Cloud and Machine Learning(CSCI-GA.3033-025)

Performance Analysis of various Deep Learning Models on High Performance Computing Systems

Pramit Mallick (pm2758)
Pranshu Sehgal (ps3588@nyu.edu)

Abstract

Machine learning and deep learning models are increasingly being used in common applications - from Apple's face id to Google's autocomplete feature. The widespread use and research in this field are giving birth to more complex models and architectures. Even in the prototyping phases, the complexities of the models have become very important (as recently demonstrated by the BERT model [14]). This it has become imperative to study the way various algorithms work in the hope of optimizing and accelerating the performance of the systems that are used to train these large models and for deployment as well. While various studies [7,8,10,16] in the past have been conducted to study the behavior of the various networks in various system, we do a comparative study of some of the deep networks for the imagenet classification task [15] using GPU traces, memory allocation and CPU execution times.

Experiment design:

For a comprehensive study of the performance of deep learning models on the NYU high-performance cluster "Prince", we take up the classification task using the imagenet dataset. While the suggested way to run the experiments was to request interactive jobs on Prince, we realized that this babysitting process would take up a lot of time. Instead, we ran batch jobs with the same configurations as specified by the "Chung" reservation. This, however, leads to certain complications due to the memory and the time limit specified for the dedicated nodes and GPUs. One of the first challenges we noticed was that the 1-hour time limit on the jobs was not sufficient for an epoch of training to finish. However, we reasoned that to analyze

the performance of the training, we do not need the model to go through an entire epoch. We just need sample points of the forward propagation, loss calculation, and backward propagation during the training. Thus we can use the nvprof's "timeout" feature where the profiler stops the executable after the prescribed time and only provides the profile and the traces of the execution so far. For our experiments, we set the timeout to 10mins (the training is guaranteed to start by this time).

To get a complete picture of how networks behave, given the devices, it is necessary for us to iterate over various models and their hyperparameters. For out study, we conduct our experiments on the following models -

1. Alexnet [11]
2. Resnet18 [12]
3. VGG16 [13]

These models are trained on both the k80 and p40 GPUs. For the hyperparameters, we iterate across increasing batch sizes and observe the difference in the behaviors of the models and the systems. We choose the values of [4,8,16,32,64] as the batch sizes. Iterating over the learning rate would not have any difference to the system performances as the learning rate is just a scalar operation to the weights of the network and changing it would not change the behavior of the model.

In the following sections, we highlight our observations using both the visual profiler and the log outputs from the nvprof tool.

A little bit about the nvprof tool - The NVIDIA profiling tool enables us to understand and optimize the performance of our CUDA, OpenACC or OpenMP applications. The Visual Profiler is a graphical profiling tool that displays a timeline of the application's CPU and GPU activity, and that includes an automated analysis engine to identify optimization opportunities. The nvprof profiling tool enables you to collect and view profiling data from the command-line.

Here we first analyze one of our experiments with the Visual Profiler to see the various metrics one can measure using it.

Visual Profiler:

Alexnet_64_0.01_p40_full_29852

This profile contains the summary output of the Alexnet training using a batch size of 64, a learning rate of 0.01 on the p40 GPU.

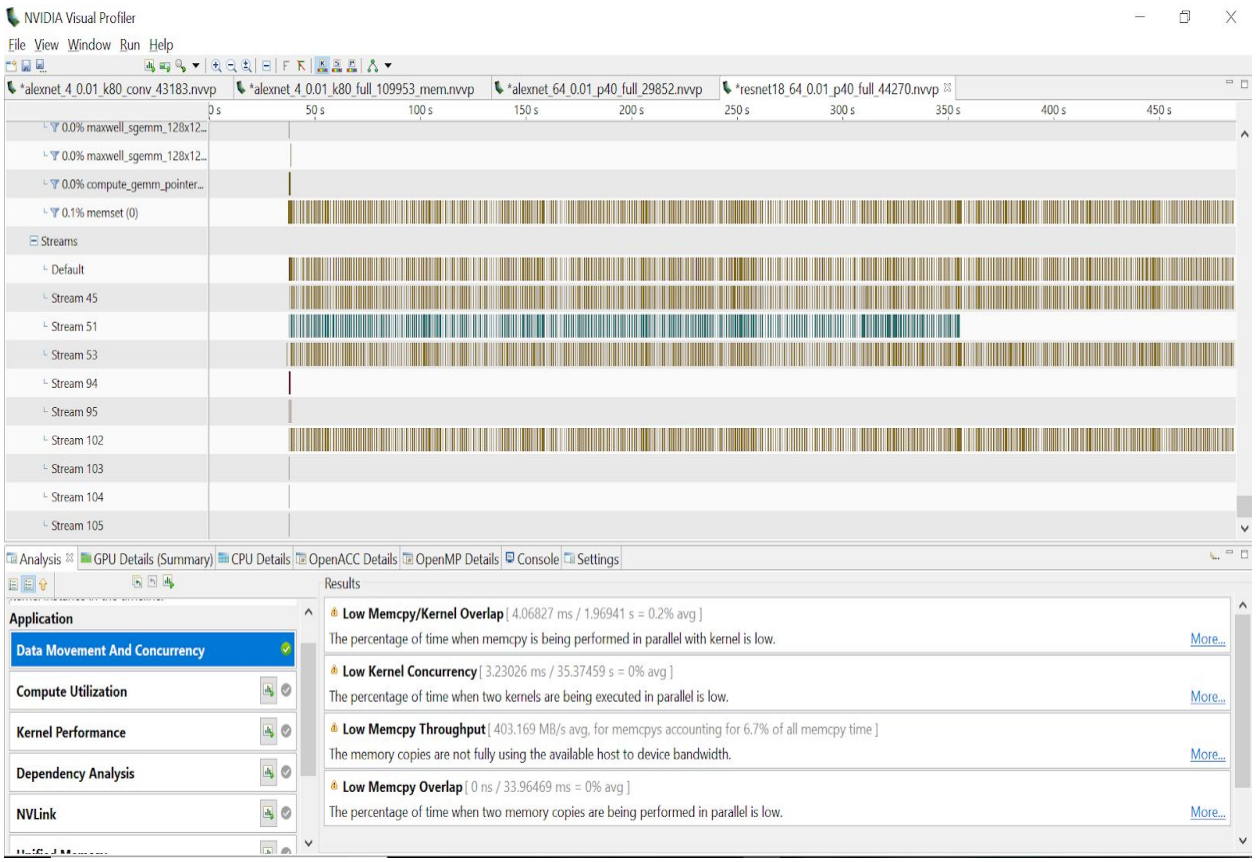
We observe the following:

1) Low Kernel Concurrency: 675.774 us / 12.37502s = 0% avg
   The %age of time when two kernels are being executed in parallel is low.

2) Inefficient Memcpy Size: Small memory copies do not enable the GPU to fully use the host to device bandwidth.

3) Low Memory Copy Throughput: 17.4% of memory copy time is used up. The memory copies are not fully using the available host to device bandwidth.

4) Low Memory Copy Overlap: The percentage of time when two memory copies are being performed in parallel is low. In Overlapping computation and data transfers, the memory copy and kernel execution occur sequentially. On devices that are capable of concurrent copy and compute, it is possible to overlap kernel execution on the device with data transfers between the host and the device. Whether a device has this capability is indicated by the asyncEngineCount field of the cudaDeviceProp structure (or listed in the output of the deviceQuery CUDA Sample).



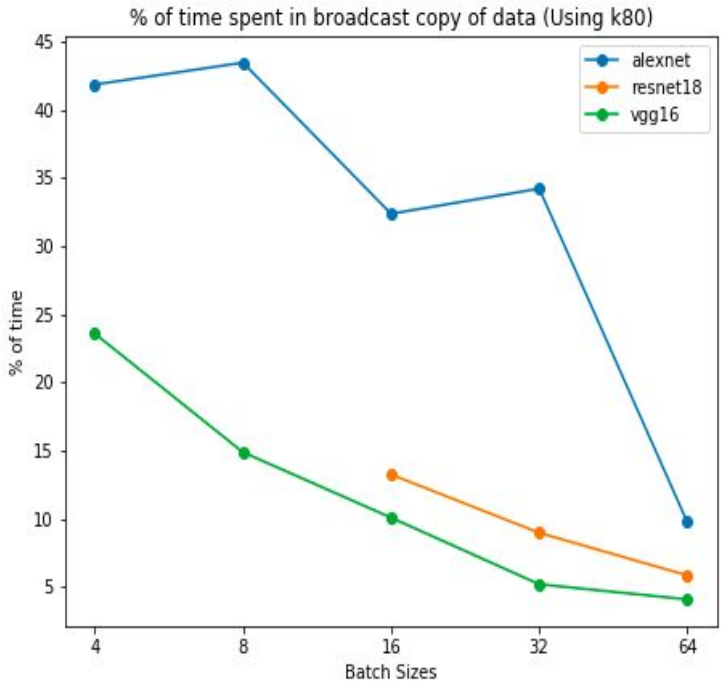
Using Log Files:

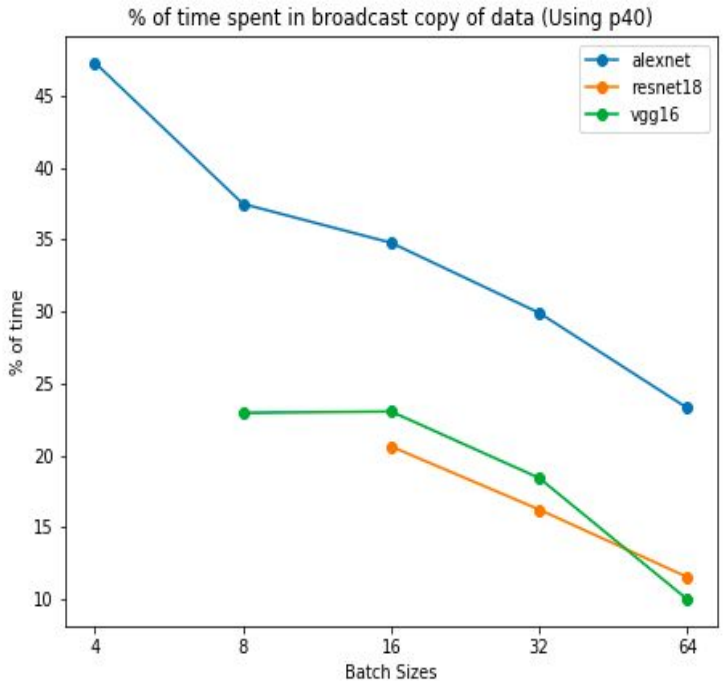
While the Visual Profiler provides us with lots of valuable inferences, each output file is over several GBs making it infeasible to extensive analysis using them. Thus we stick to using the log outputs for deeper analysis.

Studying the memory copy characteristic of models across various batch sizes -

While studying the output logs of the profiler, we identified some of the most time-consuming GPU kernel calls. Among them, one of the kernels was responsible for copying data to the device, namely - the “ncclBroadcastKernel_copy_i8”[1] kernel. This kernel call is of a “broadcast” type. Typically, in distributed and multiprocessing settings, a broadcast is one of the standard collective communication techniques. During a broadcast, one process sends the same data to all processes in a communicator. One of the main uses of broadcasting is to send

out input to a parallel program or send out configuration parameters to all processes[2]. In our setting, this would be broadcasting the data or the model parameters to the global and shared memory of the GPU.
Thus we train the different models and vary the batch sizes and observe the change in the percentage of the total time consumed.



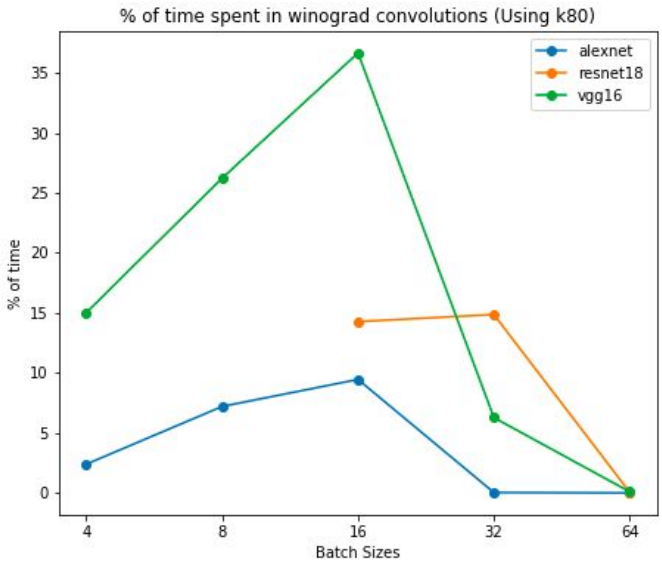


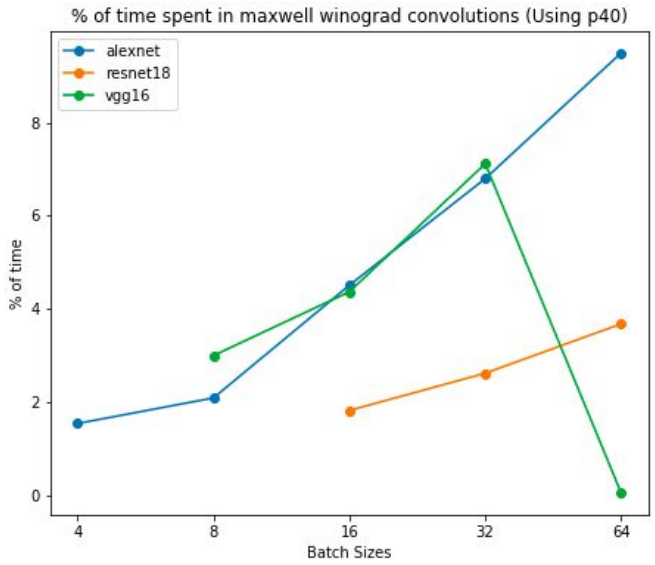
We can see here a definite trend as we increase the batch size regardless of the model used. With smaller batch sizes, the program is occupied most of the time in the broadcast copy call (almost half of the time - about 45%). This makes sense as with smaller batch sizes (hence lesser data per batch), the computation (model eval + update, etc) must take lesser time, thus the program would primarily be occupied in data transfers and copying, making the execution memory bound in nature. This hypothesis is proved as we increase the batch size and we see a trend of decreasing percentage of the time. With a batch size of 64, we see that the time a program spends in broadcasting is considerably lesser, suggesting that training with large batch sizes would be more efficient. However, we must take into consideration the limited memory capacity of the GPU while increasing the batch size as well.

We see no significant difference across the two GPUs, however, we can see that there is some difference in the way alexnet uses broadcasts versus resnet and vgg.

Studying Convolution operations of models across various batch sizes -

Since we are using the imagenet data set, we know that all the models used here will use convolutional layers and filters in their architecture. Thus it is interesting to see how the different models use different convolution operations across different batch sizes.





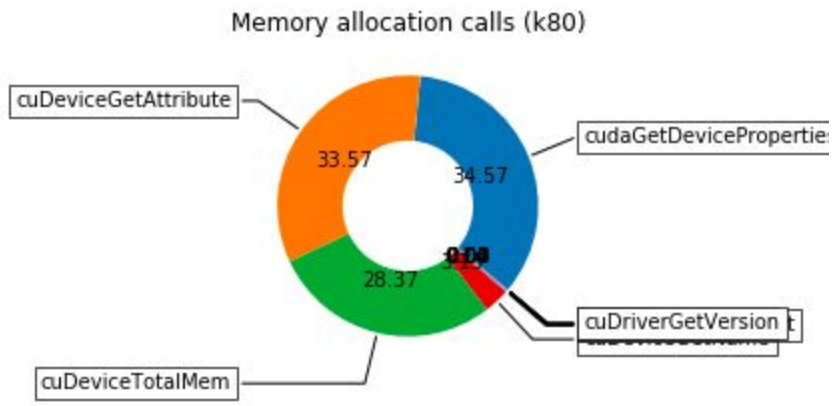
Here we plot the percentage of time spent in the “winograd” convolution operation. The Winograd algorithm is actually a fast matrix multiplication algorithm[3]. In the context of deep learning and CNNs, this algorithm has been adopted and modified to make convolution operations fast[4]. Winograd’s filtering algorithms compute with minimal complexity, convolution over small tiles, which makes them fast with small filters and small batch sizes. [4] shows us how for large filters, conventional FFT (Fourier transformed) filters are very fast. But it argues that most of the state of the art CNNs use small (3x3) filters. Thus motivated by the need for fast convolution filtering operations, the authors suggest the winograd algorithm.
We can see here that Pytorch in its backend, uses CUDA’s implementation of the winograd algorithm. However, we also noticed that the algorithm used was

different in the k80 and p40 GPUs. In the k80 GPU, we observed traces to the "winograd3x3Kernel" kernel call, whereas in the p40 GPU, we observed a different kernel - "maxwell_scudnn_winograd_128x128_ldg1_ldg4_tile148n_nt". The two different kernels could be because of the different architectures of the k80 and the p40 GPUs. The TeslaK80 has the Kepler architecture and can support compute capabilities of up to 3.7, whereas the TeslaP40 has the Pascal architecture and can support compute capabilities of up to 6.1[5]. These differences would mean they call different kernels (optimized to their own hardware and drivers) which is why it wouldn't be fair to compare the two GPUs on this front.

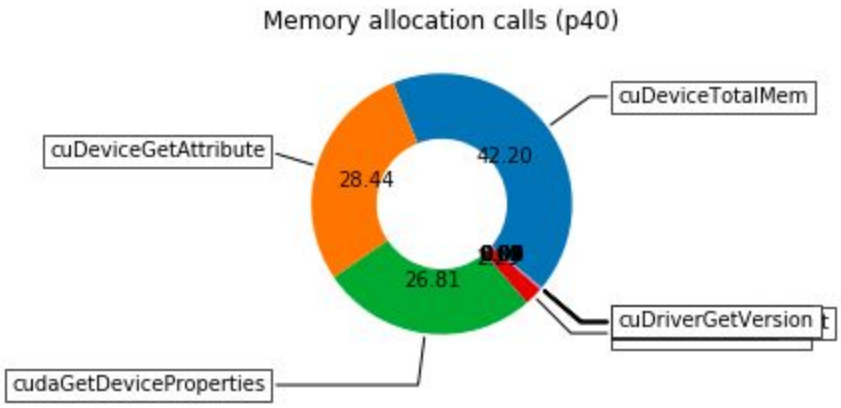
However, when we see the across models in k80, we observe that in general the time spent in convolution operations first increases and then decreases with increasing batch sizes. According to [4], we should expect a high value in lower batch sizes as well, but empirical evidence doesn't show it. A possible explanation for this could lie in the very metric we are measuring. We are recording the time spent in the operation as a percentage of the entire program execution. In lower batch sizes, as observed in the previous section, we see that the execution is memory bound. Thus the percentage of time in the compute sections of the code are less. Thus making the values low here. The main thing to note here is that as we increase the batch to considerable size (say, 16) then the time spent in convolution operations is more, as is expected. However, with a very large batch size (of 64), we see that the time spent in the winograd convolution reduces. This confirms the results of [4] that claim fast computation of convolution operations for small kernels and small batch sizes. While this is apparent in the k80 experiments, the same cannot be said of the experiments in the p40 GPU. We see a similar trend using alexnet as the model, but the evidence suggests that the convolution operations in resnet and vgg consume more time (and possibly more efficient) as the batch size increases.

Memory allocation -

Apart from profiling the entire program execution, we also experiment and record the memory allocations of the various models across different batch sizes. We were hoping to find some differences in the memory use patterns that would suggest the different ways in which the algorithms worked, but we found no significant variance (neither across the models nor across the varying batch sizes). The only difference we found was across the GPUs. Below we plot a donut chart highlighting the percentages of time spent in each of the memory related API calls.



Some of the API calls were - "cuDeviceGetAttribute", "cuDeviceTotalMem", "cudaSetDevice", etc.
We see that while there is variability across the GPUs in terms of the amount of time spent in each of the calls, which can be attributed again to the difference in the compute capabilities and the architectures of the devices, the most common calls were "cudaGetDeviceProperties", "cuDeviceGetAttribute" and "cuDeviceTotalMem".

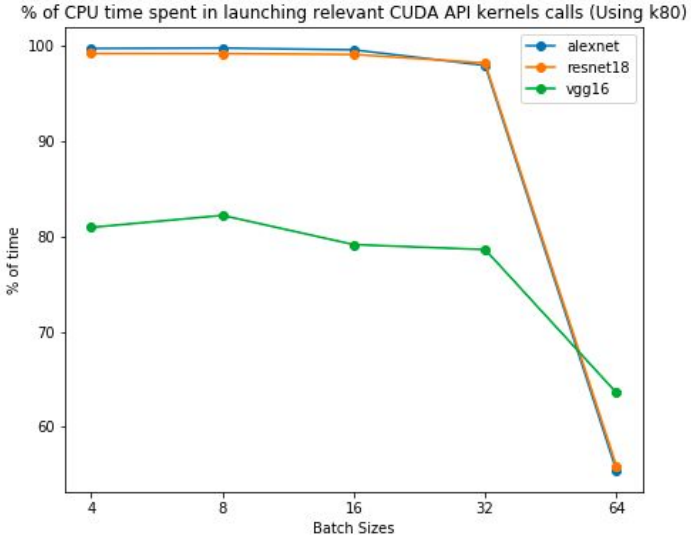


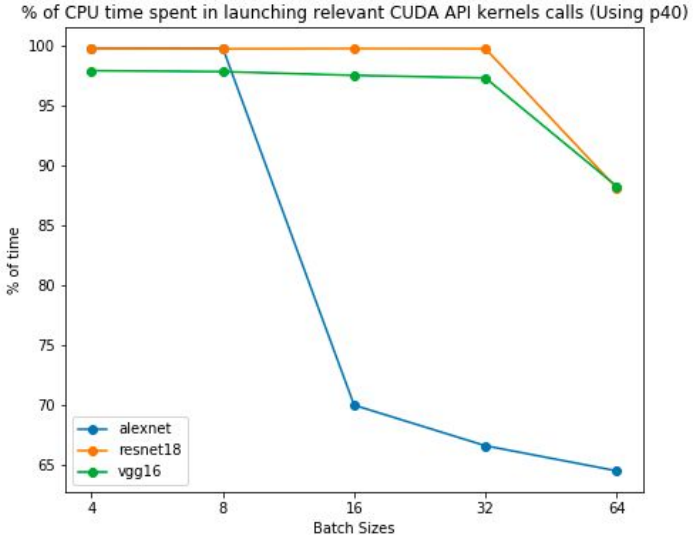
Studying backpropagation characteristic of models across various batch sizes -

For our experiments, we also tried to study the performance of the models on subsections of the code. In general, any machine learning experiment involves a couple of general stages - data preprocessing, training and inference. Most of the data preprocessing are usually done on the CPU. It can also be performed on the GPUs, but most of the times the preprocessing is done just once and used in later iterations. Most of the heavy work of any ML experiment is in training the model. Inference (or simple forward propagation is done as part of the training itself, to calculate the loss).
Here we tried to isolate subsections of the code and put them in a cudaprofile block. We use python wrappers[6] for the function to help us isolate the subsections. We mainly profile the backpropagation part of the code which can be done by isolating the loss calculation and the weights update. Following is the change to the code -

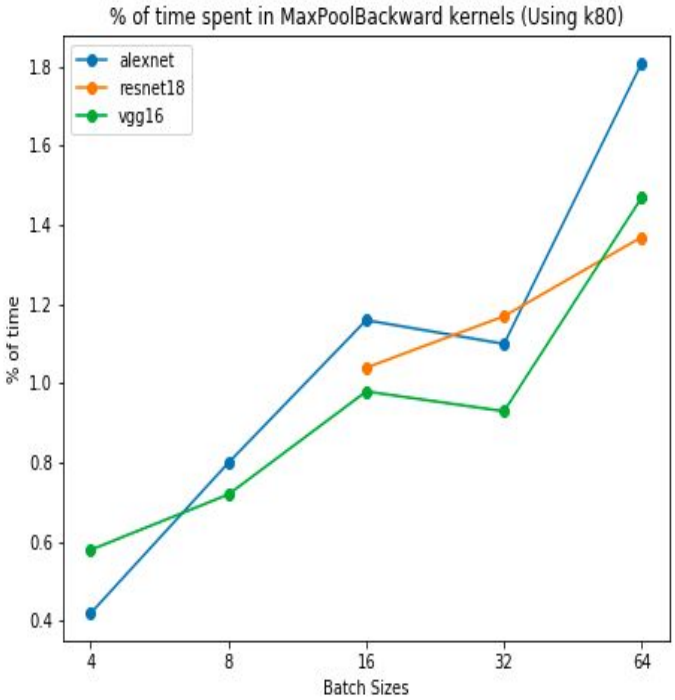
```
cudaprofile.start()
optimizer.zero_grad()
loss.backward()
optimizer.step()
cudaprofile.stop()
```

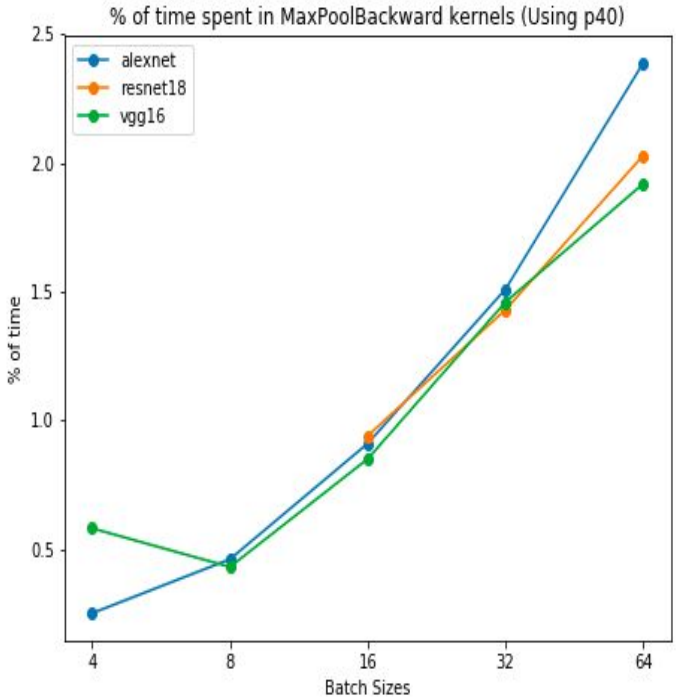
Here we first notice the time spent in launching the various API kernels, to be precise, the "cudaLaunchKernel" API call. We see that while most of the time, the system is busy making various kernel calls (almost a 100% of the time) which suggests that quite a lot of the computation and activity (data transfer, etc) in these two models is performed on the GPU devices. The amount of time is only reduced in very large batch sizes (64), and in these cases, the log files show that the remaining amount of time the other main API called is "cudaEventDestroy". This is the case with both k80 and p40.
However, for vgg execution in k80 in particular, we see that the time spent on launching kernels is lesser and the time cleaning up is more, which suggests that perhaps the implementation of the program (and thus subsequently the calls) aren't as optimized for the vgg model in that particular hardware.





We also plot a kernel call which we definitely knew was involved in the backpropagation process. Since there were so many kernel calls, we were unable to pinpoint the main back-prop calls. However, we know that max pooling is definitely a part of the process. So here we plot the calls to the “MaxPoolBackward” kernel.





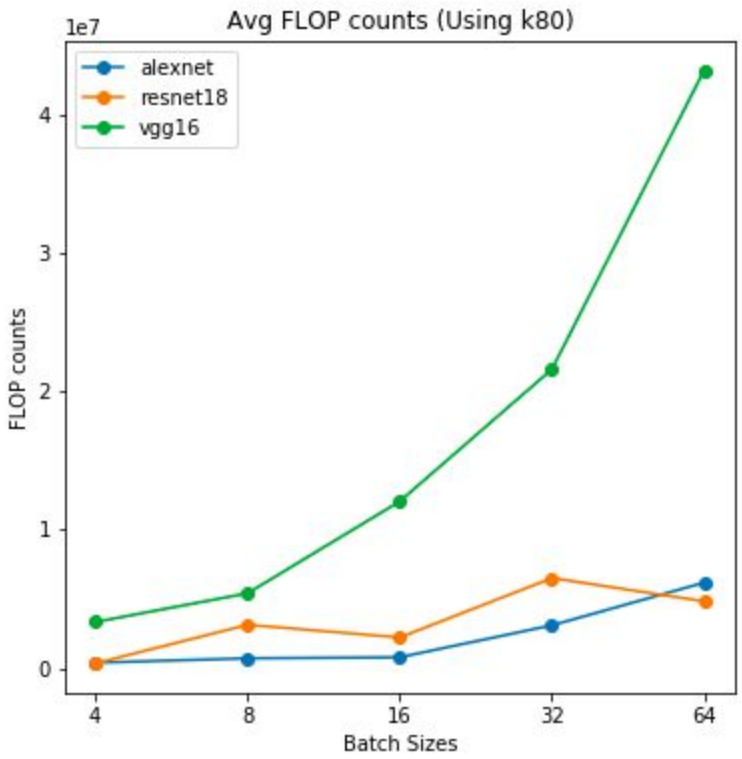
Max pooling is a sample-based discretization process. The objective is to down-sample an input representation (image, hidden-layer output matrix, etc.), reducing its dimensionality and allowing for assumptions to be made about features contained in the sub-regions binned. This is done in part to help over-fitting by providing an abstracted form of the representation. As well, it reduces the computational cost by reducing the number of parameters to learn and provides basic translation invariance to the internal representation.
In backpropagation, the gradient is only propagated to the maximum element in the filter region with a factor of 1, while the rest are given a gradient of 0. The "MaxPoolBackward" kernel is responsible for evaluating the gradient here. We can see according to the plots above that there is a general trend of the increasing amount of time spent on the kernel call as the batch size increases. This is to be expected as the larger the batch size, the more time the kernel would take to compute the gradient.

Studying the FLOP counts of different algorithms at different batch sizes:

FLOP (floating-point operations) stands for one addition, subtraction, multiplication, or division of two floating-point numbers. A Flop serves as a basic unit of computation.

It is normally used to estimate the complexity of an algorithm: We can express number of flops as a (polynomial) function of the problem dimensions, and simplify by keeping only the leading terms.
Although it is not a very accurate predictor of computation time, it is a rough predictor of complexity of an algorithm. In practice, the division is more expensive to compute. However, we approximate it to one flop to simplify the overall calculation. In this way, the number of flops indicates the cost of performing a sequence of operations. Note that, the flop count is just a rough measure of how expensive an algorithm can be. Many more aspects need to be taken into account to accurately estimate practical runtime. It is still quite useful in predicting the algorithmic cost.

The graph below tell us about the algorithmic cost of various algorithms differing in batch sizes on k80 GPU.



When the batch size is small the effect of backpropagation on the number of flop counts is more, so we see a higher average count, but when the batch size becomes very large, the effect of backpropagation starts to get dominated by the higher batch size and we see a higher count in the flops.

According to the graph, vgg16 seems to be a more complex algorithm than resnet18 and alexnet.

Conclusion-

In this report we outline some of the characteristics and behavior of the various models as we increase the batch size. In general, we can say that with lower batch sizes, the program execution is memory bound as the system spends a lot of time in data processing and transfers. We also note that the lower level function calls are different not only across devices (due to different compute capabilities and

architectures) but also across the batch sizes. This tells us that in its backend, Python and its underlying C tensor libraries (in "aten") perform various optimizations by choosing specific algorithms that would help in faster execution of the models.
Our code, shell scripts, and the output log files can be found here - https://github.com/pramitmallick/examples/tree/master/imagenet
https://github.com/pranshu777/imagenet

Some of the larger output files (*.nvvp) files can be found here - https://drive.google.com/open?id=14_g04r3PZeqYL7eiTLrdfLv7DZizET4R

References-

1. SDK Documentation - https://docs.nvidia.com/deeplearning/sdk/nccl-archived/nccl_235/nccl-developer-guide/index.html
2. MPI tutorial - http://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/
3. Winograd Algorithm - https://en.wikipedia.org/wiki/Coppersmith%E2%80%93Winograd_algorithm
4. Fast Algorithms for Convolutional Neural Networks - https://arxiv.org/pdf/1509.09308.pdf
5. CUDA and device compute capabilities - https://en.wikipedia.org/wiki/CUDA
6. CUDA Profiling wrapper for Python - https://github.com/bshillingford/python-cuda-profile
7. DeepProf: Performance Analysis for Deep Learning Applications via Mining GPU Execution Patterns - https://arxiv.org/pdf/1707.03750.pdf
8. https://upcommons.upc.edu/bitstream/handle/2117/106267/127332.pdf
9. CUDA Toolkit Documentation - https://docs.nvidia.com/cuda/profiler-users-guide/index.html
10. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions - https://arxiv.org/pdf/1802.04730.pdf
11. ImageNet Classification with Deep Convolutional Neural Networks - http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
12. Deep Residual Learning for Image Recognition - https://arxiv.org/pdf/1512.03385.pdf
13. Very Deep Convolutional Networks for Large-Scale Image Recognition - https://arxiv.org/pdf/1409.1556.pdf

14. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding - https://arxiv.org/pdf/1810.04805.pdf
15. ImageNet: A Large-Scale Hierarchical Image Database - https://www.researchgate.net/profile/Li_Jia_Li/publication/221361415_ImageNet_a_Large-Scale_Hierarchical_Image_Database/links/00b495388120dbc339000000/ImageNet-a-Large-Scale-Hierarchical-Image-Database.pdf
16. Analyzing Machine Learning Workloads Using a Detailed GPU Simulator - https://arxiv.org/pdf/1811.08933.pdf