# GRNN: Low-Latency and Scalable RNN Inference on GPUs

Connor Holmes, Daniel Mawhirter
Department of Computer Science
Colorado School of Mines
Golden, Colorado
{cholmes,dmawhirt}@mymail.mines.edu

Yuxiong He
Microsoft Business AI and Research
Seattle, Washington
yuxhe@microsoft.com

Feng Yan
Department of Computer Science
University of Nevada
Reno, Nevada
fyan@unr.edu

Bo Wu
Department of Computer Science
Colorado School of Mines
Golden, Colorado
bwu@mines.edu

## Abstract

Recurrent neural networks (RNNs) have gained significant attention due to their effectiveness in modeling sequential data, such as text and voice signal. However, due to the complex data dependencies and limited parallelism, current inference libraries for RNNs on GPUs produce either high latency or poor scalability, leading to inefficient resource utilization. Consequently, companies like Microsoft and Facebook use CPUs to serve RNN models.

This work demonstrates the root causes of the unsatisfactory performance of existing implementations for RNN inference on GPUs from several aspects, including poor data reuse, low on-chip resource utilization, and high synchronization overhead. We systematically address these issues and develop a GPU-based RNN inference library, called GRNN, that provides low latency, high throughput, and efficient resource utilization. GRNN 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. Evaluated on extensive benchmarking and real-world applications, we show 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 reduction.

***CCS Concepts*** • **Computer systems organization** → **Architectures**; **Heterogeneous (hybrid) systems**;


*EuroSys '19, March 25–28, 2019, Dresden, Germany*

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303949

***Keywords*** recurrent neural networks, GPUs, deep learning inference

**ACM Reference Format:**
Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. 2019. GRNN: Low-Latency and Scalable RNN Inference on GPUs. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3302424.3303949

## 1 Introduction

Recurrent Neural Networks (RNNs) are a class of important deep neural networks widely deployed in various applications, including text classification [25, 38], question answering [31, 37], speech recognition [12, 16], and machine translation [11, 26]. The key feature of such models is that they carry information across the input sequence through an internal state, preserving the inherent context and hence providing higher modeling accuracy for sequential data. As such, RNNs present both data reuse and complex data dependencies through the repeated execution of the cellular computation graph, demanding quite different optimization techniques compared to other popular network classes like Convolutional Neural networks (CNNs).

RNN deployment consists of two stages that have drastically different computational properties. During the training stage, the RNN model is supplied with a training data set and the weights of the model are iteratively trained through the back-propagation algorithm [20]. To improve training efficiency, modern deep learning systems use large batch sizes, which introduces sufficient data parallelism and enables efficient resource utilization. Once the model is trained, the second stage is to serve the model to perform inference for real requests. To meet Service-Level Agreements (SLAs), a responsive inference engine can only batch several requests, leading to limited data parallelism.

Taking into account the difference between training and serving, companies like Facebook primarily use GPUs for

training RNN models and CPUs for serving them [19]. Notably, a recent RNN inference library called DeepCPU shows that optimized RNN inference on the CPU outperforms two GPU-based implementations by more than 10X [40]. On the one hand, ignoring GPUs for serving means a significant resource waste, especially because many data centers are already equipped with GPUs [7]. Furthermore, it is intuitive that GPUs should be suitable for serving RNN models given that the basic operators in such models, like matrix multiplications are efficiently executed on GPUs. RNN models are a large part of modern data center workloads, comprising 29% of Google's workload on Tensor Processing Units as of 2017 [22].

Motivated by the availability and potential of GPUs for serving RNN models, we characterize three state-of-the-art GPU-based implementations of RNN inference, namely TensorFlow [6], cuDNN [10] and TensorRT [5]. We find that TensorFlow's GPU implementation has up to 90X higher latency than its CPU implementation for multiple common model sizes. Analysis of the source code shows that the GPU implementation repeatedly loads the model weights many times, causing both high latency and low throughput. TensorRT, despite supporting sophisticated optimizations for the operators, has the same problem. CuDNN is the only implementation that addresses the problem and yields better latency than DeepCPU for most configurations. However, it hardly scales to even modest batch sizes (e.g., 5) and wastes the opportunity to take advantage of the GPU's massive parallelism. Moreover, cuDNN often achieves lower hardware efficiency, measured as the fraction of achieved throughput over theoretical peak throughput, on the GPU than DeepCPU does on the CPU.

In this work, we address the following research question: Can a GPU-based RNN inference library achieve low latency, high throughput, and efficient resource utilization? Specifically, the library should provide lower latency than the state-of-the-art CPU implementation even when the model or batch size is small (i.e., limited data prallelism). Moreover, it should outperform all the existing GPU implementations when there is an opportunity to use moderately large batch sizes to improve throughput.

We present a GPU-based library, named GRNN, for serving RNN models to provide a definite answer to this question. To minimize unnecessary global memory accesses and increase data reuse, GRNN applies the persistent threads technique [17, 34, 35, 42] to stash the model in the register files and on-chip shared memory. Although some other GPU-based implementations use the a similar technique, GRNN stands out by systematically addressing three technical challenges. First, to reduce global synchronization overhead, GRNN employs a novel output-based tiling technique to perform only one global synchronization in each time step while satisfying all the data dependencies between operators. Second, to achieve high on-chip resource utilization given GPU's complex architecture, GRNN leverages a flexible thread-to-computation mapping strategy that can make various trade-offs to balance hardware resource usage. Third, to quickly find out the optimal implementation for a given RNN model from a tremendous configuration space, GRNN accurately ranks the performance of different configurations and employs an efficient pruning process that introduces negligible overhead.

GRNN [1] is written in CUDA [2] and supports standard interfaces as cuDNN does. It can be easily integrated in existing deep learning frameworks, such as TensorFlow, Caffe [21], and PyTorch [28] for RNN serving. In our evaluation on a wide spectrum of configurations for two most popular RNN models (i.e., LSTM and GRU), GRNN outperforms the state-of-the-art CPU and GPU implementations by up to 17.46X and 9.2X, respectively. GRNN provides up to 14.6X lower latency for moderate batch sizes on two real-world RNN models. On average, GRNN shows at least 24% better utilization than any of the optimized implementations.

In summary, this paper makes the following major contributions: 1) Characterize existing GPU implementations to understand their limitations for RNN inference; 2) Develop novel flexible tiling and mapping techniques to efficiently utilize the GPU; 3) Propose an accurate comparative model to search for optimal configurations with negligible overhead; 4) Implement a GPU-based library called GRNN that integrate all the proposed techniques to serve RNN models; 5) Evaluate GRNN on benchmarks and real-world applications and demonstrate GRNN's lower latency, higher throughput, and more efficient hardware utilization over state-of-the-art implementations on both CPUs and GPUs.

## 2 Background

### 2.1 Computational Properties of RNN Inference

RNNs have recursive cells that carry over a hidden state to maintain context information. In each iteration, the cell takes one element of the input sequence (e.g., a word in document classification application or a waveform sample from an audio recording) and the previous hidden state as inputs, updates the hidden state, and generates an output. Thus, the length of the input determines how many times the cell is executed. RNNs carry information across elements in the same input sequence, presenting both a challenge (data dependencies) and an opportunity (data reuse) for non-trivial performance optimization.

This paper focuses on two popular variations of RNNs, LSTM and GRU, to illustrate the computational properties. GRU and LSTM have 3 and 4 gates, respectively. Figure 1 shows the operators of one LSTM cell and their dependencies. To produce one output ($o_t$) in iteration $t$, the cell takes an input element ($x_t$) and the hidden state ($h_t$), and executes 8 independent matrix multiplications, two multiplications per

[1] https://github.com/cmikeh2/grnn

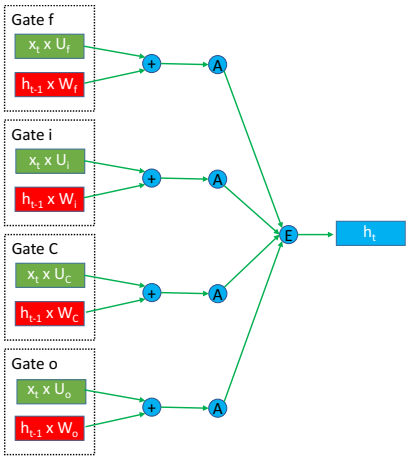



**Figure 1.** Dependency structure of an LSTM cell. Element-wise operations represented by circles are simplified to be single nodes. Green rectangles are time independent matrix multiplications, whereas the red ones are time dependent.

gate. Since the matrix multiplications dominate execution time, for simplicity we use one element-wise operator shown as $E$ to represent the 5 element-wise operations in each iteration. GRU's operators have more complicated dependencies, the details of which will be discussed later in Section 5.

As shown in prior work [40], we can classify the matrix multiplications in two groups. The first group, represented by top rectangles in the gates with $U$ weight matrices, depend solely on the input sequence, while the second group, containing all the other matrix multiplications with $W$ weight matrices, have recursive dependence. We can then partition the computation into two phases. In phase one, we concatenate the elements of the input sequence as one matrix and precompute the sequence-independent matrix multiplications for each iteration. In the second phase, we execute the second group of multiplications as well as the remaining operators to produce the final output. The first phase achieves high throughput as a large matrix multiplication. Consequently, the second phase becomes the bottleneck for RNN inference.

### 2.2 GPU Architecture and Optimization Considerations

As shown in Figure 2, a GPU consists of tens of streaming multi-processors (SMs), a shared L2 cache, the interconnect network, and the off-chip global memory. In the newest Nvidia Volta architecture, each SM is partitioned into shared memory, L1 cache, and 4 scheduling partitions, each capable of independently executing a number of threads. A function running on the GPU is called a kernel. The GPU driver launches a group of thread blocks, all executing the same kernel function. A hardware scheduler dispatches the thread

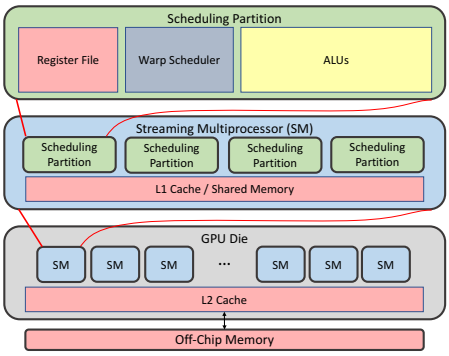



**Figure 2.** A hierarchical view of Nvidia GPU architecture.

blocks to the SMs. Within each SM, the threads of the resident thread blocks are organized into warps (32 threads on Nvidia GPUs), which are further scheduled by the warp scheduler to run on one of the scheduling partition.

GPU threads have different kinds of restrictions for inter-thread communication, depending on how closely in the hierarchy the threads are related. Threads in different thread blocks can communicate with each other through global memory, different warps in the same thread block can communicate through shared memory, and threads in the same warp can communicate through the register file. Since higher-level memory of the hierarchy provides dramatically more bandwidth and lower latency, it is critical to maximize data reuse in registers and shared memory.

## 3 Demand of Low-Latency and Scalable RNN Inference

In this section, we investigate open-sourced and proprietary RNN inference libraries to understand their performance limitations. We then identify the opportunities and challenges for building a low-latency, scalable, GPU-based inference engine.

### 3.1 Poor performance of Open-Sourced GPU-Based Inference Engines

To understand the RNN inference performance of state-of-the-art deep learning systems, we experiment with TensorFlow (v1.10) on a machine with an Intel CPU and an Nvidia GPU (details in Section 8). We run both CPU and GPU-based implementations on a LSTM cell, with 64, 256, and 1024 for the input and hidden sizes. Surprisingly we find that the GPU implementation has 6.3X worse latency than the CPU implementation, despite the 8.1X higher theoretical floating point throughput for the GPU.

TensorFlow's GPU implementation fuses the eight independent matrix multiplications of LSTM into a single one to provide better throughput by increasing the amount of work done by a kernel invocation. This also automatically ensures that all gate dependencies are prepared simultaneously.

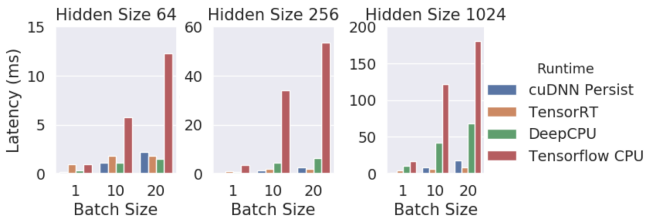



**Figure 3.** Latency comparisons of TensorFlow CPU and 3 proprietary libraries for RNN Inference.

However, despite following the general guidelines of optimizing GPU performance (i.e., increasing data parallelism), the implementation suffers from inefficient resource utilization due to two limitations. **L1 - Repeated data loading.** On each iteration, the entire weight matrix must be loaded from global memory, or in best case scenario the L2 cache. **L2 - Large kernel launch overhead.** TensorFlow invokes one or more kernel functions for each iteration, depending on the degree of operation fusion. The implicit barrier between kernel invocations satisfies the data dependencies but incurs non-trivial overhead.

### 3.2 Poor Scalability of Proprietary Libraries

We study three popular proprietary libraries: DeepCPU [40], TensorRT [5], and cuDNN [10] (default RNN API). DeepCPU is a highly optimized RNN inference library for CPUs widely deployed in Microsoft's production systems. TensorRT and cuDNN are Nvidia's libraries to accelerate inference and training, respectively. Figure 3 demonstrates that all the proprietary libraries outperform TensorFlow's CPU implementations. Note that since DeepCPU is not publicly available, we use the performance numbers reported in the original paper on a similar CPU. We observe that while cuDNN outperforms TensorRT for 5 out of the 9 configurations, it experiences poor scalability. Increasing the batch size from 1 to 20 leads to on average 18.4X latency degradation, which indicates that the implementation does not well reuse shared weight data across the batched inputs.

Interestingly, we observe that DeepCPU outperforms cuDNN for the smallest model. For larger models, cuDNN produces superior performance to DeepCPU due to increased data parallelism, indicating DeepCPU's limited scaling to large model sizes. However, the cuDNN's better performance may simply come from the significantly larger throughput of the GPU instead of a more efficient implementation. For example, when the hidden size is 256 and batch size 10, DeepCPU reaches 14% of theoretical floating point throughput, while cuDNN only achieves 6.26% of the theoretical throughput.

To summarize, we find that all the proprietary libraries have a serious limitation: **L3 - Poor scalability in either model size or batch size.** While the DeepCPU's poor scalability comes from the moderate theoretical throughput of the CPU, the GPU libraries poor scalability roots in the inefficient implementations that cause low floating point throughput.

### 3.3 Opportunities and Challenges

Although the small dimensionality of RNNs leads to limited data parallelism, it also suggests that the total working set is small. For example, an LSTM model with hidden dimension of 1024 needs $1024 \times 1024 \times 4$ (number of gates) $\times$ 4 (number of bytes of a weight) $= 16M$ bytes for the weight data. The Nvidia Volta GPU has in aggregate 20MB register file space and 10MB shared memory, which is large enough to fit the model. However, despite the reuse of the weights and the state across time steps, they cannot stay in the on-chip memory across kernel invocations. Fortunately, the persistent threads based approach [17] well addresses the problem by persisting the weights in register file and shared memory across time steps (addressing **L1**). Specifically, it launches just enough threads to saturate the GPU, which at the beginning load the weights in the register file, perform the operators, and synchronize with each other at the end of each time step. As such, we just launch one kernel to perform computation for the whole sequence (addressing **L2**). Once the weight data can be reused in the register file, the implementation has potential to improve scalability (addressing **L3**).

While the sufficient on-chip memory resource and the persistent threads based approach provide great opportunities to substantially improve the performance of RNN inference, an efficient implementation faces three challenges.

**C1.** As prior work [30, 36] shows, global synchronization of persistent threads has non-trivial overhead. However, basic implementations may incur too many synchronizations to handle the data dependencies between operators, canceling the benefits of the persistent threads based approach.

**C2.** Once operators or partial operators are assigned to persistent thread blocks, mapping the many threads to operators for maximum efficiency remains a complicated question. The problem is exacerbated by the various types of hardware resources each SM has, such as shared memory, registers, ALUs, warp schedulers, and so on.

**C3.** The numerous mapping configurations that exist at the global level (distributing operators to SMs) and the SM level (mapping threads to computation) create a tremendous kernel configuration space to navigate. Since optimization goals are oftentimes conflicting (e.g., improving data reuse may incur computation overhead), the configurations represent various trade-offs for resource utilization. Exhaustive search for the optimal configuration incurs prohibitive overhead.

## 4 Overview of GRNN

GRNN is a GPU-based library to serve RNN models with low latency, high throughput, and efficient resource utilization

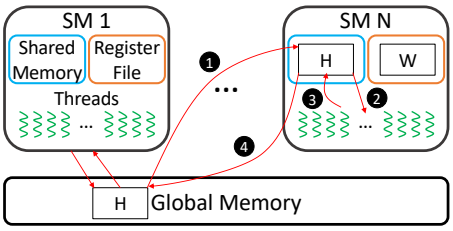



**Figure 4.** Overview of GRNN's workflow for a time step.

through a combination of tiling, mapping, and modeling techniques. It supports standard interfaces as cuDNN RNN family of operators such as LSTMs and GRUs. It can be easily integrated into existing DL frameworks such as Tensorflow, Caffe [21] and PyTorch [28] for RNN serving.

GRNN builds on top of the persistent threads technique. At the beginning of the inference, it tiles the output, and loads the corresponding weight data for each tile to the register files and persists them across multiple steps of the entire RNN computation. As shown in Figure 4, GRNN then performs each time step as follows. First, GRNN replicates the global hidden state $H$ in the shared memory of each SM (❶). Next, GRNN maps the threads to the weight matrix and the state (❷) and executes highly optimized operators to produce an updated local copy of $H$ (❸). Finally, GRNN synchronizes all SMs and merges the local slices of $H$ into the global copy (❹). Note that when the model is too large to fit in the register file, GRNN's implementation defaults to the traditional approach (i.e., fusing independent matrix multiplications) to perform inference. In this case, the amount of data parallelism is sufficient to exploit the GPU's abundant compute resources.

GRNN addresses the three challenges mentioned in Section 3 to perform efficient inference for RNNs. To avoid global synchronization overhead (**C1**), GRNN carefully organizes the data layout of the model and employs output-based tiling. As such, GRNN only requires one global synchronization for each time step, though the numerous operators have complex data dependencies. The number of synchronizations is optimal because the SMs have to update the global copy of the hidden state through a synchronization to move to the next time step. In addition, GRNN includes a highly optimized implementation of global synchronization tailored for the unique features of RNNs. To maximize on-chip resource utilization (**C2**), GRNN implements a flexible mapping strategy, which balances register usage, locality of shared memory accesses, and the critical path of the numerical operators. To navigate the tremendous kernel configuration space and select the optimal kernel configuration (**C3**), GRNN leverages an accurate performance model to predict the top $K$ configurations with negligible overhead, where $K$ is tunable. GRNN then generates and compiles $K$ kernels corresponding to the predicted configurations. After a calibration process to run all the $K$ kernels, the one with the best performance is returned for serving real requests. We next explain each of these techniques in detail.

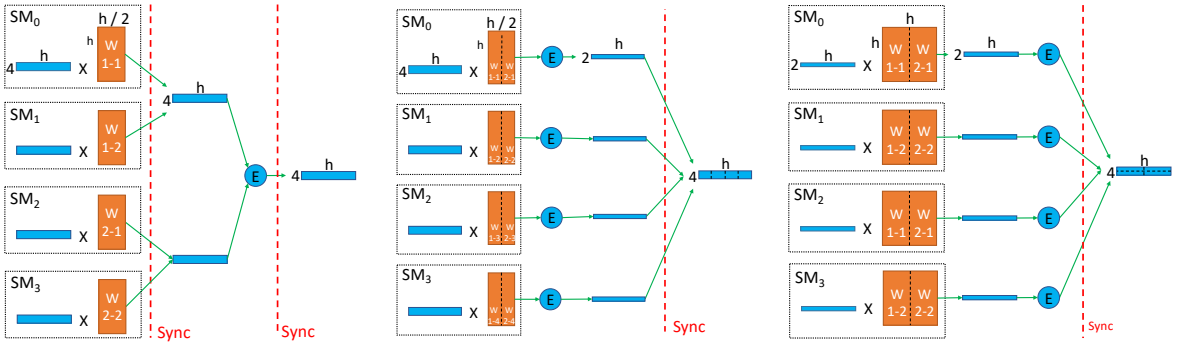
## 5 Tiling-Based Persistent Kernel

In this section, we first describe GRNN's strategy to tile the output matrix and the computation across different SMs to reduce synchronization overhead, applicable to both LSTM and GRU. We then present the special considerations for GRU and its additional dependencies.

The persistent threads based approach provides a mechanism to persist data in register files, but it does not imply how to partition the computation for optimal performance. Yet different partitioning strategies have dramatically different performance characteristics and results. A basic persitent approach assigns entire operators to SMs. Prior work [4] applies this strategy to accelerate a model to generate sound waveforms and produces state-of-the-art performance. However, such a strategy is inappropriate for RNNs, because the weight matrix of a single matrix multiplication is often too large to fit into the register file of a single SM. For example, one of the weight matrices for an RNN cell with hidden size 128—a relatively small cell—requires 256 KB of memory. This already consumes the entirety of the register file on an SM without including other necessary execution data, such as indexing variables. Therefore, partitioning operators across SMs is essential for non-trivial models.

Figure 5(a) demonstrates a more advanced approach to using persistent threads. Given two independent matrix multiplications in each timestep and a GPU of 4 SMs, this approach uses two SMs to perform each multiplication. It splits the weight matrix into halves, each being persisted in one SM. The aggregate register files successfully address the capacity problem, but the approach has to perform a global synchronization after the matrix multiplications for the output vectors to be ready for the following element-wise operators. After performing the element-wise operators, this approach still needs another global synchronization to produce one single state vector for the next timestep. For LSTM, the approach incurs 8 more synchronizations other than the final synchronization due to the 8 independent matrix multiplications. As this approach initializes tiling by partitioning the inputs, we refer to it as input-based tiling.

To minimize synchronization overhead, GRNN instead tiles the output between SMs. Working backwards through the dependencies from the output tile, GRNN determines which weights from each of the weight matrices will be required to produce the assigned output tile. These weights are then co-located on the same SM, enabling the SM to perform all element-wise operations and activations without any inter-SM communication within a timestep until the last global synchronization. As the example shows in Figure 5(b), the output is split vertically into 4 equal-sized tiles, each of which should be produced by a distinct SM. Consequently,




**(a)** Input-based tiling strategy that tiles each matrix multiplication separately.

**(b)** Output-based tiling configuration with vertical splits of the output.

**(c)** Output-based tiling configuration with both vertical and horizontal splits.

**Figure 5.** Illustration of tiling strategies for matrix multiplications with output dependencies.

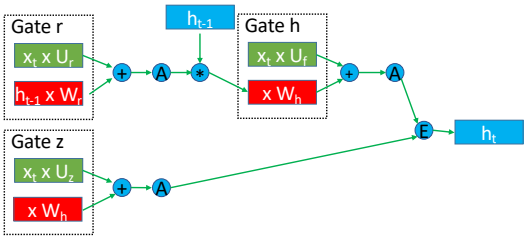
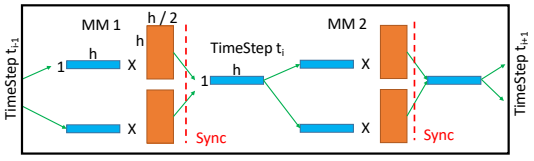
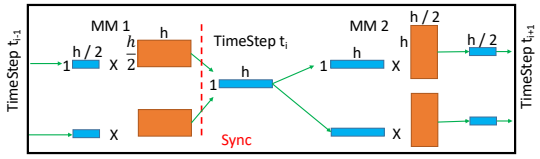



**Figure 6.** Dependency structure of the canonical GRU cell. Critically, the $h$ gate is dependent on the output of the $r$ gate and the final hidden state requires direct outputs from the $h$ and $z$ gates.




**(a)** Basic synchronization strategy which synchronizes between each dependent matrix multiplication.




**(b)** Single synchronization strategy that precomputes partial sums and uses reduction to eliminate a synchronization.

**Figure 7.** Synchronization strategies for GRU Cells

each SM persists a quarter of each weight matrix and perform a sequence of operations. This technique eliminates the synchronizations in the of the input-based tiling method after each matrix multiplication, and minimizes the number of global synchronizations.

Figure 5(b) shows just one way to tile the output, while GRNN supports arbitrary tile sizes. For example, assuming that a whole weight matrix can fit into the register file, GRNN can select a tile size of $2 \times h$ as illustrated in Figure 5(c). Because each tile on the top shares the column indices with the tile below it, every matrix is duplicated in the register files, increasing register pressure. However, the benefit is that the state vector, once loaded from shared memory, is reused $h$ times, 2 times more compared to the previous method due to the doubled width of the persisted weight matrix. Flexible tiling adapts GRNN to different workload and hardware.

### 5.1 Special Considerations for GRU

The GRU cell, unlike LSTM, has dependent matrix multiplications. As Figure 6 shows, a GRU cell has 3 gates, each containing 2 multiplications. Since the $h$ gate depends on the $r$ gate, the time-dependent multiplication (the bottom multiplication) in the $h$ gate must wait till the multiplications in the $r$ gate finish. To ease the discussion, we focus on these two dependent multiplications and explain GRNN's technique to minimize synchronization overhead.

Figure 7(a) illustrates how a basic method deals with the dependency. Each of the matrix multiplications is tiled by vertically partitioning the weigh matrices. Since the second matrix multiplication needs the full output of the first matrix multiplication, this implementation has to perform two global synchronizations, each after an multiplication. To eliminate one synchronization, GRNN partitions the weight matrix of the first multiplication horizontally and adds a global synchronization after it to produce the output vector by reducing the partial sums as shown in Figure 7(b). Hence, the second multiplication can be performed with the full input vector and vertically partitioned weight matrix. Note that then the output vectors are not merged but directly passed to the first multiplication in the next timestep. Although

the one-synchronization method reduces synchronization overhead, it incurs additional computation overhead for the reduction. Due to these trade-offs, selecting between them will be addressed on a model-by-model basis in Section 7.

# 6 Flexible Thread Mapping to Balance SM Resource Usage

Once GRNN assigns an output tile to a thread block, the block should perform a series of operators on the persistent weight data in the register file and the state vector in shared memory. There exist a variety of ways or configurations to map the threads to the computation, especially because a thread block size is non-trivial, typically over a hundred. The mapping configurations have drastically different degrees of impact on data reuse, critical path of the computation, and latency hiding. The goal of GRNN is to develop a flexible mapping strategy, which enables various mapping configurations that contains the optimal and have clear trade-offs for modeling.

Since matrix multiplications dominate execution time, we focus on them to explain the key techniques of GRNN. To further simplify the discussion, we assume the thread block only needs to compute one matrix multiplication. Then a mapping configuration determines which threads should work together to produce which output elements in the results matrix. As mentioned in Section 2, the thread block has a hierarchical organization of threads. A thread block consists of a number of warps, each of which further contains a constant number of threads. GRNN leverages this 2-level organization and implements a 2-level thread mapping. First, GRNN partitions the output matrix and assigns each partition to a warp. Second, GRNN assigns the threads inside each warp to specific output elements.

For the first-level mapping, GRNN simply tiles the output matrix vertically and uses each warp to produce a distinct equal-sized tile. Hence, a warp performs a smaller matrix multiplication that has three steps. The first step loads the state vector into the register file. GRNN should strive to minimize the number of loads of the state vector. The second step computes the partial sums, because the number of threads is typically larger than the number of output elements. Finally, GRNN reduce the partial sums to produce the final outputs. For each output element, GRNN should try to minimize the number of partial sums for reduced computation time. We next show two contrasting mapping configurations to optimize step 1 and step 2, respectively. We then present GRNN's flexible mapping strategy to cover a spectrum of configurations for the best trade-off.

## 6.1 Mapping for Minimized Shared Memory Accesses

This mapping configuration uses the whole warp to produce the output elements sequentially. All the threads run in parallel when producing each element. Figure 8 (a) shows an example, which assumes that the vector length is 4, the weight matrix's dimensions are $8 \times 4$, and the warp has 8 threads. The numbers in the weight matrix demonstrate the IDs of the threads, whose registers the corresponding weights persist on. Since all the weights in the same row belong to the same thread, that thread accesses the same element in the state vector to perform the dot produces. Hence, this mapping configuration minimizes shared memory accesses. The downside, however, is that the warp produces 8 partial sums on the different threads for the reduction step, incuring non-trivial computation overhead.

## 6.2 Mapping for Minimized Reduction Overhead

This mapping configuration addresses the large reduction overhead by assigning as few threads as possible to produce an output elements. Specifically, given $N$ threads and $K$ output elements, a *work group* of $N/K$ threads are mapped to each output element without wasting any thread. In the example shown in Figure 8 (b), 2 threads for each output element produce 2 partial sums, substantially reducing reduction overhead compared to the previous mapping configuration. However, a thread now persists 4 weights in the same column, indicating that it has to access 4 different elements in the state vector. In other words, the state vector has to be loaded 4 times, without any reuse.

## 6.3 Fully Configurable Mapping

The two aforementioned mapping configurations represent two extreme ways to map threads to computation. Which produces better performance depends on a number of factors, including shared memory access latency/bandwidth, warp size, matrix multiplication dimensions, and so on. It is not surprising if a compromised mapping configuration achieves superior performance to both of them due to a better trade-off between communication and computation overhead. For instance, Figure 8 (c) shows another configuration that uses a work group of 4 threads to produce 2 output elements sequentially. It demands 2 loads of the state vector (better than the first mapping configuration) and produces 4 partial sums (better than the second mapping configuration), which may turn out to be the optimal for final performance. This insight motivates GRNN to implement a fully configurable mapping strategy. GRNN introduces the concept of work groups of configurable sizes, each producing a subset of the output elements sequentially.

In practice, GRNN deals with various types of operators and multiple dominant matrix multiplications for each timestep. We find the fully configurable mapping strategy a powerful idea which can be generally applied. Particularly, though the basic idea remains the same, we apply its two variations to GRU because of the more complex dependencies compared to LSTM. We omit the details due to the space limitations.

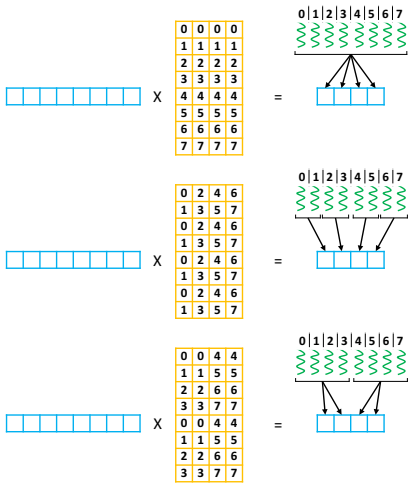


**Figure 8.** Three thread-to-data mapping strategies.

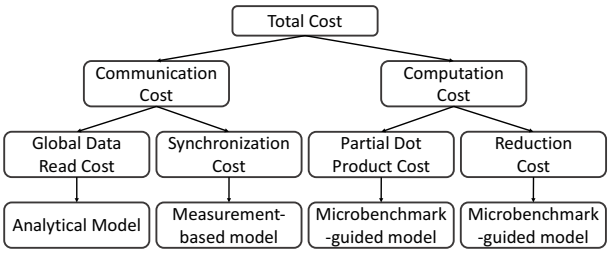



**Figure 9.** Overview of the performance model.

Model Input parameters
- Hidden Size (HS); Batch Size (BS)

Hardware Parameters
- Number of SMs (SMs); Warp Size(SW)
- Number of Warp Schedulers (NS)

Tunable Parameters
- Tile Width (TW); Tile Height (TH)
- Work Groups per Thread Block (GPB); Work Group Size (WS)

Measured Parameters
- Synchronization Cost (SC); L2 Cache Bandwidth (L2B)
- FMA Cost (FC); FMAs per shuffle (FPS)

Derived Parameters
- $global\_data\_SM = TH \times HS$ (1)
- $active\_SMs = ceil(^{HS}/_{TW}) \times ceil(^{BS}/_{TH})$ (2)
- $global\_data\_all = active\_SMs \times global\_data\_SM$ (3)
- $thread\_block\_size = GPB \times WS$ (4)
- $sm\_reg\_pressure = TW \times HS + SW \times thread\_block\_size$ (5)
- $warps\_per\_scheduler = \frac{thread\_block\_size}{SW \times NS}$ (6)
- $sub\_tile\_width = {}^{TW}/_{GPB}$ (7)
- $reduction\_width = \frac{WS}{sub\_tile\_width}$ (8)
- $Sequential\_legnth = \frac{HS}{reduction\_width}$ (9)

Costs
- Data Movement Cost
  - $GC = \frac{global\_data\_SM \times ceil(^{2 \times activ\_SMs}/_{SMs})}{FC \times L2B}$ (10)
- Partial Dot Product Cost
  - $Baseline\_PC = TH \times sequential\_length$ (11)
  - $PC = \frac{Baseline\_PC}{S1(warps\_per\_scheduler) \times S2(sub\_tile\_width)}$ (12)
- Reduction Cost
  - $Baseline\_RC = log_2(reduction\_width) \times FPS \times subtile\_width \times TH$ (13)
  - $RC = \frac{Baseline\_RC}{S3(warps\_per\_scheduler)}$ (14)
- Total Cost
  - $TC = GC + PC + RC + SC$ (15)

**Figure 10.** Performance Model.

## 7 Performance Modeling and Configuration Selection

The previous two sections describe GRNN's capability of arbitrarily tiling the outputs to divide work among SMs and flexibly map threads to computation to balance resource utilization. These techniques introduce 4 parameters, which compose a tremendous configuration space. For example, for a LSTM model of hidden size 256 and batch size 20, the total number of configurations in the space is over 100,000. Exhaustive search is prohibitive, especially because the kernel is templated to enable unrolling, so running a configuration requiring a distinct compilation.

To quickly find high-performing configurations, GRNN constructs a performance model using a hybrid approach, combining analytical models with light-weighted measurements and benchmarking results. Instead of predicting execution time, the model aims at ranking the performance of all the configurations. We next describe the performance model.

### 7.1 Performance Model

Figure 9 shows a top-down view of the performance model. We estimate the cost of a configuration for a single timestep according to two components: communication cost and computation cost. The communication cost arises from two sources: state matrix movement and synchronization. The state matrix must be loaded from global memory to shared memory at the beginning of the timestep and stored back at the end. We develop an analytical model to estimate the cost of that movement in Section 7.1.2. The cost of global synchronization can be easily estimated by a rule-based model. We break down the computation cost into partial dot product cost and reduction cost because GRNN's mapping strategy (explained in Section 6) divides the matrix multiplication into two phases: computation of partial sums and reduction. It is difficult to build accurate analytical models for these two costs due to the various optimizations applied by the compiler and the architecture. We address this problem by designing micro-benchmarks to facilitate modeling as explained in Section 7.1.3. The performance model sums up the 4 costs shown at the bottom of Figure 9 to produce the total cost, as their execution cannot be overlapped due to data dependencies.

#### 7.1.1 Model Parameters

As Figure 10 shows, the performance model uses five categories of parameters. The parameters in the first two categories (model inputs and hardware parameters) are straightforward to obtain. The third category contains the tunable parameters introduced by GRNN's tiling and mapping techniques. The fourth category has parameters measured by micro benchmarks. The last category contains all the derived parameters.

During each timestep, each SM reads in the hidden state (left-hand side matrix for the matrix multiplication). As Equation (1) shows, its height equals $tile\_height$ of the output tile, its width the same as hidden size. Since each SM processes a single output tile, Equation (2) computes the number of active SMs as the number of tiles. From these two equations, Equation (3) determines the total amount of data loaded from the global memory to shared memory. Critically this demonstrates that increasing tile width decreases total memory reads proportionally, but increasing the tile height has no effect.

The thread block size is the product of the number of work groups and the work group size, which is used to compute the register pressure for each SM (Equation 5) and number warps assigned to each warp scheduler (Equation (6)). The register pressure metric is not used to model performance, but helps prune configurations whose register pressure is larger than the capacity of the register file. The sub-tile width computed by Equation (7), together with the work group size, determines how many partial dot products are produced (i.e., reduction width) as shown in Equation (8). Finally, Equation (9) computes the number of Fused Multiply-Adds (FMAs) each thread performs named sequential length.

#### 7.1.2 Communication Cost

**Global data loading cost.** The global data loading cost of a configuration determines the effective latency of loading the hidden state. We assume that for all model configurations all memory requests are able to hit L2 cache—the L2 cache is more than a magnitude larger than the maximum hidden state footprint and prefetching helps to ensure data resides in L2. Pairs of SMs share a bus to access inter-SM resources, including the global L2 cache. So if the number of thread blocks (i.e., active SMs) crosses half of the SMs, the cost is doubled due to congestion on the shared bus. Equation (10) captures this effect and normalizes the cost to the number of FMAs.

**Synchronization cost.** While the synchronization cost is high from an absolute standpoint, our implementation has practically no marginal cost associated with it. As such, for model configurations, such as LSTM, where all potential configurations have the same number of synchronizations per timestep, the synchronization cost is not included in the model. Otherwise (e.g., the GRU implementation with 2 synchronizations per timestep), the cost of the synchronization is modeled as two round trip accesses to L2 cache, which holds the global variables for the synchronizations.

#### 7.1.3 Computation Cost

**Partial dot produce cost.** Modeling the computation of partial dot products analytically is difficult due to two reasons. First, the compiler and architecture implement significant optimizations like instruction re-ordering and multi-threading to hide latency under low utilization conditions. Second, when the hardware resources are saturated, these optimizations tend to have little or no benefits. For example, when we increase the number of warps per warp scheduler from 1 to 2, multi-threading helps the concurrent warps hide each other's memory access latency, producing significant benefit. But when the number of warps assigned to that scheduler is already large, further increasing its load does not increase throughput. The same rationale also applies to data reuse, which beyond some point do not improve throughput due to the saturated data path.

Based on this insight, we design a micro-benchmark to run a number of FMAs with varied numbers of warps per scheduler and the degrees of data reuse (controlled by sub-tile size). We use the benchmarking results to fit two functions $S1$ and $S2$ to predict speedups for increased number of warps per scheduler and increased sub-tile sizes, respectively, over a baseline with one warp per scheduler and no data reuse. Note that we choose to not include both metrics in one single function to reduce benchmarking overhead (linear vs quadratic cost). Given the tile height (the number of output elements a thread needs to work on) and sequential length (the number of FMAs to perform for each output element), the baseline cost in terms of the number of FMAs is given by Equation (11). We then estimate the final cost by dividing the baseline cost by the product of the two predicted speedups.

**Reduction cost.** The reduction cost is also difficult to model analytically due to similar reasons as for the partial dot product cost. Fortunately, since all the input data for this phase are in the register files, we only need to fit one speedup function (S3) for the number of warps per scheduler metric. To estimate the cost of the baseline, we assume a warp scheduler is only assigned one warp. The number of reductions is given by the product of sub-tile width and tile height. To normalize the reduction cost to FMAs, we measure the latency of a single shuffle operation to implement reduction, which is as long as $FPS$ (7 for the Nvidia Volta architecture) FMAs. Equation 13 shows the formula to compute the baseline cost in terms of FMAs, and Equation 14 applies the speedup function (S3) to take into account the benefit of having concurrent warps.

| Model Parameters | | LSTM | | | | GRU | |
|---|---|---|---|---|---|---|---|
| **Hidden Size** | **Batch Size** | **GRNN Top-5** | **cuDNN Traditional** | **cuDNN Persistent** | **DeepCPU** | **GRNN Top-5** | **DeepCPU** |
| 64 | 1 | 0.2 | 0.96 | 0.19 | 0.31 | 0.18 | 0.7 |
| 64 | 10 | 0.23 | 1.79 | 1.14 | 1.1 | 0.19 | 1.1 |
| 64 | 20 | 0.24 | 1.8 | 2.24 | 1.5 | 0.2 | 1.5 |
| 256 | 1 | 0.3 | 1.1 | 0.21 | 0.74 | 0.28 | 0.9 |
| 256 | 10 | 0.38 | 1.89 | 1.28 | 4.4 | 0.32 | 3.7 |
| 256 | 20 | 0.49 | 1.98 | 2.54 | 6.4 | 0.43 | 5.4 |
| 1024 | 1 | 0.63 | 4.43 | 1.22 | 11 | 0.68 | 8.4 |
| 1024 | 10 | 4.3 | 6.1 | 9.07 | 42 | 3.16 | 36 |
| 1024 | 20 | 8.01 | 8.51 | 18.42 | 68 | 6.32 | 60 |

**Table 1.** Execution latencies for LSTM RNNs, measured in milliseconds. All configurations have sequence length 100.

### 7.2 Configuration Selection

GRNN implements the following procedure to select a high-performing configuration. Starting from a full set of possible configurations, GRNN first removes the configurations that use more registers than the SMs can provide. GRNN next applies the performance model to rank all the remaining configurations and selects the top $K$ configurations, where $K$ is a small positive integer specified by the user. GRNN finally compiles and runs each of these configurations, and uses the fastest configuration to serve real requests. The procedure effectively reduces the number of configurations to benchmark from tens of thousands to a small constant, usually less than 10. Given a model of hidden size 256, this translates to 4 orders of magnitude improvement of the cost. Thanks to the accurate performance model, when $K$ is 1, the selected configuration achieves 96.7% of the optimal throughput. When $K$ is 5, the 98.0% of optimal throughput is achieved.

## 8 Evaluation

In this section, we evaluate GRNN against state-of-the-art implementations on a wide spectrum of benchmark configurations and two real-world models. Our evaluation shows that GRNN outperforms the other implementations in terms of latency, scalability, and achieved throughput. The highlights of the results are as follows:

- GRNN always outperforms the state-of-the-art CPU implementation even if the model and batch sizes are both small, with a maximum latency reduction of 94%. GRNN scales up to 7.4X better than state-of-the-art GPU implementations while producing the lowest latency for most of the configurations. On average, GRNN improves resource utilization over all the other implementations.
- GRNN's performance model is highly accurate. Its Top-1 and Top-5 configuration performs only 1.03X and 1.02X worse than the optimal configuration found through exhaustive search.
- For end-to-end inference, GRNN achieves up to 14.6X speedup over state-of-the-art GPU implementations for two real-world models with non-trivial architectures.

### 8.1 Experimental Setup

**Machine environment.** GPU runtimes are benchmarked on an Nvidia Titan V system with a Xeon E3-1286 v3 host processor paired with 32 GB of RAM and running Ubuntu 16.04. The Titan V has 80 SMs and a total of 5120 FP32 cores alongside 12 GB of HBM2 memory. The theoretical peak throughput of the Titan V is 13.67 TFLOPs. The DeepCPU configurations are from a dual socket E5-2650 v4 system, each socket having 12 cores at 2.2 GHz.

**Comparison systems.** While numerous deep learning environments exist, we choose DeepCPU, and cuDNN (v7.2)'s traditional and persistent threads based implementations. DeepCPU is the highest performing CPU implementation with public data and it easily outperforms the CPU implementations of popular frameworks such as Caffe, CNTK, and Tensorflow. cuDNN's heavily optimized GPU implementations are considered state of the art for the GPU. Both the traditional (non-persistent) and persistent versions are considered because cuDNN does not provide extensive guidance on which implementation to use within a deep learning framework and each has a different latency-throughput curve. TensorRT is a popular inference engine compatible with deep learning training environments. Since TensorRT makes direct calls to cuDNN for RNNs, we do not explicitly include its performance. Note that cuDNN implements a simplified GRU that eliminates the dependency between matrix multiplications. Since GRNN implements the canonical GRU cell, performance comparisons between the two libraries would be meaningless and are not included.

### 8.2 Benchmark Results

We vary the hidden and batch dimensions with a fixed sequence length of 100 and the input dimension the same as the hidden dimension. We run each configuration 1000 times and report the average latency. Table 1 shows all the latency results and Figure 11 plots the speedup results. We next

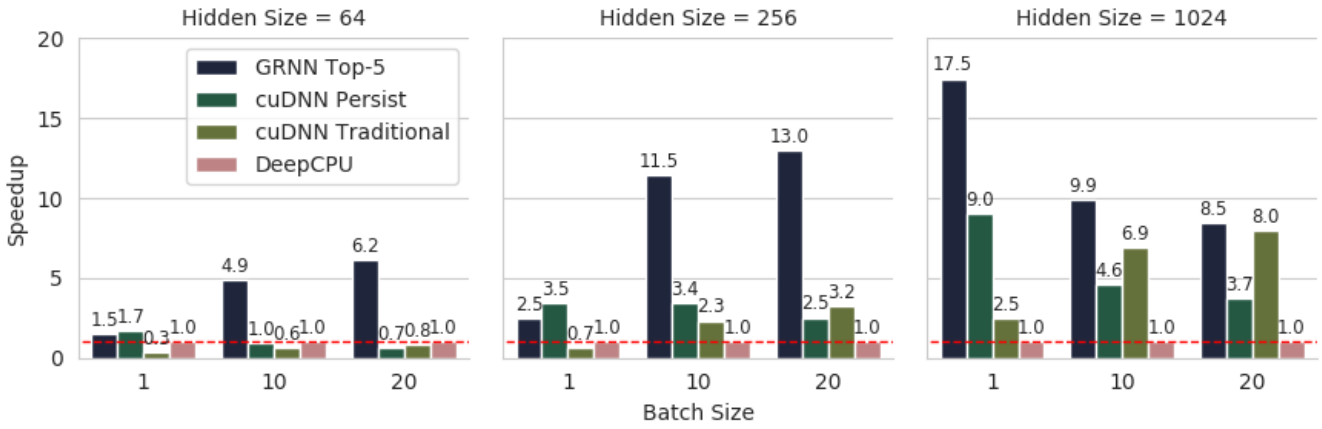



**Figure 11.** Speedup provided by each library compared to the DeepCPU baseline

discuss them by comparing GRNN with each of other evaluated implementations. The latencies reported measure the layer execution time with the cost of sending/receiving inputs/outputs over the PCIe bus. This represents a worst-case scenario for GRNN, as most models require other operations, such as embedding lookups or classification, to be performed before and after layer execution, meaning that oftentimes the data already resides or requires further execution on the GPU.

**GRNN vs cuDNN Traditional:** cuDNN Traditional is built directly on top of the highly effective cuBLAS [1] matrix multiplication kernel. It yields on average 6.67X and 4.21X higher latency than GRNN when the hidden size is 64 and 256, respectively. The reason is that the traditional cuDNN implementation performs a new kernel launch each iteration, the overhead of loading the weights into the register file on each iteration dominates execution time. GRNN's ability to persist the weights and reduce synchronization overhead plays a critical role when the data parallelism is limited for small model size and batch size. When the hidden size is 1024, the abundant data prallelism makes it easier to exploit compute resources. Hence, the performance gap between GRNN and cuDNN shrinks, but GRNN still reduces the latency by 40.6% on average.

**GRNN vs cuDNN Persistent:** cuDNN Persistent, like GRNN, persists its weights between timesteps in the register file. However, cuDNN persistent does not appear to reuse weights effectively in order to provide scalable performance. When the batch size is 5 (a common size used in production [40]), GRNN provides on average 3.46X speedup over cuDNN Persistent, even if cuDNN outperforms GRNN for unbatched inference (possibly due to heavy assembly-level optimizations). As the batch size further increases to 20, the performance gap becomes quite large. For example, when hidden size is 64, GRNN is 9.2X faster than cuDNN Persistent.

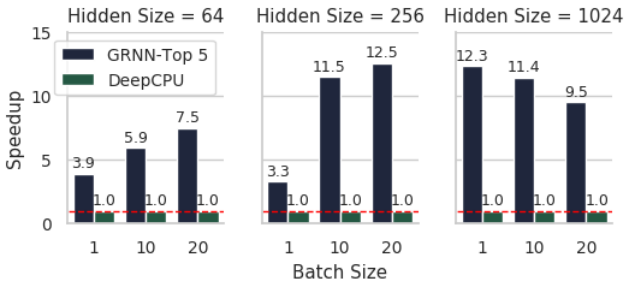



**Figure 12.** Speedup provided by GRNN for GRU models over the DeepCPU baseline.

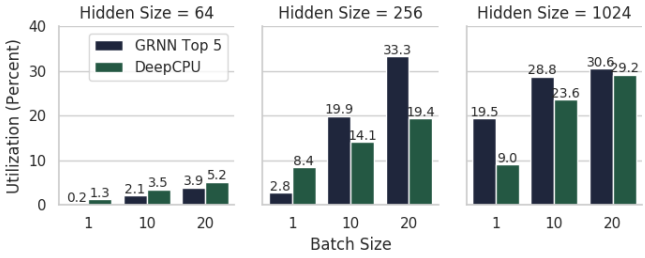



**Figure 13.** Normalized achieved throughput by DeepCPU and GRNN.

**GRNN vs DeepCPU:** The key optimization of DeepCPU is minimizing data movement between last-level shared cache and private cache. It enables DeepCPU outperforms cuDNN Traditional and cuDNN Persistent for some LSTM configurations. For instance, when the hidden size is 64 and batch size 1, DeepCPU is 3.1X faster than cuDNN Traditional. For the same hidden size and batch size 20, DeepCPU improves latency by 16.7% over cuDNN Persistent. However, DeepCPU

| Work Groups | Group Size | Latency | Cost | Cost Rank |
|---|---|---|---|---|
| 128 | 2 | 0.57 | 449 | 17 |
| 64 | 4 | 0.5 | 322 | 3 |
| 32 | 8 | 0.5 | 252 | 1 |
| 16 | 16 | 0.6 | 443 | 14 |
| 8 | 32 | 0.93 | 561 | 35 |

**Table 2.** Model parameters: hidden size = 256, batch size = 20. Output tile dimensions: width = 32, height = 2.

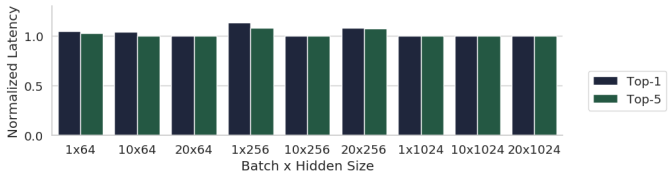



**Figure 14.** Normalized latency over Oracle.

scales poorly in model sizes and shows significantly worse results for hidden size 1024. GRNN is the only GPU implementation that surpasses DeepCPU for all the LSTM configurations with up to 17.5X speedup. For the GRU configurations, GRNN also produces superior performance, reducing the latency by up to 12.5X compared with DeepCPU. Although the GPU architecture is more difficult to optimize due to its various types of hardware resources, GRNN also demonstrates better throughput once normalizing against maximum floating point throughput compared with DeepCPU for LSTM. As Figure 13 shows, GRNN on average achieves 15.7% of the GPU's theoretical throughput, while DeepCPU achieves 12.6% of the CPU's theoretical throughput.

### 8.3 Accuracy of the Performance Model

Due to space limitations, we only show the accuracy of the performance model for LSTM in Figure 14 but the results for GRU are similar. Observe that the Top-1 predicted configuration matches the optimal configuration by the Oracle for 5 out of the 9 input configurations. In the worst case, GRNN's Top-1 configuration shows 1.14X latency increase over the Oracle. The Top-5 configuration reduces the maximum latency increase to 1.08X. On average, Top-1 and Top-5 configurations show just 1.03X and 1.02X latency increases, respectively, while searching orders-of-magnitude more quickly than a brute force search. Note that randomly selecting configurations would yield poor performance, because on average, the Top-1 configuration outperforms the median configuration by 1.8X. These results confirm the complexity of the trade-off to balance resource utilization and the necessity to build a sophisticated performance model to address it.

To more closely examine the results, we use the model configuration of hidden size 256 and batch size 20 (See Table 2). While all of these configurations produce the same output tile with the same number of threads, the least performant configuration has 1.9X worse latency than the fastest configuration. The performance model is able to discern that the reduction costs are too high for the bottom two configurations (118 and 236, respectively). Similarly, it could discern the partial produce cost (409) is too high for the first configuration. The model successfully selects the third configuration as the top-1 result.

| Hidden | Batch | Single | Double |
|---|---|---|---|
| 64 | 1 | 0.181 | 0.28 |
| 64 | 10 | 0.186 | 0.285 |
| 64 | 20 | 0.201 | 0.307 |
| 256 | 1 | 0.275 | 0.372 |
| 256 | 10 | 0.323 | 0.411 |
| 256 | 20 | 0.431 | 0.498 |
| 1024 | 1 | 6.46 | 0.683 |
| 1024 | 10 | 64.76 | 3.162 |
| 1024 | 20 | 129.52 | 6.324 |

**Table 3.** Comparison between Top-5 performance of Single and Double Synchronization GRU

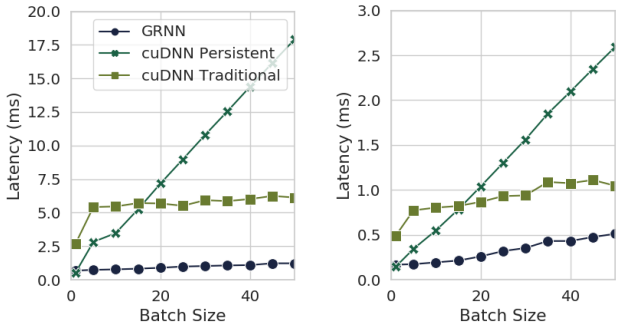
### 8.4 Single vs Double Synchronization implementations of GRU

Recall that to reduce the number of necessary synchronizations, GRNN introduces a novel tiling method that trades off one synchronization for an extra reduction. Table 3 shows that this single synchronization tiling method improves performance over the default two synchronization approach by 1.15X to 1.5X across small to medium hidden sizes. The optimization is effective because for small models, synchronization overhead dominates execution time. However, as the model size increases, the incurred overhead also increases dramatically due to extra data movement from global memory and redundant computation, while the synchronization overhead remains roughly the same. This causes the one synchronization approach to lose performance compared with the other approach for large hidden sizes like 1024 as shown in Table 3.

Since both tiling strategies provide strong performances at different regions of the optimization space, both are included as options for the GRU performance model. The marginal costs are tracked between different tiling strategies, allowing GRNN to select the appropriate tiling strategy across the entire optimization space. As demonstrated in the topline results in Table 1, the total Top-5 heuristic successfully leverages the strengths of each tiling strategy.

### 8.5 Real World Models

To more rigorously evaluate the performance of GRNN, we use two real-world models with multi-layer RNNs. The char-RNN model [23] is a character-level language model with




**(a)** Latency and scalability results for text classification.

**(b)** Latency and scalability results for CharRNN.

**Figure 15.** Performance results for real-world models.

3 LSTM cells, each with input and hidden sizes 128 and sequence length 100. The text classification model [41] is a 2-cell LSTM network with input and hidden sizes of 256 and sequence length 20. We do not run GRU-based models on cuDNN because it does not support the canonical GRU architecture.

Figures 15 shows latency and scalability in batch sizes for cuDNN's implementations and Top-5 GRNN for the two real-world models. Similar to the results on benchmarks, while the persistent cuDNN implementation yields low latency for unbatched inference, it exhibits poor scalability and across the range of batch sizes degrades dramatically in performance. In contrast, the traditional cuDNN implementation produces high latency for unbatched inference but provides high scalability, having marginal increases in latency for additional batched inputs. GRNN combines the best properties of those two implementations and provides comparable latency for unbatched inference to cuDNN Persistent while scaling at a similar rate to cuDNN Traditional.

## 9 Discussion

GRNN's techniques can be effectively applied to cell designs that differ from LSTM and GRU cells. Output-driven tiling, SM thread mapping, and the performance modeling fundamentals are fully portable across RNN cell designs. Output driven tiling and SM thread mapping are applied directly to the fused hidden state matrix multiplication, which will change dimensionally but not structurally for a new cell design. If the cell includes dependent gates, additional synchronization can be inserted between dependencies, as in the two-synchronization GRU implementation. Furthermore, the partial matrix multiplication, used in the single synchronization GRU implementation, can be inserted for cells with dependent gates to reduce the number of synchronizations by up to 50 percent, with similar performance ramifications as in GRU. Finally, the components of the performance model are agnostic to the actual cell topology, so long as the number of synchronizations, gate dimensions, and number of gates are supplied.

Parameter persistence does not benefit other neural network as effectively as RNNs, although efficient register file usage may still provide value. At a high level, convolutional neural networks and multi-layer perceptrons (MLP) do not reuse weight matrices in the course of a single inference, eliminating the primary opportunity that GRNN exploits. Furthermore, CNNs and MLPs tend to use larger matrix multiplications that do achieve high throughput in traditional GEMM kernels, again in contrast with RNNs. However, for CNNs where the input and intermediates tend to have larger footprints than the weight matrices themselves—layer fusion to avoid unnecessary reloading of the intermediates is a similar technique that uses the register file to achieve higher performance. Orthogonally, for models when persistence may not improve performance, avoiding DRAM accesses through parameter persistence may provide energy efficiency benefits. We leave this to be explored in future work.

GRNN uses only the GPU for inference, but heterogeneous solutions that utilize both the CPU and GPU may provide higher combined throughput. For datacenter RNN serving, however, heterogeneous solutions introduce overhead from bidirectional communication between the host and the GPU at each timestep. Given the throughput difference between the CPU and GPU (10X in the systems studied by this paper), the maximum performance benefit would be on the order of 10% before accounting for the aforementioned overhead. Performing CPU inference would also inhibit the ability of the host processor to batch new requests or perform embedding lookups, a common RNN layer that achieves dramatically better resource utilization on CPU than GPU. However, for Systems on Chip (SoCs) that share portions of the memory hierarchy, such as mobile processors with a unified last level cache (LLC), a heterogeneous implementation may achieve better inference performance.

## 10 Related Works

**Full-fledged deep learning systems.** Recent years have seen the rise of a variety of deep learning systems, such as TensorFlow [6], PyTorch [28], CNTK [29], Caffe [21], Theano [32], and MXNeT [8] to name a few. All those systems support inference by default, but the focus is on improved productivity in declaring deep learning models and accelerated training through distributed systems and accelerators. During training, they can leverage large batch sizes to achieve high throughput. However, serving enforces certain SLAs and hence substantially limits the maximum

batch size. As such, these full-fledged systems tend to perform suboptimally due to the lack of sufficient data parallelism [9, 40]. These systems' default GPU-based implementations for RNNs produce particularly poor performance because they fail to exploit the inherent data reuse.

**Specialized Inference engines.** TensorFlow Serving [27] is an open-sourced inference engine developed by Google to serve TensorFlow models. It supports dynamic batching and user-defined SLAs. However, it shares with TensorFlow the same set of operators to perform inference, inheriting the performance problems when serving RNN models. Clipper [13] is a recent serving system that addresses both latency and throughput by intelligently assembling models defined in different frameworks. Since it reuses operators in existing systems, it does not mitigate the low-performance issues for RNNs. BatchMaker [15] is specially designed to improve inference speed for RNNs on GPUs. Built on top of MXNeT, BatchMaker enables cell-level batching and reduces waiting time. Similar to the above mentioned systems, it also reuses the default operators implemented in a full-fledged system (in this case MXNeT). All these inference engines perform optimizations at a high-level through, for example, scheduling without tackling the implementation problem of the RNN operators themselves, which dictate the maximum performance of an individual timestep. On the contrary, the TVM [9] and XLA [3] compilers can generate high-performance inference implementations for deep learning models such as RNNs. We may implement the proposed techniques in this work in one of those compilers to support more applications.

**RNN libraries.** DeepCPU [40] is the state-of-the-art CPU-based library to serve RNN models, which is deployed in Micorosoft's production system and outperforms default TensorFlow and CNTK by more than 10X. DeepCPU's key contribution is to persist the weight data in private cache to minimize data movement. Similarly, PersistentRNN [14] leverages the persistent threads technique [17] to stash the weight data in register files. But it only supports basic RNN models, ignoring more complex yet more popular models like LSTM and GRU. cuDNN [10] implements a persistent version for LSTM and a simplified GRU model, which outperforms DeepCPU for many configurations. However, as we discussed earlier, cuDNN's scalability is unsatisfactory and often achieves low floating point throughput.

**Model compression.** A notable trend for serving machine learning models is to reduce the model size through pruning [18, 39] and quantization [24, 33]. Interestingly, such techniques even further broaden the applicability of GRNN because the reduced models can be more easily persisted in the GPU. Moreover, one may even include GRNN to estimate inference performance when, for instance, iteratively pruning the mdoel.

## 11 Conclusion

In this paper, we present a GPU-based RNN inference library named GRNN with low latency, high scalability, and efficient resource utilization. GRNN features an output-oriented tiling technique to minimize synchronization overhead, a flexible mapping technique to balance on-chip hardware resource usage, and an accurate comparative performance model to select high-performing configurations from a tremendous configuration space with negligible overhead. Experiments on various benchmark settings and two real-world models show that GRNN reduces latency by up to 94% compared with the state-of-the-art CPU implementation and improves throughput by up to 14.6X compared with state-of-the-art GPU implementations.

## 12 Acknowledgment

We thank the anonymous reviewers for their insightful comments and suggestions. We thank Dr. Peter Pietzuch for shepherding this paper. The effort of this project is funded by National Science Foundation Grant 1618912 and an NSF CAREER award.

## References

[1] Dense linear algebra on gpus. https://developer.nvidia.com/cublas. Accessed: 2018-10-1.

[2] NVIDIA CUDA. http://www.nvidia.com/cuda.

[3] The accelerated linear algebra compiler framework. https://www.tensorflow.org/performance/xla/, 2018.

[4] Nv-wavenet: Better speech synthesis using gpu-enabled wavenet inference. https://devblogs.nvidia.com/nv-wavenet-gpu-speech-synthesis/, 2018.

[5] Nvidia tensort - programmable inference accelerator. https://developer.nvidia.com/tensorrt, 2018.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[7] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax : Qos awareness and increased utilization of non-preemptive accelerators in warehouse scale computers. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.

[10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[11] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734, 2014.

[12] Jan Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 577–585, 2015.

[13] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, 2017. USENIX Association.

[14] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Y. Hannun, and Sanjeev Satheesh. Persistent rnns: Stashing recurrent weights on-chip. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 2024–2033, 2016.

[15] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency RNN inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 31:1–31:15, 2018.

[16] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, pages 6645–6649, 2013.

[17] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing*, page 14, May 2012.

[18] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015.

[19] Kim M. Hazelwood, Sarah Bird, David M. Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, pages 620–629, 2018.

[20] Robert Hecht-Nielsen. Neural networks for perception (vol. 2). chapter Theory of the Backpropagation Neural Network, pages 65–93. Harcourt Brace & Co., Orlando, FL, USA, 1992.

[21] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.

[22] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[23] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. Character-aware neural language models. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 2741–2749, 2016.

[24] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 2849–2858, 2016.

[25] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. Recurrent neural network for text classification with multi-task learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 2873–2879, 2016.

[26] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 1412–1421, 2015.

[27] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ML serving. *CoRR*, abs/1712.06139, 2017.

[28] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[29] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, 2016.

[30] Jeff A. Stuart and John D. Owens. Efficient synchronization primitives for gpus. *CoRR*, abs/1110.4623, 2011.

[31] Ming Tan, Bing Xiang, and Bowen Zhou. Lstm-based deep learning models for non-factoid answer selection. *CoRR*, abs/1511.04108, 2015.

[32] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[33] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[34] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey S. Vetter. Enabling and exploiting flexible task assignment on GPU through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*, pages 119–130, 2015.

[35] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. Flep: Enabling flexible and efficient preemption on gpus. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[36] Shucai Xiao and Wu-chun Feng. Inter-block GPU communication via fast barrier synchronization. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–12, 2010.

[37] Caiming Xiong, Victor Zhong, and Richard Socher. Dynamic coattention networks for question answering. *CoRR*, abs/1611.01604, 2016.

[38] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alexander J. Smola, and Eduard H. Hovy. Hierarchical attention networks for document classification. In *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics:*

*Human Language Technologies, San Diego California, USA, June 12-17, 2016*, pages 1480–1489, 2016.

[39] Jiecao Yu, Andrew Lukefahr, David J. Palframan, Ganesh S. Dasika, Reetuparna Das, and Scott A. Mahlke. Scalpel: Customizing DNN pruning to the underlying hardware parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 548–560, 2017.

[40] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 951–965, Boston, MA, 2018. USENIX Association.

[41] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 649–657, 2015.

[42] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. Versapipe: a versatile programming framework for pipelined computing on GPU. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, pages 587–599, 2017.