Fast GPU Convolution for CP-decomposed Tensorial Neural Networks

Alexander Reustle, Tahseen Rabbani, and Furong Huang

Department of Computer Science, University of Maryland
`areustle@cs.umd.edu, trabbani@umd.edu, furongh@cs.umd.edu`

Abstract. We present a GPU algorithm for performing convolution with decomposed tensor products. We experimentally find up to **4.85x** faster execution times than Nvidia's cuDNN for some tensors. This is achieved by extending recent advances in compression of CNNs through use of tensor decomposition methods on weight tensors. Progress had previously been limited by a lack of fast operations to compute the decomposed variants of critical functions such as 2D convolution. We interpret this and other operations as a network of *compound convolution and tensor contraction on the decomposed factors* (i.e., *generalized tensor operations*). The prior approach sees such networks evaluated in a pairwise manner until the resulting output has been recovered, by composing functions in existing libraries such as cuDNN. The computational cost of such evaluations depends upon the order in which the index sums are evaluated, and varies between networks. The sequence of pairwise generalized tensor operations that minimizes the number of computations often produces large intermediate products, incurring performance bottlenecks when communicated with the scarce global memory of modern GPUs. Our solution is a GPU parallel algorithm which performs 2D convolution using filter tensors obtained through CP-decomposition with minimal memory overhead. We benchmark the run-time performance of our algorithm for common filter sizes in neural networks at multiple decomposition ranks. We compare ourselves against cuDNN traditional convolutions and find that our implementation is superior for lower ranks. We also propose a method for determining optimal sequences of pairwise tensor operations, achieving a minimal number of operations with memory constraints.

Keywords: tensor methods, neural network inference, parallel algorithms

1 Introduction

Tensor decomposition methods have emerged as a means of training highly accurate convolutional neural networks while greatly reducing the number of model parameters [31, 3, 9, 21, 17]. So-called *Tensorial Neural Network* methods involving the CANDECOMP/PARAFAC (CP) decomposition [16, 31] in particular demonstrate significant promise; some models achieve accuracy scores 99% those of larger models with 1% of the parameters. These *Tensorial Neural Networks* express individual layers in deep neural networks as a graph of operations between input data and the factor tensors obtained from decomposing that layer's parameter tensor. This graph is referred to as a *tensorial neural network*, and these operations are *generalized tensor operations* [31].

Although these methods demonstrate significant promise, they are hampered by slow training speeds even on state-of-the-art hardware.

In this work we present our analysis of the causes of these slowdowns, along with part of our solution: a new GPU kernel to compute the forward pass of the critical 2D convolution using filter tensors derived from the CP decomposition. Written in CUDA, we refer to this as a *Fusion* method since it fuses the component operations of participating tensors, performing them in a single pass. This algorithm takes advantage of the many-thread concurrency of the GPU to execute the fused operation in parallel while minimizing communication with global memory. This algorithm does not achieve a minimal number of floating point operations; it redundantly recomputes intermediate values in parallel to avoid excessive global memory access. We benchmark our algorithm for runtime performance against the highly-optimized convolution algorithms in Nvidia's cuDNN library. As Figure 6 shows our fusion algorithm is largely superior for a variety of common convolutional layer sizes found in neural networks [19].

We also propose an alternative approach, a graph search algorithm which determines the optimal sequence of pairwise operations needed to evaluate the *tensorial neural network*. This approach is a modification of a similar method in the field of quantum many-body physics: the `netcon` [28] solver for optimal sequences of *tensor network* contractions [5, 7] where no convolution operations are involved. We extend this as `gnetcon` to support all *generalized tensor operations*, rigorously defined later in this paper.

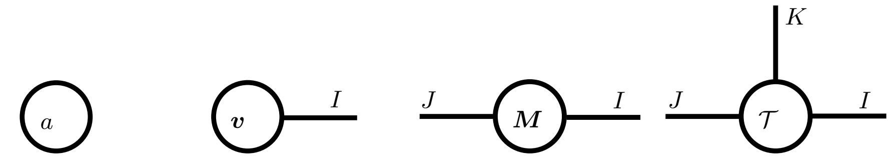
The remainder of this work is organized as follows: the rest of Section 1 describes important background information on tensor networks, tensorial neural networks and the CP decomposition. Section 2 analyzes the 2D convolution operation in both the full-sized and CP decomposed domains. Section 3 describes our GPU algorithm for performing the convolution forward pass with decomposed filter factors. Section 4 presents our work on identifying optimal pairwise sequences for evaluating tensorial neural networks. In Section 5 we outline our testing and benchmarking methodology, and in Section 6 we present the results. We discuss related works in Section 7, and our conclusions in Section 8.

1.1 Generalized Tensor Operations

Following the convention in quantum physics [25, 11], Figure 1 introduces *tensor diagrams*, graphical representations for multi-dimensional mathematical objects. Here an array (scalar/vector/matrix/tensor) is represented as a *node* in the graph, and its *order* is denoted by the number of *edges* extending from the node, where each edge corresponds to one *mode* (whose *dimension* is denoted by the number associated to the edge).

An algebra of primitive tensor operations has been presented which when compounded generalize existing neural network architectures [31]. This extends the *tensor network* concept originally from the field of quantum and condensed matter physics [6] with operations that equate to higher-order multilinear evaluations of individual layers in a neural network, along with derivative and backpropagation rules.

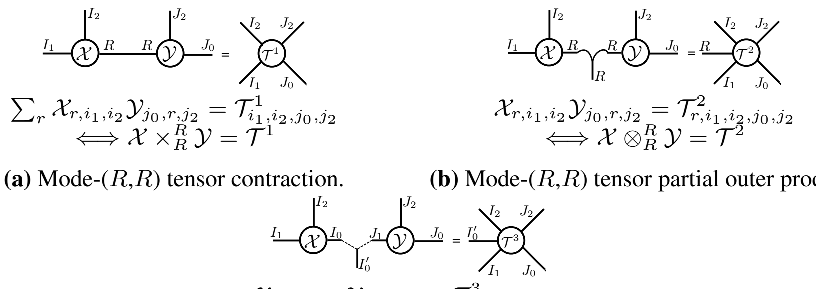
Three primitive operations are defined, the compound of which are *generalized tensor operations*. Figure 2 presents the primitives for *generalized tensor operations* on



(a) Scalar **(b)** Vector **(c)** Matrix **(d)** Tensor

Fig. 1: Tensor Diagrams of a scalar $a \in \mathbb{R}$, a vector $\boldsymbol{v} \in \mathbb{R}^I$, a matrix $\boldsymbol{M} \in \mathbb{R}^{I\times J}$, and a 3-order tensor $\mathcal{T} \in \mathbb{R}^{I\times J\times K}$.

high-order tensors, extending the matrix/vector operations[1] using tensor diagrams. In tensor diagrams, an operation is represented by linking edges from the input tensors, where the type of operation is denoted by the shape of line that connects the nodes: solid line stands for *tensor contraction/multiplication*, curved line is for *tensor partial outer product*, and dashed line represents *tensor convolution*. The *tensor contraction* generalizes matrix multiplication, while the *tensor partial outer product* generalizes the outer product for fibers of the operands. Finally, the *tensor convolution* can be defined by any convolution operation $*$ defined for 2 tensors.



$$\sum_r \mathcal{X}_{r,i_1,i_2}\mathcal{Y}_{j_0,r,j_2} = \mathcal{T}^1_{i_1,i_2,j_0,j_2} \iff \mathcal{X} \times^R_R \mathcal{Y} = \mathcal{T}^1$$

(a) Mode-(R,R) tensor contraction.

$$\mathcal{X}_{r,i_1,i_2}\mathcal{Y}_{j_0,r,j_2} = \mathcal{T}^2_{r,i_1,i_2,j_0,j_2} \iff \mathcal{X} \otimes^R_R \mathcal{Y} = \mathcal{T}^2$$

(b) Mode-(R,R) tensor partial outer product.

$$\mathcal{X}_{:,i_1,i_2} * \mathcal{Y}_{j_0,:,j_2} = \mathcal{T}^3_{:,i_1,i_2,j_0,j_2} \iff \mathcal{X} *^{I_0}_{J_1} \mathcal{Y} = \mathcal{T}^3$$

(c) Mode-(I_0,J_1) tensor convolution .

Fig. 2: Primitives of generalized tensor operations. $\mathcal{X} \in \mathbb{R}^{I_0\times I_1\times I_2}$ and $\mathcal{Y} \in \mathbb{R}^{J_0\times J_1\times J_2}$ are input tensors, and $\mathcal{T}^1 \in \mathbb{R}^{I_1\times I_2\times J_0\times J_2}$, $\mathcal{T}^2 \in \mathbb{R}^{I_0\times I_1\times I_2\times J_0\times J_2}$ and $\mathcal{T}^3 \in \mathbb{R}^{I'_0\times I_1\times I_2\times J_0\times J_2}$ are output tensors of corresponding operations. Existing tensor operations are only defined on lower-order $\mathcal{X}$ and $\mathcal{Y}$ such as matrices and vectors.

A *generalized tensor operation* can be arbitrarily complicated, which can take more than two tensors as inputs, and multiple edges are linked simultaneously among the tensors (an example is Figure 4b). In such a compound operation, different orders of evaluating the primitive operations yield the same result, though at the cost of different

[1] In Figure 2, we illustrate these operations with simple examples of third-order tensors $\mathcal{X}$ and $\mathcal{Y}$, but they also apply for higher-order tensors as rigorously defined in [31].

computational complexities. In general, it is NP-hard to obtain the best order to evaluate a compound operation with multiple tensor operands [20]. Using various tensor decomposition methods, Su et al. [31] convert the layers of existing network architectures into higher-order tensor network mappings, then use generalized operations to evaluate and retrain the nets. In doing so they demonstrate a significant reduction in model size while maintaining or in some cases improving upon the accuracy of the original network.

1.2 CANDECOMP/PARAFAC Decomposition

The CP decomposition [16] is a factorization of an order M tensor as a sum of R outer products between M vectors, where R is the tensor rank of the decomposition and each component vector's length is equal to the length of the corresponding mode in the original tensor. For example consider a 4-order tensor $\mathcal{K} \in \mathbb{R}^{T\times S\times H\times W}$, with component vectors $\boldsymbol{t}^{(r)} \in \mathbb{R}^T, \boldsymbol{s}^{(r)} \in \mathbb{R}^S, \boldsymbol{h}^{(r)} \in \mathbb{R}^H, \boldsymbol{w}^{(r)} \in \mathbb{R}^W$, the rank R decomposition of $\mathcal{K}$ is the sum (1).

$$\mathcal{K} = \sum_{r}^{R} \boldsymbol{t}^{(r)} \otimes \boldsymbol{s}^{(r)} \otimes \boldsymbol{h}^{(r)} \otimes \boldsymbol{w}^{(r)} \tag{1}$$

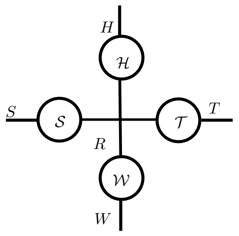


Fig. 3: Tensor diagram of CP decomposition of $\mathcal{K}$.

where $\otimes$ denotes the vector outer product. It is common to concatenate the matching component vectors into M matrices of R columns. In our example this would produce four matrices $\mathcal{T} \in \mathbb{R}^{T\times R}, \mathcal{S} \in \mathbb{R}^{S\times R}, \mathcal{H} \in \mathbb{R}^{H\times R}, \mathcal{W} \in \mathbb{R}^{W\times R}$. We can now express the CP decomposition of $\mathcal{K}$ element-wise as 2.

$$\mathcal{K}_{tshw} = \sum_{t,s,h,w,r} \mathcal{T}_{tr} \cdot \mathcal{S}_{sr} \cdot \mathcal{H}_{hr} \cdot \mathcal{W}_{wr} \tag{2}$$

Using our tensor diagram notation described earlier, the CP decomposition of K would factorize the 4th-order tensor into four matrices, each sharing the R mode in a 4-way contraction, as demonstrated in Figure 3 and in Equation (3).

$$\mathcal{K} = \mathbf{1} \times_R^R (\mathcal{S} \otimes_R^R \mathcal{H} \otimes_R^R \mathcal{W} \otimes_R^R \mathcal{T}) \tag{3}$$

where $\mathbf{1}$ is an all-ones vector of length R.

1.3 Problem Description

The method of [31] has a drawback — the computational and memory cost of the existing implementation of the *generalized tensor operations* is high. This problem is mainly due to the following challenges:

1. Existing neural network frameworks like Tensorflow [10] and Pytorch [27] use tuned GPU library functions [4] to perform the critical operations of convolution and dense matrix multiplication. No such operations exist for tensor operations on decomposed kernels.

2. Consequently, researchers evaluate decomposed operations as a sequence of pairwise tensor operations composed of existing functions like `tf.nn.conv2d`. This introduces memory-access overhead due to materializing the intermediate products. In some cases the storage requirements for these intermediate products exceeds the available GPU memory.
3. Moreover, there is often no pre-existing implementation of the optimal pairwise sequence which minimizes the number of floating point operations when memory is constrained. Discovering such sequences is in general NP-hard.

Our **goal** is then to provide implementation schemes that are computation- and space- efficient. We propose two alternative solutions:

1. Fusing all the component operations in a *generalized tensor operation* into a memory minimal fused operation, avoiding computation and memory overhead/bottlenecks of the intermediate products.
2. Finding sequences for performing pairwise operations for any *generalized tensor operations* to achieve minimal number of floating point operations under some memory constraint.

1.4 Contributions

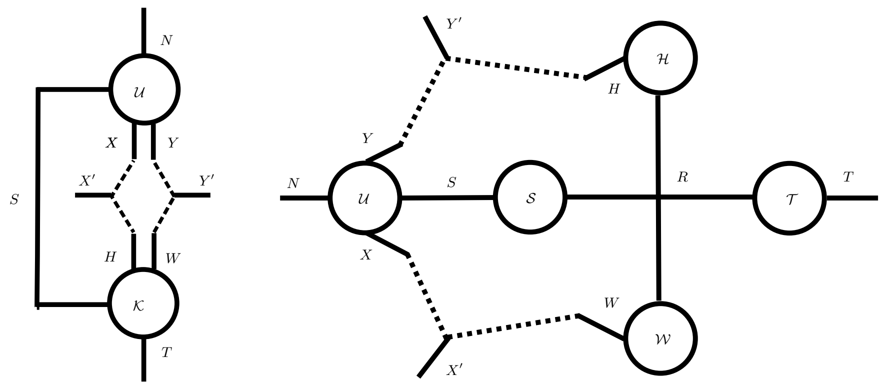
We introduce a *Fusion* method, a new GPU kernel to compute the forward pass of the critical 2-D convolution using filter tensors derived from the CP decomposition. We also propose an alternative approach, `gnetcon` a graph search algorithm which determines the optimal sequence of pairwise *generalized tensor operations* needed to evaluate the forward pass. Our fused approach implements a space/time trade-off. We store some small intermediate products to reduce onerous redundant computation, while redundantly computing other intermediates to maintain data locality and eliminate excessive global memory access. We confirm the speed of this approach empirically in Figure 6 and Table 2. Similarly, we find that our fusion method significantly reduces total global memory usage as presented in Figure 8 and Table 3. Further details provided in Sections 5 & 6.

2 Convolution in Tensorial Neural Networks

The goal of our research is to take neural network layers which have been decomposed using the tensor decomposition methods (a.k.a., Tensorial Neural Networks (TNNs)) in [31] and execute them efficiently on a GPU in parallel. We consider a layer in a TNN as a generalized neural network, that is a graph of *Generalized Tensor Operations* on the multiple component tensors within a layer. The component tensors consist of the input tensor and the weight tensors which have been decomposed from a higher-order convolutional layer.

2.1 Convolution-layer in CNN

A traditional 2D-convolutional layer is parameterized by a 4-order kernel $\mathcal{K} \in \mathbb{R}^{H \times W \times S \times T}$, where H, W are height/width of the filters, and S, T are the numbers of input/output



(a) $\mathcal{U}\left(*_H^Y \circ *_W^X \circ \times_S^S\right)\mathcal{K}$ **(b)** $\mathcal{U} \times_S^S \mathcal{S}(*_H^Y \circ \otimes_R^R)\mathcal{H}(*_W^X \circ \otimes_R^R)\mathcal{W} \times_T^T \mathcal{T}$

Fig. 4: Tensor diagram of **(a) Convolutional-layer in CNNs** and **(b) CP-decomposed Convolutional-layer in TNNs**. Both layers map a 4-order input tensor $\mathcal{U} \in \mathbb{R}^{N\times S\times X\times Y}$ to another 4-order output tensor $\mathcal{V} \in \mathbb{R}^{N\times T\times X'\times Y'}$, where X, Y and X', Y' are heights/widths of the input and output feature maps.

channels. Our implementation and experiments were conducted using the "channels first" data format for both the input tensor $\mathcal{U}$ and the output tensor $\mathcal{V}$. As illustrated in Figure 4a, the operation is compound as in Equation (4), where multiple primitive operations along different modes are executed. Specifically, two tensor convolutions at mode-(Y, H), mode-(X, W) and one tensor contraction at mode-(S, S) are performed simultaneously:

$$\mathcal{V} = \mathcal{U}\ \left(*_H^Y \circ *_W^X \circ \times_S^S\right)\ \mathcal{K} \tag{4}$$

Commonly convolution in neural networks is implemented as a cross-correlation [4], as we do here. The element-wise direct convolution of input tensor $\mathcal{U}$ with the filter tensor $\mathcal{K}$ is expressed in Equation (5), although optimized variants such as Fast Fourier transform and Winograd convolution are often preferred for performance.

$$\mathcal{V}_{nty'x'} = \sum_{s,h,w} \mathcal{K}_{tshw} \cdot \mathcal{U}_{ns(y'+h)(x'+w)} \tag{5}$$

2.2 CP-decomposed Convolution-layer in TNN

A CP-decomposed Convolution-layer in tensorial neural network is parameterized by 4 decomposed kernels $\mathcal{S} \in \mathbb{R}^{S\times R}$, $\mathcal{H} \in \mathbb{R}^{H\times R}$, $\mathcal{W} \in \mathbb{R}^{W\times R}$ and $\mathcal{T} \in \mathbb{R}^{T\times R}$, as shown in Figure 4b. The weight kernel $\mathcal{K}$ in Figure 4a is CP-decomposed as

$$\mathcal{K} = \mathbf{1} \times_R^R (\mathcal{S} \otimes_R^R \mathcal{H} \otimes_R^R \mathcal{W} \otimes_R^R \mathcal{T}) \tag{6}$$

where $\mathbf{1}$ is an all-ones vector of length R, and H, W are height/width of the filters, and S, T are the numbers of input/output channels. Both tensor contraction $\times_R^R$ and tensor partial outer product $\otimes_R^R$ are primitives of generalized tensor operations as defined in Figure 2.

TNN allows interactions between adjacent kernels through shared edges, crucial for modeling general multi-dimensional transformations without loss of expressive power. As illustrated in Figure 4b, each mode of the input tensor $\mathcal{U}$ interacts with the one of the decomposed kernels. The forward pass is as follows: $\mathcal{U}_0 = \mathcal{U} \times_S^S \mathcal{S}$, $\mathcal{U}_1 = \mathcal{U}_0(*_H^Y \circ \otimes_R^R)\mathcal{H}, \mathcal{U}_2 = \mathcal{U}_1(*_W^X \circ \otimes_R^R)\mathcal{W}$ and $\mathcal{V} = \mathcal{U}_2 \times_T^T \mathcal{T}$, where $\mathcal{U}_0, \mathcal{U}_1$ and $\mathcal{U}_2$ are intermediate objects. We combine the above equations into a sequence of generalized tensor operation

$$\mathcal{V} = \mathcal{U} \times_S^S \mathcal{S}(*_H^Y \circ \otimes_R^R)\mathcal{H}(*_W^X \circ \otimes_R^R)\mathcal{W} \times_T^T \mathcal{T}. \tag{7}$$

Consider the naive single-element calculation for convolution between an order-4 input tensor $\mathcal{U}$, and four decomposed kernels $\mathcal{S}$, $\mathcal{H}$, $\mathcal{W}$ and $\mathcal{T}$

$$\mathcal{V}_{nty'x'} = \sum_{s,h,w,r} \mathcal{T}_{tr} \cdot \mathcal{S}_{sr} \cdot \mathcal{H}_{hr} \cdot \mathcal{W}_{wr} \cdot \mathcal{U}_{ns(y'+h)(x'+w)} \tag{8}$$

A single element in Equation (8) minimally requires $(5SHWR)$ floating point operations to compute. However computing $\mathcal{V}_{n(t+1)y'x'}$ shares many common operations with $\mathcal{V}_{nty'x'}$ a fact generally true for $\mathcal{V}_{nt(y'+1)x'}$,$\mathcal{V}_{nty'(x'+1)}$ and other elements covered by the same filter. Computing the fiber of all T output channels using Equation (8) takes $(5TSHWR)$ floating point operations. To compute $\mathcal{V}_{n:y'x'}$ while minimizing redundant computation for an entire fiber of T output channels, we must share intermediate products.

Applying the distributive property of scalar arithmetic, and storing the intermediate accumulation in a vector of length R, we can express 8 as 9:

$$\begin{aligned} \boldsymbol{a}_r &= \sum_{s,h,w} \mathcal{S}_{sr} \cdot \mathcal{H}_{hr} \cdot \mathcal{W}_{wr} \cdot \mathcal{U}_{ns(y'+h)(x'+w)} \\ \mathcal{V}_{nty'x'} &= \sum_{r} \mathcal{T}_{tr} \cdot \boldsymbol{a}_r \end{aligned} \tag{9}$$

Computing 9 would require $(4RSHW + 2TR)$ floating point operations for a fiber of T elements. This is more efficient when $T < {}^1\!/_2 SHW$, a fact commonly true in neural network layers. With the added trade off of storing the intermediate accumulation vector. Similar intermediate storage options are available for the other modes of the tensor.

3 GPU Fused CP Convolution Operation

All results presented in this paper report performance obtained on the Nvidia RTX 2080-Ti (Turing) GPU. This device has a theoretical maximum single precision (32-bit) floating point performance of 14.1 TFLOPS, which it achieves with 4352 CUDA cores spread across 68 streaming multiprocessors on chip [8]. This parallelism is exposed to the programmer in an organizational hierarchy of computation with *threads*, *warps*, *blocks* and *grids*.

Threads are the smallest unit of compute and are organized into *warps* of at most 32 sequential threads executing simultaneously, and further grouped into *blocks* of execution that may communicate via shared memory. A *grid* of blocks is executes in an arbitrary sequence to perform the desired calculation on data resident in the devices global memory. The 2080-Ti has theoretical peak bandwidth for global memory access of 616 GB/s. While impressive this is insufficient to promptly deliver data to the all active threads on chip. Thus many compute kernels are performance limited by memory load/store operations, not FLOP count. Such kernels are known as *bandwidth-bound* kernels. The 2080-Ti alleviates this bottleneck somewhat with a small L1/L2 cache, which can improve the performance of *bandwidth-bound* kernels that exploit data locality. The GPU also exposes a set shared memory that may be accessed collaboratively by threads in a block, a core method of thread communication. The other is the use of warp-level primitives, where threads in a warp communicate directly.

3.1 Naive Implementation

Algorithm 1 Naive CP Convolution single element

```
Input: $n, t, x', y'$
1: for $r < R$ do
2:     for $s < S$ do
3:         for $h < H$ do
4:             for $w < W$ do
5:                 $\mathcal{V}_{n,t,y',x'} \mathrel{+}= \mathcal{T}_{tr}\mathcal{S}_{sr}\mathcal{H}_{hr}\mathcal{W}_{wr}\mathcal{U}_{n,s(y'+h)(x'+w)}$
6:             end for
7:         end for
8:     end for
9: end for
```

A naive implementation of equation (8) can be seen in algorithm 1. The deeply nested loops redundantly recompute many subproduct terms, and share neither intermediate products, nor common input elements. This latter part causes it to suffer from substantial bandwidth constraints as adjacent threads load repeated data elements from global memory.

3.2 Proposed Implementation

For our algorithm to utilize the parallelism provided by the GPU we must split the input tensor into small independent tiles of data. The output elements obtained from these tiles are computed at the block level, independently of other blocks in the grid. Thus we introduce a redundant computation trade off: intermediate results obtained from data elements fully in the block are shared, while those in the in boundary regions must be recomputed in each tile.

The fused kernel accepts input data from tensor $\mathcal{U}$ in the "channels first" data format, the format most favorable to cuDNN. Thus a 4-order tensor $\mathcal{U}$ has modes batch size (N), input channels (S), feature height (Y), and feature width (X) ordered from least to most frequently varying.

In our algorithm 2, a block of threads is allocated for each tile of input. At kernel launch, the threads are assigned coordinates denoting which y', x' input they will convolve, and which subset of s channels they will contract. As input channels often

Algorithm 2 GPU Fused CP Convolution

```
Input: $n, t, x', y'$ ,$\mathcal{U}$ ,$\mathcal{T}$ ,$\mathcal{S}$ ,$\mathcal{H}$ ,$\mathcal{W}$
    {Allocate Length $R$ vectors for local intermediates}
 1: $\boldsymbol{q} \leftarrow 0$
 2: $\boldsymbol{p} \leftarrow 0$
 3: for $s$ <S ; stride == S_TILE do
 4:     $\boldsymbol{p} \leftarrow 0$
 5:     for $h < H$ do
 6:         for $w < W$ do
 7:             $\boldsymbol{p} \leftarrow \boldsymbol{p} + \mathcal{U}_{n,s(y'+h)(x'+w)} \circ \mathcal{H}_{h:} \circ \mathcal{W}_{w:}$
 8:         end for
 9:     end for
10:     $\boldsymbol{q} \leftarrow \boldsymbol{q} + \boldsymbol{p} \circ \mathcal{S}_{s:}$
11: end for
12: for $t < T$ do
13:     $a \leftarrow \langle \boldsymbol{q}, \mathcal{T}_{t:} \rangle$
14:     $a \leftarrow$WarpReduce($a$, S_TILE_INDEX)
15:     SyncWarp
16:     if S_TILE_INDEX== 0 then
17:         $\mathcal{V}_{nty'x'} \leftarrow a$
18:     end if
19: end for
```

outnumber threads available for contraction, threads loop over channels, striding by the channel tile size "S_TILE". Each thread maintains a local vector of length R into which intermediate values accumulate. Assuming small R this vector can be maintained entirely in a thread's local registers. Threads read input data from a few contiguous regions of global memory, to maximally utilize global memory bandwidth.

Algorithm 2 assumes the tensor dimensions and the memory constraints are known at runtime, which is satisfied in TNNs. It further assumes that minimizing FLOPs count within the provided memory constraint is sufficient for finding an optimal sequence. If gnetcon fails to satisfy the memory constraint, another approach (such as fusion) with a smaller memory footprint must be used.

Most global data loads occur in the inner-most loop at line 7. Warps load contiguous chunks of $\mathcal{U}$ and perform an element-wise Hadamard product on the length R row vectors of the filter factors $\mathcal{H}$ and $\mathcal{W}$. Here we exploit the data locality of the L1 and L2 cache. The participating elements from $\mathcal{U}$ will be shared by adjacent warps. Sequential warps which depend upon that region of memory are very likely to find it in the cache when executing, avoiding global memory loads. Warps also share the same row of the filter tensors $\mathcal{H}$ & $\mathcal{W}$, which is also cached.

The vector $\boldsymbol{p}$ is local to the thread and never shared. It is element-wise scaled with a slice of the input channel tensor $\mathcal{S}$ for each thread at line 10. Input channel sizes are commonly large in neural network layers. Accessing that chunk of global memory is likely to eject the previously cached loads and cause a reduction in performance, so we perform it independently of the operation at line 7.

After striding over the input channels of the tensor, each thread now contains a length R vector $\boldsymbol{q}$, which is a partial sum. Looping over output channels $t \in T$, at line 13 each thread performs an inner product calculation with its local $\boldsymbol{q}$ and the output channel tensor for t, accumulating the product in the scalar a. These threads are adjacent, thus executing in the same warp, thus at line 14 we use the warp level primitives to share intermediates, performing a reduction on the values in a spread across each thread. This performs the reduction in at most $\log_2(32) = 5$ operations, and accumulates the final sum in the a variable of the warp with index 0. Line 15 is a barrier for the warp of threads ensuring they have all completed, thus the accumulated sum is correct. This synchronization barrier does not extend to the rest of the block, and so does not hinder performance. Finally the thread which has accumulated the result writes it out to global memory in the output tensor $\mathcal{V}$.

4 Optimal Operation Sequences

Minimizing the number of floating point operations in the evaluation of an input tensor is a standard approach we consider and measure against the fusion approach. To develop an algorithm which could determine the minimal number of operations needed to evaluate an input in a tensorial representation of a neural network, we extend the techniques used in networks consisting solely of contractions, which are very well-studied. Consider the following sequence of tensors,

$$\sum_{i,j,k,l} \mathcal{X}_{ijk}\mathcal{Y}_{ikl}\mathcal{Z}_{mln} + \sum_{p,q} \mathcal{W}_{pr}\mathcal{V}_{qs}, \tag{10}$$

where the sums are over same-dimensional modes between tensors, i.e., contractions. A classical question arises when attempting to evaluate such a network: should memory consumption or the number of intermediary operations be minimized? In this section, we concern ourselves with the latter. For a simpler example, consider a matrix multiplication such as $\boldsymbol{A} = \boldsymbol{B} \times \boldsymbol{C} \times \boldsymbol{D}$ which consists of a sequence of contractions $\boldsymbol{A}_{ij} = \sum_k \boldsymbol{B}_{ik}\boldsymbol{C}_{kl}\boldsymbol{D}_{lj}$. One may ask if $(\boldsymbol{BC})\boldsymbol{D}$ or $\boldsymbol{B}(\boldsymbol{CD})$ costs less floating-point operations. Efficient contraction of tensor networks has a vast literature, and appears in many quantum computational chemistry problems. The problem rapidly becomes intractable if the network contains many tensors with a total collection of many modes.

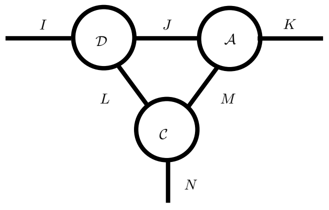
Finding the optimal contraction sequence which minimizes the number of floating point operations is known to be NP-hard, but fast algorithms do exist. One such well-known breadth-first algorithm is `netcon()` due to Pfeifer et al. [28]. We refer to the summary of the breadth-first approach as described in [28]:

1. Let $L_1 = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ be the set of tensors in the network.
2. Let i be an index counter from 2 to n. For each i:

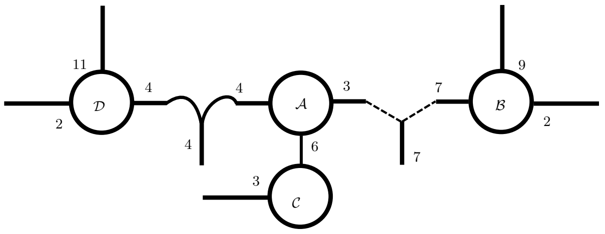
(a) Let L_c be the set of all possible subnetworks created by contracting i tensors from L_1.
(b) For each pair of sets L_d, L_{c-d}, $1 \leq d \leq \lfloor \frac{c}{2} \rfloor$, and for each $\mathcal{T}_a \in L_d$, $\mathcal{T}_b \in L_{c-d}$ such that each element of L_1 appears at most once in the subnetwork $(\mathcal{T}_a \mathcal{T}_b)$:
   i. Compute the cost c of contracting $\mathcal{T}_a$ with $\mathcal{T}_b$.
   ii. If $\mathcal{T}_a$ and/or $\mathcal{T}_b$ are not in L_1, then add the cost of constructing of $\mathcal{T}_a$ and/or $\mathcal{T}_b$ to c.
   iii. Let the contraction sequence $\mathcal{S}$ for constructing this subnetwork be written $S = (\mathcal{T}_a \mathcal{T}_b)$. If $\mathcal{T}_a$ and/or $\mathcal{T}_b$ are not in L_1, then optimal contraction sequences for $\mathcal{T}_a$ and $\mathcal{T}_b$ will have been recorded already. In $\mathcal{S}$, replace the occurrences of $\mathcal{T}_a$ and/or $\mathcal{T}_b$ with these sequences.
   iv. Locate the subnetwork in L_c which corresponds to $(\mathcal{T}_a \mathcal{T}_b)$. If c is the cheapest cost for constructing this subnetwork, record c and the associated contraction sequence $\mathcal{S}$ against this subnetwork.

3. The optimal cost c_{best} and associated sequence $\mathcal{S}_{\text{best}}$ are recorded against the only element of L_n.

`netcon()` contains a cost-capping feature: subnetworks may be rejected for operation if the intermediate product exceeds a predefined memory constraint and other cheaper paths to the final product are available. While `netcon()` may be used to evaluate a network such as Figure 5a, it cannot be used to evaluate the generalized tensor operations which involve, for instance, convolutions and partial outer products. For instance, one layer of the network $\mathcal{A} *_7^3 \mathcal{B} \times_6^6 \mathcal{C} \otimes_4^4 \mathcal{D}$ in Figure 5b.



(a)



(b)

Fig. 5: Example networks. **(a)** An example of tensor network. **(b)** An example of 1 layer of a deep tensorial neural network involving generalized operations.

One of the core contributions of this paper is a generalization of `netcon()` which we refer to as `gnetcon()` capable of finding the optimal operation sequence for a given tensorial neural network. `gnetcon()` modifies `netcon()` by introducing an updated cost model to handle any generalized operation. For $\mathcal{U} \in \mathbb{R}^{I_0 \times I_1 \times \ldots \times I_{m-1}}$, $\mathcal{V} \in \mathbb{R}^{J_0 \times J_1 \times \ldots \times J_{n-1}}$, we introduce the following floating point operation complexities [2] to

[2] Note that for convolution cost (12), we assume no Fast-Fourier Transform is used.

obtain optimal pairwise sequence for the generalized tensor operations:

$$\text{cost}[\mathcal{U} \times_l^k \mathcal{V}] = O((\textstyle\prod_{u=0}^{m-1} I_u)(\prod_{v=0,v\neq l}^{n-1} J_v)) \tag{11}$$

$$\text{cost}[\mathcal{U} *_l^k \mathcal{V}] = O((\textstyle\prod_{u=0}^{m-1} I_u)(\prod_{v=0}^{n-1} J_v)) \tag{12}$$

$$\text{cost}[\mathcal{U} \otimes_l^k \mathcal{V}] = O((\textstyle\prod_{u=0}^{m-1} I_u)(\prod_{v=0,v\neq l}^{n-1} J_v)). \tag{13}$$

Furthermore, `gnetcon()` maintains the cost-capping feature of `necton()` and thus can handle predefined memory constraints. As an example, running `gnetcon(`$\mathcal{A} *_7^3 \mathcal{B} \times_6^6 \mathcal{C} \otimes_4^4 \mathcal{D}$`)` for the generalized tensor operation in Figure 5b, we obtain the optimal pairwise operation sequence $((\mathcal{A} \times_6^6 \mathcal{C}) *_7^3 \mathcal{B})) \otimes_4^4 \mathcal{D})$. The above costs 38,016 floating point operations, whereas a naive implementation in a order such as, $(((\mathcal{A} *_7^3 \mathcal{B}) \times_6^6 \mathcal{C}) \otimes_4^4 \mathcal{D})$, costs 45,360 floating point operations.

Using `gnetcon()`, we can execute a forward pass in a tensorial neural network according to an operation-minimizing strategy.

Forward Passes in CP-decomposed Convolution-layer in TNN As Generalized Tensor Operation Sequences In a CP-convolutional layer as shown in Figure 4b, the forward pass is a general tensor operation sequence $\mathcal{V} = \mathcal{U} \times_S^S \mathcal{S}(*_H^Y \circ \otimes_R^R)\mathcal{H}(*_W^X \circ \otimes_R^R)\mathcal{W} \times_T^T \mathcal{T}$. Once the dimensions of $\mathcal{U}$, $\mathcal{S}$,$\mathcal{T}$,$\mathcal{H}$ and $\mathcal{W}$ are set, Equation (7) is passed into `gnetcon()` to determine the optimal order of operations. Similarly, our`gnetcon()` determines the minimal number of floating operations needed to carry out a forward pass in one layer of a convolutional neural network (CNN). We measure the time needed to execute a forward pass in the order recommended by `gnetcon()` and use these times as a baseline comparison against the fusion approach.

5 Performance Benchmarking

To test the correctness and performance of our CP Convolution kernel we relied heavily on facilities provided by the CUDA library. All tests were conducted on a private workstation running 64-bit Ubuntu 16.04, with a 12 core Intel Xeon CPU E5–1650 v4 CPU and 64GB RAM. The GPU is an Nvidia RTX 2080-Ti with driver Version: 418.87.00. Our implementation was compiled using CUDA Version: 10.1.243, while cuDNN version 7.4.2.24 was used as both a benchmark and an oracle for operator correctness.

5.1 Correctness Testing

All output values from the fused CP convolution kernel were verified as correct to a floating point tolerance of $\epsilon = 10^{-5}$. The testing procedure was as follows. In advance, we determined a list of input and filter tensor shapes. For all shapes in the list we allocated in GPU global memory a 4th-order input tensor $\mathcal{U}$ with uniformly random elements between 0 and 1. These elements were generated by cuRAND using a constant seed of "1234". We also generated four filter matrices with S, T, Y, and X rows respectively, and R columns, where R is the rank of the CP decomposition. We vary R for the tests and benchmarks, allowing it to take on the values $[1, 2, 4, 8, 16]$.

The 4-order, rank R filter tensor $\mathcal{K}$ was composed from the CP factor matrices by applying equation 1. The traditional convolution $\mathcal{V} = \mathcal{U} * \mathcal{K}$ was computed using the function *cudnnConvolutionForward*, and a cuDNN convolution algorithm gotten with the `CUDNN_CONVOLUTION_FWD_PREFER_FASTEST` selection. The CP convolution was calculated using our algorithm to produce $\mathcal{V}$. Implicit padding values were chosen to replicate a "SAME" padding often used in deep neural networks.

Correctness was determined by comparing each element of tensors $\mathcal{V}$, and $\mathcal{V}'$ for approximate equality to within a floating point tolerance of $\epsilon = 10^{-5}$. This was done using both a CPU library function and a custom GPU comparison function. The CPU library was the C++ *DocTest* unit test framework. The custom GPU comparison implemented "close enough" comparison from [14] §4.2.2, Eq. 37. Both were used for verifying the final kernel.

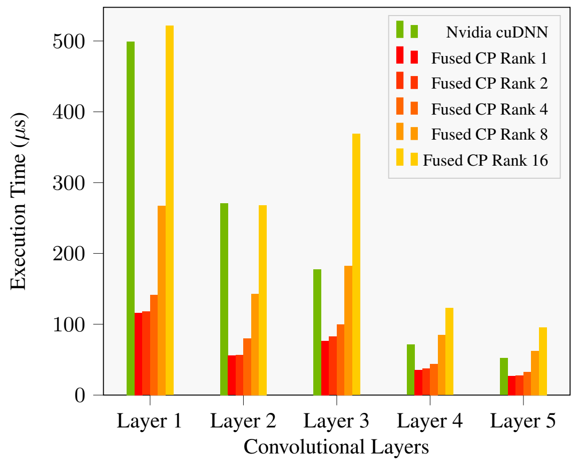


Fig. 6: Fused CP Convolution Execution Time of different operators on common convolution layers of neural networks Nvidia cuDNN with full-sized filter tensors vs. our Fused CP Convolution operation at various decomposed tensor ranks.

5.2 Fused CP Convolution Performance Benchmarking

Input tensors for timing benchmarks were generated using the same method used for correctness testing. The sizes and shapes of these tensors are described in more detail in table 1. All are stored on device in GPU global memory before benchmarking began. The cuDNN library provides many algorithm options for forward convolution in the channels first data format. We executed the one selected by cuDNN using the `CUDNN_CONVOLUTION_FWD_PREFER_FASTEST` algorithm preference. Many of these algorithms require some amount of "workspace memory" to be allocated in global memory, which was fully provided for our tests.

Timing values were captured by decorating the kernel launch code with profiling calls using the `cudaEventRecord` features exposed by the CUDA API. Calls for start and stop to `cudaEventRecord` were placed immediately before the kernel launch, and after the `cudaSynchronizeDevice` call to ensure full recording of only the kernel launch and complete execution. Each operation was repeated 47 times and the duration of each run was captured. Results presented are the arithmetic mean of all run durations expressed in microseconds (μs).

5.3 Pairwise Sequential Convolution Performance Benchmarking

We also benchmark the performance of our optimal pairwise sequence of *generalized tensor operations* using the TensorFlow deep neural network framework [10]. We express the graph of tensors in tensor diagram and generate the order to evaluate shared edges. Results are obtained using the built-in `tf.test.Benchmark` class and Tensorflow version 1.14.

Table 1: Convolution Operation Data Sizes. Tensor sizes taken from common convolution layers in neural networks. All batch sizes are 1. (S): # of input channels. (Y): feature tensor height. (X): feature tensor width. (T): # of output channels. (H): filter height. (W): filter width. (Rank) $\in \{1, 2, 4, 8, 16\}$.

Convolution Layer	$(S\ Y\ X)$	$(T\ H\ W)$	# Features.	# Filter params.	# CP Filter params.
1	(3, 224, 224)	(96, 11, 11)	150,528	34,828	121·Rank
2	(48, 55, 55)	(256, 5, 5)	145,200	307,200	314·Rank
3	(256, 27, 27)	(384, 3, 3)	186,624	884,736	646·Rank
4	(192, 13, 13)	(384, 3, 3)	32,448	663,552	582·Rank
5	(192, 13, 13)	(256, 3, 3)	32,448	442,368	454·Rank

6 Benchmarking Results

Our benchmark results are summarized in figure 6 and table 2. As Figure 6 shows our fusion algorithm is superior to cuDNN in most low-rank instances for common convolutional layers. Fused CP convolution is the faster in most cases below rank 16.

The fastest relative performance occurred at the rank 1 decomposition of layer 2. **Our code performed 4.85x faster than cuDNN with 75% of the memory usage**. The superior performance of our Fused CP convolution kernel extends even to higher ranks. The rank 4 run of deep layer 4 ran **1.61x faster, while using 26.1x less memory**.

A deeper look suggests that our algorithm scales linearly with rank, note the rank step size increments in sequential powers of 2. Another observation is that all ranks of fused CP convolution scale very well with channel depth, but somewhat poorly with input feature height and width. This is a limitation shared by cuDNN for traditional convolution.

Table 2: Execution Time in μs. cuDNN convolution benchmarked against our Fused CP-convolution operator, and our Pairwise optimal sequencer implemented in TensorFlow. Data sizes defined in table 1.

	cuDNN	Fused CP Convolution (μs)					Pairwise Sequential CP Convolution (μs)				
Layer:	(μs)	Rank 1	2	4	8	16	Rank 1	2	4	8	16
1	499.4	**116.3**	**117.9**	**141.7**	**267.6**	521.5	3069.5	3178.9	3308.4	3519.5	3720.0
2	270.7	**55.8**	**56.9**	**80.1**	**143.2**	**267.9**	2079.7	1911.0	1838.0	1961.8	2073.7
3	177.4	**76.6**	**82.5**	**99.8**	182.5	368.8	2033.9	2047.5	2066.2	1985.0	1936.4
4	71.3	**35.4**	**37.3**	**44.2**	84.7	122.8	1810.0	1812.5	1907.5	1860.3	1759.7
5	52.1	**26.9**	**27.9**	**32.8**	62.6	95.5	1741.4	1790.6	1779.8	2399.3	1761.5

The pairwise sequential operation implemented in TensorFlow does not share the same superior performance characteristics as our fused convolution kernel. This is starkly visible in figure 7 which plots the execution time of the pairwise sequential forward convolution for various ranks of the different convolutional layers. We contribute much of this to the overhead introduced by tensorflow, which will automatically manage migration of intermediate tensors in and out of device memory during execution. Nevertheless this comparison is warranted as TensorFlow, along with the other popular neural network frameworks, remains the only widely available means of evaluating a *tensorial neural network* layer.

Our pairwise sequencer, implemented in TensorFlow, is the current state-of-the-art for TNN implementations, which are not currently addressed by other libraries like cuDNN. It is entirely possible that we registered slower execution times exclusively due to TensorFlow overhead. The cuDNN “baseline” we compare our pairwise sequencer against is an “equivalent” convolution after merging the sequence of operations being processed by our pairwise sequencer into one convolution; therefore not a fair comparison. A fair comparison would be to implement the sequence directly in cudnn library calls, a non-trivial task that we defer to a future work.

Turning to the memory usage in Figure 8 and Table 3, we see that the fused operator uses the least memory in all cases. We use a log-scale along the vertical axis for better visualization due to the scale difference between layers. All operations materialize the input and output feature tensors, which account for the bulk of the memory footprint in the shallower layers. The increased cuDNN memory usage in later layers is largely a consequence of the extra “workspace memory” these particular cuDNN algorithms require. The values for the pairwise sequential convolution memory usage express the cumulative total of all participating tensors, including intermediate products. The full impact of the large footprint is alleviated somewhat by TensorFlow, which will act to stream intermediate data out of GPU memory and into host memory to reduce global memory pressure. Thus the true allocation size maximally resident in the GPU at during pairwise sequencing is often lower than the cumulative values.

Our two approaches are state-of-the-art for TNNs. There are directions to improve our approaches such as FFT, Winograd, GEMM lowering, or Tensor Core approaches. Such implementations represent possible future work.

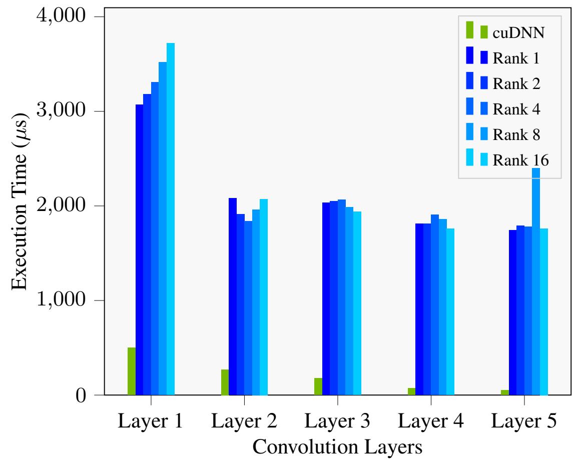


Fig. 7: Pairwise Sequential Forward Convolution Execution Time of different operators on common convolution layers of neural networks Nvidia cuDNN with full-sized filter tensors vs. our pairwise sequential CP Convolution operation at various decomposed tensor ranks.

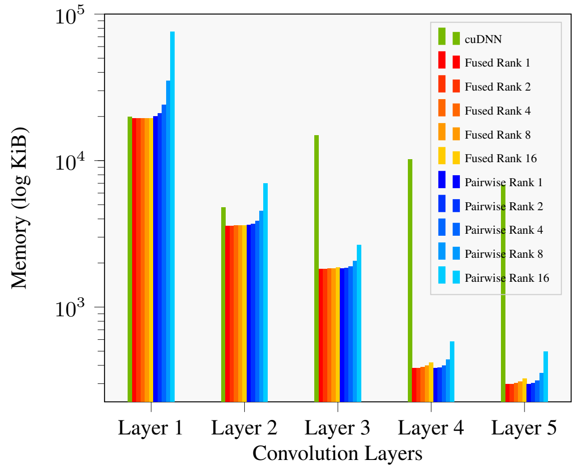


Fig. 8: Global Memory Usage (log KiB) of different operators on common convolution layers of neural networks, including traditional cuDNN convolution, Fused CP convolution, and pairwise sequential CP convolution at various decomposed tensor ranks.

Table 3: Global Memory Usage (KiB) of different operators on common convolutional layers of neural networks, including traditional cuDNN convolution, Fused CP convolution, and pairwise sequential CP convolution at various decomposed tensor ranks.

	cuDNN	Fused CP Convolution (KiB)					Pairwise Sequential CP Conv. (KiB)				
Layer:	(KiB)	Rank 1	2	4	8	16	Rank 1	2	4	8	16
1	19834	**19405**	**19405**	**19406**	**19408**	**19412**	19992	20972	24108	35085	75854
2	4810	**3593**	**3595**	**3597**	**3602**	**3612**	**3628**	**3687**	**3877**	4539	6999
3	14880	**1825**	**1828**	**1833**	**1843**	**1863**	**1832**	**1847**	**1895**	**2059**	**2659**
4	10174	**383**	**385**	**389**	**398**	**417**	**383**	**387**	**399**	**439**	**583**
5	6825	**298**	**299**	**301**	**310**	**324**	**299**	**303**	**315**	**355**	**498**

7 Related Works

In the field of unsupervised learning, advancement has come through applying tensor decomposition methods to the problem of learning latent variable models [1]. Much research on tensor decompositions is directed at approximations of the CP decomposition described in Section 1.2; leading to research applying tensors and tensor decompositions to neural networks. In [18], the authors replace fully connected layers and the flattening step necessary to transition to them from convolutional layers with a novel pair of tensor-based layers, retaining structural information. Another instance of tensorizing existing neural network layers appears in [21] where the authors reduce the processing time incurred on convolution kernels through sequential application of CP decomposed factor tensors. This early work demonstrated the potential for considerable speedups in CPU implementations of convolution using tensor decomposition.

Ideas for fully tensorizing neural networks are also popular. The authors in [24] transform the dense matrix weights of fully connected layers into decomposed tensors represented with the Tensor-train decomposition from [26]. This can be applied to all fully connected layers in a network, resulting in a totally tensorized neural network. Tensor decomposition methods were used to prove theoretical guarantees on the generalization error of two-layer neural networks in [12], and deep CP decomposed CNNs by [22], with state of the art generalization guarantees proven by statistical bounds on generalization error.

Other fields have focused on efficient tensor contraction due to the primacy of contraction in molecular chemistry models. The Tensor Contraction Engine automatically compiles contraction code for high performance computing environments [2]. This was extended to GPU code in [23].

An alternative approach described by [15] consists in first using transposition to matricize each tensor in the calculation, then applying a fast GEMM kernel, and transposing again for the final result. This approach takes advantage of existing kernels but the transposition operations are total overhead. [30] attempt to avoid the transposition cost by performing the transpositions as data is loaded into shared memory. This in conjunction with specialized kernels avoids the overhead of transposition. Separately [29] avoid transposition by developing `stridedBatchedGemm` for single-mode contractions. In effect looping over GEMM operations without reshaping the tensor; which can experience poor memory access patterns if the tensor modes are highly rectangular. Recently [13] exploit domain specific properties of data reuse in tensor contractions, applying their insights to devise an explicit code generator and demonstrate superior performance on most test cases of the TCCG benchmark.

None of these tensor contraction works support *generalized tensor operations*, and in particular do not generalize to operations between more than two tensors, nor support convolution along any tensor mode. Therefore, our work is unique and novel.

8 Conclusion

The fused CP convolution has considerable potential as a viable and high performing operation in future deep learning tasks, particularly for tensorial neural networks. The runtime performance is superior to the current baseline for exactly the types of low-rank approximations we expect to be of most interest to neural network researchers. The reduction in global memory usage is considerable when compared to both a traditional convolutional layer, and an optimal pairwise sequential evaluation. We speculate that the alleviation of global memory limits will enable researchers to find other novel ways to use the available memory, for example, on larger training batches.

The fusion approach is not without downsides however. Foremost is the required engineering and development time. Next, the lack of flexibility for use in other network architectures based on alternative tensor decompositions. The fused CP convolution is not applicable to either a Tucker or Tensor Train decomposition. All alternative *tensorial neural network* layers that employ *generalized tensor operations* would need custom fused operators.

These drawbacks are not shared with the optimal pairwise sequencer approach. By taking advantage of existing library functions and their gradient operations, current TNN training schemes can proceed with existing implementations, while potentially benefiting from a reordering of pairwise evaluations. Unfortunately even when optimally sequenced, pairwise evaluation must store intermediate values, limiting use of scarce GPU global memory.

Source code for this work including implementations and benchmark tests is available at https://github.com/Areustle/ParallelTNNLayers.

Bibliography

[1] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M. Kakade, and Matus Telgarsky. Tensor decompositions for learning latent variable models. *Journal of Machine Learning Research*, 15:2773–2832, 2014.

[2] Alexander A. Auer, Gerald Baumgartner, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Lu, Marcel Nooijen, Russell Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[3] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

[4] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv:1410.0759 [cs]*, October 2014. arXiv: 1410.0759.

[5] Andrzej Cichocki, Namgil Lee, Ivan V Oseledets, Anh Huy Phan, Qibin Zhao, and D Mandic. Low-rank tensor networks for dimensionality reduction and large-scale optimization problems: Perspectives and challenges part 1. *arXiv preprint arXiv:1609.00893*, 2016.

[6] Andrzej Cichocki, Namgil Lee, Ivan V. Oseledets, Anh Huy Phan, Qibin Zhao, and Danilo P. Mandic. Low-rank tensor networks for dimensionality reduction and large-scale optimization problems: Perspectives and challenges PART 1. *CoRR*, abs/1609.00893, 2016.

[7] Andrzej Cichocki, Anh-Huy Phan, Qibin Zhao, Namgil Lee, Ivan Oseledets, Masashi Sugiyama, Danilo P Mandic, et al. Tensor networks for dimensionality reduction and large-scale optimization: Part 2 applications and future perspectives. *Foundations and Trends® in Machine Learning*, 9(6):431–673, 2017.

[8] Nvidia Corporation. Nvidia Turing GPU Architecture. https://nvidia.com/en-us/geforce/news/geforce-rtx-20-series-turing-architecture-whitepaper, 2018. [Online; accessed 09-Sept-2019].

[9] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014.

[10] Martín Abadi et al. Dean, Tucker, Yu, and TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[11] Lars Grasedyck, Daniel Kressner, and Christine Tobler. A literature survey of low-rank tensor approximation techniques. *GAMM-Mitteilungen*, 36(1):53–78, 2013.

[12] Majid Janzamin, Hanie Sedghi, and Anima Anandkumar. Generalization bounds for neural networks through tensor factorization. *CoRR*, abs/1506.08473, 2015.

[13] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noel Pouchet, Atanas Rountev, and P. Sadayappan. A Code Generator for High-Performance Tensor Contractions on GPUs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 85–95, Washington, DC, USA, February 2019. IEEE.

[14] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[15] T. Kolda and B. Bader. Tensor Decompositions and Applications. *SIAM Review*, 51(3):455–500, August 2009.

[16] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[17] Jean Kossaifi, Aran Khanna, Zachary Lipton, Tommaso Furlanello, and Anima Anandkumar. Tensor contraction layers for parsimonious deep nets. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2017 IEEE Conference on*, pages 1940–1946. IEEE, 2017.

[18] Jean Kossaifi, Zachary C. Lipton, Aran Khanna, Tommaso Furlanello, and Anima Anandkumar. Tensor regression networks. *CoRR*, abs/1707.08308, 2017.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[20] Chi-Chung Lam, P Sadayappan, and Rephael Wenger. On optimizing a class of multidimensional loops with reductions for parallel execution. *Parallel Processing Letters*, 7(2):157–168, 1997.

[21] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[22] Jingling Li, Yanchao Sun, Jiahao Su, Taiji Suzuki, and Furong Huang. Understanding Generalization in Deep Learning via Tensor Methods.

[23] Wenjing Ma, Sriram Krishnamoorthy, Oreste Villa, and Karol Kowalski. GPU-Based Implementations of the Noniterative Regularized-CCSD(T) Corrections: Applications to Strongly Correlated Systems. *Journal of Chemical Theory and Computation*, 7(5):1316–1327, May 2011.

[24] Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry P. Vetrov. Tensorizing neural networks. *CoRR*, abs/1509.06569, 2015.

[25] Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

[26] I. V. Oseledets. Tensor-train decomposition. *SIAM J. Sci. Comput.*, 33(5):2295–2317, September 2011.

[27] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[28] Robert N. C. Pfeifer, Jutho Haegeman, and Frank Verstraete. Faster identification of optimal contraction sequences for tensor networks. *Physical Review E*, 90(3), September 2014. arXiv: 1304.6112.

[29] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka. Tensor Contractions with Extended BLAS Kernels on CPU and GPU. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 193–202, 2016.

[30] Paul Springer and Paolo Bientinesi. Design of a high-performance GEMM-like Tensor-Tensor Multiplication. *CoRR*, 2016.

[31] Jiahao Su Su, Jingling Li Li, Bobby Bhattacharjee, and Furong Huang. Tensorial neural networks: Generalization of neural networks and application to model compression. *CoRR*, abs/1805.10352, 2018.