Supervised Learning Based Algorithm Selection for Deep Neural Networks

Shaohuai Shi*, Pengfei Xu*, Xiaowen Chu*†
*Department of Computer Science, Hong Kong Baptist University, Hong Kong
†HKBU Institute of Research and Continuing Education, Shenzhen, China
{csshshi, pengfeixu, chxw}@comp.hkbu.edu.hk

***Abstract*—Many recent deep learning platforms rely on third-party libraries (such as cuBLAS) to utilize the computing power of modern hardware accelerators (such as GPUs). However, we observe that they may achieve suboptimal performance because the library functions are not used appropriately. In this paper, we target at optimizing the operations of multiplying a matrix with the transpose of another matrix (referred to as NT operation hereafter), which contribute half of the training time of fully connected deep neural networks. Rather than directly calling the library function, we propose a supervised learning based algorithm selection approach named MTNN, which uses a gradient boosted decision tree to select one from two alternative NT implementations intelligently: (1) calling the cuBLAS library function; (2) calling our proposed algorithm TNN that uses an efficient out-of-place matrix transpose. We evaluate the performance of MTNN on two modern GPUs: NVIDIA GTX 1080 and NVIDIA Titan X Pascal. MTNN can achieve 96% of prediction accuracy with very low computational overhead, which results in an average of 54% performance improvement on a range of NT operations. To further evaluate the impact of MTNN on the training process of deep neural networks, we have integrated MTNN into a popular deep learning platform Caffe. Our experimental results show that the revised Caffe can outperform the original one by an average of 28%. Both MTNN and the revised Caffe are open-source.**

***Index Terms*—Linear Algebra; Matrix Multiplication; Transpose; GPU; Deep Neural Networks**

I. Introduction

Deep neural networks have recently achieved great success in varieties of AI applications [1]. The forwarding and backwarding phases in the backpropagation based training process of a deep neural network require two different forms of matrix multiplication $C = A \times B$ and $C = A \times B^T$, where $A \in R^{m \times k}$, $B \in R^{k \times n}$ ($B \in R^{n \times k}$ for the latter form) and $C \in R^{m \times n}$. In this paper, we call the first form NN operation (N means no transpose) and the second form NT operation (T means transpose). The time complexity of schoolbook matrix multiplication is $O(m \times k \times n)$, which makes it very time-consuming for large matrices. Nowadays, there exist many optimized software libraries for matrix operations, including OpenBLAS, Intel MKL, cuBLAS, etc. As GPUs have become mainstream hardware accelerators, the cuBLAS library from NVIDIA becomes a major linear algebra library for many deep learning software tools [2].

Some recent work have been proposed to understand and improve the performance of NN operations on GPUs [3][4]. Considering the complexity of GPU architectures, it is very challenging to design a single algorithm or a single set of kernel configuration that is optimal for all cases; hence auto-tuning method has become an attractive approach to choosing the best algorithms or kernel configurations for GPUs [5][4]. However, the NT operations have not received much attention from the research community. Our previous work show that many state-of-the-art deep learning software tools overlook the importance of NT operations and only achieve suboptimal performance for some deep neural networks [2].

In this paper, we first show that the performance of NT operation by cuBLAS is often much lower than that of NN operation on recent GPUs. We then propose a simple method called TNN which implements the NT operation by carrying out efficient out-of-place matrix transpose first and then performing an NN operation. In general, TNN outperforms cuBLAS for large matrices, but it is not as efficient as cuBLAS for small matrices. In order to achieve the best average performance, we design an algorithm selection method named MTNN, which can intelligently select the appropriate algorithm to carry out the NT operations based on some GPU architecture information and matrix sizes. Notice that the idea of algorithm selection dates back to 1976 [6] and becomes very successful in recent years to choose optimal implementation from a set of algorithms [7][8][9]. To verity the effectiveness of MTNN, we integrate it into a popular real world deep learning platform Caffe [10] which relies on cuBLAS to accelerate its NN and NT operations on GPUs. We evaluate the performance of MTNN and revised Caffe on two modern GPUs: NVIDIA GeForce GTX1080 and Titan X Pascal, and the experimental results show that (1) our MTNN solution achieves up to 54.03% improvement on average over the NT operation of cuBLAS; and (2) the revised Caffe[1] achieves 28% speedup over the original Caffe on the tested GPUs.

The rest of the paper is organized as follows. We present the motivation of this work in Section II, and then introduce the related work in Section III. The TNN method is described in Section IV, followed by our MTNN framework in Section V. Experimental results are presented in Section VI. We conclude the paper and discuss our future work in Section VII.

[1]Our source codes can be found here: https://github.com/hclhkbu/caffe-optimized

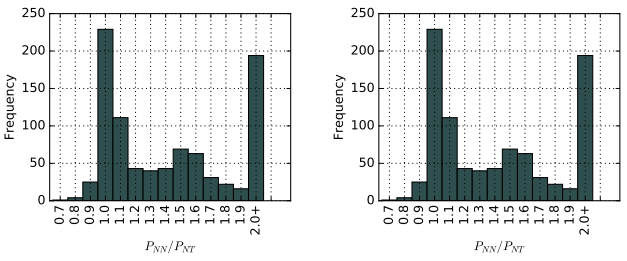
II. MOTIVATION

On deep neural networks, especially the fully connected networks, matrix-matrix multiplication (i.e., NN operations) and matrix-matrix-transpose multiplication (i.e., NT operations) are the two major computational tasks for the training process. Both types of matrix multiplication are commonly implemented by the SGEMM routine of BLAS library in practice. The standard SGEMM has the form: $C = \alpha \cdot op(A) \times op(B) + \beta \cdot C$, where *op* represents whether the matrix is transposed, and α and β are scalars. To simplify the calculation, we ignore the second term (i.e., $\beta = 0$) and set $\alpha = 1$. In cuBLAS, the SGEMM API is "cublasSgemm", in which the second and the third parameters are the values of *op* for *A* and *B* respectively. The value of *op* can be "CUBLAS_OP_T" (transpose) or "CUBLAS_OP_N" (no transpose). To understand the performance difference between NN and NT operations in cuBLAS, we conduct experiments to evaluate the running time performance of SGEMM for NN and NT operations with different sizes of input matrices. The experiments are conducted on two NVIDIA Pascal GPUs: GTX 1080 and GTX Titan X Pascal with CUDA-8.0.

TABLE I. THE EXPERIMENTAL GPU HARDWARE WITH CUDA-8.0

GPU Model	Cores	Memory	OS	Core frequency
GTX1080	2560	8 GB	Ubuntu 14.04	1607 MHz
Titan X	3584	10 GB	Ubuntu 14.04	1417 MHz

We use $P_{algorithm}$ to denote the performance of a specific $algorithm$ with the unit of GFLOPS. To illustrate the difference between P_{NN} and P_{NT}, we run experiments for 1000 cases with different matrix sizes (i.e., both the width and the height of A and B are 2^k where k ranges from 7 to 16) and show the distribution of resulted P_{NN}/P_{NT} in Fig. 1. It is noted that, in most cases, the performance of NN is much better than that of NT because there is no overhead of matrix transpose. The percentages of the number of cases that P_{NN} is higher than P_{NT} are 71% and 62% on GTX1080 and Titan X respectively. More surprisingly, there are around 20% of cases with $P_{NN}/P_{NT} \geq 2.0$ on both GPUs. The low performance of NT of cuBLAS may be caused by the inefficient memory access to the elements of *B*. Another possible reason is that cuBLAS uses the slow in-place matrix transpose algorithm to reduce the memory footprint [11]. Observing this low efficiency issue, we are motivated to propose a method (TNN) for NT operations which finds the transpose of *B* first and then calls NN function of cuBLAS to finish the calculation of $A \times B^T$ on GPUs. The performance of TNN is better than cuBLAS in most cases, but still there exist cases that cuBLAS outperforms our TNN. To this end, we further design an algorithm selection method to select an appropriate algorithm from the set {TNN, NT of cuBLAS} based on a supervised learning algorithm. Notice that TNN requires that the GPU memory is large enough to store the additional B^T. If that is not the case, our framework will simply choose the original NT operations.



(a) GTX1080 (b) TitanX

Fig. 1. The frequency of performance ratio of P_{NN} over P_{NT} among 1000 tested cases on each GPU. The last value (i.e., 2.0+) of x-axis means: $P_{NN}/P_{NT} \geq 2.0$.

III. RELATED WORK

SGEMM algorithm in cuBLAS has been highly optimized on GPUs by kernel optimization [3][12][13][14] and auto-tuning algorithms [15][5]. The information of different levels of GPU memory access latency [16][12] and instruction computation [3] are extracted to help increase the parallelism of GPU kernels, which can achieve excellent performance that is close to the theoretical hardware capacity based on the block-based matrix-matrix multiplication algorithm. Targeting at Fermi GPU of DGEMM (GEMM in double precision), R. Nath et al. [13] propose a double blocking algorithm to reduce the impact of latency in accessing registers and the shared memory, which can achieve up to 58% of the peak performance. Even though there is a well-designed kernel on GPU, the discrepancy among distinct GPUs would require different configurations to obtain best performance. Instead of conducting detailed kernel analysis, auto-tuning methods have been investigated to select the optimal configuration to achieve better performance of the kernel [15][5].

However, little work has been done to evaluate the performance of the NT operations. Since $B^T_{ji} = B_{ij}$, we can perform NT by changing the access of a row to the corresponding column of matrix *B* with SGEMM routine. However, it might cause extra penalty due to uncoalesced global memory access and conflicted shared memory access when fetching the column elements of matrix *B*. The kernel optimization of NT is challenging because its performance depends not only on the GPU architecture, but also on the input matrix size. Therefore, instead of optimizing the kernel algorithm, we first propose a simple approach called TNN as an alternative to SGEMM. We notice that TNN can significantly outperform SGEMM in many cases, but sometimes its performance could be worse than SGEMM. To this end, we formulate an algorithm selection problem in order to select the appropriate algorithm for each NT operation.

Machine learning approaches become useful in choosing more efficient algorithms with high accuracy [7][9][8]. Spillinger et al. [7] exploit SVM model [17] to predict the better implementation of matrix multiplication algorithm at runtime among two implementations of MKL and CARMA on three different CPU platforms, which achieves about 26% per-

formance improvement on average. Beside the SVM model applied to solve algorithm selection problem [7][9], the decision tree classifier is also used to solve the automatic selection of sparse matrix representation on GPUs and it obtains no more than 1.05x average slowdown compared to the existing ideal approach [9]. In this paper, we make use of machine learning techniques to choose the more efficient algorithm between our proposed TNN and the original cuBLAS implementation to improve the performance in calculating $C = A \times B^T$.

IV. TNN: TRANSPOSE BEFORE MULTIPLY

As we already show in Fig. 1, directly calculating $C = A \times B^T$ through cuBLAS API is usually inefficient. We propose a simple TNN method which replaces the one-step NT operation by two-step operation, i.e., transposing *B* first and then making use of NN. The overall performance can be improved if $T_{TNN} = T_{transpose} + T_{NN} < T_{NT}$, where $T_{algorithm}$ is the computation time of $algorithm$. Noted that $T_{transpose}$ includes the time of GPU memory allocation and release.

Matrix transpose is a memory bound operation [18]. There are two very different ways to perform matrix transpose: in-place and out-of-place. The in-place matrix transpose does not require extra memory space, but it is generally factored as a product of disjoint cycles, which makes parallelization so difficult in rectangular matrices [11]. The state-of-the-art implementation of in-place matrix transposition achieves only 51.56 GB/s and 22.74 GB/s on GTX 980 (with a peak memory bandwidth of 224 GB/s) and Telsa K20 (with a peak memory bandwidth of 208 GB/s) respectively [11]. On the contrary, the out-of-place matrix transposition can exploit the GPU shared memory to achieve an efficient utilization of GPU memory bandwidth. In [18], the optimized transpose kernel achieves up to 80% of peak bandwidth on tested GPUs, which is much higher compared to the in-place algorithm.

Algorithm 1 TNN

```
1: procedure TNN(A, B, C, m, n, k)
2:     BT = cudaMemAlloc(n*k*sizeof(float));
3:     transposeOnGPU(B, n, k, BT);
4:     cublasSgemm(..., CUBLAS_OP_N, CUBLAS_OP_N, ...);
5:     cudaFree(BT);
```

We conduct experiments with 1000 cases ($m, n, k = 2^7, 2^8, ..., 2^{16}$) for both TNN and NT methods. Experimental results of NT and TNN are shown in Fig. 2. It is noticed that there are some cases that NT outperforms the TNN method, especially when k is small (e.g., there are up to half of the cases that NT is better than TNN when k is 128 on both GPUs). Among all the tested cases, the maximum speedup of TNN over NT is 4.7x, whilst the maximum speedup of NT over TNN is 15.39x. There is a great portion of cases (about 41.5% on GTX1080 and 43% on TitanX) that are shaded with red rectangles. Therefore, to perform faster calculations of $C = A \times B^T$, we should choose the NT algorithm or the TNN algorithm appropriately.

V. MTNN: A SUPERVISED-LEARNING BASED ALGORITHM SELECTION METHOD

In this section, we first formulate the algorithm selection problem as a classification problem for two given input sizes of matrices and a specific GPU platform. Let the class: -1 denote $P_{TNN} > P_{NT}$ and the class: 1 denote $P_{TNN} \leq P_{NT}$. Given a GPU platform: G, size of matrix $\mathbf{A}$ ($m \times k$) and size of matrix $\mathbf{B}$ ($n \times k$), there exists a function: $f : (G, m, n, k) \mapsto \{-1, 1\}$. We need to learn a function $\hat{f}$ such that: $\hat{f} = argmin \sum_{(G,m,n,k)\in\Omega} ||\hat{f}(G, m, n, k) - f(G, m, n, k)||$. The learning of function $\hat{f}$ can be regarded as a binary classification problem. There are 4 main steps of our supervised-learning based method MTNN. First, we need to construct the training and testing data set with proper preprocessing of data by benchmarking the performance of NT and TNN. Second, we learn a decision model (i.e., $\hat{f}$) from training samples with supervised machine learning algorithms. Third, we evaluate the learned model on the testing data set. Lastly, we apply the trained model to predict the better implementation (i.e., NT or TNN) in calculating $C = A \times B^T$.

A. Data Collection

According to the results of Fig. 1, we choose a range of matrices with sizes in $S = \{2^i | i = 7, 8, ..., 16\}$. For all m, n and k ($m, n, k \in S$), which has 1000 combinations, we test the performance of NT and TNN in calculating $C = A \times B^T$. Let $P_{NT}(m, n, k)$ and $P_{TNN}(m, n, k)$ denote the performance of NT and TNN respectively with two matrices *A* and *B*, where $A \in R^{m\times k}$ and $B \in R^{n\times k}$. The difference between $P_{NT}(m, n, k)$ and $P_{TNN}(m, n, k)$ is denoted by $D(m, n, k)$. If $D(m, n, k) \geq 0$, then $label = 1$; otherwise $label = -1$. Besides the variety of input size of matrices, the GPU platform can also be different. Thus, we extract the features to represent different GPUs. Combined with different values of the characteristics of GPU, the input sample **x** has 8 dimensions: 5 from GPU specification and 3 from matrix size. The first 5 dimensions are the size of GPU global memory (gm), the number of SMs (sm), core clock (cc), memory bus width (mbw) and the size of L2 cache ($l2c$). Note that the feature generation is an $O(1)$ computation, which is crucial to reduce the overhead of the predictor in runtime. The format of input sample **x** is: $(gm, sm, cc, mbw, l2c, m, n, k)$, $label$. For each type of GPU, 1000 cases are tested; but some samples that cannot be fitted into memory are not included into evaluation. So the number of valid samples on each GPU is less than 1000 (i.e., 891 on GTX1080 and 941 on TitanX). We do not need to normalize the input feature by using decision tree. By contrast, each dimension of the input feature should be normalized to the range of (0, 1) when training SVMs.

B. Model Training

Given the training set: $\mathbf{S} = \{\mathbf{x} | \mathbf{x} = (G, m, n, k)\}$, where G represents GPU characteristics, the classifier $\hat{f}$ is learned. If $\hat{f}(\mathbf{x}) = -1$, then we choose TNN; otherwise we choose NT.

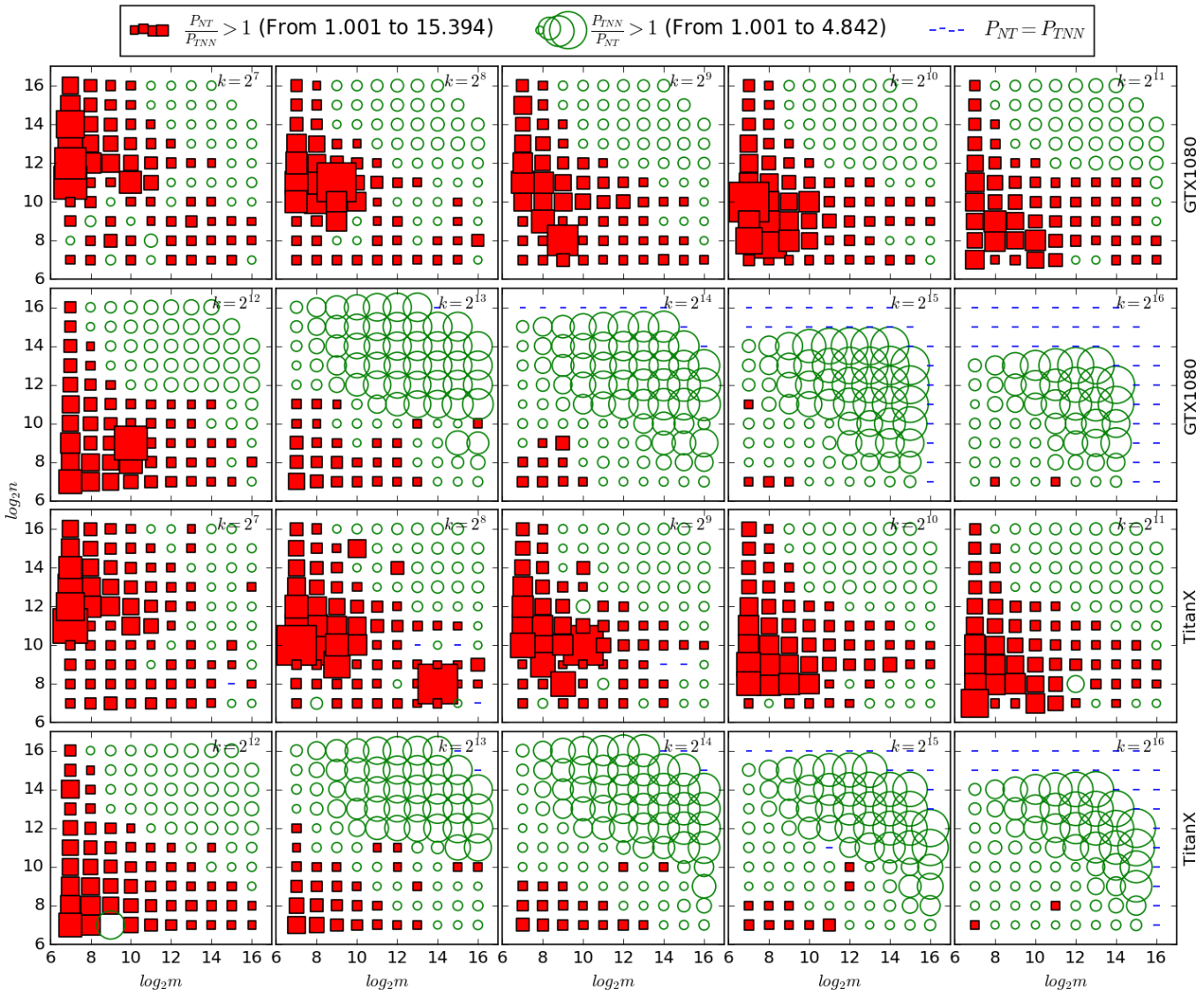
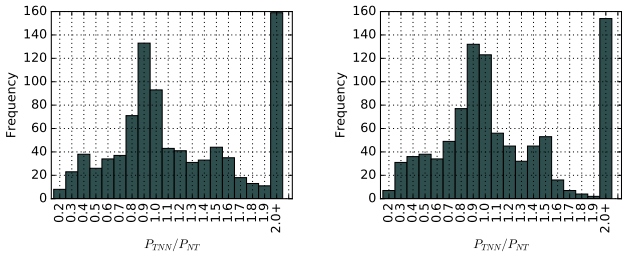


Fig. 2. The performance comparison between NT by cuBLAS and TNN in calculating $C = A \times B^T$. The red rectangle symbol in the legend indicates that the performance of NT is better than TNN; the green cycle symbol indicates that the performance of NT is worse than TNN; and the blue dash symbol indicates that the performances of NT and TNN are equal. The size of the rectangle and cycle symbols reflects the value of P_{NT}/P_{TNN} and P_{TNN}/P_{NT} respectively: larger symbol size indicates higher ratio value. The top two rows are for GTX1080, and the bottom two rows are for Titan X.



(a) GTX1080 (b) TitanX

Fig. 3. The frequency of performance ratio of P_{TNN} over P_{NT} among 1000 tested cases on each GPU. The last value (i.e., 2.0+) of x-axis means: P_{TNN}/P_{NT} is greater than or equal to that value

Learning Algorithm. SVM is a powerful learning algorithm in solving classification problems. And it is successfully applied to solve algorithm selection problems related to matrix-matrix multiplication [7][8]. Another powerful learning algorithm: decision tree (DT) is also prosperously used in solving the problem of automatic best algorithm selection [9], and there is an extended algorithm of decision tree named gradient boosted decision tree (GBDT) [19]. A flexible framework XGBoost [20] of GBDT implementation is used in this paper. We choose GBDT as our learning algorithm for three main reasons: 1) It does not require the input feature normalization since the decision tree is a recursive partitioning based algorithm, which reduces the overhead of the feature preprocess in runtime. 2) Among 10 popular supervised

learning algorithms, boosted decision tree outperforms other algorithms, including SVM and traditional decision tree on many tested dataset [21]. 3) The prediction time complexity is acceptable, say $O(h)$, where h is the depth of the trained decision tree and can be restricted to a fixed value.

Parameter Configuration. We need to consider two main impacts when setting the parameters. On one hand, it is crucial that the depth of the decision tree should not be too deep; otherwise it will increase the overhead of the predictor in runtime. On the other hand, we need to set the proper parameters such that the prediction accuracy is high enough. In this paper, we set the maximum depth of the decision tree to be 8 and the number of estimators for boosting is also 8. We set step size shrinkage (eta) to be 1, and the minimum loss reduction ($gamma$) to 0, which make the boosting algorithm more progressive.

Training. Instead of training model separately for different GPUs, we hope that the model is robust to different GPU hardware, so we put all the input features (8-dimension vector, including 5 characteristics of GPU) into one model. We randomly split the dataset into training data set (80%) and testing data set (20%). Note that in the 80% training data set, there include 80% samples from each GPU, and the remainder is used as testing data set. To validate whether the chosen model can generalize our dataset or not, 5-fold cross-validation is presented in this work. After the evaluation of cross-validation, the whole data set is used as training data to learn the final model that can be put into real-world applications.

Integration. We use the learned model as our predictor of the selection system to choose the better algorithm between NT and TNN. After the model has been well trained, the final algorithm in calculating $C = A \times B^T$ is derived, and we call it MTNN which is shown in Algorithm 2.

Migration to Other Platforms. The MTNN method can be migrated to other GPU platforms by repeating the model training progress, which will generate the new selection model on the target platform to predict the better algorithm in calculating the matrix-matrix-transpose multiply.

Algorithm 2 MTNN

```
1: procedure MTNN(A, B, C, m, n, k, g)
2:     int label = predictor(g, m, n, k);
3:     if (label == 1) then
4:         cublasSgemm(..., CUBLAS_OP_N, CUBLAS_OP_T, ...);
5:     else
6:         TNN(A, B, C, m, n, k);
```

VI. Evaluation

We first evaluate the accuracy of the predictor, and then we present the overall performance improvement with the trained predictor (i.e., the performance of MTNN).

A. Performance of Classification

To evaluate the performance of the classification algorithm, we use the metric of classification accuracy to measure the classifiers. The average accuracy of our pre-defined 5-fold cross-validation is 90.51%, which means that the predictor makes the calculation of $C = A \times B^T$ faster in 90.51% cases. Since the testing data set is an imbalanced set with a larger number of negative samples than positive samples, both accuracies of the negative and the positive classes are recorded. Table II shows the details of the accuracy of the 5-fold cross-validation.

TABLE II. Accuracies of the 5-fold cross-validation

Class	Minimum	Maximum	Average
Negative	91.36%	93.30%	92.05%
Positive	86.49%	92.31%	88.39%
Total	89.40%	91.94%	90.51%

We also make a comparison with SVM algorithms, including axial basis function kernel (SVM-RBF) and polynomial kernel (SVM-Poly), both of which are commonly used in supervised machine learning. We use libSVM [22] as SVM implementation. The parameters for SVM are: $C = 1000.0$ and $gamma = 0.01$, and the input feature is normalized to the range of (0, 1). The learning algorithm of traditional decision tree (DT) is also included into the comparison to show GBDT has a better performance in terms of accuracy and running efficiency. In the tested experimental environment (Table III) for learning algorithms, the performances of classifiers are shown in Table IV. From Table IV, in terms of prediction accuracy, GBDT is much better than both SVM and DT. Regarding the training and prediction efficiency, GBDT outperforms both types of SVMs. Even though the prediction time of GBDT is slightly longer than that of DT, it could be neglectable (only 0.005 ms) compared with the overhead of matrix-matrix-transpose multiplication.

TABLE III. The experimental environment for classifiers

CPU	Memory	OS	Frequency
Intel CPU i7-3820	64 GB	Ubuntu 14.04	3.6 GHz

TABLE IV. Comparison with SVM and DT

Classifier	Accuracy (%)	Train Time (ms)	Predict Time (ms)
GBDT	**90.51**	7	0.005
SVM-RBF	81.66	47	1.2
SVM-Poly	77.68	30	1.07
DT	87.84	**1**	**0.004**

B. Performance of Selection

In this section, we show how much performance can be improved by MTNN, which is integrated with the trained predictor. In MTNN, the integrated predictor is trained with all the data set to achieve higher performance instead of just using 80% data for training because the more data the higher accuracy in general. With 100% data as training set, the trained predictor with GBDT achieves 96.39% accuracy in classification, which means the selection system makes the correct decision to choose the better algorithm between NT and TNN in 96.39% cases.

Before presenting the statistic results of MTNN compared to NT and TNN, a visualized comparison between MTNN and

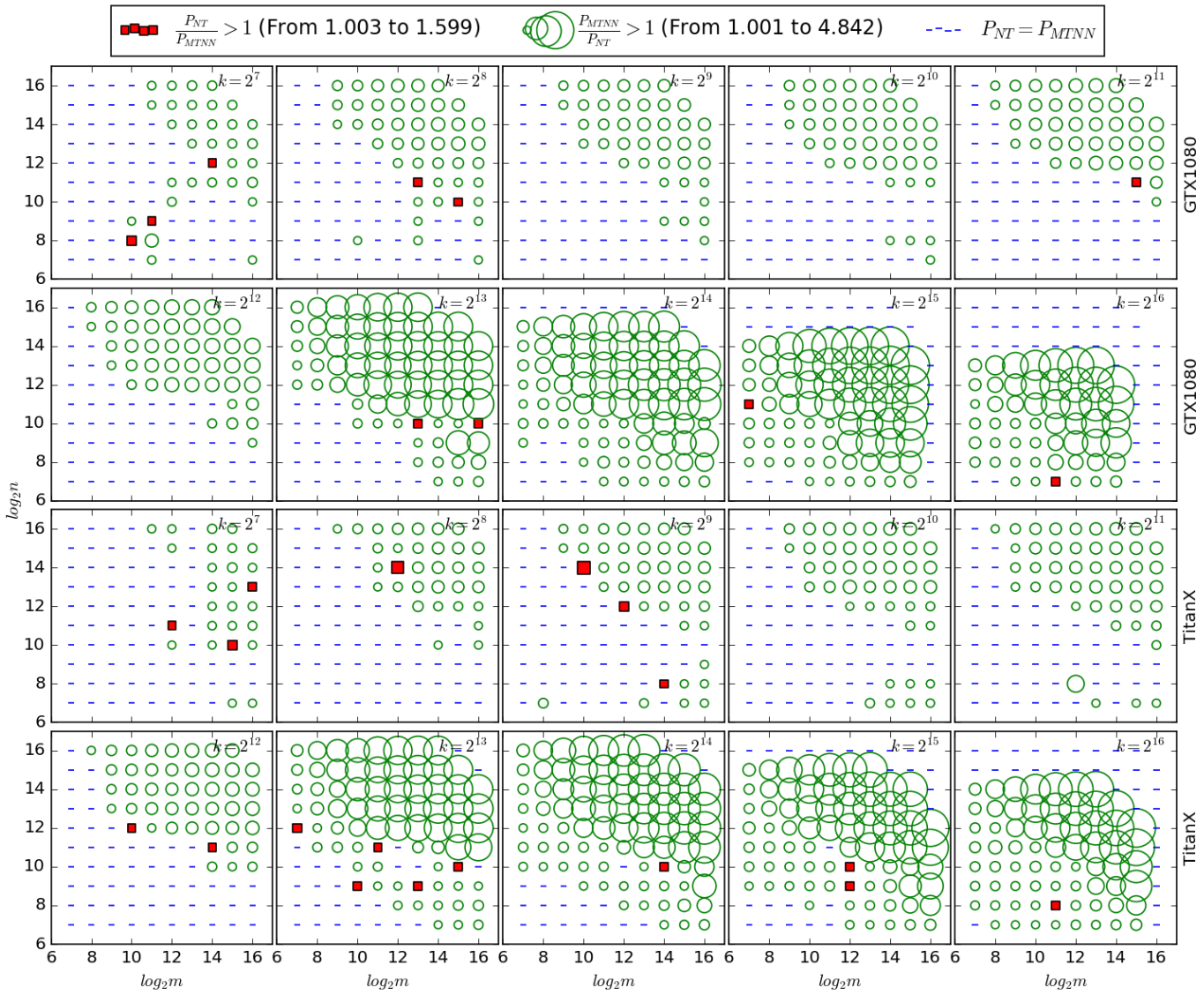
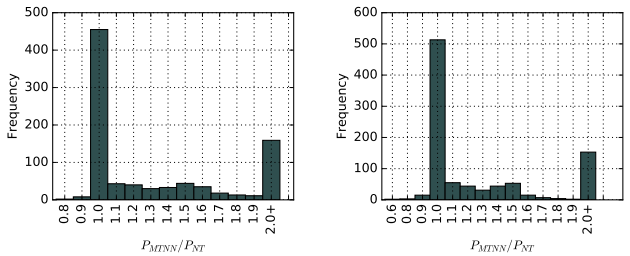


Fig. 4. The performance comparison between NT and MTNN method in calculating $C = A \times B^T$. The rectangle symbol in the legend indicates that the performance of NT is better than MTNN, and the cycle symbol green color indicates that the performance of NT is worse than MTNN, and the dash symbol with blue color indicates that the performances of NT and MTNN are equal.

NT on our tested GPUs is shown in Fig. 4. Compared to Fig. 2, the red rectangles, which indicate that the performance of TNN is worse than NT, are reduced to a very small portion by the MTNN method. In other words, in most cases, the performance of MTNN is better than or equal to NT; and only in a minority of cases, the performance of MTNN is worse than NT. The statistic frequency on the performance of MTNN over NT is shown in Fig. 5, which shows that there is only a small portion of cases that NT outperforms MTNN on both GPUs. In Fig. 2, the maximum value of P_{NT}/P_{TNN} is 15.394, while Fig. 4 displays that the maximum of P_{NT}/P_{MTNN} is only about 1.6.

Similar to the work in [7] and to make further comparisons in a statistical way, we use GOW (Gain over Worst) to denote **G**ain in performance of MTNN **O**ver the **W**orst algorithm at



(a) GTX1080 (b) TitanX

Fig. 5. The frequency of performance ratio of P_{MTNN} over P_{NT} among tested cases on each GPU. The last value (i.e., 2.0+) of x-axis means: P_{MTNN}/P_{NT} is greater than or equal to that value.

each sample. GOW is calculated by:

$$GOW = \frac{P_{MTNN} - min(P_{NT}, P_{TNN})}{min(P_{NT}, P_{TNN})} \quad (1)$$

Let LUB (Loss under Best) denote the percent **L**oss of MTNN **U**nder the **B**est algorithm for each sample, which is calculated by:

$$LUB = \frac{P_{MTNN} - max(P_{NT}, P_{TNN})}{max(P_{NT}, P_{TNN})} \quad (2)$$

We can define some metrics to measure the performance of MTNN compared to NT and TNN. The description of metrics is displayed in Table V. And the corresponding evaluated values are shown in Table VI.

TABLE V. METRICS DESCRIPTION

Metric	Description
MTNN vs NT	Average percent improvement of using MTNN versus versus always choosing TN
MTNN vs TNN	Average percent improvement of using MTNN versus versus always choosing TNN
GOW_{avg}	Average GOW in all samples
GOW_{max}	Maximum GOW in all samples
LUB_{avg}	Average LUB in all samples
LUB_{min}	Maximum LUB in all samples

TABLE VI. VALUES OF PERFORMANCE METRICS OF MTNN IN %

Metric	GTX1080	TitanX	Total
MTNN vs NT	57.78	50.48	54.03
MTNN vs TNN	21.51	22.31	21.92
GOW_{avg}	79.44	73.20	76.23
GOW_{max}	1439.39	957.44	1439.39
LUB_{avg}	-0.15	-0.40	-0.28
LUB_{min}	-25.07	-71.62	-71.62

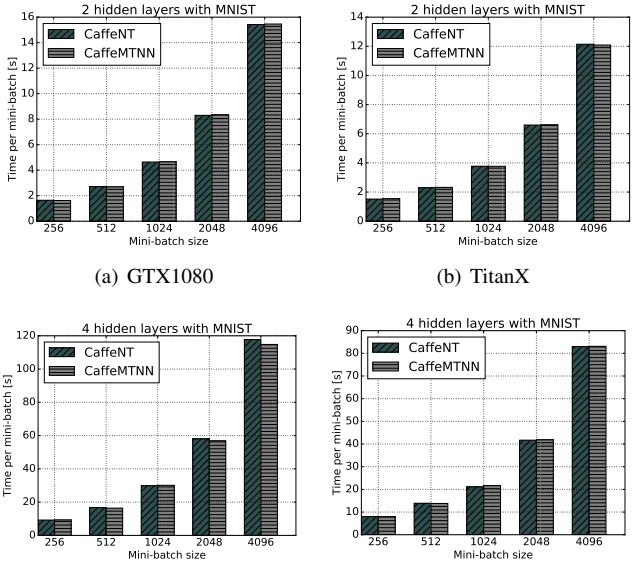
From Table VI, we can see MTNN achieves 54.03% performance improvement compared to using the NT algorithm only, and 21.92% compared to TNN on average. Compared to the worst cases of NT and TNN, MTNN achieves up to 76.23% performance improvement on average and up to 1439.39% in some particular cases. There are some cases that the predictor makes the wrong decision, but the performance slowdown is only about 0.28%. In other words, compared to the best cases of NT and TNN, the performance of MTNN is only 0.28% worse when the predictor makes a wrong choice. Between these two GPUs, the speedup of time efficiency on GTX1080 is slightly higher than that on TitanX.

C. Evaluation with Caffe

TABLE VII. CONFIGURATION OF FULLY CONNECTED NETWORKS

Data set	MNIST	Synthetic
Input	784	26752
Output	10	26752
2 hidden layers	2048-1024	4096-4096
4 hidden layers	2048-2048-2048-1024	4096-4096-4096-4096

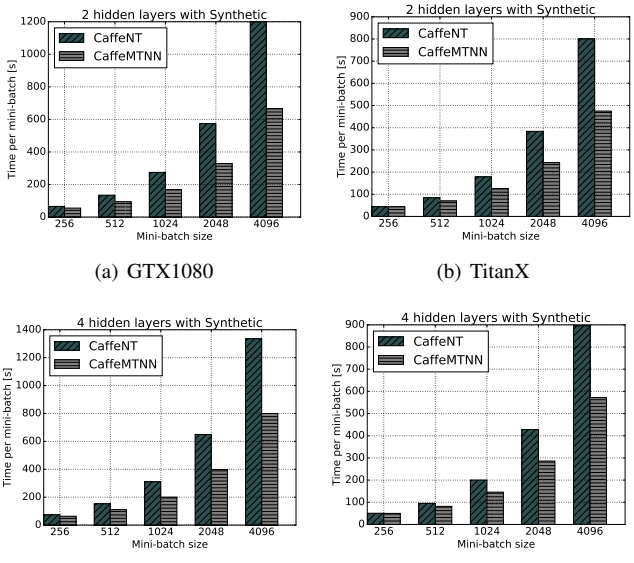
To test the performance of MTNN in the real-world application, we integrate the MTNN algorithm into Caffe [10]. Two types of fully connected networks are used: one is with the MNIST data set whose input and output dimensions are small, and the other is with a synthetic data whose input and output dimensions are large. For each type of fully connected network, 2 and 4 hidden layers are configured. The details of network configurations are shown in Table VII. The performance comparison of these two types of networks running on the original version of Caffe (CaffeNT) and Caffe with MTNN (CaffeMTNN), are displayed in Fig. 6 and Fig. 7, respectively.



(a) GTX1080 (b) TitanX

(c) GTX1080 (d) TitanX

Fig. 6. The performance comparison with MNIST between CaffeNT and CaffeMTNN



(a) GTX1080 (b) TitanX

(c) GTX1080 (d) TitanX

Fig. 7. The performance comparison on larger FCN between CaffeNT and CaffeMTNN.

By integrating our method to Caffe, the performance of the optimized Caffe accomplishes an overall improvement of 1.74% with MNIST data set, while the performance improvement is 28.2% with synthetic data set.

On one hand, from Fig. 6, it is noted that the training time speed of CaffeNT and CaffeMTNN is very close when the mini-batch size is not greater than 4096. The main reason of these phenomena is that with specific number of neurons

in two adjacent layers (e.g., $l1$ and $l2$) and the mini-batch size (mb), the size of matrix-matrix-transpose multiplication is decided by $l1$, $l2$ and mb. If the values of $l1$, $l2$ and mb are too small, the performance of TNN has no advantage compared to the original NT of cuBLAS, which can be explained with the performance comparison in Fig. 4 (there are many dash symbols on the left-bottom side of the figure, so MTNN can only be on the par with NT of cuBLAS). There exists a particular case that MTNN is slightly worse than NT of cuBLAS. The reason of this minor slowdown is that the predictor makes the wrong prediction, but it may occur in a very small probability since the accuracy of the predictor is up to 96%. On the other hand, from Fig. 7, with the larger neural network (the input size and the output size are both 27652 in our tested case) and the larger mini-batch size (larger than 512), the speedup of CaffeMTNN is significant. And the matrix-matrix-transpose multiplication can be mapped to the cases in the right-top side of Fig. 4 where it has numerous green cycles, which means the deep neural networks can benefit from the higher performance algorithm of MTNN.

VII. Conclusion and Future Work

In this paper, we first illustrate the performance issue of cuBLAS in calculating the matrix-matrix-transpose multiplication compared to the matrix-matrix multiplication by benchmarking a variety of cases. Our experiments show the universality of the limitation of cuBLAS on Pascal GPUs. To avoid using the low performance NT operation of cuBLAS, we propose a simple solution (named TNN) which carrys out the efficient out-of-place tranpose algorithm first and then makes use of matrix-matrix multiplication algorithm. TNN can outperform cuBLAS in many cases, but sometimes it is even worse. To the end, we formulate an algorithm selection problem (or classification problem) which is solved by using machine learning approaches. Using the boost gradient decision tree algorithm, we design the MTNN algorithm to work out the matrix-matrix-transpose multiplication. Our MTNN method achieves 54.03% performance improvement compared to cuBLAS on Pascal GPUs. Last, the MTNN method is applied to a deep learning toolkit: Caffe, and the optimized Caffe achieves some speedup on fully connected networks, and it achieves about 28.2% speedup on average with two GPU cards of NVIDIA GTX 1080 and Titan X Pascal.

The transpose algorithm we used is an out-of-place method, which results in double memory footprint and it cannot run normally if there is no enough memory. Therefore, we plan to exploit in-place matrix transpose algorithm and to find a good trade-off between memory overhead and throughput.

VIII. Acknowledgements

We would like to thank all the reviewers for their insightful comments and valuable suggestions. This work is supported by Shenzhen Basic Research Grant SCI-2015-SZTIC-002. We also gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan X Pascal GPU used for this research.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," *arXiv preprint arXiv:1608.07249*, 2016.

[3] J. Lai and A. Seznec, "Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs," in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*. IEEE, 2013, pp. 1–10.

[4] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, "Understanding the GPU microarchitecture to achieve bare-metal performance tuning," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2017, pp. 31–43.

[5] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, design, and autotuning of batched GEMM for GPUs," in *International Conference on High Performance Computing*. Springer, 2016, pp. 21–38.

[6] J. R. Rice, "The algorithm selection problem," *Advances in computers*, vol. 15, pp. 65–118, 1976.

[7] O. Spillinger, D. Eliahu, A. Fox, and J. Demmel, "Matrix multiplication algorithm selection with support vector machines," *EECS Department, University of California, Berkeley*, 2015.

[8] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse matrix format selection with multiclass SVM for SpMV on GPU," in *Parallel Processing (ICPP), 2016 45th International Conference on*. IEEE, 2016, pp. 496–505.

[9] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on GPUs," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 99–108.

[10] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[11] J. Gomez-Luna, I.-J. Sung, L.-W. Chang, J. M. González-Linares, N. Guil, and W.-M. W. Hwu, "In-place matrix transposition on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 776–788, 2016.

[12] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*. IEEE, 2008, pp. 1–11.

[13] R. Nath, S. Tomov, and J. Dongarra, "An improved MAGMA GEMM for Fermi graphics processing units," *The International Journal of High Performance Computing Applications*, vol. 24, no. 4, pp. 511–515, 2010.

[14] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, "Fast implementation of DGEMM on Fermi GPU," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 35.

[15] Y. Li, J. Dongarra, and S. Tomov, "A note on auto-tuning GEMM for GPUs," in *International Conference on Computational Science*. Springer, 2009, pp. 884–892.

[16] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, Jan 2017.

[17] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[18] G. Ruetsch and P. Micikevicius, "Optimizing matrix transpose in CUDA," *Nvidia CUDA SDK Application Note*, vol. 18, 2009.

[19] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[20] T. Chen and C. Guestrin, "Xgboost: Reliable large-scale tree boosting system," in *Proceedings of the 22nd SIGKDD Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA*, 2016, pp. 13–17.

[21] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 161–168.

[22] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.