GRNN: Low-Latency and Scalable RNN Inference on GPUs

By Matthew Lougheed (lougheem), Jiho Yoo (jihoyoo), Zineb Benameur El Youbi (zinebbe)

1. Problem and Motivation

Due to their effectiveness in modeling sequential data, RNNs are gaining more and more attention but because of the complexity of data dependencies and limited parallelism, current inference libraries for RNNs on GPUs produce either high latency or poor scalability, causing inefficient resource utilization.

This paper compares several RNN frameworks/libraries on GPU and explains the main causes of such poor performance amongst these implementations.

Then the paper tries to address these problems using :

- Poor performance of Open-Sourced GPU-Based Inference Engines
- Poor Scalability of Proprietary Libraries

2. Hypothesis

The main idea behind GRNN is to reorganize the data, thread mapping, and performance modeling techniques.

3. Solution Overview

GRNN strategy to reduce latency and attain better performance measure is achieved by implementing:

- an output-oriented tiling technique to minimize synchronization overhead : where every SM is responsible for computing one output tile
- a flexible mapping technique to balance on-chip hardware resource usage
- an accurate comparative performance model to select high-performing configurations from a tremendous configuration space with negligible overhead.

4. Advantages and achievements

GRNN provides low latency, high throughput, and efficient resource utilization. It minimizes global memory accesses and synchronization overhead, as well as balancing on-chip resource usage through novel data reorganization, thread mapping, and performance modeling techniques.
GRNN has been evaluated on extensive benchmarking and real-world applications,
If compared to the state-of-the-art CPU implementation, GRNN achieves the following performance:

- Reduces latency by up to 94%
- Improves throughput by up to 17.5X

The paper shows that GRNN outperforms the state-of-the-art CPU inference library by up to 17.5X and state-of-the-art GPU inference libraries by up to 9X in terms of latency.

Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis

Matthew Lougheed, Jiho Yoo, Zineb Benameur El Youbi

Motivation

As GPUs are able to run DNN workloads at several orders of magnitude faster and cheaper than on CPUs, they are being used in clusters to offer cloud-scale services such as the video analysis service investigated in this paper. It is critical to sustain high utilization on these GPUs to not cancel out the gains in throughput that the hardware accelerators achieve over CPUs. No single stream or application will demand the throughput that GPU/TPU accelerators can provide, so batching is required. Towards this, “sharding inputs via a distributed frontend onto DNNs on backend GPUs” to achieve maximum throughput and maintain satisfactory latency is necessary. For some “live” applications, execution should be within the 100s of milliseconds, while other “batch” applications should be completed within hours. Loading networks into memory on GPUs has a high cost of hundreds of milliseconds, so load balancing should attempt to co-locate models.

DNNs execute more efficiently when their inputs are batched, so batch sizes are another variable that must be determined dynamically to approach maximum throughput. The processing cost of a given input is “squish,” meaning that it is dependent upon the size of the batch within which it is run as larger batches of input allow DNNs to execute more efficiently.

Additionally, it sometimes occurs that groups of DNNs feed into each other, so groups must be scheduled from the cluster level. There may be a latency Service Level Objective (SLO) given for the entire query, so each model must be scheduled with its own latency SLO that meet the query’s requirements.

Nexus architecture

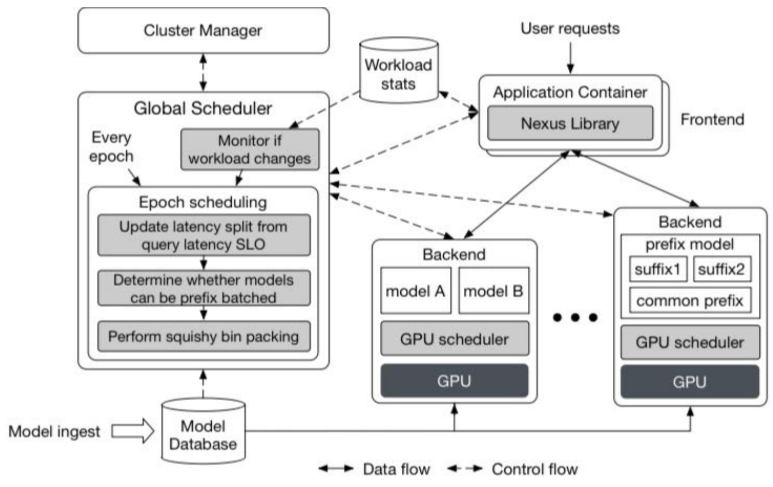

Figure 6: Nexus runtime system overview.

Developers can write and deploy applications (where they can use the Nexus library). When models are uploaded from here, a profiler “measures the execution latency and memory use for different batch sizes." The global scheduler uses statistics from runtime to dynamically add or remove frontend nodes and "invokes the epoch scheduler to decide which models to execute and at what batch size, and which backend to place the models on so as to balance the load and maximize utilization." "Allocation, scheduling, and routing updates happen at the granularity of an epoch," ~30-60s.

Three main, novel techniques are used:

1) Uses a batch-aware scheduler… (6.1)

In a round-robin manner, the scheduler uses the profiled information on a session and assigns it the largest batch size such that the latency SLO can still be met. It is important for the computation time to take less than half the entire SLO. Then, in the remaining computation time of the duty cycle, it can merge the execution of multiple sessions within a single duty cycle as long as the SLOs for each session are not violated.

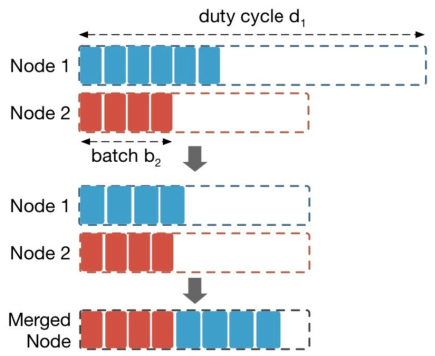

Figure 7: Merge two nodes into one. Use the smaller duty cycle as new duty cycle for both nodes. Update the batch size accordingly and re-estimate the batch latency. If sum of latencies doesn't exceed new duty cycle, the two nodes can be merged.

2) Takes queries of DNN incovations and automatically optimizes the queries to "assign optimal batch sizes to the components of the query so as to maximize overall execution throughput of the query while staying within latency bounds." (6.2) They use dynamic programming, with a maximum granularity, in an algorithmic solution that runs in time quadratically proportional with the quotient of total latency over the minimum granular time step considered.

3) "Nexus … allows batching of parts of networks with different batch sizes. This enables the batched execution of specialized networks." (6.3)

Their analysis, in comparison to TensorFlow Serving and Clipper, achieves 1.8 to 12.7 times more throughput while staying within latency bounds 99% of the time. When latency did go above the given bounds, they found that it was only slightly over and was brought back to baseline within one epoch. It is unsure why 99% was chosen and how this choice would effect overall throughput.

__Discussion:__

- **What is a Transformer model and what is the difference with RNN?**

The **Transformer** is a deep machine learning model used primarily in the field of NLP.

Like RNNs, Transformers are designed to handle ordered sequences of data, such as natural language, for various tasks such as machine translation and text summarization. However, unlike RNNs, Transformers do not require that the sequence be processed in order. So, if the data in question is natural language, the Transformer does not need to process the beginning of a sentence before it processes the end. Due to this feature, the Transformer allows for much more parallelization than RNNs during training.

- **Why are RNN's so underserved?**

The major disadvantage of RNNs are the vanishing gradient and gradient exploding problem. It makes the training of RNN difficult in several ways.

1. It cannot process very long sequences if it uses tanh as its activation function
2. It is very unstable if we use ReLu (rectified linear unit) as its activation function
3. RNNs cannot be stacked into very deep models
4. RNNs are not able to keep track of long-term dependencies

- **LTSM vs GRU**

GRU is related to LSTM as both are utilizing different ways of gating information to prevent vanishing gradient problems. GRU vs LSTM:

  - The GRU controls the flow of information like the LSTM unit, but without having to use a ***memory unit***. It just exposes the full hidden content without any control.
  - GRU is relatively new, the performance is on par with LSTM, but computationally ***more efficient*** (*less complex structure*). So it is being used more and more.

- **Batch size vs optimization’s results?**

In general, larger batch increases latency (bad) and throughput (good). GRNN should be restricted to fewer choices when large batch size leads to too high a latency that will cause SLO miss.

- **Limitations of LSTM**

Long Short-Term Memory or LSTM recurrent neural networks are capable of learning and remembering over long sequences of inputs.

LSTMs work very well if the problem has one output for every input, like time series forecasting or text translation. But LSTMs can be challenging to use when we have very long input sequences and only one or a handful of outputs. This is often called sequence labeling, or sequence classification.

This problem can sometimes be addressed by truncating or summarizing the sequence or by performing random samoling.