Compute vs Data Transfer: Memory Optimizations for Neural Networks

Group 7: Debajit Chakraborty (18EC10012), Hardik Tibrewal (18EC10020), Hardik Aggarwal (18CS10021), Mukul Mehta (18CS10033), Rutav Shah (18CS10050), Yarlagadda Srihas (18CS10057), Shubham Mishra (18CS10066), Rashil Gandhi (18CS30036)

Table of Contents:

Introduction:

Graphics processing units (GPUs) have emerged as the main computational workhorse for Deep Learning applications, due to their high performance, low cost, and wide availability. But there is a critical limitation in GPUs: the entire data used for GPU computation must be put into GPU memory for execution. Since the physical memory of GPUs is usually very limited, the data scales of CNNs often exceed their memory. The mainstream solution is to reduce the memory consumption of DL training processes as far as possible. Reusing feature maps to save memory usage is a classic method that has been studied in many existing works. By using a CPU-GPU transfer, the memory usage in the GPU can be significantly decreased. These methods usually pay main attention to the efficiency of memory consumption; however, the execution time efficiency of these methods has been significantly overlooked. Since there are many types of intricate intermediate data, the potential of memory reuse methods has still not yet been fully exploited. Preliminary investigations show that there are two issues that have significant impacts on the training performance and memory efficiency of CNNs.

1. Single memory optimization cannot handle the diversity of the different types of layers and causes performance degradation. The layer type in a CNN model significantly affects the execution time of memory optimizations. Some layers are more suitable for extra forward computation (such as ReLU and norm layers), while others are more suitable for CPU-GPU transfer (such as conv layers). Significant layer-aware performance improvements can be obtained by choosing appropriate optimization methods for each of the appropriate layers. However, most existing strategies only employ a single optimization technique for all CNN layers, and this kind of single, uniform optimization approach mismatches the inherent heterogeneity of CNNs with different types of layers.
2. There are diverse types of important intermediate data (such as feature maps, gradient maps, workspace data, and cuDNN handles) that need to be taken into account in the memory reuse of CNNs. In general, the memory requirements of workspaces and cuDNN handles are lower than those of feature maps and gradient maps. As a result, feature maps and gradient maps consume the majority of GPU memory. Investigations show that after certain memory optimizations are applied, the memory occupied by workspace data and cuDNN handles too can no longer be ignored. The latter becomes very large during DL training processes after optimization. Here, workspaces are responsible for convolutional computation and cuDNN handles are responsible for holding cuDNN contexts.

On the basis of these findings, the authors of the Layup paper have presented a strategy that can optimize the GPU memory usage for CNNs. Experiments show that Layup can significantly extend the scale of extra-deep network models on a single GPU, with a relatively small loss of performance. The major aspects of their implementation are: 1) Characterizing the optimization costs of different CNN layers and identifying the impacts of different memory-optimized methods on the execution time, from which a lightweight heuristic to direct the choice of optimization methods with less time overhead is derived. 2) A fast layer-adaptive memory management method that selects different memory optimizations

for different layers, based on the types of layers. 3) A multi-type data reuse strategy that improves the efficiency of the memory reuse of CNNs by exploiting as many opportunities as possible for the memory reuse of multi-type data, and particularly for gradient maps, workspaces, and cuDNN handles.

Related Works:

Recent works have attracted a lot of attention in recent years and various methods have been proposed. One of the strategies involves a divide and conquer strategy known as Model parallelism where the CNN model is divided into several smaller chunks and allocated to different nodes which run in parallel. However, bottlenecks such as division strategies design, extra overhead and convergence guarantees are some problems with model-parallelism.

Several lightweight and memory-efficient methods achieve memory savings by extra forward computation where we use heuristics to decide which layers are to be checkpointed. Thus we leverage the computational resources of GPU to calculate extra so as to save the transfer time from CPU to GPU and vice versa. These can be calculated by methods such as sublinear memory optimization (e.g., memonger in MXNet) and its extensions. However, uncertainties such as deteriorating execution time for successive layers and oversubscription of GPU due to extra computation degrade the performance of the training process.

Layrub reuses the feature and gradient maps from algorithmic and architectural perspectives. The primary limitation of this work is that its inter-layer memory reuse strongly depends on the transfer bandwidth between CPU and GPU, which may introduce significant costs and long delays due to bandwidth contention. Most existing approaches primarily focus on feature maps and gradient maps. Unlike these, our strategy tries to achieve both better execution time and higher memory efficiency by checkpointing layers and performing extra-forward computation.

Original Implementation:

Selection of Layer-adaptive Dynamic Memory Optimization Methods

An important observation is that the ratios of the transfer cost and the extra forward cost are different for different layers. It is reasonable to set a threshold to determine whether to select the transfer or the extra forward strategy for a specific layer. The threshold is determined by comparing the cost of CPU-GPU transfer with the cost of extra forward computation:

$$Threshold_{Layer(i)} = \frac{Cost_{Transfer}}{Cost_{extra\text{-}forward}}. \quad (1)$$

After substitution by the appropriate equations, we get:

$$Threshold_{Layer(i)} = \frac{InputSize_{Layer(i)}}{FLOPs_{Layer(i)}} \times \frac{maxFLOPS \times UtilRate}{Bandwidth}, \quad (2)$$

where the first part of the right-hand side of Equation (2) is only related to the size of the input data and the type of the layer; the second part depends on the hardware configuration of

the node, which is independent of the network model. For a specific node, the second part can be regarded as a fixed coefficient.
For the situation where $Threshold_{Layer(i)} \leq 1$, that is, $Cost_{Transfer}$ is less than or equal to $Cost_{extra\text{-}forward}$ the data transfer and the calculation process can completely overlap, and the overhead for data transfer can be regarded as zero. It is therefore clear that data transfer is preferable in this situation. More attention should be paid to the situation where $1 \leq Threshold_{Layer(i)} \leq 2$. In this situation, the overhead for data transfer is still less than that of one extra forward, meaning that it is still better to use data transfer to obtain the feature map than to use the extra forward process. For the situation where $Threshold_{Layer(i)} \geq 2$, the overhead for data transfer is greater than that for the extra forward approach, meaning that the extra forward approach becomes the better choice.

In summary, $Threshold_{Layer(i)} = 2$ is a watershed. When $Threshold_{Layer(i)} \geq 2$, the layer is transfer-sensitive; otherwise, it is compute-sensitive. Based on this, a layer-adaptive data placement strategy is described as follows:

- For the compute-sensitive layer, in the forward pass, the input feature map of the current layer is asynchronously transferred to CPU memory. In the backward pass, the strategy fetches the feature map that is temporarily stored in CPU memory and will be used later. The CPU-GPU data transfer overlaps with the computation.
- For the transfer-sensitive layer, the strategy drops the input feature map of the current layer in the forward pass. However, the memory block is kept unreleased and reused for the next feature map. In the corresponding backward pass, the strategy recovers the dropped feature maps using extra forward computation.

A profiling-based method to perform pure forward propagation on the target CNN model before the actual training is also proposed. A CUDA event API is used to measure the forward time and transfer time for each layer. Separate CUDA streams are used to implement asynchronous data transfer and extra forward computation in the pipeline, respectively, to ensure that the training performance is not seriously affected. One CUDA stream is used for memory copy operations between the host and the device, and the other is used to manage extra forward tasks on the device.

Multi-type Intermediate Data Reuse Strategy
Gradient Maps: The gradient map reuses the memory space of the corresponding feature map and can achieve up to 50% memory savings. However, this intra-layer memory reuse method introduces some data dependency hazards between the gradient of parameter and the gradient of activation data. The synchronization control introduced to resolve these hazards reduces the degree of the computational parallelism, which impairs the training performance; it also cannot be used directly on the activation layer, which is unacceptable for CNNs, since they usually have a large number of activation layers.
To optimize the memory usage of the gradient map, a new memory optimization method based on a sliding window, which involves no such hazards and can be applied to any type of layer, is proposed. A backward propagation usually consists of a series of layer-wise computations, and only a single-layer backward computation can be performed at any one

time. Based on this situation, a sliding window is typically used for backward computation. It contains a single layer at any given moment and slides from the last layer to the first during the backward computation. This motivates us to reuse the GPU memory for gradient maps layer-by-layer.
It is worth noting that $w + 1$ memory blocks are required for the gradient maps, where w is the width of the network. As a result, the memory cost of the optimized gradient maps is $O(w)$, which is linearly correlated to the width of the network.
In the optimization strategy for the feature map, a slight performance overhead is incurred due to the synchronization of pipelines.

Workspace and cuDNN handle: These types of intermediate data have not been a subject of much interest when it comes to memory optimizations. As noted earlier though, these can become a significant bottleneck after optimization techniques to feature maps and gradient maps have been applied. Fortunately, the optimization strategies for the workspace and cuDNN handles are relatively straightforward.
After the calculation of the ith layer is completed, the memory block corresponding to the workspace becomes idle and can then be directly reused by the workspace of the next layer. The optimization strategy for the cuDNN handle is carried out in the same way. Based on these optimizations, constant memory space can be reused for the workspace and cuDNN handles.

Profiling Results:
The most important observation during preliminary profiling of the transfer and compute times of the different layers of the convolutional neural network. The transfer time of the feature maps for each layer was much greater than the compute time (except for the last few layers). This is quite different from the experimental results of the paper, where the convolutional layers were taking more time for an extra forward computation compared to the time for transfer between the CPU and the GPU.
This trend was observed for different devices, as well as different batch sizes. The profiling results of the transfer and compute times, as well as the ratio of the two, are shown below for each layer for different set-ups.

Plots for:- GPU: Tesla P100-PCIE-16GB, VGG Net, Batch size = 10

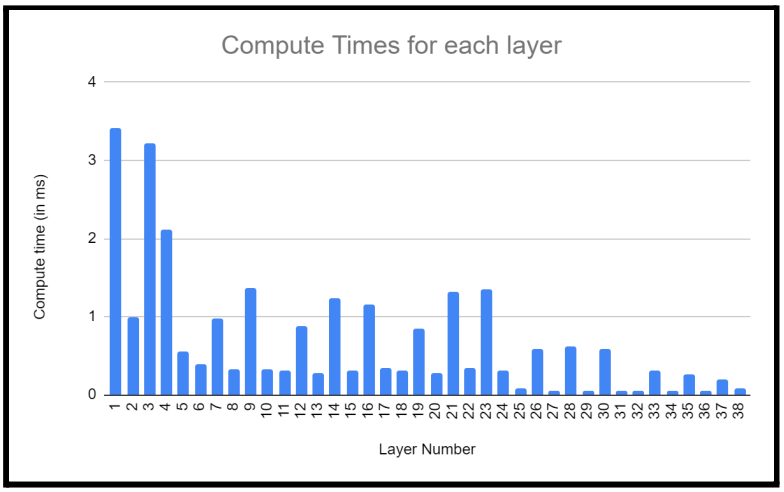


Fig 1.1

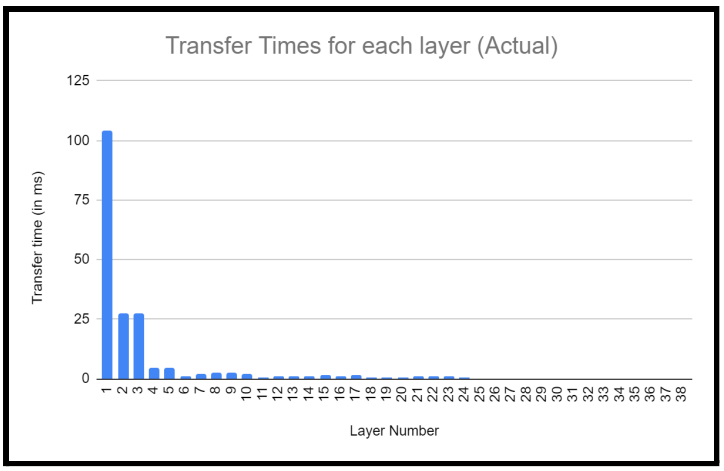


Fig 1.2

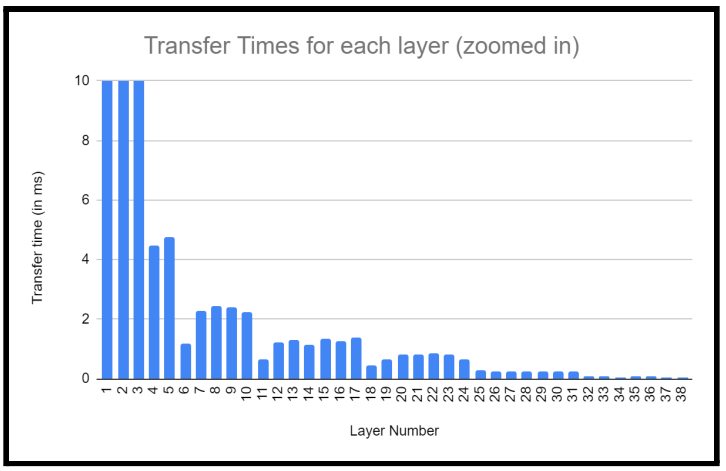


Fig 1.3

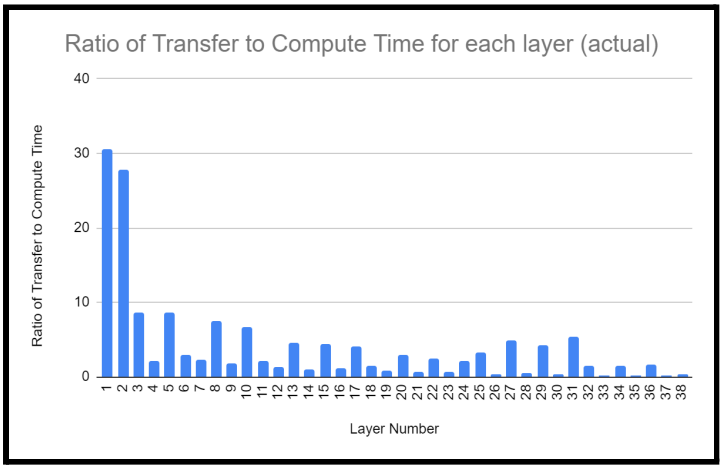


Fig 1.4

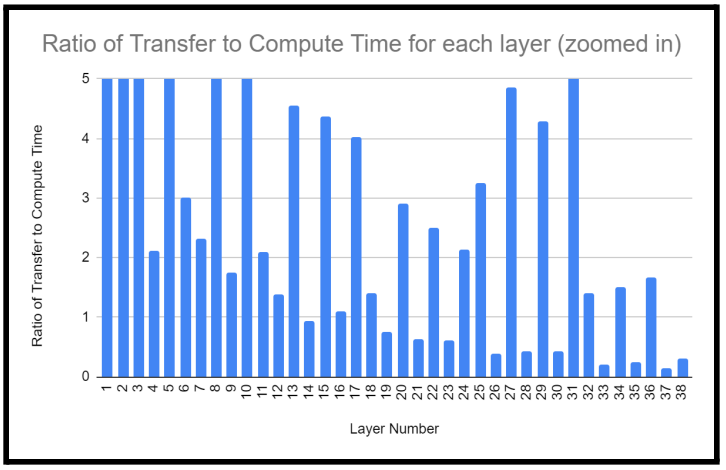


Fig 1.5

Plots for:- GPU: Tesla P100-PCIE-16GB, VGG Net, Batch size = 20

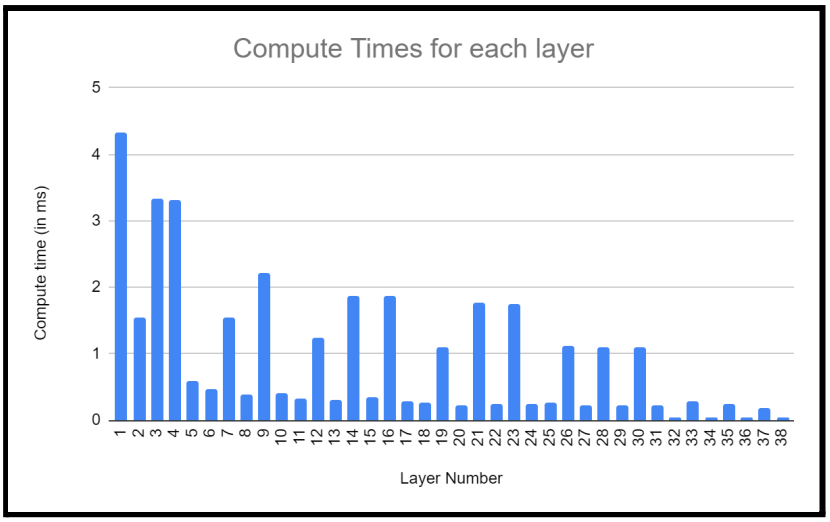


Fig 2.1

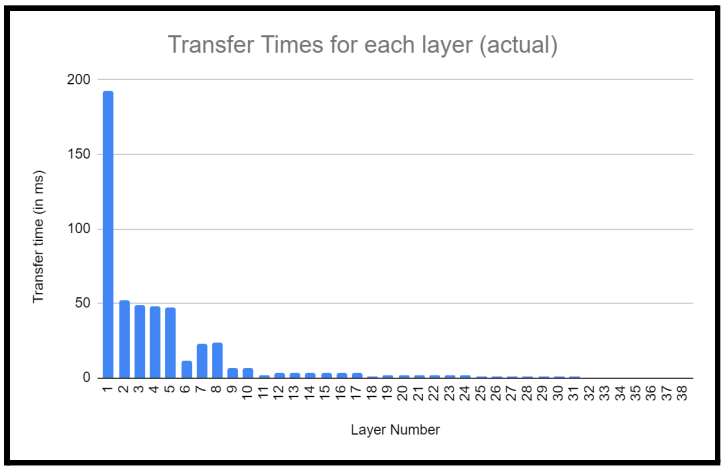


Fig 2.2

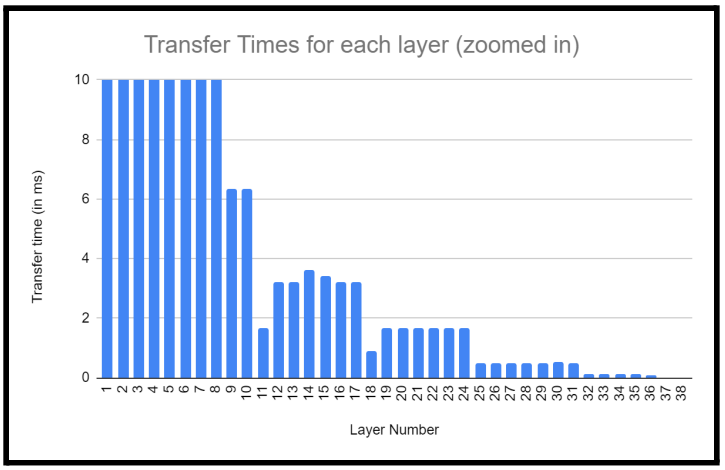


Fig 2.3

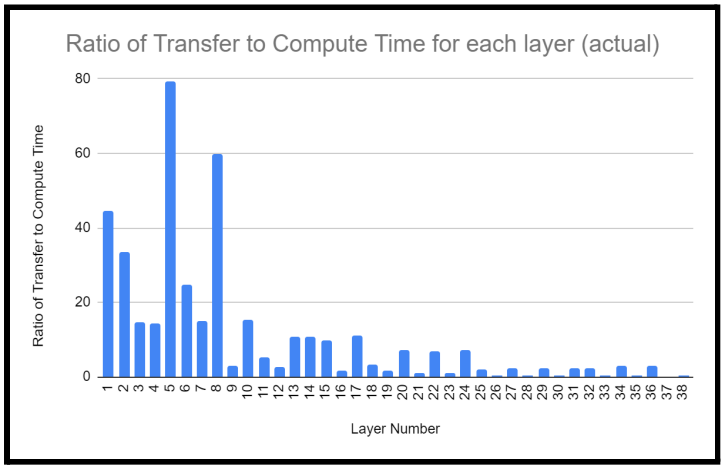


Fig 2.4

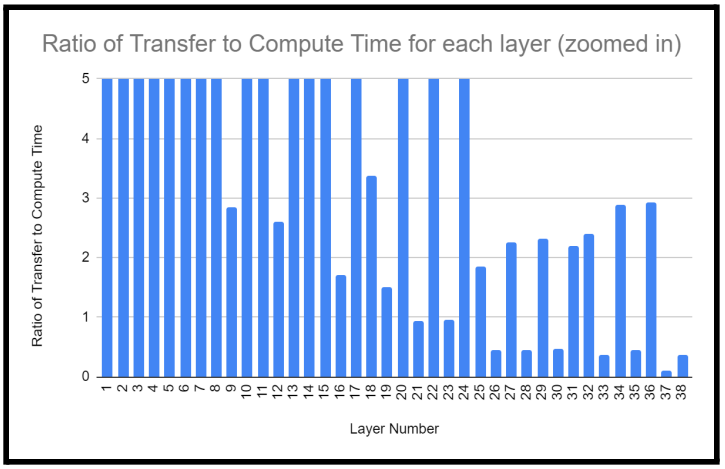


Fig 2.5

Plots for:- GPU: GeForce Titan X, VGG Net, Batch size = 10

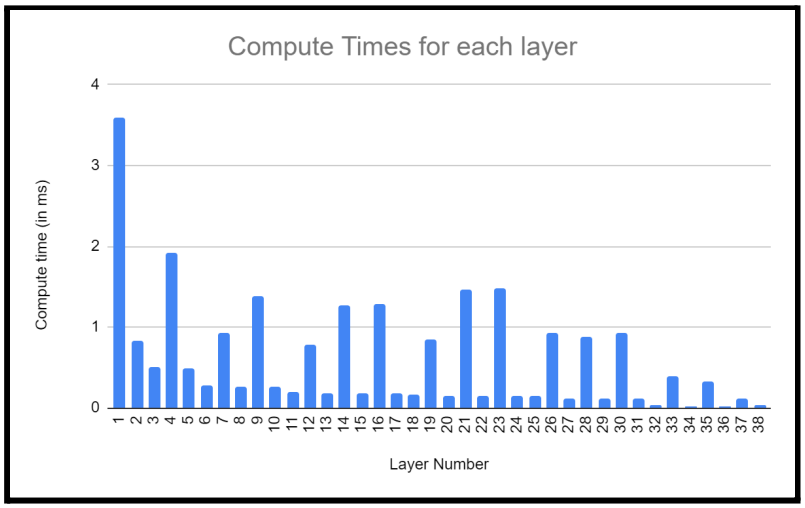


Fig 3.1

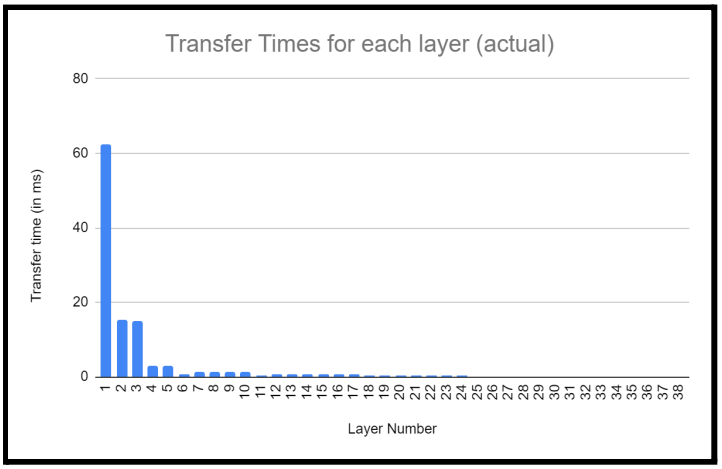


Fig 3.2

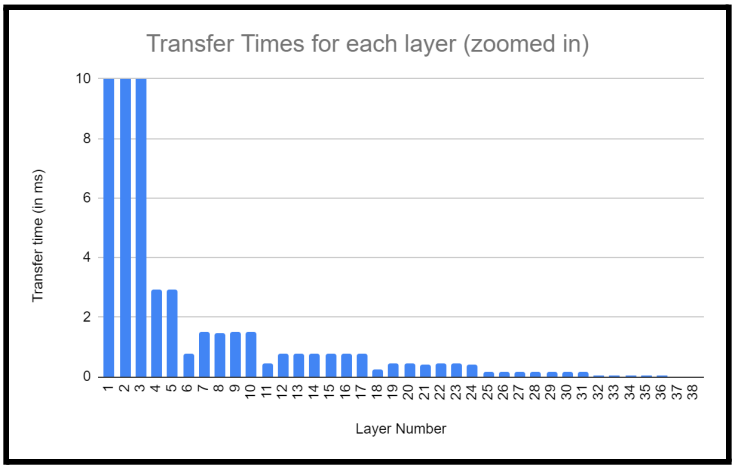


Fig 3.3

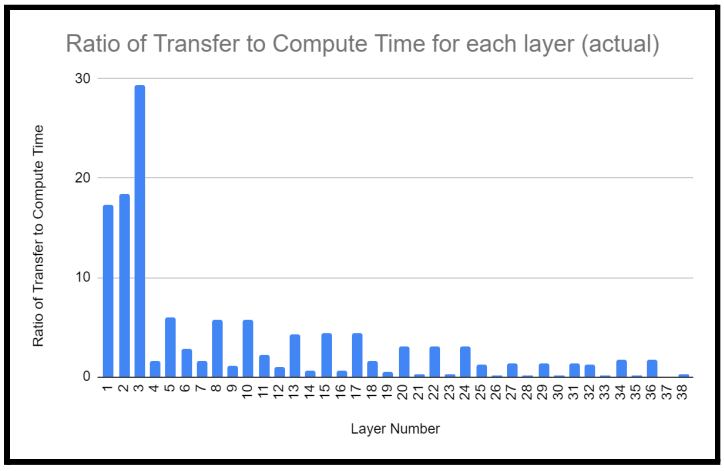


Fig 3.4

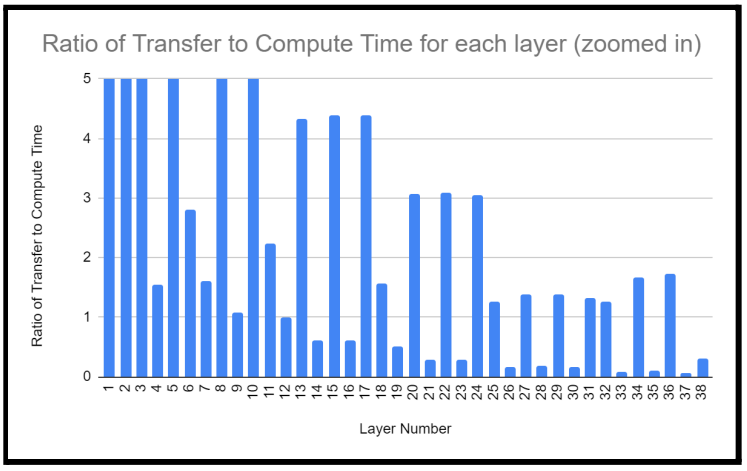


Fig 3.5

Profiling and Methodology

As we can see from the plots of the ratio of transfer to compute times for each layer, it becomes apparent that almost every layer is transfer-sensitive (since the ratios are greater than 1 for most layers), even the convolutional layers which are expected to be compute-sensitive owing to the complexity of convolution. Therefore, layer-based profiling does not seem to work in this case.

The paper also fails to mention how they perform the extra forward computation during the backward pass without spending time on at least one CPU-GPU transfer. The authors' approach to the backward pass requires the feature map for the transfer sensitive layers to be calculated using extra-forward computation, but the starting point for all extra forward computations has to be present in the GPU, leading to huge memory usage which goes against the goal of memory optimisation, or fetched from the CPU, adding to the transfer costs which have not accounted for in their calculations to assign a particular layer as transfer or compute sensitive.

Keeping these shortcomings and observations in mind, we have adopted a checkpointing based approach for memory optimisation keeping the time required for execution in mind, by mixing the approaches of extra-forward computation on CPU-GPU memory transfer. The basic approach, which shall be elaborated further, is as follows:

1) A layer is marked as a checkpoint if the time to compute the feature map of the layer starting from the previous checkpoint is more than the time to transfer the feature map between the CPU and GPU. The input layer is the first checkpoint, and the remaining checkpoints are marked before the training of the neural network starts.
2) In the forward pass, the feature map of a layer is transferred to the CPU if and only if the layer is a checkpoint. Otherwise, as soon as the calculations concerning the feature map are over, it is dropped, thus freeing up space in the GPU and providing memory optimization by reuse.
3) In the backward pass, the feature map of the last checkpoint is used to begin the calculations of the backward pass through extra forward computation. Simultaneously, the feature map of the checkpoint before this is fetched from the CPU memory using a different data stream. This overlap of the backward calculations and fetching the feature map required later reduces the time required for execution while ensuring memory reuse.

Checkpoint Assignment (Algorithm 1):

In this section, we will describe our methodology and experimental setup for the layer-adaptive dynamic selection of memory optimization methods. As discussed in the previous section, that is how the approach mentioned in the original paper implementation generated poor results. The paper implementation defines a new variable *Threshold* which is given by:

$$Threshold = T_{transfer} / T_{compute}$$

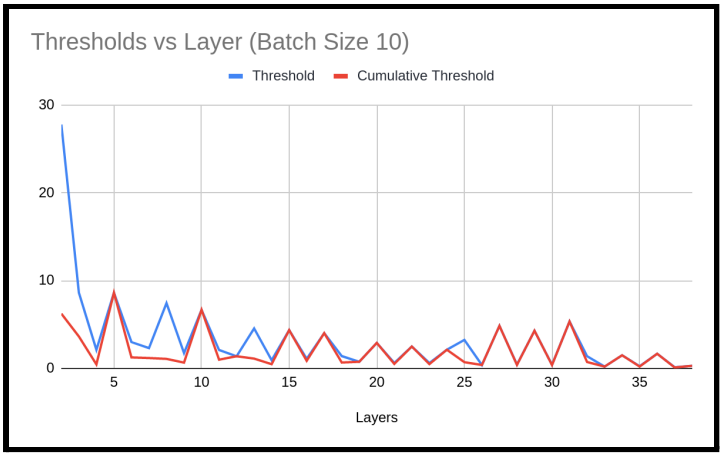
where $T_{transfer}$ is the time taken to transfer the feature maps from device to host and $T_{compute}$ is the time to make a forward pass through the layer.
We modify this equation to

$$Cumulative\_Threshold = T_{transfer}/T_{cumulative\_compute}$$

Here we introduce a new variable called $T_{cumulative\_compute}$ which is the time sum of the forward computation from the last checkpoint to the current layer. A layer is called a checkpoint layer if the value of *Threshold* at layer *'i'* drops below 1. We only save the feature maps of the checkpoint layer and drop the outputs of all the others giving us our desired memory optimization.

This indicates that at layer *'i'*, the time taken to transfer from device to CPU is now less than the time to go back to the last checkpoint layer and make subsequent forward passes till we reach the current layer. Hence we make the current layer *'i'* as a checkpoint. We also save the output feature maps to the CPU and reset the $T_{cumulative\_compute}$ to 0. We make the Input Layer (Layer 0) to be a checkpoint by default. We observe that the ratio of transfer_time and compute_time is different layers. For layers having large computation requirements like convolution layers and fully connected layers, the value of $T_{compute}$ is significantly larger than transfer_time. Hence a layer is more likely to become a checkpoint layer as the number of convolutional layers increases between itself and the last checkpointed layer. The plot for thresholds v/s Layers is shown in **Figures 4.1 to 4.4**.
In practice, to capture the transfer cost and forward cost of each layer, we also propose a modified profiling-based method to perform pure forward propagation on the target CNN model before the actual training (see function Profile). Specifically, we use a CUDA event API to measure the forward time and transfer time for each layer.

Threshold and Cumulative threshold values for each layer - GPU: Tesla P100-PCIE-16GB, VGG Net



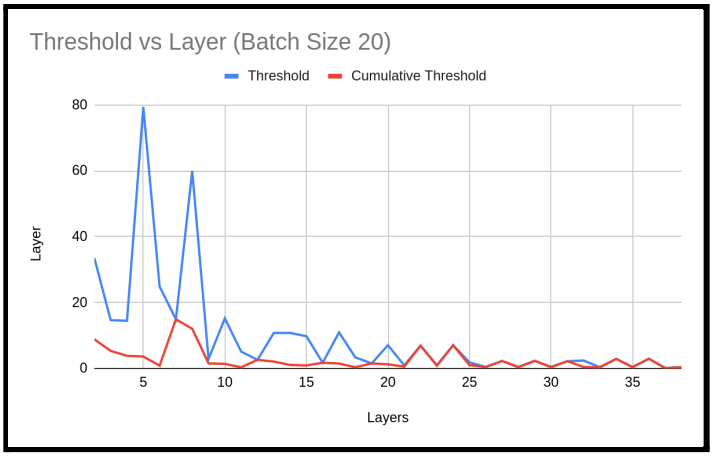


Fig. 4.1 (Batch size = 10) **Fig. 4.2** (Batch size = 20)

Threshold and Cumulative threshold values for each layer - GPU: GeForce Titan X, VGG Net

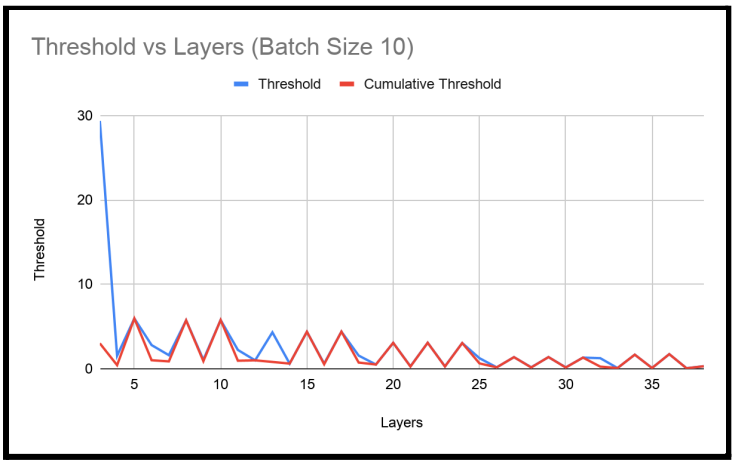


Fig. 4.3 (Batch size = 10)

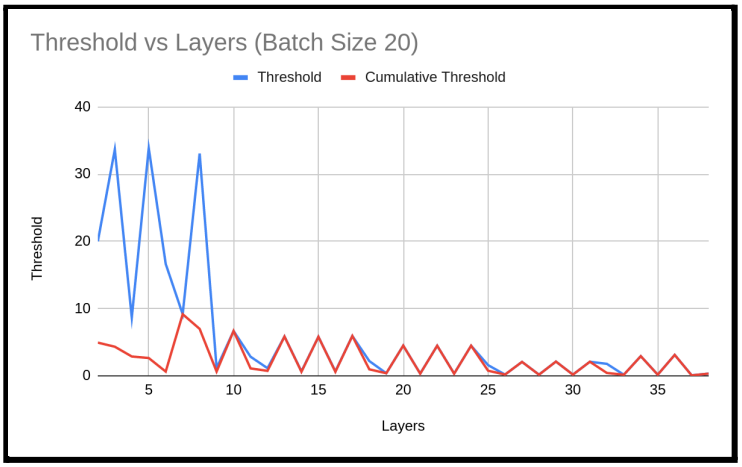


Fig. 4.4 (Batch size = 20)

Figures 4.1 to 4.4 show the values of Threshold and Cumulative Threshold values for different layers of our model consisting of 37 layers (VGG-16).

Profile

```
t_cumulative_compute = 0
layer[0] → is_ckpt = True
for i = 0 → L do
|       forward_timer → start() // start timer
|       layer[k] → Forward()
|       Forward_time[i] = forward_timer → stop() // record forward time of each layer
|       t_cumulative_coumpute += forward_time[i]
|       Transfer_timer → start() // start
|       timerlayer[i] → TransferToHost()
|       transfer_time[i]=transfer_timer → stop() // record transfer time of each layer
|       if transfer_time[i]/t_cumulative_coumpute < 1.0 then
|       |       layer[i] → is_ckpt = True
|       |       t_cumulative_compute = 0.0
|       else
|       |       layer[i] → is_ckpt = False
|       end
end
```

Although a profile-based method is helpful in determining the time cost of memory optimization methods, it is impractical in some distributed multiple GPUs-based

environments, because of the huge number of GPU nodes with different resources, the variety of the CNN workloads, and the limitation of the interconnect bandwidths.

Algorithm 2:

Forward Pass

```
for current_layer in layers do
|       current_layer→allocateOutput()
|       if current_layer is a checkpoint then
|       |       Synchronize(transfer_stream)
|       |       if current_layer greater than the second checkpoint then
|       |       |       Previous-to-previous checkpoint→freeOutputMem()
|       |       end
|       |       if current_layer is not the first checkpoint then
|       |       |       Previous checkpoint→TransferDeviceToHost(transfer_stream)
|       |       end
|       end
|       if current_layer is the first layer then
|       |       current_layer→copy_input_batch()
|       end
|       if current_layer is the last layer then
|       |       current_layer→copy_output_batch()
|       end
|       current_layer→Forward(compute_stream)
|       Synchronize(compute_stream)
end
Synchronize(compute_stream)
Synchronize(transfer_stream)
```

The forward pass is implemented as per the above algorithm. For all layers that are not checkpoints, the output buffer is dropped immediately when it is no longer needed. For checkpoint layers, the output buffer is first transferred to the host before it is freed, and is retrieved later during the backward pass. All computations occur on a separate cudaStream **compute_stream**, while all transfers occur on a cudaStream **transfer_stream**. These streams are declared as a part of the Model class. The checkpoint layer pointers and indices have been obtained during profiling.

Points to note:

- When a layer's forward computation is to be done, the memory for the input and output buffers must have been allocated. Since we call allocateOutput() **before** the forward pass computation begins, both buffers will have been allocated by the time Forward() is called. [The input buffer will have already been allocated by the previous layer in its call to allocateOutput()]
- allocateOutput() also **frees the input buffer** memory of the previous layer [if the buffer is not the output of a checkpoint layer], since it is no longer needed for the rest

of the forward pass. Checkpoint layers' output buffers are freed using the freeOutputMem() function.

- The outputs of checkpoint layers are transferred to CPU memory using a separate transfer stream, which can occur **asynchronously** with the compute stream. The transfer stream is synchronized with the host only at checkpoints. This means that the device to host transfer occurs in parallel with the forward computation of the following layers till the next checkpoint is reached.
- When we reach a checkpoint, we start the **previous** checkpoint's output buffer transfer. This ensures that the **last** checkpoint's output buffer remains in GPU memory, and thus will not need to be brought back from the CPU at the beginning of the backward pass.
- At each checkpoint, the output buffer of the **previous-to-previous** checkpoint is freed. This is done since the transfer of this output buffer is guaranteed to have been completed due to the synchronization of the transfer stream with the host.

Backward Pass

```
cur_cktpt = LastCheckPoint
next_cktp = LastLayer
for i = L → 0 do
|       if PreviousCheckPoint exists then
|       |       PreviousCheckPoint → TransferHostToDevice(transfer_stream)
|       for k = cur_ckpt+1 → next_ckpt do
|       |       layer[k] → AllocateBackward()
|       |       layer[k] → Forward()
|       end
|       for k = next_ckpt → cur_ckpt do
|       |       layer[k] → AllocateGradients()
|       |       layer[k] → Backward()
|       end
|       Synchronize(transfer_stream);
|       Synchronize(compute_stream);
|       if next_ckpt == LastLayer then
|               calculateLoss()
|       for k = cur_ckpt+1 → next_ckpt do
|       |       layer[k] → FreeMemory()
|       end
|       next_ckpt = cur_ckpt
|       cur_ckpt = PreviousCheckpoint
end
```

We proceed to implement our Backward Pass as mentioned in the above-mentioned algorithm. We start by iterating through the checkpoints which were generated during profiling.

Salient Features:

- The extra forward computation is done between the checkpoints where by profiling we found out that the time taken by the transfer from CPU to GPU **took more time** than implementing cumulative forward passes between two successive checkpoints.
- We notice that for doing the backward pass we need the values of input feature maps(*in_batch*) as well as the output feature maps(*out_batch*) for some layers however we can only retrieve the ***out_batch*** we had stored during the forward pass. Thus we carry out the **extra forward computation** from the layer after the current checkpoint layer to the next checkpoint layer thus eliminating the need for *in_batch* for the current checkpoint layer which is not available yet.
- Our main use of parallelization is made now as we realise that during the time taken for extra forward computation we can **asynchronously** fetch the previous checkpoint's output feature maps(*out_batch*). This activity is carried out asynchronously in another cudaStream which in our case is **transfer_stream**. Our default stream is the **compute_stream**.

Our Backward Pass can be described as following

1. We start the forward pass from the layer after the current checkpoint layer to the next checkpoint layer and compute the feature maps.
2. We asynchronously fetch the output feature map of the previous checkpoint and keep it in our current memory.
3. To ensure that the computation is completed we synchronize both the streams and then we proceed to the previous checkpoint.

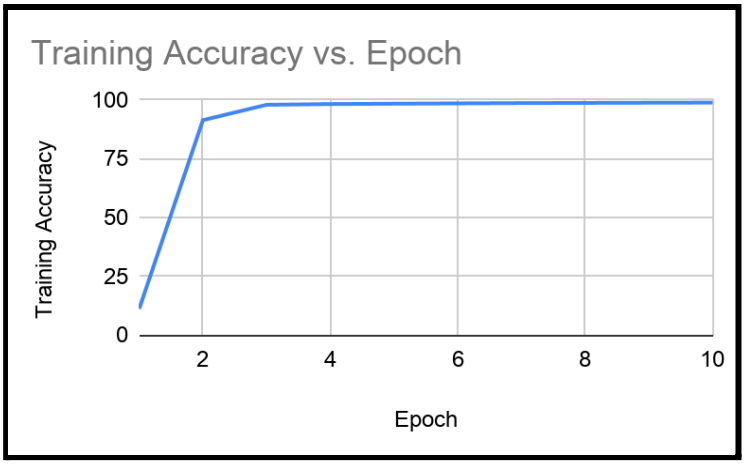
Some points which we need to keep in mind

1. Since we had dropped the output feature maps during the forward pass, we need to **allocate the memory** for each layer's variables. However, the links between out_batch of the current layer and in_batch of the next layer are not maintained and thus we **need to link** them as well.
2. After running the backward pass between two checkpoints we need to **free all the extra memory** or else it will soon run out of memory. However, this can also result in dropping the outputs for the output layer which is used to calculate the loss and so we need to either **save it in another variable or calculate it in the same function**.

Evaluation:

We compare the performance of our approach with two major approaches which can be considered as the oracle of our approach for the **GPU Memory** requirement (Transferring the feature maps for every layer to CPU) and **Compute Efficiency** (Pre-allocate memory in the GPU and do not drop any feature map). **Our approach strikes a balance between the two**. Note: The feature maps indicated below include input and output features, input and output gradients.

Sanity check of the forward and the backward pass: We perform a simple evaluation on MNIST Dataset using a simple 5-Layer CNN with checkpointing enabled since training on VGG was difficult due to the lack of dedicated GPUs. We observe a sharp rise in accuracy using the above-mentioned toy model.



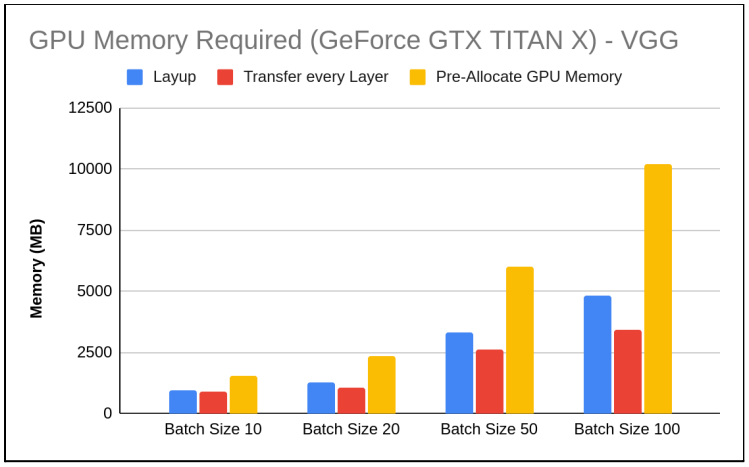
Description of the baseline approaches:

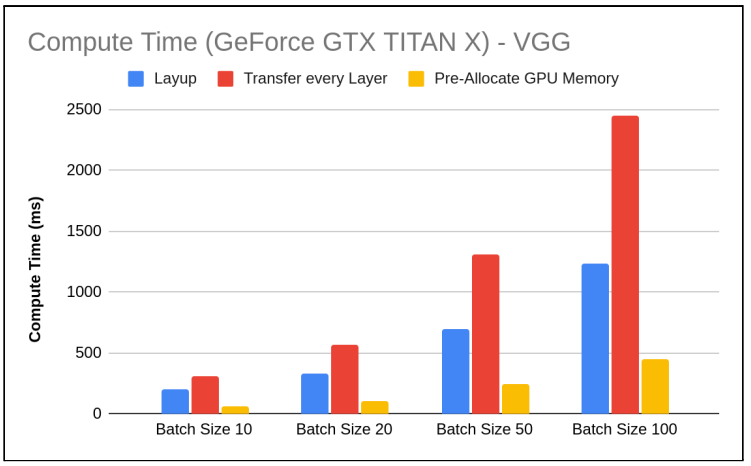
a) Transfer the feature maps of each and every layer to the CPU during the forward pass asynchronously. For Backward pass, while computing gradients for layer *'i'* transfer the features of layer *'i-1'* from CPU in parallel. This method uses the bare **minimum GPU memory** required to perform the forward and backward pass in a single GPU without model parallelism.

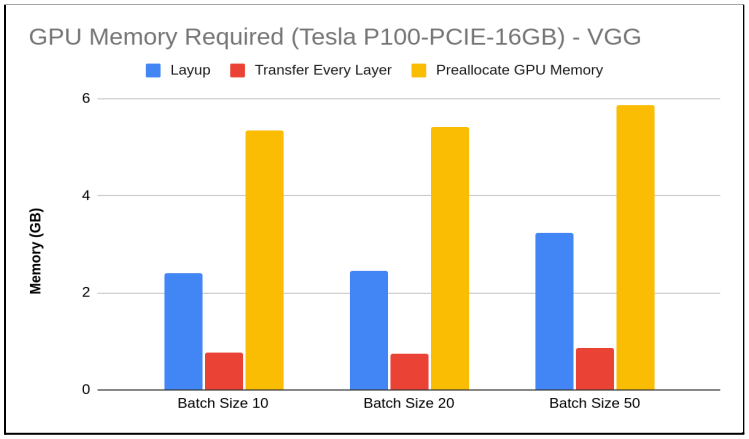
b) Pre-allocate the GPU memory for all the features maps during the initialization of the model. This way we do not waste any time transferring/recomputing the features during the forward/backward pass. This method takes the **minimum time required** to complete the forward and backward pass using a single GPU without model parallelism.

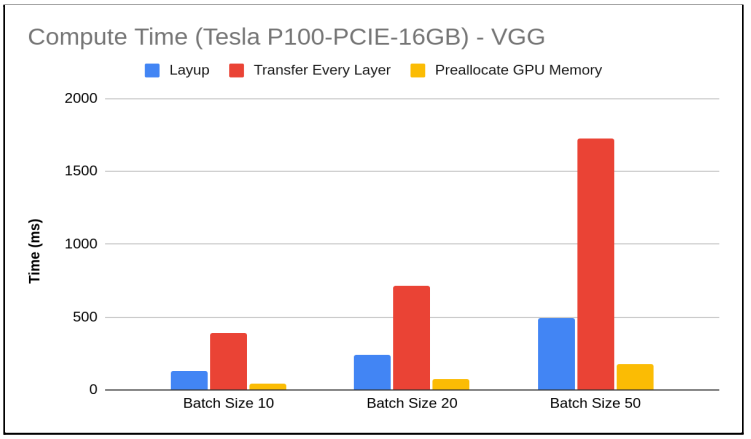
Note:

We evaluate the performance using two different GPUs - **GeForce GTX TITAN X-12Gb and Tesla P100-PCIE-16GB on VGG Model.**



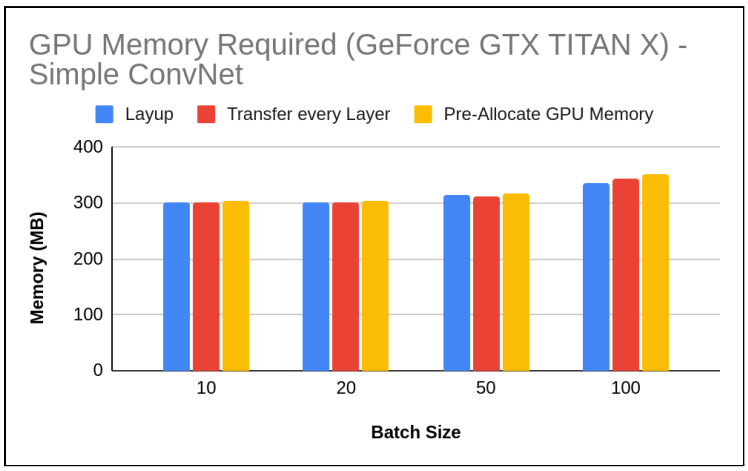


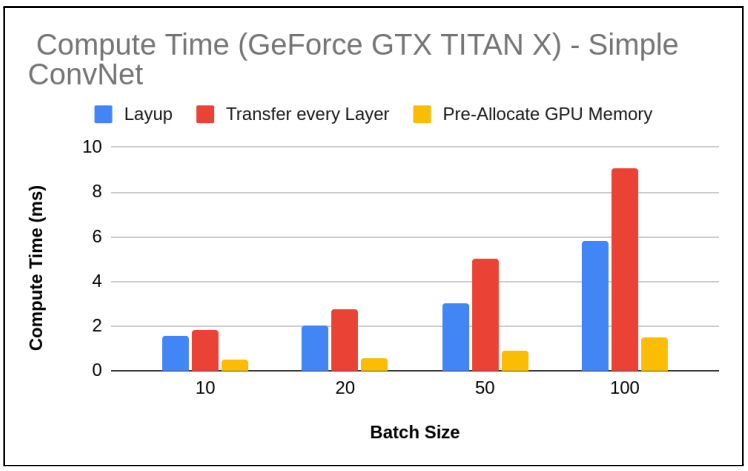




We notice that our approach strikes a perfect balance between the two. Transferring every layer gives us an advantage of utilizing very little GPU memory, but the compute time footprint of this method is quite high. Pre-allocating memory for all the feature maps has a compute time advantage but requires a lot of GPU memory, since a large amount of GPU memory isn't reused. We use a profiling based approach similar to Layup to dynamically decide which layers must be transferred to the CPU and which layers' feature maps must be freed. This approach is not as slow as complete layer transfer and also does not demand heavy GPU memory.

Next, we try to answer the question, what about the tradeoff in **smaller models**? We have shown in the results above that in the case of deep models which throw GPU budget constraints, our approach is a much more feasible option. However, that is not the case for smaller models. The computational advantage of pre-allocating GPU memory and not reusing it (no CPU-GPU transfer) clearly outperforms our method. However, the memory optimization becomes more significant as the memory footprint scales with the batch size.





Conclusion:

Following the Layup paper, we have successfully demonstrated the trade-off between Extra Forward Computation and CPU-GPU Transfer as optimisations in training in CNNs.
We have also addressed the ambiguities in the original paper.
Thus, using our approach we can train big neural networks using a lower GPU budget without sacrificing precious compute speed.

Appendix:

Description of VGG

Layer Num	Layer Description
0	Input (24 x 24, 3 channels)
1	Conv2D (64 kernels, 3x3 kernel dim, stride = 1, padding = 100)
2	Max Pool (2 x 2)
3	Relu
4	Conv2D (64 kernels, 3x3 kernel dim, stride = 1)
5	Relu
6	Max Pool (2 x 2)
7	Conv2D (128 kernels, 3x3 kernel dim, stride = 1)
8	Relu
9	Conv2D(128 kernels, 3x3 kernel dim, stride = 1)
10	Relu
11	MaxPool (2x2)
12	Conv2D(256 kernels, 3x3 kernel dim, stride = 1)
13	Relu
14	Conv2D(256 kernels, 3x3 kernel dim, stride = 1)
15	Relu

16	Conv2D(256 kernels, 3x3 kernel dim, stride = 1)
17	Relu
18	MaxPool (2x2)
19	Conv2D(512 kernels, 3x3 kernel dim, stride = 1)
20	Relu
21	Conv2D(512 kernels, 3x3 kernel dim, stride = 1)
22	Relu
23	Conv2D(512 kernels, 3x3 kernel dim, stride = 1)
24	Relu
25	MaxPool (2x2)
26	Conv2D(512 kernels, 3x3 kernel dim, stride = 1)
27	Relu
28	Conv2D(512 kernels, 3x3 kernel dim, stride = 1)
29	Relu
30	Conv2D(512 kernels, 3x3 kernel dim, stride = 1)
31	Relu
32	MaxPool(2x2)
33	Dense(4096)
34	Relu
35	Dense(4096)
36	Relu
37	Dense(10)
38	Output Layer (SoftMax CrossEntropy)

Statistics Sheet:
https://docs.google.com/spreadsheets/d/1c1vOtQ-HIxpws8HPsDk61n02IBQJaIPDDdqtjaq6ISs/edit?usp=sharing

Code Link:
https://github.com/ShahRutav/Layup-HP3

References:
Layup: Layer-adaptive and Multi-type Intermediate-oriented Memory Optimization for GPU-based CNNs
1604.06174] Training Deep Nets with Sublinear Memory Cost
CS 179: GPU Programming
Convolutions with cuDNN – Peter Goldsborough
https://papers.nips.cc/paper/2016/file/a501bebf79d570651ff601788ea9d16d-Paper.pdf