Efficient Data Partitioning for Deep Neural Networks Training

Ajay Krishna A
EID : aa78737
University of Texas at Austin
ajaykrishna1111@utexas.edu

Saurabh Gupta
EID : sg47384
University of Texas at Austin
saurabhg@utexas.edu

Reshma Rajarama Nayak
EID : rn7998
University of Texas at Austin
reshma.rnayak@utexas.edu

***Abstract*—Deep Neural Networks (DNNs) have almost become ubiquitous recently and is consistently finding its place into many interesting domains such as image recognition, voice recognition, object detection, pattern recognition and is being deployed commercially in smart speakers, self-driving cars, search pattern completion, etc. Training of these useful huge DNNs is extremely computationally intensive and parallel processors such as GPUs are being utilized to speed-up the task. The scheme of data and task partitioning among these parallel Processing Units (PUs) directly impacts the performance boost achieved by such parallelization. This work studies the trade-offs involved in various partitioning schemes and proposes an architecture agnostic efficient approach that combines the advantages of the commonly used schemes for effective performance boost up.**

***Index Terms*—Data parallelism, Model parallelism**

I. Introduction

Over the past few years, Deep Neural networks (DNNs) have revolutionized various domains including image recognition [1, 2], natural language processing [3], pattern recognition [4], object detection [5], voice recognition [6] etc. In few of these domains, DNNs has begun to exceed human performance. For e.g., human top-5 classification error rate on the large scale ImageNet dataset has been reported as 5.1% [7]; Ensemble of Resnet models [8], which won the ILSVRC 2015 classification task, reported an accuracy of 3.57%. Such high accuracy level is made possible by the ability of the DNNs to extract relevant features from the raw input, as opposed to the hand picked features used in previous approaches. The extraction of superior multi-level features typically requires networks of huge depth and more number of channels. Recent evidences [9, 10] reveal the importance of network depth for the accuracy. Leading results in ILSVRC 2015 [9, 10, 11, 12] employ neural networks of depth of upto 30; newer models have depth in the range of 100-1000 layers [8]. This report discusses the challenges involved in training such deep neural networks and proposes the energy efficient way to get the maximum performance in GPU accelerated implementation of training of these networks. This report is organized as follows:

- Section II discusses the motivation behind the focus on acceleration of the DNNs training.
- Section III gives an overview of common approaches used for acceleration of DNN training and what our work focuses on.
- Section IV covers the general data partitioning schemes employed in the training.
- Section V describes the key concerns for the chosen evaluation platform and how the proposed work addresses it.
- Section VI provides details on the implementation methodology and experimental setup.
- Section VII includes the results and key inferences from the observations.

II. Background and motivation

DNNs consists of stacks of layers, where each layer performs a non-linear operation on the weighted sum of the inputs. Different layers take different section of their inputs for their operation - for instance, fully-connected layer (Fig. 1) uses all the inputs, whereas convolutional layer (Fig. 2), to utilize the spatial locality in the input feature map, uses only a subsection of input defined by the dimension of the filters. These weights and filters need to be fine tuned for

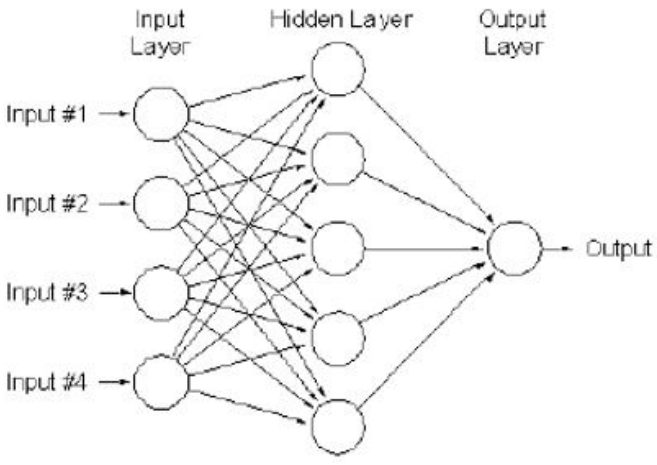

Fig. 1. Fully connected layer

the application the neural network is being used for; which is termed as *training* of the neural network. Once the weights are tuned for the application, it can be used for the purpose it is designed for, termed as *inference*. Typically, training of the DNNs is done by backpropagation (BP) algorithm with an optimization method such as gradient descent [12].

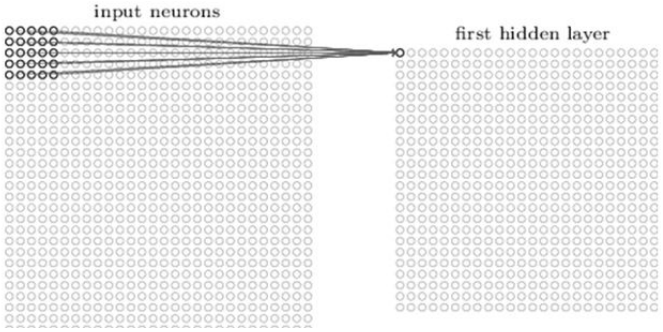

Fig. 2. Convolutional layer

The training is posed as a minimization problem of error as a function of weights. In supervised training, a training dataset consists of huge sets of input and expected output pairs. Each input (In) propagates through the network in the forward direction and is compared against the expected output (Out_exp) to compute the loss function L. Forward propagation of a layer l with input tensor X_l, weight tensor F_l and output tensor O_l is given by:

$$O_l = f(X_l * F_l) \tag{1}$$

where * represents convolution or multiplication operation for convolutional layers and fully-connected layers respectively, f is the non-linear activation function. The computed error is then propagated in the reverse direction through the layers for updating the weight tensors. This involves two steps :
Step 1 : Each layer propagating the derivative of the error $dL/d0_l$ with respect to its output 0_l to its input X_l. This is computed using the chain rule :

$$\frac{dL}{dX_l} = \frac{dL}{dO_l} . \frac{dO_l}{dX_l} \tag{2}$$

where,

$$\frac{dO_l}{dX_l} = f'(X_l * F_l) * F_l \tag{3}$$

where f'(x) is the derivative of f(x) w.r.t x.
Step 2 : Each layer updates its weights F_l based on its input X_l and backpropagated error gradient dL/dZ_l.

$$\frac{dL}{dF_l} = \frac{dL}{dO_l} . \frac{dO_l}{dF_l} \tag{4}$$

where,

$$\frac{dO_l}{dF_l} = f'(X_l * F_l) * X_l \tag{5}$$

$$F_l += \eta . \frac{dL}{dF_l} \tag{6}$$

where η is the learning rate, which determines the size of the step taken in the gradient descent algorithm. This process is diagrammatically illustrated for a convolutional layer in Fig. 3. Thus, training the network involves three phases - forward propagation (FP), backpropagation (BP) and kernel update (KU). Key considerations in DNN training are the following :

- As the networks grow deeper and wider (more number of channels), the amount of weights to be trained increases. This necessitates large datasets and more number of epochs to converge.
- Each phase in training has about same amount of computations as the forward propagation. Thus, each epoch of training has about 3 times more computations than inference.
- The backpropagation (BP) of the network needs the output of the intermediate layers for the input gradient propagation. This implies that training has increased storage requirement.
- Training with gradient descent requires high precision for quick convergence.
- Weight kernel update is done once for a set of input data (batch) with average of the accumulated weight gradient to speedup as well as stabilize the training process.

Huge datasets and large number of epochs involved in training the deep networks make training very computationally intensive process. Also, with recent trend of Federated learning [13] - to move portions of training to the EDGE device to address the security concerns and with advent of models designed for mobile devices [14] to offer anytime, anywhere access, the energy efficiency of the training process also demands proper attention. These factors motivates us to explore the DNN training problem space and look for an optimal solution from both performance and energy perspective. GPUs have been the most commonly used platform for DNN acceleration in the past decade. Hence, this work utilizes GPU as the evaluation platform for the exploration.

III. Related work

DNN acceleration have been an active research field in recent years. These works mainly focuses on the following concepts of DNNs: (i) DNN inference and training algorithm provides immense opportunity to parallelize, (ii) Inherent resiliency of the algorithms for errors, (iii) Reusability of the tensors, (iv) Dataflow optimization opportunity exploiting the spatial locality. Many research work focussed on accelerating the inference of DNNs. Han et al. [14] proposed pruning of the weights to make weight tensors sparse, thereby, reducing the overall memory bandwidth consumption; also showed better performance and lower power by exploiting the sparsity of activation inputs. Alwani et al. [15] proposed fusion of the concurrent convolutional layers to reduce the global memory bandwidth and also improved performance over a bandwidth constrained system.
Many of these concepts is not directly applicable to the training problem space as it poses a different set of challenges. For e.g., training requires the intermediate layers' output during backpropagtion, hence, fusion of convolutional layers might not provide significant benefits as it does not reduce the global memory bandwidth in training, compared to inference. Acceleration of DNN training is being focused in few of the recent research works, especially those exploiting different forms of

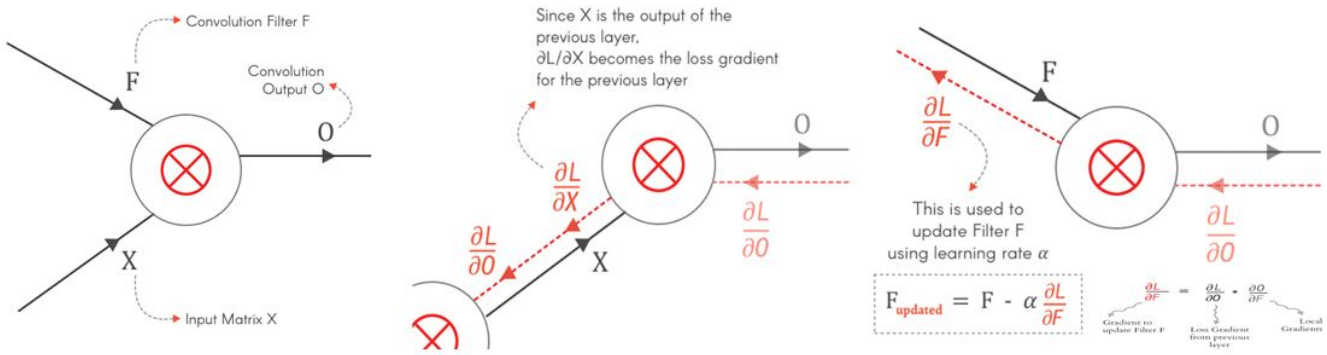

Fig. 3. Training of convolutional layer

parallelism available during training. Krizhevsky et al. [1] implemented a form of data parallelism, where the batches were divided among two GPUs for training. Dean et al. [17] split a DNN into disjoint subsets and trains each subset on a dedicated device. This approach reduced the communication costs for synchronizing network parameters. Wu et al. [18] split up the batches among GPUs and switched to model parallelism within a GPU. Gao et al. [19] exploited parallelization on tiled domain-specic hardware with additional optimizations to minimize communications among the multiple devices. Zhihao et al. [20] proposes a framework for identifying the efficient partitioning scheme to obtain best performance. Clearly, the recent research trend in DNN training have been to identify the best partitioning scheme.

IV. Data partitioning schemes

With the approach to accelerate the DNNs training with lot of parallel Processing Units (PUs), the way the data is partitioned among the PUs affect the performance benefit obtained significantly. The commonly deployed form of partitioning are discussed below: As different partitioning schemes has different characteristics such as memory footprint, synchronization cost,

A. Data Parallelism

Data Parallelism (DP) explores the parallelism in the input feature space domain. The batch is subdivided to mini-batches and distributed among the PUs. As shown in Fig 4 for a simple 2 PUs scenario, each PUs can independently compute the output, hence there is no synchronization or reduce operation required in the forward propagation. This also implies that all the PUs would be using the same weight kernel in the forward and backward direction. Similarly, back-propagation does not require any synchronization. However, the weight needs to be updated by the average of the accumulated gradient. Hence, Kernel Update step requires all PUs to synchronize and share the gradients amongst each other as shown in Fig 5.

B. Model Parallelism

Model parallelism (MP) explores the parallelism available in the single layer. For e.g., Convolutional layers typically

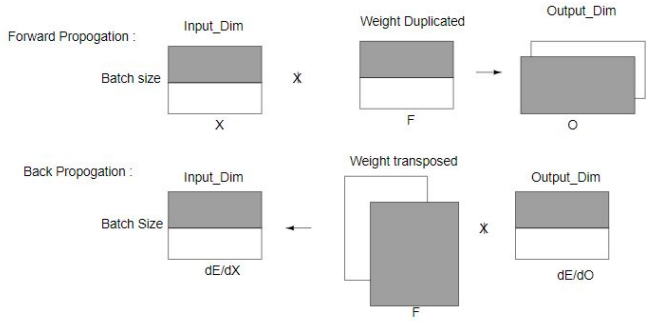

Fig. 4. Data Parallelism - Forward and back propagation

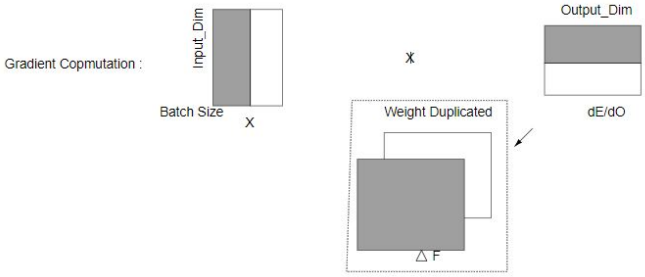

Fig. 5. Data Parallelism - Kernel update

have multiple output channels and each can be computed in parallel by reading in only a subset of the weight tensors. Model parallelism can also be visualized as the weight tensors being split among the PUs and data partitioned accordingly. In this scheme of division, depending on the way the weight tensor is partitioned (e.g. input channels partition or output channel partition), one of the forward or the reverse path would demand synchronization among the PUs and communication of partial tensors between them. One option of dividing the output channels of a convolutional layer is shown in Fig. 6 among 2 PUs, which requires synchronization in the back propagation. The Kernel Update, however, will not require any communication or synchronization as the weight tensor is subdivided among the PUs as shown in Fig. 7.

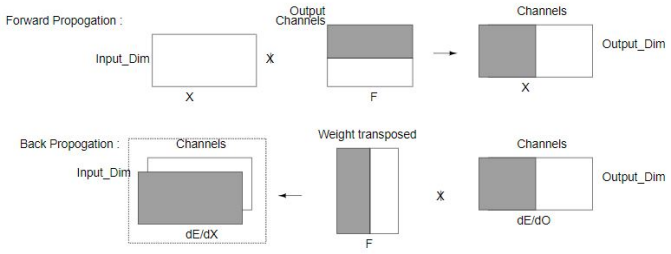

Fig. 6. Model Parallelism - Forward and back propagation

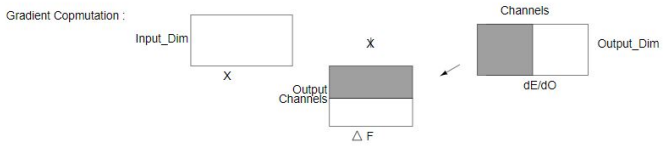

Fig. 7. Model Parallelism - Kernel update

V. Proposed Work

Typical NVIDIA GPUs have the following compute hierarchy - cluster of Streaming Multiprocessors(SMs) - each containing many CUDA cores and following memory hierarchy - each SM has a private local memory (shared physically between L1 and shared memory in Kepler architecture and separate in Pascal architecture), L2 cache shared among SMs. The Shared memory is a programmable local memory, which programmer can use to improve reuse for irregular cache-unfriendly tensor accesses. The shared memory is limited to 64KB per block in recent architectures; this sets the limit for the programmable re-usability of data. The GPUs, typically, have global memory of around 12GB. This limits the model size and batch size that can be used for training in GPU. GPUs memory is optimized for bandwidth, rather than latency; GPUs rely on latency hiding by warp and block switching when the current warp is stalled due to cache misses. Thus, the important goals that this work focused for GPU optimization of training are achieving good GPU occupancy (number of SMs doing useful work), efficient reuse of tensors from shared memory, global memory footprint reduction and efficient latency hiding.

Data parallelism scheme divides the samples in the batch sizes between SMs. To achieve the GPU occupancy goal, the batch size needs to be significantly large. Training requires the intermediate layers' values also to be retained in memory for each individual sample. Hence, increasing batch size contradicts to the goal of lower memory footprint. Also, with huge dimensionality of weight tensors, division of batches would not lead to efficient reuse.

Model parallelism scheme achieves the intra-layer parallelism by dividing the weights between the SMs. Since, the weights are partitioned, it is small enough to fit in the shared memory and thereby increase the reuse. However, typical neural network architectures have few layers which cannot fully fill in the GPU compute resources, thereby achieving poor GPU occupancy.

The proposed hybrid approach fuses the two complementing data and model parallelism schemes - add on the batch size parallelism on top of the existing intra-layer parallelism. This approach (1) will be able to occupy GPU better than the MP scheme, (2) still retain reuse benefit because of weight partition, (3) have smaller footprint than equivalent DP scheme implementation, (4) have better latency hiding by providing more blocks, thereby providing more warps for GPU to context switch.

VI. Experimental Methodology

A. Implementation

1) Implementation of various partitioning schemes :

(a) *Data Parallel implementation:* In this scheme, the data samples in a batch is partitioned between different SMs. Each block processes one data sample.

(b) *Model Parallel implementation:* In this scheme, the weight tensor is partitioned between different SMs . In the current implementation , the output channels is divided across different SMs. Each block works on a single output channel. The partitioned weight matrix can now fit into the local shared memory, which can help eliminate the traffic to the higher levels of memory hierarchy. Since the weight matrix is reused for convolution across the input feature map, we expect significant performance benefit with the use of shared memory for storing the weight tensor.

(c) *Hybrid implementation:* Here, we divide both the weights and the batch across different SMs. Each block processes one channel of one sample of the batch. Since each block operates on one channel, we still leverage the advantage of shared memory usage for reducing latency of memory accesses. The number of blocks scale up as the batch size * output channel dimension increases. Hence, better GPU occupancy and efficient latency hiding possible with reasonable batch size.

2) CUDA Primitives - building blocks for DNNs: We developed configurable primitive functions in CUDA, supporting data and model parallelism flavors for convolutional, average, max pooling and fully connected layers. These function automatically chooses the best configurations for kernel launch based on the tensors dimensions and batch size. It supports logistic and RELU activations. The initial workspace is setup in GPU by yet another parametrized layer define functions.

The parameters for a sample primitive method for convolutional layer is described below.

Convolutional Forward Propagation Primitive Method definitions

```
conv_forward_pass (N, C, H, W, K, R, S,
pad_H, pad_W, str_H, str_W, out_H, out_W,
function, input, weight, bias, output)
```

N - Batch size
C - Input channels

H - Input feature height
W - Input feature width
K - Output channels
R - Filter Height
S - Filter Width
pad_H - Vertical Padding dimension
pad_W - Horizontal Padding dimension
str_H - Horizontal stride
str_W - Vertical Stride
out_W - Output feature Width
out_H - Output feature height
function - Activation function - Logistic/RELU
input - Input feature map
weight - weight matrix for the layer
bias - layer bias
output - Output feature map

Similar to the sample above, different primitive functions are developed for different layers in the network like pooling and fully connected. These primitive functions can be invoked using parameters corresponding to different layers, which can be stitched together to build the complete neural network.

Using these CUDA primitives, we build data parallel, model parallel and hybrid versions of three different CNN architectures:- Lenet-5, Alexnet, Resnet-18.

B. Experimental Setup

1) Neural Network Architectures: We evaluate the performance of data parallel, model parallel and hybrid versions of Lenet-5, Alexnet and Resnet-18.

(a) *Lenet-5:* LeNet-5 is one of the basic networks containing 7 layers. It was developed in 1998 in order to recognize handwritten digits from the MNIST dataset [21]. Fig.8 shows the structure of LeNet-5. It has two sets of convolutional and sub-sampling layers followed by fully connected layers in the end.

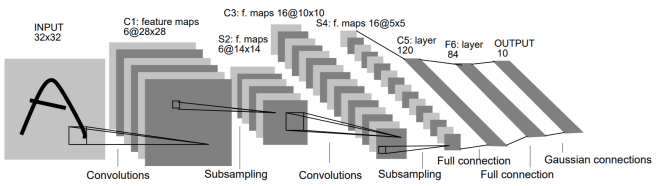

Fig. 8. LeNet-5 structure

(b) *AlexNet:* AlexNet employs much deeper filters compared to LeNet-5. The maximum number of output channels reaches as high as 384. It reduced the top-5 error rate for ImageNet dataset to 15.3% and emerged as the ILSVRC 2012 competition winner. The structure for the AlexNet is shown in Fig. 9.

(c) *Resnet-18:* ResNet was proposed in 2015 and emerged as the ILSVRC 2015 competition winner. It uses concept of residual blocks which employs a "skip connection (identity function)". A typical residual block structure is shown in Fig. 10. This structure ensures that the accuracy of the network doesn't go down by adding more convolution layers. The structure was able to obtain a top-5 accuracy of 3.57% which beats human level performance. We create a 18 layer model of this approach which is popularly also known as ResNet18.

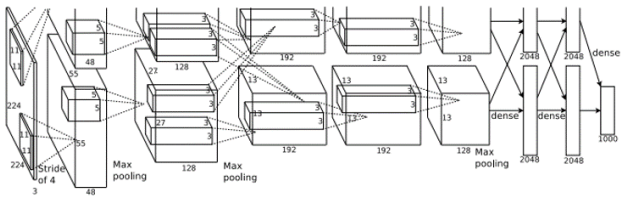

Fig. 9. AlexNet structure

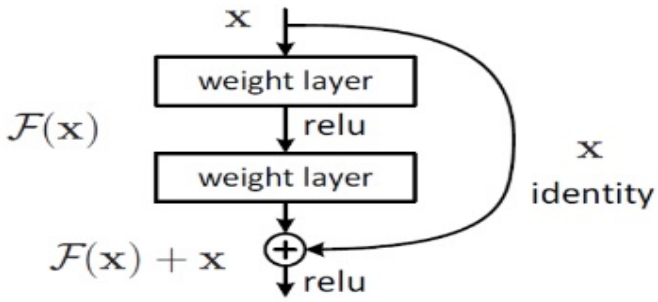

Fig. 10. Residual block structure of ResNet

The number of parameters and computations for the neural network layers are shown in Table I

TABLE I
NO OF PARAMETERS AND COMPUTATIONS IN NEURAL NETWORKS

Architecture	*No of Params*	*No. of computations (FLOPs)*
LeNet-5	60840	3.41×10^5
AlexNet*	28567498	1.08×10^8
ResNet-18*	10862986	1.13×10^7

*Scaled down versions.

MNIST [21] and CIFAR-10 [22] datasets are used for evaluating the performance of these networks.

2) Tools: We use Nvidia Visual Profiler(nvprof) to evaluate the performance of the three neural network applications. Nvprof has the capability of profiling the applications to get kernel wise metrics on execution time, memory transfers, dependency analysis, compute resource utilization etc. These metrics were used to understand the bottlenecks in execution performance for the three networks.

Nvidia System Profiling tool enables the sampling of power, clock and thermal behaviour of the CPU. The profiler samples these parameters during different intervals which is then averaged to provide the final metrics. We collect the application's power using the system profiling. Power is multiplied by

the execution time to obtain the energy of the application execution.

3) Hardware Platforms: We evaluate the neural network architecture implementations in three different hardware platforms. We choose devices with two different architectures namely Kepler and Pascal, to understand the performance difference across different architectures. Also, we study the performance difference for two devices of different capabilities within the same architecture. The configuration details of the three different platforms is shown in the table II

TABLE II
HARDWARE PLATFORM CONFIGURATIONS

Device	***Tesla K40s***	***GTX 1080 Ti***	***Tesla P100***
Architecture	Kepler	Pascal	Pascal
Number of SMs	15	28	56
Total global memory	11440 MBytes	11178 MBytes	16281 Mbytes
L2 Cache Size	1572864 bytes	2883584 bytes	4194304 bytes
Shared mem/block	49152 bytes	49152 bytes	49152 bytes

VII. RESULTS AND ANALYSIS

A. Performance Comparison

Fig. 11 shows the per epoch execution time for the AlexNet model with different batch size over the K40 machine. There are three important points to consider:

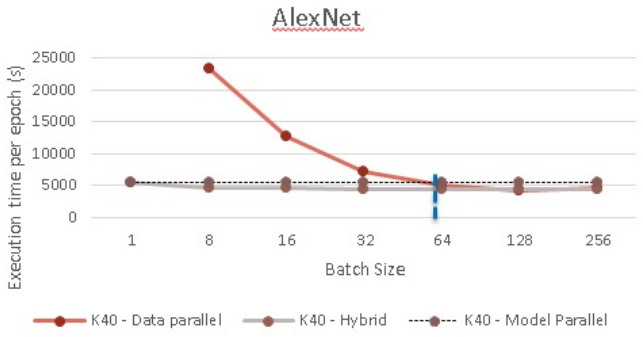

Fig. 11. AlexNet: Batch size sweep for execution time per epoch

(a) The data parallel approach performs poorly at low batch sizes. The execution time per epoch decrease sharply as we increase batch size from to 8 to 32. We the reach a saturation point as batch size is further increased to 64. This saturation implies that the GPU resources are fully occupied at this stage. Further increase in batch size provides diminishing returns but increases the memory footprint linearly.

(b) The model parallel approach isn't dependent on batch size and thus represents a horizontal line. This approach performs much better than the data parallel approach for the lower batch sizes. The data parallel implementation overtakes the model parallel implementation between batch size of 32 to 64. The explanation is as follows. As the batch size increases there is lot more work available for the GPU in data parallel approach and it's able to utilize it's resources fully. On the other hand, the number of channels in model parallel approach is fixed by the neural network architecture and isn't user configurable. Since some layers tend to have fewer channels than the other, the overall implementation becomes less than optimal.

(c) The hybrid approach starts with roughly the same performance as the model approach with batch size of 1. Further, this approach reaches saturation at batch size of as low as 8. The execution time per epoch at this saturation is better than the data parallel performance even at batch size of 128. We see two main reasons for the same. The hybrid approach take the benefit of model parallel implementation by storing weight matrix in local memory and thus achieving higher bandwidth and lower latency from the local memory. Second, as we increase batch size the it occupies the GPU fully which wasn't possible in regular model parallel approach. Thus it take benefits of both the approaches.

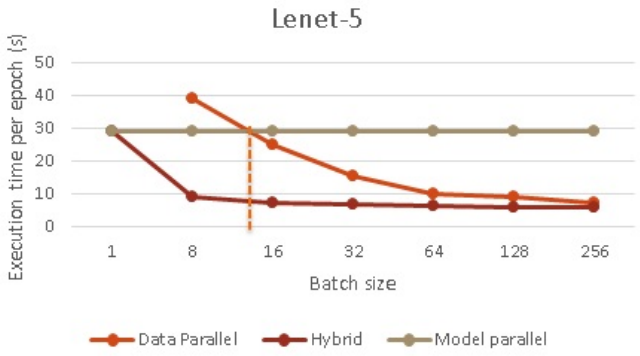

Fig. 12. LeNet: Batch size sweep for execution time per epoch

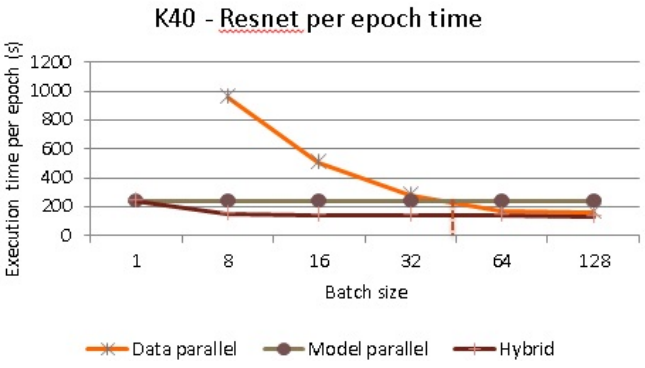

Fig. 13. ResNet: Batch size sweep for execution time per epoch

Fig. 12 and Fig. 13 show the similar metrics for Lenet-5 and ResNet18 model respectively. The ResNet18 has similar trend as AlexNet since both of them have deep channels. LeNet shows an interesting data point. Since it has only 6 output channels in its convolution layer, the model parallel

approach shows its limitation. The data parallel approach achieves crosses the model parallel performance for batch size of little less than 16. The benefit of the hybrid approach gets further pronounced here since it achieves the best performance irrespective of the neural network in hand.

B. Memory profiling

Fig. 14 shows the L1, L2, device memory and the shared memory loads for each of the 3 approaches. Compared to data parallel approach, the model parallel implementation has 30% lesser L1 cache loads and 14% lesser L2 cache loads. The device memory loads gets reduced significantly from 188k to 11k achieving 95% reduction. The reduction gets converted to shared memory loads. Since shared memory resides on-chip, it provides much higher bandwidth and lower latency,the number of data loads for the hybrid approach is equivalent to the model parallel approach which is as expected since hybrid approach too stores weights locally similar to model parallel approach.

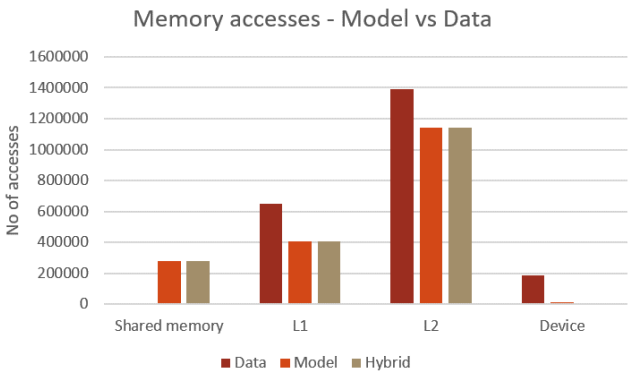

Fig. 14. Memory load comparison for data, model and hybrid schemes

C. Kernel Profiling

The reduction in memory accesses with model parallel approach is too overwhelming to achieve only <5% improvement with hybrid approach compared over data approach with higher batch sizes. We expected hybrid approach to maintain the benefit of fewer memory accesses even at higher batch sizes and show significant improvement over the data parallel approach at saturation. To dig further, we profiled each kernel of the AlexNet separately. Each kernel's execution time per epoch has been shown in Fig. 15. We observed two important points which are explained below:

(a) We see that the model parallel approach provides significant performance benefit for the initial convolution layers but performs worse as we reach to the last few convolution layers. The reason for the same can be explained based on the input feature size that the convolution layer works on. For the initial layers, the input feature map is of 32x32x(no. of channels) which reduces to 3x3x(no. of channels) as we go deeper crossing multiple pooling layers. Since a weight matrix is reused only 3x3 times in the final layers, this reuse benefit is overridden by the time taken by the threads to store data from global to shared memory. On the other hand for initial layers, weight matrix is reused 32x32 times. Thus, for the first layer we see a benefit of 25% from data to model approach but see a inverse trend for the end few layers. Cumulatively, we still get benefit from the model approach but not as much as expected.

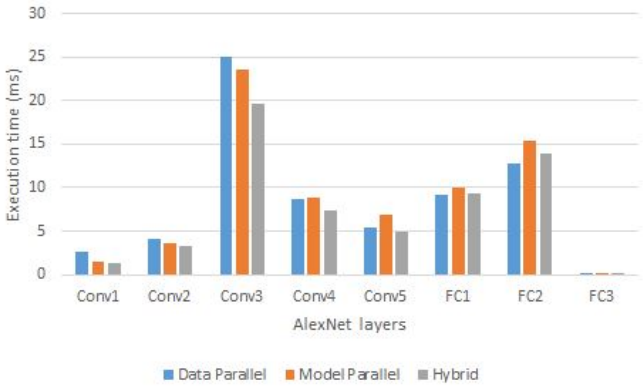

Fig. 15. Layer-wise profiling of AlexNet

(b) The fully connected layers perform much better with the data parallel approach. The reason is that model parallel approach for fully connected layers requires all GPU processing units to synchronize after each forward computation. On the other hand, data parallel approach doesn't pose any such condition, since each data sample can be processed independently by each processing unit. This in effect further reduces the overall benefit of model parallel/hybrid approach over the data parallel approach.

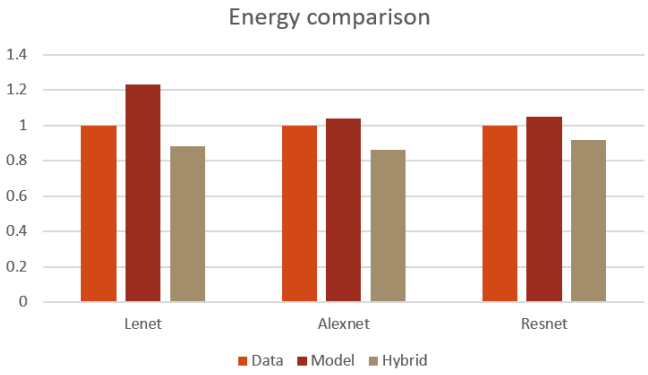

Fig. 16. Energy comparison for data, model and hybrid approach with data parallel approach as reference

D. Energy efficiency

Fig. 16 shows how the energy for model and hybrid approach compared with the data parallel approach. The numbers for model and hybrid approach are drawn relative to data parallel approach over K40 GPU. Both the data and hybrid energy numbers are obtained with batch size of 64 for sake

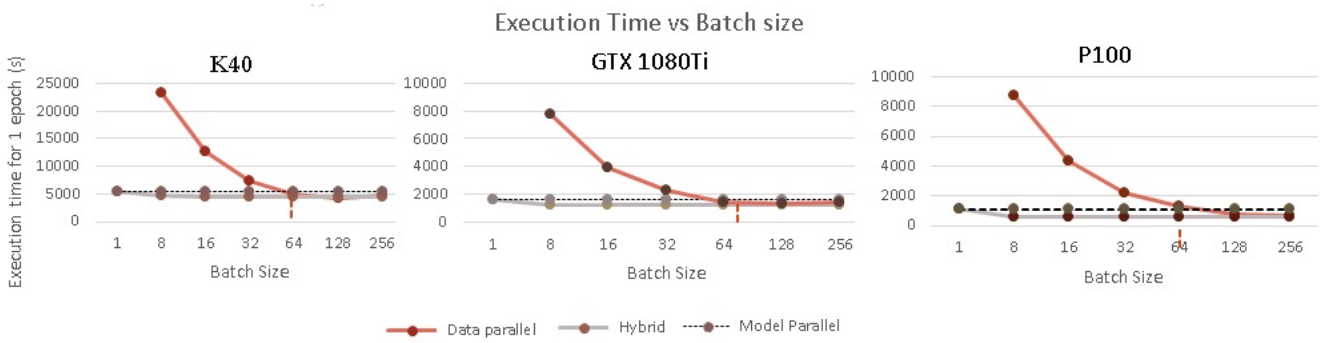

Fig. 17. Execeution time comparison of data, model and hybrid approach over the three different GPUs

of fair comparison. The absolute number are listed in Table III. We see that the hybrid approach edges out both data and model approach even in energy aspect. This can be reasoned as:

(a) The model and hybrid approach have much more references to the local shared memory which will lesser energy per access as compared to an access to device memory.
(b) The execution time for hybrid approach is better which lowers the overall energy expended per epoch.

TABLE III
ENERGY EXPENDED FOR EACH APPROACH PER EPOCH

Approach/Network	*LeNet-5*	*AlexNet*	*ResNet18*
Data	966.94 J	139866.4 J	62754.9 J
Model	1190.231 J	145218.5 J	65928.98 J
Hybrid	906.87 J	135866.2 J	58754.9 J

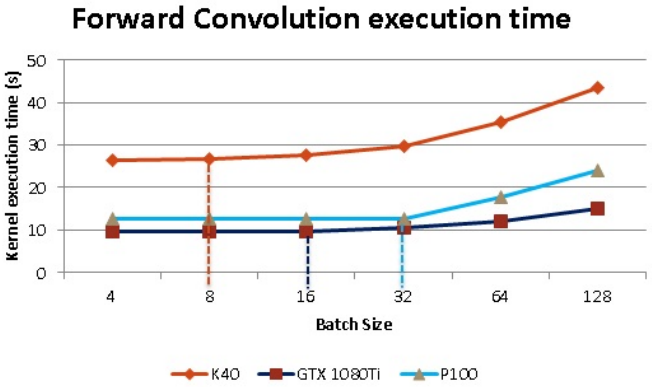

Fig. 18. Individual data parallel kernel execution time over three machines for different batch sizes

E. Micro-architectural considerations

We analyze the AlexNet network further over three different machines to understand the implications of the underlying hardware.

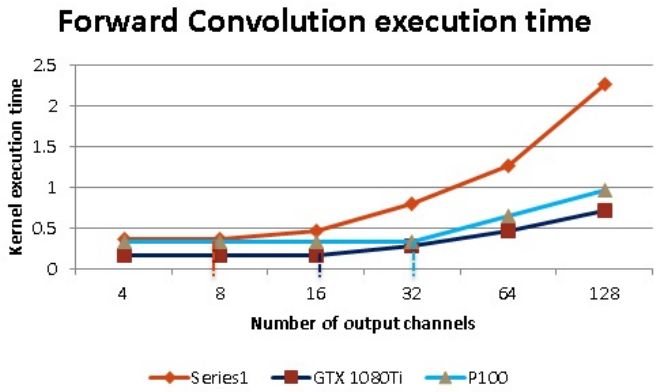

Fig. 19. Individual model parallel kernel execution time over three machines for different number of output channels

1) Data Parallelism: Individual kernel execution: Fig. 18 shows the execution time of the forward convolution kernel of AlexNet over the K40, GTX 1080Ti and P100 GPU. We see that the execution time of the kernel remains constant until we increase the batch size to 16, 32 and 64 on the K40, GTX 1080Ti and the P100 machine respectively. We explain this based on the number of available streaming multiprocessor units (SMs) on the different machines. K40 machine has 15 SMs whereas the GTX 1080Ti and P100 machine have 28 and 56 SMs respectively. To feed work to all the SMs, we need a minimum batch size equal to the number of SMs on the machine. Further, this is the minimum batch size to get all the SMs working. We would need even bigger batch size for the GPU to hide memory latency efficiently. This in turn causes the memory footprint to increase. Thus, although one might expect to gain better performance out of a bigger machine, we need to scale the memory size as well to keep the GPU busy.

2) Model Parallelism: Individual kernel execution: Fig 19 shows a sweep of output channels of a convolution layer over the three machines. We see a similar trend here as well. More and more number of output channels are required to actually occupy P100 machine which has higher number of SM units. Since number of output channels is fixed by neural network architecture, this isn't user configurable. Thus, by moving to

a better machine with an architecture with shallow channels (lower number of channels) will not lead to any performance boost.

3) Comparison with hybrid approach: Fig. 17 shows the overall execution time per epoch for the data, model and hybrid schemes for the three machines. We want to highlight two important observations here:

(a) We see that irrespective of the underlying hardware, the hybrid approach performs best among the three approaches. The reason is that first the output channels occupy the GPU as much as possible. If the GPU resources aren't occupied fully, additional batch size in hybrid approach is able to utilize them. Thus, for any given model and any underlying machine, hybrid scheme works well.

(b) In all the three machines, the hybrid approach saturates at much lower batch size and better performance as compared to the data parallel approach. This implies that we get optimal performance from the GPU at much lesser memory footprint. This can help to train much bigger networks over a single GPU itself which otherwise require multiple GPU and have much higher communication penalty due to the slower PCIe bus communication.

VIII. Conclusion

This work explored the common data partitioning schemes deployed in DNN training and proposed and evaluated the hybrid approach - which fuses the two complementing schemes. It is shown that the hybrid approach always performs better than the two reference schemes for all the three network architectures and three different machine configurations. Further, this work evaluated the schemes based on 32x32 input image size. With an image size of 224x224, the amount of reuse in the initial convolutional layers will significantly increase. Thereby, hybrid approach will see even more performance benefits.

References

[1] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." In Advances in neural information processing systems, pp. 1097-1105. 2012.

[2] Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 1-9. 2015.

[3] Lopez, Marc Moreno, and Jugal Kalita. "Deep Learning applied to NLP." arXiv preprint arXiv:1703.03091 (2017).

[4] Yun, Kyongsik, Alexander Huyen, and Thomas Lu. "Deep Neural Networks for Pattern Recognition." arXiv preprint arXiv:1809.09645 (2018).

[5] Zhao, Zhong-Qiu, Peng Zheng, Shou-tao Xu, and Xindong Wu. "Object detection with deep learning: A review." IEEE transactions on neural networks and learning systems (2019).

[6] Hinton, G., L. Deng, D. Yu, G. Dahl, and A. Mohamed. "rahman, Jaitly N, Senior A, Vanhoucke V, Nguyen P, Sainath T, Kingsbury B. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups." IEEE Signal Processing Magazine 29 (2012): 82-97.

[7] Russakovsky, Olga, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang et al. "Imagenet large scale visual recognition challenge." International journal of computer vision 115, no. 3 (2015): 211-252.

[8] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770-778. 2016.

[9] Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 1-9. 2015.

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

[11] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).

[12] Ian Goodfellow , Yoshua Bengio and Aaron Courville, Deep Learning. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[13] Wang, Shiqiang, Tiffany Tuor, Theodoros Salonidis, Kin K. Leung, Christian Makaya, Ting He, and Kevin Chan. "Adaptive federated learning in resource constrained edge computing systems." IEEE Journal on Selected Areas in Communications (2019).

[14] Han, Song, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. "EIE: efficient inference engine on compressed deep neural network." In 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), pp. 243-254. IEEE, 2016.

[15] Alwani, Manoj, Han Chen, Michael Ferdman, and Peter Milder. "Fused-layer CNN accelerators." In The 49th Annual IEEE/ACM International Symposium on Microarchitecture, p. 22. IEEE Press, 2016.

[16] Shin, Sungho, Yoonho Boo, and Wonyong Sung. "Fixed-point optimization of deep neural networks with adaptive step size retraining." In 2017 IEEE International conference on acoustics, speech and signal processing (ICASSP), pp. 1203-1207. IEEE, 2017.

[17] Dean, Jeffrey, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior et al. "Large scale distributed deep networks." In Advances in neural information processing systems, pp. 1223-1231. 2012.

[18] Wu, Yonghui, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun et al. "Google's neural machine translation system: Bridging the gap between human and machine translation." arXiv preprint arXiv:1609.08144 (2016).

[19] Gao, Mingyu, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. "Tetris: Scalable and efficient neural network acceleration with 3d memory." In ACM SIGARCH Computer Architecture News, vol. 45, no. 1, pp. 751-764. ACM, 2017.

[20] Jia, Zhihao, Matei Zaharia, and Alex Aiken. "Beyond data and model parallelism for deep neural networks." arXiv preprint arXiv:1807.05358 (2018).

[21] http://yann.lecun.com/exdb/mnist/

[22] Krizhevsky, Alex, and Geoffrey Hinton. Learning multiple layers of features from tiny images. Vol. 1, no. 4. Technical report, University of Toronto, 2009.

[23] LeCun, Yann, Lon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86, no. 11 (1998): 2278-2324.