Streaming and Scheduling DNN Inference Operations

Bedant Agarwal
bedantagarwal9@iitkgp.ac.in
Indian Institute of Technology
Kharagpur
Kharagpur, WB, India

Divyang Mittal
divyangmittal44@gmail.com
Indian Institute of Technology
Kharagpur
Kharagpur, WB, India

Jyoti Agarwal
jyotiagrawal851999@gmail.com
Indian Institute of Technology
Kharagpur
Kharagpur, WB, India

Jyotisman Das
jyotisman@iitkgp.ac.in
Indian Institute of Technology
Kharagpur
Kharagpur, WB, India

Prabhpreet Singh Sodhi
pprabh2007@hotmail.co.uk
Indian Institute of Technology
Kharagpur
Kharagpur, WB, India

Prakhar Sharma
prakhar6sharma@gmail.com
Indian Institute of Technology
Kharagpur
Kharagpur, WB, India

Prashant Ramnani
ramnani.prashant@gmail.com
Indian Institute of Technology
Kharagpur
Kharagpur, WB, India

Robin Babu Padamadan
robinb2009@gmail.com
Indian Institute of Technology
Kharagpur
Kharagpur, WB, India

ABSTRACT

Advancement in Deep Neural Network models is pressing upon the requirement of efficient use of GPU resources. In this work we have leveraged the fact that in CNN models, GPU utilization decreases drastically in the final layers. Thereby, allowing the possibility of running multiple GPU kernels simultaneously and concurrently. We have implemented S^{3}DNN, a paper which employs supervised scheduling method and data fusion to improve the efficiency of kernels and improves the overall throughput of the GPU.

CCS CONCEPTS

• **Parallel Programming** → **CUDA**; • **Neural Networks** → *DNN Pipelines.*

KEYWORDS

Neural Networks, Parallel Programming, CUDA, CUDNN

ACM Reference Format:
Bedant Agarwal, Divyang Mittal, Jyoti Agarwal, Jyotisman Das, Prabhpreet Singh Sodhi, Prakhar Sharma, Prashant Ramnani, and Robin Babu Padamadan. 2018. Streaming and Scheduling DNN Inference Operations. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 MOTIVATION

Graphic processing units (GPU) are nowadays being widely used to accelerate the computation intensive programs that are data parallel, in the form of GPGPU. One such example of these kind of programs is Deep Neural Network(DNN). DNN involve large amount of computation due to the vast amount of data involved and their complex and large number of layers. Much of the recent success of AI is backed by heavy use of DNN in backend to solve tasks involving natural language processing, computer vision, recommendation systems, etc. The importance of these tasks has lead to the development of large number of optimization techniques to make these neural networks efficient and accurate. But there has been little development with respect to scheduling of workloads on GPU and improving the system level efficiency. Most of the DNN pipelines employed to solve Computer Vision problems involve use of CNN based architectures where the initial layers involve convolution operation over a large image and the final layers use feed forward networks. The number of parameters and operations performed in these initial layers is significantly greater than those in the last layers. As a result, it has been observed that many of these pipelines follow a trend where they show gradual decrease in kernel utilization in the final layers. This provides an opportunity to run multiple kernels simultaneously.

Furthermore, the development of CNN models such as AlexNet and YOLO have resulted in extraordinary performance on the task of object recognition. Thus necessitating their use in an autonomous car. From the vehicle perspective, we need to ensure minimal GPU response time to allow comfortable functioning.

All this discussion has called attention upon development of a scheduling system for GPU kernels which should be deadline sensitive and should be able to pre-empt jobs if required. Major part of this work is based upon the implementation of S3DNN [3] which aims at achieving these goals and analysis of the results against baseline scheduling policies and system without any scheduling.

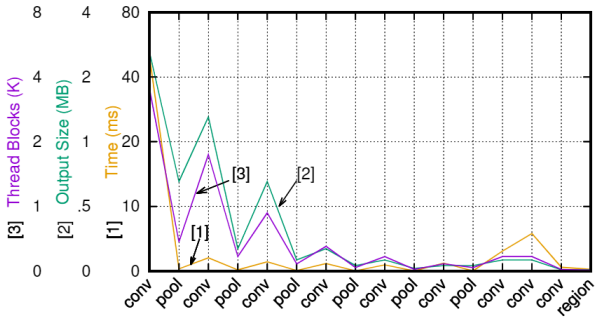

Fig. 1: Resource usage pattern of DNN workloads.

2 PROBLEM DEFINITION

The objective of this project is to develop scheduling algorithms that simultaneously maximize concurrency-induced benefits and real-time performance of multiple such DNN workloads used for autonomous driving. Typically, the final layers of a DNN instance exhibit a gradually decreased GPU resource utilization. This observation can be leveraged to concurrently schedule multiple kernel operations in parallel using CUDA streams. To achieve this we have primarily implemented the following paper - "s3DNN[3] Supervised Streaming and Scheduling for GPU Accelerated Real Time DNN Workloads".

The work has been divided into five parts. First, implementation of the forward functions for a convolution layer with an option of pooling and a fully connected layer. Second, creation of kernels so that these functions can be called in a GPU environment. These functions use CUDA streams to parallelize the execution. Third, implementation of a baseline algorithm for kernel scheduling. Fourth, the implementation of the supervised scheduling algorithm from the s3DNN paper for effective scheduling. Finally, implementation of an AlexNet[1] CNN pipeline. We also analysed results of this pipeline.

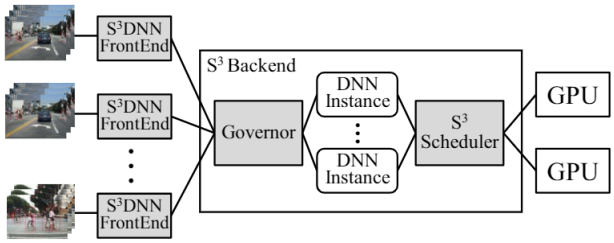

Fig. 2: Design overview of S^3DNN.

3 METHODOLOGY

The first thing that we have implemented is forward pass for a convolution layer. This was done using the CUDNN and CUBLAS library provided by CUDA. These forward passes were then converted into callable kernel modules. These modules copy the memory from host to device, execute the code at device and finally return the results back to host. These kernels were employed to create a custom CNN pipeline filled with convolution layers which was used for profiling.

In order to use a CNN model, a pre-trained weight file is needed, since we have implemented a custom CNN pipeline random weights were used for this purpose. Training an accurate weight file is an offline procedure that usually takes several days or weeks, done by "learning" features from large scale image datasets. In this report, we are not concerned with improving the training process, rather, we focus on efficient execution of DNNs for real-time object detection.

As a baseline model, we first implemented a custom CNN forward pass without any optimisation. This model takes one frame as input and produces the output for that single frame. We improved this by implementing DNN forward passes using batched kernel operations.

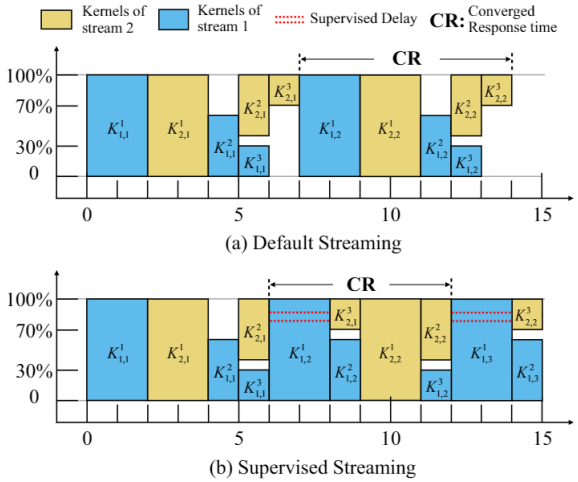

Fig. 3: Concurrency under CUDA stream without supervised streaming (inset (a)) and with supervised streaming (inset (b)).

We then implemented a baseline scheduling policy using CUDA streams to concurrently schedule multiple DNN forward pass pipelines. To enable concurrency, each DNN instance is mapped to a CUDA stream. Total number of cuda streams are fixed a priori. Whenever a new image arrives, a new thread is created to map it to one cuda stream. Frames arriving in a continuous series, are mapped to a particular stream in round robin fashion and inside a stream they are processed sequentially. But different streams may be potentially overlapped which is also the reason for performance boost.

Finally, we implemented the scheduling policy mentioned in the paper - "S3DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads". The policy/algorithm is described in detail in the figure shown below.

3.1 Scheduling Algorithm explained

GPU hardware is often constrained by either the hardware architecture or register/shared memory size. For example, GPUs with compute capability 2.x can support up to eight blocks per SM, if register/shared memory size is not the bottleneck.

In the Algortihm we have defined a few variables **tbRatio** - metric to measure the proportion of the demanded thread blocks by a

Algorithm Supervised streaming and scheduling algorithm

```
Require: Q   ▷ Queue of kernels to be scheduled
Require: G   ▷ Group of kernels that can execute concurrently
Require: C   ▷ Critical queue that will be submitted to GPU
 1: function ENQUEUE(Q[],k)
 2:     for q in Q do
 3:         UpdateSlacks(Q)
 4:         if Slack(k) > Slack(q) then
 5:             Q.InsertBefore(k, q)
 6: function DEQUEUE(Q[])
 7:     UpdateSlacks(Q)
 8:     if G ≠ ∅ then
 9:         go to assign
10:     h ← HeadOf(Q)
11:     G.insert(h)
12:     if tbRatio(h) < 1 then
13:         for t in Q-h do
14:             if tbRatio(t) < 1 then G.insert(t)
15:             if tbRatio(G) > 1 then break
16:     else go to assign
17:     if tbRatio(G) < 1 then
18:         h' ← HeadOf(Q)
19:         p ← LookAhead(h')
20:         if tbRatio(p) < 1 then
21:             G.reserve(p); C.insert(h')
22:             go to submit
23:         else go to assign
24:     C ← G; G ← ∅
25:     Submit(C); Q.Remove(C)
```

kernel to the total number of thread blocks provided by the GPU hardware. **Q** : queue of kernels to be scheduled. **G** : any subset of kernels that can execute concurrently. **C** : the subset of kernels with the highest priority in the scheduling queue. There are two major functions defined in Algorithm 2: Enqueue and Dequeue. The algorithm needs three input data structures, including a scheduling queue (denoted by Q), any subset of kernels that can execute concurrently, and the subset of kernels with the highest priority in the scheduling queue . Function Enqueue is invoked at kernel arrivals and completions, which sorts the released kernels in Q using LSF (Lines 2-5). It first updates the remaining slacks for each kernel (line 3), and then inserts each newly arrived kernel k into Q according to LSF (Lines 4-5). Dequeue is not invoked immediately after a kernel's (e.g. k0's) completion, but S3DNN delays this invocation a little bit until k0's successor kernel is pushed into the queue. It first updates slacks for kernels in the scheduling queue (Line 7), and then checks if G is empty (Line 8). If G is not empty, then the scheduler directly submits kernels within G for execution (Line 9). Next, the kernel h at the head of the scheduling queue is pushed into G (Lines 10-11). Then the scheduler checks whether h can fully occupy the GPU by calculating its tbRatio. If tbRatio of h is smaller than 1 (Line 12), indicating it may be concurrently executed with other kernels, then the scheduler will seek to put more kernels in Q whose tbRatio is also less than 1 (Lines 13- 15); else h is directly submitted to GPU device for execution (Line 16). After all potential small kernels in Q are merged into G, the scheduler checks whether the tbRatio of G is still less than 1 (Line 17). If so, the scheduler looks ahead the successor kernels of the ones residing in Q in the order of priorities, in order to identify any such kernels that have not been released but with tbRatio < 1 (Lines 18-22). Finally, kernels placed in G will be sent to C, which will be further submitted to GPU for execution (Lines 24-25).

From the implementation perspective, A custom class object is created for each pipeline to be executed. This object stores the state of this pipeline containing the info such as numLayers, deadline to be met, threadBlock ratio for the current layer, stream of this layer, cublas handler and cudnn handler. A global vector is maintained for the list of process G. Finally a priority queue is maintained for all the submitted kernels which are sorted by the least slack first.

4 OPTIMIZATIONS PERFORMED

4.1 CUDA streams

All CUDA calls are either synchronous or asynchronous w.r.t the host. Kernel Launches are asynchronous and automatically overlap with host. A stream is a queue of device work. The host places work in the queue and continues on immediately. Device schedules work from streams when resources are free CUDA operations are placed within a stream. In this way, cuda streams enable concurrent execution of different kernels thereby ensuring efficient utilization of GPU device.

4.2 Memory operations

There are three types of memory

- Device Memory - Allocated using cudaMalloc
- Pageable Host Memory - Default allocation (e.g. malloc etc). This Can be paged in and out by the OS
- Pinned (Page-Locked) Host Memory - Allocated using special allocators .This cannot be paged out by the OS

The cudaMemcpy() places transfer into the default stream thus making the process synchronous i.e must complete prior to returning. In contrast to that we have used cudaMemcpyAsync(, &stream). This places transfers into the given stream and returns immediately and thus enables concurrency. To ensure all the memory commands are concurrent, We need to follow all these conditions -

- The memory copy transfer is placed in a different non-default stream.
- Host uses only pinned memory for transferable memories.
- The asynchronous API cudaMemcpyAsync is used
- There should be no other memory copy occurring in the same direction at the same time.

4.3 Batched Operations

We have implemented batched operations using functions like *cublasSgemmBatched* available in cuBLAS and cuDNN libraries. In the batched implementation, a batch of frames are passed to the model and output for all such frames are produced at once. In particular, the optimized batched matrix multiplications can substantially outperform the non-batched version and reach around 84.8 % of the performance upper bound.

5 RESULTS ABLATION STUDY

We performed all our experiments using Google Colab. The virtual machine assigned to us consisted of Nvidia tesla T4, which is based

Batch-sizes	2	4	8	16
Cuda Streaming				
1 streams	1150(ms)	1300(ms)	1570(ms)	2135(ms)
4 streams	1157(ms)	1287(ms)	1590(ms)	2188(ms)
Supervised Streaming				
1 streams	88(ms)	172(ms)	325(ms)	631(ms)
4 streams	92(ms)	170(ms)	327(ms)	636(ms)

Table 1: Performance of different methods for various different batch-sizes

on the latest turing micro-architecture and features 2560 cuda cores, 320 tensor cores and 16 GB of DDR6 memory, and an Intel Xeon 2.2 Ghz with 2 cores and support for 2 threads per core and 13 GB of RAM. We compare our implemented S3DNN algorithm to the baseline code with no concurrency. We also compared the same method with concurrency achieved through CUDA-streams. For the DNN we used AlexNet as the architecture.

We evaluate the performance of different methods by comparing their runtime. For each experiment we ran the batch 4 different times and report the total time taken as the performance of the corresponding method. The Results for different experiments are reported in the table below.

One interesting observation we had was that having concurrency using Nvidia streams had no effect on the runtime of both algorithms. We suspect this difference in results from the original paper is caused by the choice of our backend DNN architecture. As AlexNet has more number of parameters than Yolo[2] and consists of majorly Convolution layers only. Since, the speedup caused by the concurrency is the result of scheduling the linear layers together, thus these results become more intuitive.

Even though CUDA doesn't provide functionality for checking the memory usage of program during kernel execution but running commands such as "nvidia-smi" parallely in backend allowed us to have a rough idea about the percentage of GPU used. We observed that using the Supervised streaming and scheduling not only results in drastic reduction in runtime but it also results in more GPU utilisation.

6 CONTRIBUTIONS

Task 1 - Implementation of CNN pipelines - Jyoti Aggarwal, Jyotisman Das
Task 2 - Baseline scheduling policy using CUDA streams - Prashant Ramnani, Robin Babu P., Prabpreet Singh Sodhi
Task 3 - Scheduling policy - Divy ang Mittal, Prakhar Sharma, Bedant Aggarwal

REFERENCES

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) *(NIPS'12)*. Curran Associates Inc., Red Hook, NY, USA, 1097–1105.
[2] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. arXiv:1804.02767 [cs.CV]
[3] H. Zhou, Soroush Bateni, and Cong Liu. 2018. S^3DNN : $SupervisedStreamingandSchedulingforGPU - AcceleratedReal - TimeDNNWorkloads$. *2018 IEEE Real-Time and Embedded Technology and Applications Symposiu* –201.