagenet classification using binary convolutional neural networks,” in *Proc. European Conference on Computer Vision (ECCV)*, 2016, pp. 525–542.

[19] M. Courbariaux, Y. Bengio, and J.-P. David, “Binaryconnect: Training deep neural networks with binary weights during propagations,” in *Advances in Neural Information Processing Systems*, vol. 28, 2015, pp. 3123–3131.

[20] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang, “FPGA-accelerated dense linear machine learning: A precision-convergence trade-off,” in *Proc. International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 160–167.