Applying Fast Matrix Multiplication to Neural Networks

Ahmed Khaled
Cairo University
Giza, Egypt
akregeb@gmail.com

Amir F. Atiya
Cairo University
Giza, Egypt
aatiya1@gmail.com

Ahmed H. Abdel-Gawad
Cairo University
Giza, Egypt
ah.hamdy@gmail.com

ABSTRACT

Recent advances in deep neural networks have enabled impressive performance in computer vision, natural language processing, and other fields, yet they remain computationally very intensive to train or use. We consider the use of Winograd's Algorithm for fast matrix multiplication in feedforward neural networks and we find that speedups of 10% − 30% are possible for fully connected layers in large networks.

CCS CONCEPTS

• **Computing methodologies** → *Vector / streaming algorithms*; *Machine learning*;

KEYWORDS

Strassen's algorithm, Winograd's algorithm, GPU matrix multiplication, Fast Matrix Multiplication, Neural Networks

ACM Reference Format:
Ahmed Khaled, Amir F. Atiya, and Ahmed H. Abdel-Gawad. 2020. Applying Fast Matrix Multiplication to Neural Networks. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3341105.3373852

1 INTRODUCTION

Deep neural networks are evolving towards solving problems with ever increasing complexity. Problems including the use of multi-input types, such as handling video, sound, and text simultaneously are yet to be solved. However, the increasing complexity of such types of problems necessitate using larger and more powerful networks, possibly possessing tens of thousands of neurons. For these, the computational load becomes excessive, and methods to speed-up the training are needed.
Both training and inference in feedforward neural networks can be modelled as a sequence of matrix-matrix multiplications and element-wise operations, with matrix multiplications taking the bulk of computation time, being at the heart of convolutional [3] as well as fully-connected layers. The current widely-used algorithms for matrix multiplication in neural networks are variants of the GotoBLAS Algorithm [7]. The algorithm's asymptotic runtime is $O\left(n^3\right)$ where n denotes the leading dimension of the matrices. In comparison, Strassen's algorithm is a divide-and-conquer algorithm whose asymptotic runtime is approximately $O(n^{2.8074})$ but which has not been widely used because it is faster only for very large matrices [6].
In this paper, we consider the use of Winograd's algorithm, a Strassen-like matrix multiplication algorithm, in neural networks. We implement a GPU variant of Winograd matrix multiplication [11] as a TensorFlow [1] custom operator and find that for very wide layers and large batch sizes (number of neurons, features, and training samples > 5K) the algorithm yields moderate speedups in inference and training.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SAC'20, March 30-April 3, 2020, Brno, Czech Republic
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3373852

2 BACKGROUND

2.1 Fully Connected Layers

A fully-connected layer in a neural network computes an affine transformation of its input vectors followed by a non-linearity. Suppose that we have a batch of N input vectors $\{x^1, \ldots, x^N\}$ of dimensionality K. Let $X \in \mathbb{R}^{N\times K}$ be the matrix formed by stacking the input vectors together in a row-wise fashion:

$$X = \begin{bmatrix} x_1^1 & x_2^1 & x_3^1 & \ldots & x_K^1 \\ x_1^2 & x_2^2 & x_3^2 & \ldots & x_K^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^N & x_2^N & x_3^N & \ldots & x_K^N \end{bmatrix}.$$

Let M be the output dimensionality for the layer, let $W \in \mathbb{R}^{K\times M}$ be the weights matrix and let $B \in \mathbb{R}^{N\times M}$ be the matrix formed by stacking the bias vector b row-wise N times:

$$B = \begin{bmatrix} b_1 & b_2 & b_3 & \ldots & b_M \\ b_1 & b_2 & b_3 & \ldots & b_M \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_1 & b_2 & b_3 & \ldots & b_M \end{bmatrix}.$$

Then the forward computations for the layer can be written as:

$$Y = \sigma(XW + B)$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is the activation function, with the understanding that when applied to a matrix it is applied element-wise and $Y \in \mathbb{R}^{N\times M}$ is the layer output matrix. For the backward pass, given a loss function L suppose that $\frac{\partial L}{\partial Y}$ has been propagated down the graph to the layer, then we have

$$\frac{\partial L}{\partial W} = X^T \left(\sigma'(XW + B) \odot \frac{\partial L}{\partial Y}\right),$$
$$\frac{\partial L}{\partial B} = \sigma'(XW + B) \odot \frac{\partial L}{\partial Y}.$$

where $\sigma' : \mathbb{R} \to \mathbb{R}$ is the derivative of σ and $\odot$ denotes the Hadamard (element-wise) product of matrices.

2.2 Fast Matrix Multiplication

Strassen's algorithm is a bilinear divide-and-conquer algorithm aimed at efficient matrix multiplication [6]. We use a variant from the same family (Winograd's algorithm) implemented on the GPU following [11]: suppose that $A \in \mathbb{R}^{N \times K}$, $B \in \mathbb{R}^{K \times M}$ and we'd like to compute $C = AB$. Assuming that the matrix sizes are even, we can write a block decomposition of the matrices as follows:

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

The straightforward method of multiplying the block matrices would require 8 multiplications and 4 additions. Instead, Winograd's algorithm (Algorithm 1) provides us a way of computing C using 7 multiplications and 18 additions. If the matrix products in Algorithm 1 are computed with the straightforward algorithm, then ignoring the cost of extra additions we can expect an asymptotic reduction in the number of operations of $\frac{7}{8}$. Applying it recursively to the resultant matrices for r levels results in a $(\frac{7}{8})^r$ expected reduction in the number of operations. In practice, there are numerical stability issues as well significant overhead associated with using more than two levels [10]. In this work, we consider only a single such level of recursion for single-precision floating point numbers and at most two for double-precision floating point numbers as more levels of recursion were not found to be useful for the matrix sizes tested.

Algorithm 1 Winograd's Algorithm for $C = A \times B$

```
$S_1 = A_{21} + A_{22}$    $M_1 = S_2 \times S_6$         $V_1 = M_1 + M_2$
$S_2 = S_1 - A_{11}$       $M_2 = A_{11} \times B_{11}$   $V_2 = V_1 + M_4$
$S_3 = A_{11} - A_{21}$    $M_3 = A_{12} \times B_{21}$   $V_3 = M_5 + M_6$
$S_4 = A_{12} - S_2$       $M_4 = S_3 \times S_7$         $C_{11} = M_2 + M_3$
$S_5 = B_{12} - B_{11}$    $M_5 = S_1 \times S_5$         $C_{12} = V_1 + V_3$
$S_6 = B_{22} - S_5$       $M_6 = S_4 \times B_{22}$      $C_{21} = V_2 - M_7$
$S_7 = B_{22} - B_{12}$    $M_7 = A_{22} \times S_8$      $C_{22} = V_2 + M_5$
$S_8 = S_6 - B_{21}$
```

Using the previous formulation directly requires large memory, instead another variant from [11] is used with only two temporary matrices, with matrix addition and matrix multiplication kernels being used from cuBLAS, the CUDA library for BLAS (Basic Linear Algebra Subprograms). See Algorithm 2. Because the algorithm calls cuBLAS's sgemm routine, all the optimizations for multiplying matrices in Steps 3, 6, 9, 11, 13, 18, and 21 carry over from cuBLAS. However, there is limited parallelism between each of these operations and between the matrix addition operations. Given enough memory, further parallelism and/or streaming optimizations could be employed to execute multiple such operations in parallel.

3 APPLYING FAST MATRIX MULTIPLICATION

For time usage and speedup measurements we use CUDA v9.2 and TensorFlow 1.9.0 running on a 64-bit Intel Core i7-5500U with a 1,155 peak single-precision T-Flop/s NVIDIA GeForce GTX 850M GPU with 4 GB of DDR3 GPU memory.

The result graphs are shown in Figure 1. The graph was created by randomizing the weight and input matrices and comparing the time taken by Winograd's algorithm against the sgemm routine from cuBLAS [13], a widely-used (in Deep Learning and Machine Learning applications) implementation of BLAS on the GPU by NVIDIA.

The lines shown indicate the speedup in matrix multiplication for a single fully connected layer with different configurations. The number of training samples (i.e., the M dimension) must be large for benefits to show up, at least for the current implementation. The larger the number of features or the number of neurons in layers, the more speedup can be expected. Our results therefore indicate that for typical neural networks the matrices are too small, but are promising for applications that require neural networks in which the number of features and the sizes of the layers are very large.

Algorithm 2 Computing $C = A \times B$ with two temporary matrices

Step	Operation	Function
1	$T_1 = A_{11} - A_{21}$	MA
2	$T_2 = B_{22} - B_{12}$	MA
3	$C_{21} = T_1 \times T_2$	MM
4	$T_1 = A_{21} + A_{22}$	MA
5	$T_2 = B_{12} - B_{11}$	MA
6	$C_{22} = T_1 \times T_2$	MM
7	$T_1 = T_1 - A_{11}$	MA
8	$T_2 = B_{22} - T_2$	MA
9	$C_{12} = T_1 \times B_{22}$	MM
10	$T_1 = A_{12} - T_1$	MA
11	$C_{12} = T_1 \times B_{22}$	MM
12	$C_{12} = C_{12} + C_{22}$	MA
13	$T_1 = A_{11} \times B_{11}$	MM
14	$C_{11} = C_{11} + T_1$	MA
15	$C_{12} = C_{12} + C_{11}$	MA
16	$C_{11} = C_{11} + C_{21}$	MA
17	$T_2 = T_2 - B_{21}$	MA
18	$C_{21} = A_{22} \times T_2$	MM
19	$C_{21} = C_{11} - C_{21}$	MA
20	$C_{22} = C_{22} + C_{11}$	MA
21	$C_{11} = A_{12} \times B_{21}$	MM
22	$C_{11} = T_1 + C_{11}$	MA

4 RELATED WORK

Strassen's algorithm and its variants have been implemented for GPUs first by Li et al. [12] and then by Lai et al. [11] (used here) which has remained the state-of-the-art until the recent work of Huang et al. [10] whose implementation shows speedups even for relatively small ($N > 1500$) matrices and has a much smaller memory footprint (not requiring extra temporary matrices beyond those ordinarily used in matrix multiplication algorithms) even when using two levels of recursion. Because of its better response to rectangular matrices, it could be used for matrix multiplication in 2D convolution where the multiplied matrices are typically very rectangular, see Table II in [14]. Exploring the use of Huang et al.'s algorithm in feedforward and convolutional neural networks is an

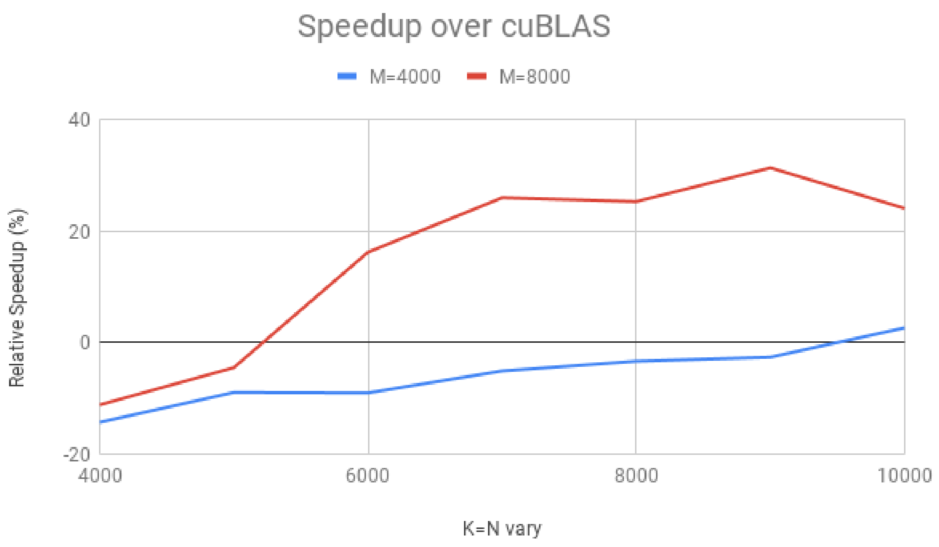

Figure 1: Results (Speedup) of Algorithm 2 for $M \times N$ and $N \times K$ matrix multiplication in fully connected layers. The speedup is defined as $\left(\frac{\text{Time with Vanilla Matrix Mult.}}{\text{Time with Algorithm 2}} - 1\right) \times 100\%$. Positive means Algorithm 2 is faster.

avenue for future research.

Earlier work by Huang et al. [9] provided several implementation of Strassen's algorithm on the CPU, with some achieving a switch point at $N = 500$ for double-precision matrices. However, currently the BLAS-like Library Instantiation Software Framework (BLIS) framework used as a basis for the CPU implementation fares worse in comparison to Intel's Math Kernel Library (MKL) [8] for single-precision numbers on matrices with dimension $< 8K$. On implementing the algorithm in [9], we found that for the tested matrix sizes even though some speedup was obtained against BLIS, there was no speedup against MKL.

Rao and Ramana [5] apply a parallelized implementation of Strassen's algorithm to deep neural networks and demonstrate speedups over the classical algorithm, but their results are severely hampered in that they do not compare against an efficient implementation of matrix multiplication (from e.g., Intel's MKL or GotoBLAS) but against their own unoptimized implementation of the classical algorithm.

Cong and Xiao [4] applied Strassen's algorithm to matrix multiplication but with matrices in which the scalars are $K \times L$ kernels or feature maps instead of numbers, the multiplication operation is replaced by the convolution operation, and the addition operation is the addition of feature maps. They demonstrate that the usage of Strassen's algorithm applied to this form of matrix multiplication results in up to 64% time savings in certain convolutional layers. Exploring the use of algorithm 2 or similar algorithms on GPUs in combination with primitives for convolution is an avenue of future work.

More generally, approximate methods have been used to accelerate matrix multiplication and other tensor operations in neural networks: Osawa et al. [14] use a low-rank approximation and find that the impact on accuracy is relatively small, however their implementation was not efficient enough to demonstrate practical speedup. Adelman & Silberstein [2] use column-row sampling and find little impact on accuracy, but the efficient implementation of approximate schemes is still open. Looking beyond training, Tschannen et al. [15] use a neural network architecture capable of learning Strassen-like decompositions of matrix multiplication with the goal of speeding up inference in pretrained networks by representing matrix multiplication as two-layer sum-product networks (SPNs) with weights taking ternary ($\{-1, 0, 1\}$) values and training these weights to minimize the error on the network. They obtain a theoretical reduction in the number of multiplications up to 100% with a minimal loss of accuracy but experimental results are still lacking. Because their focus is on inference, their approach can be combined with our approach (or any of the other approaches mentioned for training).

5 CONCLUSION

While our findings show that small neural networks are unlikely to benefit much from fast matrix multiplication, our results are promising for wide networks. As neural networks used in practice become wider, using Winograd's algorithm or any of its variants is likely to yield small but significant speedups, especially over the whole period of training. We believe our results will encourage more research and experimentation with other neural network architectures or matrix multiplication algorithms.

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K.

Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *ArXiv e-prints* (March 2016). arXiv:cs.DC/1603.04467

[2] M. Adelman and M. Silberstein. 2018. Faster Neural Network Training with Approximate Tensor Operations. *ArXiv e-prints* (May 2018). arXiv:1805.08079

[3] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *ArXiv e-prints* (Oct. 2014). arXiv:1410.0759

[4] Jason Cong and Bingjun Xiao. 2014. Minimizing Computation in Convolutional Neural Networks. In *Artificial Neural Networks and Machine Learning – ICANN 2014*, Stefan Wermter, Cornelius Weber, Włodzisław Duch, Timo Honkela, Petia Koprinkova-Hristova, Sven Magg, Günther Palm, and Alessandro E. P. Villa (Eds.). Springer International Publishing, Cham, 281–290.

[5] Dharmajee D.T.V. and K.V. Ramana. 2018. Speeding Up Training of Deep Neural Networks on Multi-Core Processors Using Open MP. *Journal of Advanced Research in Dynamical and Control Systems* 10 (May 2018), 556–565.

[6] J.-G. Dumas and V. Pan. 2016. Fast Matrix Multiplication and Symbolic Computation. *ArXiv e-prints* (Dec. 2016). arXiv:cs.SC/1612.05766

[7] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (May 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[8] Jianyu Huang, Leslie Rice, Devin A. Matthews, and Robert A. van de Geijn. 2016. Generating Families of Practical Fast Matrix Multiplication Algorithms. *CoRR* abs/1611.01120 (2016). arXiv:1611.01120 http://arxiv.org/abs/1611.01120

[9] J. Huang, T. M. Smith, G. M. Henry, and R. A. van de Geijn. 2016. Implementing Strassen's Algorithm with BLIS. *ArXiv e-prints* (May 2016). arXiv:cs.MS/1605.01078

[10] J. Huang, C. D. Yu, and R. A. van de Geijn. 2018. Implementing Strassen's Algorithm with CUTLASS on NVIDIA Volta GPUs. *ArXiv e-prints* (Aug. 2018). arXiv:cs.MS/1808.07984

[11] 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

[12] J. Li, S. Ranka, and S. Sahni. 2011. Strassen's Matrix Multiplication on GPUs. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. 157–164. https://doi.org/10.1109/ICPADS.2011.130

[13] NVIDIA. 2018. Nvidia cuBLAS. Available Online. https://developer.nvidia.com/cublas. (2018).

[14] K. Osawa, A. Sekiya, H. Naganuma, and R. Yokota. 2017. Accelerating Matrix Multiplication in Deep Learning by Using Low-Rank Approximation. In *2017 International Conference on High Performance Computing Simulation (HPCS)*. 186–192. https://doi.org/10.1109/HPCS.2017.37

[15] M. Tschannen, A. Khanna, and A. Anandkumar. 2017. StrassenNets: Deep Learning with a Multiplication Budget. *ArXiv e-prints* (Dec. 2017). arXiv:1712.03942