Training of Deep Learning Pipelines on Memory-Constrained GPUs via Segmented Fused-Tiled Execution

Yufan Xu
yf.xu@utah.edu
University of Utah
Salt Lake City, Utah, USA

Saurabh Raje
saurabh.raje@utah.edu
University of Utah
Salt Lake City, Utah, USA

Atanas Rountev
rountev@cse.ohio-state.edu
Ohio State University
Columbus, Ohio, USA

Gerald Sabin
gsabin@rnet-tech.com
RNET Technologies
Dayton, Ohio, USA

Aravind Sukumaran-Rajam
a.sukumaranrajam@wsu.edu
Washington State University
Pullman, Washington, USA

P. Sadayappan
saday@cs.utah.edu
University of Utah
Salt Lake City, Utah, USA

Abstract

Training models with massive inputs is a significant challenge in the development of Deep Learning pipelines to process very large digital image datasets as required by Whole Slide Imaging (WSI) in computational pathology and analysis of brain fMRI images in computational neuroscience. Graphics Processing Units (GPUs) represent the primary workhorse in training and inference of Deep Learning models. In order to use GPUs to run inference or training on a neural network pipeline, state-of-the-art machine learning frameworks like PyTorch and TensorFlow currently require that the collective memory on the GPUs must be larger than the size of the activations at any stage in the pipeline. Therefore, existing Deep Learning pipelines for these use cases have been forced to develop sub-optimal "patch-based" modeling approaches, where images are processed in small segments of an image [5, 7, 13, 15]. In this paper, we present a solution to this problem by employing tiling in conjunction with check-pointing, thereby enabling arbitrarily large images to be directly processed, irrespective of the size of global memory on a GPU and the number of available GPUs. Experimental results using PyTorch demonstrate enhanced functionality/performance over existing frameworks.

CCS Concepts: • **Computing methodologies → Modeling methodologies**; *Machine learning*; **Neural networks**; • **Software and its engineering → Software performance**; *Compilers.*

Keywords: DNN, GPU, Large image training, Fusion, Tiling, Memory-constrained execution, Checkpointing

ACM Reference Format:
Yufan Xu, Saurabh Raje, Atanas Rountev, Gerald Sabin, Aravind Sukumaran-Rajam, and P. Sadayappan. 2022. Training of Deep Learning Pipelines on Memory-Constrained GPUs via Segmented Fused-Tiled Execution. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22), April 02–03, 2022, Seoul, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3497776.3517766

1 Introduction

Deep learning has transformed many applications of image processing. However, a few domains with massive image data, such as digital pathology and brain fMRI analysis, face significant challenges in developing deep learning models due to memory limitations. Virtually all deep learning today uses the computational power of GPUs, which offers significant performance improvement as compared to CPUs. But GPUs have much less memory (usually 32 GiB or less). Training of these Deep Learning pipelines requires that the activations computed at each layer in the forward pass are used to compute the gradients in the backward pass, where the layers are processed in reverse order. Therefore, popular machine learning frameworks like PyTorch [18] and TensorFlow [1] normally store the forward activations at all layers until the backward pass commences, and thus the total set of activations must fit within GPU global memory. While saving/reloading activations from host memory is possible, the low bandwidth between host and GPU has a drastic impact on performance and hence this option is not used in PyTorch or TensorFlow. This memory-constrained usage limitation has forced researchers in these domains to use sub-optimal models, either by coarsening the input data (e.g., brain fMRI analysis [3]) or by use of suboptimal "patch" based modeling using smaller slices of data from full images (e.g., digital pathology [16]). In this paper, we develop a static compile-time analysis and transformation approach to overcome this

CC '22, April 02–03, 2022, Seoul, South Korea

ACM ISBN 978-1-4503-9183-2/22/04.
https://doi.org/10.1145/3497776.3517766

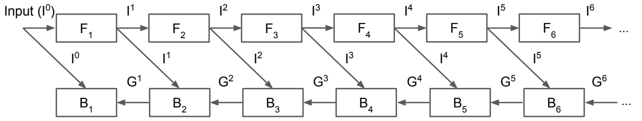

Figure 1. Example of a sequential DNN pipeline with 6 layers. For layer l, F_l is the operator of the forward function, and B_l is the corresponding backward function generated by the ML framework; I^l represents an input activation (I^0 is the input) and G^l represents a gradient.

problem, along with a demonstration via a prototype implementation using the popular PyTorch machine learning framework.

Our approach to enable the training of deep learning pipelines on memory-constrained GPUs is to combine checkpointing and recomputation with tiled execution. When the operators of a sequence of consecutive layers in a DNN pipeline are amenable to *compatible tiling* and fused execution of tiles across the layers, the memory requirements can be dramatically reduced. However, not all sequences of consecutive DNN layers can be compatibly tiled and fused. Therefore, we develop an approach to identify consecutive operators in a DNN pipeline that are mutually compatible for fused-tiled execution, which we term an *FT segment* in the DNN pipeline. We develop compile-time analyses for the identification of feasible FT segments, and the determination of effective tile sizes for efficient fused-tiled execution of the layers within an FT segment.

We use the name *SFT* for our approach: *Segmented Fused-Tiled* execution. The main contributions of the paper are:

- An abstraction to characterize DNN operators and sequences of DNN operators with regards to compatibly tiled and fused (*FT*) execution (Sections 3 and 4.1);
- A compile-time algorithm for partitioning the layers of a DNN pipeline into a sequence of FT segments for tiled execution with checkpoint/recompute (Section 4.2);
- A compile-time algorithm for identifying tensor slice sizes for efficient fused-tiled execution of FT segments (Section 4.3);
- A PyTorch-based implementation of the new SFT approach to train deep learning pipelines on a memory-constrained GPU (Section 5);
- An experimental evaluation demonstrating efficient execution of DNN training pipelines with massive input images (up to 20K × 20K pixels) on a single GPU with only 11 GiB memory (Section 6).

2 Background

2.1 Forward and backward propagation

Figure 1 shows an example of a DNN training pipeline. During the forward pass, the forward operators (F_n, $n = 1, 2, \ldots$) are evaluated in *layer* order. The input activation tensors for each neural network layer (I^n, $n = 0, 1, 2, \ldots$) must be saved until they are used to compute the gradient by the appropriate backward operator, as shown by the diagonal edges between the forward and backward operators. After the output layer (F_6 in this example), a loss function is evaluated and the gradient of the loss (G^6) is computed to start the backward pass. In the backward pass, the operators (B_n, $n = 1, 2, \ldots$) are evaluated in reverse order. Since all inputs I^n must be saved until the start of the back propagation, the memory requirement grows linearly with the number of neural network layers.

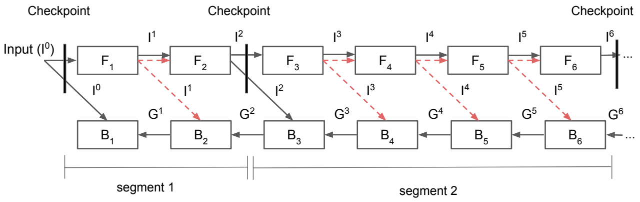

Figure 2. Checkpointing a network with 6 layers. The red dashed lines represent recomputation of the activations that were not stored. The backward pass therefore requires local forward passes.

2.2 Memory reduction via checkpoint/recompute

The total memory required for DNN training can be reduced by saving only a subset of activations during the forward pass and recomputing the unsaved activations when they are needed during the backward pass [8]. The nodes that save input activations in the forward pass are called checkpoint nodes, while the remaining "non-checkpoint" nodes release the memory for their activations after their use in the forward pass. Figure 2 shows a checkpoint strategy for the DNN pipeline from Figure 1. There are two checkpoint segments; vertical bars in the figure represent the checkpoint locations. The first segment contains F_1 and F_2, and the second segment contains all layers from F_3 to the end of the network. During the backward pass within a segment, the activations of the forward operators of all layers are recomputed for all non-checkpoint nodes in the segment, and are kept in memory until they are used during the back propagation for that segment.

Several efforts have developed schemes for checkpoint/recompute execution during training; an overview is presented by Rojas et al. [20]. However, none of these schemes can be used when the size of a single activation is too large to fit in GPU memory, i.e., the scenario we address in this paper.

2.3 Fused-Tiled execution

Tiling and fusion have been used in the design of accelerators for inference in DNNs [2, 24]. Tiling and fusion allow a subset (tile) of the input activation data to be moved into the accelerator, and then the tile is processed through a series

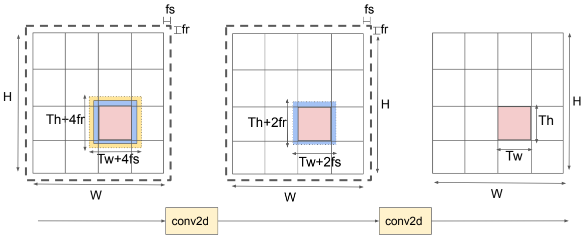

Figure 3. Tile size computation in a convolutional network with 2D convolutions using stride 1 with a kernel size $R \times S$; fill/padding values are $f_r = (R-1)/2$ and $f_s = (S-1)/2$.

Table 1. Description of convolution parameters.

	Description		Description
B	Batch size	K	Output channel
H	Height of Input	R	Height of Kernel/Filter
W	Width of Input	S	Width of Kernel/Filter
O_h	Height of Output	f_r	Padding value in H
O_w	Width of Output	f_s	Padding value in W
C	Input channel	p	Stride size

of individual layers to generate the output tile. For such a fused-tiled execution, additional "halos" must be available for slices of input activations, as described below. Figure 3 shows a short network segment with a sequence of two 2D convolutional operators. Let the kernel size be $R \times S$ with a stride of 1. In a standard untiled 2D convolution, the input is padded/filled such that the output activation size matches the input activation size. The vertical fill size is $f_r = (R-1)/2$ and the horizontal fill size is $f_s = (S-1)/2$, allowing the application of the kernel to boundary activations. The dashed box surrounding the entire activation represents the filled shape of the input, accounting for the padding.

In a tiled execution, each computational tile produces a 2D slice of the full activation. In order to produce a slice of size $T_h \times T_w$ at the output of the second conv2d stage, a slightly larger input data slice of size $(T_h + 2f_r) \times (T_w + 2f_s)$ is needed. Thus, in order to compute the $T_w \times T_h$ tile output of the fused convolution (the pink shaded area in Figure 3), the input to the second convolution must be $(T_w + 2f_s) \times (T_h + 2f_r)$, which is represented by central pink tile with the blue fill halo. Similarly, to produce the $(T_w{+}2f_s) \times (T_h{+}2f_r)$ output tile after the first convolution, the input to the fused convolutions must be $(T_w + 4f_s) \times (T_h + 4f_r)$ (the yellow, blue, and pink areas).

The above example has only shown the expanding *halo* of the data slices that must be computed by a sequence of stages during forward propagation. For fused-tiled execution of the combined forward/backward pipeline for DNN training, additional inter-dependencies on tile sizes must be considered, as elaborated later in the paper. Another challenge is the identification of opportunities for fused-tiled execution for arbitrary DNN pipelines.

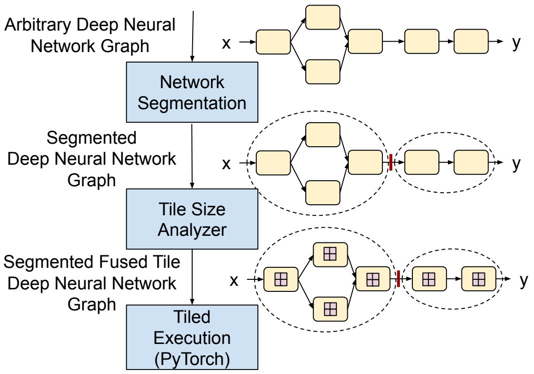
Arbitrary Deep Neural Network Graph
x
y
Network Segmentation
Segmented Deep Neural Network Graph
x
y
Tile Size Analyzer
Segmented Fused Tile Deep Neural Network Graph
x
y
Tiled Execution (PyTorch)

Figure 4. Overview of our approach: FT segmentation (Section 4.2), tile-size analyzer (Section 4.3), and PyTorch execution management (Section 5).

3 Overview of Solution

In this section, we describe our solution to the problem of training deep learning pipelines when GPU memory is insufficient to hold large activations, as encountered in the analysis of WSI (Whole Slide Imaging) in digital pathology. We devise an approach (the first to our knowledge) for fused-tiled execution of the combined operator graph comprised of the forward operators provided by the user and the backward operators automatically generated by an ML framework like PyTorch.

Figure 4 presents a high-level overview of our approach.

1. The first step in our analysis is the partitioning of an arbitrary DNN graph into segments of consecutive layers that can be compatibly fused and tiled. While the forward function can represent an arbitrary DAG, a linear order of execution of the layers (operators) of the forward graph is assumed to be pre-determined by the user, as is common in ML frameworks like TensorFlow and PyTorch. We find maximal sets of consecutive DNN layers whose operators are mutually compatible with respect to tiling and fusion. The entire DNN graph is partitioned into such *FT* sets, with saved activations (checkpoints) between segments and fused-tiled execution within each FT segment. We describe how we formalize compatibility of operators in Section 4.1 and details of the algorithm for identifying maximal FT segments in Section 4.2.
2. Within each FT segment, all operators can be executed in a fused-tiled fashion, with an identical number of tiles for all operators in the segment. However, the tile sizes for these operators have inter-dependencies that have to be analyzed to determine the minimal buffer sizes for correct fused-tiled execution of that FT segment. This analysis is described in Section 4.3.
3. Some details of our fused-tiled implementation in PyTorch are discussed in Section 5. Experimental results for three DNN pipelines (VGG-16, VGG-19 [21], and DarkNet[19])

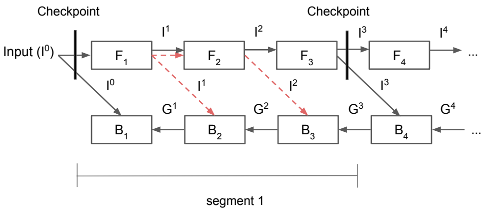

Figure 5. Tiling dataflow in a segment with 4 layers. The checkpoint-segment contains operator 1, 2, 3.

are presented in Section 6, demonstrating the ability to process large images as needed for whole-slide image analysis in digital pathology [7] (results for 10K × 10K and 20K × 20K images are presented).

Figure 5 shows a small three layer segment of a neural network used to compare the dataflow for a baseline execution and the proposed segmented fused-tiled execution (our scheme). The steps involved in training using each of these implementations are given in Table 2. For the baseline execution, all activations $I^n (n = 1, 2, 3)$ are saved in memory. As a result, the input activations just go forward through the network $F1 \dots F4$, the loss is computed, and then back propagation computes gradients $G^n (n = 1, 2, 3)$. All activations must be stored concurrently.

The fused-tiled execution first breaks up the input activation into some number of tiles (e.g., 16), denoted with a subscript $I_i, i = 0, \dots, 15$, and then runs each tile through the forward pass of an FT segment. In the example, the forward and backward operation for all layers (including the loss) are computed after all tiles have gone through forward layer F_3 in the network segment depicted. The back propagation for layers 1 through 4 proceeds tile-by-tile as follows. Gradient G_0^3 (i.e., tile 0, layer 3) is computed using the checkpointed activation I_0^3 and the just computed gradient G^4 (i.e., a tile is recomputed). In order to compute I_0^3, tile 0 is processed from checkpoint 1 through checkpoint 2 (i.e., through F_1, F_2, and F_3). Note, during this forward recompute pass, all tile 0 activations are saved (I_0^1 and I_0^2 in this example). The recomputed I_0^1 and I_0^2 are used to compute the gradients G_0^1 and G_0^2. After all gradients are computed for tile 0, all of the temporarily saved recomputed tile activations have been freed. Next, the remaining tiles are processed (i.e., $1, \dots, 15$ in this example) sequentially.

In the fused-tiled execution each checkpoint activation needs to be saved, along with the activations for each layer of a single tile (which can be arbitrarily small). Fused-tiled segments allow full intermediate activations to never be fully saved (only tiles). Adding more layers to an FT segment reduces the number of full activations that must be saved, but only increases the memory for a segment by a tile. This provides a significant memory savings compared to checkpoint/recompute, especially in many popular networks where the large activations are between convolution and pooling layers. In these networks, these large activations never need to be fully saved and nearly arbitrarily large input images can be processed.

4 Problem Formalization & Algorithm Details

4.1 Problem Formalization

An example of fused-tiled (FT) execution was seen in Fig. 3, where each of the 16 tiles of the second conv2d operator could be executed by fusion with a corresponding tile for the first operator. A chain of such conv2d operators can clearly also be executed in FT fashion. For a DAG of operators to be executable in a fused-tiled manner, each operator must be FT-compatible with respect to one or more pairs of compatible dimensions of input/output tensors, and the interconnected operators must be mutually FT-compatible. We formalize this below.

4.1.1 FT-Compatible Operators:

An operator is defined as *FT-compatible* with respect to a pair of input/output tensor dimensions if a slice of the output tensor with extent T_d along some dimension can be computed using only a slice of the input tensor with extent $\sigma_d T_d + \delta_d$ along the input's dimensions, for constants σ and δ. For example, consider the 2D convolution operator (for simplicity without stride/dilation parameters):

$$Out[n, k, h, w] = \sum_{c,r,s} In[n, c, h + r, w + s] * Ker[k, c, r, s] \quad (1)$$

Consider a slice of the output tensor $Out[T_n, T_k, T_h, T_w]$, with slices of size T_n,T_k,T_h,T_w, respectively along the batch, channel, height, and width dimensions. In order to compute such a slice of the output tensor, only a subset of elements of the input tensor will be needed. As previously illustrated in Figure 3, the minimal slice of the input tensor will be of size $In[T_n, C, T_h + R - 1, T_w + S - 1]$, where C is the number of input channels and R and S are the stencil size along the height and width directions. Thus, the *conv2D* operator is FT-compatible with respect to the batch, height, and width dimensions of the input/output tensors, but not with respect to the channel dimension. The parameters relating the FT-compatible dimensions are: $\sigma_n = 1, \delta_n = 0; \sigma_h = 1, \delta_h = R-1; \sigma_w = 1, \delta_w = S - 1$.

An operator with FT-compatible dimensions can be efficiently executed in a tiled manner, where slices of the output tensor can be produced using slices of the input tensor. Although the set of slices of the input tensors required to produce disjoint slices of the output tensors are not disjoint (as was illustrated in Fig. 5), the amount of redundant computations will be relatively low when the slice sizes are chosen to be large. We define FT-compatible segments as the group of connected operators in a DNN pipeline with mutually consistent FT-compatible dimensions.

4.1.2 FT-compatible Segments:

Two connected operators in a DNN pipeline are *FT-compatible* if they are both FT-compatible with respect to

Table 2. Sequences for the computation for a baseline training and segmented fused tiled training. $Fl(I_i^a)$ depicts the l^{th} forward layer with an input I^a for the i^{th} tile. $Bl(G_i^a)$ is the backward operation for layer l with input gradient G_i^a. i in the Tiling-Checkpoint method represents the number of the tiles that we partition the input into. Saved activations are denoted with a bar on the top.

Base	$\overline{F1(I^0)}$	$\overline{F2(I^1)}$	$\overline{F3(I^2)}$	$F4(I^3)$	$B4(Loss)$	$B3(G^3)$	$B2(G^2)$	$B1(G^1)$		
Tile-Checkpoint	$F1(I_i^0)$	$F2(I_i^1)$	$\overline{F3(I_i^2)}$	$F4(I_i^3)$	$B4(Loss)$	$\overline{F1(I_i^0)}$	$\overline{F2(I_i^1)}$	$B3(G_i^3)$	$B2(G_i^2)$	$B1(G_i^1)$
	$i\in 0..15$					$i\in 0..15$				

at least one common tensor dimension. An *FT-compatible segment* is a set of adjacent layers in a DNN pipeline for which all operators (we only reason with respect to the forward operators since the backward operators have the same FT-compatibility properties as the corresponding forward operators) are all mutually FT-compatible with respect to at least one common tensor dimension. The *FT-compatible dimensions* of an FT-compatible segment are the common set of dimensions that are FT-compatible for all the operators in the set of DNN layers constituting the segment. A maximal FT-compatible segment is one that cannot be extended on either side without violating FT-compatibility.

Figure 6 shows a sequence of four operators (grey colored oval shapes) and the input/output tensors (yellow colored rectangles). Each operator’s computation can be represented as a single perfectly nested loop or a sequence of perfectly nested loops that can be tiled with hyper-rectangular tiles. Further, i) any dimension of any tensor operand (input or output to the operator) can only have a single tileable loop index in its access expression, and ii) any loop index is used to index at most one dimension of any tensor. The above properties define a map from each tensor operand’s data dimension to the operator’s loop iteration space index, as illustrated in Fig. 6. Consider the *conv2D* operator defined in Eq. 1. It represents a 7D loop nest that has five tileable loops (we do not consider the small kernel stencil loops as tileable) corresponding to batch, input channel, output channel, image height, and image width. These five tileable iteration space dimensions are represented as 5 vertices within each *conv2D* operator in Figure 6. Each input/output multi-dimensional tensor has a vertex for each distinct dimension, within the yellow rectangles representing the tensors (we do not explicitly model the *conv2D* operators’ weight matrix (*Ker*) in this graph, but only the tensors that “flow” on the edges of the forward operator graph). The maps between each tensor dimension to the corresponding iteration-space dimension of the operator are also marked as edges connecting the corresponding vertices in the figure. It may be seen that the composition of these tensor-dimension-to-loop-index maps results in connected components in a graph comprised of the union of vertices from all operators in the graph. In the example of Fig. 6, there is a maximal FT-segment that includes all four operators, with respect to the batch (B) index. However, the minimal tile sizes for such an FT-segment would require the full extents along H and W, which would be infeasible for the massive images in digital pathology WSI (Whole Slide Imaging). But a smaller FT-segment exists, comprised of the first three operators (first two convolutions and one maxpooling), which is FT-compatible with respect to three indices (B, H, W), where much smaller tile sizes can be used because the H and W dimensions are also tileable for this smaller FT-segment.

Listing 1. Data Structures

```
class EdgeMeta{
    Vector<int> size;
    Vector<int> inpToIter;
    Vector<int> iterToOut;
    Vector<float> delta;
    Vector<float> scale;
    OpMeta* src;
    OpMeta* target;
}
class OpMeta{
    int id;
    int readyCnt;
    Vector<int> iterSpace;
    Vector<int> fullIterSpace;
    Vector<EdgeMeta> inEdges;
    Vector<EdgeMeta> outEdges;
    OpMeta *dominator;
    OpMeta *postDominator;
    Map<pair<EdgeMeta*,int>,vector<pair<EdgeMeta*,
        int>>> connection;
}
```

A set of operators that form a connected component represents a set of loop indices in the iteration spaces of those operators that are FT-compatible. Listing 2 shows the algorithm that identifies dimensions in the operator graph that are compatible with fused-tiled execution (connected components) and Listing 1 shows the corresponding data structures. First, we iterate through each node in the graph (Line 2). In Line 3, we set the connection map of a given node to the empty set. Then, for the given node, we iterate through each input edge (Line 4). In Lines 7 and 8, we map each dimension of the input edge to the iteration space in

Listing 2. Generate Graph IR

```
1   constructGraph(Vector g, int memCapacity){
2   for(i=0, i < len(g), i++)
3    g[i].connection[] = {}
4    for parent in g[i].inEdges
5     tmpIterToInpMap[] = {}
6     //connect input space to iteration space
7     for d in 0 to len(parent.inpToIter)
8      tmpIterToInpMap[parent.inpToIter[d]] = d
9     for outMeta in g[i].outEdges
10     //using iter. space, connect inp. to out.
           space
11     for inpDimId, iterId in tmpIterToInpMap
12      g[i].connection[<parent,inpDimId>].insert(
            <outMeta,outMeta.iterToOut[iterId]>)
13  }
```

tmpIterToInpMap, i.e., we construct an iteration space to input dimension map. In Line 9, we iterate through each output edge. Note that *iterToOut* contains the iteration space to output dimension map. Using *iterToInp* and *iterToOut*, we map each input edge dimension to the corresponding output edge dimension (Lines 11 and 12). Thus, after execution of the algorithm in Listing 2, we have an internal representation of a graph with the information illustrated in the example of Figure 6.

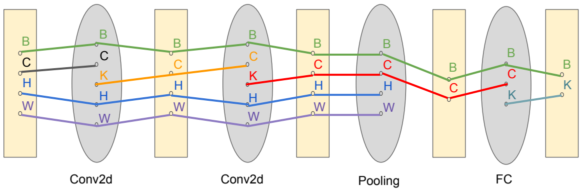

Figure 6. Illustration of FT-compatible segments

4.2 Partitioning of DNN pipeline into FT-compatible segments

In this subsection, we explain how we partition the DNN pipeline into segments separated by checkpoints. The operators within a segment can be executed in a fused-tiled manner. Listing 3 shows the corresponding algorithms. Any node whose output size is less than the memory capacity ($mayTile$) can be checkpointed, whereas a node whose output does not fit in memory ($mustTile$) must be executed using the fused-tile strategy. Lines 2 to 4 classifies each operator as "must tile ($mustTile == True$)" or "may tile ($mustTile == False$)" based on the output tensor size and memory capacity. The loop in line 5 iterates over each node in the operator graph. We ignore "may tile" nodes in Line 6. For each "must tile" node, we try to find a fused-tileable path containing the current node that starts at a checkpoint and ends at a checkpoint

Listing 3. Partition DNN pipeline into Fused-Tileable segments

```
1   findSegments(Vector g, int memCapacity){
2     for n in len(g)
3       mustTile[n] =
4          (∏(g[n].outEdges.size[:]) < memCapacity)
5     for(i=0, i < len(g), i=end)
6       if(!mustTile[i]) continue
7       start = i; end = i + 1
8       //include all post dominators
9       // in the current segment
10      Queue q{i}
11      while(!q.empty())
12        prev_end = end
13        n = q.pop
14        if(g[n].postDominator.id >= end)
15          end = g[n].postDominator.id
16          while(end<len(g)&&mustTile[end])
17            end++
18          q.enque(prev_end:end)
19      FTdims = FTdimsIntersect(g,start, end)
20      bool success = checkMemCapacity(start, end,
21                                 FTdims);
22      if(!success) return(OUT_OF_MEMORY)
23  }
24
25  FTdimsIntersect(Vector g, int start, int end){
26   Map<<InTensorMeta*,int>,bool> FTdims
27   for parent in g[start].inEdges
28    for i in 0 to len(parent.size)
29     queue q({g[start],parent,i})
30     while(!q.empty())
31      node, inpTensor, dim = q.pop()
32      if(!len(node.connection[<inpTensor,i>]) !=
            len(node.outEdges))
33        FTdims[<parent,i>]=False
34        break
35      else
36        for (outMeta, odim) in node.connection[<
              inpTensor,i>]
37         if(outMeta.target < end)
38          q.enque({outMeta.target, outMeta, odim})
39     FTdims[<parent,i>]=True
40    return FTdims
41  }
```

(each checkpoint node must be "may tile"). For any such valid segment, the dominators of all nodes in the segment should either be a checkpoint ("may tile") or be present in the segment. Moreover, all post dominators of a given node, except the post dominator of the end node, must be present in the partition. Lines 10 to 17 ensure these properties. Initially, we add the start node to a queue (Line 9). For each node in the queue, we check whether its post dominator is present in the current segment. If not, we move the endpoint of the current segment to a node that can be checkpointed and is topologically greater than or equal to the current post dominator (Lines 14-17). All nodes between the old endpoint and new endpoint are added to the queue in Line 18. As they are processed, their post dominators are also included in this segment.

Once a valid partition is segment, we identify all the compatible fused-tiled dimensions in function FTdimsIntersect (Lines 25 to 40). This is done by taking each dimension of the start operator of the segment and checking whether there is a fused-tileable path from start to end. For each operator, we check if the corresponding input dimension is fused-tileable (Line 32 to 33). If not, we mark the dimension as not fused-tileable compatible (False) and move to the next dimension. If it is fused-tileable compatible, we enqueue all operators that use the current operator's output and are a part of this segment, along with the corresponding dimension to the queue (Line 36 to 38). Once all dimensions are processed $FTdims$ is returned, which indicates whether each dimension is fused-tileable compatible. In Line 20, we check if the current partition can fit in memory and if not, we throw an "OUT_OF_MEMORY" error (as a suitable SFT was not possible).

4.3 Determination of buffer sizes for SFT execution

Given a graph segmentation generated using Listing 4.2 and a minimal number of tiles, the exact required tile size of each operation must be computed. The checkpoints required to support a segmentation define the output tensors that must be stored and the dimensions that can be tiled. However, the exact tile sizes will vary for each operator due to the relationship of the input tensors to the iteration space and output tensors for a given operation. For instance, a 2D max pool with a stride of 2 results in a reduction in the output width and output height by a factor of 2. In contrast, the $[B, C, K, T_w, T_h]$ iteration space for a 2D convolution operation generating a tensor tile of size $[B, K, T_h, T_w]$ must have a valid input tensor of size $[B, C, T_h + 2f_r, T_w + 2f_s]$ (see Table 3 for definitions for some common neural network operations).

The $computeTileSize$ function in Listing 4 calculates the required iteration space and tensor tile size for every node in a segment's autograd network graph. The required tensor dimensions are deduced from the characterization of each operation, as per the tensor dimension dependencies from Table 4. The iteration space is deduced using a mapping of the tensor dimensions to the iteration space dimensions.

Table 4 defines the tile size required for each input tensor to process a tiled iteration space for two common operations ($convolution$ and $maxpool$) using the tiled tensor dimension requirements in Table 3 and inherent operation characteristics. Each input tensor has a set of vectors that define the required input size during the computation of an iteration space tile. Each vector represents an entry for each dimension in the tensor or iteration space (e.g., $[Batch, Channel, Height, Width]$). These metadata vectors support the tile size propagation for each input tensor (i) through each operation's iteration space via i.size[i]=n.iterSpace[idx]*i.scale[:]+i.delta[:] in Line 20 to 22. The delta (δ) and (σ) vectors contain the parameters described in Section 4.1.1. $inpToIter$ defines the mapping between an input tensor's dimensions and the operation's iteration space, e.g., $[0, 1, 2, 3]$ for max pool and $[0, 1, 3, 4]$ for convolution where iteration space dimension 2 (K) is not included in the input tensor (which has dimensions $[B, C, H, W]$). Using these propagation functions, the required tile sizes for the source tensors of each edge can be computed and updated.

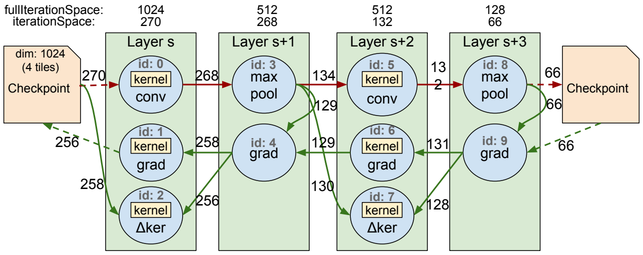

Figure 7. Example neural network segment with two convolution and two max pooling layers. The numbers on each edge represent the T_h and T_w iteration space required assuming a global ($fullIterationSpace$) output tensor tile size of $[1, C, 64, 64]$, a global ($fullIterationSpace$) input iteration space of $[1, C, 1024, 1024]$, and the creation of 4 tiles in each dimension (16 total tiles), i.e., $IN[1, C, T_h = 256, T_w = 256]$ and $OUT[1, K, T_h = 64, T_w = 64]$. Edge annotation denote the T_h (and T_w) tile size required for the tensor.

Figure 7 depicts an example autograd graph segment for a series of convolution (with kernels of size $R \times S$) and max pooling (with a stride of $p = 2$) stages. The segments IN and OUT tensor are checkpointed in memory. The forward graph is shown by red arrows, and the backward graph for gradient ($grad$) and delta kernel (Δker) operations are shown with green arrows. Pooling layers do not have any learned parameters and hence do not have a Δ kernel update operation. The edges are annotated with the width and height tile size (T_h=T_w) (assuming the batch size $T_b = B = 1$ and the channel dimension is not streamable due to the convolutions), as computed by Listing 4. The Δker nodes are leaf nodes that update the kernel, but have no subsequent output tensors. Therefore, these nodes base their input iteration space requirements solely on the initial $iterSpace$. Note that while the output tile size being generated is 64, the grad operation requires a halo of size $2 * f_r$ and thus, the forward pass must generate a tile of size 66.

Table 3. Input tile shape required to process an iteration space tile and the corresponding output tensor shape for some common neural network forward operators.

	IN	**Iter. Space**	**OUT**
Linear	$[T_b, C]$	$[T_b, C, K]$	$[T_b, K]$
Convolution	$[T_b, C, T_h + 2f_r, T_w + 2f_s]$	$[T_b, C, K, T_h, T_w]$	$[T_b, K, T_h, T_w]$
Max Pool	$[T_b, T_c, T_h \times p, T_w \times p]$	$[T_b, T_c, T_h, T_w]$	$[T_b, T_c, T_h, T_w]$
Avg Pool	$[T_b, C, H, W]$	$[T_b, C, H, W]$	$[T_b, C]$
ReLU	$[T_b, T_c, T_h, T_w]$	$[T_b, T_c, T_h, T_w]$	$[T_b, T_c, T_h, T_w]$
Softmax	$[T_b, T_c, H, W]$	$[T_b, T_c, H, W]$	$[T_b, T_c, H, W]$

Table 4. Tile propagation functions from the tile iteration space to the corresponding input tensor tile range for two common neural network operations. The δ and σ vector define the required input tensor dimensions to compute the iteration space dimensions (referenced through the $inpToIter$), corresponding to the tensor shape requirements in Table 3.

Operation	Convolution				Max Pool		
Function	Forward	Grad	Δ Ker		Forward	Grad	
Input Tensor	act_{in}	$grad_{in}$	$grad_{in}$	act_{in}	act_{in}	$grad_{in}$	act_{in}
Delta (δ)	[0,0,2f_r,2f_s]	[0,0,2f_r,2f_s]	[0,0,0,0]	[0,0,2f_r,2f_s]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]
Scale (σ)	[1,1,1,1]	[1,1,1,1]	[1,1,1,1]	[1,1,1,1]	[1,1,p,p]	[1,1,p,p]	[1,1,p,p]
inpToIter	[0,1,3,4]	[0,2,3,4]	[0,2,3,4]	[0,1,3,4]	[0,1,2,3]	[0,1,2,3]	[0,1,2,3]

Listing 4. computeTileSize

```
1  computeTileSize(Graph auto_graph, Vector
       initialReadyNodes, int numTiles){
2   readyNodes.add(initialReadyNodes)
3   //Initialize the default iteration space
4   for n in auto_graph
5      n.iterSpace[:] = n.fullIterSpace[:]
6   for n in initalReadyNodes
7     n.iterSpace[:] = n.fullIterSpace[:]/numTiles
8   //Process all nodes whose outputs are "ready"
9   while(readyNodes.notEmpty())
10   n = readyNodes.pop()
11   //Aggregate output tensor sizes to determine
12   //iteration space required for node n
13   rs = {0} //reduced output tensor size
14   for outMeta in n.outEdges
15     rs[:] = max(rs[:], outMeta.size[:])
16   for (i, idx) in enumerate(n.iterToOut)
17     n.iterSpace[i] = rs[idx]
18   //Update required input tensor sizes
19   for iMeta in n.inTensors:
20     for (i, idx) in enumerate(iMeta.inpToIter)
21       iMeta.size[i] = n.iterSpace[idx]*
22                         iMeta.scale[:]+
23                         iMeta.delta[:]
24     iMeta.src.outEdges[i].size[:] = iMeta.size
         [:]
25     //Identify ready neighbors
26     iMeta.src.readyCnt++
27     if iMeta.src.readyCnt==len(iMeta.src.
         outEdges)
28       readyNodes.add(iMeta)
29 }
```

The $computeTileSize$ function in Listing 4 computes the required iteration space ($n.iterSpace$) and initialize it for each terminating node in Line 5, and tensor ($tensor.size$) tile sizes for the operations in the *auto_grad* graph segment based on the number of tiles required for each dimension (e.g. $numTiles$) in Line 7. The function performs a reverse traversal of an autograd computation graph (see an example graph Figure 7) to calculate the tile sizes. The reversal traversal adds all leaf nodes, like Δker (with no input edges in the segment) to a ready queue ($readyNodes$) in Line 2. Once all output edges are ready, the required tile sizes can be updated and the node can be added to $readyNodes$. Note that a node will become ready when its last output neighbor marks itself as ready in Line 26 to 27.

The required input tensor tile sizes can be computed iteratively using the propagation functions in Table 4 (for simplicity without stride/dilation parameters). The iteration space tile size can be computed by using a max reduction over all output tensors Line 13 and 14. When the $readyQueue$ is empty, all nodes have been processed and the required tile sizes have been updated (this ends the while loop in Line 9).

5 Implementation

We implement *SFT* in PyTorch[1], because it is a prevalent machine learning framework that has support for checkpoint/recompute using *nn.Sequential* and *checkpoint_sequential*. Our *SFT* PyTorch implementation includes custom tiled forward operators, custom tiled autograd functions (i.e., backward functions), static analysis, memory management for tiling, and training.

[1] The software is available at https://github.com/HPCRL/SFT-CC2022-AE

The tool works in two phases: static analysis and regular training execution. The static analysis is performed only once, before the start of the tiled training execution. The primary PyTorch modifications include:

- Customized tiled autograd functions: The tiled forward functions are not symmetric to the backward functions, we do not rely on auto-generated gradient functions in PyTorch. The autograd functions must compute the "ghost" cells (determined statically), and the ghost cells may differ during the forward and backward operations. Further, the updates by the $\Delta kernel$ must correspond to the correct tile despite the overlap "ghost" cells. These details require modifications to the existing autograd functions.
- Optimized PyTorch internal context manager for recomputation in checkpoint segments across different tiled executions: By default PyTorch uses a naive context management strategy to save activations and meta-data for recomputation. There is no memory buffer reuse crossing multiple tiled executions, which wastes GPU memory and eventually results in out-of-memory errors. Also, redundant buffer allocations slow down the training process. We optimize the current context manager by using caching to share the same buffer across different tiled executions.
- Two new operators to handle our *SFT* checkpoint segments. The split node starts the segment and fetches the tiled-input data. The join node marks the end of the segment, collects all tiled-output pieces, and merges them into one tensor for further computation. We also extended the *nn.Sequential* container to support tiled execution with checkpointing. The existing *nn.Sequential* in Pytorch only accepts a tensor as input and produces an output tensor. However, *SFT* execution requires additional meta-information such as tile size, tile position, and padding size, which is obtained from our static tile size analyzer (Figure 4). This information is propagated along the chain of operators to provide the required metadata for forward and backward propagation.

We have intentionally created our APIs to be very close to the existing PyTorch API so that the users can easily port existing code to use the *SFT* framework.

6 Evaluation

6.1 Experimental setup

All experiments presented in this paper are performed using PyTorch v1.8.0a0 and Python v3.9.5. We used the Nvidia CUDA compiler version v11.3.109 and the Nvidia deep neural network library – CUDNN v8.2. The experiments were carried out on two Nvidia GPU systems. An Nvidia 2080 Ti GPU was paired with an AMD Ryzen Threadripper 3990X 64-Core CPU and CPU RAM is 128 GiB in the first platform. The second platform has an Nvidia A100 GPU paired with an Intel(R) Xeon(R) Gold 6238R CPU @ 2.20GHz 56 core CPU and CPU RAM is 384 GiB. Both of the machines run Ubuntu 20.04. These machines represent different GPU generations (Turing and Ampere). We evaluate VGG-16, VGG-19[21]and DarkNet[19] networks to demonstrate the efficacy of our approach.

We evaluate three strategies to perform a training epoch for a single input image (one forward and one backward pass using a batch size of 1) using a Mean Squared Error (MSE) loss function:

1. **Pytorch-Base**: A standard PyTorch[18] implementation of the training computing the forward and backward operations, where all intermediate activations are stored.
2. **Pytorch-CheckPoint**: A sequential checkpoint strategy using the $checkpoint\_sequential$ function [9] in PyTorch. This strategy splits the network into a given number of segments s. We evaluated s values from 2 (always included) to $2\sqrt{L}$, where L is the length of the chain (and the total number of activation tensors stored is minimized).
3. **SFT**: The Segmented Fused-Tiled training strategy outlined in the paper. We split the original input along the height and width dimensions and execute the input in segmented tiles with our checkpoint/recompute PyTorch implementation.

For each model, we vary input image sizes from 512 to 20480 (20K) square (I×I). Image sizes of $10K$ and $20K$ represent ×4 and ×2 magnification for WSI used in pathology. The number of tiles along H and W dimension are varied as 2^n, where $n = 1, 2, 3, 4, 5$ (i.e., 5 different configurations are evaluated).

We measure the execution time for each of the three strategies and get the mean time over 5 runs. The execution time for our method is selected by the best value among the 5 tile size configurations. The execution time is stable for all three strategies over multiple executions. The input image and model use $float32$ precision and the image data layout is $NCHW$.

6.2 Experimental results

Figure 8 shows the experimental results. The square red cross represents the execution time obtained by the standard **Pytorch-Base** strategy, and its absence from the graph means that an out of memory error was encountered when attempting to train an image of the given size. Yellow crosses represent the results obtained with the **Pytorch-Checkpoint** strategy for the shortest execution time among all possible number of segments (from 2 to $2\sqrt{L}$). If both **Pytorch-Base** and **Pytorch-Checkpoint** strategies are available, we only show the execution time of **Pytorch-Base** (as checkpoint/recompute is not required). The blue dot shows the result obtained with our *SFT* strategy. The image size ($H == W$) is plotted on the X-axis and the execution time (in seconds) is on the Y-axis. Since A100 has more device memory than 2080Ti (40GiB compared to 11GiB), the non-tiled strategies can process a larger input image size on the A100. Since VGG-16, VGG-19 and DarkNet networks

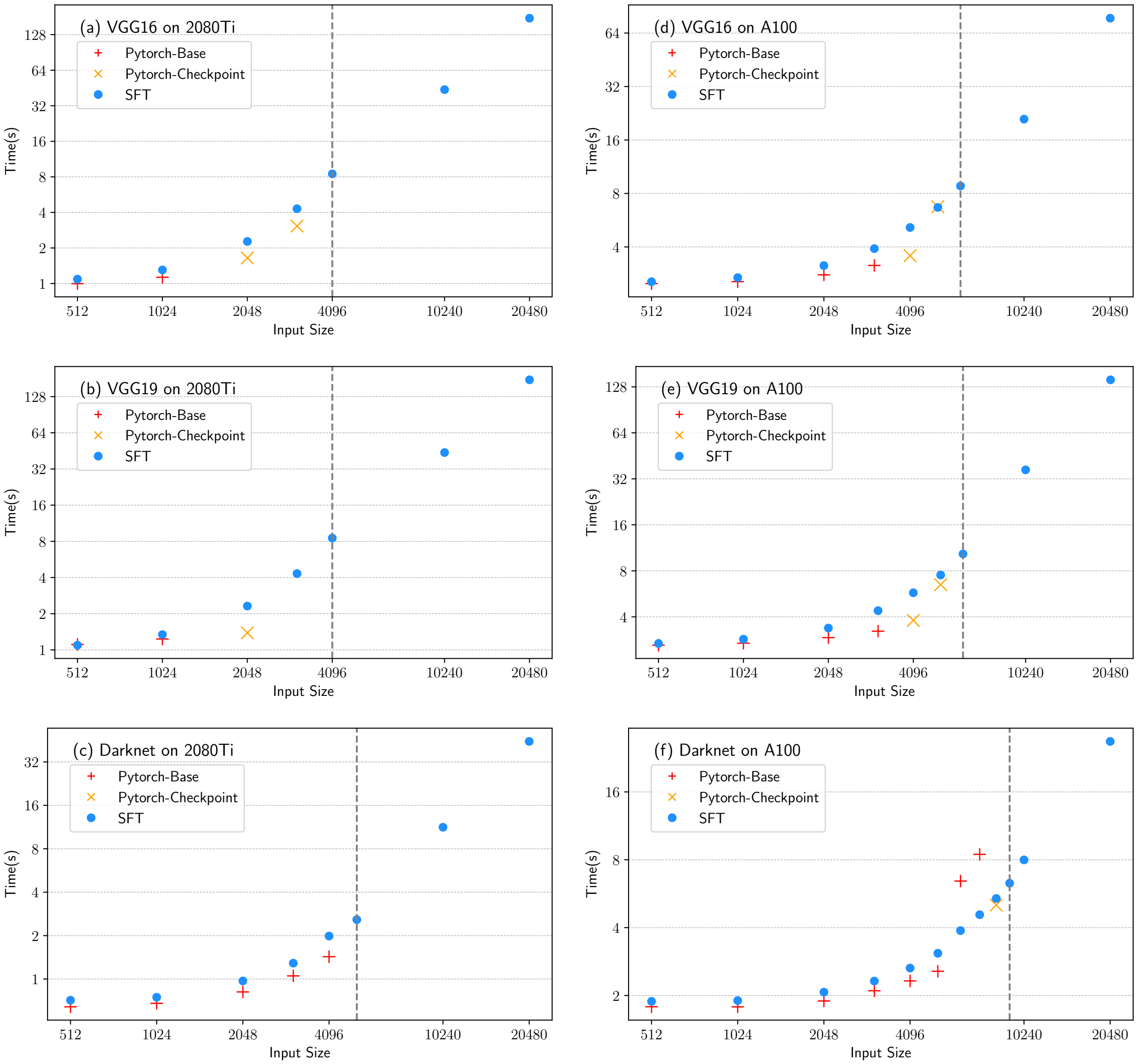

Figure 8. Experimental results VGG16, VGG19 and darknet on NVIDIA A100 and 2080Ti.

do have a huge number of trainable parameters (59 MB for VGG-16, 78.4MB for VGG-19 and 79.7 MB for DarkNet), the GPU device memory is mainly consumed by inputs and intermediate activation required during training. For example, an input image of size $10K \times 10K$ stored in single precision ($float32$) would need a total GPU memory of 118GiB, 129GiB and 43GiB respectively for VGG-16, VGG-19 and Darknet to store all activations and models.

In all six plots, we observe that *SFT* training is able to process much larger input images than either the standard **PyTorch-Base** strategy or the **PyTorch-checkpoint** strategy. The vertical grey dashed line depicts the first image size that neither **Pytorch-Base** nor **Pytorch-Checkpoint** can handle for a given network and GPU pair. *SFT* has a small runtime overhead (as compared to **Pytorch-Base** or **PyTorch-checkpoint**) when the input activations fit in memory. However, the runtime continues to scale linearly with the image size.

7 Related Work

7.1 Operator Fusion

Considerable efforts have been directed at operator fusion, where two or more operators in a DNN pipeline are merged together to create a single combined kernel. The benefits of operator fusion include a reduction in kernel launch overhead. For example, this is a key optimization performed by the XLA [22] compiler. Several other efforts have also addressed such operator fusion [4, 17].

We note that the way “fusion” is used in this work is rather different from the operator-fusion described above, where two or more operators in a DNN pipeline are fused together to create a single combined kernel. In contrast, we fuse the execution of corresponding tiles in a sequence of stages by invoking each operator's kernel on small tiles of data. Thus, we do not generate any fused kernel code, but simply reuse existing kernels by changing the size of the input/output activations and the order of execution, as compared with standard execution of the operators.

7.2 Out-of-core training

KARMA [23] is a system that combines out-of-core training with checkpointing. Out-of-core training involves swapping out large activations from the GPU to the CPU memory to alleviate the memory bottleneck. Using a novel performance model, KARMA automates this decision-making of swapping versus recomputing activations for a given neural network. However, KARMA does not enable processing large-scale images where a single activation layer does not fit on a GPU.

7.3 Multi-GPU Model Parallelism

Recent work leverages “model parallelism” [6, 25] to distribute the activations required during training across multiple GPUs. For instance, GEMS [13] provides a system for hybrid model and data parallelism, along with relaxed synchronisation. While they mention whole slide imaging in the potential applications, their approach uses a well known "patch" based analysis. Specifically, the experiments are conducted on $1K \times 1K$ input size. Other prior approaches based on model parallelism also require this sub-optimal technique to handle large WSI images.

7.4 Unified Memory

Tensorflow Huge Model Support [7] incorporates Nvidia unified memory (UM) [14] along with several GPU memory optimization techniques to train standard CNNs. This work can support whole slide imaging (albeit they only demonstrate ×4 resolution and requires 100s of GPU days for training). However, the open-source implementation is not stable enough to run common neural networks like VGG16. Therefore, we could not compare this against *SFT*. For a single GPU setting, swapping the sections of the image back and forth across host memory and device memory would make the entire training process bandwidth bound. The PCIe bandwidth is about 5 orders of magnitude lower than the GPU peak performance at 32-bit precision. As a result, as reported by Chen at al. [7], HMS gets 30× slower as the image size (in pixels) quadruples (from $11K \times 11K$ image to $21.5K \times 21.5K$ image), while *SFT* time just increases by 4×, i.e., proportional to increase in image size.

7.5 Convolutions Over Distributed Memory

DistConv [10] is an extension built over Livermore Big Artificial Neural Network toolkit [11] that performs distributed convolutions over a cluster of GPUs. Given the size of the image, distributed convolutions place a lower bound on the *number of GPUs* that are required to train the network. For a 100K × 100K image, for example, twenty 32GB GPUs will be required to simply hold the input image. The first stage of an image DNN pipeline typically increases the size of the output activations by a factor of 5-10 (the number of channels increases from 3 to at least 32, while the image height/width gets halved); further increasing the number of GPUs required. We were also not able to set up LBANN as per the documentation given. The version incompatibility among dependencies causes multiple compilation and linkage errors.

DistDL [12] shows performance improvement over *DistConv* and provides a PyTorch extension to partition a large image into multiple smaller disjoint images over a cluster of CPUs. During the forward or backward passes, neighbourhood regions of the image, as required by the convolution kernel, are sent over the network. However, the current implementation does not have a GPU backend support, making an empirical comparison impossible.

8 Conclusion

This paper develops a segmented fused-tiled (*SFT*) approach to enable the training of Deep Neural Networks using very large images, overcoming a significant current limitation in popular ML frameworks like PyTorch and TensorFlow. We develop algorithms to generate fused-tiled-compatible segments and for determination of the memory requirements for the fused-tiled segments. Fused-tiled execution was enabled in the PyTorch framework and was evaluated experimentally with VGG-16, VGG-19, and Darknet DNN pipelines on an Nvidia 2080Ti and an A100 machine. The implementation shows that there is minimal overhead for the tiled implementation and that the tiled implementation scales to large input image sizes well. Our developments enable arbitrarily large input image sizes to be used directly for training machine learning models instead of the current practice of using image pre-coarsening or patch-based processing in domains like digital pathology and computational neuroscience.

Acknowledgments

Research reported in this publication was supported in part by the National Institute Of Biomedical Imaging And Bioengineering of the National Institutes of Health under Award Number R41EB032722, and by the National Science Foundation through awards 2018016, 2119677, and 2118737. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health or the National Science Foundation.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[3] Riddhish Bhalodia, Shireen Y Elhabian, Ladislav Kavan, and Ross T Whitaker. 2018. Deepssm: A deep learning framework for statistical shape modeling from raw images. In *International Workshop on Shape in Medical Imaging*. Springer, 244–257.

[4] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Alexandre V Evfimievski, and Prithviraj Sen. 2018. On optimizing operator fusion plans for large-scale machine learning in systemml. *arXiv preprint arXiv:1801.00829* (2018).

[5] Gabriele Campanella, Matthew G Hanna, Luke Geneslaw, Allen Miraflor, Vitor Werneck Krauss Silva, Klaus J Busam, Edi Brogi, Victor E Reuter, David S Klimstra, and Thomas J Fuchs. 2019. Clinical-grade computational pathology using weakly supervised deep learning on whole slide images. *Nature medicine* 25, 8 (2019), 1301–1309.

[6] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. 2018. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv preprint arXiv:1809.02839* (2018).

[7] Chi-Long Chen, Chi-Chung Chen, Wei-Hsiang Yu, Szu-Hua Chen, Yu-Chan Chang, Tai-I Hsu, Michael Hsiao, Chao-Yuan Yeh, and Cheng-Yu Chen. 2021. An annotation-free whole-slide training approach to pathological classification of lung cancer types using deep learning. *Nature communications* 12, 1 (2021), 1–13.

[8] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[9] Torch Contributors. 2018. Periodic checkpointing in pytorch. https://pytorch.org/docs/stable/checkpoint.html.

[10] Nikoli Dryden, Naoya Maruyama, Tom Benson, Tim Moon, Marc Snir, and Brian Van Essen. 2019. Improving strong-scaling of CNN training by exploiting finer-grained parallelism. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 210–220.

[11] Brian Van Essen, Hyojin Kim, Roger A. Pearce, Kofi Boakye, and Barry Chen. 2015. LBANN: livermore big artificial neural network HPC toolkit. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, MLHPC 2015, Austin, Texas, USA, November 15, 2015*. ACM, 5:1–5:6. https://doi.org/10.1145/2834892.2834897

[12] Russell J Hewett and Thomas J Grady II. 2020. A linear algebraic approach to model parallelism in deep learning. *arXiv preprint arXiv:2006.03108* (2020).

[13] Arpan Jain, Ammar Ahmad Awan, Asmaa M. Aljuhani, Jahanzeb Maqbool Hashmi, Quentin G. Anthony, Hari Subramoni, Dhableswar K. Panda, Raghu Machiraju, and Anil Parwani. 2020. GEMS: GPU-Enabled Memory-Aware Model-Parallelism System for Distributed DNN Training. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41405.2020.00049

[14] Wenqiang Li, Guanghao Jin, Xuewen Cui, and Simon See. 2015. An Evaluation of Unified Memory Technology on NVIDIA GPUs. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 1092–1098. https://doi.org/10.1109/CCGrid.2015.105

[15] Ming Y Lu, Drew FK Williamson, Tiffany Y Chen, Richard J Chen, Matteo Barbieri, and Faisal Mahmood. 2021. Data-efficient and weakly supervised computational pathology on whole-slide images. *Nature Biomedical Engineering* (2021), 1–16.

[16] Pooya Mobadersany, Lee AD Cooper, and Jeffery A Goldstein. 2021. GestAltNet: aggregation and attention to improve deep learning of gestational age from placental whole-slide images. *Laboratory Investigation* (2021), 1–10.

[17] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[19] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[20] Elvis Rojas, Albert Njoroge Kahira, Esteban Meneses, Leonardo Bautista-Gomez, and Rosa M. Badia. 2020. A Study of Checkpointing in Large Scale Training of Deep Neural Networks. *CoRR* abs/2012.00825 (2020). arXiv:2012.00825 https://arxiv.org/abs/2012.00825

[21] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[22] TensorFlow. 2019. XLA: Optimizing Compiler for TensorFlow. https://www.tensorflow.org/xla.

[23] Mohamed Wahib, Haoyu Zhang, Truong Thao Nguyen, Aleksandr Drozd, Jens Domke, Lingqi Zhang, Ryousei Takano, and Satoshi Matsuoka. 2020. Scaling Distributed Deep Learning Workloads beyond the Memory Capacity with KARMA. *CoRR* abs/2008.11421 (2020). arXiv:2008.11421 https://arxiv.org/abs/2008.11421

[24] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2348–2359.

[25] Wentao Zhu, Can Zhao, Wenqi Li, Holger Roth, Ziyue Xu, and Daguang Xu. 2020. Lamp: Large deep nets with automated model parallelism for image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 374–384.