Supplementary Document ApproxTrain: Creation of Custom TF ops with Approximate Multipliers

Jing Gong*, Hassaan Saadat*, Hasindu Gamaarachchi‡*, Haris Javaid§,
Xiaobo Sharon Hu† and Sri Parameswaran*
*School of Computer Science and Engineering, UNSW Sydney, Kensington NSW 2052 Australia
‡Garvan Institute of Medical Research, Darlinghurst NSW 2010 Australia §AMD, Singapore
†Department of Computer Science and Engineering University of Notre Dame, Notre Dame, IN 46556 USA

***Abstract*—In ApproxTrain, we create custom TF *ops* to support different types of DNN layers with different approximate multipliers. In these custom TF *ops* of ApproxTrain, all multiplications (both in forward and backpropagation) are replaced by approximate multiplications. To equip our custom *ops* with AMSim, we developed GPU-accelerated custom CUDA kernels for the implementation of our custom TF *ops*. These custom CUDA kernels are needed because the standard *ops* available in the open-source TF library use closed-source cuDNN/cuBLAS libraries in the backend that cannot be modified.**

Currently, ApproxTrain contains four operators that cover a large portion of DNN architectures: (1) AMCONV2D (approximate Conv2D *op*); (2) AMDENSE (approximate Dense *op*); (3) AMMATMUL (approximate matrix multiplication *op*); and (4) AMMHA (approximate Multi-Head Attention *op*). These four operators enable the support of four types of layers/operators: Dense layer, the MatMul *op*, the Conv2D layer and Multi-Head Attention layer.

In this document, we present a detailed description of our approach to create two custom TF *ops* and the underlying CUDA kernels.

I. Custom TF op Creation Overview

The overview of the internals of creation and compilation of our custom *ops* in ApproxTrain is depicted in Figure 1. The main component is the approximate operator C++ class inside the blue dashed box which has multiple operations such as input validation, serializing tensors to linear arrays, memory allocation, and performing computations. The computational part of approximate operator C++ class includes functions to calculate feedforward propagation and back propagation by invoking our custom CUDA kernels or CPU kernels[1]. Custom CUDA kernels (explained in Section VI) are responsible for linear algebra operations and data rearrangement and are equipped to use AMSim. The AMSim is implemented as a device function for running on GPU. As stated before, custom CUDA kernels are written from scratch because the closed-source cuDNN and cuBLAS libraries cannot be modified to use approximate multipliers.

All CUDA kernels are compiled by NVCC, and the C++ operator class is compiled with g++. Then, the compiled

[1]The CPU implementation was used for validating our GPU implementation and benchmark, but could also be used by a user who does not have GPU access at the cost of higher run-time.

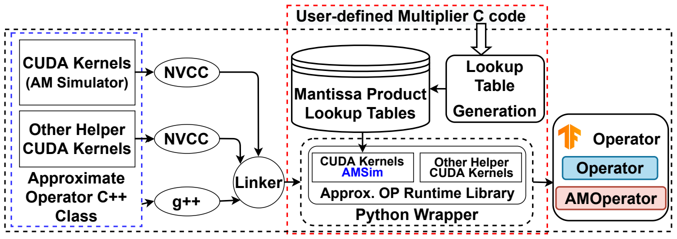
User-defined Multiplier C code
CUDA Kernels (AM Simulator)
NVCC
Other Helper CUDA Kernels
NVCC
Approximate Operator C++ Class
g++
Linker
Mantissa Product Lookup Tables
Lookup Table Generation
CUDA Kernels AMSim
Other Helper CUDA Kernels
Approx. OP Runtime Library
Python Wrapper
Operator
Operator
AMOperator

Fig. 1: Overview of creation of custom ops for ApproxTrain.

C++ object files are linked with the complied CUDA kernel objects to form the approximate operator shared library. This approximate operator run-time library is then enclosed in a python wrapper which is then registered into the standard TF library. Note that the compilation steps above only need to be done once. Instead of replacing the corresponding original operators in TensorFlow, the new approximate operators are kept alongside the original ones. Given user-defined approximate multiplier C codes, LUTs can be obtained by *Lookup Table Generation* (explained in I), as depicted in Figure 1. The obtained LUTs are loaded into the approximate operator run-time library during run-time to simulate different functional models of approximate multipliers. These python wrappers have the same parameters as original operators, and the users simply need to change the name of the original operators to the approximate ones to simulate the approximate multipliers, as demonstrated in the code listings above.

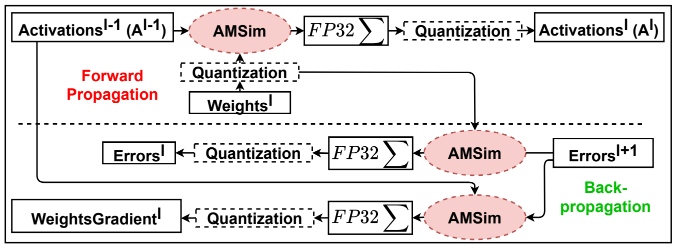

Fig. 2: DNN forward and backpropagation using approximate multipliers.

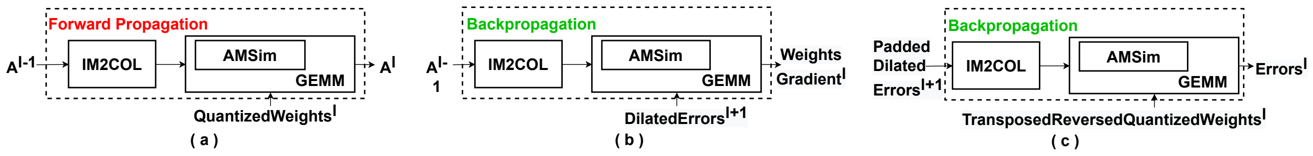

Fig. 3: AMCONV2D Forward propagation and back-propagation implementation overview.

II. Conv. Layer Custom Op with Approximation (AMCONV2D)

This section explains the approach used to realize the forward and backward propagation in the custom approximate operator AMCONV2D.

A. Forward propagation

Forward propagation takes activations A^{l-1} and $QuantizedWeights^l$ to compute output activations A^l (Figure 2). We use the IM2COL+GEMM approach [2], [4] to perform forward propagation, because this approach exposes fine-grained parallelism suitable for GPU acceleration. Figure 3 (a) illustrates this approach. In Figure 3 (a) A^{l-1} is the input to IM2COL operation. The output of IM2COL and $QuantizedWeights^l$ are subjected to the GEMM operation. The GEMM kernel contains AMSim that can invoke native hardware multiplications (* operator) or perform approximate multiplications using LUTs. Algorithm 1 describes our approach for forward propagation on GPU. First, the sizes of GPU global memory arrays are computed (line 2 in Algorithm 1) and allocated (line 3). Then, the IM2COL Kernel is invoked on the GPU (line 4 in Algorithm 1), followed by the GEMM kernel (line 5). Details of these GPU kernels will be discussed later in Section VI.

Despite not being shown in Algorithm 1 for simplicity, a loop that invokes the kernels iteratively on tiles of the array A^{l-1} is implemented to enable our framework to train large architectures and datasets. This is because the CUDA grid (group of blocks of CUDA threads) dimension along the y-axis is limited to 65535 [1], and thus large input data cannot be fit into the GPU grid entirely.

B. Backpropagation

The backpropagation involves two gradient computations: weights gradient and gradient for the preceding layer (labeled as $Errors^l$ in Figure 2). Algorithm 2 elaborates our backpropagation approach for efficient GPU implementation. Similar to Algorithm 1, here too, we first calculate the GPU array sizes and allocate them in lines 2-3 of Algorithm 2. Lines 4-5 are the invocation of the kernels for the weights gradient (explained below), and lines 6-8 are for the preceding layer gradient (explained below). Note that for backpropagation, we also implemented tiling (as we have explained for forward propagation) despite not being shown in Algorithm 2.

1) Weights Gradient Computation: We restructured the weights gradient computation to exploit the IM2COL+GEMM approach as illustrated in Figure 3 (b). We first subject $Errors^{l+1}$ to dilation (inserting zeros between elements based on the *stride* parameter). Then, this $DilatedErrors^{l+1}$ is fed to GEMM (Figure 3 (b)).

As opposed to forward propagation, mapping backpropagation to GEMM along with IM2COL to efficiently exploit GPU architecture is challenging. A naive method to implement the mapping of computation of weights gradient to GEMM would be to implement a separate GPU kernel to perform the dilation operation and invoke it before the GEMM kernel. However, this naive method would be inefficient due to two reasons. First, invoking a kernel unnecessarily adds extra performance overhead. Second, a dilated array would require several times the memory as the original array (depending on *stride* value), consequently reducing the number of non-zero elements that can be stored in the GPU global memory, thus requiring more tiling (similar to tiling explained in forward propagation above). Instead of such a native approach, we implicitly perform this dilation inside the IM2COL_Weight_Kernel (a modified IM2COL kernel) by skipping elements in A^{l-1} that correspond to zero (line 4 of Algorithm 2) if the $Error^{l+1}$

Algorithm 1 Approximate Forward propagation

```
input: $A^{l-1}$, $W^l$, $S$, $P$, $LUT$   ▷ $A^{l-1}$: activation from layer $l-1$; $W^l$: weight from layer $l$; $S$: stride; $P$: Padding; $LUT$: mantissa product LUT
output: $A^l$   ▷ $A^l$: activation from layer $l$
1: function Approximate Forward Propagation
2:     PSize,ColSize ← calculate_sizes($A^{l-1}$,$W^l$,$P$,$S$) ▷ $PSize$: The size of padding $ColSize$: The size of Im2Col results
3:     allocate_GPU_memory(PSize,ColSize,LUT)   ▷ For $A^{l-1}$, Columns(output of IM2COL), $W^l$, $a^l$ and $LUT$.
4:     Columns ← IM2COL_kernel($A^{l-1}$, PSize, ColSize)
5:     $A^l$ ← GEMM_kernel(Columns, $W^l$, LUT ...)   ▷ ... refer to m,n,k,lda,ldb and ldc that are ommited for simplicity. $A^l$ is the output activation
6: end function
```

Algorithm 2 Approximate Backpropagation

```
input: $a^{l-1}$, $W^l$, $Error^{l+1}$, $Stride$   ▷ $a^{l-1}$ is the activation from layer $l-1$),$W^l$ is the weight from layer $l$
output: $W^{l\prime}$, $Errors^l$   ▷ $W^{l\prime}$ is the gradient of $W^l$
1: function Approximate Backpropagation
2:     PSize, ColSize ← calcualte_sizes($a^{l-1}$, $W^l$, $Errors^{l+1}$)▷ $PSize$: The size of padding $ColSize$: The size of Im2Col results
3:     allocate_GPU_memory(PSize, ColSize, LUT)  ▷ For $a^{l-1}$, Columns, $DilatedError^{l+1}$, $W^{l\prime}$ and $Errors^l$
4:     $Columns_{a^{l-1}}$ ← IM2COL_Weight_Kernel($a^{l-1}$, ColSize, PSize)
5:     $W^{l\prime}$ ← GEMM_Kernel($Columns_{a^{l-1}}$, $Error^{l+1}$, LUT ...)
6:     $Columns_{PDError^{l+1}}$ ← IM2COL_PLG_Kernel($Error^{l+1}$, ColSize, PSize)   ▷ IM2COL kernel for preceding layer gradient (PLG), $PDError^{l+1}$ is the $PaddedDilatedError^{l+1}$
7:     $(W^l)^T_r$ ← Reverse_Transpose_kernel($W^l$)
8:     $Errors^l$ ← GEMM_Kernel($Columns_{PDError^{l+1}}$, $(W^l)^T_r$, LUT...)
9: end function
```

array was dilated.

2) Preceding Layer Gradient: We also restructure the computation of the preceding layer gradient to exploit the IM2COL+GEMM approach as shown in Figure 3 (c). For this, we first subject $Errors^{l+1}$ to dilation (inserting zeros between elements as explained before), followed by padding (inserting zeros around the image along height and width dimensions). This $PaddedDilatedErrors^{l+1}$ is the input to IM2COL as shown in Figure 3 (c). Then, we subject $QuantizedWeights^{l}$ to transposition and reversal of elements. This $TransposedReversedQuantizaedweights^{l}$ is fed as the input to the GEMM operation as shown in Figure 3 (c) along with the output of IM2COL.

Exploiting GEMM in preceding layer gradient ($Errors^{l}$ in Figure 3 (c)) of AMCONV2D for efficient execution on the GPU is even more non-trivial since both transposition and reversal of elements in $Weights^{l}$ are involved, in addition to padding and dilation of $Errors^{l+1}$.

Instead of having a separate kernel for dilating $Errors^{l+1}$ which would cause additional kernel invocation overhead, we integrate the dilation operation into the IM2COL_PLG_Kernel (a modified IM2COL Kernel that performs padding and dilation) where each thread copies a zero into IM2COL results if the current pixel is at a dilated position. Unfortunately, unlike in backpropagation for weights gradient where the second operand to GEMM requires dilation, the dilation must be performed on the input to the IM2COL. Thus, we cannot simply skip elements as is done for weights gradient computation despite the need for more GPU memory.

Transposition and the reversal of $QuantizedWeights^{l}$ can be implicitly done inside the GEMM kernel by manipulating the array index when accessing the second operand for GEMM. However, this would be highly inefficient because the global memory access pattern would not enable memory coalescing. Thus, here it is better to sacrifice some time to invoke a separate kernel that solely performs the reversal and transposition of $QuantizedWeights^{l}$, so that more time can be saved during the memory accesses of GEMM operation.

Since AMCONV2D is implemented by the GEMM approach, all multiplications are done in GEMM kernel; thus, we replace accurate multiplication in GEMM with AMSim device function to enable simulation.

III. Dense Layer Custom Op with Approximation (AMDENSE)

Unlike in the convolution layers, in the dense layer, each neuron receives input from all neurons in the preceding layer (see Figure 4). Like AMCONV2D described above, forward propagation and backpropagation of AMDENSE need to be implemented to realize training. Compared to AMCONV2D, AMDENSE occurs in a small proportion of the total computation and thus contributes to a tiny fraction of the total training time. Thus, CUDA optimization efforts are not as crucial as for AMCONV2D.

A. Forward propagation

Forward propagation can be mapped to a matrix-vector multiplication where weights in the dense layer are a 2-

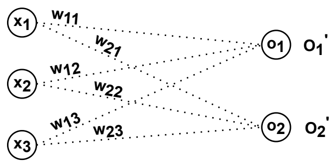

Fig. 4: AMDENSE implementation illustration.

dimensional matrix, and the activations from the preceding layer are a 1-dimensional vector. This is shown using a simplified example in Figure 4 where the dense layer output is computed as: $\begin{pmatrix} o_1 \\ o_2 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} w_{11}x_1+w_{12}x_2+w_{13}x_3 \\ w_{21}x_1+w_{22}x_2+w_{23}x_3 \end{pmatrix}$; where x is the activations, w is the weights. We implemented a separate matrix-vector multiplication CUDA kernel for this rather than using the previously used GEMM kernel, because shared memory-based tiling is superfluous for a 1-D vector.

B. Backpropagation

Similar to AMCONV2D, backpropagation in AMDENSE also involves computations of weights gradient and preceding layer gradient.

1) Weights Gradient Computations: The gradient of weights in the AMDENSE layer l is computed as $\delta_{out}a_{in}^{T}$ where a_{in} is the activation from preceding layer $l-1$ and δ_{out} is the error backpropagated from succeeding layer $l+1$. The gradient of weight in Figure 4 is computed as $\begin{pmatrix} w_{11}^{'} & w_{12}^{'} & w_{13}^{'} \\ w_{21}^{'} & w_{22}^{'} & w_{23}^{'} \end{pmatrix} = \begin{pmatrix} o_1^{'} \\ o_2^{'} \end{pmatrix}\begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}$. The same matrix-vector multiplication CUDA kernel is used here.

2) Preceding Layer Gradient Computations: The gradient of input in the AMDENSE layer l is calculated as $(w)^{T}\delta_{out}$ where $(w)^{T}$ is the transpose of the weights in the layer l. The gradient of input of given example in Figure 4 can be computed as $\begin{pmatrix} x_1^{'} \\ x_2^{'} \\ x_3^{'} \end{pmatrix} = \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{pmatrix}\begin{pmatrix} o_1^{'} \\ o_2^{'} \end{pmatrix}$. For the computation of the preceding layer gradient, we use the same matrix-vector kernel used for forward propagation. The transposition of the vector is implicitly handled because the elements are anyway stored linearly in memory.

We replace accurate multiplications in matrix-vector kernel with AMSim, considering that the matrix-vector kernel contains all multiplications of AMDENSE.

IV. MatMul Layer Custom Op with Approximation (AMMATMUL)

AMMATMUL is an approximate matrix multiplication operator that can be used to evaluate matrix multiplication with approximate multipliers and is a crucial component of Approximate Multi-Head Attention (AMMHA, refer to next sub-section V). Unlike AMCONV2D and AMDENSE, the backpropagation of AMMATMUL does not require a dedicated implementation, i.e., backpropagation can share the same Operator C++ class (refer to the blue-dashed box in Figure 1) as forward propagation. The backpropagation of AMMATMUL can be realized by properly calling the Operator C++ Class, as mentioned above.

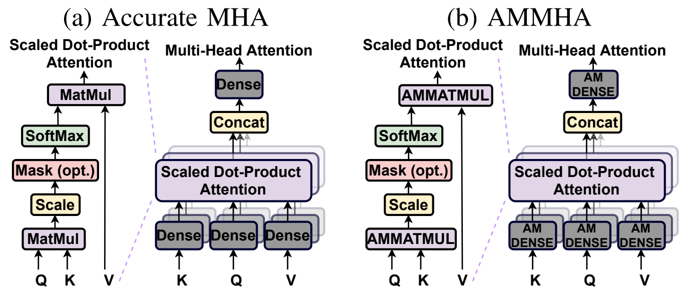

Fig. 5: Multi-Head Attention layer illustration. Sub-figure (a) is accurate Multi-Head Attention (MHA). Sub-figure (b) depicts the approximate Multi-Head Attention (AMMHA). K, Q and V represent queries, keys and values [5].

A. Forward propagation

Forward propagation of AMMATMUL is essentially a batched-matrix multiplication where the two input tensors are A with shape $(Batch, RowA, ColA)$ and B with shape $(Batch, RowB, ColB)$. The shapes of A and B must satisfy the following conditions: the $ColA$ is equal to the $RowB$. The output of batched-matrix multiplication can be computed as the 2-D matrix multiplication on inner two dimensions (Row, Col) of two input tensors for $Batch$ times. The 2-D matrix multiplication on the inner two dimensions (Row, Col) of two input tensors can be mapped to the GEMM kernel without extra implementation. Thus, forward propagation can be restructured as a loop that invokes the GEMM kernel iteratively on batches of tensor A and B.

B. Backpropagation

Unlike AMCONV2D and AMDENSE, the backpropagation of AMMATMUL does not involve the *Weights Gradient Computations* since AMMATMUL does not contain *Weights*, instead, two input tensors A and B.

Preceding Layer Gradient Computations: The gradient of input tensor A in AMMATMUL is computed as batched-matrix multiplication between gradient and transposed tensor B. For example, given two input tensors $A = \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$ and $B = \left(\begin{smallmatrix} e & g \\ g & h \end{smallmatrix}\right)$ where the forward propagation C is $\left(\begin{smallmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{smallmatrix}\right)$, the gradient of input tensor A is calculated as $\left(\begin{smallmatrix} o_1' & o_2' \\ o_3' & o_4' \end{smallmatrix}\right)\left(\begin{smallmatrix} e & f \\ g & h \end{smallmatrix}\right)^T$. Similarly, the gradient of input tensor B is computed as batched-matrix multiplication between transposed tensor A and gradient, which equals $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)^T\left(\begin{smallmatrix} o_1' & o_2' \\ o_3' & o_4' \end{smallmatrix}\right)$ for the given example. Note that TensorFlow handles transposition in the above example before running batched-matrix multiplication. Thus, gradient computation of tensor A and B can share the same Operator C++ Class used by forward propagation.

All multiplications in AMMATMUL are computed by the GEMM kernel; thus, we replace accurate multiplication in GEMM with AMSim device function to enable simulation.

V. Multi-Head Attention Layer Custom Op with Approximation (AMMHA)

The multi-Head Attention (MHA) layer is first proposed in the work [5]. It is a self-attention mechanism that has been widely adopted in many natural language processing (NLP) models, i.e., Transformer [5] and BERT [3]. Supporting MHA in ApproxTrain promotes the evaluation of approximate multipliers in the context of training NLP models. Here, we omit the principle of MHA and readers are referred to [5] for detailed explanation.

As depicted in Figure 5 (a), all multiplications in MHA are in two operators, MatMul and Dense. Thus, MatMul and Dense must be replaced with approximate layers to enable approximate training/inference. The approximate Multi-Head Attention layer (AMMHA) does not require C++ implementation as AMMHA can be directly built in python using existing C++ implementations of AMMATMUL and AMDENSE (refer to subsection III and IV). Thus, in Figure 5 (b), MatMul operators in the Scaled Dot-Product Attention are replaced with AMMATMUL, and Dense layers in MHA are replaced with AMDENSE.

AMMATMUL and AMDENSE are both equipped with AMSim to enable approximate multiplier simulation.

VI. Other Custom CUDA Kernels

The custom AM ops described above utilize several custom CUDA kernels. We developed these kernels to replace the kernels offered by the closed-source cuDNN and cuBLAS library. These custom CUDA kernels (which involve multiplication) are equipped to use AMSim to perform multiplication. In simple terms, these kernels may call the approximate multiplier functions with two operands as the arguments instead of using the '*' operator to multiply the two operands. A brief description of these custom kernels is given below.

GEMM kernel: The GEMM kernel is a highly optimized kernel that uses a 2-D threading indexing model with 16x16 as the CUDA thread block size. 16x16 tiles of the input matrices are fetched to fast GPU shared memory (on-chip SRAM) from global memory to be used for repeated memory accesses.

IM2COL kernels: There are three separate IM2COL kernels, as we mentioned before: 1. IM2COL (line 4 of the Algorithm 1) for forward propagation 2. IM2COL_Weight_Kernel (line 4 of the Algorithm 2) for weights gradient and 3. IM2COL_PLG_Kernel (line 6 of the Algorithm 2) for preceding layer gradient. IM2COLs mentioned above are implemented by utilizing a 1-D threading indexing model with 256 as the CUDA thread block size.

IM2COL: Each thread in IM2COL first locates the element position of A^l (the output of forward propagation), then locates the patch's element position (a flattened window) corresponding to A^l. The above two steps are needed to copy input data into the correct output position. Finally, the element in the input is located and copied to the IM2COL output.

IM2COL_Weight_Kernel: The IM2COL_Weight_Kernel first locates the element position of $WeightsGradient^l$ rather than A^l in forward propagation since its output is the $WeightsGradient^l$. Then, the IM2COL_Weight_Kernel locates the element position in the patch related to $WeightsGradient^l$. Finally, the IM2COL_Weight_Kernel locates the element in A^{l-1} and copies it to the IM2COL output; note that skipping elements is performed here if $stride$ is greater than 1.

IM2COL_PLG_Kernel: Similar to IM2COL and IM2COL_PLG_Kernel, IM2COL_PLG_Kernel first locates the element position of the preceding layer gradient ($Errors^{l-1}$ in Figure 3 (c) and then locates the element position in the patch. After the above two steps, the element position of input is located. However, this element position of input is computed based on the size of $PaddedDilatedErrors^{l+1}$ rather than $Errors^{l+1}$ (note the input data to IM2COL_PLG_Kernel is still $Errors^{l+1}$, but the size of input data is set to that of the $PaddedDilatedErrors^{l+1}$); thus, an additional procedure is implemented for each thread to check if current position is dilated position or not. The native IM2COL could handle padding, but the computation for the size of padding is different from forward propagation and weights gradient, despite not being explained here.

Transpose-And-Reverse Kernel: The TransposeAndReverse Kernel is a custom CUDA kernel that swaps dimensions of data and reverses elements order. It uses a 2-D threading indexing model with 32x32 as the CUDA block sizes. It first gets the index of a pixel along the height and width dimension of input to reverse the elements. Then, it gets the index of the dimension that is to be swapped in the following procedure. Then, this kernel starts swapping and reversing elements by manipulating the index. This kernel improves spatial locality by rearranging data order; thus, when GEMM kernel loads data into shared memory, memory coalescing occurs.

Matrix-Vector Multiplication Kernel:

Matrix-vector multiplication custom CUDA kernel is implemented by 1-D threading mode with 1024 threads in each block. Each thread will operate multiplication n times (n depends on the length of the vector).

VII. Conclusion

This document supplements the paper describing our open-source DNN framework ApproxTrain. In ApproxTrain, we create custom TF *ops* to support different types of DNN layers with different approximate multipliers. This document presented a detailed description of our approach to create two custom TF *ops* and the underlying CUDA kernels.

References

[1] CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2022-04-03.

[2] K. Chellapilla, S. Puri, and P. Simard. High Performance Convolutional Neural Networks for Document Processing. In Tenth International Workshop on Frontiers in Handwriting Recognition, Université de Rennes, 2006.

[3] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs], May 2019.

[4] S. Chetlur et al. cuDNN: Efficient Primitives for Deep Learning. arXiv:1410.0759 [cs], December 2014. arXiv: 1410.0759.

[5] A. Vaswani et al. Attention is All you Need. In Advances in Neural Information Processing Systems, volume 30. Curran Associates, Inc., 2017.