TeMCO: Tensor Memory Compiler Optimization across Tensor Decompositions in Deep Learning Inference

Seungbin Song
Yonsei University
Seoul, Republic of Korea
seungbin@yonsei.ac.kr

Ju Min Lee
Yonsei University
Seoul, Republic of Korea
jumin@yonsei.ac.kr

Haeeun Jeong
Yonsei University
Seoul, Republic of Korea
haeeun.jeong@yonsei.ac.kr

Hyunho Kwon
Yonsei University
Seoul, Republic of Korea
hyunho@yonsei.ac.kr

Shinnung Jeong
Yonsei University
Seoul, Republic of Korea
shin0403@yonsei.ac.kr

Jaeho Lee
Yonsei University
Seoul, Republic of Korea
jaeho@yonsei.ac.kr

Hanjun Kim
Yonsei University
Seoul, Republic of Korea
hanjun@yonsei.ac.kr

ABSTRACT

Since the increasing complexity of deep learning models, tensor decomposition is one of the promising solutions that reduce computational complexity in deep learning models. By decomposing a convolution layer with a large weight tensor into multiple layers with smaller weight tensors, tensor decomposition can reduce the number of operations and weight memory spaces. However, existing tensor decomposition schemes face difficulties in reducing peak memory usage of the entire inference. The decomposed layers produce the reduced-sized tensors during inference, but the reduced tensors should be restored to their original sizes due to skip connections and non-decomposed activation layers between the decomposed layers. To reduce the peak memory usage of the end-to-end inference of the decomposed models, this work proposes a new tensor memory optimization scheme and its prototype compiler, called TeMCO. TeMCO replaces the original internal tensors used in the skip connections with reduced internal tensors derived by the decomposed layers. In addition, TeMCO fuses the decomposed layers and the non-decomposed activation layer and thus keeps the reduced internal tensors produced without restoring them. Thanks to the optimizations, this work reduces memory usage of internal tensors by 75.7% for 10 models of 5 deep learning architectures.

CCS CONCEPTS

• **Software and its engineering → Compilers**; • **Computing methodologies → Factorization methods**; **Neural networks**.

KEYWORDS

Tensor Decomposition, Deep Learning, Compilers

ACM Reference Format:
Seungbin Song, Ju Min Lee, Haeeun Jeong, Hyunho Kwon, Shinnung Jeong, Jaeho Lee, and Hanjun Kim. 2024. TeMCO: Tensor Memory Compiler Optimization across Tensor Decompositions in Deep Learning Inference. In *Proceedings of the 53rd International Conference on Parallel Processing (ICPP'24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3673038.3673048

ICPP '24, August 12–15, 2024, Gotland, Sweden

1 INTRODUCTION

As deep learning models become increasingly complex, model compression techniques have been devised to reduce the computational overheads of deep learning inferences. The model compression schemes, such as pruning [9, 22, 23], quantization [12, 21, 41, 49], knowledge distillation [6, 25, 40, 47], and tensor decomposition [10, 16, 28, 29, 39, 42–46], reduce the size and computational complexity of models while maintaining a high level of accuracy.

Existing works [16, 29, 42–45] employ tensor decomposition techniques in deep learning models to reduce the number of operations needed to perform convolutions with marginal accuracy drop. Tensor decomposition [10, 28, 39, 46] reduces computational complexity by using mathematical techniques, decomposing a convolution into a composition of smaller convolutions. Since tensor decomposition factorizes a large-weight tensor of an original convolution into several smaller-weight tensors, a convolution sequence of these tensors produces an output that is approximately equivalent to the output of the original convolution. Consequently, the existing works replace convolution layers in the original model with decomposed convolution sequences, which generate the same output tensor shape as the original convolutions.

Although tensor decomposition effectively reduces the number of operations, existing techniques miss a crucial opportunity to reduce peak memory usage. Peak memory usage is critical since it can affect the restriction of running a deep learning model fitting into the DRAM capacity of a GPU and can hamper researchers' flexibility to study machine learning algorithms [32]. Decomposed convolution sequences generated by tensor decomposition can temporarily produce reduced-sized tensors inside. However, because the internal tensor that holds the input and output of the decomposed convolution sequences retains the same size as in the original convolution, the peak memory usage remains unchanged. Therefore, tensor decomposition cannot reduce peak memory usage due to the internal tensors.

Furthermore, reducing the peak memory usage of a decomposed deep learning model becomes a more challenging problem when considering deep learning models with skip connections [8, 11, 34]. The skip connections preserve the output of a block and are later used as input for distant layers. In this case, the reduced tensors should be restored to their original sizes for the skip connections and reserved in memory. Therefore, models with skip connections occupy significant memory space to preserve these tensors during

inference, resulting in a decomposed model exhibiting a similar memory usage trend as the original model.

To effectively reduce memory usage in the inferences of tensor-decomposed models, this work proposes TeMCO: **Te**nsor **M**emory **C**ompiler **O**ptimization across tensor decompositions in deep learning inference. We make a key finding that compiler optimization can effectively transform decomposed models by substituting reduced tensors for internal tensors. First, TeMCO replaces the uses of the original tensors with the reduced tensors in the skip connections, copying the restore layers at the end of the connections. Second, TeMCO fuses non-decomposed activation layers and decomposed convolution layers. The fused layers do not allocate the input and output internal tensors and perform computations only with the reduced tensors. Consequently, these compiler transformations allow the reduced tensors in the decomposed sequences to be fully utilized without restoration throughout the whole inference.

This work evaluates the prototype TeMCO compiler using 10 models of 5 deep learning architectures, which includes image classification with convolution-based deep learning models [17, 37] and the models with skip connections [8, 11] and image segmentation with UNet [34]. This work compares the performance with baseline Tucker-decomposed models [39], applying optimizations of TeMCO. The evaluation results show that TeMCO successfully reduces peak memory usage of internal tensors by 75.7%, with from 1.08× to 1.70× inference time overheads in batch sizes. Furthermore, the TeMCO's optimizations do not drop the accuracy of the decomposed models.

Contributions of this work are:

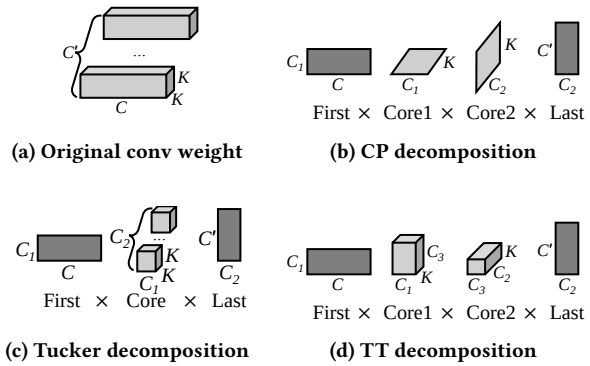
- TeMCO's compiler optimization scheme that reduces peak memory usage of internal tensors in tensor-decomposed model inference,
- skip connection optimization that replaces tensors in skip connections with the precedent reduced tensors,
- and layer fusion that fuses non-decomposed activation layers with decomposed convolution layers to utilize reduced tensors throughout the model inference.

2 BACKGROUND & MOTIVATION

In this section, we introduce more details of widely-used tensor decomposition techniques (Section 2.1) and provide a more precise analysis of peak memory usage in the decomposed deep learning models (Section 2.2). Based on this analysis, we propose motivations and ideas for reducing peak memory usage in the decomposed models (Section 2.3).

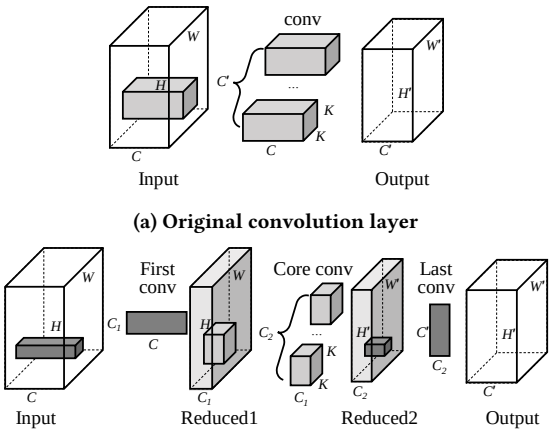
2.1 Tensor Decomposition

Tensor decomposition, such as Canonical Polyadic (CP) decomposition [10], Tucker decomposition [39], and Tensor-Train (TT) decomposition [28] decomposes a weight tensor of a convolution layer into low-ranked factor matrices and core weights. Figure 1 shows how each tensor decompositions decompose a tensor into small tensors. The first and the last weight tensors are 2D factor matrices, and the shapes of the core weight tensors depend on the tensor decomposition methods. The multiplication of the decomposed weight tensors approximates the original weight tensor.



(a) Original conv weight (b) CP decomposition (c) Tucker decomposition (d) TT decomposition

Figure 1: Tensor decomposition types



(a) Original convolution layer

(b) Decomposed convolution sequence

Figure 2: A tensor decomposition example on a convolution layer

We can construct a *decomposed convolution sequence* that mimics the original convolution layer using these decomposed tensors. Figure 2 shows the original convolution layer and the decomposed convolution sequence constructed by tensor decomposition. The first and the last tensors in Figure 1 become the weights of the 1×1 convolution layers in Figure 2b, respectively, and the core tensor(s) in Figure 1 are the weight(s) of the core convolution layer(s) in Figure 2. Note that the core convolutions are various in the types of tensor decomposition, but the first and last convolution layers are consistent among tensor decomposition methods.

The channel C of the input tensor reduces to C_1 after being convoluted by the first decomposed convolution layer. Then, the core convolutions perform a small-sized convolution, generating a reduced internal tensor with channel size C_2. The last decomposed convolution layer restores the channels of the reduced internal tensor from C_2 to C', which is the channel size of the original output. Here, the first and the last convolution layers play a key role in reducing and restoring internal tensors. In the remainder of the paper, we alias the *first convolution layer* as *fconv*, the *last convolution layer* as *lconv*, and call the internal tensors within

a decomposition sequence (Reduced1, Reduced2 in Figure 2b) as *reduced tensors*.

Previous work [42, 44, 45] shows that tensor decomposition reduces FLOPS and inference time of deep learning models. However, their scheme does not reduce the peak memory usage by internal tensors. The following analysis shows the limitations of the tensor decomposition schemes on the point of memory efficiency.

2.2 Peak Memory Usage Analysis

As the initial step of analyzing peak memory usage overheads, this work analyzes the peak memory usage of tensor-decomposed sequences in Figure 3. The peak memory usage of a model inference consists of the memory usage by two types of tensors: *weight tensors* and *internal tensors*. Weight tensors save a model's weight parameters. On the other hand, internal tensors hold intermediate results, such as feature maps and skip connections throughout the model inference.

For the weight tensors, deep learning frameworks [1, 30] load the whole weight tensors at the beginning of model inference. The size of the weight tensors in the two convolution layers (Figure 3a) is as follows in Equation (1):

$$CC'K^2 + C'C''K'^2 \tag{1}$$

Here, tensor decomposition can reduce the memory usage of the weight tensors by setting the channel sizes of reduced tensors smaller than the sizes of the original tensors. The size of weight tensors in the decomposed convolution sequences (Figure 3b) is as follows in Equation (2):

$$CC_1 + C_1C_2K^2 + C_2C' + C'C_3 + C_3C_4K^2 + C_4C'' \tag{2}$$

On the point of internal tensors, the deep learning frameworks [1, 30] dynamically allocate and free the memory spaces of the internal tensors. In other words, the frameworks allocate only the internal tensors required by the currently running layer and free the tensors that will not be used in future inference. Therefore, the peak memory usage of the internal tensors can be calculated as the maximum of each layer's input plus output tensor sizes. The peak memory usage by the internal tensors in the two convolution layers (Figure 3a) is as follows in Equation (3):

$$MAX(CHW + C'H'W',\ 2C'H'W',\ C'H'W' + C''H''W'') \tag{3}$$

However, tensor decompositions cannot reduce the peak memory usage of the internal tensors because of the non-decomposed activation layer. The peak memory usage by internal tensors in the decomposed convolution sequences (Figure 3b) is as follows in Equation (4):

$$\begin{aligned} MAX(&CHW + C_1HW,\ C_1HW + C_2H'W',\ C_2H'W' + C'H'W',\\ &2C'H'W',\ C'H'W' + C_3H'W',\ C_3H'W' + C_4H''W'',\\ &C_4H''W'' + C''H''W'') \end{aligned} \tag{4}$$

Tensor decomposition sets the channel sizes of the reduced internal tensors (C_1 to C_4) are smaller than the sizes of the original tensors (C to C''). Therefore, the Equation (4) is reduced to $2C'H'W'$. The peak memory usage by internal tensors in Equation (4) is similar to the usage in Equation (3), and the peak memory usage $2C'H'W'$ comes from the internal tensors (Output1 and Input2) of the activation layer. Therefore, the internal tensors used by the non-decomposed activation layer take a large portion of the peak memory usage of inference in Figure 3b.

2.3 Motivation

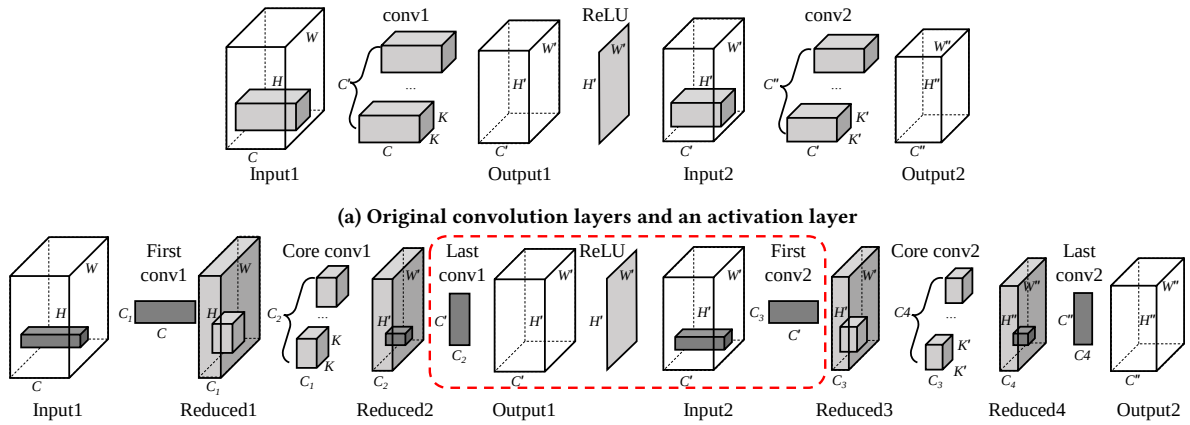
The existing tensor-decomposed model misses an opportunity to reduce the peak memory usage of the internal tensor, even though tensor decomposition generates tensors with reduced-sized channels in decomposed convolution sequences. Due to *skip connections* and *activation layers*, the reduced tensors should be restored to their original channel sizes. To demonstrate the effect of skip connections and activation layers, Figure 4 shows the memory usage of the internal tensor in both the original and the decomposed models using Tucker decomposition [39]. Figure 4a and Figure 4b depict the memory usage during 4-batch inference with UNet [34] and VGG-16 [37], respectively.

In Figure 4a, the memory usage of skip connections takes 76.2% of the peak memory usage by internal tensors in the UNet-decomposed model. The UNet structure is an hourglass shape with skip connections horizontally connecting the downsampling and upsampling blocks. In the decomposed model inference, decomposed convolution sequences in the downsampling blocks restore reduced tensors to their original sizes, and the model leaves the original tensors in the skip connections. These original tensors reside until the upsampling blocks consume them. Therefore, memory usage by skip connections in the decomposed model is similar to that of the original model.

On the other hand, the memory usage by internal tensors peaks at the computation of non-decomposed activation layers in VGG-16, as shown in Figure 4b. VGG has a linear sequence of convolution, activation, and pooling layers. Decomposed convolution sequences in the VGG-decomposed model reduce the internal tensor sizes when performing core convolutions. However, the sequences soon restore the reduced tensors to their original sizes to be processed in non-decomposed activation layers. Therefore, the peak memory usage by non-decomposed layers in the decomposed model is similar to the peak in the original model.

To solve this problem, a new optimization is needed to transform a decomposed deep-learning model to use only reduced tensors as depicted in Figure 5. Figure 5 displays the optimized convolution sequence consisting of reduced tensors, achieved by fusing the *lconv 1* and ReLU with the *fconv 2*, and removing Output1 and Input2 from Figure 3b. Since Figure 3b did not contain the skip connections, the solution needs more detailed steps about skip connection.

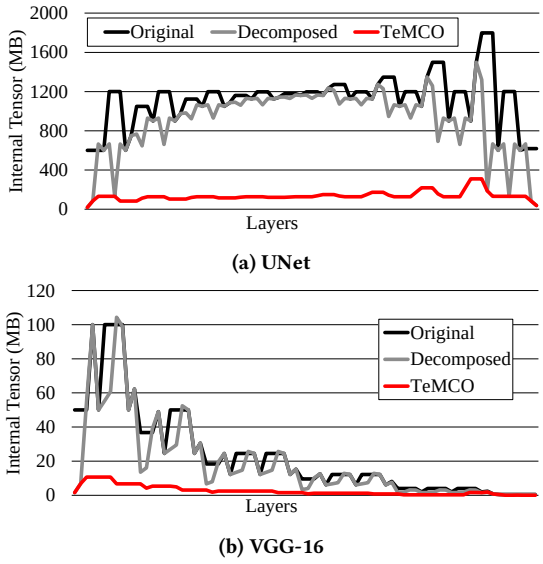
In detail, this work proposes TeMCO consisting of two important tensor memory compiler optimization. First, TeMCO suggests skip connection optimization. Initially, TeMCO examines the program dependence graph of the internal tensors in the skip connections and identifies predecessor reduced tensors. Then TeMCO substitutes the original tensors with the reduced tensors within the skip connections, adding restorations of the reduced tensors at the end of skip connections. Second, TeMCO fuses non-decomposed activation layer with two decomposed convolution layers as same



(a) Original convolution layers and an activation layer

(b) Decomposed convolution sequences and an activation layer

Figure 3: A tensor decomposition example on convolution layers alongside non-decomposed activation layer



(a) UNet

(b) VGG-16

Figure 4: The memory usage by internal tensors of UNet and VGG-16 with batch size 4, RTX 4090

as Figure 5, and the fused layer performs computations only with reduced tensors. The restorations of skip connections can also be hidden in the fused layers by applying further transformations.

Note that TeMCO transforms the decomposed model while preserving the original semantics. Therefore, the optimizations maintain the accuracy of the decomposed models. For example, previous work [42, 45] proposes training algorithms for decomposed models with high accuracy. If a tensor decomposition scheme provides a pre-trained decomposed model, TeMCO can reduce the peak memory usage of inference while preserving its model's accuracy.

3 DESIGN AND IMPLEMENTATION

This work proposes TeMCO that optimizes tensor-decomposed models to reduce memory usage by internal tensors. TeMCO's optimization procedure (Figure 6) consists of two main components. The first component is *skip connection optimization* (Section 3.1) that replaces skip connections with reduced tensors. The second component is *activation layer fusion* (Section 3.2) that fuses non-decomposed activation layers with neighboring *lconv* and *fconv*. TeMCO further applies *layer transformations* (Section 3.3) that optimize concatenations and *lconv* layers in skip connections.

3.1 Skip Connection Optimization

TeMCO optimizes skip connections to reduce the peak memory usage of the decomposed model. In Algorithm 1, skip connection optimization consists of the following steps: 1) find skip connections with tensor liveness analysis results, 2) identify predecessor reduced tensors and restore layers, 3) calculate computation and memory overheads of copying restore layers, 4) and move the copied layers before the uses of the skip connection and replace the skip connection with the reduced tensor.

First, to find skip connections in a model, TeMCO analyzes layers of the model from start to end and calculates the liveness of the tensors (Lines 11 to 16) in Algorithm 1. The tensor liveness analyzer records the first definition of a tensor and the last use of the tensor. With this *begin* and *end* information, the compiler measures the lifespan of the tensor (Line 18) and identifies skip connections. In Figure 7a, TeMCO recognizes tensor `b` as a skip connection.

With the skip connections, TeMCO finds predecessor reduced tensors and restore layers in Algorithm 2. The `FindReduced` function (Lines 17 to 33) recursively traverses predecessors of a skip connection and returns the restore layers in order. The order of the layers is determined with the `Compare` function (Lines 8 to 9). Since the execution order affects the peak memory usage of internal tensors, `Compare`(a, b) calculates the peak memory usage of $a \rightarrow b$ and $b \rightarrow a$ sequences and compares them. This comparison is sub-optimal in the two predecessors. Previous work [19, 31, 50] proposes execution scheduling algorithms to reduce peak memory usage,

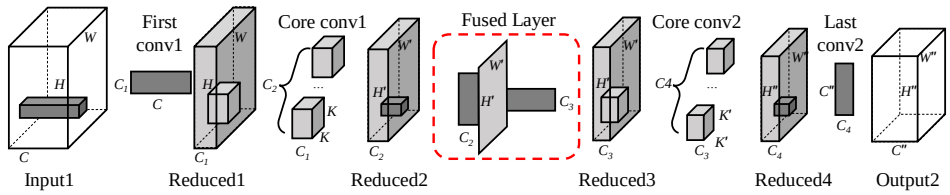


Figure 5: TeMCO optimized convolution sequence result with activation layer fusion

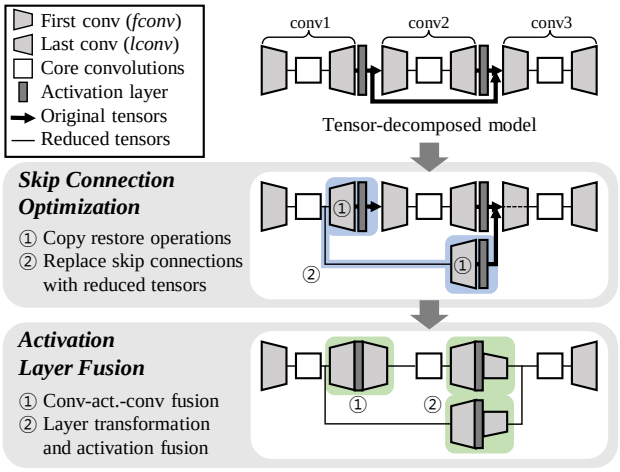


Figure 6: Overall structure of TeMCO

```
a1  = conv1.fconv(in)
a2  = conv1.cconv(a1)
a   = conv1.lconv(a2)
b   = relu(a)
c1  = conv2.fconv(b)
c2  = conv2.cconv(c1)
c   = conv2.lconv(c2)
d   = relu(c)
e   = concat(d,b)
f1  = conv3.fconv(e)
f2  = conv3.cconv(f1)
f   = conv3.lconv(f2)
out = relu(f)
return out
```

(a) Decomposed model

```
a1  = conv1.fconv(in)
a2  = conv1.cconv(a1)
a   = conv1.lconv(a2)
b   = relu(a)
c1  = conv2.fconv(b)
c2  = conv2.cconv(c1)
c   = conv2.lconv(c2)
d   = relu(c)
a'  = conv1.lconv(a2)
b'  = relu(a')
e   = concat(d,b')
f1  = conv3.fconv(e)
f2  = conv3.cconv(f1)
f   = conv3.lconv(f2)
out = relu(f)
return out
```

(b) Skip connection opt. model

Figure 7: Skip connection optimization example

and we will develop our scheduling algorithm by augmenting them. `FindReduced` returns *lconv* as a leaf node using `IsLConv` (Lines 1 to 7) that returns whether the layer type is a convolution with 1×1 kernel and increases the channel size. In Figure 7a, `FindReduced`(b) returns [`b=relu(a)`, `a=conv1.lconv(a2)`].

Back to Algorithm 1, TeMCO calculates computation and memory overheads of copying restore layers. The `Overhead` function (Lines 1 to 9) calculates FLOPS and peak memory usage of restore

Algorithm 1: Skip connection optimization

Input : An ordered tensor node list L in SSA form
Output : Skip-connection-optimized tensor node list O

```
// PRED(v,G): predecessor list of v in G
// SUCC(v,G): successor list of v in G
// DISTANCE(a,b): distance of two node a, b
// SIZE(v): tensor size of v by shape inference
// FLOPS(v): FLOPS of v by calculating weight sizes
1  Function Overhead(n,l):
2      c ← 0, m ← SIZE(n)
3      for e in l.list do
4          c ← c + FLOPS(e)
5      end
6      for p in PRED(n,G) do
7          m ← m + SIZE(p)
8      end
9      return c ≤ COMPUTE_THRESHOLD and l.peak ≤ m
10 live ← {}, O ← L, G ← PDG(L) // program dependence graph
   // Tensor liveness analysis
11 for n in L do
12     live[n].begin ← n
13     for p in PRED(n, G) do
14         live[p].end ← n
15     end
16 end
17 for n in L do
       // Identify skip connections with distance
18     d ← DISTANCE(live[n].begin, live[n].end)
19     if d >DISTANCE_THRESHOLD then
           // Find reduced tensors and restore operations
20         l ← FindReduced(n, G)
           // Calculate overheads
21         if Overhead(n,l) then
22             for s in SUCC(n) do
                   // Insert operations l before s
23                 O ← InsertBefore(O,s,COPY(l.list))
24             end
25 end
```

layers. If the length of the restore layer list is long and requires many layers to restore the skip connection, or the peak memory usage of copying the layers is much higher than not copying the layers, the algorithm decides not to copy the layers. Currently, the computation threshold is set to FLOPS of the corresponding parts of the original model without decomposition.

Algorithm 2: Find restore layers and reduced tensors

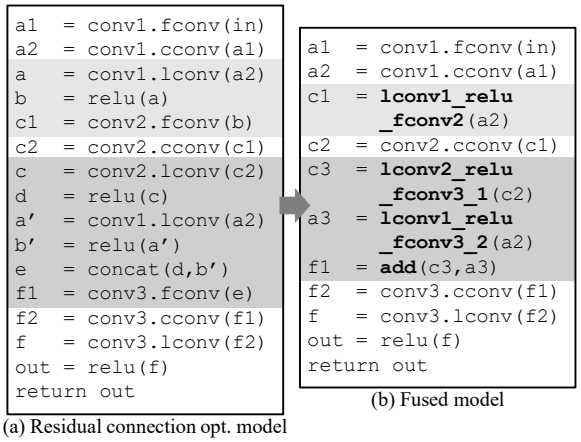
Input : A tensor node v, a program dependence graph G
Output : Results res of reduced tensor node list, size, peak memory usage

```
1  Function IsLConv(v):
2      op ← OP(v) // operator of v
3      if op.type = conv then
4          if op.kernel_size = op.stride = (1, 1) then
5              if op.out_channels > op.in_channels then
6                  return True
7      return False
8  Function Compare(a, b):
9      return a.size + b.peak < b.size + a.peak
10 Function Peak(l, v):
11     peak ← 0, resided ← 0
12     for e in l do
13         peak ← MAX(resided + e.peak, peak)
14         resided ← resided + e.size
15     end
16     return ← MAX(resided + SIZE(v), peak)
17 Function FindReduced(v, G):
18     if IsLConv(v) then
19         res.list ← [v]
20         res.size ← SIZE(v)
21         res.peak ← SIZE(v) + SIZE(PRED(v, G)[0])
22         return res
23     else
24         predList ← [ ]
25         for n in PRED(v, G) do
26             predList ← predList ∪ FindReduced(n,G)
27         end
28         orderedList ← ORDER(Compare, predList)
29         res.list ← CONCAT(e.list for e in orderedList) ∪ [v]
30         res.size ← SIZE(v)
31         res.peak ← Peak(orderedList, v)
32         return res
33     end
```

After evaluating computation and memory overheads, TeMCO copies the restore layers and inserts them before the use of the skip connections (Line 23). In Figure 7, the compiler copies `a = conv1.lconv(a2)` and `b = relu(a)` and inserts before `e` (renaming `a` to `a'` and `b` to `b'`). Finally, the compiler replaces the skip connection of tensor `b` with the compressed tensor `a2`, whose channel size is smaller than that of the original skip connection. As a result, the skip connection optimization reduces peak memory usage.

3.2 Activation Layer Fusion

Although the compiler optimizes the skip connections, the original tensors still reside in the model (Figure 8a), which take the large portion of peak memory usage as described in Section 2.2. Therefore, TeMCO fuses a non-decomposed activation layer with neighboring decomposed convolutions to reduce peak memory usage of internal tensors. To clarify our definition, activation layer fusion means the fusion of a *lconv*-activation-*fconv* sequence, not



(a) Residual connection opt. model (b) Fused model

Figure 8: Activation fusion example

the fusion [3, 27] of a convolution-activation pair, nor the activation data compression [5, 14] in deep learning training.

The fused kernel does not allocate output and input internal tensors (Output1 and Input2 in Figure 3b) and performs the computations with the reduced tensors only (Figure 5). In Figure 8, the compiler fuses `conv1.lconv`, `relu`, and `conv2.fconv` into `lconv1_relu_fconv2`. As a result, the fused kernel does not use the original tensors (`a`, `b`) and only uses the reduced tensors (`a2`, `c1`).

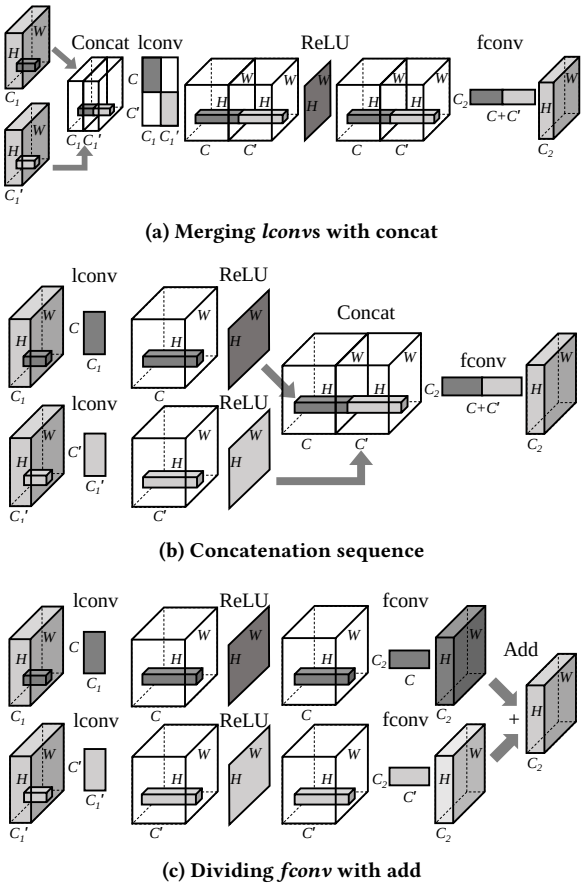
In between *lconv* and *fconv*, there are non-decomposed activations like ReLU [2], SiLU [7], and pooling layers. These non-decomposed layers are element-wise and perform their layers on all elements in tensors. Because *fconv* conducts convolutions on each resulting channel and accumulates the multiplied results, all the individual activation-applied values are required to produce correct results. Therefore, a compiler cannot reorder *lconv*-activation-*fconv* sequences nor omit one of these layers. To fuse these sequences, this work implements GPU-parallelized fused kernels with CUDA.

Listing 1 briefly describes a fused kernel implementation of *lconv*-relu-*fconv* and *lconv*-relu-pool-*fconv* in Figure 5. This work implements the fused kernel to perform parallelized operations over C_3, H, and W dimensions. Here, *lconv* and *fconv* are channel-wise 1×1 convolutions, which restores and reduces each channel, respectively. The fused kernel first performs nested and tiled convolutions of *lconv* on input tensor `IN` and weight tensor `W1`, resulting in the output channel C'. With the accumulated result `v1`, the fused kernel second applies activation `relu` and saves the result into a tile `tile`. If a pooling layer is included in the fusion, the fused kernel applies the pooling `pool` over H and W dimensions in the tile `tile`. After applying activation and pooling, the fused kernel performs 1×1 convolution over the channel C' and updates the result `v3` to the output tensor `OUT`. Listing 1 illustrates a fused kernel of both with and without pooling layers, but the actual implementation has an individual kernel of each case.

The fused kernel does not allocate the whole size of the original tensor Output1 and Input2; instead, the kernel uses tiled buffers `tile` in the shared memory. This means applying activation layer fusion can skip allocating the original internal tensors and reduce peak memory usage.

```
Tensor fused_kernel(
        Tensor IN,W1,B1,W2,B2,int p=1){
/* T: tile size
   bx,by,bz: block idx of x,y,z dimension
   tx,ty,tz: tile idx of x,y,z dimension
   p: kernel size of pooling layer,
      p<2 for no pooling  */
c2 = bx*T + tx; //index of C' with x
h  = by*T + ty; //index of H' with y
w  = bz*T + tz; //index of W' with z
v3 = B2[c2]; //load bias
for(i=0; i<C/T; i++){
  v1 = B1[c]; //load bias
  for(j=0; j<C1/T; j++){
    //load IN and W1 into tile
    tileIN[tx][ty][tz] = IN[j*T+tx][h][w];
    tileW1[tx][ty] = W1[i*T+tx][j*T+ty];
    __syncthreads();
    //perform lconv
    for(k=0; k<T; k++)
     v1 += tileIN[k][ty][tz]*tileW1[tx][k];
    __syncthreads();
  }
  //perform activation and load W2
  tile[tx][ty][tz] = relu(v1);
  tileW2[tx][ty] = W2[c2][i*T+ty];
  __syncthreads();
  //perform pooling
  if(p>=2){
    v2 = pool(tile[tx][p*ty][p*tz],...,
              tile[tx][p*ty+p-1][p*tz+p-1]);
    __syncthreads();
    tile[tx][ty][tz] = v2;
    __syncthreads();
  }
  //perform fconv
  for(l=0; l<T; l++)
    v3 += tile[l][ty][tz]*tileW2[tx][l];
  __syncthreads();
}
//update result to OUT
OUT[c2][h][w] = v3;
return OUT;
}
```

Listing 1: Pseudo code of fused kernels



(a) Merging *lconv*s with concat

(b) Concatenation sequence

(c) Dividing *fconv* with add

Figure 9: Transformations for a concatenation layer

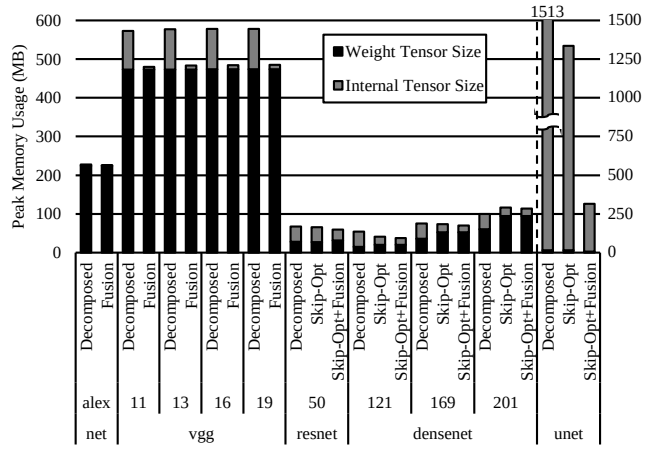
3.3 Layer Transformation

TeMCO optimizes the layers alongside a concatenation layer and an add layer that merges skip connections by applying concatenation layer transformation. deep learning models with skip connections [11, 34] use concatenation layers to accumulate skip connections, The concatenation layer in the original decomposed model concatenates the decompressed internal tensors in the channel dimension (Figure 9b). Because the concatenated original tensors take a large portion of the peak memory usage, the compiler transforms concatenation-*fconv* sequences to apply activation layer fusion.

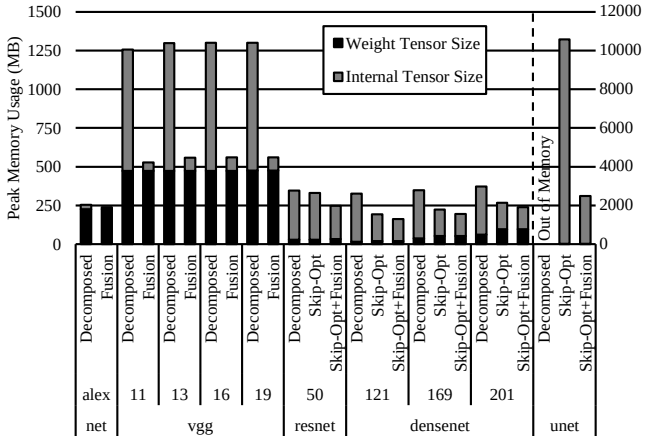
In Figure 9b, a *fconv* layer is a 1×1 convolution that performs convolution for each channel row with length $C + C'$ in the concatenated tensor. The concatenation layer concatenates two input tensors over the channel dimension. We can divide the channel row vector and the weight matrix regarding the channel C and C'. If we apply convolutions for each divided channel row with the corresponding weights and add the results (Figure 9c), the result is the same as applying convolution over the original channel row with the original weights (Figure 9b). In other words, the compiler safely transforms the concatenation-convolution sequence into the convolution-add sequence. After transforming the concatenation, the compiler can apply activation layer fusion for each sequences, generating two fused kernels.

On the other hand, the concatenation-convolution sequence can be transformed into Figure 9a by merging *lconv*s and applying concatenation on input reduced tensors. To generate weights of merged *lconv*, the compiler locates the weights in the diagonal positions and adds zero padding on the rest of the weight tensor. This transformation is applicable when the two sequences have the same activations. By applying this transformation, weight sizes increase, but the compiler can make a single *lconv*-activation-*fconv* sequence. This transformation reduces the computation overheads of calling numerous fused kernels.

ResNet and ResNet-based models [8, 33] use add layers when accumulating skip connections on ResNet blocks. An add sequence in the form of Figure 9c can be transformed to Figure 9a. If the compiler transforms the add sequence to the merged *lconv* sequence, the compiler can apply activation layer fusion on the sequence. By adopting these layer transformations, this work extends the applicability of activation layer fusion and thus reduces the peak memory usage for tensor-decomposed models.

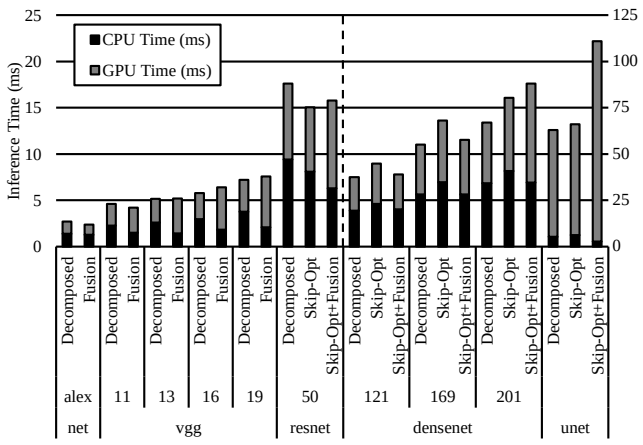


(a) Batch size 4

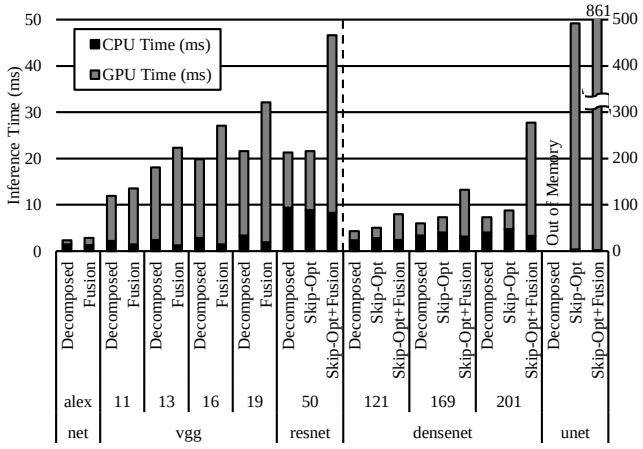


(b) Batch size 32

Figure 10: Peak memory usage of the 10 models' inferences



(a) Batch size 4



(b) Batch size 32

Figure 11: End-to-end inference time of the 10 models

4 EVALUATION

4.1 Experimental Setup

This work implements a prototype compiler of TeMCO with PyTorch 2.2 [30] and fused kernels (Section 3.2) with CUDA. As evaluation hardware, this work uses Ryzen 9 3950X with 128 GB memory and RTX 4090 with 24 GB GPU memory. Benchmark sets include image classification of AlexNet [17], VGG [37], ResNet [8] and DenseNet [11] and image segmentation of UNet [34]. Image classification uses the ILSVRC 2012 [35] dataset, and image segmentation uses the Carvana [36] dataset. This work applies Tucker decomposition [39] to the 10 models with a decomposition ratio of 0.1 and uses the decomposed models as a baseline denoted as **Decomposed** in the following graphs. Since AlexNet and VGG do not have skip connections, this work only applies activation layer fusion for the models represented as **Fusion**. This work applies both skip connection optimization and layer fusion to models such as ResNet, DenseNet, and UNet, which have skip connections, and evaluates the effect of skip connection optimization without and with layer fusion, indicated as **Skip-Opt** and **Skip-Opt+Fusion**, respectively.

4.2 Peak Memory Usage

This work measures the peak memory usage of end-to-end inference of 10 models with 4-batch inference. Figure 10 shows the peak memory usage by weight and internal tensors.

Compared to the original model, this work reduces memory usage of internal tensors by 75.7% in geomean. For AlexNet and VGG, TeMCO fully benefits from activation layer fusion, reducing memory usage of internal tensors by 49.4% and 90.7%, respectively. For ResNet, TeMCO reduces memory usage of internal tensors by 30.7% because ResNet has deep-depth skip connections, inducing a high amount of computations to copy restore layers. To alleviate this effect, skip connection optimization selectively optimizes the skip connections based on the overhead comparison. On the other hand, DenseNet reduces 54.0% of internal tensor size because it has numerous skip connections. UNet has an hourglass structure with skip connections, so TeMCO reduces 79.3% the memory usage of internal tensors. For DenseNet and UNet, the skip connection optimization copies the restore layers, the layer transformation merges the *lconv*, and the layer fusion finally generates a fused kernel. Merging *lconv* requires more memory space for weights but

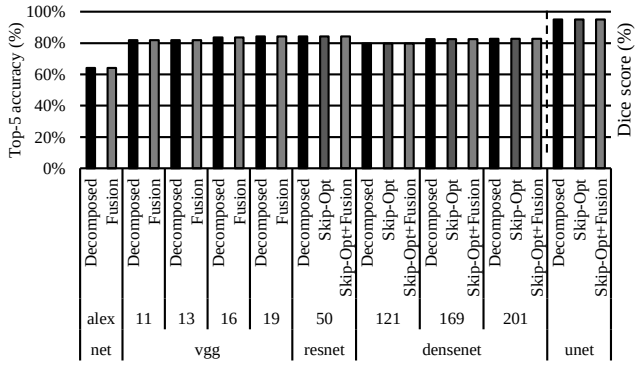


Figure 12: Accuracy of 10 model's inference

reduces the total peak memory usage by reducing the number of fused kernels. Consequently, TeMCO allows the reduction of the internal tensor sizes and avoids the out-of-memory problem for complex deep learning architectures.

4.3 Inference Time

Figure 11 shows the end-to-end inference of the 10 models with CPU and GPU time. The inference time of the optimized models is 1.08× and 1.70× longer than the decomposed models in batch size 4 and 32, respectively.

For AlexNet and VGG, activation layer fusion reduces CPU overheads because the number of layer calls is reduced. Since the fused kernel performs tiled operations for large inputs, the GPU overheads of the fused kernels increase as the batch size increases. As the depth of the model increases, the model has more fused kernels, causing more computational overheads. For DenseNet and UNet, skip connection optimization causes CPU overheads because it copies restore operations, increasing the number of layer calls. Layer fusion and transformation reduce CPU overheads by reducing the number of layer calls, but GPU overheads increase as batch size and model depth increase.

4.4 Accuracy

Figure 12 shows top-5 accuracy for AlexNet, VGG, ResNet, DenseNet, and dice score [36] for UNet. The compiler optimizations of TeMCO do not drop the accuracy of decomposed models because they preserve the original semantics of the models. This work applies decompositions with a decomposition ratio of 0.1 and performs direct training for this evaluation. Previous work [42, 45] proposes training schemes for decomposed models for high accuracy. If a tensor decomposition scheme provides a pre-trained decomposed model, TeMCO can reduce the peak memory usage by internal tensors while preserving their accuracy.

5 RELATED WORK

Tensor decomposition: Most tensor decomposition methods on deep learning models aim to achieve speedups and reduce computational costs while maintaining the accuracy drops in the tolerance range. Prior work employs rank-1 expansion [13], SVD [20, 38], CP decomposition [10, 18], Tucker decomposition [42, 44], Tensor-Train (TT) decomposition [24, 26, 28], and Tensor-Ring (TR) decomposition [46] to convolutions. While the previous work focuses on leveraging accuracy and speedup trade-offs using tensor decomposition methods, this work reduces memory usage by internal tensors of decomposed model inference. Furthermore, TeMCO's optimization schemes are applicable to decomposition methods [10, 28, 39] that decompose convolutions into 2-dimensional factor matrices and core convolutions.

Internal tensors memory usage reduction: Previous work [4, 15, 48] proposes methods to reduce memory usage of deep learning devices through kernel division. These proposed methods divide layers into iterations of sub-operations and merge the results of the sub-operations into one. In this way, the methods overcome the limits of scratchpad memory on accelerators. This work will improve the performance of fused kernels by applying these methods and generalizing the implementation to CPU and accelerators.

Previous work [19, 31, 50] proposes layer scheduling to minimize memory usage. The layer scheduling affects the peak memory usage of internal tensors because the liveness of tensors depends on the execution order of the operations. The compiler of this work performs skip connection optimizations and reorders the execution scheduling of the layers. This work can further reduce memory usage by finding optimal execution scheduling from previous work.

Internal tensor compression is also discussed in the field of deep learning training. Previous work [5, 14] proposes compressed training schemes that compress internal tensors between inference and weight updates. However, the compressed internal tensors should be decompressed to apply weight updates or further operations, and then the memory usage peaks when performing these operations. On the other hand, the fusion scheme of this work provides fused kernels that perform computations with reduced tensors without restoring them. The skip connection optimization of this work can be extended to training the decomposed models to reduce memory usage by long-lived internal tensors.

6 CONCLUSION

This work proposes tensor memory compiler optimization schemes called TeMCO that reduce the memory usage of internal tensors in decomposed model inference. TeMCO replaces skip connections with reduced tensors by applying static analysis and copying predecessor restore operations. TeMCO also fuses decomposed factor matrix convolutions and non-decomposed activations to perform computations with the reduced tensors without restoring them. This work shows that the prototype compiler reduces memory usage of internal tensors by 75.7% for 10 decomposed models of 5 deep learning architectures.

ACKNOWLEDGMENTS

We thank the CoreLab members for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions. This work is supported by No. RS-2024-00358765, No. RS-2020-II201361, No. RS-2022-II220050, No. RS-2023-00277060, and No. RS-2024-00395134 funded by the Ministry of Science and ICT. This work is also supported by Samsung Electronics. *(Corresponding author: Hanjun Kim)*

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[2] A. F. Agarap. 2018. Deep Learning using Rectified Linear Units (ReLU). *CoRR* (2018). arXiv:1803.08375

[3] M. Alwani, H. Chen, M. Ferdman, and P. Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12.

[4] A. Artemev, Y. An, T. Roeder, and M. van der Wilk. 2022. Memory safe computations with XLA compiler. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.).

[5] J. Chen, L. Zheng, Z. Yao, D. Wang, I. Stoica, M. Mahoney, and J. Gonzalez. 2021. ActNN: Reducing Training Memory Footprint via 2-Bit Activation Compressed Training. In *Proceedings of the 38th International Conference on Machine Learning*, M. Meila and T. Zhang (Eds.). 1803–1813.

[6] J. Cho and B. Hariharan. 2019. On the Efficacy of Knowledge Distillation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

[7] S. Elfwing, E. Uchibe, and K. Doya. 2018. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural Networks* (2018), 3–11.

[8] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[9] Y. He, X. Zhang, and J. Sun. 2017. Channel Pruning for Accelerating Very Deep Neural Networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

[10] F. Hitchcock. 1927. The Expression of a Tensor or a Polyadic as a Sum of Products. *Journal of Mathematics and Physics* (1927), 164–189.

[11] G. Huang, Z. Liu, L. van der Maaten, and K. Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[12] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[13] M. Jaderberg, A. Vedaldi, and A.Zisserman. 2014. Speeding up Convolutional Neural Networks with Low Rank Expansions. arXiv:1405.3866 [cs.CV]

[14] S. Jin, C. Zhang, X. Jiang, Y. Feng, H. Guan, G. Li, S. Song, and D. Tao. 2021. COMET: a novel memory-efficient deep learning training framework by using error-bounded lossy compression. *Proc. VLDB Endow.* (2021), 886–899.

[15] A. Khan, N. Rink, F. Hameed, and J. Castrillon. 2019. Optimizing tensor contractions for embedded devices with racetrack memory scratch-pads. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 5–18.

[16] H. Kim, M. Khan, and C. Kyung. 2019. Efficient Neural Network Compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[17] A. Krizhevsky, I. Sutskever, and G. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.).

[18] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky. 2015. Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition. arXiv:1412.6553 [cs.CV]

[19] J. Lee, S. Jeong, S. Song, K. Kim, H. Choi, Y. Kim, and H. Kim. 2023. Occamy: Memory-efficient GPU Compiler for DNN Inference. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[20] L. Liebenwein, A. Maalouf, D. Feldman, and D. Rus. 2021. Compressing neural networks: Towards determining the optimal layer-wise decomposition. *Advances in Neural Information Processing Systems*, 5328–5344.

[21] D. Lin, S. Talathi, and S. Annapureddy. 2016. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of The 33rd International Conference on Machine Learning*.

[22] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. 2017. Learning Efficient Convolutional Networks Through Network Slimming. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

[23] J. Luo, J. Wu, and W. Lin. 2017. ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

[24] F. Meng, Y. Wu, Z. Zhang, and W. Lu. 2024. TT-CIM: Tensor Train Decomposition for Neural Network in RRAM-Based Compute-in-Memory Systems. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2024), 1172–1183.

[25] S. Mirzadeh, M. Farajtabar, A. Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh. 2020. Improved Knowledge Distillation via Teacher Assistant. *Proceedings of the AAAI Conference on Artificial Intelligence* (2020).

[26] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov. 2015. Tensorizing Neural Networks. In *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Eds.).

[27] NVIDIA. 2024. NVIDIA CuDNN. https://developer.nvidia.com/cudnn.

[28] I. Oseledets. 2011. Tensor-Train Decomposition. *SIAM Journal on Scientific Computing* (2011), 2295–2317.

[29] Y. Pan, J. Xu, M. Wang, J. Ye, F. Wang, K. Bai, and Z. Xu. 2019. Compressing Recurrent Neural Networks with Tensor Ring for Action Recognition. *Proceedings of the AAAI Conference on Artificial Intelligence* (2019), 4683–4690.

[30] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, Lu Fang, J. Bai, and S. Chintala. 2019. PyTorch: an imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., 8026–8037.

[31] Y. Pisarchyk and J. Lee. 2020. Efficient Memory Management for Deep Neural Net Inference. arXiv:2001.03288 [cs.LG]

[32] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. Keckler. 2016. vDNN: virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*.

[33] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. 2022. High-Resolution Image Synthesis With Latent Diffusion Models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 10684–10695.

[34] O. Ronneberger, P. Fischer, and T. Brox. 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, N. Navab, J. Hornegger, W. Wells, and A. Frangi (Eds.). 234–241.

[35] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* (2015), 211–252.

[36] B. Shaler, DanGill, Maggie, M. McDonald, Patricia, and W. Cukierski. 2017. Carvana Image Masking Challenge. https://kaggle.com/competitions/carvana-image-masking-challenge

[37] K. Simonyan and A. Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun (Eds.).

[38] C. Tai, T. Xiao, Yi Zhang, X. Wang, and Weinan E. 2016. Convolutional neural networks with low-rank regularization. arXiv:1511.06067 [cs.LG]

[39] L. Tucker. 1966. Some mathematical notes on three-mode factor analysis. *Psychometrika* (1966), 279–311.

[40] F. Tung and G. Mori. 2019. Similarity-Preserving Knowledge Distillation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

[41] K. Wang, Z. Liu, Y. Lin, Ji Lin, and S. Han. 2019. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[42] L. Xiang, M. Yin, C. Zhang, A. Sukumaran-Rajam, P. Sadayappan, B. Yuan, and D. Tao. 2023. TDC: Towards Extremely Efficient CNNs on GPUs via Hardware-Aware Tucker Decomposition. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 260–273.

[43] Y. Yang, D. Krompass, and V. Tresp. 2017. Tensor-Train Recurrent Neural Networks for Video Classification. In *Proceedings of the 34th International Conference on Machine Learning*, Doina Precup and Yee Whye Teh (Eds.).

[44] M. Yin, H. Phan, X. Zang, S. Liao, and B. Yuan. 2022. BATUDE: Budget-Aware Neural Network Compression Based on Tucker Decomposition. *Proceedings of the AAAI Conference on Artificial Intelligence* (2022), 8874–8882.

[45] M. Yin, Y. Sui, S. Liao, and B. Yuan. 2021. Towards Efficient Tensor Decomposition-Based DNN Model Compression With Optimization Framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 10674–10683.

[46] L. Yuan, C. Li, D. Mandic, J. Cao, and Q. Zhao. 2019. Tensor Ring Decomposition with Rank Minimization on Latent Space: An Efficient Approach for Tensor Completion. *Proceedings of the AAAI Conference on Artificial Intelligence* (2019), 9151–9158.

[47] B. Zhao, Q. Cui, R. Song, Y. Qiu, and J. Liang. 2022. Decoupled Knowledge Distillation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 11953–11962.

[48] H. Zheng, Y. Liu, C. Hsu, and T. Yeh. 2024. StreamNet: Memory-Efficient Streaming Tiny Deep Learning Inference on the Microcontroller. *Advances in Neural Information Processing Systems* 36.

[49] C. Zhu, S. Han, H. Mao, and W. J. Dally. 2017. Trained Ternary Quantization. In *International Conference on Learning Representations*.

[50] L. Zhu, L. Hu, Ji Lin, W. Chen, W. Wang, C. Gan, and S. Han. 2023. PockEngine: Sparse and Efficient Fine-tuning in a Pocket. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1381–1394.