Accuracy-Aware Approximate Matrix Multiplication and Convolution for CUDA

Keyur Joshi
University of Illinois at Urbana-Champaign
kpjoshi2@illinois.edu

Ziang Wan
University of Illinois at Urbana-Champaign
ziangw2@illinois.edu

ABSTRACT

Matrix multiplication and convolution are two of the most time consuming operations performed when using a neural network. Researchers have proposed several methods that perform such operations in an approximate manner, sacrificing accuracy in favor of performance. ApproxHPVM is a parallel program representation which applies approximations in an accuracy aware manner, while attempting to maximize performance. AxProf is a framework for accuracy profiling of approximate algorithms.

We propose adding approximate matrix multiplication and convolution to the suite of approximations available in ApproxHPVM. We implement these approximations in CUDA and use AxProf to measure their performance and accuracy. This data can then be used by the ApproxHPVM autotuner to automatically select these algorithms when appropriate for performance gains.

1 INTRODUCTION

Neural networks are used for a variety of real-life applications, such as in self-driving cars, spam detection, and automatic image classification. Networks that take images as input typically consist of multiple convolution layers, followed by fully-connected layers. The most time consuming operations performed in these layers are convolution and matrix multiplication respectively. These operations often become the bottleneck in increasing the performance of neural networks.

To combat this, researchers are exploring approximate algorithms for matrix multiplication and convolution. These approximate algorithms can provide significant performance gains, while maintaining an acceptable level of accuracy. Recently, researchers proposed AxProf [5], a framework for accuracy profiling of approximate algorithms. AxProf helps developers to ensure that the implementation of an algorithm follows a provided accuracy specification. Researchers have also proposed ApproxHPVM, a representation for parallel programs that performs accuracy-aware approximations to improve performance of neural networks. However, currently it focuses on hardware approximations.

We propose adding software approximations to the suite of approximations supported by ApproxHPVM. We implemented our approximate matrix multiplication and convolution algorithms in CUDA. We then used AxProf to test these algorithms with a variety of inputs to measure their effects on performance and accuracy. Using our results, the ApproxHPVM autotuner can determine when our approximate algorithms can be used in lieu of exact algorithms while keeping accuracy at acceptable levels.

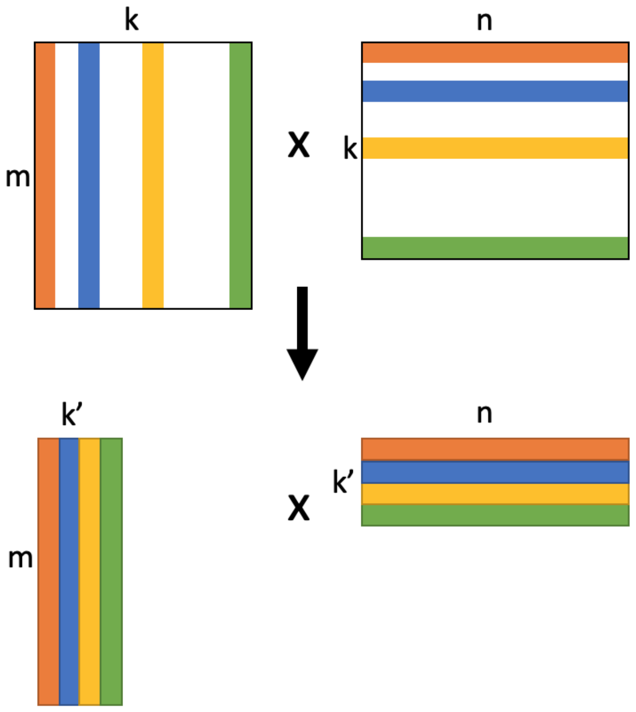

Figure 1: Sampling for Approximate Matrix Multiplication

2 APPROACH

We propose implementing software approximations for matrix multiplication and convolution operations. We then test our approximations with AxProf to measure performance and accuracy effects.

2.1 Approximate Matrix Multiplication

We perform approximate matrix multiplication via the method proposed by Drineas et al. [1]. Figure 1 illustrates the sampling operation at the heart of this method.

The method consists of four steps:

(1) Weigh the columns of the first matrix and the rows of the second matrix according to a metric calculated from their ℓ_2 norms.
(2) Perform weighted random sampling with replacement using these weights to generate smaller matrices.
(3) Scale the sampled rows and columns by scaling multipliers. These multipliers compensate for the sampling operation by scaling the matrix rows and columns appropriately.

(4) Perform multiplication on the sampled and adjusted matrices. As the sampled matrices are smaller, this operation can be performed faster.

The result of this operation is approximately equal to the product of the original two matrices. The exact specification of accuracy is provided in the same paper [1].

We further extend this approach by experimenting with different variations for the four steps above.

Step 2 modification: sampling without replacement: The original approach uses weighted random sampling with replacement. As a result, the same row or column may be selected to appear more than once in the sampled matrix. We experiment with weighted random sampling without replacement, which ensures that each row or column occurs at most once in the sampled matrix. The advantage of this approach is that more rows and columns from the original matrix are represented in the sampled matrix, potentially improving accuracy. However, sampling without replacement is more complex and requires additional time and memory.

Step 2 modification: sampling without randomization: The original approach uses random sampling when choosing rows and columns. We experiment with deterministic sampling that uses the weights to determine the priority. This alternate approach ensures that the highest-weighted rows and columns are always selected, which may potentially improve the accuracy of the result.

Step 3 modification: no compensatory scaling: The original approach scales each of the sampled rows and columns using a multiplier. We experiment with forgoing this step and directly using the unscaled rows and columns for multiplication. This method saves time, and we believe that it may give better accuracy when combined with some of the other modifications listed above.

We experiment with applying the original approach as well as combinations of one or more of the proposed modifications. It is possible that some modifications are more beneficial at particular ranges of sampling rates. It is also possible that the best set of modifications depends on the density of the input matrix, as density is an important factor in determining the weights of the rows and columns. Therefore, we test our approach with various inputs of different density. We compare the performance of our approaches to the GEMM functionality available via CuBLAS [11].

2.2 Convolution as Matrix Multiplication

The convolution operation applies a small filter to an input image. As part of this operation, each pixel in the input “image" has to be multiplied to each value in the filter. It is possible to express this operation as a matrix multiplication. This is done by converting the image into a toeplitz matrix [15]. The converted image can then be multiplied by a column vector containing the filter values. The resulting column vector is the result of the convolution operation.

The process of generating the toeplitz matrix is called “patching" while the process of converting the matrix product into the output of the convolution operation is called “unpatching". We can use the approximate matrix multiplication approach discussed in Section 2.1 to perform a sampled multiplication to save time. This is especially useful if some of the filter values are very small compared to other filter values. The contributions of small filter values can be ignored with minimal loss in precision. Further, we can try the different variations proposed in Section 2.1 to see which variation works best for convolution. The performance of this approach can be compared to the convolution functionality available via CuDNN [12].

2.3 Using AxProf

To test the effects of our approximations on accuracy and performance, we use AxProf [5]. AxProf provides a simple framework for generating inputs, accuracy profiling, and visualization of approximate algorithms. The configuration space for approximate matrix multiplication includes the modifications proposed in Section 2.1, one or more of which may be applied at any time, as well as the sampling rate. AxProf runs the algorithms multiple times on each configuration, for multiple inputs. AxProf then produces a summary of the performance and accuracy statistics for each configuration and input. Finally, AxProf produces visualizations that show how the performance and accuracy is affected by sampling rate for the various algorithms.

3 METHODOLOGY

We implement approximate matrix multiplication on both CPU and GPU. Approximate convolution is then performed via approximate matrix multiplication.

3.1 Approximate Matrix Multiplication

As shown in Figure 1, a critical component of approximate matrix multiplication is the sampling process to generate two smaller matrices. The goal is to choose k' positions out of k positions, where position i represents the i^{th} column of the LHS matrix and the i^{th} row of the RHS matrix.

We implemented our approximate matrix multiplication functions with a similar API as the `TensorGemmCPU` and `TensorGemmGPU` functions present in ApproxHPVM. Our functions take an additional parameter, the *sampling rate*, which is the ratio of k' to k. For the GPU version, we used custom kernels to perform the weighted sampling and scaling operations. We used the CUDA Thrust library [13] to implement these kernels in an efficient manner. Specifically, we use the GPU tree reduction provided by `reduce()` while calculating weights.

While most steps in our method are straightforward, our three different sampling methods are complex. We provide details of the three methods below. For the three methods, given a weight array `weights[1..k]`, they output another array `selected[1..k']`.

3.1.1 Weighted random sampling with replacement. (Algorithm 1) This is the method initially proposed for approximate matrix multiplication [1]. The position with a higher weight has a higher chance to be selected. One position can be selected multiple times. For the CPU version, the algorithm is implemented as plain C++ code. For the GPU version, the entire algorithm is implemented as a custom GPU kernel.

3.1.2 Weighted random sampling without replacement. (Algorithm 2) This algorithm and its analysis are presented in [2]. The position with a higher weight has a higher chance to be selected. However, unlike sampling with replacement, each position can only occur once in the output. For the CPU version, calculating

Algorithm 1: Weighted Random Sampling With Replacement

```
Input  : weights[1..k]
Output : selected[1..k']
for $i \leftarrow 1$ to $k'$ do
    randNum ← random number in [0,1);
    l = 1;
    while randNum > 0 do
        randNum -= weights[l];
        l += 1;
    end
    selected[i] = l - 1;
end
```

`samplingRands` is implemented as plain C++ code, while `sortByKey` is implemented using a priority queue. For the GPU version, calculating `samplingRands` is implemented as a custom kernel, while `sortByKey` is implemented using the Thrust library API function `sort_by_key()`.

Algorithm 2: Weighted Random Sampling Without Replacement

```
Input  : weights[1..k]
Output : selected[1..k']
for $i \leftarrow 1$ to $k$ do
    if weights[i] == 0 then
        samplingRands[i] = MIN-NEG-FLOAT;
    else
        randNum ← random number between (0,1);
        samplingRands[i] = log(randNum) ÷ weights[i];
    end
end
seqeunce = [1,2,3 .. $k$];
sortByKey(toSort=sequence[1..$k$], keys=samplingRands[1..$k$]);
selected[1..$k'$] = sequence[1..$k'$];
```

3.1.3 Weighted deterministic sampling without replacement. (Algorithm 3) This algorithm is a deterministic variation of weighted random sampling without replacement. The k' positions with the highest weights will be selected. For the CPU version, `sortByKey` is implemented using a priority queue. For the GPU version, `sortByKey` is implemented using the Thrust library API `sort_by_key()`.

Algorithm 3: Weighted Deterministic Sampling Without Replacement

```
Input  : weights[1..k]
Output : selected[1..k']
sortByKey(toSort=sequence[1..$k$], keys=weights[1..$k$]);
selected[1..$k'$] = sequence[1..$k'$];
```

3.2 Convolution as Matrix Multiplication

For convolution, the first step is to convert the image and kernel into matrices. Next, the matrix multiplication is performed. Finally, the product is converted to the output of convolution. For conversion to and from the matrix form, we used and modified pre-existing code in the ApproxHPVM repository. We then combined these steps with a call to our approximate matrix multiplication functions.

4 EVALUATION

We setup the ApproxHPVM [6] testing environment, then added our software approximate operations to ApproxHPVM. Then we used AxProf [5] to test approximate matrix multiplication and approximate matrix convolution against the exact versions in ApproxHPVM on both CPU and GPU. We present graphs of several best performing variations and data on all variations of the two operations.

4.1 Evaluation Approach

4.1.1 Testing environment. We ran our experiments on a computer with an Intel i7 CPU @ 1.8GHz with 8 cores, 32 GB RAM, and an NVIDIA GeForce MX150 graphics card. We used CUDA version 10.1, cuDNN version 7.5, and cuBLAS version 10.0.

Dataset	LHS Dimensions	RHS Dimensions	Density
MNIST	10000×784	784×128	19.27%
CIFAR-10	10000×3072	3072×192	100%

Table 1: Statistics for Evaluation Datasets

4.1.2 Testing procedure and dataset. We tested our operations on two datasets: the MNIST dataset [8] and the CIFAR-10 dataset [7]. Table 1 lists the dimensions and density of the datasets. The MNIST dataset is very sparse, while the CIFAR-10 dataset is around 4 times larger and much more dense.

To test the approximate matrix multiplication, we multiply the entire dataset with a weight matrix. This operation is commonly performed in a fully connected layer in neural networks. We then measure the performance gain and accuracy loss against the exact matrix multiplication.

4.1.3 Test metrics. For each test, we measure:

- Total running time
- Sampling overhead vs. matrix multiplication time
- Accuracy of the approximate result, expressed in both ℓ_2 and ℓ_1 error:

$$L_1^e = \frac{L_1(approximate - exact)}{L_1(exact)}$$

$$L_2^e = \frac{L_2(approximate - exact)}{L_2(exact)}$$

4.2 Approximate Matrix Multiplication

Figures 2 and 3 show the relation between the sampling rate on the X axis and ℓ_2 error on the Y axis for the GPU version of our code for the MNIST and CIFAR datasets respectively. We compare three

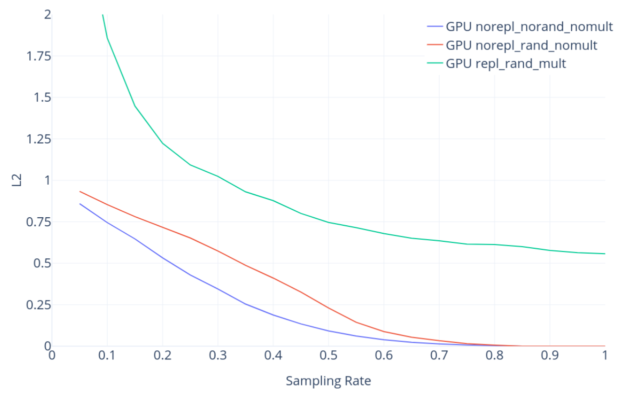

Figure 2: MNIST ApproxMM GPU ℓ_2 Error (Best Variations)

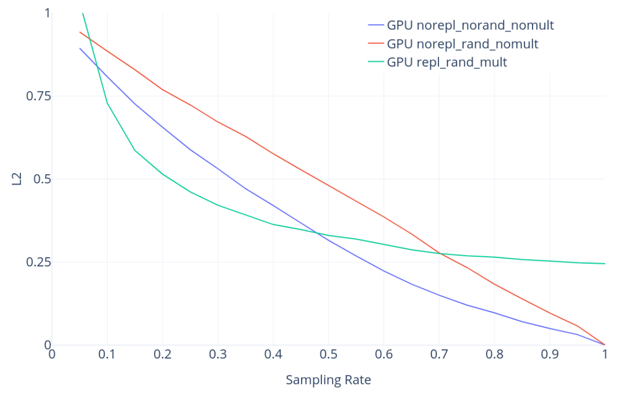

Figure 3: CIFAR ApproxMM GPU ℓ_2 Error (Best Variations)

different variations of the algorithm that showed the best tradeoff between error and runtime:

- *norepl_norand_nomult*: Deterministic sampling without replacement and without using rescaling multiplier
- *norepl_rand_nomult*: Randomized sampling without replacement and without using rescaling multiplier
- *repl_rand_mult*: Randomized sampling with replacement and using rescaling multiplier

As expected, the error decreases monotonically as the sampling rate approaches 1. For the variations that perform non-replacement sampling, the error drops to 0 at sampling rate 1. This is not true for the variation that performs replacement sampling, as not all positions are included in the sampled matrices even at sampling rate 1 due to some positions being repeated. Further, while the results for the non-replacement variations are consistent, the error for the replacement sampling variation is lower for the higher density CIFAR dataset. This shows that using non-replacement sampling is more consistent across datasets of different density.

Figures 4 and 5 show the relation between the sampling rate on the X axis and time on the Y axis for the GPU version of our code

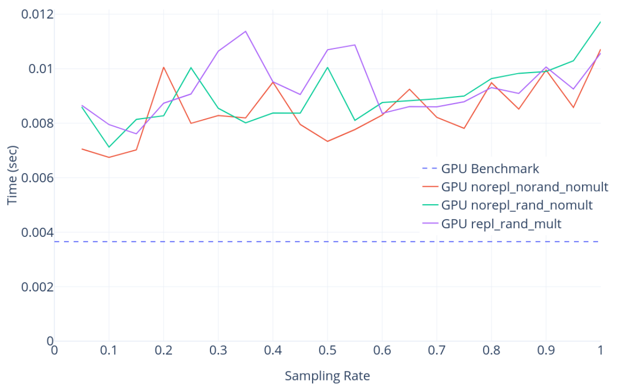

Figure 4: MNIST ApproxMM GPU Time (Best Variations)

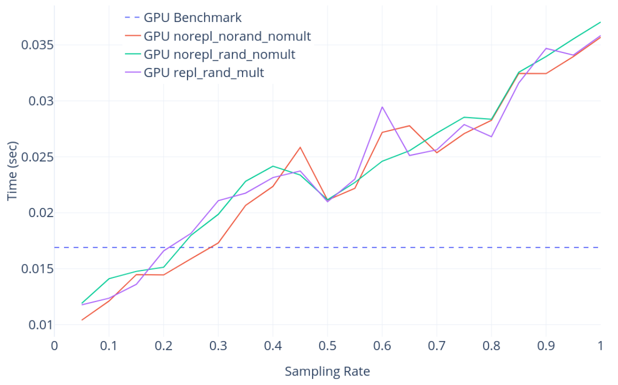

Figure 5: CIFAR ApproxMM GPU Time (Best Variations)

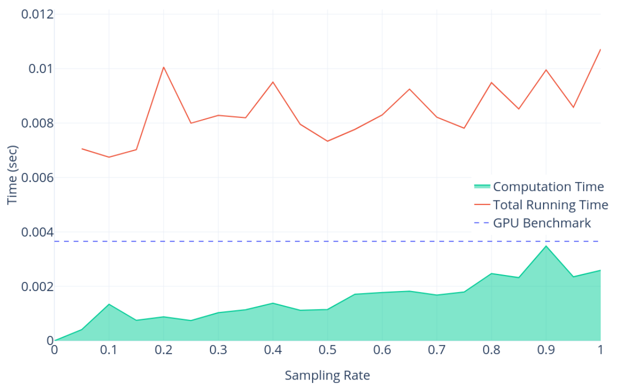

Figure 6: MNIST ApproxMM GPU Sampling Overhead (Best Variation)

for the MNIST dataset. We compare the same three variations as those used in Figure 2. We also include the benchmark time set by

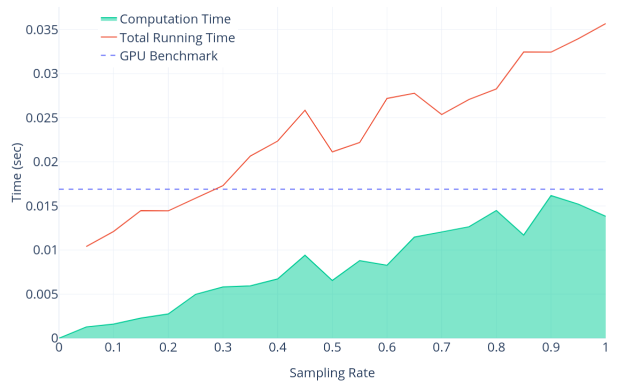

Figure 7: CIFAR ApproxMM GPU Sampling Overhead (Best Variation)

the non-approximate method included in ApproxHPVM. The total runtime grows with sampling rate for the CIFAR dataset, but not for the MNIST dataset. It is mostly consistent across the different variations, and is often higher than the benchmark time. However, the majority of this runtime is overhead required to perform the sampling operation. Figures 6 and 7 show the time overhead. The lower shaded region shows the actual computation time, while the higher line shows the total runtime. If this overhead can be reduced or removed, then our method can be faster than the benchmark.

Variation	Time	ℓ_2 Error	ℓ_1 Error
norepl_norand_mult	10.2	0.31	0.29
norepl_norand_nomult	7.3	0.09	0.06
norepl_rand_mult	9.1	0.42	0.38
norepl_rand_nomult	10.1	0.23	0.22
repl_rand_mult	10.7	0.75	0.74
repl_rand_nomult	7.6	0.88	0.87

Table 2: Statistics for all GPU Variations at 50% Sample Rate for MNIST dataset

Table 2 gives statistics for all the variations we tested for the GPU version of the code at 50% sampling rate for the MNIST dataset. We do not include the results for the CPU results as most matrix multiplication computations are performed on the GPU for neural networks.

5 FUTURE WORK

Our evaluation for approximate convolution was unsuccessful due to errors in the code for conversion between matrix multiplication and convolution. We are currently coordinating with the authors of that code to resolve this issue and add convolution to our evaluation.

For GPU multiplication, our approximate versions were slower than the exact version in most cases. Our versions also used significantly more memory. The reason was that we had to copy data to the sampled matrices from the original matrices. We must do this in order to use cuBLAS for matrix multiplication. GPU data copying is the major contributor to sampling overhead (Figures 6 and 7), and it also incurs up to double memory usage. We believe that the solution to reducing this overhead is to write our own matrix multiplication kernel that directly uses the sampled rows and columns from the original matrix instead of explicitly creating a sampled matrix.

6 RELATED WORK

6.1 Approximate Matrix Multiplication

We base our approach to approximate matrix multiplication to the one proposed by Drineas et al. [1]. While this report proposes the approach and presents theoretical analysis, it does not appear to carry out any evaluation of the approach. We modified this approach in various ways and noticed that our modifications gave better results in practice.

Manne and Pal [10] propose a deterministic approach to approximate matrix multiplication. It reduces the matrix multiplication operation to a set of linear equations and then solves them approximately. By controlling the level of approximation while solving the set of linear equations, the error of the product matrix can be made arbitrarily low. However, this approach relies on complex tenchiques that may be difficult to implement on GPUs, which are used for neural networks today.

Sarlos [14] proposed using random projections for approximate matrix multiplication. Magen and Zouzias [9] proposed another sampling-based approach that improves upon the approaches by Drineas et al. [1] and Sarlos [14].

6.2 Approximate Convolution

Figurnov et al. [3] propose using perforation to speed up the convolution operation. They only perform convolution at some locations of the output and use interpolation to obtain the rest of the output. This operation is performed in an input-agnostic manner. Our approach using sampled matrix multiplication ensures that the sampling is performed so that larger values in the input have greater representation, providing better accuracy.

Some 2D convolution filters are separable i.e. they can be represented as a pair of vertical and horizontal 1D filters. Performing two 1D convolutions is faster than performing a 2D convolution, and is a method for speeding up the computation for separable filters. Similarly, Hearnand and Reichel [4] propose a method for approximate convolution when one of the matrices has a low-rank approximation. The problem with both of these approaches is that it requires some control over the filter being used. In neural networks, however, the filters are automatically created and may not satisfy the requirements necessary for applying these approximations. Our method is not dependent on particular filter properties.

7 CONCLUSION

We implemented approximate matrix multiplication and convolution via sampling. We tested our approaches with various modifications and found which modifications provided the best results in terms of accuracy. We provide our results in an easy to parse format so that it can be used by the ApproxHPVM autotuner to decide when our approximations can be safely applied. The source code of our project is available in https://gitlab.engr.illinois.edu/hsharif3/hpvm/tree/approx_keyur_ziang/llvm/projects/hpvm-tensor-rt

REFERENCES

[1] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. 2006. Fast Monte Carlo Algorithms for Matrices I: Approximating Matrix Multiplication. *SIAM J. Comput.* 36, 1 (July 2006), 132–157. https://doi.org/10.1137/S0097539704442684

[2] Pavlos S. Efraimidis and Paul G. Spirakis. 2006. Weighted random sampling with a reservoir. *Inform. Process. Lett.* 97, 5 (March 2006), 181–185. https://doi.org/10.1016/j.ipl.2005.11.003

[3] Mikhail Figurnov, Aizhan Ibraimova, Dmitry P Vetrov, and Pushmeet Kohli. 2016. Perforatedcnns: Acceleration through elimination of redundant convolutions. In *Advances in Neural Information Processing Systems*. 947–955.

[4] Tristan A Hearn and Lothar Reichel. 2014. Fast computation of convolution operations via low-rank approximation. *Applied Numerical Mathematics* 75 (2014), 136–153.

[5] Keyur Joshi, Vimuth Fernando, and Sasa Misailovic. 2019. Statistical Algorithmic Profiling for Randomized Approximate Programs. In *ICSE*. IEEE, Montréal, Canada.

[6] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. HPVM: Heterogeneous Parallel Virtual Machine. In *PPoPP*. ACM, New York, NY, USA, 68–80.

[7] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report. University of Toronto.

[8] Yann LeCun and Corinna Cortes. 2019. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/.

[9] Avner Magen and Anastasios Zouzias. 2011. Low rank matrix-valued Chernoff bounds and approximate matrix multiplication. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 1422–1436.

[10] Shiva Manne and Manjish Pal. 2014. Fast Approximate Matrix Multiplication by Solving Linear Systems. *arXiv preprint arXiv:1408.4230* (2014).

[11] NVIDIA. 2019. cuBLAS | NVIDIA Developer. https://developer.nvidia.com/cublas. [Online; accessed 11-May-2019].

[12] NVIDIA. 2019. NVIDIA cuDNN | NVIDIA Developer. https://developer.nvidia.com/cudnn. [Online; accessed 11-May-2019].

[13] NVIDIA. 2019. Thrust | NVIDIA Developer. https://docs.nvidia.com/cuda/thrust/index.html. [Online; accessed 11-May-2019].

[14] Tamas Sarlos. 2006. Improved approximation algorithms for large matrices via random projections. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*. IEEE, 143–152.

[15] Wikipedia contributors. 2019. Toeplitz matrix — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Toeplitz_matrix&oldid=892607331. [Online; accessed 11-May-2019].