Exploring GPU Architectural Optimizations for RNNs

Suchita Pati
Computer Sciences Department
University of Wisconsin-Madison
spati@cs.wisc.edu

Abstract
GPUs have been widely adopted for the training of Deep Learning applications employing Convolutional Neural Networks (CNNs). However, Recurrent Neural Network (RNN) based applications, such as those for Natural Language Processing and Speech and Text Recognition, are becoming increasingly popular but have received significantly less attention. RNNs possess a different set of characteristics than CNNs, such as serialization between time-steps and the need to remember recent events, and are unable to take advantage of many of the common GPU hardware and software optimizations for CNNs. To overcome this, we propose holistic changes to the design of GPUs to optimize them for RNNs.

1 Introduction

Deep Learning has become immensely popular in the last decade, dominating fields such as robotics, virtual reality and autonomous vehicles. Given this, research on hardware for deep neural networks (DNNs) has also become imperative leading to the proposal of several accelerators [7–9, 15, 18, 19, 31, 34, 38]. However, more general-purpose GPUs are also widely used for running DNNs since they are more flexible and better able to adapt to new algorithms than accelerators, which quickly become obsolete due to rapidly evolving algorithms. Therefore, GPUs have been extensively used for both CNN training and inference [28–30].

GPUs have also been used for RNNs, although RNNs are less well studied than CNNs because CNNs are considered more widely used in practice [12]. However, RNNs are also an important application that systems must optimize for [17, 21, 24]. For example, Facebook runs 300 trillion ML inferences a day, most of which are RNNs [16], and Google found that 29% of the Tensor Processing Unit (TPU) workloads are RNNs (in comparison, only 5% are CNNs) [24]. RNNs present a unique challenge for GPUs, since each RNN job often uses kernels with few threads that cannot fully utilize the GPU [13]. Consequently, single RNN jobs cannot benefit from increasing the number of computational units.

Moreover, batching RNN jobs to increase hardware utilization is impractical for RNN inference, since many of the application domains where RNNs are widely used, such as speech recognition, have real-time constraints [17, 24, 43]. For example, data center data shows that an RNN inference job must be completed in 7-10 ms [24, 43]. As a result, RNN performance significantly lags that of CNNs across a wide variety of platforms [35, 36]. We further discuss RNNs and their unique processing model in Section 2.

The fundamental differences between RNNs and CNNs necessitate a different set of GPU optimizations that have not been previously explored. As discussed in Section 5, prior work on RNNs has primarily focused on software solutions such as efficient strategies for mapping the RNNs on to the GPUs [14, 44]. In contrast, we propose to holistically rethink the GPU architecture – including the microarchitecture, memory system, and coherence protocol, to optimize GPUs for RNNs.

2 Background and Challenges

An example RNN is shown in Figure 1. In a single hidden layer recurrent network, given that \( x_t \) is the input at time \( t \) and \( h_{t-1} \) is the recurrent layer activation matrix at time \( t-1 \), then the recurrent layer activations are calculated as follows:

\[
h_t = f(W x_t + U h_{t-1} + b)
\]

where \( W \) is the input-hidden weight matrix and \( U \) is the recurrent weight matrix and \( b \) is a bias term.

RNNs contain loops which allow them to capture and remember information across multiple iterations (or timesteps). These loops can be unrolled, as shown on the right of Figure 1. Each of the unrolled iterations passes information to the next iteration. This sequential dependency between iterations limits parallelism. Although input batching is one of the optimizations that has been adopted to overcome this issue for CNNs, this is not feasible for RNN (inference) due to strict latency requirements [24, 43].

The number of times the loop is unrolled, \( N \), represents the sequence length of the RNN. Unrolling the RNN also allows it to be formulated as a series of matrix operations using optimized matrix multiplication routines like General Matrix Multiply (GEMM). The hidden state vector (the memory) is calculated by looking at the previous hidden state \( h_{t-1} \) and...
the input \( h_t \) at the current step. In theory, RNNs could be unrolled an infinite number of times, which would be equivalent to remembering everything that has happened previously. In practice, the large memory footprint and bandwidth limitations of RNNs limit them to only remembering the most recent events. Despite this practical limitation, RNNs are well suited for domains where prior events persist and influence subsequent ones [2, 5, 6, 20, 40, 42].

3 Proposal

To overcome the challenges mentioned in Section 2, we propose several changes to the microarchitecture, memory system and coherence protocol of GPUs. The key insight underlying these innovations is that RNNs have producer-consumer parallelism that can be exploited at multiple levels to improve performance and energy efficiency.

3.1 Microarchitecture

Currently, RNNs must wait for all activation elements to be computed before the result can be sent to the next timestep. This incurs unnecessary overhead because some of the activations have been computed much earlier than others. By sending a partially computed activation element or activation matrix from the current timestep to carry out computations of the next timestep in parallel, we can exploit more parallelism at the microarchitectural level. To do this, we will map each timestep’s computations to a different Streaming Multiprocessor (SM) via an intelligent scheduling mechanism. Thus, one SM will be a producer of activations for the next timestep, which is consumed by a different SM. Moreover, we will also distribute a timestep’s computations across multiple SMs to improve parallelism, similar to prior work [14]. To ensure this scheduling is only used when desired, we will add a programmer-controlled flag.

3.2 Memory System

Since the activations are produced and consumed by different (sets of) SMs, memory system optimizations are also required.

Reduce Redundant Storage: We propose using the L1 cache for storing updated activations. Although prior work stores activations in shared memory [14], this requires multiple activation copies per SM whereas we only require one copy. By storing fewer copies per SM, we enable larger recurrent layer sizes. However, using caches requires TLB accesses which may increase the access latency but it would be small compared to repeatedly loading activations from global memory.

Reduce Memory Bandwidth: Storing the activations in caches significantly reduces the memory bandwidth requirements. However, caches pose an additional challenge since the activations may be evicted. Therefore, an additional mechanism is required to retain all the activations in the cache. To do this efficiently, we will build on prior work on locking cache ways [11], or making the shared memory globally addressable and coherent [27].

Reduce Communication Overhead: Although storing the activations in shared memory [14] avoids some of the overheads of caches, it also has its own set of issues because the shared memory must copy the data back and forth from the global memory to propagate updates. This incurs significant latency overhead as it requires updates to the global memory at the producer SM and loading of the updated activations into the shared memory of the consumer SM at every timestep. Thus, we will optimize the GPU coherence protocol [41] to propagate updated activations directly to the consumers’ L1 cache.

4 Evaluation

There has been a significant effort towards building infrastructure for accurately simulating state-of-the-art GPU architectures in GPGPU-Sim (such as NVIDIA’s Pascal and Volta GPUs [23, 25], including tensor cores [39]) with support for executing Deep Learning benchmarks that use cuBLAS, cuDNN, PyTorch, and TensorFlow [33]. Using the updated GPGPU-Sim infrastructure will allow us to rapidly prototype our ideas, while being confident that the changes are reflective of real hardware. As a first step, we have been augmenting GPGPU-Sim to simulate RNNs from DeepBench [35, 36], which has both training and inference implementations for the three most popular variants of RNN algorithms – Vanilla, GRU [10] and LSTM [22]. With slight modifications, we have been able to execute them with various batch sizes, number of hidden layer units, and timesteps, which will help us evaluate our ideas for different RNNs. In addition to this, we plan to compare against end-to-end solutions such as DeepSpeech2 [1], PRNN [14], and SRU [32].

5 Related Work

Recently, there has been an increasing amount of work on optimizing RNNs, especially at the software level. Research on matrix multiplication level optimizations has resulted in optimized cuDNN implementations that concatenate multiple inputs as a single large matrix through batching or concatenation of weight matrices for networks with wide hidden layers [?] for GPUs. Since our applications are written using cuDNN, our design is built on top of these optimizations. A similar effort occurred on the CPU side with the DeepCPU library [43]. However, CPUs do not provide the extent of parallelism we would require to implement our design.

GPUs have also been recently equipped with dedicated and specialized units for matrix multiplications (tensor cores [3]) which can significantly boost performance as RNNs gain considerably from reduced precision [14]. Since our infrastructure models current NVIDIA GPUs and has support for tensor cores, this is another feature that our work builds upon.

Most prior work on optimizing RNNs for GPUs focuses on modifying the network architecture, pruning network parameters and creating efficient mapping of the network on GPUs [14, 26, 32, 37, 45]. For example, Persistent RNNs [14]
Although this approach works well for some RNNs, it also retains the recurrent weight matrix within SMs over all timesteps. This approach is more general.

References


Suchita Pati


