Accelerating Neural Network Training using Arbitrary Precision Approximating Matrix Multiplication Algorithms

Grey Ballard
Wake Forest University
Winston-Salem, NC, USA
ballard@wfu.edu

Jack Weissenberger
Wake Forest University
Winston-Salem, NC, USA
jack.weissenberger@gmail.com

Luoping Zhang
Wake Forest University
Winston-Salem, NC, USA
zhanl317@wfu.edu

ABSTRACT

Matrix multiplication is one of the bottleneck computations for training the weights within deep neural networks. To speed up the training phase, we propose to use faster algorithms for matrix multiplication known as Arbitrary Precision Approximating (APA) algorithms. APA algorithms perform asymptotically fewer arithmetic operations than the classical algorithm, but they compute an approximate result with an error that can be made arbitrarily small in exact arithmetic. Practical APA algorithms provide significant reduction in computation time and still provide enough accuracy for many applications like neural network training. We demonstrate that APA algorithms can be efficiently implemented and parallelized for multicore CPUs to obtain up to 28% and 21% speedups over the fastest implementation of the classical algorithm using one core and 12 cores, respectively. Furthermore, using these algorithms to train a Multi-Layer Perceptron (MLP) network yields no significant reduction in the training or testing error. Our performance results on a large MLP network show overall sequential and multithreaded performance improvements of up to 25% and 13%, respectively. We also demonstrate up to 15% improvement when training the fully connected layers of the VGG-19 image classification network.

CCS CONCEPTS

• **Mathematics of computing → Mathematical software performance**; • **Computing methodologies → Neural networks**.

KEYWORDS

neural networks, multilayer perceptrons, APA matrix multiplication

ACM Reference Format:
Grey Ballard, Jack Weissenberger, and Luoping Zhang. 2021. Accelerating Neural Network Training using Arbitrary Precision Approximating Matrix Multiplication Algorithms. In *50th International Conference on Parallel Processing Workshop (ICPP Workshops '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458744.3474050

1 INTRODUCTION

Fast matrix multiplication algorithms are those that perform fewer than the $2n^3 + O(n^2)$ floating point operations (flops) performed by the classical algorithm. For example, Strassen's original fast matrix multiplication algorithm performs $O(n^{2.81})$ flops [31]. The true complexity of matrix multiplication, typically measured as the exponent ω for complexity $O(n^{\omega})$ is an open question, but the current tightest upper bound is 2.37286 [2]. Upper bounds in this range correspond to theoretical algorithms that are not expected to be practical.

One of the reasons such algorithms are impractical is that they are based on so-called *Arbitrary Precision Approximating* (APA) algorithms, which we describe in detail in § 2. In exact arithmetic, these algorithms compute an approximation of the correct result, where the error is polynomial in a nonzero parameter of the algorithm [7]. That is, in exact arithmetic, the error can be made arbitrarily small. In floating point arithmetic, however, there is a lower bound on the approximation error that depends on the working precision and properties of the algorithm. Thus, APA algorithms are often considered to have insufficient accuracy for most applications and have largely been overlooked as practical tools despite their performance potential and ability to outperform exact algorithms [4].

Our goal in this paper is to demonstrate that APA algorithms can offer practical performance improvements for applications that are tolerant to error in matrix multiplications, notably the training phase of neural networks. Training large neural networks can be incredibly computationally expensive. For example, OpenAI's most recent language model, GPT-3, spent the equivalent of thousands of days training on a petaflop machine [8]. Even a slight reduction in flops could save a significant amount of time and money. These steep costs have spurred a surge of research into more efficient hardware, better algorithms, and innovative techniques for trading off accuracy for performance. For instance, low-precision arithmetic has been shown to decrease running time with little to no effect on the ultimate learning task [12, 16], and new floating point formats have been developed and supported in hardware to implement highly efficient low- and mixed-precision computation [17, 34].

Matrix multiplication in particular is a bottleneck computation for many neural networks. Forward and backward propagation in training the weights of fully connected layers requires matrix multiplication with dimensions given by the sizes of the layers and number of batch samples. Training convolutional and other types of layers can also be cast as matrix multiplication, either via monolithic multiplications or batches of smaller multiplications [9, 11]. In this paper we focus on Multi-Layer Perceptron (MLP) networks that rely on a sequence of fully connected layers [13]. Because the sizes of the layers in MLP networks continue to grow, various techniques have been used to reduce the computational demands of the training phase. For example, low-rank tensor approximation of the weights can reduce both memory and computation [21], and fast matrix

ICPP Workshops '21, August 9–12, 2021, Lemont, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8441-4/21/08…$15.00
https://doi.org/10.1145/3458744.3474050

multiplication (the Strassen-Winograd algorithm) has been applied to the bottleneck matrix multiplications [18].

Our contribution is the use of APA matrix multiplication algorithms to address this problem. In particular we

(1) curate a collection of both well-established and recently discovered practical APA algorithms;
(2) extend the framework of [4] to generate efficient multithreaded code for all of them, achieving up to 21% performance improvement over the best parallel classical implementation;
(3) demonstrate the robustness of learning accuracy to approximate matrix multiplications;
(4) present multithreaded performance improvements of a synthetic MLP of up to 13% over the classical algorithm; and
(5) show performance improvements for training the fully connected layers of the VGG-19 network of up to 15%.

2 PRACTICAL ARBITRARY PRECISION APPROXIMATING MATRIX MULTIPLICATION ALGORITHMS

2.1 Fast Matrix Multiplication

Nearly all fast matrix multiplication algorithms are based on a rule for multiplying matrices of fixed size, and the reduction in asymptotic complexity stems from using the rule recursively on general matrices. For example, Strassen's algorithm is specified by a rule for multiplying two 2×2 matrices (denoted $\langle 2,2,2\rangle$) using 7 multiplications instead of the classical algorithm's 8 multiplications. Applying the rule to $n \times n$ matrices, we split each matrix into quadrants, and the 7 multiplications are multiplications of $(n/2) \times (n/2)$ matrices. For even better efficiency, we can consider larger fixed sizes and find rules that require a lower percentage of multiplications compared to the classical rule. The number of multiplications in a rule is known as the *rank*, so Strassen's is a rank-7 algorithm. Algorithms have been derived both analytically and computationally, and there exists a vast set of improvements leading to the current world record [2, 10, 22, 33].

2.2 APA Algorithms

A key characteristic of matrix multiplication, first demonstrated by Bini et al. [6], is that it can be approximated to arbitrary accuracy with less computation than required by exact algorithms. An APA algorithm is one that takes as input A, B, and a scalar parameter $0 < \lambda < 1$ and computes

$$\hat{C} = A \cdot B + \lambda E + O(\lambda^2), \qquad (1)$$

where E is an error matrix that depends on A and B and corresponds to the leading term of the error polynomial. Thus, in exact arithmetic, letting $\lambda \longrightarrow 0$ achieves arbitrarily small approximation error. In floating point arithmetic, choosing too small a value for λ leads to accumulation of roundoff error that exceeds the approximation error. We discuss optimizing λ in § 2.3.

In theory, APA algorithms can be converted to exact algorithms at the cost of an extra logarithmic factor of n, which is typically hidden by an arbitrarily small increase in the exponent [5]. In practice, n is not large enough to ignore the logarithmic factor, so we consider each APA algorithm as is.

For a concrete example, we reproduce the rule for the algorithm developed by Bini et al. [6] for the $\langle 3,2,2\rangle$ case (multiplying a 3×2 matrix A by a 2×2 matrix B), where we use the following notation for input and output matrices:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{31} & C_{32} \end{bmatrix}.$$

Bini's rule is given by

$$\begin{aligned}
M_1 &= (A_{11} + A_{22}) \cdot (\lambda B_{11} + B_{22}) \\
M_2 &= A_{22} \cdot (-B_{21} - B_{22}) \\
M_3 &= A_{11} \cdot B_{22} \\
M_4 &= (\lambda A_{12} + A_{22}) \cdot (-\lambda B_{11} + B_{21}) \\
M_5 &= (A_{11} + \lambda A_{12}) \cdot (\lambda B_{12} + B_{22}) \\
M_6 &= (A_{21} + A_{32}) \cdot (B_{11} + \lambda B_{22}) \\
M_7 &= A_{21} \cdot (-B_{11} - B_{12}) \\
M_8 &= A_{32} \cdot B_{11} \\
M_9 &= (A_{21} + \lambda A_{31}) \cdot (B_{12} - \lambda B_{22}) \\
M_{10} &= (\lambda A_{31} + A_{32}) \cdot (B_{12} - \lambda B_{22})
\end{aligned}$$

$$\begin{aligned}
\hat{C}_{11} &= \lambda^{-1}(M_1 + M_2 - M_3 + M_4) \\
\hat{C}_{12} &= \lambda^{-1}(-M_3 + M_5) \\
\hat{C}_{21} &= M_4 + M_6 - M_{10} \\
\hat{C}_{22} &= M_1 - M_5 + M_9 \\
\hat{C}_{31} &= \lambda^{-1}(-M_8 + M_{10}) \\
\hat{C}_{32} &= \lambda^{-1}(M_6 + M_7 - M_8 + M_9).
\end{aligned}$$

For example, this rule computes the first entry of the output matrix as $\hat{C}_{11} = A_{11}B_{11} + A_{12}B_{21} - \lambda A_{12}B_{11}$. Thus, in this case, the first entry of the error matrix E in eq. (1) is $E_{11} = A_{12}B_{11}$.

We highlight several properties of the rule which are common across all APA algorithms we consider. First, the rule requires fewer multiplications than the classical one (rank 10 instead of 12 for $\langle 3,2,2\rangle$ here). Next, each multiplication is between a linear combination of entries of A and a linear combination of entries of B, each output entry is computed as a linear combination of the outputs of the multiplications, and each coefficient in the linear combinations is a (Laurent) polynomial in λ. The coefficients include both positive and negative powers of λ, which explains why small values of λ can lead to significant roundoff error. For Bini's algorithm all coefficients are monomial with degree between -1 and 1.

Because of this general pattern, we can encode APA and other fast algorithms succinctly by their linear combination coefficients. For example, encoding the first multiplication M_1 in Bini's algorithm can be done using a triplet of matrices:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} \lambda & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} \lambda^{-1} & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}. \qquad (2)$$

The first two matrices specify the linear combinations taken of entries of A and B, and the third matrix specifies the contributions of M_1 to the entries of $\hat{C}$. Ten such triplets completely specify Bini's algorithm.

2.3 Numerical Error of APA Algorithms

In floating point arithmetic, the lower bound on the numerical error of APA algorithms depends on the working precision and two parameters of the algorithm [7]. The working precision, also referred to as machine precision, is the upper bound on relative error incurred by basic operations in a given floating point format and depends on the number of fractional bits used in the format. We use the notation 2^{-d} for working precision, where $d = 52$ for double and $d = 23$ for single precision (note that $2^{-52} \approx 10^{-16}$ and $2^{-23} \approx 10^{-7}$).

The two parameters of the APA algorithm specify the contribution of the approximation and roundoff errors, respectively. The first parameter, σ, is the smallest positive exponent of the error polynomial and represents the approximation error. Equation (1) shows the error as a polynomial of λ whose leading term is linear in λ. If an algorithm satisfies eq. (1) with $E \neq 0$, then $\sigma = 1$. However, if $E = 0$, then $\sigma > 1$ is the degree of the leading monomial. Larger σ implies smaller error due to the algorithm, though all of the APA algorithms we consider have $\sigma = 1$.

The second parameter, φ, is the largest (in absolute value) negative exponent of the algorithm, computed as the largest sum of negative exponents across all triplets of matrices. This parameter represents the effect of roundoff error caused by floating point arithmetic involving the largest intermediate values computed by the algorithm. For example, the triplet given in eq. (2) yields a sum of negative exponents of $0 + 0 + 1 = 1$, and in the case of Bini's algorithm, no other triplet has a larger sum, so $\varphi = 1$ for that algorithm. Smaller φ implies smaller error due to roundoff, and the APA algorithms we consider exhibit a range of values.

Given these two contributions to the numerical error, λ can be optimized to balance the effects based on parameters σ and φ (and d). As shown by Bini, Lotti, and Romani [7], the optimal λ should be set to $\Theta(2^{-d/(\sigma+\varphi)})$. Using this value of λ, the numerical error incurred by the algorithm will be bounded by $O(2^{-d\sigma/(\sigma+\varphi)})$. Taking Bini's algorithm as an example, we have $\sigma = \varphi = 1$, so the error is $O(2^{-d/2})$, or the square root of working precision. Note that if multiple recursive steps are used, then φ increases proportional to the number of steps, so straightforward optimization of λ results in error of $O(2^{-d\sigma/(\sigma+s\varphi)})$ for s recursive steps. The parameters and minimum error for the algorithms we consider are presented in § 2.5.

We show empirical error results for uniform random inputs of varying dimension in Fig. 1, as compared to the classical algorithm. We measure the relative Frobenius norm error, or $\|C - \hat{C}\|_F / \|C\|_F$, where $\hat{C}$ is computed by each algorithm and C is computed using the classical algorithm in double precision. In order to choose the optimal λ value for each algorithm, we tested the 5 powers of 2 closest to the theoretical optimal value and chose the best.

Overall, we see little fluctuation of the error over matrix dimension, and the theoretical error bound is an upper bound on all empirical errors. Note that the legend is ordered according to the error parameters, and the empirical error generally follows this ordering. The two most accurate APA algorithms are the least recently discovered: Bini's $\langle 3, 2, 2\rangle$ and Schonhage's $\langle 3, 3, 3; 21\rangle$ (note that we indicate the rank of the algorithm here to distinguish from the alternative rank-20 algorithm with the same dimensions). Algorithms that offer more potential speedup tend to be less accurate. Two algorithms with smaller error than expected are $\langle 5, 5, 5\rangle$ and $\langle 7, 2, 2\rangle$, which is explained by the coefficients of those algorithms including fractional pre-factors. While ϕ is computed using the exponents of the largest intermediate term of λ, the leading term of $\langle 5, 5, 5\rangle$ has a constant of $1/4$, lessening its magnitude. We study the effects of the matrix multiplication error on neural network training and test accuracy in § 4.2.

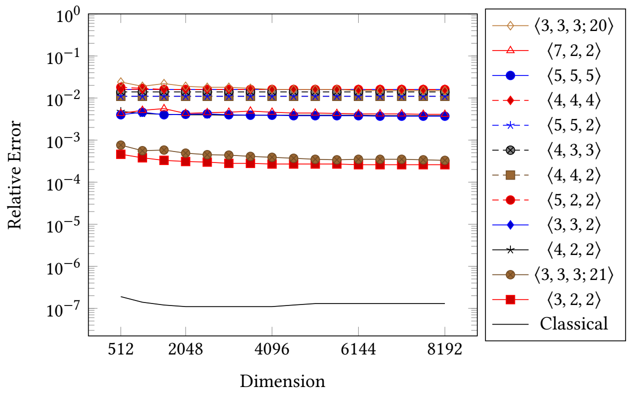


Figure 1: Relative Frobenius norm error for APA algorithms on random inputs.

2.4 Practical Algorithms

We are particularly interested in algorithms with rules for *small* fixed sizes because they have more promise for practical performance. This is because larger fixed sizes result in multiplications of small submatrices, and classical matrix multiplication performance degrades for smaller dimensions. For instance, consider a rule for dimensions $\langle 4, 4, 4\rangle$ applied to matrices of reasonable size, less than dimension 10,000. After one recursive call, the submatrices are of size less than 2500, and after two recursive calls, the dimension is less than 625. At this size, the reduction in number of flops is offset by a reduction in performance, which may result in longer running time.

Instead of focusing on the exponent of the asymptotic complexity of fast algorithms, for practical algorithms we are more interested in the constant reduction in flops of a single recursive level (a single use of the rule of the algorithm). This is because in practice, for reasonable matrix dimensions, only 1 or 2 recursive levels will yield performance improvement [4]. We also prefer algorithms with fewer nonzero coefficients in the linear combinations, because while less costly than multiplications, the matrix additions are less efficient (they are memory bandwidth bound) and prevent achieving the ideal speedup given by the reduction in multiplications. For dimensions $\langle m, n, k\rangle$ and rank r, the ideal speedup for a single recursive step is given by mnk/r, and two recursive steps would

Ref	Dims	Rank	Speedup	σ	φ	Error
-	$\langle 2,2,2\rangle$	8	-	1	0	1.2e-7
[6]	$\langle 3,2,2\rangle$	10	20%	1	1	3.5e-4
[1]	$\langle 4,2,2\rangle$	13	23%	1	2	4.9e-3
[25]	$\langle 3,3,2\rangle$	14	29%	1	3	1.9e-2
[25]	$\langle 5,2,2\rangle$	16	25%	1	3	1.9e-2
[25]	$\langle 3,3,3\rangle$	20	35%	1	6	1.0e-1
[23]	$\langle 3,3,3\rangle$	21	29%	1	2	4.9e-3
[27]	$\langle 7,2,2\rangle$	22	27%	1	5	7.0e-2
[29]	$\langle 4,4,2\rangle$	24	33%	1	3	1.9e-2
[28]	$\langle 4,3,3\rangle$	27	33%	1	3	1.9e-2
[29]	$\langle 5,5,2\rangle$	37	35%	1	3	1.9e-2
[26]	$\langle 4,4,4\rangle$	46	39%	1	3	1.9e-2
[30]	$\langle 5,5,5\rangle$	90	39%	1	3	1.9e-2

Table 1: Properties of APA algorithms. Speedup and error are computed assuming 1 recursive step.

enable a possible speedup of $(mnk/r)^2$. These speedups are typically not fully attained because of degradation in performance for smaller matrix dimensions and the overhead of matrix additions. In the experimental results of this work, we use only 1 recursive step for all algorithms.

2.5 APA Algorithm Properties

Table 1 shows the key performance and accuracy properties of the APA algorithms we consider. Each row corresponds to an algorithm, and the first row includes the classical algorithm for comparison. For all algorithms, we assume only 1 recursive step is used, though the speedup and error for more steps can be readily calculated from the parameters. The first column gives the reference where the algorithm was first specified. The second block column demonstrates the possible performance improvement, where speedup is calculated as $(mnk/r - 1) \cdot 100\%$ for $\langle m, n, k\rangle$ and rank r. The third block column shows error parameters, and the error is calculated as $2^{-d\cdot\sigma/(\sigma+\varphi)}$ with $d = 23$, corresponding to single precision.

3 PARALLEL FAST MATRIX MULTIPLICATION

We exploit the common algorithmic structure across APA (and exact) algorithms to develop a unified strategy of high performance multithreaded implementation using C++/OpenMP. We build upon the work of Benson and Ballard [4], using code generation to apply the strategy to each algorithm.

3.1 Experimental Platform

All experiments are performed on a dual-socket Intel Xeon E5-2620 (Sandy Bridge) server with 2 sockets each with 6 cores. Each socket has a 15 MB L3 cache, and each core has a 256 KB L2 cache and 32 KB L1 data cache. Each core has a clock rate of 2.00 GHz (with turbo boost disabled) and peak single precision flop rate of 32 GFLOPS. Our code is compiled with GCC version 7.5.0, and we use Intel's Math Kernel Library (MKL) version 2019.4.243. We use single precision in all experiments.

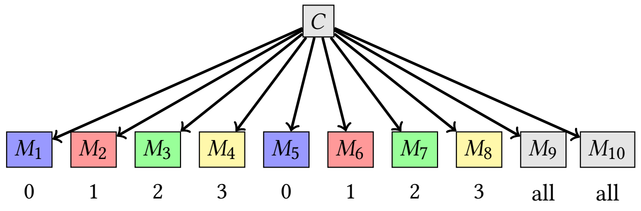


Figure 2: Illustration of hybrid parallelization strategy for $r = 10$ and 4 threads. Each thread is assigned two multiplications to compute using single-threaded gemm and the two remaining multiplications are performed using multithreaded gemm.

3.2 Hybrid Parallelization Strategy

As described in § 2.2, each algorithm can be encoded by a set of triplets of coefficient matrices. From this representation, we generate recursive code that computes the linear combinations for the inputs to each multiplication and combines the outputs. The linear combinations are computed using a "write-once" strategy that was found to be most efficient in terms of memory bandwidth and performance. The multiplications are either computed recursively or via a call to gemm, the interface to a highly efficient BLAS implementation of the classical algorithm. Because we use only 1 recursive step in our experiments, there are no recursive calls, and every multiplication is performed by gemm. We note that the original code generation tool was designed primarily for exact algorithms, and we generalized it to apply to all of the APA algorithms we consider here, some of which have more complicated coefficients than previously considered.

We adopt the "hybrid" parallelization strategy proposed by Benson and Ballard [4]. All linear combinations are parallelized in a straightforward way, in order to maximize the memory bandwidth of the machine. The multiplications are parallelized as follows: given r multiplications and p threads, with $r = p \cdot q + \ell$ for integers q and $\ell < p$, we assign each thread q multiplications to be performed independently, and the remaining ℓ multiplications are performed by all threads using the multithreaded implementation of gemm. Figure 2 shows the hybrid strategy for $r = 10$ (Bini's algorithm) and $p = 4$.

The hybrid strategy is efficient because each thread can achieve close to the peak performance of a core when computing independent matrix multiplications, even for relatively small problems. The alternative strategy of employing multithreaded gemm for each of the r multiplications (known as "DFS") suffers performance degradation for small problems, where the parallel implementation attains a smaller fraction of peak. The hybrid strategy also perfectly load balances the computation across threads, as opposed to the alternate strategy of assigning the ℓ remainder multiplications to ℓ different threads (known as "BFS"), leaving the other $p - \ell$ threads idle.

3.3 Sequential Performance

Figure 3a reports the sequential performance of the APA algorithms in comparison to the most efficient implementation of the classical algorithm, MKL's single-precision gemm. The y-axis is the *effective* GFLOPS, which is measured as $1\text{e-}9 \cdot 2n^3/$ time. That is, the

GFLOPS reported for APA algorithms is not true performance, as they perform fewer flops than the classical algorithm. We use this metric to be able to compare relative times across algorithms performing different amounts of computation. The machine peak for a classical algorithm is given by the horizontal dotted line.

We vary the dimension from 512 up to 8192 and see that all algorithms outperform classical for dimensions larger than 2000 or so. The highest performing algorithm is $\langle 4, 4, 4\rangle$, and at dimension 8192, it is 28% faster than `gemm`. Ignoring the cost of the matrix additions, $\langle 4, 4, 4\rangle$ performs 39% fewer flops than the classical algorithm; the drop to 28% achieved improvement is because of the overhead of matrix additions and the reduced performance of `gemm` on smaller matrices. We note that $\langle 4, 4, 2\rangle$ is also high performing, achieving a 25% observed improvement (out of a theoretical 33%), along with $\langle 3, 3, 3; 20\rangle$ and $\langle 5, 5, 5\rangle$.

3.4 Parallel Performance

We report parallel performance in Figs. 3b and 3c for running with 6 threads (one socket) and 12 threads (both sockets). Overall, speedups of APA algorithms over classical are reduced in the parallel case. Again, the theoretical speedup is based on a reduction in the multiplications, and the overhead of additions is the biggest impediment to realizing that speedup. In the parallel case, the additions can become an even larger bottleneck because the additions are memory bandwidth bound, and the memory bandwidth does not scale with the number of cores [4]. While we can expect close to linear speedup with cores on the multiplications (which are perfectly load balanced), linear scaling is impossible to achieve with the additions. We also note that when the hybrid method uses all threads on remainder multiplications, the smaller dimensions make it harder for multithreaded `gemm` to scale as well as it can on the original matrix dimensions.

In the performance results for 6 threads (Fig. 3b), we see that many of the algorithms start to outperform classical around dimension 2000, though some poor performance is observed for algorithms and particular matrix dimensions. The fastest algorithms (e.g., $\langle 4, 4, 4\rangle$ and $\langle 4, 4, 2\rangle$) achieve a speedup of up to 25% over `gemm` and exceed the machine peak for classical algorithms. We note that the $\langle 4, 4, 2\rangle$ algorithm has 24 subproblems, which is a multiple of 6, so there are no remainder subproblems that require all threads.

Figure 3c shows the results for 12 threads. Here we see a majority of the algorithms are slower than the classical algorithm, even for large matrices. We attribute the poor performance to the effect of lower `gemm` performance for smaller matrices, as well as a lack of NUMA-aware optimization. The "ramp-up" range of `gemm` performance is much shallower for 12 threads than for 6 threads, not achieving the plateau performance until dimension 4000 or so. This implies that the dimension of the submultiplications does not fall on the plateau for any APA algorithm, so the remainder multiplications suffer from poor parallel performance. The algorithm with no remainder multiplications, $\langle 4, 2, 2\rangle$, does not suffer this problem and maintains higher performance. It exceeds `gemm`'s effective performance at dimension 4000, exceeds the peak parallel performance of any classical algorithm for larger dimensions, and achieves a 21% speedup over `gemm` at dimension 8192 for an effective rate of 389 GFLOPS.

4 APPLICATION OF APA ALGORITHMS IN MULTI-LAYER PERCEPTRON NETWORK TRAINING

4.1 MLP Network Structure

To illustrate the effectiveness of APA algorithms in neural network training, we consider Multi-Layer Perceptron networks, as shown in Fig. 4. We use TensorFlow 2.2.0 to build the networks and implement custom operators for matrix multiplication. For internal layers, we use the custom operator for both forward propagation and gradient calculation. For fair comparison against the classical algorithm, we use a custom classical operator that directly calls `gemm`, which significantly outperformed TensorFlow's built-in matrix multiplication operator for fully connected layers.

We show an MLP network with two hidden layers in Fig. 4. The dimensions in the figure match those of the accuracy experiment described in § 4.2. For the performance experiment of § 4.3, we use an MLP with four hidden layers and varying numbers of nodes in the hidden layers.

4.2 Accuracy

To measure the effect of the matrix multiplication error introduced from the APA algorithms on neural network accuracy, we trained an MLP network on the MNIST dataset [20] and measured its accuracy over the course of training.

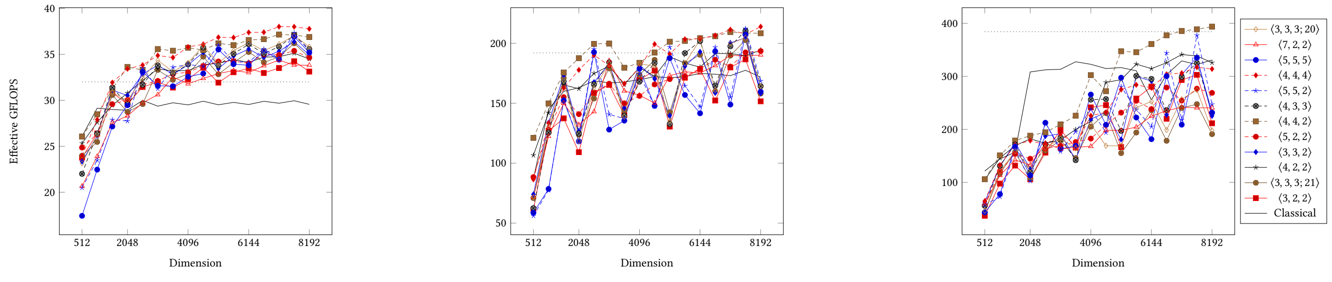
MNIST is a common machine learning dataset of 70,000 images of hand-written numerical digits. Each of the images are composed of 28 by 28 grayscale pixels, which are flattened into a vector of length 784. The dataset is split into 60,000 examples for the training set and 10,000 examples for the test set.

The MLP network is composed of 4 fully-connected layers with 784, 300, 300, and 10 nodes, respectively. The output layer has one node for each digit 0-9. Training is performed using batched stochastic gradient descent with a batch size of 300. APA algorithms are used for the middle multiplication of the network (yielding matrix multiplication dimensions $300 \times 300 \times 300$), while the input and output layers use the classical matrix multiplication algorithm. The APA algorithms are also used during back propagation in the corresponding multiplications in the center of the network. One network is trained for 50 epochs for each of the APA algorithms, and one is trained using the classical matrix multiplication algorithm.

Figure 5a plots the accuracy on the training data over each epoch, and Fig. 5b shows the test accuracy over epochs. The matrix multiplication error introduced from these algorithms has minimal effect on the training error; all algorithms yield convergence to nearly full accuracy after 20 epochs or so. We see more variability in the test error due to the sensitivity inherent in the generalizability of the model, though all algorithms achieve between 97% and 99% test accuracy.

4.3 Performance

The neural networks used for performance benchmarking were 6-layer MLPs (4 hidden layers), and were trained on the MNIST dataset. We base the structure of the network on the fully connected canonical model of ParaDnn [32]. The purpose of these experiments was to measure the speed up in training time provided by the APA



(a) One thread **(b) Six threads** **(c) Twelve threads**

Figure 3: Single- and multi-threaded square matrix multiplication performance, relative to $2n^3$ operations. The machine peak with respect to classical algorithms is given as a dotted line.

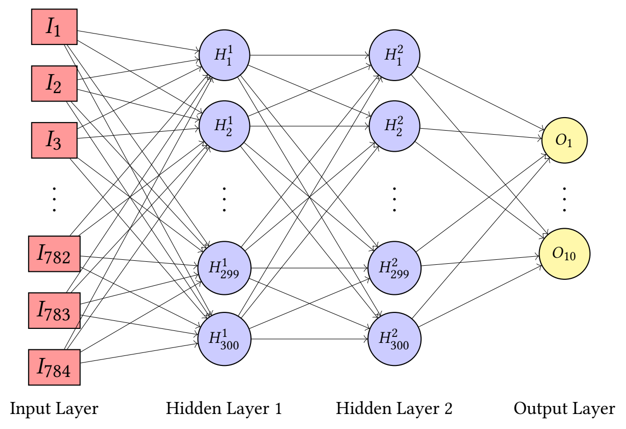


Figure 4: Multi-Layer Perceptron network structure

algorithms and not to measure the accuracy of the networks. The APA algorithms are used only in the multiplications in the hidden layers during both forward and backward propagation; the standard operation was used in the input and output layers.

Following the methodology of ParaDnn, we measure the performance of each algorithm with the number of nodes in each hidden layer varying from 512 to 8192. We also match the batch size of each network to the number of nodes in the hidden layer to produce square matrix multiplications in the hidden layers. Fig. 6 shows the empirical results using 1, 6, and 12 threads.

As shown in Fig. 6a, the single-threaded results begin to show speed up over the classical algorithm when the matrix dimensions are at least 1024. The highest performing algorithm is ⟨4, 4, 4⟩, which obtains a 25% speed up at dimension 8192. All algorithms outperform classical for dimensions 4096 and 8192.

Figure 6b shows the results for six threads (one socket). As described in § 3.4, the APA algorithms do not scale to higher numbers of cores as efficiently as `gemm`, due to both the overhead of matrix additions (a memory bandwidth bottleneck) and reduced parallel performance of `gemm` for smaller dimensions. In this case, not all APA algorithms outperform the classical algorithm. The highest performing algorithms are ⟨4, 4, 2⟩ and ⟨4, 4, 4⟩, which obtain an improvement of 13% for dimension 8192.

In the case of 12 threads (both sockets), as shown in Fig. 6c, most APA algorithms underperform the classical algorithm. These results match the behavior of standalone matrix multiplications using 12 threads (see § 3.4); the lack of scaling is due in large part to the poor performance of multithreaded `gemm` on remainder multiplications of small dimensions. The algorithm with a number of sub-multiplications that is a multiple of 12, ⟨4, 4, 2⟩, continues to perform well and is slightly faster than the classical algorithm for dimensions 4096 and 8192, attaining speedups of up to 7%.

5 APPLICATION OF APA ALGORITHMS IN VGG-19

VGG-19 [24] is a convolutional neural network typically trained on the ImageNet dataset as a classification task. The model is one of the highest performing algorithms for this task, and the network is composed of 19 layers, 16 convolutional and 3 fully connected. The 3 fully connected layers consist of 25,088, 4096, and 1000 nodes, creating very large matrix multiplications in forward and backward propagation. By using faster APA algorithms for the multiplication, we are able to speed up the training of the network.

Replacing the classic matrix multiplication algorithm in these layers with the ⟨4, 4, 2⟩ algorithm provides up to a 10% speed up of the fully connected layers during training with 6 threads and a 15% speed up using a single thread. Figure 7 shows the per-batch training time of the fully connected layers across batch sizes.

6 CONCLUSION

Arbitrary Precision Approximating algorithms offer significant practical performance improvements over exact fast matrix multiplication algorithms. They outperform classical matrix multiplication at smaller matrix dimensions and yield greater benefits as matrix dimensions increase. APA algorithms sacrifice accuracy, achieving error of only a fractional root of the working precision, but they can be practical for applications that do not demand high accuracy. The results presented here show that neural network training, particularly for Multi-Layer Perceptron networks, is robust to such matrix multiplication error, and APA algorithms are a useful tool for accelerating the costly training computations. We observe that for APA algorithms offering a theoretical speedup of up to 39%, we can achieve up to 28% and 21% improvements for single- and

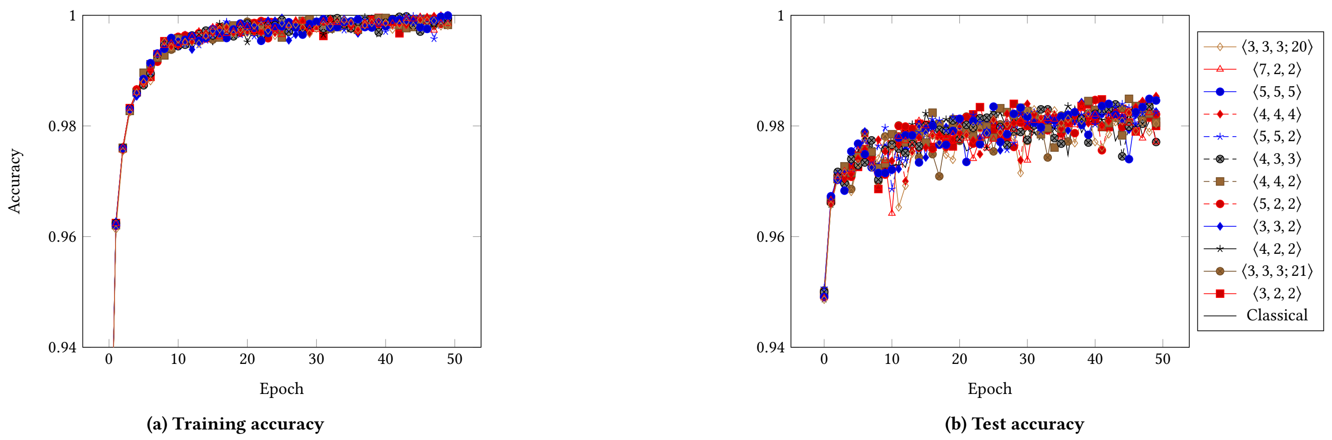


Figure 5: MLP network accuracy for MNIST data set

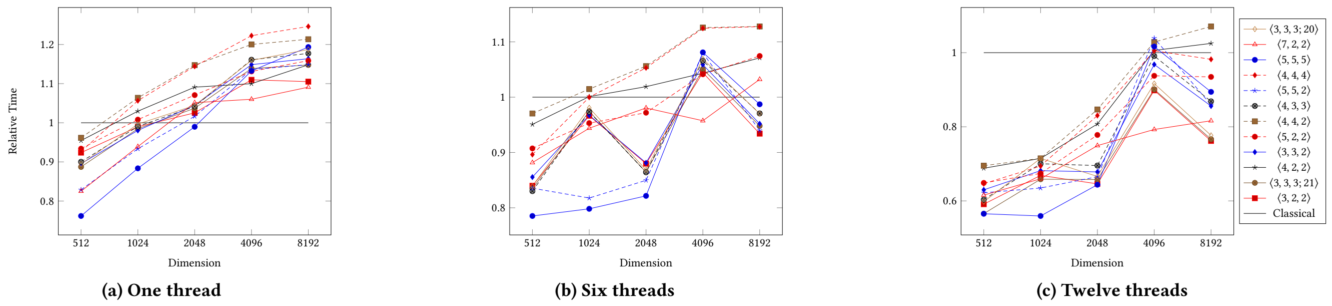


Figure 6: Network training time relative to using classical matrix multiplication

multi-threaded implementations of individual matrix multiplications, respectively, which translate to up to 13% speedup in overall training time on a multicore CPU.

While the accuracy and performance results are promising for MLP networks and the MNIST dataset, we believe that APA algorithms can also be effective in state-of-the-art networks and more difficult machine learning problems. For example, the VGG-19 network for large-scale visual recognition [24] is based on a deep convolutional network and is bottlenecked by large fully connected layers. Our result are able to show a 15% performance improvement using the sequential algorithm, and a 10% improvement using parallel algorithm. This highlights the practical applications on training large networks.

As illustrated by Benson and Ballard [4], the highest performing fast algorithms for rectangular matrix multiplications often have dimensions that match the aspect ratio of the problem. We consider in this paper only square matrix multiplications, and we see that square (or nearly square) dimensions of $\langle 4,4,4\rangle$ and $\langle 4,4,2\rangle$ are fastest. Algorithms with more skewed aspect ratios will likely perform better for problems with matching skewed matrix multiplication dimensions. We note that an algorithm for dimensions $\langle m,n,k\rangle$ can be translated into an algorithm for $\langle n,m,k\rangle$ and any other reordering of the dimensions [4]. For very large matrices, it may be worth considering more than one recursive step of the same algorithm or a combination of two or three different algorithms across recursive steps (uniform, non-stationary algorithms [3]).

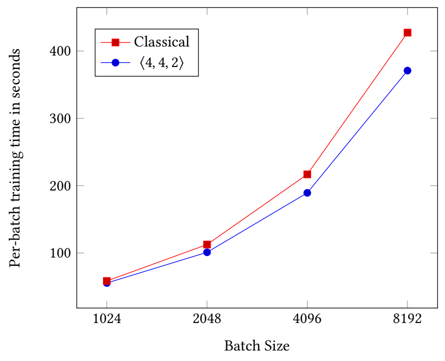
Finally, we would like to extend the code generation techniques to other hardware platforms, including GPUs. Because GPUs can offer higher efficiency for matrix multiplications, we wish to demonstrate the potential of APA algorithms in that environment. Techniques used to accelerate Strassen's algorithm on the GPU [15, 19] can be generalized to other fast algorithms as done in work for multicore CPUs [4, 14]. We believe improved performance benefits can be achieved by APA algorithms on GPU architectures due to the relatively higher memory bandwidth.

Acknowledgments. This material is based upon work supported by the National Science Foundation Grant No. 1942892 and a Wake Forest University Undergraduate Research and Creative Activities (URECA) Research Fellowship.

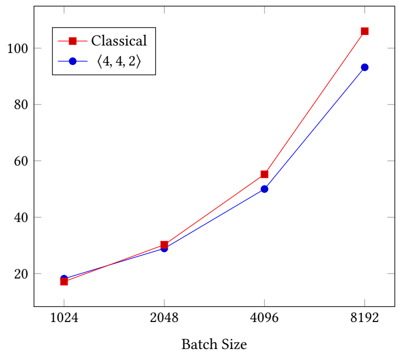
REFERENCES

[1] V.B. Alekseev and A.V. Smirnov. 2013. On the exact and approximate bilinear complexities of multiplication of 4x2 and 2x2 matrices. *Proceedings of the Steklov Institute of Mathematics* 282, 1 (2013), 123–139. https://doi.org/10.1134/S0081543813070079

[2] Josh Alman and Virginia Vassilevska Williams. 2021. A Refined Laser Method and Faster Matrix Multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on*



(a) One thread



(b) Six threads

Figure 7: Per-batch training time on the fully connected layers of VGG-19 using $\langle 4, 4, 2\rangle$ and Classical

Discrete Algorithms (SODA 21). 522–539. https://doi.org/10.1137/1.9781611976465.32

[3] Grey Ballard, Austin R. Benson, Alex Druinsky, Benjamin Lipshitz, and Oded Schwartz. 2016. Improving the Numerical Stability of Fast Matrix Multiplication. *SIAM J. Matrix Anal. Appl.* 37, 4 (2016), 1382–1418. https://doi.org/10.1137/15M1032168

[4] Austin R. Benson and Grey Ballard. 2015. A Framework for Practical Parallel Fast Matrix Multiplication. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Francisco, CA, USA) *(PPoPP 2015)*. ACM, New York, NY, USA, 42–53. https://doi.org/10.1145/2688500.2688513

[5] D. Bini. 1980. Relations between exact and approximate bilinear algorithms. Applications. *CALCOLO* 17, 1 (1980), 87–97. https://doi.org/10.1007/BF02575865

[6] D. Bini, M. Capovani, F. Romani, and G. Lotti. 1979. $O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication. *Information Processing Letters* 8, 5 (1979), 234–235. https://doi.org/10.1016/0020-0190(79)90113-3

[7] Dario Bini, Grazia Lotti, and Francesco Romani. 1980. Approximate solutions for the bilinear form computational problem. *SIAM J. Comput.* 9, 4 (1980), 692–697. http://epubs.siam.org/doi/10.1137/0209053

[8] Tom Brown et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[9] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. *cuDNN: Efficient Primitives for Deep Learning*. Technical Report 1410.0759. arXiv. http://arxiv.org/abs/1410.0759

[10] D. Coppersmith and S. Winograd. 1987. Matrix multiplication via arithmetic progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, New York, United States) *(STOC '87)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/28395.28396

[11] Evangelos Georganas, Kunal Banerjee, Dhiraj D. Kalamkar, Sasikanth Avancha, Anand Venkat, Michael J. Anderson, Greg Henry, Hans Pabst, and Alexander Heinecke. 2019. *High-Performance Deep Learning via a Single Building Block*. Technical Report 1906.06440. arXiv. http://arxiv.org/abs/1906.06440

[12] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32nd International Conference on Machine Learning (ICML '15, Vol. 37)*, Francis Bach and David Blei (Eds.). PMLR, Lille, France, 1737–1746. http://proceedings.mlr.press/v37/gupta15.html

[13] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. 1989. Multilayer feedforward networks are universal approximators. *Neural networks* 2, 5 (1989), 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

[14] Jianyu Huang, Leslie Rice, Devin A. Matthews, and Robert van de Geijn. 2017. Generating Families of Practical Fast Matrix Multiplication Algorithms. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium*. 656–667. https://doi.org/10.1109/IPDPS.2017.56

[15] Jianyu Huang, Chenhan D. Yu, and Robert A. van de Geijn. 2020. Strassen's Algorithm Reloaded on GPUs. *ACM Trans. Math. Software* 46, 1, Article 1 (March 2020), 22 pages. https://doi.org/10.1145/3372419

[16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research* 18, 1 (2017), 6869–6898. https://jmlr.org/papers/v18/16-456.html

[17] Dhiraj D. Kalamkar et al. 2019. *A Study of BFLOAT16 for Deep Learning Training*. Technical Report abs/1905.12322. arXiv. http://arxiv.org/abs/1905.12322

[18] Ahmed Khaled, Amir F. Atiya, and Ahmed H. Abdel-Gawad. 2020. Applying Fast Matrix Multiplication to Neural Networks. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic) *(SAC '20)*. Association for Computing Machinery, New York, NY, USA, 1034–1037. https://doi.org/10.1145/3341105.3373852

[19] P. Lai, H. Arafat, V. Elango, and P. Sadayappan. 2013. Accelerating Strassen-Winograd's matrix multiplication algorithm on GPUs. In *20th Annual International Conference on High Performance Computing*. 139–148. https://doi.org/10.1109/HiPC.2013.6799109

[20] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-Based Learning Applied to Document Recognition. *Proc. IEEE* 86, 11 (November 1998), 2278–2324. https://doi.org/10.1109/5.726791

[21] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry Vetrov. 2015. Tensorizing Neural Networks. In *Advances in Neural Information Processing Systems*, Vol. 28. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2015/file/6855456e2fe46a9d49d3d3af4f57443d-Paper.pdf

[22] V. Pan. 1984. How Can We Speed Up Matrix Multiplication? *SIAM Rev.* 26, 3 (1984), 393–415. https://doi.org/10.1137/1026076

[23] Arnold Schönhage. 1981. Partial and total matrix multiplication. *SIAM J. Comput.* 10, 3 (1981), 434–455. https://doi.org/10.1137/0210032

[24] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*. https://arxiv.org/abs/1409.1556

[25] A.V. Smirnov. 2013. The bilinear complexity and practical algorithms for matrix multiplication. *Computational Mathematics and Mathematical Physics* 53, 12 (2013), 1781–1795. https://doi.org/10.1134/S0965542513120129

[26] A.V. Smirnov. 2014. *The Approximate Bilinear Algorithm of Length 46 for Multiplication of 4 x 4 Matrices.* Technical Report 1412.1687. arXiv. https://arxiv.org/abs/1412.1687

[27] A.V. Smirnov. 2015. A bilinear algorithm of length 22 for approximate multiplication of 2 x 7 and 7 x 2 matrices. *Computational Mathematics and Mathematical Physics* 55, 4 (2015), 541–545. https://doi.org/10.1134/S0965542515040168

[28] A.V. Smirnov. 2016. *An Approximate Bilinear Algorithm of Length 27 for Multiplication of 3 x 3 and 3 x 4 Matrices.* Technical Report. ResearchGate. https://www.researchgate.net/publication/299599750

[29] A.V. Smirnov. 2016. *On the Approximate Bilinear Algorithms for Multiplication of N x N and N x 2 Matrices.* Technical Report. ResearchGate. https://www.researchgate.net/publication/308992223

[30] A.V. Smirnov. 2018. *An Approximate Bilinear Algorithm for Multiplying 5 x 5 Matrices of Length.* Technical Report. ResearchGate. https://www.researchgate.net/publication/329800232

[31] V. Strassen. 1969. Gaussian elimination is not optimal. *Numerische Mathematik* 13 (1969), 354–356. Issue 4. https://doi.org/10.1007/BF02165411

[32] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. 2020. A Systematic Methodology for Analysis of Deep Learning Hardware and Software Platforms. In *Proceedings of Machine Learning and Systems (MLSys '20, Vol. 2)*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.). 30–43. https://proceedings.mlsys.org/paper/2020/hash/c20ad4d76fe97759aa27a0c99bff6710-Abstract.html

[33] V. Williams. 2012. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the 44th Annual Symposium on Theory of Computing* (New York, New York, USA) *(STOC '12)*. ACM, 887–898. https://doi.org/10.1145/2213977.2214056

[34] D. Yan, W. Wang, and X. Chu. 2020. Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS '20)*. 634–643. https://doi.org/10.1109/IPDPS47924.2020.00071