openLG: A Tunable and Efficient Open-source LSTM on GPUs

Zhaowen Shan
School of Computer Science
Sun Yat-Sen University
Guangzhou, China
shanzhw@mail2.sysu.edu.cn

Xuanteng Huang
School of Computer Science
Sun Yat-Sen University
Guangzhou, China
huangxt57@mail2.sysu.edu.cn

Zheng Zhou
School of Computer Science
Sun Yat-Sen University
Guangzhou, China
zhouzh93@mail2.sysu.edu.cn

Xianwei Zhang*
School of Computer Science
Sun Yat-Sen University
Guangzhou, China
zhangxw79@mail.sysu.edu.cn

***Abstract*—Long-Short-Term Memory (LSTM) neural networks have demonstrated exceptional proficiency in capturing both short-term and long-term dependencies within input sequences, rendering them invaluable across diverse applications, including DNA basecalling, speech recognition, reading comprehension, and scientific numerical forecasting. Despite their significance, the closed-source nature of state-of-the-art libraries, particularly the *cuDNN* LSTM inference kernel on *NVIDIA* GPUs, poses challenges in extending, optimizing, and understanding their internal workings.**

In this paper, we propose `openLG`, an open source framework to implement high performance LSTM kernel on basis of *NVIDIA* template kernel library *CUTLASS*. With flexible and transparent template configurations, developers are able to select the most suitable parameter combination for various scenarios. We formalize the LSTM computation as the time dependent and independent partitions, and then exploit efficient *CUTLASS* GEMM kernel for both parts. Evaluations conducted on NVIDIA GPUs demonstrate that `openLG` achieves up to 78% speedup on *DeepBench* benchmarks and an average speedup of 10% on realistic applications, effectively surpassing the performance of *cuDNN*.

***Index Terms*—LSTM, *CUTLASS*, GPU, Kernel Optimization**

I. Introduction

Featuring proficient capture of both short-term dependencies and proactively exploit long-term dependencies, Long-Short-Term Memory (LSTM) has exhibited exceptional performance across diverse sequence learning tasks, including but not limited to DNA basecalling [1] [2], speech recognition [3] [4] [5], reading comprehension [6] [7], and scientific domain numerical forecasting [8] [9] [10]. Besides, LSTM models account for 21% of the tensor processing unit (TPU) workloads in Google data centers [11], emphasizing the pivotal role of LSTM in computational tasks. Specifically, LSTM incorporates a recurrent connection, wherein the output from the preceding timestep serves as an input to the current timestep. By storing temporal information in a memory format, LSTMs facilitate the discernment of protracted dependencies within input sequences. This mechanism enhances the model's capacity to capture and analyze intricate dependencies spanning across temporal dimensions.

The LSTM training and inference processes are broadly accelerated on GPUs, which offer massive parallelism to facilitate matrix multiplication operations. However, a fundamental challenge arises in the application of LSTMs due to dependencies within input sequence, hindering the concurrent computation of distinct time step. LSTMs exhibit a reduced degree of parallelism compared to other neural networks [12], such as Convolutional Neural Networks (CNNs). In the domain of DNA basecalling, exemplified by the state-of-the-art basecaller *Bonito* developed by ONT, the process of basecalling for a 3 Gbps (Giga basepairs) human genome necessitates nearly 6 hours when executed on a robust server-grade GPU [13]. Notably, the LSTM inference phase contributes to almost 70% of the overall runtime. Due to the significance of LSTM in various domains and its suboptimal performance, it is crucial to devise optimization methods to enhance its operational efficiency.

The LSTM inference kernel that currently achieves state-of-the-art performance on GPUs is located within *NVIDIA* closed-source neural network library, *cuDNN*. In many cases, there is a desire to extend the LSTM inference kernel in *cuDNN*, or one may wish to customize optimizations, such as quantized inference, building upon the foundation of *cuDNN*. But this is not possible due to the lack of theoretical explanations and absence of source code for the underlying algorithm. The internal working mechanisms of the kernel, whether it is implemented in pure CUDA, PTXAS, or any other language, and whether assembly-level optimizations have been applied, remaining as undisclosed secrets. Furthermore, the *cuDNN* LSTM implementation encompasses multiple versions, and upon determination of the problem scale, it selects a suitable kernel through a table lookup method. Nevertheless, it is

* Xianwei Zhang is the corresponding author.

important to note that the *cuDNN* LSTM inference kernel may not invariably include the optimal kernel for a specific problem size.

Recently, *NVIDIA* delivers a template-based kernel library *CUTLASS* [14], aiming to bridge the gap between high level computation description and low level kernel implementation. It allows users to compose one kernel with several "micro" kernels, each is respectively responsible for data loading, thread dispatching or cooperative computing.

In order to alleviate the burden imposed by closed-source libraries while achieving performance comparable to or surpassing that of the *cuDNN* LSTM across a diverse array of input scales, we introduce `openLG` based on *CUTLASS*. This innovative approach is designed to adeptly fine-tune template settings for operators within the LSTM kernel. Employing an optimized tuning strategy, `openLG` systematically refines template configurations for LSTM kernel. During the tuning process, `openLG` uses greedy-algorithm based tuning strategy, thereby effectively constraining the search space for template settings pertinent to the LSTM kernel. Ultimately, `openLG` produces a high-performance LSTM kernel with accompanying source codes, potentially compatible with both *NVIDIA* and other vendors.

In summary, the contributions of this paper are:

- We emphasize the importance of removing closed-source impediments in the context of LSTM applications. The white-box and transparent LSTM implementation offers developers more flexibility to address performance concerns and resource constraints.
- We formalize the LSTM computation as time dependent and independent partitions. For further performance improvement, we integrate LSTM computation layout re-organization into the transparent *NVIDIA CUTLASS* implementation for the efficient computation of both parts.
- Evaluations across various LSTM inference benchmarks and real-world applications demonstrate that the proposed framework achieves the comparable performance against the manually optimized vendor proprietary libraries such as *cuDNN*.

The rest of the paper is organized as follows. Section II introduces the background. Section III elaborates the motivation and proposed design. Section IV presents the experimental methodology, and Section V analyzes experimental results. Section VI discusses related work. Section VII concludes the paper.

II. Background

A. Long Short-Term Memory

LSTM neural networks excel in capturing the local temporal characteristics of a sequence, whereas attention mechanisms are more adept at learning long-term dynamics [15]. In specific tasks, such as DNA basecalling, LSTM demonstrates superiority over transformers [2]. Therefore, it is crucial to fully comprehend LSTM structure and boosting computational efficiency.

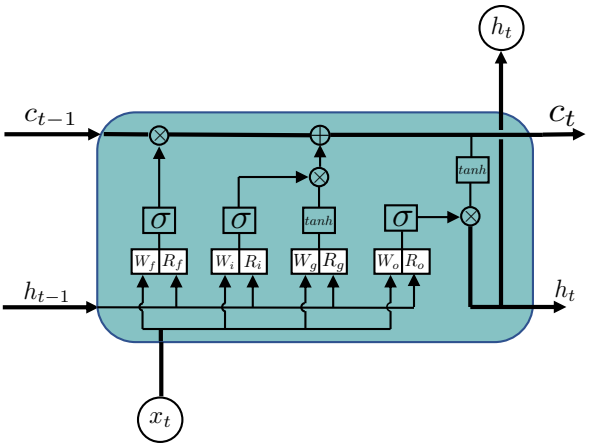


Fig. 1: A LSTM cell, which will be executed over time steps. Here, x_t, h_{t-1}, and c_{t-1} represent the input for the current time step, while h_t and c_t are the outputs for this time step.

As shown in Fig. 1, a LSTM layer consists of a single LSTM cell, which is computed repeatedly in each time step. A LSTM cell contains a cell state and four gates: the forget gate, input gate, output gate, and cell update gate. The LSTM layer iteratively processes each timestep of the input sequence, receiving the input matrix (x_t) and the hidden matrix (h_{t-1}) from the previous output at each step. These four gates collaborate to update the cell state and produce the output. Specifically, the input gate (i_t) evaluates the influence of the current input on the cell state, the forget gate (f_t) removes irrelevant information from the cell state, the cell update gate (g_t) selects relevant input information for potential updates to the cell state, and the output gate (o_t) determines the output's content and generates it. Previous studies have categorized matrix multiplications into two groups. The first group, linked to the W weight matrices, is solely dependent on the input sequence. Conversely, the second group, associated with the R weight matrices, shows recursive dependence. Additionally, the matrix multiplications with the R weight matrices in LSTM computations present two types of dependencies: intra-sequence, requiring gate activations before cell updates or output generation, and inter-sequence, where the computation at a timestep depends on the output from the previous timestep (h_{t-1}). This complexity hampers the potential for efficient parallelization. Therefore, utilizing the unique network structure of LSTM for optimal parallel acceleration is a critical issue.

B. GPU Architecture and Templated Kernels

Modern GPUs are designed with hierarchical execution units to achieve high compute and memory efficiency. Meanwhile, the CUDA software is co-designed with corresponding hardware hierarchies. Fig. 2 illustrates the three level abstractions: block, warp and thread. Heterogeneous tasks are offloaded onto GPUs with a grid of cooperative thread blocks, each containing several (up to 32) warps. And, 32 consecutive threads forms one warp, executing synchronously in lock-step

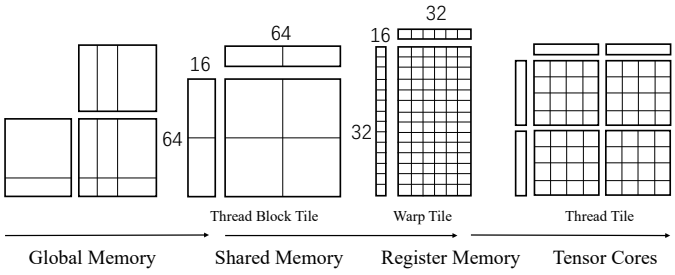


Fig. 2: *CUTLASS* design diagram with hierarchical organization. GPU threads are orchestrated in block, warp and thread levels, which constitute the tunable parameters in our templated LSTM kernel implementation.

fashion. Each block is equipped with fast on-chip scratchpad memory (i.e., shared memory) for efficient intra-block local data sharing. Similar with CPU, each thread is the minimal unit of instruction execution, with its own context, including stack, registers and branch masks. Fig. 2 shows the programming model in *CUTLASS*. As we move data from global memory to shared memory, and then to registers, the storage capacity diminishes while the speed increases.

To achieve efficient matrix multiplication, matrices in global memory need to be partitioned into tiles and then iteratively loaded into shared memory. Subsequently, shared memory data are tiled again and read into registers, following the layout constraints required by the cooperative Tensor Core instructions. In Fig. 2, the block and warp level partitions are described by two triads: `(64, 64, 16)` and `(32, 32, 16)`, where each tuple stands for the number of elements that will be loaded and computed in every iteration alongside the M, N and K dimensions. The open source implementation enable developers to produce kernel with variant configurations like tile shapes and data formats. The transparency and flexibility template setting make the generated kernel more accommodative with different LSTM problem scales, hardware generations and resource constraints.

III. Design and Implementation

A. Overview

The state-of-the-art method for LSTM inference on *NVIDIA* GPUs is found in the proprietary *cuDNN* library. This closed-source implementation poses two main challenges for users: 1). It limits user customization. For instance, users seeking faster inference speeds might consider methods like quantization for acceleration. However, deep learning compilers such as TVM, and frameworks like *PyTorch*, do not support quantized LSTM inference on GPUs. The closed nature of *cuDNN* further restricts users from modifying it for such enhancements. 2). It hinders portability. For example, although *PyTorch* now supports other AI accelerator as a backend, their deep learning library sometimes struggle to fully leverage the hardware's capabilities due to a later start and less manpower. Therefore, developing an open-source LSTM inference code based on template programming, which matches or surpasses the performance of *cuDNN*, would greatly facilitate user extensions and ease migration to other hardware devices. In this work, we propose a transparent LSTM kernel implementation with data partition based on the time dependency. For generating performant kernel, we design a greedy algorithm to efficiently search suitable parameter configurations. Moreover, we integrate `openLG` into the popular deep learning framework to provide the user-friendly interface.

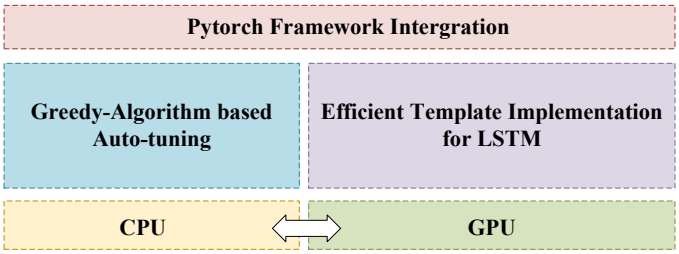


Fig. 3: Implementation overview of `openLG`, including open source template-based LSTM kernel, greedy algorithm for efficient parameter searching as well as the *PyTorch* framework integration.

B. Template Implementation of LSTM

This subsection provides a detailed introduction to the implementation details of template implementation for LSTM. In deep learning frameworks like PyTorch, the input x for LSTM is typically represented as (*sequence length*, *batch size*, *feature dimension*). The term *sequence length* refers to the number of time steps, signifying that the five equations in (1) are computed at every time step. In GPU programming, a direct method involves initiating a GPU kernel for each of the five computational formulas in (1), considering the time steps. This results in launching a total of 5 $\times$ *sequence length* GPU kernels. However, this approach is highly inefficient, as it incurs significant time overhead from the host CPU with each kernel initiation. Consequently, the optimal strategy involves amalgamating several smaller matrix multiplications into a larger one, thereby fully exploiting the parallel computation capabilities of the GPU. As described in the background, matrix multiplications in LSTM can be divided into two categories. The group involving W is time-independent, allowing for the precalculation and merging of matrix multiplications from time step t=1 to t=*sequence length*, similar to the computation of Wx shown in Fig.4(b), where x_1 to x_T are organized collectively. Moreover, in matrix multiplications involving both R and W, the four parameter matrices can be consolidated into a single expanded matrix, as shown in Fig. 4(b), for computing Wx and Rh_{t-1}. In this process, W_f, W_i, W_g, W_o are merged into a single matrix W, and R_f, R_i, R_g, R_o are unified into one matrix R. This process constitutes a reorganization of the LSTM computation layout. Thus, the final pseudocode for the template implementation for LSTM is presented in Algorithm 1. It primarily includes

a matrix multiplication with Wx and a sequence of smaller matrix multiplications within the loop using Rh_{t-1}. These matrix multiplications are executed using template programming kernels, particularly the cutlass library on the GPU, as previously stated. This implementation approach is inspired by Nvidia [16] and GRNN [17], and generally combine these concepts to develop the code.

C. Greedy-Algorithm based Tuning

The LSTM template kernel's final implementation is outlined in Algorithm 1. Before deploying the LSTM template kernel in practical applications, parameter tuning for network architectures of specific sizes is essential. This tuning involves adjusting the tile sizes at different storage hierarchy levels in *CUTLASS-GEMM*(W, x) and *CUTLASS-GEMM*(R, h_{t-1}) during GPU matrix operations. The tile sizes comprise the block tile, warp tile, and thread tile, as illustrated in Figure 2. A practical tuning method involves exploring the entire parameter space, with the outer loop iterating through the parameters of *CUTLASS-GEMM*(W, x). For each parameter set selected in this outer loop, a thorough traversal of the parameter space for *CUTLASS-GEMM*(R, h_{t-1}) is conducted. The time complexity of this nested search method is $O(n^2)$, where n is the number of possible template parameters for the GEMM kernel. Our implementation of LSTM separates the calculations of Wx and Rh, ensuring these two matrix operations do not interfere with each other. Consequently, we have optimized the aforementioned tuning algorithm. We begin with a single loop to tune the parameters of *CUTLASS-GEMM*(W, x), and after determining the optimal parameters, use another loop to adjust *CUTLASS-GEMM*(R, h_{t-1})'s parameters. This method effectively reduces the algorithm's time complexity from $O(n^2)$ to $O(2n)$.

D. Integration with PyTorch

`openLG` primarily consists of two components: the first is a template implementation for LSTM, and the second is a greedy-algorithm based tuning system. Once the network size is determined by the user, it can be input into the tuner. The tuner then automatically conducts a search and adjustment process to yield a high-performance LSTM kernel. However, such a kernel is written in CUDA code, which may not be user-friendly. We craft a Python code frontend for the generated GPU kernel, and used Python's setuptools along with *PyTorch*'s cpp_extension to bind the CUDA code with the Python code, enabling users to easily integrate it into the *PyTorch* network architecture.

IV. Evaluation Setup

This section outlines the experimental environment and the benchmark selection process. To ensure comprehensive and accurate results, tests will be conducted on the *DeepBench* benchmark, along with two real-world applications: *Bonito* and *Language Modeling*. Inference scenarios require low latency; hence, *DeepBench* is provided by Baidu with batchsizes limited to 1, 2, and 4. However, in our measurements of real-world applications, we examine a wider range of *batch size*, from 1 to 512.

A. Hardware Testbed

GPU runtimes are benchmarked using an *NVIDIA* A100 system, equipped with an AMD EPYC 7302 host processor, 32 GB of RAM, and running Ubuntu 20.04. The *NVIDIA* A100 features 432 Tensor Cores specialized for deep learning operations, complemented by 40 GB of HBM2e memory. The theoretical peak throughput of the *NVIDIA* A100 is 312 TFLOPs for Tensor Float 32 (TF32),which we used in our experiments.

TABLE I: Configurations for the computing platform.

Hardware		Software	
CPU	EPYC 7302	Operating System	Ubuntu 20.04
GPU	A100	Operator Library	*PyTorch* @1.13.1 *cuDNN* @8.5.0
Memory	32 GB	Compiler	nvcc@11.8

B. Benchmark

DeepBench, developed by Baidu, is designed to benchmark deep learning operations across a range of hardware platforms. This benchmark, widely recognized for its LSTM performance tests, has been extensively adopted in prior research [18] [19]. Therefore, this study will similarly engage in these well-established benchmark tests.

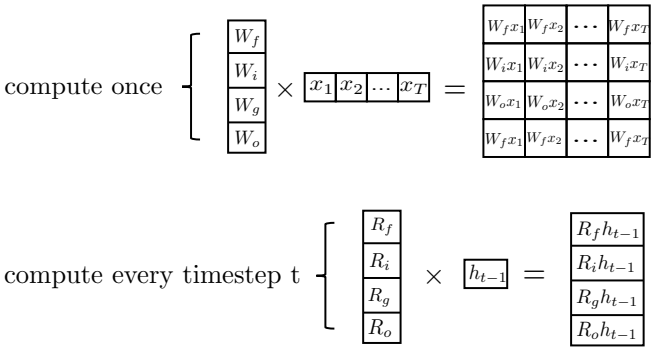
C. Real-world Applications

The *Bonito* model, which incorporates five LSTM layers each with input and hidden dimensions of 384, achieves state-of-the-art performance at translating electrical signals from nanotubes into nucleotide sequences. The LSTM's input sequence is characterized by [*sequence length*, *batch size*, *feature dimension*]. In light of the application's specific requirements, the *sequence length* for the LSTM input is fixed at 400. The *feature dimension* is uniformly set at 384. Within this framework, the *batch size* is the sole variable parameter in practical applications. Thus, the input sequence for the five LSTM layers in *Bonito* is characterized by dimensions [400, *batch size*, 384]. The *Bonito* code and test data come from this survey [2].

Language Modeling was evaluated in Lam's study [20], employing a model featuring a 1-layer 2048 unit LSTM encoder and a 1-layer 2048 unit fully connected decoder, a typical architecture in *Language Modeling* [21]. The application code is sourced from a *PyTorch* example. In this application, the LSTM's input sequence is defined by [*sequence length*, *batch size*, *feature dimension*]. The *sequence length* is set to the default of 35, the *feature dimension* is 2048, so the input sequence for LSTM layer in *Language Modeling* is characterized by dimensions [35, *batch size*, 2048].

$$
\begin{aligned}
i_t &= \sigma(W_i \cdot x_t + R_i \cdot h_{t-1} + b_i) \\
f_t &= \sigma(W_f \cdot x_t + R_f \cdot h_{t-1} + b_f) \\
o_t &= \sigma(W_o \cdot x_t + R_o \cdot h_{t-1} + b_o) \\
g_t &= tanh(W_g \cdot x_t + R_g \cdot h_{t-1} + b_g) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot tanh(c_t)
\end{aligned}
$$

(a) Computation in LSTM cell



(b) Computation organization

Fig. 4: (a) LSTM computation in one time step, where the output o_t is generated based on the hidden state h_{t-1} from previous step and current input sequence x_t. (b) Input layout for `openLG`'s LSTM kernel, where data are partitioned into the time dependent and independent parts. The time independent computation WX is conducted at once with the given input sequence, while the hidden states are produced step by step due to the time dependency.

Algorithm 1: LSTM GPU kernel Template

Input: $W, R, b, x, h_{t-1}, c_{t-1}$
Output: h_t, c_t

```
v ← CUTLASS-GEMM(W, x)
for i=1:t do
    Rh ← CUTLASS-GEMM(R, h_{t-1})
    h_t, c_t ← activation&pointwise kernels(
            Wx, Rh_{t-1}, h_{t-1}, c_{t-1}, b)
end
```

V. Results Analysis

First, we test the performance and resource utilization differences of the kernel identified by the greedy-algorithm based tuning, compared to the worst-performing kernel and kernel with randomly selected parameters. Second, we assess the performance of the LSTM GPU kernel produced by `openLG` in isolation, to confirm that our LSTM GPU kernel achieve comparable or superior speed to *cuDNN*, a hand-optimized private deep learning library. Third, for a thorough evaluation of `openLG`'s performance, we utilize two real-world models equipped with multi-layer LSTMs, demonstrating `openLG`'s end-to-end acceleration capabilities.

A. Effectiveness of Template Tuning Method

In this section, we configure a range of *CUTLASS* parameters in the LSTM template kernel and evaluate the LSTM kernel's performance under these varied configurations. This aims to highlight the significance and impact of tuning *CUTLASS* parameters. We conduct tests on *DeepBench*, iterating through a series of selectable *CUTLASS* template parameters. We identify the kernels with the best and worst performance and include a kernel corresponding to a randomly chosen parameter. The results are presented in the bar chart shown in Fig. 5.

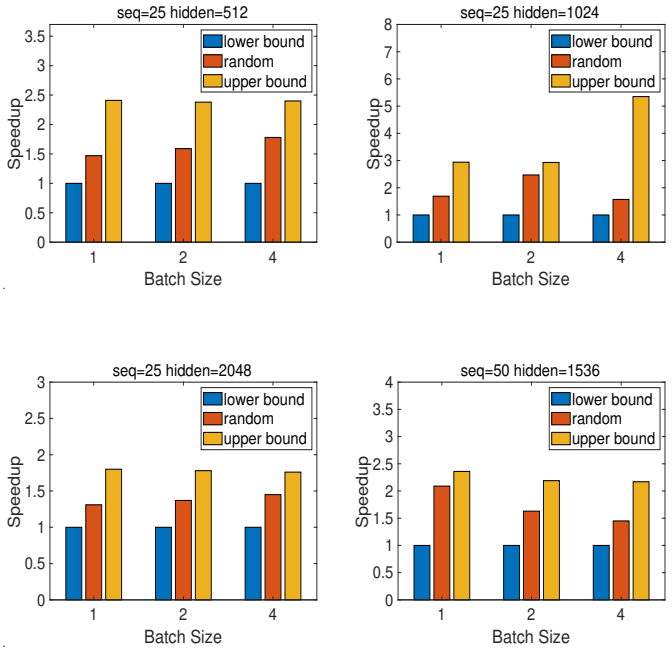


Fig. 5: Test results of greedy-algorithm based tuning on *DeepBench* demonstrate a speed comparison among the best-performing kernel identified by `openLG`, a kernel with randomly chosen parameters, and the worst-performing kernel.

With a *sequence length* of 25 and a hidden size of 512, for batchsizes of 1, 2, and 4, the best-performing kernel identified through greedy-algorithm based tuning exhibit speeds 2.41, 2.38, and 2.4 times faster than the worst-performing kernel, and 1.64, 1.49, and 1.35 times faster than kernel with randomly selected parameters, respectively. With a *sequence length* of 25 and a hidden size of 1024, for batchsizes of

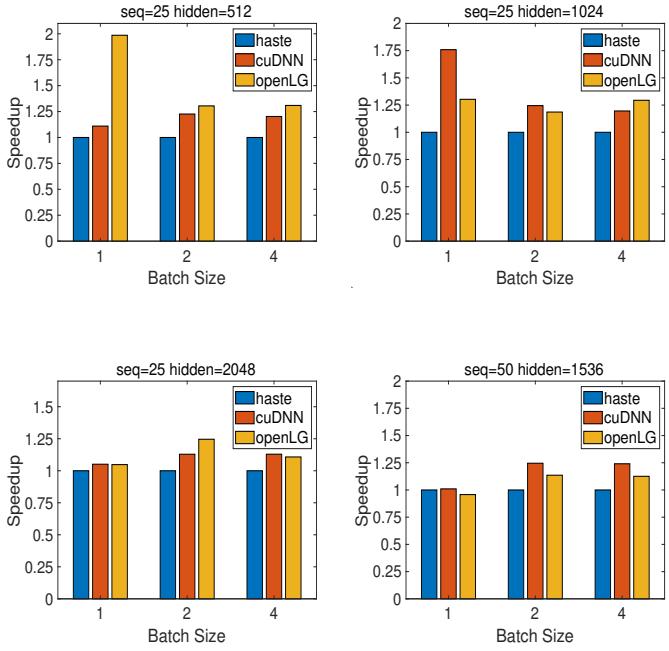


Fig. 6: Performance comparison of `openLG`, *cuDNN*, and *haste* [22] on *DeepBench*.

1, 2, and 4, the best-performing kernel identified through greedy-algorithm based tuning exhibit speeds 2.94, 2.93, and 5.35 times faster than the worst-performing kernel, and 1.73, 1.19, and 3.4 times faster than kernel with randomly selected parameters, respectively. With a *sequence length* of 25 and a hidden size of 2048, for batchsizes of 1, 2, and 4, the best-performing kernel identified through greedy-algorithm based tuning exhibit speeds 1.8, 1.78, and 1.76 times faster than the worst-performing kernel, and 1.37, 1.3, and 1.22 times faster than kernel with randomly selected parameters, respectively. With a *sequence length* of 50 and a hidden size of 1536, for batchsizes of 1, 2, and 4, the best-performing kernel identified through greedy-algorithm based tuning exhibit speeds 2.36, 2.19, and 2.17 times faster than the worst-performing kernel, and 1.12, 1.34, and 1.49 times faster than kernel with randomly selected parameters, respectively. Summarizing the final test results, the best-performing kernel identified through greedy-algorithm based tuning is, on average, 2.54 times faster than the worst-performing kernel, and 1.53 times faster than a kernel with randomly chosen parameters. This underscores the effectiveness of the tuning method.

B. Benchmark Test

The experimental results of *DeepBench* are illustrated in Fig. 6. This figure compares the acceleration ratios, using the open-source LSTM inference library *haste* as the baseline, against the LSTM kernel generated by `openLG` and those from *cuDNN*.

With a *sequence length* of 25 and a hidden size of 512, `openLG`'s kernel demonstrates enhanced performance. Specifically, at batchsizes of 1, 2, and 4, `openLG`'s kernel outperforms *haste* by factors of 1.98, 1.30, and 1.30. Meanwhile, *cuDNN*'s kernel outpaces *haste* by factors of 1.10, 1.22, and 1.20 for the same batchsizes. Respectively, `openLG`'s kernel exceeds *cuDNN*'s by factors of 1.78, 1.07, and 1.09. In summary, under the stipulated sequence and hidden size parameters, `openLG`'s kernel, on average, exhibits a 1.53-fold acceleration over *haste*, while *cuDNN* averages a 1.18-fold acceleration over *haste*. Additionally, `openLG` demonstrates a 1.31-fold edge over *cuDNN*'s kernel.

With a *sequence length* of 25 and a hidden size of 1024, `openLG`'s kernel demonstrates enhanced performance. Specifically, at batchsizes of 1, 2, and 4, `openLG`'s kernel outperforms *haste* by factors of 1.30, 1.19, and 1.29. Meanwhile, *cuDNN*'s kernel outpaces *haste* by factors of 1.75, 1.55, and 1.19 for the same batchsizes. Respectively, `openLG`'s kernel exceeds *cuDNN*'s by factors of 0.74, 1.02, and 1.08. In summary, under the stipulated sequence and hidden size parameters, `openLG`'s kernel, on average, exhibits a 1.26-fold acceleration over *haste*, while *cuDNN* averages a 1.37-fold acceleration over *haste*. Additionally, `openLG` demonstrates a 0.95-fold edge over *cuDNN*'s kernel.

With a *sequence length* of 25 and a hidden size of 2048, `openLG`'s kernel demonstrates enhanced performance. Specifically, at batchsizes of 1, 2, and 4, `openLG`'s kernel outperforms *haste* by factors of 1.05, 1.16, and 1.11. Meanwhile, *cuDNN*'s kernel outpaces *haste* by factors of 1.05, 1.13, and 1.13 for the same batchsizes. Respectively, `openLG`'s kernel exceeds *cuDNN*'s by factors of 0.99, 1.02, and 0.98. In summary, under the stipulated sequence and hidden size parameters, `openLG`'s kernel, on average, exhibits a 1.10-fold acceleration over *haste*, while *cuDNN* averages a 1.10-fold acceleration over *haste*. Additionally, `openLG` demonstrates a 1.0-fold edge over *cuDNN*'s kernel.

With a *sequence length* of 50 and a hidden size of 1536, `openLG`'s kernel demonstrates enhanced performance. Specifically, at batchsizes of 1, 2, and 4, `openLG`'s kernel outperforms *haste* by factors of 0.96, 1.14, and 1.13. Meanwhile, *cuDNN*'s kernel outpaces *haste* by factors of 1.01, 1.16, and 1.15 for the same batchsizes. Respectively, `openLG`'s kernel exceeds *cuDNN*'s by factors of 0.95, 0.98, and 0.98. In summary, under the stipulated sequence and hidden size parameters, `openLG`'s kernel, on average, exhibits a 1.07-fold acceleration over *haste*, while *cuDNN* averages a 1.10-fold acceleration over *haste*. Additionally, `openLG` demonstrates a 0.97-fold edge over *cuDNN*'s kernel.

Summarizing the results on *DeepBench*: `openLG`'s kernel outpaces *haste* by an average of 1.24 times, while *cuDNN*'s kernel is 1.19 times faster than *haste*. Comparatively, `openLG`'s kernel achieves up to 1.3 times the speed of *cuDNN*'s kernel, with the lowest at 0.92 times *cuDNN*'s speed. On average, `openLG`'s kernel accelerates 1.04 times faster than *cuDNN*'s, indicating that `openLG` matches or even surpasses *cuDNN* in speed according to the *DeepBench* results.

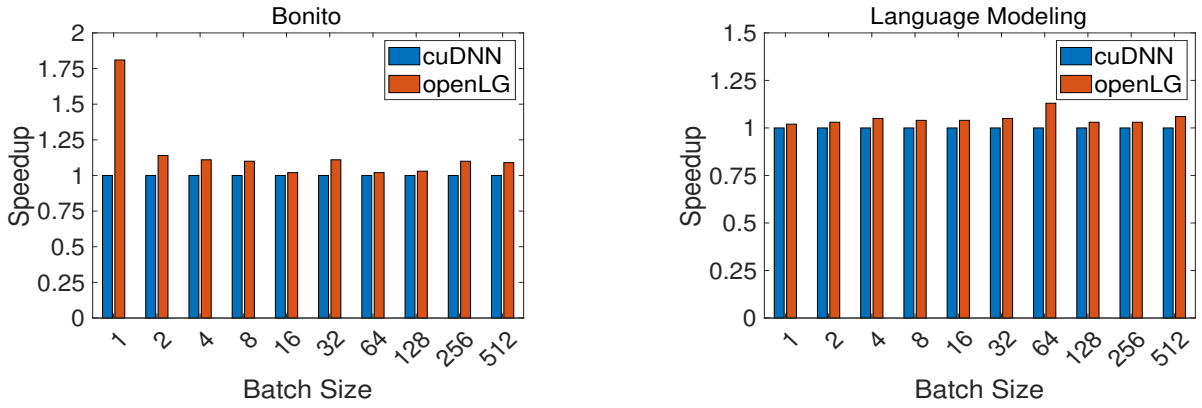


Fig. 7: Comparative test results of `openLG` and *cuDNN* in *Bonito* and *Language Modeling*.

C. Real World Models

The experimental results for real-world applications are presented in Fig. 7. Given that our *DeepBench* tests have already established the inferior performance of *haste* compared to both `openLG` and *cuDNN*, this section uses *cuDNN* as the baseline to primarily compare the performance between `openLG` and *cuDNN*.

Figure 7 shows that in the real-world application *Bonito*, the LSTM kernel generated by `openLG` consistently outperforms *cuDNN* across a series of batches. The performance of `openLG`'s kernel ranges from being 2% faster than *cuDNN* at its least to 81% faster at its best, with an average improvement of 15%. In another real-world application, *Language Modeling*, `openLG`'s LSTM kernel also surpasses *cuDNN* in performance across various batches, as shown in Figure 7. Here, `openLG`'s kernel demonstrates a speed increase over *cuDNN* ranging from 2% at its minimum to 12% at its maximum, with an average improvement of 5%. Across both applications, `openLG`'s kernel exhibits an average speed improvement of 10% over *cuDNN*, achieving and occasionally surpassing *cuDNN*'s performance level.

VI. Related Work

LSTM optimization. Zhu [23] introduces a novel pruning algorithm aimed at improving workload balance and reducing decoding overhead in sparse LSTM neural networks. However, this method is specifically effective for sparse LSTM networks and lacks general applicability. Gao [24] introduces cellular batching, a technique that dynamically assembles batches at the granularity of an LSTM cell, rather than at the level of the entire dataflow graph. This method is advantageous for efficiently handling dynamic dataflow graphs and inputs of varied lengths. This approach can be synergistically combined with kernel optimization, owing to their orthogonal relationship. Yin [25] explores the use of hardware properties to enhance model compactness, accuracy, and execution efficiency. However, integrating this system into existing machine learning frameworks poses a challenge. In another study, Zhu [26] presents a method to accelerate LSTM training by utilizing sparsity during the backward propagation phase, although this may compromise accuracy. The method proposed in this paper, `openLG`, demonstrates universality and ease of integration with PyTorch, effectively bridging the gaps identified in prior research.

Kernel optimization using *CUTLASS*. Prior research has focused on accelerating high-performance computing workloads using *CUTLASS*. ByteTransformer [27] employs *CUTLASS*'s epilogue to fuse GEMM operations, effectively reducing memory latency and enhancing the performance of single-layer BERT transformers. MegaBlocks [28] extends *CUTLASS* to support block-sparse matrices, thus facilitating efficient training of Mixture of Experts (MoEs) models. Bolt [29] leverages *CUTLASS*'s epilogue fusion feature to combine a GEMM/Conv kernel with subsequent operations into a single operator, thereby enhancing convolutional neural network performance. `openLG` expands the application range of *CUTLASS* by accelerating LSTM on GPUs, an area not directly covered by existing research.

VII. Conclusion

In this work, we propose `openLG`, a tunable template framework to overcome the opaque of vendor-proprietary libraries in developing LSTM GPU kernel. `openLG` formalizes the LSTM computation as the time dependent and independent parts, and generates performant kernel for both stages. We also leverage a greedy algorithm to search and select the most suitable template parameter combination for LSTM kernel. Evaluations conducted on both benchmarks and real-world applications demonstrate that `openLG` outperforms *haste* and *cuDNN* with open source implementation.

Acknowledgment

We appreciate the anonymous reviewers for their constructive feedback. This research was supported by the National Natural Science Foundation of China-#62332021, and the Major Program of Guangdong Basic and Applied Research-#2019B030302002, Key-Area Research and Development Program of Guangdong Province (No.2021B0101190003) and the Fundamental Research Funds for the Central Universities, Sun Yat-sen University (23xkjc016). Thanks ChatGPT for the

guidance on phrasing and word choice throughout the paper writing process.

REFERENCES

[1] O. N. PLC., "Dorado," https://github.com/nanoporetech/dorado, 2023.

[2] M. Pags-Gallego and J. de Ridder, "Comprehensive benchmark and architectural analysis of deep learning models for Nanopore sequencing basecalling," Bioinformatics, preprint, May 2022. [Online]. Available: http://biorxiv.org/lookup/doi/10.1101/2022.05.17.492272

[3] S. Basak, H. Agrawal, S. Jena, S. Gite, M. Bachute, B. Pradhan, and M. Assiri, "Challenges and Limitations in Speech Recognition Technology: A Critical Review of Speech Signal Processing Algorithms, Tools and Systems," *Computer Modeling in Engineering & Sciences*, vol. 135, no. 2, pp. 1053–1089, 2023. [Online]. Available: https://www.techscience.com/CMES/v135n2/50181

[4] M. W. Y. Lam, X. Chen, S. Hu, J. Yu, X. Liu, and H. Meng, "Gaussian Process Lstm Recurrent Neural Network Language Models for Speech Recognition," in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Brighton, United Kingdom: IEEE, May 2019, pp. 7235–7239. [Online]. Available: https://ieeexplore.ieee.org/document/8683660/

[5] D. S. Park, W. Chan, Y. Zhang, C.-C. Chiu, B. Zoph, E. D. Cubuk, and Q. V. Le, "SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognition," in *Interspeech 2019*, Sep. 2019, pp. 2613–2617, arXiv:1904.08779 [cs, eess, stat]. [Online]. Available: http://arxiv.org/abs/1904.08779

[6] Y. Lakretz, G. Kruszewski, T. Desbordes, D. Hupkes, S. Dehaene, and M. Baroni, "The emergence of number and syntax units in LSTM language models," Apr. 2019, arXiv:1903.07435 [cs]. [Online]. Available: http://arxiv.org/abs/1903.07435

[7] S. Wang and J. Jiang, "An LSTM model for cloze-style machine comprehension."

[8] Y. Xu, C. Hu, Q. Wu, S. Jian, Z. Li, Y. Chen, G. Zhang, Z. Zhang, and S. Wang, "Research on particle swarm optimization in LSTM neural networks for rainfall-runoff simulation," *Journal of Hydrology*, vol. 608, p. 127553, May 2022. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0022169422001287

[9] W. Zha, Y. Liu, Y. Wan, R. Luo, D. Li, S. Yang, and Y. Xu, "Forecasting monthly gas field production based on the CNN-LSTM model," *Energy*, vol. 260, p. 124889, Dec. 2022. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0360544222017923

[10] M. Akhoondzadeh, "Advances in Seismo-LAI anomalies detection within Google Earth Engine (GEE) cloud platform," *Advances in Space Research*, vol. 69, no. 12, pp. 4351–4357, Jun. 2022. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0273117722002411

[11] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, Jun. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3360307

[12] S. Mittal and S. Umesh, "A survey On hardware accelerators and optimization techniques for RNNs," *Journal of Systems Architecture*, vol. 112, p. 101839, Jan. 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1383762120301314

[13] G. Singh, M. Alser, A. Khodamoradi, K. Denolf, C. Firtina, M. B. Cavlak, H. Corporaal, and O. Mutlu, "A Framework for Designing Efficient Deep Learning-Based Genomic Basecallers," Genomics, preprint, Nov. 2022. [Online]. Available: http://biorxiv.org/lookup/doi/10.1101/2022.11.20.517297

[14] V. Thakkar, P. Ramani, C. Cecka, A. Shivam, H. Lu, E. Yan, J. Kosaian, M. Hoemmen, H. Wu, A. Kerr, M. Nicely, D. Merrill, D. Blasig, F. Qiao, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, and M. Gupta, "CUTLASS," Jan. 2023. [Online]. Available: https://github.com/NVIDIA/cutlass

[15] B. Lim, S. . Ark, N. Loeff, and T. Pfister, "Temporal Fusion Transformers for interpretable multi-horizon time series forecasting," *International Journal of Forecasting*, vol. 37, no. 4, pp. 1748–1764, Oct. 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0169207021000637

[16] J. Appleyard, T. Kocisky, and P. Blunsom, "Optimizing Performance of Recurrent Neural Networks on GPUs," Apr. 2016, arXiv:1604.01946 [cs]. [Online]. Available: http://arxiv.org/abs/1604.01946

[17] C. Holmes, D. Mawhirter, Y. He, F. Yan, and B. Wu, "GRNN: Low-Latency and Scalable RNN Inference on GPUs," in *Proceedings of the Fourteenth EuroSys Conference 2019*. Dresden Germany: ACM, Mar. 2019, pp. 1–16. [Online]. Available: https://dl.acm.org/doi/10.1145/3302424.3303949

[18] X. Zhang, C. Xie, J. Wang, W. Zhang, and X. Fu, "Towards Memory Friendly Long-Short Term Memory Networks (LSTMs) on Mobile GPUs," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Fukuoka: IEEE, Oct. 2018, pp. 162–174. [Online]. Available: https://ieeexplore.ieee.org/document/8574539/

[19] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar, R. Krishnamurthy, S. Gribok, B. Pasca, M. Langhammer, D. Marr, and A. Dasu, "Why Compete When You Can Work Together: FPGA-ASIC Integration for Persistent RNNs," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. San Diego, CA, USA: IEEE, Apr. 2019, pp. 199–207. [Online]. Available: https://ieeexplore.ieee.org/document/8735536/

[20] M. Lam, Z. Yedidia, C. R. Banbury, and V. J. Reddi, "Precision Batching: Bitserial Decomposition for Efficient Neural Network Inference on GPUs," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Atlanta, GA, USA: IEEE, Sep. 2021, pp. 129–141. [Online]. Available: https://ieeexplore.ieee.org/document/9563027/

[21] G. Melis, C. Dyer, and P. Blunsom, "On the State Of the Art of Evaluation in Neural Language Models," 2018.

[22] S. Nanavati, "Haste: a fast, simple, and open rnn library," https://github.com/lmnt-com/haste/, Jan 2020.

[23] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse Tensor Core: Algorithm and Hardware Co-Design for Vector-wise Sparse Neural Networks on Modern GPUs," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus OH USA: ACM, Oct. 2019, pp. 359–371. [Online]. Available: https://dl.acm.org/doi/10.1145/3352460.3358269

[24] P. Gao, L. Yu, Y. Wu, and J. Li, "Low latency RNN inference with cellular batching," in *Proceedings of the Thirteenth EuroSys Conference*. Porto Portugal: ACM, Apr. 2018, pp. 1–15. [Online]. Available: https://dl.acm.org/doi/10.1145/3190508.3190541

[25] H. Yin, G. Chen, Y. Li, S. Che, W. Zhang, and N. K. Jha, "Hardware-Guided Symbiotic Training for Compact, Accurate, yet Execution-Efficient LSTM," Jan. 2019, arXiv:1901.10997 [cs]. [Online]. Available: http://arxiv.org/abs/1901.10997

[26] M. Zhu, J. Clemons, J. Pool, M. Rhu, S. W. Keckler, and Y. Xie, "Structurally Sparsified Backward Propagation for Faster Long Short-Term Memory Training," Jun. 2018, arXiv:1806.00512 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1806.00512

[27] Y. Zhai, C. Jiang, L. Wang, X. Jia, S. Zhang, Z. Chen, X. Liu, and Y. Zhu, "ByteTransformer: A High-Performance Transformer Boosted for Variable-Length Inputs," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. St. Petersburg, FL, USA: IEEE, May 2023, pp. 344–355. [Online]. Available: https://ieeexplore.ieee.org/document/10177488/

[28] T. Gale, D. Narayanan, C. Young, and M. Zaharia, "MegaBlocks: Efficient Sparse Training with Mixture-of-Experts."

[29] J. Xing, L. Wang, S. Zhang, J. Chen, A. Chen, and Y. Zhu, "Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance," Oct. 2021, arXiv:2110.15238 [cs]. [Online]. Available: http://arxiv.org/abs/2110.15238