Improve Language Model Serving Efficiency with Fine-grained and Stateful Scheduling

by

Lingfan Yu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
May, 2024

Professor Jinyang Li

© Lingfan Yu

All Rights Reserved, 2024

Dedication

To my parents

Acknowledgments

First of all, I am grateful to my advisor, Prof. Jinyang Li, for her years of continuous guidance and support throughout my Ph.D. study. I am grateful for her insightful research suggestions. I have learned a lot from her about the proper way to think and carry out research.

I would like to thank my dissertation committee members Professor Aurojit Panda, Anirudh Sivaraman, Lakshminarayanan Subramanian, and Sai Qian Zhang for their time and effort in reviewing my dissertation. The constructive feedback they provided at my dissertation defense was enlightening and helped me further expend the scope of my dissertation.

I am also very thankful to Pin Gao, who was a visiting scholar from Tsinghua University and the coauthor of BatchMaker project. It was Pin that first introduced me to the world of Machine Learning System research. His intelligence and skills were crucial to the success of BatchMaker. It was a very enjoyable experience working with him.

I want to thank Minjie Wang and the entire Deep Graph Library team for having me on the DGL project. As a founding member of DGL, I learned a lot about open-source project development, management, and community collaboration. I also learned important skills like programming GPU code while developing DGL.

I would like to thank Prof. Michael Walfish and his student Cheng Tan (now a Professor at Northeastern University) for their guidance in the first year of my Ph.D. Although that was a tough year that I barely survived, working with Mike and Cheng helped me transition from a task-driven undergraduate caring only about getting things done as instructed to a PhD student

constantly reminding himself to think about the reasoning behind any task and the high-level picture of a research project.

I also want to thank all my colleagues and labmates including Chien-Chin Huang, Jinkun Lin, Tao Wang, Anqi Zhang, Ding Ding, Zhanghan Wang, and many others, for their support, encouragement, and inspiring discussions.

Abstract

The world has witnessed the remarkable success of large language models (LLMs), led by the fast-growing popularity of ChatGPT. However, it is challenging to serve these language models and deliver both high throughput and low latency due to the iterative nature of language models. This thesis identifies two key issues impacting the performance of existing systems: (1) Coarse-grained batching at the request level results in wasteful computation for requests with variable input and output lengths; (2) The lack of stateful context management results in duplicate computation for applications that engage in multi-turn interactions with the LLM model. Two systems, *BatchMaker* and *Pensieve*, are then presented to address these issues.

BatchMaker proposes a technique called cellular batching to improve the throughput and latency of language model inference. Existing systems use batch execution of the dataflow graphs of a fixed set of requests. By contrast, *BatchMaker* makes finer-grained batching decisions at each token processing step, and dynamically assembles a batch for execution as requests join and leave the system.

Pensieve is a system optimized for multi-turn conversation LLM serving. It maintains the conversation state across requests from the same conversation by caching previously processed history to avoid duplicate processing. *Pensieve*'s multi-tier caching strategy utilizes both GPU and CPU memory to store and retrieve cached data efficiently. *Pensieve* also generalizes the recent PagedAttention kernel to support attention between multiple input tokens whose KV cache is spread over non-contiguous GPU memory.

Experiments on various workloads show that *BatchMaker* improves throughput by 25-80% while reducing latency by 18-90% latency, and *Pensieve* improves 33-100% throughput and reduces latency by 40-77%.

Table of Contents

Dedication **iii**

Acknowledgments **iv**

Abstract **vi**

List of Figures **xii**

List of Tables **xiv**

1 Introduction **1**

1.1 Evolution of Language Models 2

1.2 Importance of Model Serving 3

1.3 Challenges in Serving Language Models 4

1.3.1 Coarse-grained Batching 4

1.3.2 Lack of Context State Management 5

1.4 Contributions 6

1.4.1 Batching at finer token granularity 6

1.4.2 Stateful Conversation Context Management 8

2 BatchMaker **9**

2.1 Overview 9
2.2 Background 10
2.2.1 A primer on recurrent neural networks 10
2.2.2 Training vs. inference, and the importance of batching 12
2.2.3 Existing solutions for batching RNNs 14
2.3 Our Approach: Cellular batching 16
2.3.1 Batching at the granularity of cells 17
2.3.2 Joining and leaving the ongoing execution 17
2.4 System Design 19
2.4.1 User Interface 19
2.4.2 Software Architecture 19
2.4.3 Batching and Scheduling 22
2.4.4 An example of TreeLSTM scheduling 25
2.5 GPU Optimization 26
2.6 Implementation 28
2.7 Evaluation 28
2.7.1 Experimental Setup 29
2.7.2 Application Performance: LSTM 31
2.7.3 Reasons for the performance gain 34
2.7.4 Application Performance: Sequence to Sequence 39
2.7.5 Application Performance: TreeLSTM 42
2.8 Related Work 45
2.9 Applicability 49
2.10 Summary 49

3 Pensieve 51
3.1 Overview 51
3.2 Background 52
3.2.1 LLM and the Attention Mechanism 52
3.2.2 How LLM is Served 53
3.3 Motivation and Challenges 55
3.3.1 Motivation 55
3.3.2 Challenges 56
3.4 System Design 59
3.4.1 System Overview 60
3.4.2 A unified batch scheduler 61
3.4.3 KV cache management 62
3.4.4 Multi-token attention for non-contiguous cache 67
3.5 Implementation 69
3.5.1 Optimization: Prioritize data retrieving over eviction 70
3.6 Evaluation 70
3.6.1 Experimental Setup 70
3.6.2 End-to-end Pensieve Performance 74
3.6.3 Eviction Policy 76
3.6.4 Conversation with More Turns 78
3.6.5 User Reaction Time 80
3.6.6 Efficiency of Multi-token attention Kernel 81
3.6.7 Unified Scheduling 82
3.7 Related Works 83
3.7.1 LLM inference and serving systems 83
3.7.2 Non-LLM specific DNN serving systems 84

3.7.3 Techniques addressing the GPU memory limit 84
3.7.4 Techniques that speed up LLM inference 86
3.8 Summary 87

4 Conclusion **88**
4.1 Future Work 88
4.1.1 Unbalanced workload when used with pipeline parallelism 89
4.1.2 Multi-host load balancing 89
4.1.3 Support for various storage backends 90
4.1.4 Conversations with similar texts 90

Bibliography **92**

List of Figures

2.1 An unfolded chain-structured RNN 10
2.2 An unfolded tree-structured RNN. 10
2.3 Latency vs. throughput for computing a single step of LSTM 12
2.4 Existing systems perform graph batching 14
2.5 The timeline of graph batching and Cellular Batching 16
2.6 System architecture of BatchMaker 20
2.7 LSTM performance on the WMT-15 Europarl dataset using 1 GPU 32
2.8 LSTM performance using MXNet with different bucket widths 34
2.9 Request queuing and computation time for LSTM on WMT-15 Europarl dataset . 35
2.10 CDF of the sequence length in the WMT-15 Europarl dataset 37
2.11 Performance under different sequence length variations 38
2.12 Seq2Seq with feed previous decoder 39
2.13 Performance of Seq2Seq on the WMT-15 Europarl German-to-English dataset . . 41
2.14 TreeLSTM performance on the TreeBank dataset with a maximum batch size 64 . 43
2.15 TreeLSTM performance on a synthetic dataset 45

3.1 Inference of Transformer-based Large Language Models 52
3.2 Existing serving systems send a cumulative history 55
3.3 Execution time for prompt prefill and generations 56

3.4 Normalized attention operation cost vs context size 57
3.5 Layout of a request's KV-tokens. 57
3.6 Multi-token attention with Non-contiguous Context Cache 58
3.7 Overview of Pensieve 60
3.8 Recompute dropped tokens and prefill new prompt tokens. 65
3.9 single-token attention vs multi-token attention 67
3.10 Distribution of conversation turns in ShareGPT dataset. 72
3.11 CDF of conversation's total context size in ShareGPT dataset. 72
3.12 Normalized latency vs throughput for OPT-13B and Llama 2-13B on 1 GPU 75
3.13 Normalized latency vs throughput for OPT-66B and Llama 2-70B on 4 GPUs . . . 77
3.14 Effect of different eviction policies 78
3.15 Pensieve performance on ShareGPT dataset with at least 5 turns. 79
3.16 Pensieve performance with different average user reaction time 80
3.17 Execution time of Pensieve's multi-token attention kernel 81
3.18 Normalized latency vs throughput for different scheduler 83

List of Tables

3.1 OPT and Llama 2 model configurations 71
3.2 Dataset staticstics 71

1 Introduction

In the past decade, Deep Neural Networks (DNNs) have achieved remarkable success in many important areas such as visual recognition, natural language processing, recommendation, etc. Among the various DNN models, Language Models based on Transformer [101] are receiving widespread attention due to the incredible success of ChatGPT [76, 107], which has led the world to believe that we have made a huge step towards Artificial General Intelligence (AGI) [41, 43]. Although there is plenty of debate [11, 17, 42, 65] on whether we are close to AGI or not, it is abundantly clear that we are on the cusp of a revolution that will fundamentally change the way we obtain and process knowledge.

Even though training language models is computationally expensive, costing millions of GPU hours [56, 106], the real elephant in the room is the cost to serve them. Once trained, a model can be deployed at a large scale with the same fixed weights and used repeatedly to serve many requests for an extended period. It is estimated that it cost $3.2M to train ChatGPT on 10,000 graphics cards [95] and $700K per day [79] to operate and serve users worldwide. As a result, serving costs overshadow the cost of training over time. Meanwhile, low service response time improves user experience, especially for real-time applications like chatbots. Therefore, developing fast-serving systems that can process live requests with low latency and high throughput is crucial.

This thesis primarily focuses on the serving side of language models. We identify issues with current solutions when applied directly to language model serving, and seek to develop new

strategies and techniques to exploit features of language models and the hardware stack. We design and implement two systems, *BatchMaker* and *Pensieve*, and demonstrate their effectiveness in improving the efficiency of serving Language Models.

1.1 Evolution of Language Models

Learning-based language models have rapidly matured from experimental research to real-world deployments. It started when Recurrent Neural Networks (RNNs) were introduced to model sequence data by recursively processing each token in the sequence while maintaining an aggregated hidden representation of the sequence. Example uses of RNNs include speech recognition [6, 44], machine translation [7, 108], image captioning [102], question answering [94, 110] and video to text [37]. The attention mechanism, which nowadays is at the core of all language models, is also used as an integral part to connect encoder and decoder inside RNN Models.

In 2017, the famous Transformer model [101] by Google proposed an architecture fully based on the attention module for sequence processing. The Transformer model quickly became the foundation of many state-of-the-art models, including BERT [35], GPT [83], etc., thanks to its ability to model long-range dependencies and associate different parts with learnable importance weights, and to be executed in a massively parallel fashion, making it possible to train on a much larger dataset within a reasonable time.

Recently, the success of so-called Large Language Models (LLMs) like GPT-3 [10] and ChatGPT [76] have further demonstrated the power and potential of language models. An LLM is a generative language model based on the decoder part of Transformer architecture and has a particularly large model size (e.g. 10s or 100s of billions of parameters). It is an autoregressive model that iteratively predicts the next output token based on the current context which includes the sequence of input prompt tokens followed by output tokens generated in the previous iterations. The most popular use of LLMs is for chatbots, with applications like ChatGPT demonstrating an

astounding capability of chatting and answering questions like real humans. Other LLM-backed applications are also being applied to automate a variety of tasks, such as writing ads, responding to emails, doing literature reviews, etc. As LLM continues its explosive growth, it is imperative to develop fast and efficient LLM serving systems.

1.2 Importance of Model Serving

The typical lifecycle of a deep neural network (DNN) deployment consists of two phases. In the *training* phase, a specific model is chosen after many design iterations and its parameter weights are computed based on a training dataset. And in this phase, what matters is the system throughput, i.e. how fast we can get the model to iterate through the dataset and tune the parameters to convergence. In the *inference* phase, the pre-trained model is used to process live application requests using the computed weights. Unlike training, DNN inference places much more emphasis on low latency in addition to good throughput. As applications often desire real-time response, inference latency has a big impact on the user experience. As a DNN model matures, it is the inference phase that consumes the most computing resources and provides the most bang for the buck for performance optimization.

The popular and successful language models face the biggest performance challenge since they are designed to model variable length inputs. The dynamic nature of the language inputs makes execution scheduling less straightforward. With the increasingly large model sizes and the need to store hidden embeddings of all tokens of the context required by the attention mechanism, the memory footprint has dramatically increased. These new challenges must be addressed for efficient language model serving.

1.3 Challenges in Serving Language Models

Existing frameworks and systems are not well-suited for the properties of Language Models. Here, we discuss the challenges that arise when serving Language Models.

1.3.1 Coarse-grained Batching

The biggest performance booster for executing any Deep Neural Network model is batching, which performs the same computation for multiple inputs in a parallel manner and is extremely efficient on modern GPUs. This powerful batching technique requires that computation for different inputs be identical, which is usually the case for models like Convolutional Neural Networks (CNNs) that take in fixed-size images as its input.

However, language model, or more generally a sequence model, differs from other popular DNN architectures in that it is designed to represent a *iterative* or *recursive* computation. The input to a language model is a sequence that can vary in length or even be organized in a tree structure like TreeLSTM [93]. In the encoder setup where the model tries to produce a hidden representation for the sequence, the input length or structure is known but can vary from one input to another. In the decoder setup where the model iteratively generates tokens one at a time based on the context, the output length is unknown.

Therefore, when expressing computation in a dataflow-based deep learning system, the resulting "unfolded" dataflow graph is not fixed but varies depending on each input. This dynamic nature of Language Models makes batching requests for language models challenging as the underlying structures for each input are different.

Existing systems have focused on improving training throughput. As such, they batch at the granularity of unfolded dataflow graphs, which we refer to as *graph batching*. Graph batching collects a batch of inputs, combines their dataflow graphs into a single graph whose operators

represent batched execution of corresponding operators in the original graphs, and submits the combined graph to the backend for execution. The most common form of graph batching is to pad inputs to the same length so that the resulting graphs become identical and can be easily combined. This is done in TensorFlow [1], MXNet [14] and PyTorch [78]. Another form of graph batching is to dynamically analyze a set of input-dependent dataflow graphs and fuse equivalent operators to generate a conglomerate graph. This form of batching is done in TensorFlow Fold [58] and DyNet [67].

Graph batching harms both the latency and throughput of model inference. First, unlike training, the inputs for inference arrive at different times. With graph batching, a newly arrived request must wait for an ongoing batch of requests to finish their execution, which imposes a significant latency penalty. Second, when inputs have varying sizes, not all operators in the combined graph can be batched fully after merging the dataflow graphs for different inputs. An insufficient amount of batching reduces throughput under high load.

1.3.2 Lack of Context State Management

Large Language Models are commonly used in the conversational setup, like ChatGPT. The user and the chatbot are engaged in a dialogue that may last many turns. For the chatbot not to "lose memory" of what has been said so far when responding, the cumulative history of the dialogue must be part of the context for the Large Language Model's autoregressive generation. As existing serving systems are stateless across requests, clients send a growing log of conversation history alongside each new request as the input prompt to be processed from scratch. This causes much duplicate processing for multi-turn conversations.

How can we avoid duplicate processing of the chat history? The solution is to have the serving system save any previously processed context data with the best effort. When new requests from the same conversation arrive, the saved context data can be re-used and subsequently augmented.

Essentially, the serving system is allowed to keep some cached state containing processed context across requests. Doing so enables the serving system to exploit the opportunity that when users are actively chatting with an AI chatbot, follow-up requests usually arrive within a reasonably short time to leverage the cached state.

Caching state across requests is straightforward at the high level, but several challenges remain to make it work. First, where do we save the data? Keeping it in the GPU makes it fast to retrieve, but is very constrained by the relatively small GPU memory size, whereas putting it on disk would incur much longer load latency, hurting the user experience. A two-tier caching solution spanning both GPU and CPU memory is promising, but care must be taken to cope with each tier's capacity limit and to swap in/out the saved context data from/to the GPU without damaging performance. Second, how do we reuse saved context data efficiently when processing a new request? When restoring a conversation's context data from CPU to GPU, how do we merge it with the conversation's existing context that is already cached in the GPU? Furthermore, due to the limit of cache capacity, dropping is inevitable in some cases and what has been saved might not be complete. How do we handle dropped cache?

1.4 Contributions

We argue that we should schedule language model inference in a fine-grained and stateful manner to address the aforementioned challenges. This thesis makes the following two contributions towards efficient serving of language models:

1.4.1 Batching at finer token granularity

To avoid the inefficiency of coarse-grained graph batching, this thesis proposes a new mechanism, called *cellular batching*, that can significantly improve the latency and throughput of Language Model inference. Our key insight is to realize that an iterative language model computation is

made up of varying numbers of similar computation units connected, much like an organism is composed of many cells. As such, we propose to perform batching and execution at the granularity of cells (aka common subgraphs in the dataflow graph) instead of the entire organism (aka the whole dataflow graph), as is done in existing systems.

We build the BatchMaker inference system based on cellular batching. As each input arrives, BatchMaker breaks its computation graph into a graph of cells and dynamically decides the set of common cells that should be batched together for the execution. Cellular batching is highly flexible, as the set of batched cells may come from requests arriving at different times or even from the same request. As a result, a newly arrived request can immediately join the ongoing execution of existing requests, without needing to wait for them to finish. Long requests also do not decrease the amount of batching when they are batched together with short ones: each request can return to the user as soon as its last cell finishes and a long request effectively hitches a ride with multiple short requests over its execution lifetime.

When batching and executing at the granularity of cells, BatchMaker also faces several technical challenges. What cells should be grouped to form a batched task? Given multiple batched tasks, which one should be scheduled for execution next? When multiple GPU devices are used, how should BatchMaker balance the loads of different GPUs while preserving the locality of execution within a request? How can BatchMaker minimize the overhead of GPU kernel launches when a request's execution is broken up into multiple pieces?

We address these challenges and develop a prototype implementation of BatchMaker based on the codebase of MXNet. We have evaluated BatchMaker using several well-known RNN models (LSTM [46], Seq2Seq [92] and TreeLSTM [93]) on different datasets. We also compare the performance of BatchMaker with existing systems including MXNet, TensorFlow, TensorFlow Fold and DyNet. Experiments show that BatchMaker reduces the latency by 17.5-90.5% and improves the throughput by 25-60% for LSTM and Seq2Seq compared to TensorFlow and MXNet. The inference throughput of BatchMaker for TreeLSTM is 4× and 1.8× that of TensorFlow Fold

and DyNet, respectively, and the latency reductions are 87% and 28%.

1.4.2 Stateful Conversation Context Management

To avoid recomputation and efficiently support serving conversations that could last multiple turns, we design a stateful LLM serving system, called Pensieve, which saves hidden representations of a conversation's processed context in a two-tier GPU-CPU cache. It evicts cached data to the next tier (or discards it) according to how long the conversation has been inactive and how large the conversation's saved context has accumulated. The eviction is done at the granularity of a suffix of the conversation instead of the whole. Therefore, a conversation's saved context might span both tiers of the cache and also be partially trimmed. Pensieve uses ahead-of-time swapping and pipelined transfer to overlap computation with the data movement between cache tiers. Dropped contexts are handled via recomputation.

Evicting and restoring cause a conversation's cached context to occupy non-contiguous GPU memory. We develop a new GPU kernel to compute attention [101] between multiple input tokens and cached context residing in non-contiguous memory, which is missing in existing LLM serving systems. Our kernel is a generalized version of the PagedAttention kernel in vLLM [54].

We have built the Pensieve and evaluated its performance on the ShareGPT dataset [86] against vLLM [54], the state-of-the-art serving system that does not cache state across requests. Experiments show that Pensieve can improve serving throughput by 33-100% and reduce latency by 40-77%.

2 | BatchMaker

2.1 Overview

This chapter presents BatchMaker, a system that recognizes the dynamic and iterative nature in language model inference due to the varying input sizes and structures, and avoids the inefficiencies of making batching decisions at full request granularity. BatchMaker proposes cellular batching which breaks the request dataflow graph into recurrent common structures called cells and makes fine-grained batching decisions at the cell level. Cellular batching also allows the request to join or leave the batch whenever needed without waiting for the entire batch to finish, and thus can achieve better latency.

BatchMaker was originally developed for Recurrent Neural Network (RNN) models, which were a dominant model architecture for language models before the era of Transformer. Therefore, in this chapter, we will still use RNNs as the motivating application and evaluation workload. However, the core idea of cellular batching generally applies to all iterative language models including Transformers (discussed in §2.9). Nowadays, this fine-grained batching approach of making decisions at each generation step is widely used in Transformer-based Large Language Models (LLMs).

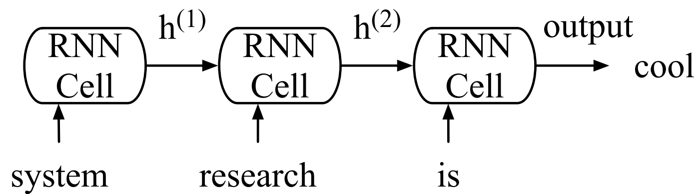


Figure 2.1: An unfolded chain-structured RNN. All RNN Cells in the chain share the same parameter weights.

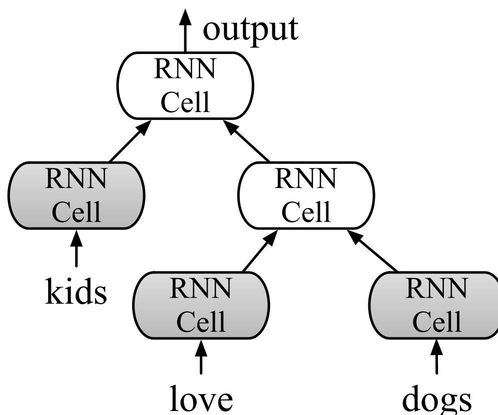


Figure 2.2: An unfolded tree-structured RNN. There are two types of RNN cells, leaf cell (grey) and internal cell (white). All RNN cells of the same type share the same parameter weights.

2.2 Background

In this section, we explain the unique characteristics of RNNs, the difference between model training and inference, the importance of batching, and how it is done in existing deep learning systems.

2.2.1 A primer on recurrent neural networks

Recurrent Neural Network (RNN) is a family of neural networks designed to process sequential data of variable length. RNN is particularly suited for language processing, with applications ranging from speech recognition [6], machine translation [7, 108], to question answering [94, 110].

In its simplest form, we can view RNNs as operating on an input sequence,

$$X = [x^{(1)}, x^{(2)}, ..., x^{(\tau)}]$$

, where $x^{(i)}$ represents the input at the i-th position (or timestep). For language processing, the input X would be a sentence, and $x^{(i)}$ would be the vector embedding of the i-th word in the sentence. RNN's key advantage comes from parameter sharing when processing different positions. Specifically, let f_θ be a function parameterized with θ, RNNs represent the recursive computation $h^{(t)} = f_\theta(h^{(t-1)}, x^{(t)})$, where $h^{(t)}$ is viewed as the value of the hidden unit after processing the input sequence up to the t-th position. The function f_θ is commonly referred to as an RNN cell. An RNN cell can be as simple as a fully connected layer with an activation function, or the more sophisticated Long Short-Term Memory (LSTM) cell. The LSTM cell [46] contains internal cell states that store information and uses several gates to control what goes in or out of those cell states and whether to erase the stored information.

RNNs can be used to model a natural language, solving tasks such as predicting the most likely word following an input sentence. For example, we can use an RNN to process the input sentence "system research is" and to derive the most likely next word from the RNN's output. Figure 2.1 shows the unfolded dataflow graph for this input. At each time step, one input position is consumed and the calculated value of the hidden unit is then passed to the successor cell in the next time step. After unfolding three steps, the output will have the context of the entire input sentence and can be used to predict the next word. It is important to note that each RNN cell in the unfolded graph is just a copy, meaning that all unfolded cells share the same model parameter θ.

Although sequential data are common, RNNs are not limited to chain-like structures. For example, TreeLSTM [93] is a tree-structured RNN. It takes as input a tree structure (usually, the parse tree of a sentence [90]) and unfolds the computation graph to that structure, as shown

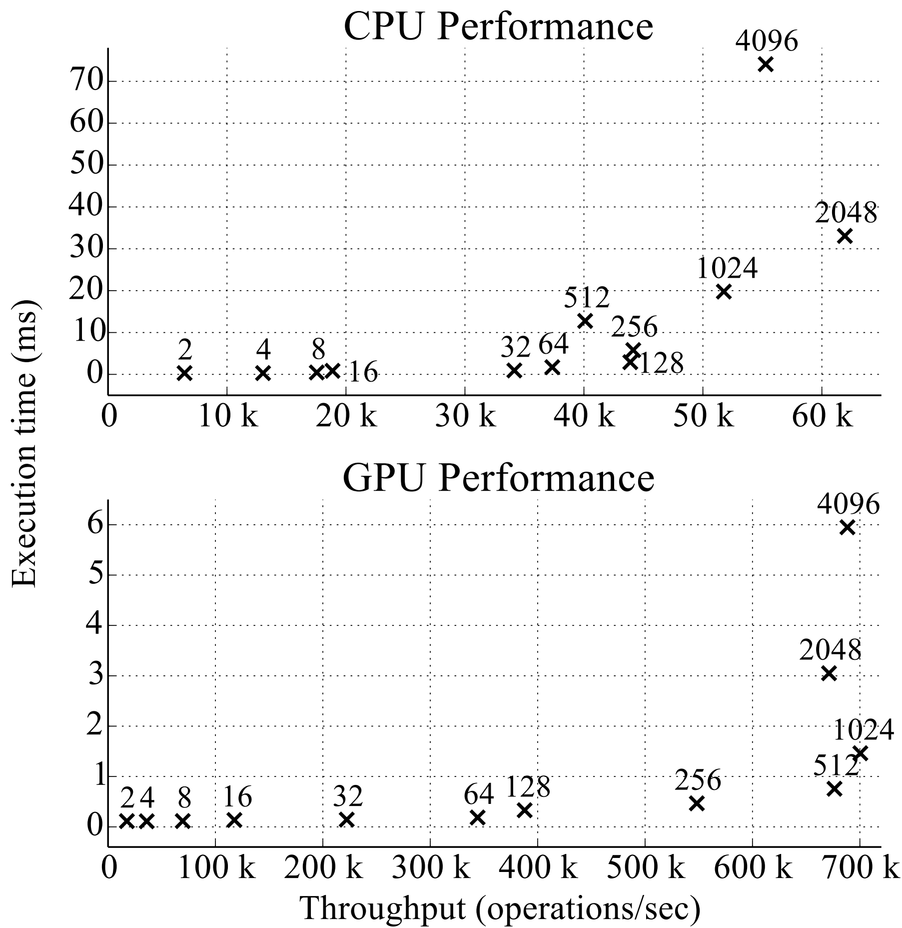


Figure 2.3: Latency vs. throughput for computing a single step of LSTM cell at different batch sizes for CPU and GPU. The value on the marker denotes the batch size.

in Figure 2.2. TreeLSTM has been used for classifying the sentiment of a sentence [77] and the semantic relatedness of two sentences [60].

2.2.2 Training vs. inference, and the importance of batching

Deploying a DNN is a two-phase process. During the offline *training* phase, a model is selected and its parameter weights are computed using a training dataset. Subsequently, during the online *inference* phase, the pre-trained model is used to process application requests.

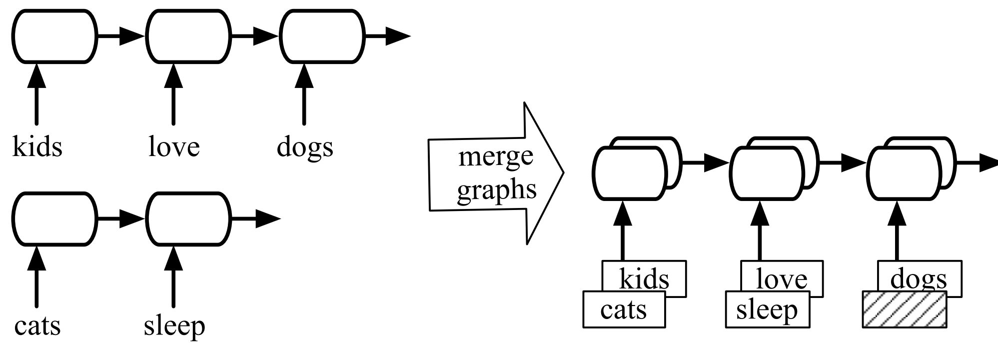
At a high level, DNN training is an optimization problem to compute parameter weights that minimize some loss function. The optimization algorithm is minibatch-based Stochastic Gradient Descent (SGD), which calculates the gradients of the model parameters using a mini-batch of a few hundred training examples, and updates the parameter weights along computed gradients for the subsequent iteration. The gradient computation involves forward-propagation (computing the DNN outputs for those training samples) and backward-propagation (propagating the errors between the outputs and true labels backward to determine parameter gradients). Training cares about throughput: the higher the throughput, the faster one can scan the entire training dataset many times to arrive at good parameter weights. Luckily, the minibatch-based SGD algorithm naturally results in batched gradient computation, which is crucial for achieving high throughput.

DNN inference uses pre-trained parameter weights to process application requests as they arrive. Compared to training, there's no backward-propagation and no parameter updates. However, as applications desire real-time response, inference must strive for low latency as well as high throughput, which are at odds with each other. Unlike training, there is no algorithmic need for batching during inference[1]. Nevertheless, batching is still required by inference for achieving good throughput.

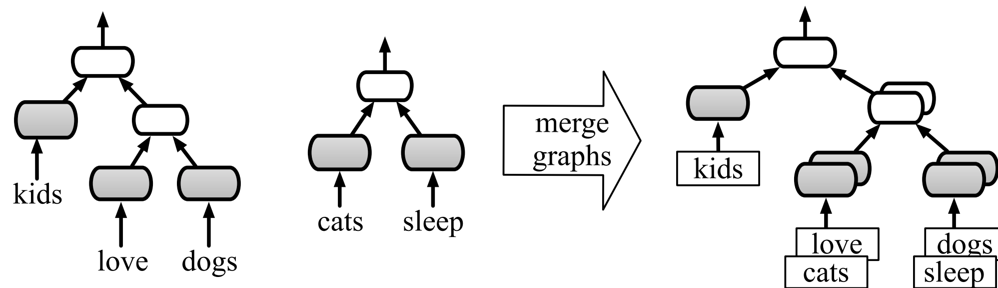
To see the importance of batching for performance, we conduct a micro-benchmark that performs a single LSTM computation step using varying batch sizes (b)[2]. The GPU experiment uses NVIDIA Tesla V100 GPU and NVIDIA CUDA Toolkit 9.0. Figure 2.3 (bottom) shows the execution time of a batch vs. the overall throughput, for batch sizes $b = 2, 4, ...2048$. We can see that the execution time of a batch remains almost unchanged at first and then increases sublinearly with b. When $b > 512$, the execution time approximately doubles as b doubles. Thus, setting $b = 512$ results in the best throughput. We also ran CPU experiments on Intel Xeon Processor

[1]The SGD algorithm used in training is best done in mini-batches. This is because the gradient averaged across many inputs in a batch results in a better estimate of the true gradient than that computed using a single input.

[2]We configure the LSTM hidden unit size $h = 1024$. The LSTM implementation involves several element-wise operations and one matrix multiplication operation with input tensor shapes $b \times 2h$ and $2h \times 4h$.



(a) Graph batching via padding



(b) Graph batching in TensorFlow Fold and DyNet

Figure 2.4: Existing systems perform graph batching

E5-2698 v4 with 32 virtual cores. The LSTM cell is implemented using Intel's Math Kernel Library (2018.1.163). As Figure 2.3 (top) shows, batching is equally important for the CPU. On both the GPU and CPU, batching improves throughput because increasing the amount of computation helps saturate available computing cores and masks the overhead of off-chip memory access. As the CPU performance lags far behind that of the GPU, we focus our system development on the GPU.

2.2.3 Existing solutions for batching RNNs

Batching is straightforward when all inputs have the same computation graph. This is the case for certain DNNs such as Multi-layer Perceptron (MLP) and Convolution Neural Networks (CNNs). However, for RNNs, each input has a potentially different recursion depth and results in an unfolded graph of different sizes. This input-dependent structure makes batching for RNNs chal-

lenging.

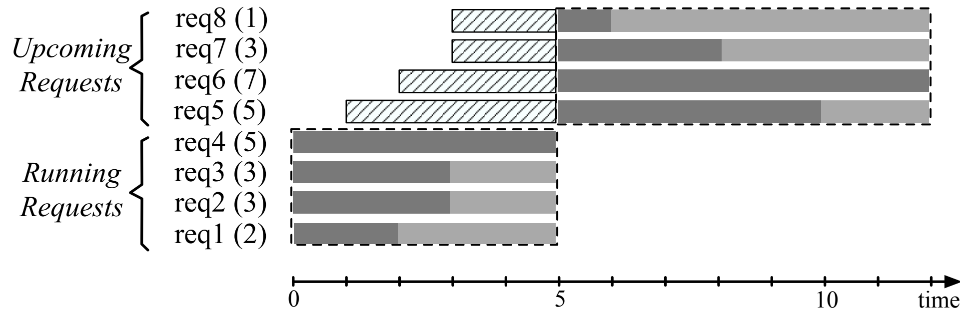
Existing systems fall into two camps in terms of how they batch for RNNs:

1. **TensorFlow/MXNet/PyTorch/Theano:** These systems pad a batch of input sequences to the same length. As a result, each input has the same computation graph and the execution can be batched easily. An example of batching via padding is shown in Figure 2.4(a). However, padding is not a general solution and can only be applied to RNNs that handle sequential data using a chain-like structure. For non-chain RNNs such as TreeLSTMs, padding does not work.

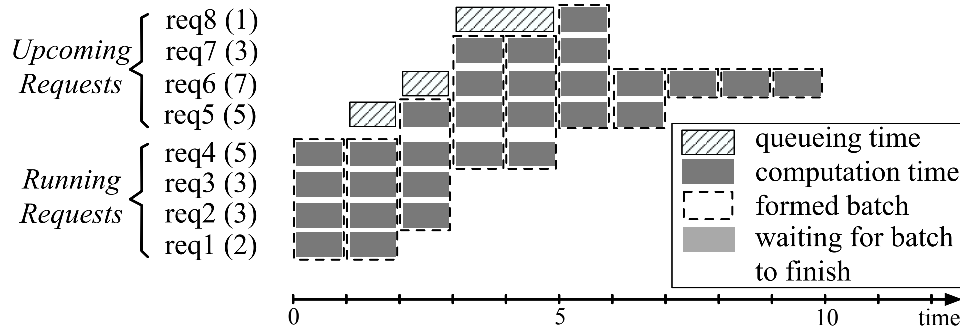
2. **TensorFlow-Fold/DyNet:** In these two recent works, the system first collects a batch of input samples and generates the dataflow graph for each input. The system then merges all these dataflow graphs into one graph where some operator might correspond to the batched execution of operations in the original graphs. An example is shown in Figure 2.4(b).

Both above existing strategies try to collect a set of inputs to form a batch and find a dataflow graph that's compatible with all inputs in the batch. As such, we refer to both strategies as *graph batching*. Existing systems use graph batching for both training and inference. We note that graph batching is ideal for RNN training. First, since all training inputs are present before training starts, there is no delay in collecting a batch. Second, it does not matter if a short input is merged with a long one because the mini-batch (synchronous) SGD must wait for the entire batch to finish to compute the parameter gradient anyway.

Unfortunately, graph batching is far from ideal for RNN inference and negatively affects both the latency and throughput. Graph batching incurs extra latency due to unnecessary synchronization because the input cannot start executing unless *all* requests in the current batch have finished. This is further exacerbated in practice when inputs have varying lengths, causing some long inputs to delay the completion of the entire batch. Graph batching can also result in suboptimal throughput, either due to performing useless computation for padding or failing to batch



(a) Graph Batching



(b) Cellular Batching

Figure 2.5: The timeline of graph batching and Cellular Batching when processing 8 requests from req1 to req8. The number shown in parenthesis is the request's sequence length, e.g. req1(2) means req1 has a sequence length of 2. Each row marks the lifetime of a request starting from its arrival time. Req1-4 are *Running Requests* as they arrive at time 0 and have started execution. Req5-8 are *Upcoming Requests* that arrive after the Req1-4.

at the optimal level for all operators in the merged dataflow graph.

2.3 Our Approach: Cellular batching

We propose cellular batching for RNN inference. RNN has the unique feature that it contains many identical computational units connected. Cellular batching exploits this feature to 1) batch at the level of RNN cells instead of whole dataflow graphs, and 2) let new requests join the execution of the current requests and let requests return to the user as soon as they finish.

2.3.1 Batching at the granularity of cells

Graph batching is not efficient for inference because it performs batching at a coarse granularity–a dataflow graph. The recursive nature of RNN enables batching at a finer granularity–an RNN cell. Since all unfolded RNN cells share the same parameter weights, there is ample opportunity for batching at the cell level: each unfolded cell of request X can be batched with any other unfolded cell from request Y. In this way, RNN cells resemble biological cells which constitute all kinds of organisms. Although organisms have numerous types and shapes, the number of cell types they have is much more limited. Moreover, regardless of the location of a cell, cells of the same type perform the same functionality (and can be batched together). This characteristic makes it more efficient to batch at the cell level instead of the organism (dataflow graph) level.

More generally, we allow programmers to define a cell as a (sub-)dataflow graph and to use it as a basic computation unit for expressing the recurrent structure of an RNN. A simple cell contains a few tensor operators (e.g. matrix-matrix multiplication followed by an element-wise operation); a complex cell such as LSTM not only contains many operators but also its internal recursion. Grouping operators into cell allows us to make the unfolded dataflow graph coarse-grained, where each node represents a cell and each edge depicts the direction in which data flows from one cell to another. We refer to this coarse-grained dataflow graph as cell graph.

There may be more than one type of cells in the dataflow graph. Two cells are of the same type if they have identical sub-graphs, share the same parameter weights, and expect the same number of identically shaped input tensors. Cells with the same type can be batched together if there is no data dependency between them.

2.3.2 Joining and leaving the ongoing execution

In graph batching, the system collects a batch of requests, finishes executing all of them, and then moves on to the next batch. By contrast, in the cellular batching, there is no notion of a

fixed batch of requests. Rather, new requests continuously join the ongoing execution of existing requests without waiting for them to finish. This is possible because a new request's cells at an earlier recursion depth can be batched together with existing requests' cells at later recursion depths.

Existing deep learning systems such as TensorFlow, MXNet and DyNet schedule an entire dataflow graph for execution. To support continuous join, we need a different system implementation that can dynamically batch and schedule individual cells. More concretely, our system unfolds each incoming request's execution into a graph of cells and continuously forms batched tasks by grouping cells of the same type. When a task has batched sufficiently many cells, it is submitted to a GPU device for execution. Therefore, as long as an ongoing request still has remaining cells that have not been executed, they will be batched together with any incoming requests. Furthermore, our system also returns a request to the user as soon as its last cell finishes. As a result, a short request is not penalized with increased latency when it's batched with longer requests.

Figure 2.5 illustrates the different batching behavior of Cellular Batching and graph batching when processing the same 8 requests. We assume a chain-structured RNN model and that each RNN cell in the chain takes one unit of time to execute. Each request corresponds to an input sequence whose length is shown in the parentheses. In the Figure, each row shows the lifetime of one request, starting from its arrival time. The example uses a batch size of 4.

At the beginning of time (t=0), the first 4 requests (req1-4) arrive. Under graph batching, these 4 requests form a batch, and their corresponding dataflow graphs are fused and submitted to the backend for execution. The system does not finish executing the fused graph until time t=5, as the longest request in the batch (req4) has a length of 5. In the meanwhile, newly arrived requests (req5-8) are being queued up and form the next batch. The system starts executing the next batch at t=5 and finishes at t=12. Under cellular batching, among the first 4 requests, the system forms two fully batched tasks, each performing the execution of a single (4-way batched) RNN cell. At

t=2, the second task finishes, causing req1 to complete and leave the system. Since a new request (req5) has already arrived, the system forms its third fully batched task containing req2-5 at t=2. After finishing this task, another two existing requests (req2,req3) depart, and two new ones are added (req6, req7) to form the fourth task. As shown in this example, cellular batching not only reduces the latency of each request (due to less queuing) but also increases the overall system throughput (due to tighter batching).

2.4 System Design

We build an inference system, called BatchMaker, based on cellular batching. This section describes the basic system design.

2.4.1 User Interface

To use BatchMaker, users must provide two pieces of information: the definition of each cell (i.e. the cell's dataflow graph) and a user-defined function that unfolds each request/input into its corresponding cell graph. We expect users to obtain a cell's definition from their training programs for MXNet or TensorFlow. Specifically, users define each RNN cell using MXNet/TensorFlow's Python interface and save the cell's dataflow graph in a JSON file using existing MXNet/TensorFlow facilities. The saved file is given to BatchMaker as the cell definition. In our current implementation, the user-defined unfolding logic is expressed as a C++ function which uses our given library functions to create a dataflow graph of cells.

2.4.2 Software Architecture

BatchMaker runs on a single machine with potentially many GPU devices. Its overall system architecture is depicted in Figure 2.6. BatchMaker has two main components: *Manager* and *Worker*.

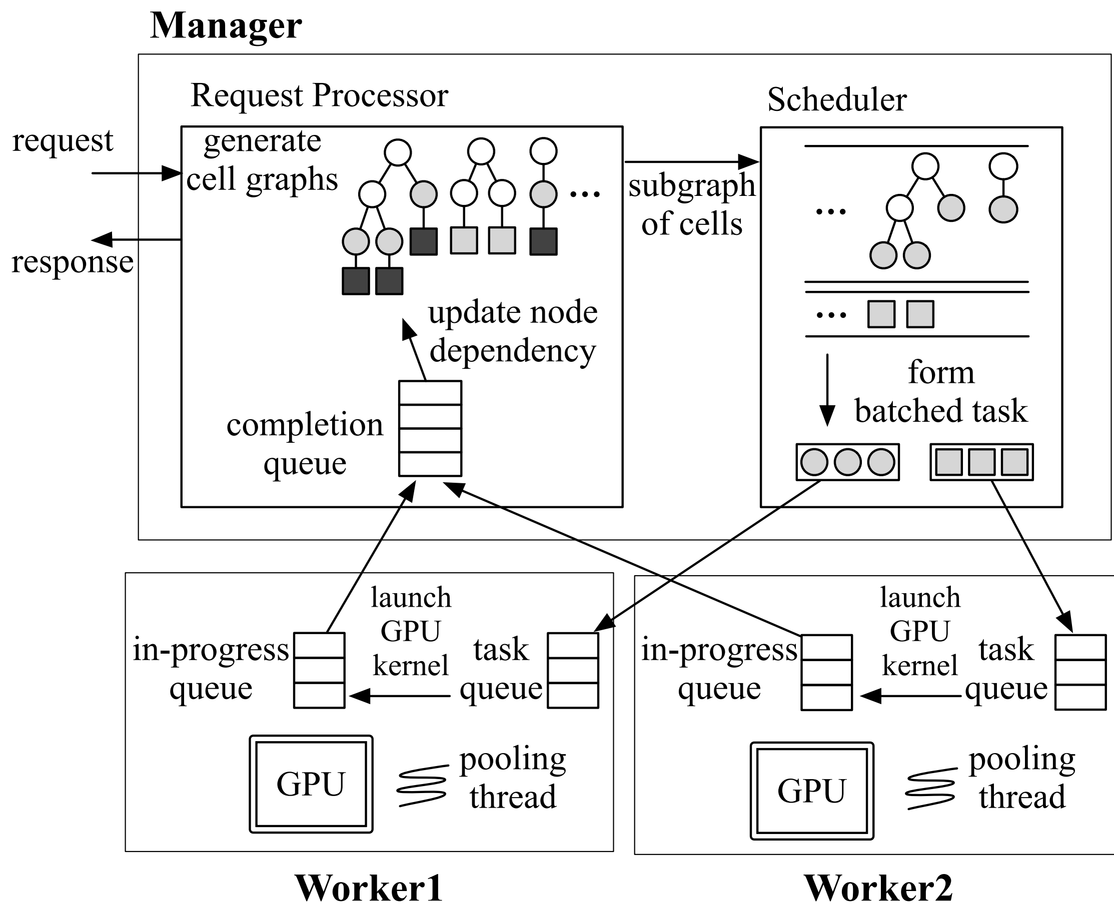


Figure 2.6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

The manager processes arriving requests and submits batched computation tasks to workers for execution. Depending on the number of GPU devices equipped, there may be multiple workers, each of which is associated with one GPU device. Workers execute tasks on GPUs and notify the manager when their tasks are complete.

System initialization. Upon startup, BatchMaker loads each cell's definition and its pre-trained weights from files. BatchMaker "embeds" the weights into cells so that weights are part of the internal state as opposed to the inputs to a cell. For a cell to be considered batchable, the first dimension of each of its input tensors should be the batch dimension. BatchMaker identifies the type of each cell by its definition, weights, and input tensor shapes.

The workflow of a request. The manager consists of two submodules, *request processor* and *scheduler*. The request processor tracks the progress of execution for each request and the scheduler determines which cells from different requests would form a batched task, and selects a worker to execute the task.

When a new request arrives, the request processor runs the user code to unfold the recursion and generates the corresponding cell graph for the request. In this cell graph, each node represents a cell and is labeled with a unique node ID as well as its cell type. Request processor will track and update the dependencies of each node. When a node's dependencies have been satisfied (its inputs are ready), the node is ready to be scheduled for execution (§2.4.3). The scheduler forms batched tasks among ready nodes of the same cell type. Each type of cell has a desired maximum batch size, which is determined through offline benchmarking. Once a task has reached a desired batch size, it is pushed into the task queue of one of the workers.

Workers execute tasks on GPUs. Since the GPU kernel execution is asynchronous, the worker moves a task from the task queue to the in-progress queue once the task's corresponding GPU kernel has been issued. The worker uses a pooling mechanism to see whether some task has

finished. It pops the finished task from the in-progress queue and pushes it into the completion queue in the request processor. The request processor updates node dependencies based on the completed task and checks whether a request is finished. If so, its result is immediately returned.

We give a more detailed example of the request workflow in §2.4.4.

2.4.3 Batching and Scheduling

The scheduler needs to make decisions on what nodes should be batched together to form a task and what tasks to be pushed to which workers. The design must take into account three factors, locality, priority, and utilization of multiple GPUs, which are often in conflict with each other.

Locality refers to the preference that 1) the same set of requests should be batched together if they are to execute the same sequence of nodes, and 2) the execution of that sequence of nodes should stick to the same GPU. The reason for both 1) and 2) is to avoid memory copy. Before execution, the batched inputs of a cell must be laid out in contiguous GPU memory. Since the batched outputs of the execution are also stored in contiguous memory, there is no need for memory copy before each cell execution when executing the same set of requests on the same GPU. Conversely, if the batch of requests changes between two successive cell executions, one must do a memory copy, called "gather", to ensure contiguous inputs. Furthermore, if the execution of successive cells switches from one GPU to another, one must copy data from one GPU to another.

Priority refers to the ability to prefer the execution of one type of cell over another. Many practical RNN models have multiple types of cells. For example, as shown in Figure 2.2, TreeLSTM has leaf cells and internal cells. The popular RNN-based Seq2Seq model has encoder cells and decoder cells. For these models, one can achieve better latency by preferentially executing DNN types that occur later in the computation graph. Therefore, in TreeLSTMs, internal nodes should be given preference over leaf nodes. In Seq2Seq models, decoder nodes should have priority over encoder nodes.

Algorithm 1: Scheduling and Batching Algorithm

```
1  Bsizes: a set of supported batch sizes.
2  CellTypes: a set of cell types, each associated with a priority.
3  MaxTasksToSubmit: the maximum number of tasks that can be submitted to a worker.
4  def Schedule(worker):
5      S ← {ct ∈ CellTypes | ct.NumReadyNodes() ≥ Bsizes.Max()};
6      if S.Size() = 0 :
7          S ← {ct ∈ CellTypes | ct.NumRunningTasks() = 0 and ct.NumReadyNodes() > 0};
8      if S.Size() = 0 :
9          S ← { ct ∈ CellTypes | ct.NumReadyNodes() > 0};
10     ct ← GetCellTypeWithMaxPriority(S);
11     Batch(ct, worker);
12 def Batch(ct, worker):
13     num_tasks ← 0;
14     while num_tasks < MaxTasksToSubmit :
15         batch ← FormBatchedTask(ct, worker);
16         if batch.Size() >= Bsizes.Min() or num_tasks = 0 :
17             SubmitBatchedTask(batch, worker);
18             UpdateNodesDependency(batch);
19             num_tasks++;
20             for subgraph ∈ batch :
21                 subgraph.pinned ← worker.id;
                   # pinned is unset once subgraph has no task running
22         else:
23             break;
24 def FormBatchedTask(ct, worker):
25     batch ← { };
26     for subgraph ∈ ct.subgraphs :
27         if subgraph.pinned ∈ {None, worker.id} :
28             for node ∈ subgraph's ready nodes :
29                 batch ← batch ⋃ {node};
30                 if batch.Size() = Bsizes.Max() :
31                     return batch;
32     return batch;
```

We design a scheduling policy to make the trade-off between locality, priority, and good utilization of multiple GPUs. We support the locality preference by constructing and scheduling a batched task containing multiple node invocations instead of a single one. To enable this, the request processor analyzes the cell graph of a request to find a *subgraph* to pass to the scheduler. A *subgraph* contains a single node or many connected nodes with the property that all external dependencies to other parts of the graph have been satisfied. Furthermore, all nodes of a subgraph must be of the same cell type. For example, in the case of Seq2Seq, a sequence of encoders cells forms one subgraph and the sequence of decoder cells forms another subgraph.

Scheduling subgraphs. The scheduling algorithm is shown in Algorithm 1. For each type of cell, scheduler maintains a queue of subgraphs (the type of a cell is the same as the type of nodes in the subgraphs). The scheduler's "*Schedule*"' function (line 4) is invoked whenever some worker becomes idle, and it picks a cell type *ct* for execution in the following order: (a) *ct* whose queue contains more ready nodes (meaning nodes whose data dependency is satisfied) than the maximum batch size (line 5); (b) *ct* whose queue contains some ready nodes but that has no running tasks (lines 6-7); (c) *ct* whose queue contains some ready nodes (lines 8-9). If there are multiple choices from the same criterion, the scheduler chooses the one with the highest cell priority (line 10). Once a cell type is chosen, scheduler invokes "*Batch*" to form batched tasks (line 11).

Batching subgraphs. Given a cell type, "*Batch*" function (line 12) selects nodes from subgraphs in the cell's queue to form a batched task (line 15) and to submit to the device for execution (line 17). "*FormBatchedTask*" function (line 24) scans the queue to select nodes whose dependencies are satisfied (line 28) for a batched task. Each invocation of "*FormBatchedTask*" forms at most one batched task. For better locality, the scheduler submits several tasks to the same worker for execution. The number of tasks submitted is limited by the configurable parameter "MaxTasksToSubmit" (line 14). Setting the limit to a small number (default is 5) avoids forming too many tasks belonging to one cell type, which gives other types of cell a chance to be scheduled, allows new

requests to join execution, and avoids the decrease in effective batch size if one cell type does not have enough ready nodes.

Once a batched task is submitted to the GPU worker, the scheduler updates the dependencies of those nodes in the batch (line 18) so a new set of cell nodes can be scheduled in subsequent batched tasks. Additionally, the scheduler pins those subgraphs to the same worker (lines 20-21) to avoid scheduling the subsequent nodes in those subgraphs to a different worker (line 27). This is crucial to the correctness of the scheduling algorithm because tasks involving the same subgraph have data dependency. Tasks submitted to the same device will execute in the order of submission, and thus their dependency is fulfilled. By contrast, there is no such guarantee for tasks submitted to different workers. Besides, this also improves locality because all nodes in the same subgraph are preferentially scheduled to the same worker. The scheduler maintains a counter for each subgraph to count how many batched tasks contain nodes from this subgraph. When this counter is decreased to zero, the scheduler unpins the subgraph from a worker so that this subgraph may be scheduled on other workers in the future.

2.4.4 An example of TreeLSTM scheduling

To give a concrete example, we show the detailed workflow of a TreeLSTM request. When a TreeLSTM request x arrives at our system, the request processor applies the user-defined unfolding function to generate the cell graph for request x. Then the request processor analyzes the cell graph and breaks it up into subgraphs based on cell types. A TreeLSTM model has two types of cell: leaf cell and internal cell. Suppose request x is a complete binary tree with 16 leaf nodes. Then its cell graph will be partitioned into 17 subgraphs: one subgraph contains 31 internal tree nodes; each of the other 16 subgraphs contains a single leaf node. The set of leaf subgraphs is immediately passed to the scheduler because their dependencies are satisfied, whereas the subgraph of internal nodes remains at the request processor.

The scheduler maintains two queues, one for the internal cell type, and the other for the leaf cell type. When the leaf cell type is scheduled, leaves of x will be put into potentially several batched tasks with leaf nodes from other requests. Batched tasks are pushed to the worker and executed. The request processor is notified when subgraphs finish. When all the leaf subgraphs of request x finish, the subgraph containing x's internal nodes has its dependencies satisfied and is then passed to the scheduler. When the internal cell type is scheduled, the scheduler puts the cells of x at successive levels of the tree in successive batched tasks to ensure that their dependencies are obeyed. The number of nodes from request x decreases at higher tree levels. But the scheduler will batch nodes from request x with nodes from other requests to keep the batch size close to the maximum allowed. Once all internal nodes of request x finish execution, the request processor gets notified. When there is no more subgraph to execute, request x departs immediately.

2.5 GPU Optimization

The manager and workers threads in BatchMaker run on the CPU and RNN cells are scheduled to execute on the GPUs. The synchronization between CPU and GPU is non-trivial and has a big impact on the utilization of GPUs. In this section, we explain two optimizations in BatchMaker that are crucial for achieving good performance on GPUs.

Keeping the GPU busy. One should not schedule a GPU kernel for execution one at a time. Doing so is terrible for performance as the GPU sits idle waiting for the next kernel to be scheduled and launched. In BatchMaker, the worker asynchronously pushes all GPU kernels for a given task to the GPU's driver queue without waiting for any to finish. To ensure that the dependencies of each kernel are satisfied, the worker performs a topological sort of all operators within the cell and pushes kernels according to the sort order to the same GPU stream. This works because the GPU driver guarantees that kernels in the same stream are executed in the FIFO order. A worker

may receive up to *MaxTaskToSubmit* number of tasks from the scheduler. The order in which the workers receive these tasks already correctly reflects the dependencies between cells. Thus, the worker also launches the kernels for multiple tasks based on their order in the task queue. By launching as many kernels as possible while obeying dependencies, we effectively reduce the kernel launch gap between operators and tasks.

Asynchronous Completion Notification. The worker cannot synchronously wait for a task to finish execution on the GPU. Nevertheless, the worker must learn quickly when a task has finished so that it can inform the manager who will issue the next set of nodes to the scheduler. Existing solutions supporting asynchronous notification use the callback mechanism provided by GPU device drivers. However, these callback mechanisms have performance limitations. For example, the NVIDIA CUDA driver's callback mechanism blocks all kernel execution until the callback function finishes[25].

To let the worker learn of task completion asynchronously, we add a *signaling kernel* to the end of each task. The signaling GPU kernel changes a signal variable, which is an unsigned integer in our implementation. The signal variable is allocated in pinned host memory which can be accessed by GPU using zero copy. Whenever a task finishes execution, the signaling kernel will execute next and increase the signal variable by one, which means the GPU has finished the execution of one more task. On the worker side, it pushes the tasks that have been issued to GPU in a FIFO queue called *in-progress queue*. The worker uses a thread to continuously poll the status of the signal variable. Once the signal variable changes, the worker learns that the task at the top of the in-progress queue has been finished. It then pushes the completed task to the manager's completion queue.

2.6 Implementation

We implemented our system using the codebase of MXNet (version 0.10.0). In our current prototype, users need to define each cell in a JSON file exported by MXNet's Python API. Users also need to provide a user-defined C++ function to generate the unfolded cell graph for each request.

During initialization, BatchMaker aims to materialize all the cells for each supported batch size on every available GPU device. Such materialization requires knowledge of the type and shape information of each operator's input/output tensors to allocate GPU memory and perform other compiler-level optimizations such as those done by NNVM [20], TensorFlow XLA [21].

We reuse the MXNet parsing mechanism to perform type and shape inference for each type of cell. However, to do that, BatchMaker needs to know how cells can be connected, and how the outputs of one cell may be used by other cells. Ideally, BatchMaker should learn this knowledge on the fly when real requests come in. To simplify implementation, our current prototype requires the user to provide an example request so that BatchMaker can apply the user-defined unfolding function to generate an example cell graph. This allows BatchMaker to perform type and shape inference and materialize cells during initialization.

During inference, BatchMaker re-use materialized cells over and over. For example, to execute an LSTM chain with a length of 5, our worker will execute the same materialized LSTM cell for 5 times.

2.7 Evaluation

We evaluate BatchMaker on microbenchmarks and several popular RNN applications with real-world datasets. Our evaluation shows that BatchMaker provides significant performance advantages over existing systems (including MXNet, TensorFlow, TensorFlow Fold and DyNet).

The highlights of our results are:

- BatchMaker achieves much lower latency than existing systems. Under moderate load (meaning that the load is less than half of the baseline system's peak), we reduce the 90-percentile latency by 37.5%-90.5% (for LSTM) and 17.5%-82.6% (for Seq2Seq) compared to TensorFlow and MXNet. For TreeLSTM, we reduce 90-percentile latency by 28% and 87% compared to DyNet and TensorFlow Fold respectively.
- BatchMaker also provides good throughput improvements. The throughput improvement over MXNet and TensorFlow is 25% (for LSTM) and 60% (for Seq2Seq). For TreeLSTM, the throughput of BatchMaker is 1.8× that of DyNet and 4× that of TensorFlow Fold.
- Our latency improvement mainly comes from reducing the queuing time of new requests. The performance advantage of BatchMaker is increased when the variance in the sequence length is large.

2.7.1 Experimental Setup

The Testbed. We run our tests on a Linux server with 4 NVIDIA TESLA V100 GPU cards are connected by NVLink; each GPU has 16GB of memory. The operating system is Ubuntu 16.04.1 LTS with Linux kernel version 4.13.0. NVIDIA CUDA Toolkit version is 9.0.

Applications, datasets, and workloads. We choose three popular RNN applications, LSTM, Seq2Seq, and TreeLSTM. All RNN cells used in these applications use hidden state size 1024. LSTM and Seq2Seq are both chain-structured RNNs. We use WMT-15 [99] Europarl German-English translation as our dataset. For LSTM, we randomly sample 100k English sentences. For Seq2Seq, we randomly sample 100k German-English sentence pairs. The maximum sentence length is 330 and the average length is 24. For TreeLSTM, We use Stanford's TreeBank [91] dataset with 10K parse trees of English sentences.

When evaluating an application, we sample a request from the dataset and issue it to the system with Poisson inter-arrival times. We adjust the average inter-arrival time to test the system's performance under varying loads.

Baseline Systems We compare against MXNet (v0.12.0), TensorFlow (v1.4), TensorFlow Fold (v0.0.1) and DyNet (v2.0). MXNet and TensorFlow are representative systems that rely on padding to achieve batching. TensorFlow Fold and DyNet are two existing systems that perform graph batching by dynamically merging a set of dataflow graphs. For chain-structured RNNs, MXNet and TensorFlow achieve much better performance than TensorFlow Fold and DyNet and thus we focus on comparing BatchMaker to these two systems for LSTM and Seq2Seq benchmarks. As padding does not work with tree-structured RNNs, we focus on comparing BatchMaker to TensorFlow Fold and DyNet for the TreeLSTM benchmarks.

Bucketing optimization for MXNet and TensorFlow. Since padding wastes computation, we reduce the amount of padding in MXNet and TensorFlow by only batching requests of similar lengths. We refer to this as the "bucketing" strategy. Specifically, we assign each incoming request to a bucket based on its length. The width of a bucket refers to the maximum difference in lengths among requests in a bucket. We use the bucket width of 10 by default, which gives the best performance for our applications (§2.7.2). Since the WMT-15 dataset has a maximum sentence length of 330, using a width of 10 results in 33 buckets in total. The i-th bucket handles requests of length in the range $(10i, 10i + 10]$. We perform round-robin scheduling across buckets. To reduce latency when running in MXNet and TensorFlow, we materialize a dataflow graph for each bucket during initialization. This is because the cost of materializing a dataflow graph is substantial, owing to compiler optimization and GPU memory allocation.

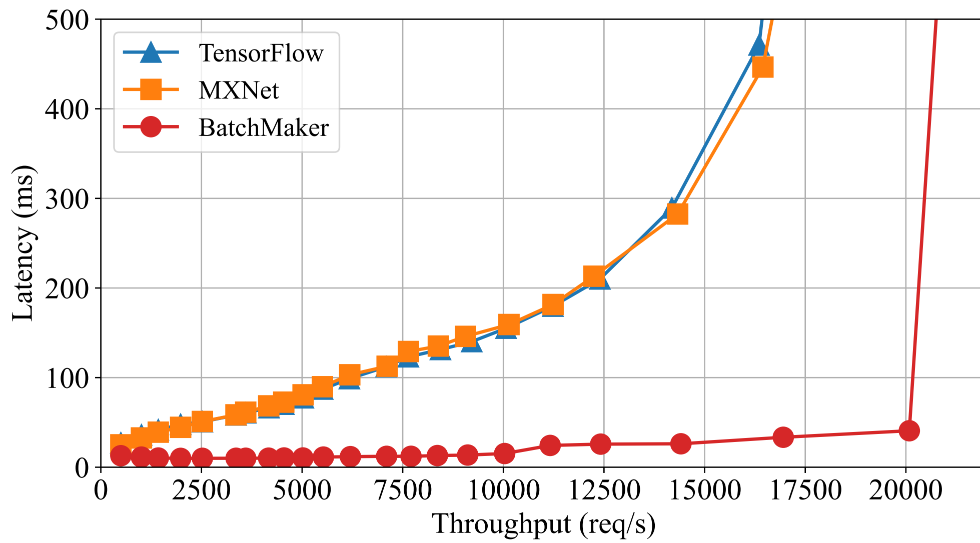
Batching configuration and optimization. Unless otherwise mentioned, we set the maximum batch size to be $b_{max} = 512$, which optimizes for throughput based on Figure 2.3 (bottom).

In our evaluation of MXNet and TensorFlow, we do not use explicit timeouts when accumulating requests to form a batch; rather, even if it's not full, a batch can start execution (as a smaller batch) as long as some GPU device is idle and it is the batch's turn to execute according to the round-robin policy. As a result, when the request rate is low, the actual batch size executed in a system could be lower than the configured maximum. As we will demonstrate in (§2.7.2), this enables a large b_{max} to achieve the same low latency as a small b_{max}. Additionally, we found that this strategy achieves lower latency than any configuration of the timeout-based strategy.

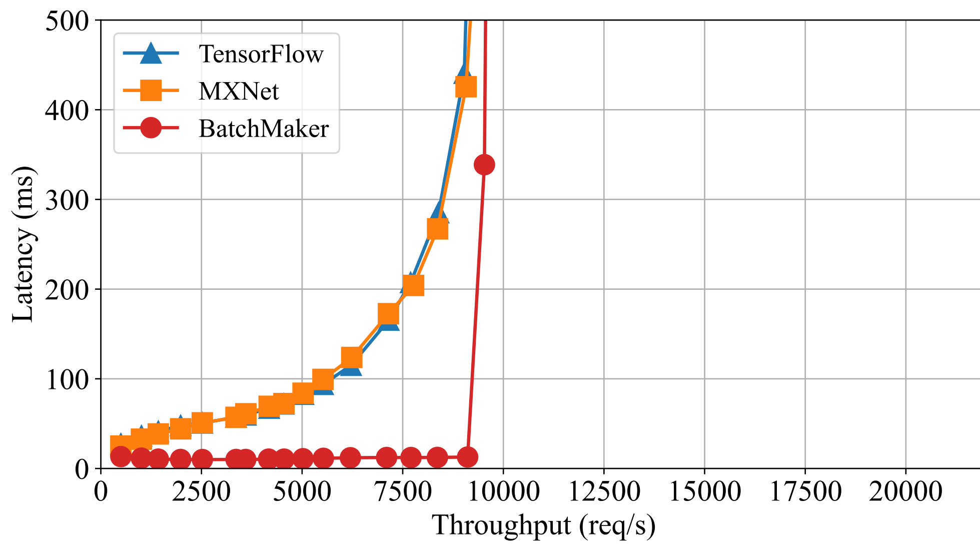
2.7.2 Application Performance: LSTM

We evaluate LSTM inference performance on the WMT-15 Europarl dataset and compare BatchMaker with MXNet and TensorFlow. Figure 2.7 shows the throughput vs. latency results of all systems as the load increases (by reducing the request inter-arrival time). We set the bucket width of 10 by default for MXNet and TensorFlow.

Latency. Figure 2.7 (a) shows the 90-percentile latency vs. throughput ($b_{max} = 512$). As can be seen in the figure, BatchMaker achieves significantly lower latency than MXNet and TensorFlow. The 90p-latency of BatchMaker stays unchanged (12ms) when the throughput is less than 8K req/s, and goes slightly up afterward until peak throughput (20K req/s). This is because when the load is moderate(< 8K req/s), BatchMaker executes most requests in batch sizes no larger than 64. And as the throughput goes up, the batch sizes increase (up to $b_{max} = 512$), leading to a gradual increase in latency. By comparison, the smallest 90p latency of MXNet and TensorFlow is 25ms and the latency increases quickly as the request rate increases. BatchMaker reduces latency because it allows incoming requests to join the currently executing batch, resulting in less queuing time. By comparison, in MXNet and TensorFlow, a new request may need to wait for multiple batches from different buckets to finish execution. Thus, the latency of MXNet and TensorFlow is much higher and increases quickly with increasing load. The latency variance for



(a) LSTM with maximum batch size 512



(b) LSTM with maximum batch size 64

Figure 2.7: LSTM performance on the WMT-15 Europarl dataset using 1 GPU. The figures plot the 90-percentile latency. MXNet and TensorFlow use a bucket width of 10.

BatchMaker is also much smaller and is caused by varying request sequence lengths. In § 2.7.3, we conduct additional experiments to confirm the reasons for BatchMaker's latency improvements.

Throughput. In Figure 2.7 (a), the peak throughput of BatchMaker is 20K req/s (b_{max} = 512), higher than those of MXNet and TensorFlow. As the load increases in MXNet and TensorFlow, a request must wait for more buckets to finish execution and each bucket also executes a larger batch of requests at a time. This causes the overall latency to shoot up beyond 500ms as the load increases to 16K req/s. By contrast, BatchMaker can maintain a low queuing delay while packing more requests into a batch as the input load increases, resulting in much a higher peak throughput.

Effect of different maximum batch sizes. Figure 2.7 (b) shows the latency vs. throughput results using a smaller maximum batch size, $b_{max} = 64$. 64 and 512 are interesting batch size choices because any batch size $b < 64$ has similar latency (for executing one step of LSTM) but lower throughput (than that of $b = 64$), and any batch size $b > 512$ has similar throughput but higher latency (than that of $b = 512$), as shown micro-benchmark in Figure 2.3 (bottom). Comparing Figure 2.7 (a) with (b), we see that $b_{max} = 512$ achieves similar latency as $b_{max} = 64$ (at low to moderate load) but much higher throughput. This is because, at low load, all systems execute with effective batch sizes much smaller than the configured maximum. Therefore, the optimal configuration for all systems is to set the maximum batch size that optimizes the throughput.

The bucket width trade-off in MXNet and TensorFlow. The granularity of buckets creates a trade-off between throughput and latency. Fine-grained bucketing reduces padding and wasteful computation. However, fine-grained bucketing uses a large number of buckets. This causes a batch for any given bucket to wait longer for batches from other buckets to finish execution before it catches its turn under the round-robin policy. By contrast, coarse-grained bucketing uses fewer buckets which results in shorter waiting time but increases the amount of padding

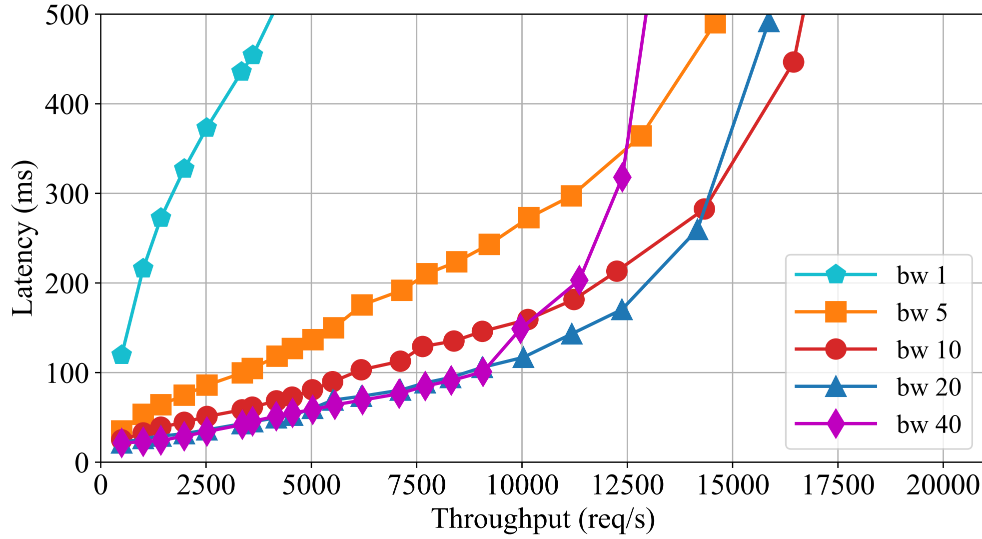


Figure 2.8: LSTM performance using MXNet with different bucket widths (bw *). The maximum batch size is 512.

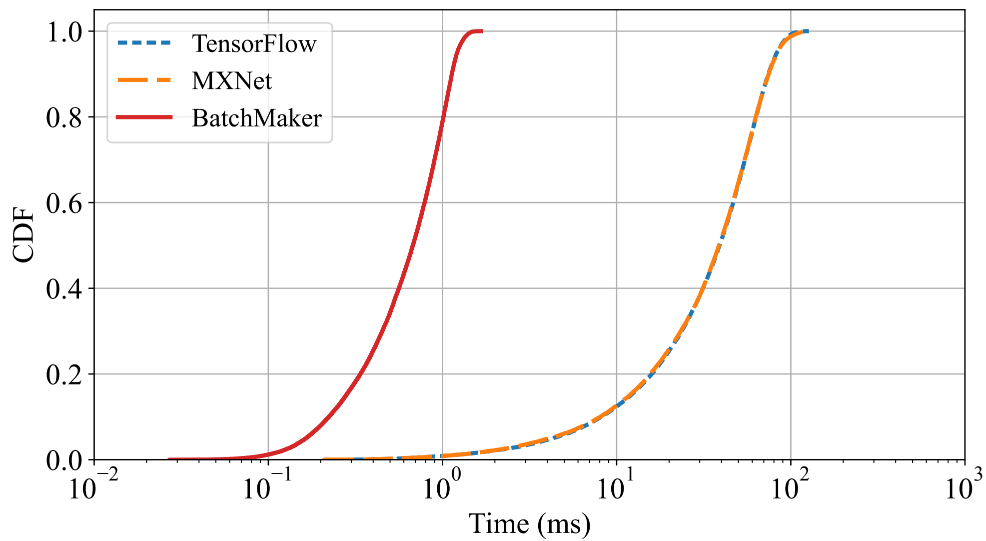
and wasteful computation.

Figure 2.8 shows the latency vs. throughput for MXNet under varying bucket widths. As the figure shows, coarse-grained bucketing (width 40) achieves better latency under low load (due to less waiting) but its peak throughput is also lower (due to more padding). On the contrary, using the smallest bucket width of 1 has the best peak throughput, but at the cost of higher latency for low to moderate load. Using the bucket width of 10 achieves a good tradeoff of low latency and high peak throughput.

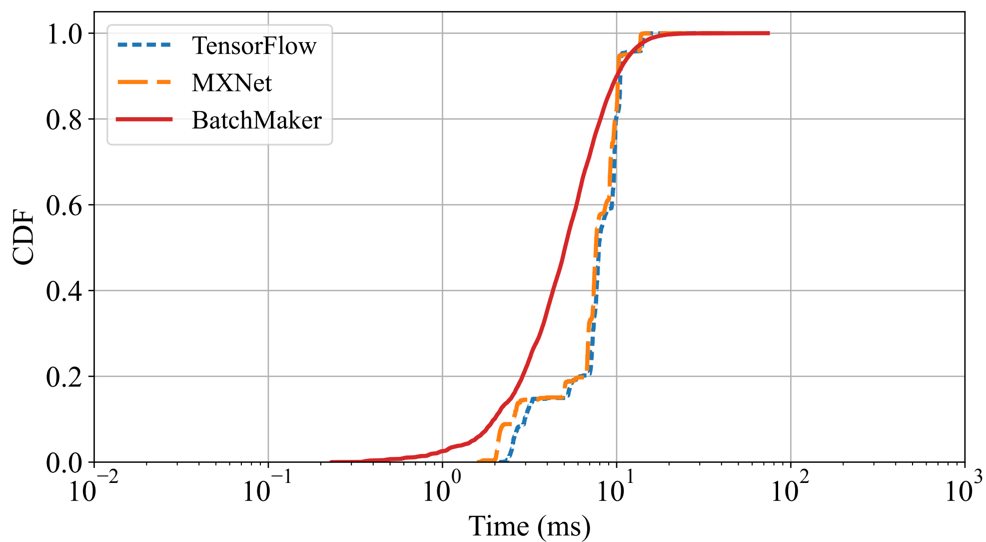
2.7.3 Reasons for the performance gain

We investigate in detail the reasons why BatchMaker achieves better latency and throughput over baseline systems.

Reasons for the latency improvement. BatchMaker reduces the latency of a request in two aspects: 1) it reduces the queuing time by allowing a newly arrived request to join the execution of existing requests. 2) it reduces computation time by allowing shorter requests to be returned



(a) CDF of queuing time



(b) CDF of computation time

Figure 2.9: Request queuing and computation time for LSTM on WMT-15 Europarl dataset under low load (5K req/s).

immediately upon completion without waiting for the longer ones. Figure 2.9 (a) and (b) show the CDF of queuing time and computation time for LSTM on the WMT-15 Europarl dataset. Queuing time is measured from a request's arrival to its start of execution. Computation time is measured from a request's start of execution to the return of the execution result by the system. The lines in Figure 2.9 correspond to the points in Figure 2.7 (a) where the throughputs of all three systems are ~5K req/s. The x-axis is shown in the log scale.

In Figure 2.9(a), the 99-percentile queuing time for BatchMaker is 1.38 milliseconds, compared to $>$ 100 milliseconds for MXNet and TensorFlow. In BatchMaker under low load, a newly arrived request waits for the current set of tasks to finish before joining the execution of an existing batch of requests. With the input load 5K req/s, we find that BatchMaker executes LSTM cells with batch size 64 most of the time, which takes takes about 185 microseconds in microbenchmarks (Figure 2.3 bottom). Due to scheduling and gathering overhead, BatchMaker needs about 250 microseconds to execute an LSTM step. Since BatchMaker submits at most 5 steps of LSTM cell to GPU (by setting the "MaxTasksToSubmit" in Algorithm 1 line 3), the incoming request can wait up to 0.25*5 = 1.25 milliseconds, which roughly matches BatchMaker's 99-percentile queuing time (1.38 ms). The queuing time of MXNet and TensorFlow is much larger; not only an incoming request has to wait for many buckets (out of 33 total buckets) to complete execution, but also each bucket's execution takes as many LSTM steps as the longest sequence in the batch.

As Figure 2.9 (b) shows, the computation time of BatchMaker is also less than that of MXNet and TensorFlow. Bucketing results in CDF lines with "jumps", as sequences with different lengths within the range of a bucket will be padded to the identical length to form a batch and complete their execution at the same time. When the bucket width is set to 10 for MXNet and TensorFlow, a request of length 21 will be padded to length 30, resulting in almost 50% padding overhead and latency increase. By contrast, BatchMaker allows any request that has completed its execution to be returned immediately, with the tradeoff of having to incur scheduling and gathering overhead in the middle of a request execution.

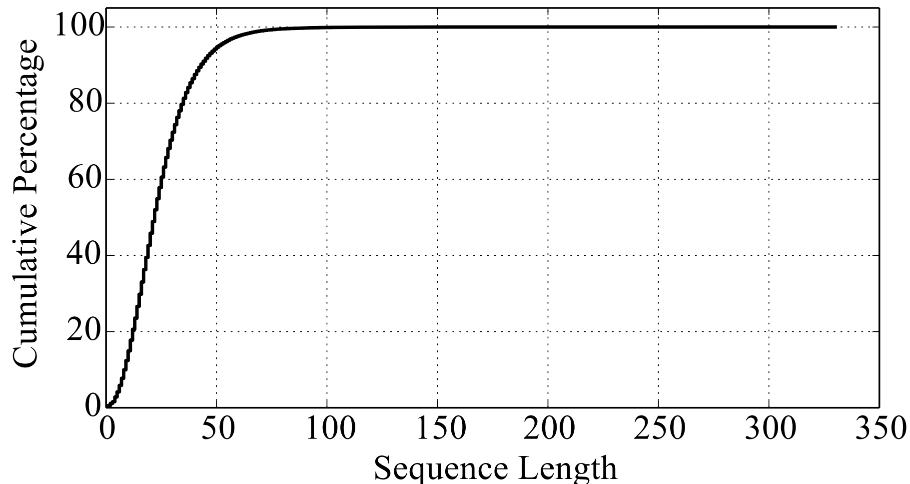


Figure 2.10: CDF of the sequence length in the WMT-15 Europarl dataset.

Comparing Figure 2.9(a) and (b), we can see that reduced queuing time is the more dominant factor for BatchMaker's latency improvement.

The impact of variable-length sequences. We examine how BatchMaker and baseline systems perform on artificial datasets with different variances of sequence lengths. Figure 2.10 shows the sequence length CDF of the WMT-15 Europarl dataset. We can see that about 99 percent of sequences have a length of less than 100. And the longest sequence length is 330. We generate an artificial dataset with a fixed sequence length of 24, which is the average sequence length of the WMT-15 dataset. Additionally, we sample two different datasets with different variances in sequence length from the WMT-15 dataset by clipping the maximum sequence length to be no longer than 50, and 100 respectively.

Figure 2.11 shows the performance of different systems under fixed-length inputs and inputs with a maximum length of 50 and 100. We can see that increasing the variance of input length causes the latency and throughput of baseline systems to get much worse. The increase in latency is due to requests waiting for more buckets as inputs with higher variance in length use more buckets. The decrease in throughput is due to baseline systems executing with smaller effective

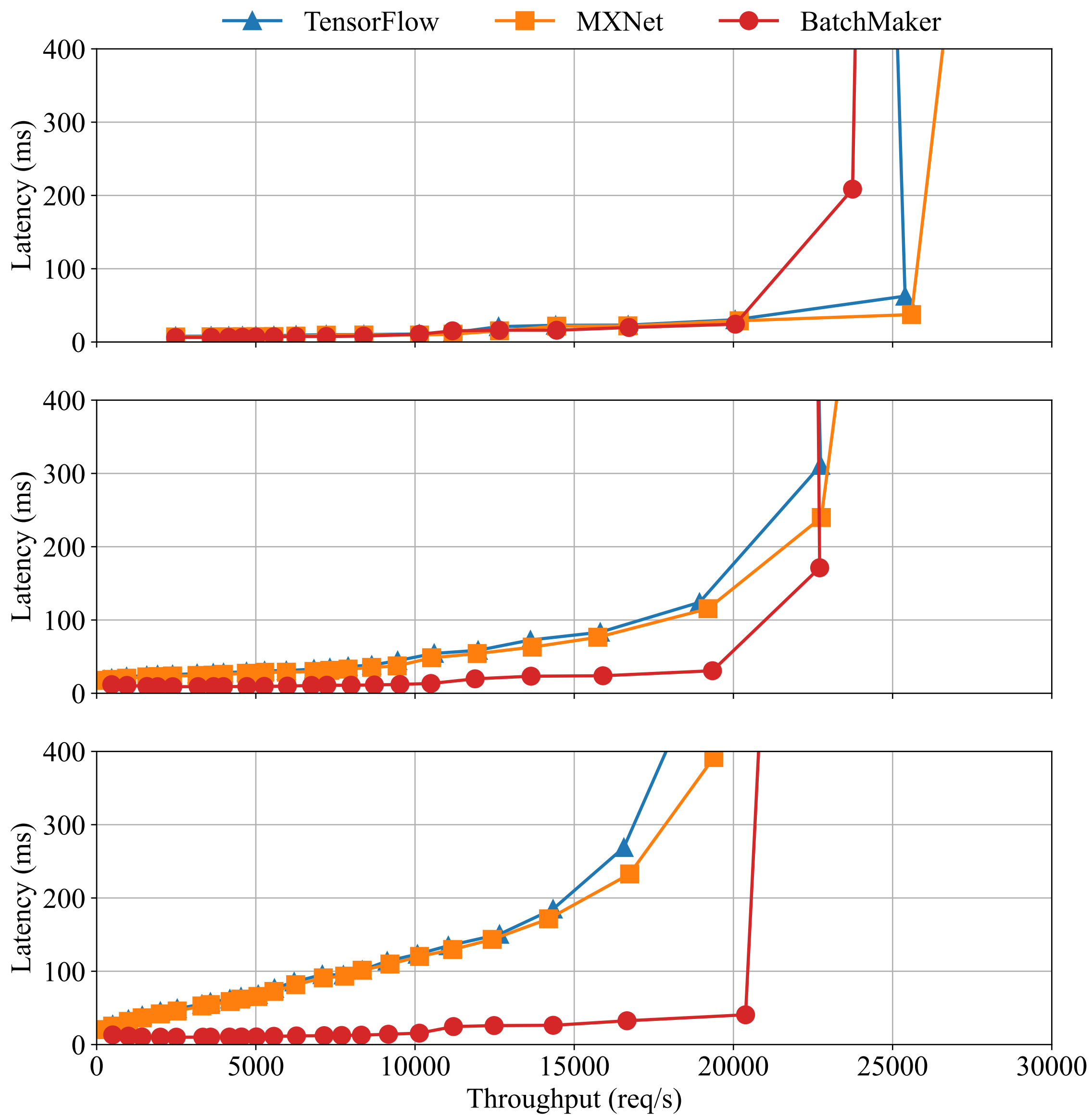


Figure 2.11: Performance under different sequence length variations. From top to bottom, the experiments use an artificial dataset of identical sequence length (24), a sampled WMT-15 dataset with maximum sequence length 50, and another one with a maximum sequence length of 100.

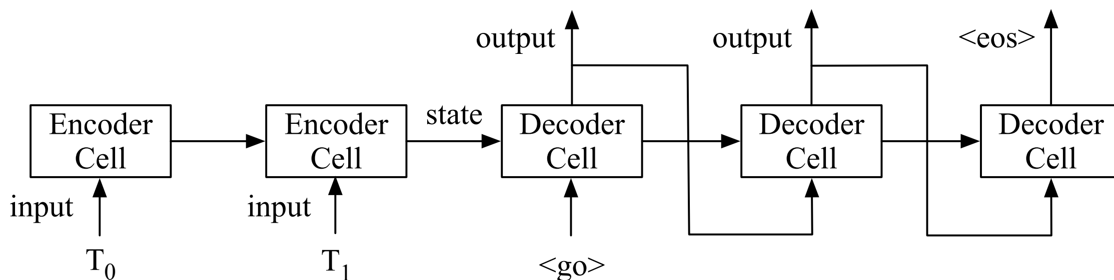


Figure 2.12: Seq2Seq with "feed previous" decoder

batch sizes when inputs with higher variance in length are spread across more buckets. By contrast, BatchMaker can maintain the same latency under low to moderate load despite increased input length variance.

Using the fixed-length artificial dataset, the baseline systems achieve better throughput than BatchMaker. Under the high load, baseline systems can form a full batch of 512 fixed-length inputs. As the execution time of one LSTM cell is approximately 784 microseconds for the batch size 512, one can execute at most $\frac{1}{784*10^{-6}*24} = 53$ batches per second for inputs with length 24. Thus, the maximum system throughput is about 27136 req/s, which is closely matched by those of the baseline systems. By contrast, the throughput of BatchMaker is about 87% of the maximum throughput, due to the overhead of scheduling and gathering. Although it is hard to see from the figure, BatchMaker still achieves better latency than baseline systems under low load by allowing new requests to join the execution of currently executing ones.

2.7.4 Application Performance: Sequence to Sequence

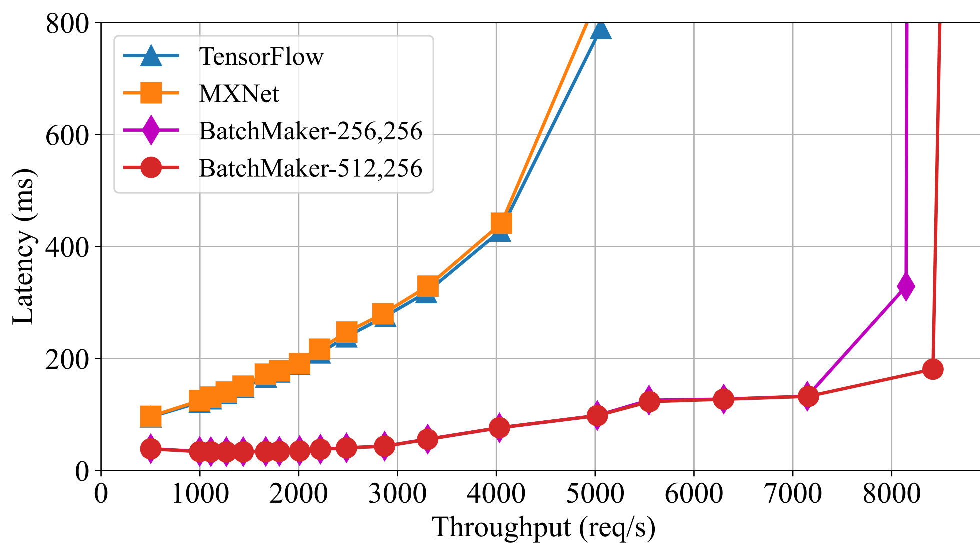
Background on Seq2Seq. Sequence to Sequence (Seq2Seq as an abbreviation) is a widely used RNN model in machine translation. A basic Seq2Seq model contains two types of RNN cells: encoder and decoder, as depicted in Figure 2.12. The encoder takes in a sequence of words as input. In each step, the encoder and decoder will convert a word to a vector by doing an embedding

lookup, and then feed this vector to an RNN Cell. The first decoder cell takes in the output state of the encoder and a $< go >$ symbol as input and computes states, which are passed to the succeeding decoder cell. In addition to the state, the decoder cell outputs a word as well, which is obtained by applying a linear transformation and an argmax[3] [22]. The output word is also fed to the next step as the input. When the decoder outputs the <*eos*> symbol, it means the decoder has finished, and there will be no more decoder steps.

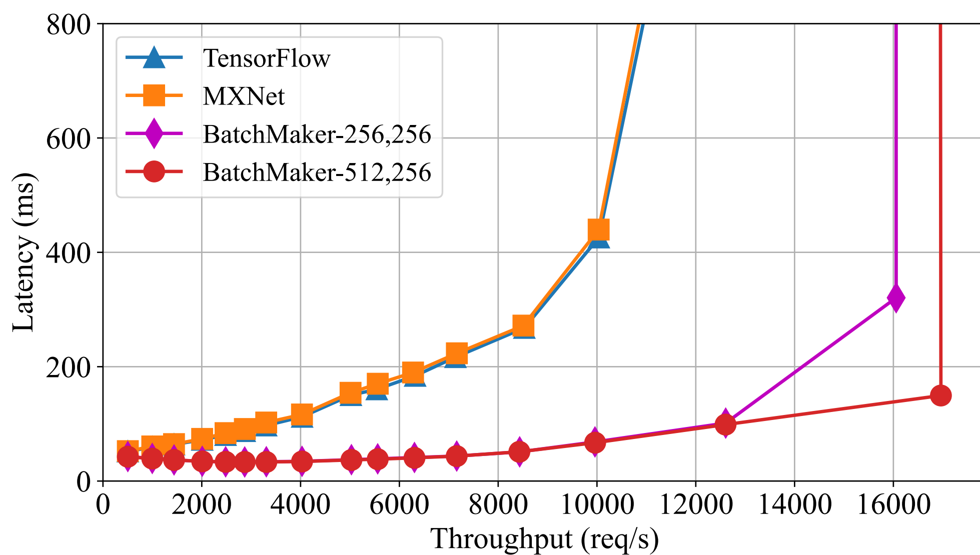
In our evaluation, we use LSTM as the RNN cell. Encoder and decoder cells do not share weights. We use the sampled WMT-15 Europarl German to English translation dataset with a vocabulary size of 30K. When doing inference using Seq2Seq, the decoded sequence length is not known a priori. Deployed systems typically configure the maximum decoding length to be the input sequence length plus a fixed threshold of extra steps. Decoding terminates when either the <*eos*> symbol is generated or when the maximum decoding length is reached. For simplicity, in our Seq2Seq experiments, for a given input German sentence, we decode until the number of steps is equal to the corresponding English sequence length. We do not use the knowledge of decoding length in any of our batching or scheduling decisions.

Batching and bucketing configuration. The Seq2Seq model is different from LSTM in that it has two phases and the computation is very unbalanced. The decoding phase constitutes about 75% of the entire computation due to performing the output projection from the hidden dimension to the vocabulary dimension, which contains a large matrix multiplication. Our microbenchmarks show that batch size 256 is the best for decoder cells while 512 remains the best for encoder cells. Since graph batching requires that all operators in the dataflow graph use the same batch size, we use $b_{max} = 256$ for MXNet and TensorFlow to optimize for decoder performance. As BatchMaker supports different batch sizes for different cells, we evaluate two configurations; one using $b_{max} = 256$ for both encoders and decoders, and the other using $b_{max} = 512$ for encoders

[3] Argmax operator is not optimized in MXNet and TensorFlow, resulting in unacceptably slow performance. We implemented an optimized argmax CUDA kernel for all systems.



(a) Seq2Seq on 2 GPUs



(b) Seq2Seq on 4 GPUs

Figure 2.13: Performance of Seq2Seq on the WMT-15 Europarl German-to-English dataset using 2 and 4 GPUs. BatchMaker-x,y denotes configuring our system to use maximum batch size x for the encoder and y for the decoder. TensorFlow and MXNet use a maximum batch size of 256 and a bucket width of 10.

and b_{max} = 256 for decoders. We have also evaluated different bucketing choices for the baseline systems and found that using the bucket width of 10 produces the best performance for baseline systems.

Multi-GPU performance. In the presence of more than one type of cells, BatchMaker can make more interesting scheduling choices when there are multiple GPU devices. Figure 2.13 shows the performance of various systems using 2 or 4 GPUs. Compared to baseline systems, the peak throughput of BatchMaker is much higher at around 8.5K req/s for 2 GPUs and 17K req/s for 4 GPUs. The latency of BatchMaker is also much lower; it is mostly flat at the beginning and goes up slowly until the throughput reaches the peak. By comparison, the latency of other systems goes up quickly as the load increases. We do not repeat the detailed performance breakdown analysis here as in section 2.7.3. One interesting feature of BatchMaker worth pointing out is that a request can leave the encoding phase sooner and also commence the execution of decoding earlier than baseline systems, thereby magnifying BatchMaker's performance improvement.

Configuring different b_{max} for encoding and decoding cells results in a small throughput improvement for BatchMaker (3.5-6%). Although using b_{max} = 512 improves the throughput of LSTM encoding cell execution by 20% (Figure 2.3), the overall throughput improvement is much less because the encoding phase constitutes only 25% of the overall computation.

2.7.5 Application Performance: TreeLSTM

As padding does not support batching for non-sequential inputs, we compare against TensorFlow Fold and DyNet for the TreeLSTM experiments.

We use the popular Stanford TreeBank dataset [91]. Although all systems under evaluation can support arbitrary tree structure, the TreeBank dataset [91] contains only binary tree samples.

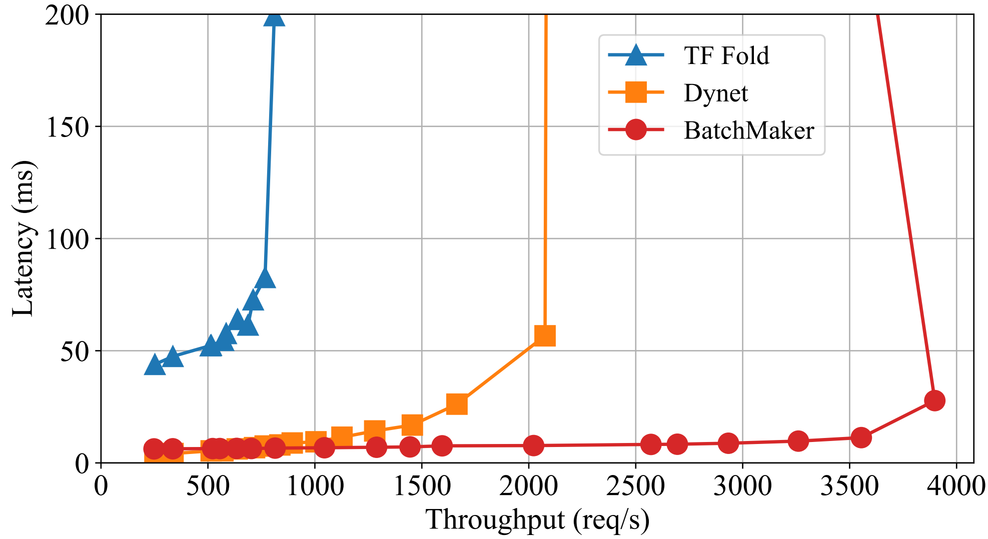


Figure 2.14: TreeLSTM performance on the TreeBank dataset with a maximum batch size 64.

Baseline system configuration and optimization. We perform microbenchmarks using various input batch sizes and find that batching at most 64 input trees achieves the best performance for TensorFlow Fold and DyNet. We note that the batch size configured for DyNet and TensorFlow Fold bounds the maximum number of input trees rather than the number of operators merged into a single batched operator. For instance, even if a batch only contains one request which is a complete binary tree with 16 leaf nodes, TensorFlow Fold can concatenate all 16 leaf nodes together and execute the leaf TreeLSTM cell at batch size 16. It can execute the level above the leaf layer with batch size 8, and so on with the root level executed with batch size 1. As this example shows, the amount of batching decreases at higher levels of the trees. To be fair to baseline systems, BatchMaker is also configured to limit the number of batched cells in a task to 64.

TensorFlow Fold and DyNet perform batching by first generating the dataflow graph for each request and then merging the dataflow graphs. For TensorFlow Fold, this step takes much longer than performing the actual computation. We optimize TensorFlow Fold by overlapping its graph

construction/merging with actual execution[4]. We did not implement similar optimization for DyNet because of its code complexity. We note that DyNet's graph construction/merging overhead is much smaller than that of TensorFlow Fold,

TensorFlow Fold does not work with the latest TensorFlow version (v1.4) and is only compatible with v1.0[5]. Hence, we evaluate TensorFlow Fold using TensorFlow v1.0 and CUDA 8.0[6]. To see the performance disadvantage of using older versions, we conduct microbenchmarks on a single LSTM step using both versions and find that using the older versions (TensorFlow v1.0 and CUDA 8.0) has a slowdown of about 20%.

Performance on the TreeBank dataset. Figure 2.14 shows the latency vs. throughput for TreeLSTM on the TreeBank dataset. Due to its slow graph construction and merging, the throughput and latency of TensorFlow Fold are much worse than DyNet. Under moderate load (at 1K req/s), the 90-percentile latency of BatchMaker is 6.8ms, compared to 9.5ms for DyNet. BatchMaker achieves much better peak throughput than DyNet (3.1K req/s vs. 2.1K req/s). The throughput difference is due to DyNet's overhead in performing runtime dataflow graph merging and insufficient amount of batching at the higher levels of the trees.

Performance on a synthetic dataset of identical input trees. How does the variance in input tree structures contribute to BatchMaker's performance improvement? To understand this, we conduct experiments using a fake dataset whose input requests have an identical tree structure (a complete binary tree of 16 leaf nodes). We implement an ideal baseline system by hardcoding in TensorFlow a dataflow graph matching the fixed binary tree structure. Each node in this dataflow graph can execute up to 64 corresponding operations, one for each input in a batch size of 64. We evaluate the performance of all systems including the ideal baseline using the fixed tree dataset.

[4]The overlapping is not perfect due to Python's poor multi-threading support.
[5]https://github.com/tensorflow/fold/issues/57
[6]TensorFlow v1.0 does not support CUDA 9

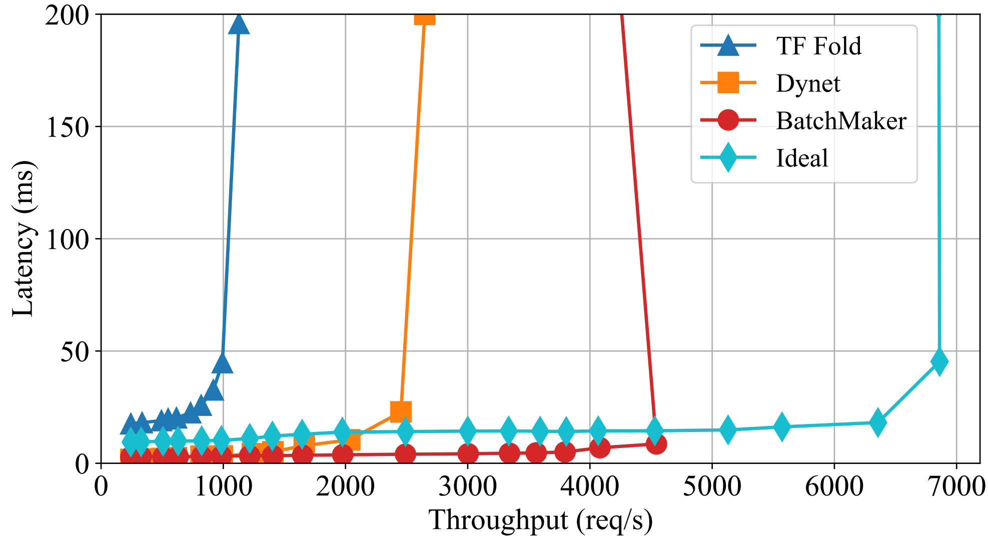


Figure 2.15: TreeLSTM performance on a synthetic dataset where each sample has the identical binary tree structure. The *ideal* line represents the ideal performance of executing these identical samples in a maximum batch size of 64.

The results are shown in Figure 2.15. As the figure shows, the peak throughput of BatchMaker is approximately 30% less than that of the ideal baseline. Note that the ideal baseline's latency is higher than that of BatchMaker and DyNet. This is because the ideal baseline executes a series of 31 TreeLSTM cells for a batch of inputs. By comparison, DyNet can batch cells within a request together if they are at the same tree depth; BatchMaker can additionally batch cells from different requests together if they arrive at different times.

2.8 Related Work

Batching via padding. Theano [9], Caffe [52], TensorFlow [1], MXNet [14], Torch [18], PyTorch [78] and CNTK [23] are widely used deep learning frameworks. Theano, TensorFlow, MXNet, and CNTK require users to build a static dataflow graph before training or inference. PyTorch is more imperative and allows the computation graph to be built dynamically as execution happens [96]. Gluon [19] is a recent package for MXNet supporting dynamic computation

graphs. When handling variable-sized inputs, all of these systems support batching via padding. CNTK [32] additionally, introduce an optimization on padding that tries to fill up padded space with shorter requests. Doing so can improve system throughput by reducing the amount of wasted computation due to padding. As we mentioned earlier, padding does not work for non-chain-structured RNNs such as the TreeLSTM. Therefore, these systems do not natively support batching for the TreeLSTM.

Batching by merging dataflow graphs dynamically. As non-chain-structured RNNs such as TreeLSTM become popular, TensorFlow Fold [58] and DyNet [67] are developed to support batching for TreeLSTMs. Both systems use a similar approach to batch TreeLSTMs (called Dynamic Batching and on-the-fly batching [68] respectively). They first generate a dataflow graph for each data sample and then attempt to merge all dataflow graphs into one graph by combining nodes corresponding to the same operation while maintaining the data dependency. Graph batching allows both systems to support the batched execution of variable computation graphs without padding, including TreeLSTM. The difference between them is that TensorFlow Fold, like our system, batches at the granularity of a cell whereas DyNet batches at the granularity of a single dataflow operator. Both TensorFlow Fold and DyNet try to batch a fixed set of dataflow graphs at a given time. By contrast, BatchMaker batches at the cell level and allows a request to dynamically join and leave the ongoing requests.

Systems specialized for inference. Several frameworks address the challenges during inference. LASER [2] and Velox [26] are systems that focus on optimizing the training and serving pipeline for traditional machine learning models such as logistic regression and matrix factorization. LASER and Velox address issues such as how to re-train models quickly upon observing additional data during deployment, how to balance exploration vs. exploitation to gain useful feedback while maintaining a good user experience. These issues are orthogonal to the prob-

lem addressed by our system, namely, how to reduce the latency of batched execution for RNN inference. Clipper [28] is a general-purpose serving system that supports a variety of machine learning system backends, such as TensorFlow, Spark MLlib [61], and Caffe [52]. It uses existing batching techniques to achieve good throughput and additionally perform dynamic batch size adjustment to match requests' latency objective.

TensorFlow-Serving [74], a system developed for serving TensorFlow models, introduces "Batch" and "Unbatch" operators to TensorFlow's dataflow graphs. It is claimed that these operators "bear similarities to the batching approach of DyNet" [74]. The current implementation for these operators is not yet ready for deployment and thus we have not compared BatchMaker with TensorFlow-Serving.

Optimization for deep learning computation. TensorRT [24] is a deep learning inference optimizer and runtime for deep learning applications. For a given neural network, its optimizer and runtime will generate a fused kernel to reduce the kernel launch overhead and memory footprint. TensorFlow XLA [21] is a domain-specific compiler that optimizes TensorFlow computation. It also applies optimization like kernel fusion to reduce the kernel launch overhead. These optimizations are orthogonal to our work and can be applied to optimize the execution of each cell in our system. Persistent RNN [36] exploits the weight-sharing feature of RNN to accelerate the computation. In particular, persistent RNN store weights in the on-chip memory of the GPU (e.g. register files) and reuse them across many timesteps. Doing so avoids loading weights from the global memory (which is much more expensive than on-chip memory) at each time step. Weight persistence reduces but does not eliminate the need for batching. Thus, it is complementary to the cellular batching; both can be used together for additional performance improvements.

Batching and pipelined execution in other systems. In database systems, one query may contain many operators like *scan, join, sort, aggregate,* etc. Instead of executing independent queries separately, there are many systems[12, 40, 45, 59, 81, 100] that batch the execution of certain operators from different queries. Since the results of these operators can be reused for different queries, the database throughput can be improved. However, multi-query batching in database systems does not always improve performance, e.g. when the load is light and there is little or no overlap in the data processed by different queries. By contrast, the recursive nature of RNNs results in completed overlapped computation across different inputs, and we leverage this feature to guarantee improved latency and throughput in all scenarios. Additionally, a key feature of the cellular batching is to enable a new request to join the execution of existing requests, which is not done in multi-query batching in the database.

Compared with traditional pipelining (in the context of hardware pipelining and software pipelining, e.g. SEDA[105]), our system is different in two aspects: 1) Traditional pipelining involves multiple processing elements each of which is sequential and operates independently of each other. By contrast, each GPU device represents a single processing element that is massively parallel and thus is best utilized using a kernel that performs batched execution. Cellular batching improves latency by allowing new requests to dynamically join existing requests in a series of batched kernel executions. 2) Different processing elements in a hardware pipeline have different functionalities. In our setting, different GPU devices have the same functionalities and can be used interchangeably. Therefore, instead of dictating a fixed pipelined path of execution across different GPU devices, it is better for performance and load balancing to use a general task scheduler to assign kernels to different GPUs, as is done in BatchMaker.

2.9 Applicability

BatchMaker's techniques can be generalized beyond Recurrent Neural Networks. In fact, the idea of making fine-grained batching decisions at each iteration step and batching requests at different steps naturally applies to any model that involves an iterative process as long as model weights are fixed and shared across different steps.

Notably, follow-up works like [111] have applied BatchMaker's solution to Generative Pre-trained Transformer (GPT) models which iteratively output the next token in the auto-regressive generation phase. The most important difference between RNNs and Transformer-based models is that an RNN model only uses the output from the preceding step whereas the attention module in Transformer models takes in hidden representations of all previous steps. Such an extension can be achieved by properly managing states of the full context during incremental decoding as in [103] and utilizing an attention implementation that supports processing a batch of requests with different context sizes (e.g. FlashAttention [30], MemoryEfficient Attention [82]).

Nowadays, this fine-grained batching technique is commonly known as in-flight batching or continuous batching, and is widely adopted in most Large Language Models (LLMs) serving systems like TensorRT-LLM [73], DeepSpeed-FastGen [47], etc.

2.10 Summary

To sum up, we present a novel approach, called cellular batching, to achieve low-latency inference for language models. Cellular batching batches the execution of inference requests at the granularity of a cell, which can be an RNN cell or a Transformer layer. Doing so allows a new request to join the execution of a batch of existing requests and to be returned as soon as its computation finishes without waiting for others in the batch to complete. We have built a serving system called BatchMaker using cellular batching.

Experiments with three popular RNN applications using real-world datasets show that BatchMaker reduces latency by 17.5-90.5% and improves throughput by 25-80% compared with state-of-the-art systems including TensorFlow, MXNet, TensorFlow Fold, and DyNet.

3 Pensieve

3.1 Overview

It is well known that Large Language Models (LLMs) require very expensive computation because LLMs have huge parameter sizes (10s or 100s of billions) with a trend of growing even larger, and because LLMs need to support a large context size (2K to 32K tokens) to be useful. There has been related work which improves the performance of LLM serving from various angles, including better batching [111], operation fusion [30], better GPU memory utilization [54], faster output generation [13, 55], low rank adaptation [48] and quantization [33] (see §3.7).

In contrast to these works, we take a step back and examine inefficiencies that arise in the context of a specific but very popular LLM use case today, aka as a multi-turn conversational chatbot. Naturally, a chatbot should be able to memorize the conversation and generate coherent responses. However, such burden is placed on the user (or a client application), which needs to keep track of the conversation history, concatenate it with a new user prompt, and send it to the chatbot as the new request. This straightforward stateless approach has severe performance issues: the chatbot has to reprocess the entire history over and over again, whose lengths grow as the conversation goes on.

In this chapter, we present Pensieve, a serving system designed for the the scenario of multi-turn conversation by providing efficient context management for Large Language Models.

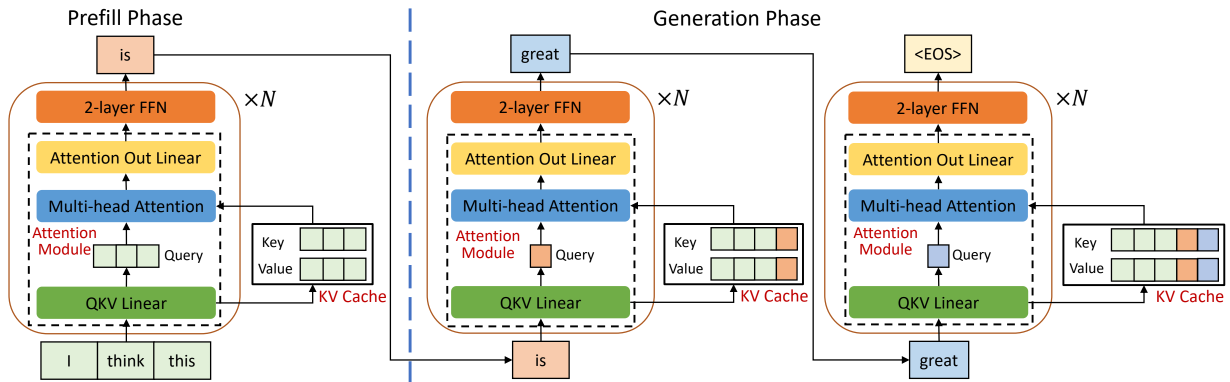


Figure 3.1: Inference of Transformer-based Large Language Models

3.2 Background

In this section, we provide a brief background on LLM, its core attention operation, and how existing systems serve LLMs.

3.2.1 LLM and the Attention Mechanism

Popular large language models, e.g. GPT-3 [10], OPT [113], Llama [97, 98], are all based on the Transformer architecture [101]. A model consists of many transformer layers, each of which is composed of an attention module, seen as the dashed box Figure 3.1, and a 2-layer feed-forward network. The model takes as input a sequence of token IDs representing the natural language sentence and feeds them through an embedding layer to obtain a continuous representation (aka embedding) for each token before feeding them through transformer layers. For simplicity, we refer to token embeddings as "tokens", and refer to token IDs as "raw tokens". LLM is autoregressive in the sense that it iteratively predicts the next output token based on the current context which includes the input prompt tokens followed by output tokens generated in the previous iterations.

The success of the Transformer originates from the capability of its attention module. For each

layer, the attention module first performs linear transformations on its input token embedding (X) to produce three new embeddings Query (Q), Key (K), and Value (V):

$$
\begin{aligned}
Q &= XW_{query} \\
K &= XW_{key} \\
V &= XW_{value}
\end{aligned}
\tag{3.1}
$$

where W_{query}, W_{key} and W_{value} are trainable weights. We refer to the K and V embeddings as KV-tokens. The module then computes all-to-all attention scores using the dot product between each pair of tokens' query and key:

$$
A = \frac{QK^T}{scale} \tag{3.2}
$$

where scale is a normalization factor. This attention score is then normalized using softmax and used for weighted aggregation of the token embeddings in V:

$$
O = softmax(A)V \tag{3.3}
$$

The above equations show the process of so-called single-head attention. In practice, multi-head attention is used, where Q, K, V produced by Equation (3.1) are divided into groups, called attention heads. Each attention head independently performs attention as Equation (3.2) and (3.3).

3.2.2 How LLM is Served

The prefill vs. generation phase To perform inference using an LLM, one needs to keep a KV cache in GPU to avoid recomputation during the autoregressive output generation. Figure 3.1 shows the typical LLM inference process adopted by systems like FasterTransformer [72], ORCA [111], vLLM [54]. It is divided into two phases: 1) The prefill phase processes all the in-

put prompt tokens together to generate the K and V token embeddings (aka KV-tokens) for each Transformer layer and initialize the KV cache with the resulting KV-tokens. The embedding of the last token of the prompt from the last layer's O tensor is used to generate the first output token; 2) The iterative generation phase takes in the token generated by the last step as a single new input token. For each layer, it computes the Q,K,V embedding for the new token, updates the KV cache, and performs attention using the new token's Q embedding with the entire context's KV-tokens.

ITERATION-LEVEL BATCHING For LLMs that have variable-sized input and output, batching granularity has a huge impact on system throughput and serving latency. If scheduling is performed at the request granularity, executing a batch of requests with different input prompt lengths requires padding tensors to the maximum length and waiting for the request with the longest output to finish. Iteration-level batching strategy, originally proposed by BatchMaker [38] for non-transformer-based sequence-to-sequence models, performs batching at token granularity. Whenever a request finishes an iterative generation step, the scheduler checks whether it has reached the end of a sequence and can leave the batch, making room for a request to start its computation immediately. As we advance into the Transformer era, ORCA [111] extends this approach to support the LLM workload.

MEMORY MANAGEMENT. For each request, the model performs iterative generation until either the special end-of-sentence token (EOS) is emitted or the preconfigured maximum decoding length is reached. Systems like FasterTransformer [72] and ORCA [111] reserve slots in KV cache for each request based on the maximum decoding size. A more recent system, vLLM [54], can dynamically grow the allocated cache slots for each request and allow these slots to reside in non-contiguous GPU memory. vLLM develops PagedAttention to handle the generation phase with non-contiguous KV cache. Existing serving systems are stateless across requests. In other

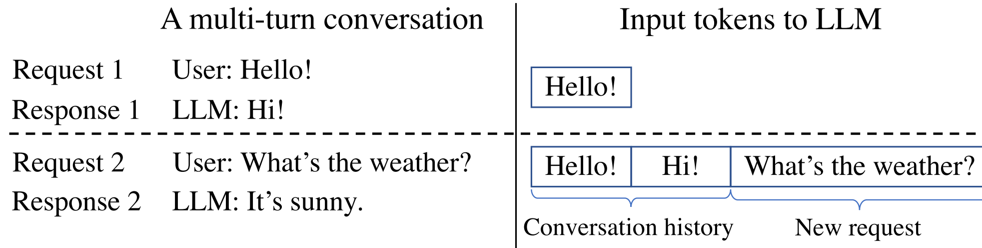


Figure 3.2: Existing serving systems send a cumulative history with each request in a multi-turn conversation.

words, they de-allocate all the cache slots used by a request as soon as it finishes.

3.3 Motivation and Challenges

3.3.1 Motivation

Existing techniques for serving Large Language Models mostly focus on improving the inference time or memory efficiency of a single request. We take a step back and examine inefficiencies that arise in the context of a specific but very popular LLM use case today, aka as a multi-turn conversational chatbot.

In a multi-turn conversation, the user has multiple rounds of conversations with the chatbot and LLM needs to be aware of the conversation history to generate an appropriate response. This is done by prefixing each new request with the entire cumulative conversation history, due to the stateless nature of existing serving systems, as shown in Figure 3.2. As the interaction between the user and chatbot goes on, conversation history grows, making the cost of the prefill phase soon overshadow that of the iterative generation phase. Unfortunately, much of the history processing is redundant.

The goal of this project is to minimize redundant computation of the conversation history. This can be done by caching any previously processed embeddings at the serving system and re-using them across requests from the same conversation. More concretely, one can save the

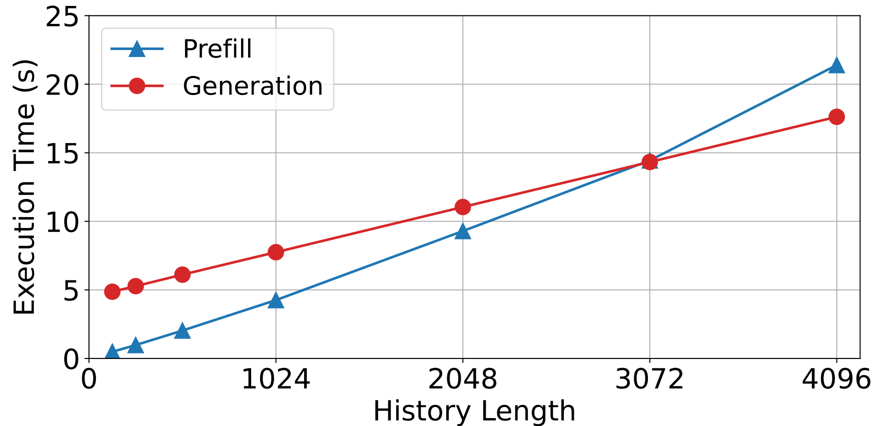


Figure 3.3: Execution time for a batch of 32 requests performing prompt (32 tokens) prefill and generations for 200 steps.

KV-tokens in the KV cache belonging to a previous request and only process the new prompts for the next follow-up request.

Figure 3.3 demonstrates the heavy cost of prompt initiation phase under an artificial workload where each request generates 200 tokens and has varying conversation history sizes. As shown in the figure, the cost of recomputing the conversation history (solid blue line) causes the cost of the prefill phase to soon outgrow the generation phase.

3.3.2 Challenges

Limited GPU memory for caching. LLM has very large model parameters, which results in large KV-tokens. For example, a 13 billion parameter GPT-3 model has 40 layers and a hidden size of 5120. Assuming the use of 16-bit half-precision numbers, storing each KV-token takes 2 * 40 (layer) * 5120 (units/layer) * 2 (bytes/unit) = 0.78MB memory space. Under the current GPU memory limit, depending on the history lengths, only a few dozen or hundreds of conversation histories can be kept in the GPU. We must extend our cache space to use the more abundant CPU memory.

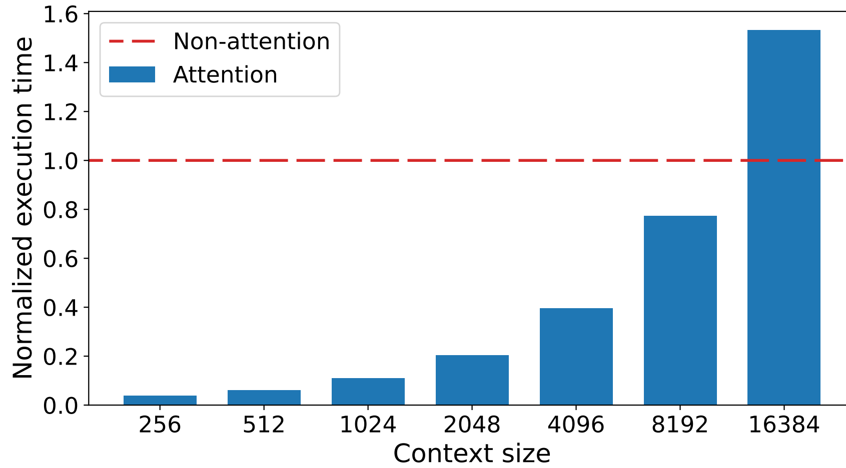


Figure 3.4: Execution time of attention operation for a chunk of 32 tokens against different context sizes. Results are normalized by the execution time of non-attention operations in a Transformer layer

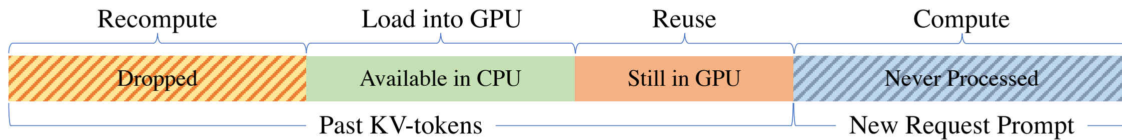


Figure 3.5: Layout of a request's KV-token context. Shaded areas indicate that the actual computation in the prefill phase happens at both ends if KV-tokens were dropped by the CPU cache.

Cache management and recovery When using a multi-tier GPU-CPU cache, the serving system must be able to evict and restore cached data. The system has to make best-effort decisions to swap cached KV-tokens from GPU to CPU to make room for requests from other conversations. Later, it needs to recover the swapped out KV-tokens as fast as possible.

When the system is under CPU memory pressure, some cached KV-tokens need to be dropped. Although each token in the context occupies the same amount of memory space, the recomputation cost of each token is different because reconstruction of the embeddings of a dropped token requires performing attention over all preceding tokens in the context. Therefore, the later a token is, the larger the context it needs to attend to. Figure 3.4 shows the execution time of performing the attention operator versus the context size normalized by the execution time of all other non-attention operators in a Transformer layer. As shown in the figure, the cost of atten-

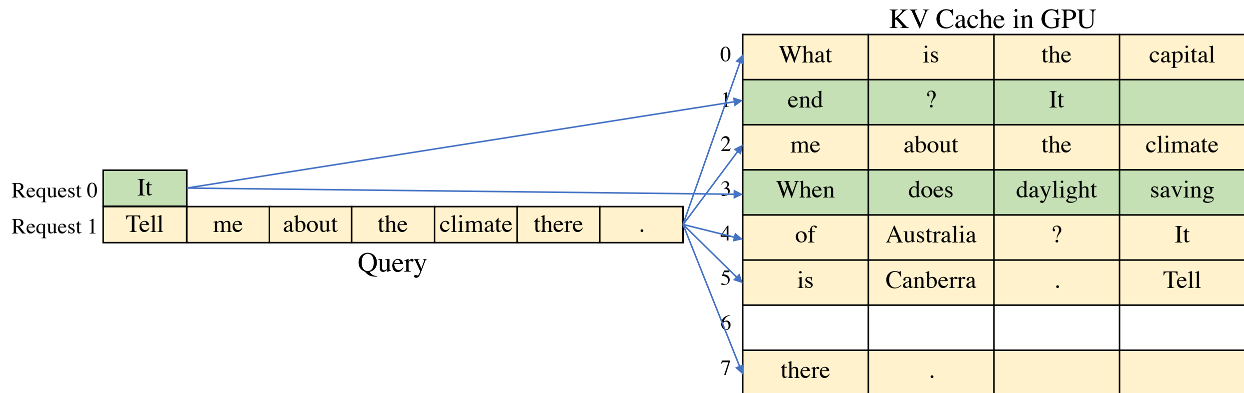


Figure 3.6: Multi-token attention with Non-contiguous Context Cache. The batch contains two requests. The left-hand side shows the Query representation for tokens being processed in each request. The right-hand side shows the context data (Key and Value representation) for each request in GPU.

tion grows linearly with context size. Hence whenever we need to drop tokens to reclaim CPU memory, it is preferable to drop leading tokens of conversations.

However, dropping past KV-tokens from the leading end brings additional complexity. Figure 3.5 illustrates the layout of a request from a resuming conversation in its prefill phase. New prompt tokens are appended to the end of the context and dropped tokens are at the beginning while tokens in between can be directly reused. Such separation of token computation at both ends of the context breaks the assumption of all existing high-performance attention kernels that the input tokens correspond to consecutive context regions.

In short, we need a low-overhead solution to manage memory space and swapping and deal with the case that some parts of the conversation history embeddings are no longer available in the system.

Handling non-contiguous KV cache. In the multi-turn conversation setup, whenever a new request comes in an existing conversation, the request's new prompt, which in all likelihood has more than one input token, needs to be processed along cached KV-tokens as part of the prefill phase. For any cached KV-token that has been previously evicted to the CPU, there is no guarantee they are restored to the GPU in a contiguous memory region next to existing KV-tokens

cached in the GPU. Figure 3.6 illustrates the computation pattern of attention. The batch contains request 0 which is currently performing generation using the last generated token, and request 1 which just arrives with a 7-token new prompt. To perform attention operation, request 0 needs to iterate over tokens in KV cache blocks 3 and 1, and request 1 needs to iterate over tokens in cache blocks 0, 4, 5, 2, and 7. Existing high-performance attention kernels, e.g. MemoryEfficientAttention [82], FlashAttention [30], require inputs to reside in contiguous memory space and therefore cannot work with non-contiguous paged memory management. Although vLLM has developed the PagedAttention GPU kernel, it is designed to be solely used in the generation phase and thus has the limitation that each request in the batch can have at most one input token. Hence, the PagedAttention kernel cannot support prompt initiation for request 1 in Figure 3.6. And due to the design of single-token PagedAttention, no trivial way is available to extend it to support multi-token attention. An alternative approach might be to process the new prompt query in the same iterative manner as incremental decoding, one token at a time so that the PagedAttention kernel can be applied. But this approach gives up the parallelization opportunity available in the prefill phase due to the extra query dimension and will lead to significantly longer computation time for long input prompts.

3.4 System Design

In this section, we describe the system design of Pensieve. At a high level, we aim to save a conversation's KV-tokens across multiple turns, if there is enough cache space to do so. To realize the potential performance benefits, we need to make cache swapping and missing token recomputation fast and efficient, by developing a few techniques to address the challenges in §3.3.

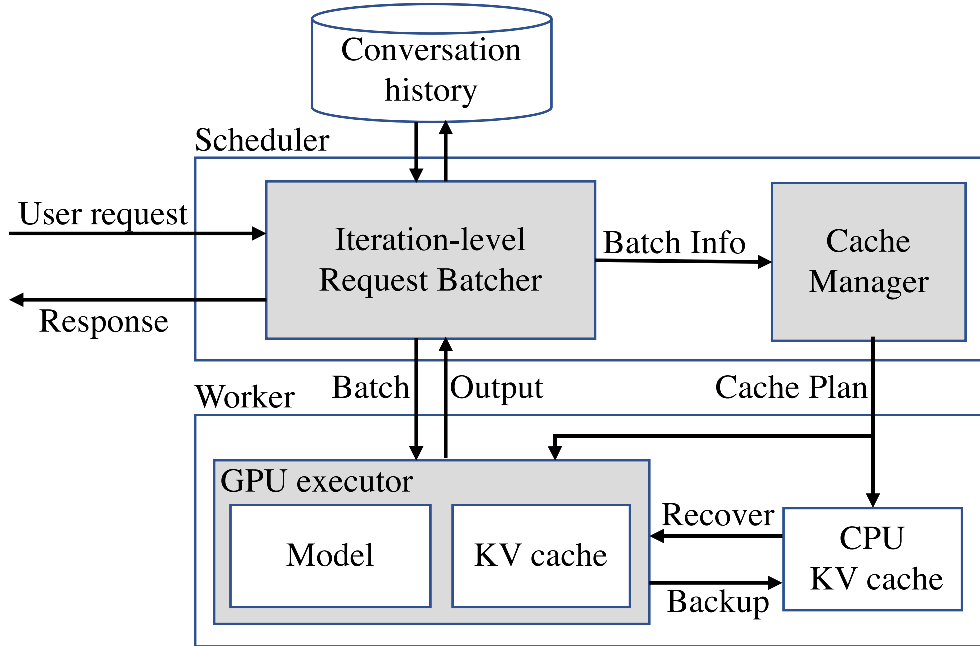


Figure 3.7: Overview of Pensieve

3.4.1 System Overview

Figure 3.7 shows the overall system architecture. Pensieve consists of a single scheduler and a worker engine to manage the GPU. The scheduler has two jobs: 1) it is responsible for batching requests for execution, and 2) it ensures that requests in the batch have sufficient GPU memory for execution. For 1), the scheduler performs fine-grained iteration-level batching [38, 54, 111] so that a new request can join the batch with existing requests while the latter is in the process of performing autoregressive generation. For 2), the scheduler manages the allocation of slots in the KV cache and when to swap between the GPU and CPU KV cache.

The worker also has two jobs: 1) it invokes kernels to execute the LLM computation for a batch of requests. 2) it performs the actual data movements between the GPU and CPU KV cache based on a batch's cache plan received from the scheduler.

3.4.2 A unified batch scheduler

Pensieve performs fine-grained iteration-level batching [38, 54, 111]. However, our batching strategy differs from existing LLM serving systems. ORCA only batches non-attention operations and handles attention operations for each request individually [111]. By contrast, Pensieve batches both non-attention and attention operations, like vLLM [54]. However, unlike vLLM which only forms a batch among requests in the same phase and thus processes each of the two types of batches in a separate model execution, Pensieve handles both the prefill phase and generation phase in a unified way. In other words, the Pensieve scheduler forms a batch of requests regardless of which phase they are in. In the same batch, some could be in their generation phase while others are in their prefill phase. Such unified batching is made possible by Pensieve's multi-token attention kernel design (§3.4.4.1).

Pensieve's iteration-level scheduler is "clocked" for action by the completion of a generation step. In particular, after each iteration of token generation, the worker returns any finished request that has either emitted the end-of-sentence token or reached maximum decoding length. The scheduler finds the next request in its wait queue to join the batch, if there is room (i.e. the number of total tokens in the batch is fewer than some pre-configured threshold). We use the simple first-come-first-serve scheduling policy when choosing which new requests to join the batch.

For unified batch formation, the scheduler concatenates the tokens to be processed from all requests. Each new request that has just joined the batch has a sequence of tokens corresponding to its prompt. Each existing request in the batch has a single token corresponding to the last generation step's output token. By combining the prefill phase together with the generation phase, we avoid running separate small kernels and can thus improve GPU utilization.

3.4.3 KV cache management

Traditionally, the KV cache in GPU only serves as a computation workspace. In Pensieve, the GPU KV cache also serves as storage space to cache the KV-tokens of recently completed requests of active conversations. Pensieve adopts a two-tier hierarchical caching strategy and uses the much larger CPU memory as the second-tier cache space. The scheduler determines the caching plan and instructs the worker to perform the actual data movements between GPU and CPU.

The scheduler tracks the amount of free GPU KV cache slots left. Before handing off a batch of requests to the worker, the scheduler tries to ensure that any new request's past KV-tokens will reside in the GPU and there is sufficient GPU memory for the execution. Specifically, suppose a request in the batch has any KV-tokens that have been swapped out to the CPU memory or dropped, the scheduler determines how many additional GPU KV cache slots are needed to swap in or re-calculate those absent KV-tokens. If there is sufficient space, the scheduler instructs the worker to perform the necessary allocation and to start swapping those KV-tokens from the CPU as part of the batch's cache plan.

3.4.3.1 Eviction Policy

With the observation from Figure 3.4 that leading tokens of a conversation are cheaper to re-compute than trailing ones, we design a fine-grained eviction policy that performs eviction from the leading end of a conservation context. However, recomputation cost alone is not sufficient to make the cache system perform well. We have no prior knowledge about whether a conversation will continue after its current request is served, and if it does, when its next request prompt will arrive. If the tokens saved are never used or only to be used in the remote future, keeping them reduces the valuable space we could use to store other tokens. Therefore, we need to make our best bet to assess the overall merit of saving a token in the system. To do so, we define a function to determine the value of tokens based on their recomputation cost, the time passed since the

conversation it belongs to was last active, and the probability of receiving another request from the same conversation after seeing a certain number of turns. We evict those with the lowest value to the CPU and drop them when necessary.

Eviction granularity. To reduce the overhead caused by frequent eviction decision-making and moving a small amount of memory over the PCIe bus, we group KV-tokens into chunks and make eviction decisions at the granularity of chunks. This chunk size is configurable in our system, but in practice, we find that a chunk size of 32 tokens works well.

Value function of a chunk. We define the following value function to capture the above-mentioned factors and determine the value of a chunk.

$$Value = \frac{Cost(s, l)}{T} \tag{3.4}$$

$Cost(s, l)$ represents the cost of recomputing a chunk of size s with a context of size l. The denominator T is the amount of time that has past since the conversation was last active, which resembles the classic Least Recently Used (LRU) eviction policy emphasizing temporal locality. We assume that our users who are actively interacting with the chatbot tend to send in their next requests within a short amount of time, and therefore longer time past is an indicator that the conversation is less likely to resume in a close future, which reduces the merit of keeping the chunk. Pensieve uses Equation (3.4) to determine the value all chunks and prioritize saving those with expensive costs to recover and those from recently active conversations.

Estimating recomputation cost. We view the cost to reconstruct the embedding of a chunk as the sum of recomputing the LLM model's attention component and non-attention ones.

$$Cost(s, l) = Cost_{attention}(s, l) + Cost_{other}(s) \tag{3.5}$$

The non-attention part consists of linear layers, layer normalizations, non-linear activations, etc., and therefore is independent of the context size. On the contrary, attention requires accessing and performing computations with all l context tokens. We profile these two parts offline with chunk size $s = 32$. However, since it's not feasible to profile all possible context sizes, we only measure context sizes that are powers of 2 and use the measured values to interpolate the cost for other context sizes. And because we make fine-grained eviction decisions on fixed-size chunks of 32 tokens, we can simplify the cost function by treating the cost of non-attention parts as a constant c.

$$Cost(l) = Cost_{attention}(l) + c \tag{3.6}$$

3.4.3.2 Ahead-of-the-time swapping

Since Pensieve tries to preserve KV-tokens in the GPU for reuse across multi-turn conversations, the scheduler does not immediately release a request's GPU cache slots as soon as it finishes, unlike existing systems [54]. Instead of waiting until the GPU cache has run out, the scheduler asks the worker to copy (aka swap out) selected KV-tokens to the CPU if less than a threshold (e.g. 25%) of the GPU cache slots are available. The corresponding GPU memory is reclaimed lazily and is not immediately released until the scheduler later decides to allocate the same slots to another conversation.

When the CPU cache does not have enough space to store the swapped out KV-tokens, the same eviction policy (Equation 3.4) is triggered to decide which KV-tokens to drop. Performing ahead-of-the-time swapping allows the scheduler to overlap the eviction decision-making and the actual memory transfer with GPU computation, thus fully hiding the latency of swapping.

3.4.3.3 Pipelined KV cache recovery.

The scheduler does not wait for a request's KV-tokens to be fully swapped in from the CPU before handing it off to the worker for execution. Rather, we follow the pipelined approach [8]

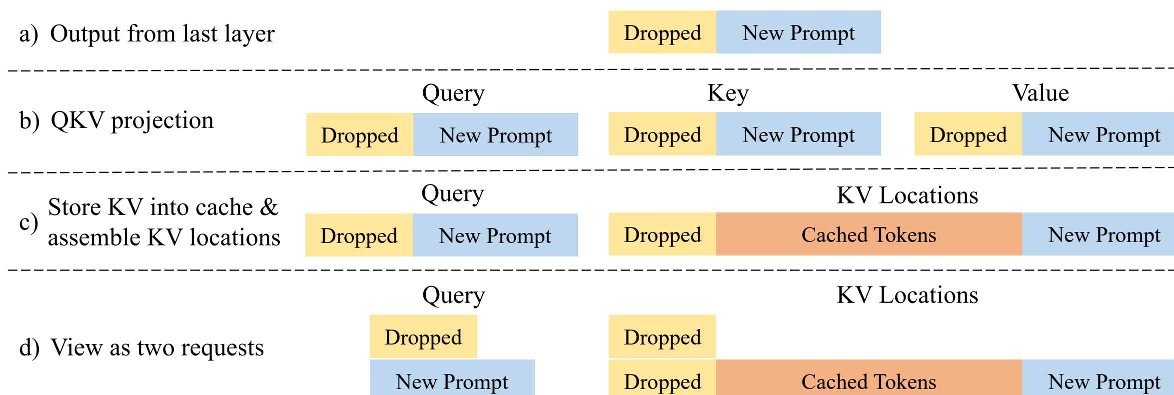


Figure 3.8: Perform attention module for a request whose leading tokens were dropped and needs to prefill both dropped tokens and new prompt tokens. Dropped tokens are prepended to new prompt tokens as they are fed through the model layers, and are viewed as two sub-requests that share the underlying context when performing attention.

to overlap computation with data transfer. Specifically, we exploit the fact that an LLM model has many layers and each layer's KV-token is only used in this layer's self-attention calculation. Instead of waiting for all layers to finish data transfer before starting the execution, we initiate the transfer layer by layer and start model computation at the same time. The worker uses GPU events to preserve data dependency: it only starts a layer's self-attention kernel once that layer's KV-tokens have been fully copied to the GPU. Pipelining transfer with computation allows us to hide the swap-in latency.

3.4.3.4 Handling dropped tokens.

If a request has some of its KV-tokens dropped due to CPU memory pressure, we resort to recomputation to handle such dropped tokens. Figure 3.5 illustrates the most general scenario for the whereabouts of a new request's KV-tokens. As we always evict cached tokens from the leading end of a conversation, the GPU cache generally holds the request's last tokens and the CPU cache holds the middle, while the rest may have been dropped.

The scheduler swaps in those tokens cached in the CPU. For dropped tokens, the scheduler will fetch their corresponding raw text tokens from the conversation history saved in a persistent

store (Figure 3.7). These retrieved raw tokens will be merged into (i.e. prepended to) the new request's prompt and become part of the batch's input tokens, as shown in Figure 3.8 (step a). In the prefill phase, the embedding of dropped tokens and new prompt tokens are concatenated together as they are fed through the model layers. At each Transformer layer, Query, Key, and Value are generated (step b). Key and Value are stored in KV cache, and Pensieve maintains the KV locations for the entire conversation context including previously cached tokens (step c), which can then be used to perform attention.

However, as discussed in §3.3.2, a side effect of dropping leading tokens is that tokens in query tensor correspond to two disconnected ranges in the context, while all existing attention kernels assume that the Query tensor region is consecutive. To address this, we view these two ranges as two sub-requests that share the underlying context. Each row in Figure 3.8 step d represents the new Query tensor and KV context locations of a sub-request of the original request. Thanks to our multi-token attention kernel design (§3.4.4) which supports variable lengths in Query tensor for different requests in the batch and accepts KV-token locations as input, Pensieve only needs to update auxiliary data structures and no memory copy is incurred.

3.4.3.5 Suspending requests during generation

Despite ahead-of-time eviction, the scheduler might still encounter scenarios when the generation phase runs out of GPU cache since requests' decoding lengths are not known a priori. In this situation, the scheduler suspends some requests' execution by taking them out of the current batch, swaps out their corresponding KV-tokens to the CPU, and puts them back in the waiting queue. It chooses which request to suspend according to the descending order of their arrival time. As suspension causes increased latency (due to waiting for the swap-out), we try to avoid it by conservatively reserving 10% of GPU cache slots for the execution of existing requests that are in the generation phase. In other words, the scheduler stops adding new requests into the running batch unless there are more than 10% free GPU cache slots.

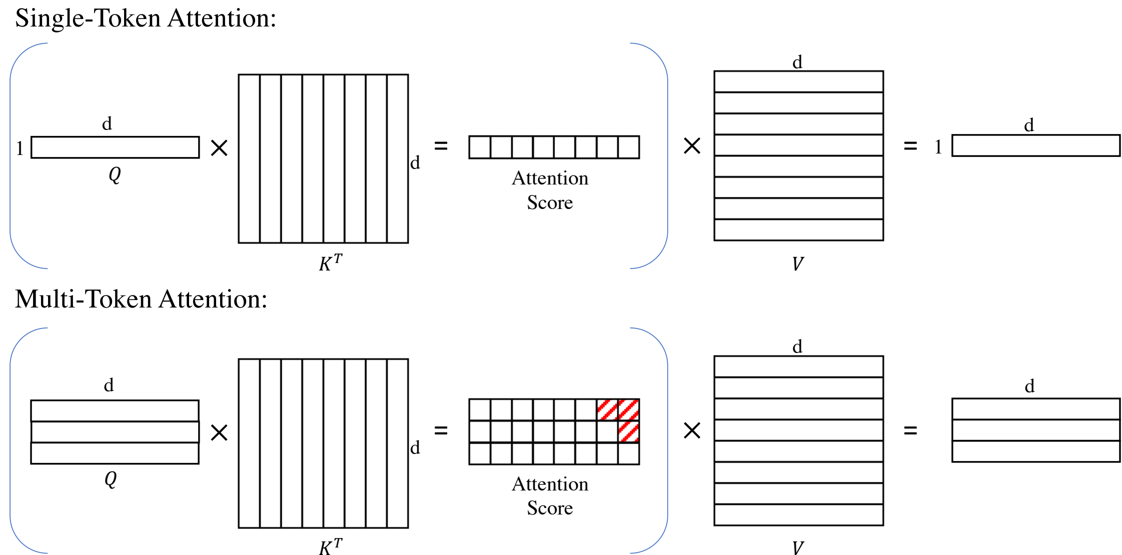


Figure 3.9: Single-token attention vs multi-token attention, softmax and rescaling omitted. d is the dimension of the embedding vector. The red shaded area in the multi-token attention score represents causal masking. K and V are logically contiguous, but physically placed at random locations when paged memory management is used.

3.4.4 Multi-token attention for non-contiguous cache

How to combine existing KV-tokens in the GPU cache with those just swapped in from the CPU? The most naive solution is to allocate a contiguous memory region in the GPU to hold them both. However, doing so would incur expensive memory copying, since KV-tokens are large. A more promising solution is to allocate separate space only for those swapped-in KV-tokens and to design an attention kernel implementation that can handle non-contiguous memory in its KV cache.

vLLM's PagedAttention kernel [54] can handle non-contiguous KV cache in the generation phase. However, for the prefill phase, it still uses existing kernels which require all KV-tokens to reside in contiguous memory. We cannot use PagedAttention because it assumes each request in a batch has exactly one input token, which is the case for generation but not prefill. Thus, we refer to PagedAttention as a single-token attention kernel because it computes the attention scores

between a single input token's query representation (Q) and the KV cache. We need to build a multi-token attention kernel that supports performing attention between the query representations (Q) of multiple input tokens per request and the KV cache over non-contiguous memory.

Our new kernel is similar to that of vLLM to the extent that both kernels need to support loading KV cache from non-contiguous GPU global memory into on-chip shared memory. Their main difference is illustrated in Figure 3.9. As vLLM's kernel performs attention between a request's single input token and the existing KV-tokens, its underlying computation can be described as two matrix-vector multiplication operations, as shown in Figure 3.9 [1]. In contrast, our kernel handles multiple input tokens for each request, computing attention scores between all pairs of input tokens and the conversation's existing KV-tokens. Therefore, its underlying computation can be described as two matrix-matrix multiplication operations, and the batched version of our kernel performs batched matrix-matrix multiplications. Like vLLM, we fuse the two multiplication operations according to [30]. Because of the additional dimension in the Q tensor, our kernel has more parallelization and tiling opportunities on the GPU. However, care must be taken to handle the new challenge that different requests in the batch have different numbers of input tokens.

When multiple input tokens of a request are handled together in a kernel, we need to apply causal masking so that a later input token does not "attend" to the earlier tokens. This requires setting a corresponding upper triangular part of the attention score matrix to 0, as shown by the red shaded area in Figure 3.9. Causal masking is not needed for the single-token attention kernel since it only processes one input token. We fuse the causal masking operation inside the multi-token attention kernel to avoid materializing the intermediate attention score matrix [30].

Matrix-matrix multiplication kernels are much more complex than those for matrix-vector multiplication because they typically use sophisticated algorithms to extract additional data reuse opportunities with GPU's on-chip shared memory and to leverage GPU's tensor core primitives. Therefore, instead of trying to extend vLLM's PagedAttention kernel for multi-token attention,

[1] The batched version of PagedAttention performs batched matrix-vector multiplications.

we base our implementation on an existing multi-token attention kernel from PyTorch and extend it to handle non-contiguous KV cache. Our kernel uses the high-performance thread-block level matrix-matrix multiplication provided by NVIDIA Cutlass template library [70].

3.4.4.1 Unifying the prefill and generation phase

As discussed in §3.4.2, in Pensieve's batch formation, new requests in their prefill phase can be grouped with existing requests in their generation phase. This is enabled by our multi-token attention kernel because single-query attention performed in the generation phase can be treated as a special case of multi-token attention with query size equal to 1.

More concretely, Pensieve's scheduler concatenates the input tokens to be processed from all requests, regardless of whether they correspond to prompts of new requests or the last step's output token from existing requests. Some auxiliary data structure is maintained to keep track of each request's corresponding region. During execution by the worker, these batched input tokens are fed through linear layers to generate each token's QKV representations. Newly generated KV-tokens are stored in the allocated slots in the GPU's KV cache. Then the worker applies our multi-token attention kernel to produce the output tokens for all requests.

3.5 Implementation

We have implemented our prototype serving system Pensieve with ~7K lines of C++/CUDA code. Pensieve manages KV cache and auxiliary data structure needed by multi-token attention on the GPU, but relies on PyTorch C++ front-end APIs to execute GPU operators in Large Language Models. Based on PyTorch's fused memory efficient attention, we develop our own fused multi-token attention kernel using NVIDIA Cutlass library to support performing attention with KV-tokens that reside in non-contiguous memory.

3.5.1 Optimization: Prioritize data retrieving over eviction

Although PCIe sends data full-duplex bidirectionally, in practice, we found that when CPU-to-GPU transfer is overlapped with GPU-to-CPU transfer, there is a significant throughput drop (18-20%) in both directions. Similar issues have been reported[2]. Since Pensieve performs KV-token backup ahead of time, there is no urgency to finish the transfer right away. To prevent evicting from slowing down swapping in past KV-token, we set up a blocking mechanism. Whenever a worker plans to issue a GPU-to-CPU copy, it first checks if there is any swap-in task in the system, and waits until the swap-in task is done. Although this conservative approach does not fully utilize duplexed PCIe bandwidth, due to the existence of generation phase, we find that this optimization performs well and we never run into the situation that the backup can't catch up.

3.6 Evaluation

In this section, we present the evaluation results of Pensieve.

3.6.1 Experimental Setup

System Environment We evaluate Pensieve on Microsoft Azure NC A100 v4 series, which are equipped with up to 4 A100-80GB GPUS. For each A100 GPU, 24 non-multithreaded AMD EPYC Milan processor cores and 220 GB system memory are supplied. Pensieve and all baselines use CUDA version 11.8 and PyTorch 2.0. We configure each system to allocate 40GB GPU memory to KV cache for a fair comparison.

Models We use the OPT [113] and Llama 2 [98] as our benchmark models. OPT has an almost identical model architecture to the famous GPT-3 [10] model while providing an open-source

[2] https://forums.developer.nvidia.com/t/data-transfers-are-slower-when-overlapped-than-when-running-sequentially/187542

Model	OPT-13B	OPT-66B	Llama 2-13B	Llama 2-70B
# layer	40	64	40	80
# hidden	5120	9216	5120	8192
# head	40	72	40	64
# KV head	40	72	10*	8
Head size	128	128	128	128
# GPU used	1	4	1	4

Table 3.1: Model parameters for OPT and Llama 2 evaluated in this work. *For Llama 2-13B, we set the number of KV heads to 10 (originally 40) to show the effect of Grouped-query Attention (GQA).

	ShareGPT
# conversations	48159
Mean # turns	5.56
Mean request input length	37.77
Mean request output length	204.58

Table 3.2: Dataset staticstics

codebase and pre-trained model parameters. Llama 2 is a more recent model that shares a similar design of GPT models but applies many advanced techniques like rotary embedding, RMS Layernorm [112], SiLU, etc. Notably, Llama 2 follows the trend of adopting Grouped-query Attention [4] which divides query heads into groups, and within each group, only one KV head is used. Such a design significantly reduces memory consumption of KV-tokens, allowing Pensieve to store more past KV-tokens.

We evaluate two different scales for each model: a small one on a single GPU, and a large one partitioned onto 4 GPUs using Tensor Parallelization as done in Megatron-LM [89]. Detailed model hyper-parameters can be found in Table 3.1. By default, Llama 2 only uses Grouped-query Attention for models with over 70 billion parameters. To demonstrate Pensieve's effectiveness when used with Grouped-query Attention, we modified Llama 2-13B and decreased the number of KV heads from 40 to 10. In all experiments, the 16-bit half-precision float format is used for both model parameters and intermediate hidden representations. Model parameters are randomly initialized.

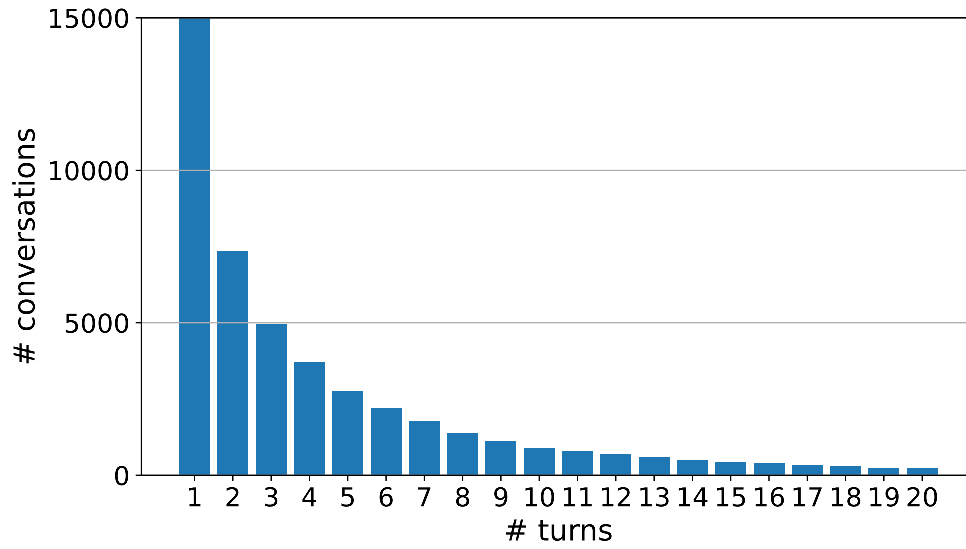


Figure 3.10: Distribution of conversation turns in ShareGPT dataset.

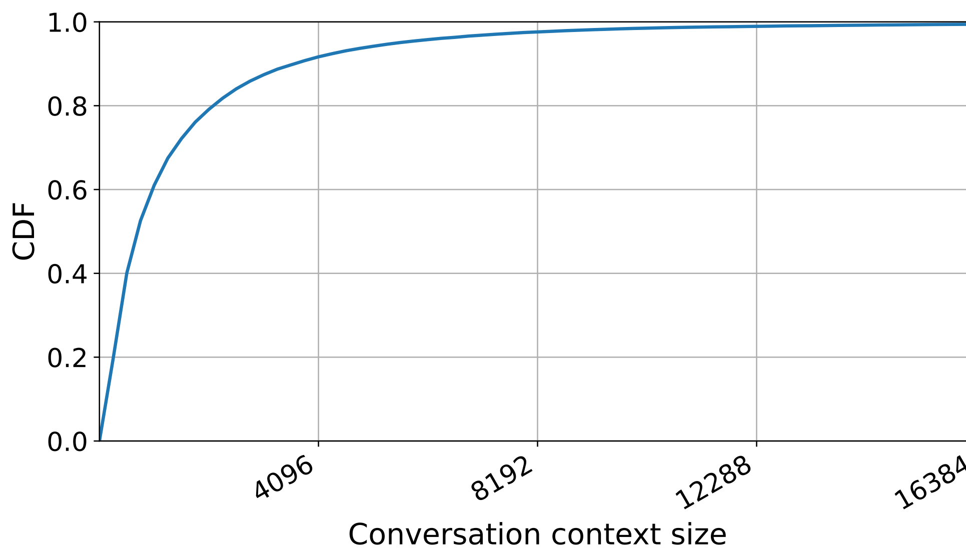


Figure 3.11: CDF of conversation's total context size in ShareGPT dataset.

Dataset We evaluate Pensieve on ShareGPT [86], a public dataset containing user-shared ChatGPT conversations[3]. Statistics about datasets are listed in Table 3.2. Figure 3.10 shows the distribution of the number of conversation turns. Figure 3.11 shows the CDF graph of conversation's total context size. In this evaluation, we limited the maximum context size to 16384 tokens and dropped 0.57% of the conversations in ShareGPT dataset that exceed this limit.

We extract the input and output lengths of each conversation turn from these datasets to construct requests. To ensure that the same computation amount is performed by each system, we ignore any early stop criterion, e.g. end-of-sentence (EOS) token, and keep generating until the ground-truth output length is reached. This ground-truth output length is only used to terminate token generation and is not disclosed to the system for making scheduling decisions.

Workload Since the dataset does not provide timestamps for when each user prompt is received, we sample the arrival time for each request from Poisson distribution under different request rates. We maintain the causal dependency for requests belonging to the same conversation: a new user prompt is only sent to the system after earlier ones from the conversation have already received their response. In addition, in real-life chatting scenarios, it takes time for users to reply. We sample such reaction time from an exponential distribution with an average reaction time of 1 minute. Hence once a conversation finishes generating output tokens for its current turn, no new request from this conversation will be emitted until at least the sampled reaction time has passed. We study the impact of reaction time later in §3.6.5.

Baseline We evaluate our system against the state-of-the-art vLLM serving system [54] (version v0.2.0), which uses a stateless serving API. For each request, a new user prompt is appended to the history and then sent together as an input request. Since vLLM is an execution engine without a perpetual serving loop that drives the engine to continue execution, we implement such a

[3] We used `https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered`, which performed additional filtering like removing HTML tags and non-English conversation.

driver that adds newly arrived requests into vLLM job queue and invokes the engine execution until all requests are fully processed.

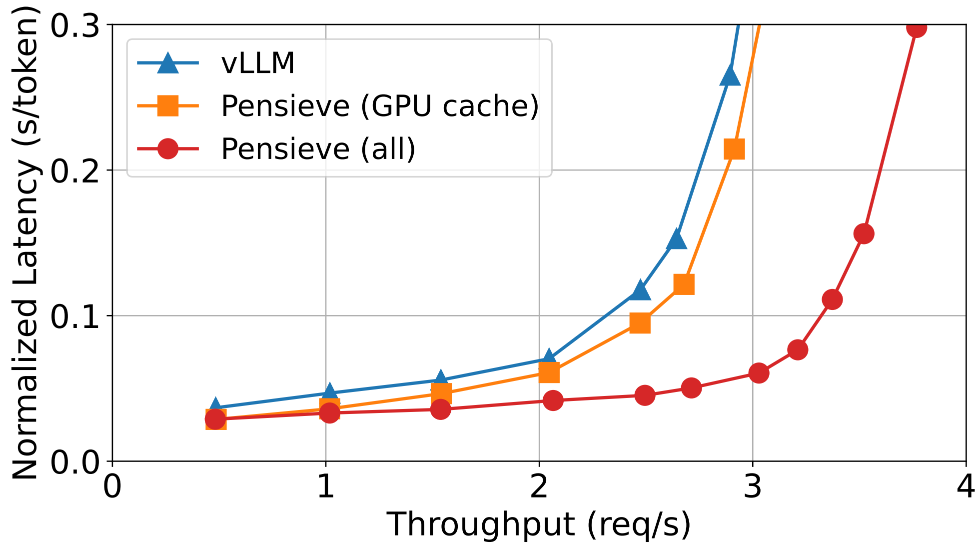
We also implement a variant of Pensieve called Pensieve (GPU cache) that shares the same eviction policy as Pensieve but that does not use CPU memory. Pensieve (GPU cache) saves past KV-tokens in spare GPU memory spaces to reuse them in the future. Whenever an eviction happens due to memory pressure, selected past KV-tokens are dropped from the GPU memory. This variant is used to demonstrate the necessity of CPU memory in Pensieve.

Performance Metric Pensieve is designed to optimize both peak throughput and latency of serving LLM in conversational scenarios. Following prior work [54, 111], we measure the achieved serving throughput and 90-percentile normalized latency, which is calculated as the end-to-end request serving latency divided by the number of output tokens.

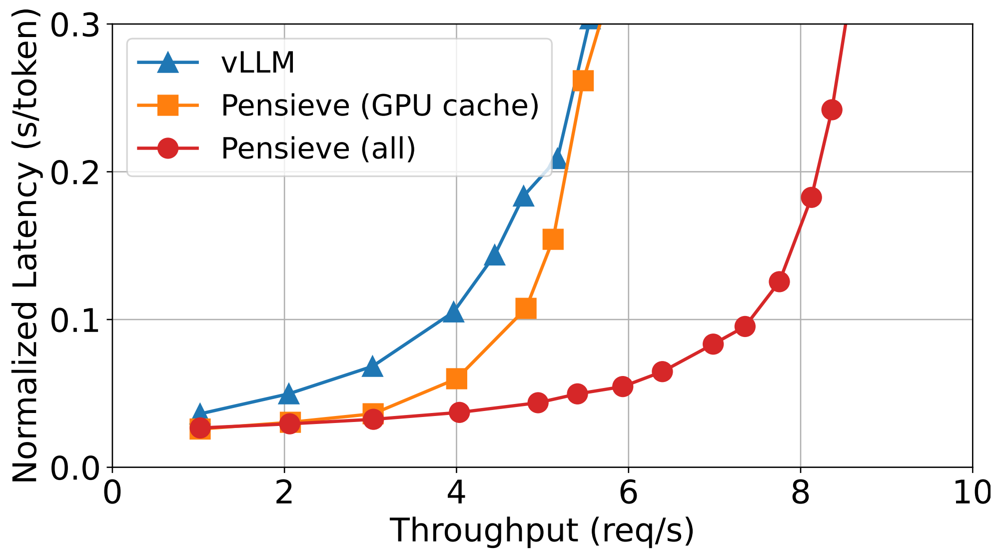
3.6.2 End-to-end Pensieve Performance

Figure 3.12 shows the normalized latency vs throughput for OPT-13B and Llama 2-13B served with a single A100-80GB GPU. Pensieve can achieve a 25% throughput gain on OPT-13B and a 52% throughput gain on Llama 2-13B over vLLM and Pensieve (GPU cache) baselines. Our improvement on Llama 2-13B is more significant because as mentioned in §3.6.1, this version of Llama 2-13B uses Group-query Attention with group size 4 (i.e. every four query heads share a single key-value head). As a result, the amount of memory required to store past KV-tokens is reduced by 4x. This allows both Pensieve (GPU cache) and Pensieve to store more past KV-tokens in the system, and therefore, have a larger chance to retrieve them in the future and avoid recomputing KV-tokens from scratch.

When Pensieve is used without CPU cache, i.e. Pensieve (GPU cache), it still reduces up to 20% of the normalized latency compared to vLLM for OPT-13B and up to 40% for Llama 2-13B before reaching the peak throughput. This is because vLLM always recomputes past KV-tokens



(a) OPT-13B



(b) Llama 2-13B

Figure 3.12: Normalized latency vs throughput for OPT-13B and Llama 2-13B on 1 GPU

for requests from a returning conversation, and thus, the number of tokens it needs to process in prefill phase is on average larger, causing it to take longer time. But under relatively higher request rates, GPU cache is quickly exhausted, and Pensieve (GPU cache) behaves similarly to vLLM as it also resorts to recomputing past KV-tokens from scratch.

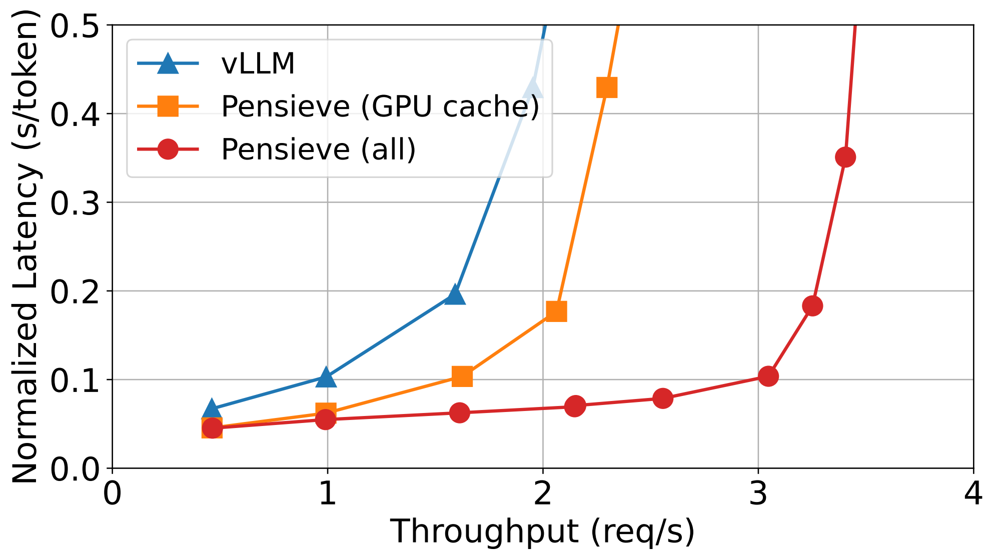
Figure 3.13 shows the performance of Pensieve on larger models, OPT-66B and Llama 2-70B, when served with four A100-80GB GPUs. Larger models amplify Pensieve's advantage over the baselines because the computation amount grows faster than the memory usage of KV-tokens. For example, from OPT-13B to OPT-66B, the model parameter size and computation amount increase by more than 4x, while the hidden size only increases 1.8x from 5120 to 9216 (Table 3.1). Since the number of GPUs and CPU memory are usually scaled linearly with the model size, Pensieve can store more past KV-tokens in its KV cache.

Pensieve is able to achieve a 1.5x throughput gain on OPT-66B and a 1.6x throughput gain on Llama 2-70B over vLLM and Pensieve (GPU cache) baselines. The improvement is more significant on Llama 2-70B because it uses Group-query Attention with group size 8, which reduces the memory requirement for past KV-tokens by 8x. This reduced memory consumption in KV-tokens increases the number of KV-tokens that can be stored in GPU cache, which significantly benefits the throughput of Pensieve (GPU cache).

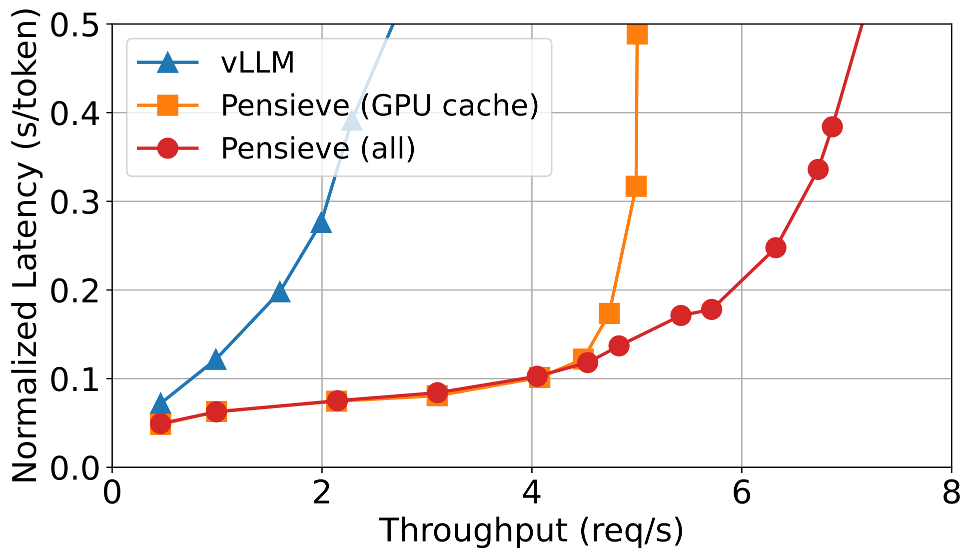
3.6.3 Eviction Policy

In this section, we compare Pensieve's caching policy (Equation 3.4) against the classic Least Recently Used (LRU) policy, which only considers the amount of time since the chunk was last accessed. We use OPT-13B as the workload for this evaluation. From Figure 3.14, both policies exhibit similar performance until reaching a workload rate of 3 requests per second, and then Pensieve's policy starts to outperform LRU.

Based on execution traces, both policies started to experience CPU cache misses starting from



(a) OPT-66B



(b) Llama 2-70B

Figure 3.13: Normalized latency vs throughput for OPT-66B and Llama 2-70B on 4 GPUs

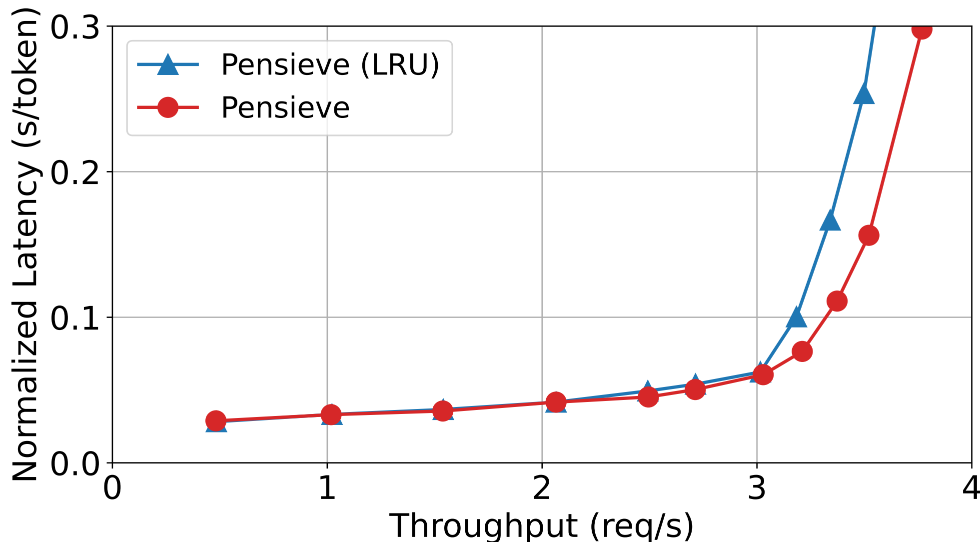
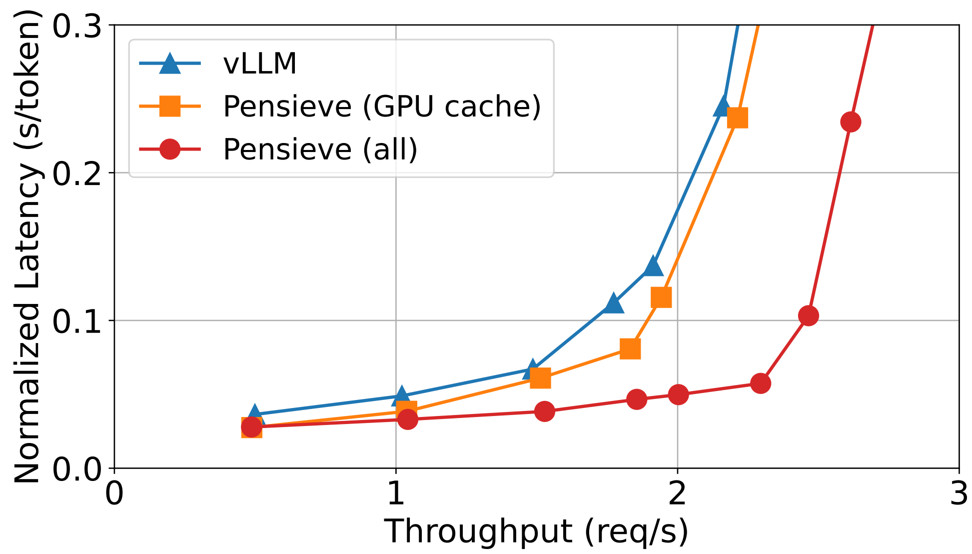


Figure 3.14: Pensieve performance for serving OPT-13B with different eviction policies.

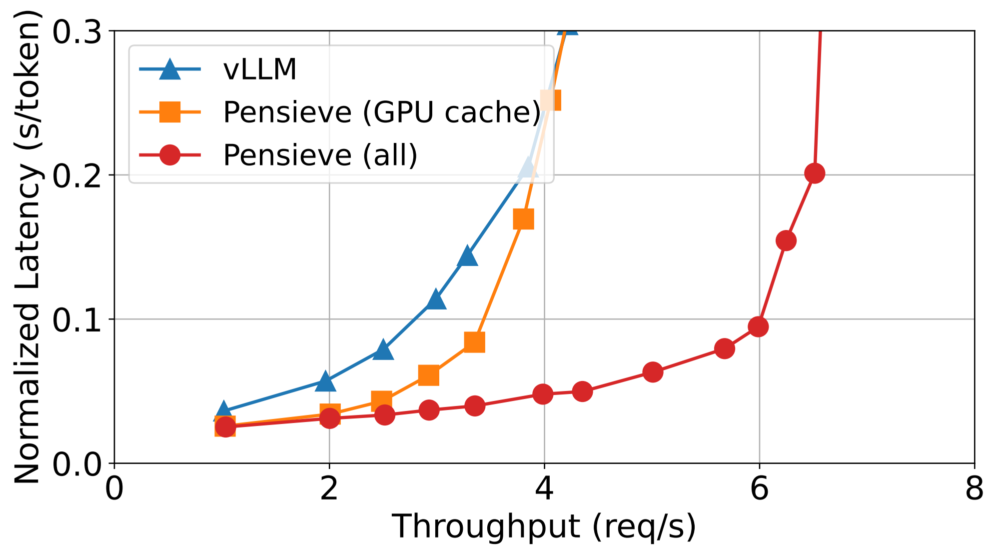
a workload of 2 requests per second, but their CPU cache hit rates remain close. Even though the cache hit rate decreases as the request rate increases, the GPU still has enough computation capacity to recover missing tokens without significantly increasing latency. After the request rate of 3.2 requests per second, both policies drop below 80% cache hit rate, but Pensieve's policy has up to 4.4 percentage points higher cache hit rate than LRU. On average, Pensieve's policy reduces the number of recomputed KV-tokens by up to 14.6%.

3.6.4 Conversation with More Turns

Pensieve is designed for multi-turn conversations. The more times past KV-tokens are reused, the more effective Pensieve is. However, as shown in Figure 3.10, ShareGPT dataset is mostly comprised of conversations with a small number of turns and we have no way to tell if a conversation will resume in the future. Worse still, storing short conversations causes cache pollution as they may evict past KV-tokens of other conversations. To evaluate the impact of conversation length on Pensieve, we craft a new dataset from ShareGPT to exclude conversations with less than 5 turns. 36% of the original dataset is retained in the new dataset.



(a) OPT-13B



(b) Llama 2-13B

Figure 3.15: Pensieve performance on ShareGPT dataset with at least 5 turns.

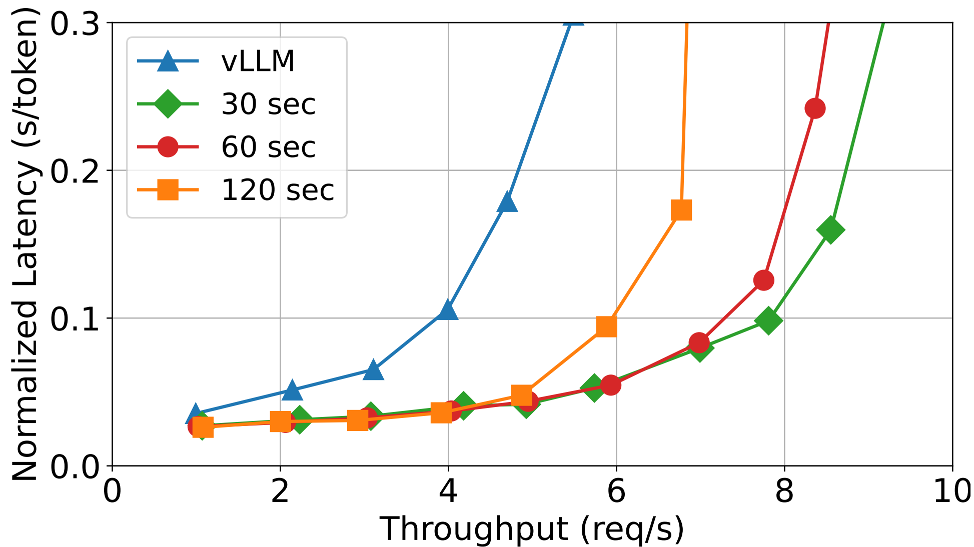


Figure 3.16: Pensieve performance with different average user reaction time using Llama 2-13B.

Figure 3.15 shows Pensieve's performance for OPT-13B and Llama 2-13B on the new dataset. Compared to Figure 3.12, we observe decreases in throughput for all systems because the workload now contains requests with longer average conversation histories. Larger history context sizes make attention operation slower, but this hurts vLLM baseline most as recomputing past KV-tokens from scratch now becomes a lot more expensive. Compared to running on the full ShareGPT dataset, Pensieve's throughput gain over vLLM increases to 30% for OPT-13B and to 70% for Llama 2-13B.

3.6.5 User Reaction Time

Another important factor that affects the performance of Pensieve is the user's reaction time. Pensieve is designed with the assumption that an active user sends out the next request shortly after receiving the response so that there is a larger chance past KV-tokens are still available in the system for reusing. Figure 3.16 evaluates the impact of different average reaction times on the performance of Pensieve using Llama 2-13B. We sample reaction time from an exponential distribution with an average reaction time of 30, 60, and 120 seconds respectively. Since vLLM is

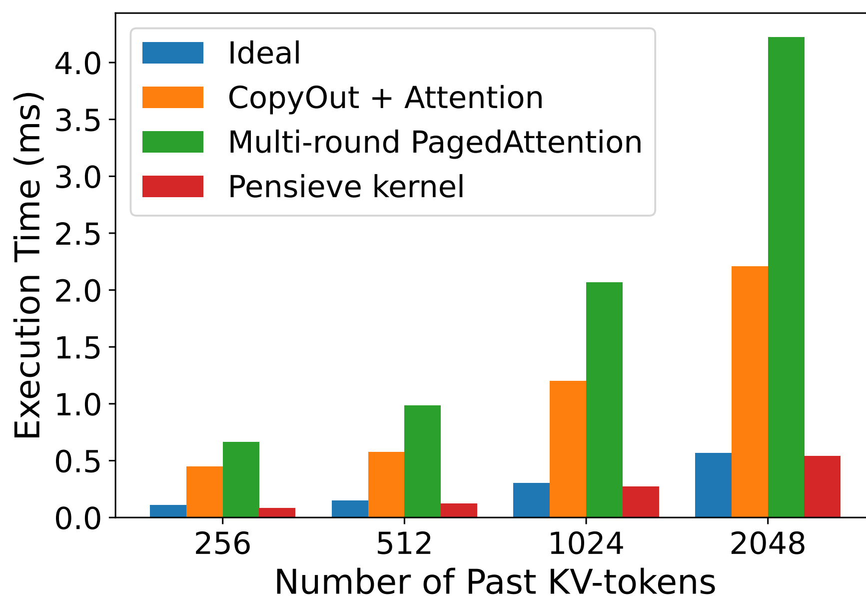


Figure 3.17: Execution time of Pensieve's multi-token attention kernel over non-contiguous context memory with different context size. Measured with batch size 32 and query size 8.

stateless and recomputes past KV-tokens for each request, even though the request arrival order changes, over a long period, vLLM should still process the same set of requests, and thus can serve as a reference point.

As can be seen from the figure, with the increase of average user reaction time, the throughput of Pensieve decreases, because the chance of past KV-tokens being dropped from the cache increases. When the reaction time increases to 120 seconds, Pensieve's throughput advantage over vLLM reduces to around 40%. Still, under low request rates, Pensieve can maintain a significantly lower latency compared to vLLM.

3.6.6 Efficiency of Multi-token attention Kernel

Figure 3.17 shows the advantage of Pensieve's multi-token attention kernel when used to prefill KV-tokens of prompts from new requests with their past KV-tokens. In this microbenchmark, a batch of 32 requests performs attention between a prompt of 8 query tokens and different num-

bers of past KV-tokens stored in non-contiguous GPU memory. As described in §3.3.1, existing attention kernels are not directly applicable. We compare against two straw-man implementations: 1) allocate additional contiguous GPU memory space to copy out past KV-tokens and then apply a fused attention kernel (orange bar), and 2) apply multiple rounds of vLLM's single-query PagedAttention to process one token from the prompt at a time (green bar). We also present the performance of the ideal situation which assumes that past KV-tokens are already in contiguous memory space (blue bar). From Figure 3.17, both straw-man solutions add significant overhead compared to the ideal performance. Copy out requires extra memory space and gathers past KV-tokens from non-contiguous space. Such cost is proportional to the number of past KV-tokens. Applying multiple rounds of PagedAttention, on the other hand, gives up parallelization opportunities on prompt tokens, leading to linear execution time to the number of tokens in the prompt. Pensieve's kernel shows slightly better performance than the ideal baseline because we offload auxiliary data computing (like calculating the cumulative sum for the sequence length of a batch) to the CPU. Since each transformer layer in the model shares the same caching plan, these auxiliary data can be reused by all layers.

3.6.7 Unified Scheduling

In this section, we evaluate the benefit of using a unified scheduler for both prefill and generation phases. We evaluated the performance of Pensieve using Llama 2-13B on the ShareGPT dataset with unified scheduling disabled so that whenever new requests are waiting in the queue, the scheduler pauses the generation phase and executes the prefill phase of new requests. Figure 3.18 shows the performance of Pensieve with and without unified scheduling. Compared to processing prefill and generation phases separately, unifying them into a single execution step avoids executing the prefill phase with a small number of requests and delaying generation phase of current running requests. As a result, Pensieve with unified scheduling achieves better throughput and

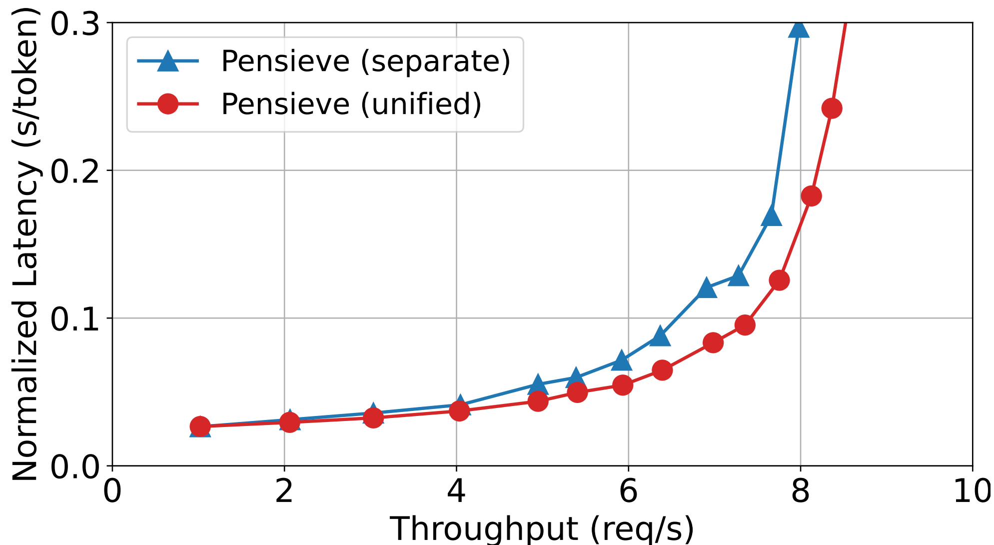


Figure 3.18: Pensieve performance of serving Llama 2-13B with and without unified scheduling.

latency.

3.7 Related Works

3.7.1 LLM inference and serving systems

Many systems have been recently developed to serve language models with better performance including vLLM [54], Orca [111], TensorRT-LLM [73], FasterTransformer [72], LightSeq [103], DeepSpeed [5] and [80]. These systems investigate different performance improvement opportunities than Pensieve. The wide range of proposed techniques include incremental decoding [72], iteration-level batching [38, 111], supporting paged KV cache over non-contiguous GPU memory [54, 73], finding an efficient multi-device partitioning plan [80], better load balancing between prefill and generation to reduce pipeline parallelism bubbles [3, 47], kernel fusion [30, 31, 82], speculative decoding [13, 55], and quantization [34, 39, 66, 109].

The system described in [80] can keep a fixed-sized KV cache for each conversation in TPU device memory. Such caching is too rigid. By contrast, Pensieve manages the cache across all con-

versations and can also swap to CPU to relieve GPU memory pressure. Another recent system, CacheGen [57], addresses the problem of reducing the context-loading delay for long contexts stored in remote network storage. CacheGen proposes an adaptive compression scheme to accelerate the transfer of the KV cache over a bandwidth-limited and bursty network. As Pensieve only keeps the KV cache in local GPU and CPU memory, CacheGen's techniques are orthogonal to those of Pensieve. For conversational chatbots, Pensieve using local cache can already bring significant performance gains.

3.7.2 Non-LLM specific DNN serving systems

Many systems like Clipper [29], TensorFlow Serving [75], Nexus [87], NVIDIA Triton Inference Server [71], and InferLine [27] serve as scheduling components of a general-purpose DNN serving system. They are mostly model-agnostic and execution backend-agnostic and apply general system techniques like batching, caching, and software pipelining to serve DNN models. Some are also in charge of properly provisioning compute resources to improve overall cluster efficiency. A few existing works target model-less serving and provide inference as a service: they automatically select models to meet the accuracy and latency requirements of a given user task. For example, INFaaS [85] and Tabi [104] serve requests with a small model and only re-route to a larger model when the output confidence score is low.

3.7.3 Techniques addressing the GPU memory limit

These include GPU-CPU swapping, recomputation, and unified memory. Most of the systems described below are not specifically targeted for LLM serving.

GPU-CPU Swapping SwapAdvisor [49] swaps weight and activation tensors for DNN training. It uses the dataflow graph to determine an optimal plan that involves operator execution or-

der, memory allocation, and swapping. Zero-Offload [84] offloads optimizer state and gradients to the CPU during LLM training with a single GPU. DeepSpeed-ZeroInference [5] and FlexGen [88] use offloading to serve LLM with a weak GPU. DeepSpeed-ZeroInference offloads entire model weights to CPU or NVMe memory. FlexGen offloads currently unused model weights, activation, or KV cache to CPU memory or disk. As a result of frequent data movement and high disk latency, these two systems require a large batch size to hide the offloading latency. Therefore, they mainly target latency-insensitive applications. Neither system persists the KV cache across requests.

Recomputation For DNN training, recomputing activation on the backward pass [16] is a popular technique used to reduce memory footprint. The decision for which tensors to compute is done at the granularity of layers [16], a group of operators within a layer (aka modules) [53], or individual operators [50]. Unlike these works, Pensieve's fine-grained recomputation is done at the token(chunk)-level, to exploit the insight that earlier tokens in a sequence incur less recomputation cost due to the causal nature of attention.

Unified Memory and Direct Host Access Swapping between CPU and GPU memory can also be achieved transparently through Unified Memory [69], which automatically triggers memory page migration between CPU and GPU using a page fault mechanism. The Direct Host Access [63] feature allows a GPU kernel to directly read from the CPU memory, thereby allowing a subset of the threads to execute as soon as their data is ready without waiting for the entire transfer to finish. It has been used to enable better overlap of data transfer and kernel computation when swapping in a model from CPU [64] or accessing large graph neural network features on the CPU [51].

For our implementation of Pensieve, we choose not to use Unified Memory nor Direct Host Access because these mechanisms trigger memory transfer only when the data is accessed by a

GPU kernel and we want to manage data movement explicitly to prefetch the KV cache. Furthermore, for Direct Host Access, since the copied data is not explicitly stored in GPU memory, it has to be loaded from the CPU again if it is repeatedly used. In our workload, the context cache is not only used to prefill the prompt tokens but also in every generation step, which makes Direct Host Access significantly less efficient.

3.7.4 Techniques that speed up LLM inference

Kernel Fusion Deep Neural Networks often have a lot of memory-bound operations, e.g., element-wise operators, that read a large amount of data while performing little computation. A common technique to improve GPU efficiency is kernel fusion [15, 21], which fuses multiple kernels as one single kernel to avoid storing intermediate data in GPU global memory and later loading them back. Attention in LLM can also benefit from kernel fusion. Memory efficient attention [82] and Flash Attention [30] fuse all attention-related operations into one kernel to avoid materializing the attention score tensor. The attention kernel we implemented in Pensieve is a generalization of aforementioned fused kernels, with support to efficiently attend new tokens with past KV-tokens stored in non-contiguous memory. Recently, Flash-decoding [31] further parallelizes the processing of KV-tokens for single query token attention, which can significantly speed up the generation phase. With prefill phase taking up a larger portion of the total computation, Pensieve's optimization for prefill phase may become even more beneficial.

Quantization Quantization techniques use lower precision floating point or integer representation to speed up execution and reduce memory footprint [39, 66]. 16-bit half precision is widely adopted in LLM inference and mixed precision training [62]. And recently, several works [33, 34, 88] have shown that quantizing to 4-bit can still maintain good LLM inference quality. In this work, we focus only on 16-bit half-precision to deliver unchanged inference results. But in general, quantization-based methods are also applicable to our system.

Speculative Decoding Recently, speculative decoding [13, 55] is proposed to speed up the generation phase in LLM inference. Instead of running a costly LLM model to sequentially produce 1 output token in each model iteration, speculative techniques use a smaller and faster model to generate a draft of a series of tokens, which get evaluated by a larger and more powerful LLM model later with more efficient parallel execution. The adoption of speculative decoding significantly reduces the execution time of generation phase, which makes Pensieve's optimization of prefill phase in multi-turn conversation serving more important.

3.8 Summary

We identify that a major inefficiency in serving Large Language Models for introduce Pensieve, a stateful LLM serving system that preserves context data in a multi-tier GPU-CPU cache and we develop a new GPU attention kernel that supports performing attention between requests' new multi-token input and their saved context stored in non-contiguous GPU memory. Experiments show that Pensieve improves serving throughput by 33-100% and reduces latency by 40-77% compared to the state-of-the-art system vLLM.

4 | Conclusion

This thesis presents two systems, BatchMaker and Pensieve, for efficient serving of language models by fine-grained and stateful scheduling.

BatchMaker proposes a novel approach called cellular batching to achieve low-latency inference on Language Models. cellular batching batches the execution of an inference request at the granularity of a cell. Doing so allows a new request to join the execution of a batch of existing requests and to be returned as soon as its computation finishes without waiting for others in the batch to complete. We demonstrated the effectiveness of this approach by building BatchMaker and evaluating using RNN workloads.

Pensieve is a serving system designed for the scenario of multi-turn conversation by providing efficient context management for Large Language Models (LLMs). Pensieve preserves context data in a multi-tier GPU-CPU cache and we develop a new GPU attention kernel that supports performing attention between requests' new multi-token input and their saved context stored in non-contiguous GPU memory.

4.1 Future Work

Although this thesis has laid out a general framework for efficient serving of language models, which includes two fundamental perspectives: fine-grained scheduling and context state management, it still has many design limitations worth further exploration.

4.1.1 Unbalanced workload when used with pipeline parallelism

With model size growing larger and larger, it is common to distribute computation across multiple GPUs. Cellular batching proposed by BatchMaker works well with Tensor Parallelism where each model layer is partitioned into equal-sized chunks and distributed onto different GPUs, and therefore all GPUs share identical computation.

However, another commonly used distribution strategy called Pipeline Parallelism partitions a model into stages and each GPU (or a group of GPUs) is only responsible for a single stage. With Pipeline Parallelism, there are multiple running batches in the system, each executing a different stage in parallel. Since new requests arrive from time to time and each request may have variable output lengths and require a different number of generation steps, each batch will have different numbers of requests. Such differences in batch size at different stages will lead to differences in computation time, causing the pipeline to stall. Therefore, it is necessary to design a mechanism to better balance the amount of work across different batches.

4.1.2 Multi-host load balancing

In real deployment, one single machine is rarely enough to handle all the requests. But in a multi-host setup, a conversation's saved history states are only useful when its future requests are also served by the same machine. Since different conversations are likely to have different numbers of turns, there is a potential load-balancing issue when some hosts are overloaded with long conversations while others are idle.

Therefore, rebalancing or job-stealing must be supported. There are at least two possible designs. One is to resort to recomputation when switching the requests of a conversation to a different machine than the one holding the history KV cache. Another is to store the KV history cache in a fast disaggregated in-memory cache service.

4.1.3 Support for various storage backends

Pensieve relies on a GPU-CPU cache to store the context data. However, even though CPU memory is generally considered cheap and abundant, it is still limited in size. For example, as calculated in §3.3.2, the key-value representation of a token for the GPT-3 model requires 0.78MB to store. A system with e.g. 200GB total memory can only store about 260K tokens. As the system keeps serving new conversations, the cache will run out of space and suffer from a low hit rate, leading to performance degradation.

An interesting direction to explore is to support more storage backends, such as a distributed database. However, remote storage tends to have higher access latency. Therefore, it is important to further explore the trade-off between recomputation and retrieving along with other techniques like compression and quantization that can reduce transmission time.

4.1.4 Conversations with similar texts

The main contribution of Pensieve is to go beyond the boundary of requests and maintain states for conversations that last multiple turns. However, it could be even more beneficial to take one step further to exploit similarities among conversations and share states across multiple conversations. At the moment, Pensieve is designed to handle unique conversations and does not consider sharing states among different conversation sessions. But in certain scenarios, conversations from different users may share similar texts. For example, a ChatBot providing customer services may receive similar service requests from users. In the meanwhile, these ChatBots tend to generate similar responses since they are trained with a limited number of template responses from the dataset. As a result, different conversations may have a high chance of having similar texts.

Therefore, one way to boost Pensieve's performance is to allow sharing states across multiple conversations. There are two main benefits. First, attention key-value states from one conver-

sation can be cached and reused to avoid processing similar texts again in another conversation. Second, the system can avoid saving multiple copies of states for the same piece of text. However, additional care is needed to properly identify texts that are similar enough to reuse and to correctly handle position-dependent model components like positional encoding.

Bibliography

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283.

[2] Agarwal, D., Long, B., Traupman, J., Xin, D., and Zhang, L. (2014). Laser: A scalable response prediction platform for online advertising. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 173–182. ACM.

[3] Agrawal, A., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., and Ramjee, R. (2023). Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*.

[4] Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. (2023). Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*.

[5] Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., et al. (2022). Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE.

[6] Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Cheng, Q., Chen, G., et al. (2016). Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*, pages 173–182.

[7] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[8] Bai, Z., Zhang, Z., Zhu, Y., and Jin, X. (2020). PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514.

[9] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, pages 1–7.

[10] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

[11] Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., et al. (2023). Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*.

[12] Candea, G., Polyzotis, N., and Vingralek, R. (2009). A scalable, predictable join operator for highly concurrent data warehouses. *Proceedings of the VLDB Endowment*, 2(1):277–288.

[13] Chen, C., Borgeaud, S., Irving, G., Lespiau, J.-B., Sifre, L., and Jumper, J. (2023a). Accelerating large language model decoding with speculative sampling. In *arXiv:2302.01318*.

[14] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015). Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

[15] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. (2018). TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594.

[16] Chen, T., Xu, B., Zhang, C., and Guestrin, C. (2016). Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*.

[17] Chen, Y., Liang, Y., and Lin, Z. (2023b). Transformer-based large language models are not general learners: A universal circuit perspective. *OpenReview*.

[18] Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*.

[19] Community, D. D. M. L. (2017a). The gluon package. http://gluon.mxnet.io.

[20] Community, D. D. M. L. (2017b). Nnvm: Open compiler for ai frameworks. https://github.com/dmlc/nnvm/.

[21] Community, T. (2017c). Tensorflow xla. https://www.tensorflow.org/performance/xla/.

[22] Community, T. W. (2017d). Argmax. https://en.wikipedia.org/wiki/Arg_max.

[23] Corporation, M. (2015). Microsoft cognitive toolkit (cntk). https://github.com/Microsoft/CNTK.

[24] Corporation, N. (2017). Tensorrt. https://developer.nvidia.com/tensorrt.

[25] Corporation, N. (2018). Cuda runtime api. http://docs.nvidia.com/cuda/cuda-runtime-api/index.html.

[26] Crankshaw, D., Bailis, P., Gonzalez, J. E., Li, H., Zhang, Z., Franklin, M. J., Ghodsi, A., and Jordan, M. I. (2014). The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809*.

[27] Crankshaw, D., Sela, G.-E., Mo, X., Zumar, C., Stoica, I., Gonzalez, J., and Tumanov, A. (2020). Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 477–491.

[28] Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. (2017a). Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627.

[29] Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. (2017b). Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627.

[30] Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*.

[31] Dao, T., Haziza, D., Massa, F., and Sizov, G. (2023). Flash-Decoding for long-context inference. https://pytorch.org/blog/flash-decoding/.

[32] Darling, W. (2016). Recurrent neural networks with cntk and applications to the world of ranking. https://www.microsoft.com/en-us/cognitive-toolkit/blog/2016/08/recurrent-neural-networks-with-cntk-and-applications-to-the-world-of-ranking/.

[33] Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. (2023). Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*.

[34] Dettmers, T. and Zettlemoyer, L. (2023). The case for 4-bit precision: k-bit inference scaling laws. In *International Conference on Machine Learning*, pages 7750–7774. PMLR.

[35] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[36] Diamos, G., Sengupta, S., Catanzaro, B., Chrzanowski, M., Coates, A., Elsen, E., Engel, J., Hannun, A., and Satheesh, S. (2016). Persistent rnns: Stashing recurrent weights on-chip. In *International Conference on Machine Learning*, pages 2024–2033.

[37] Donahue, J., Anne Hendricks, L., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K., and Darrell, T. (2015). Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2625–2634.

[38] Gao, P., Yu, L., Wu, Y., and Li, J. (2018). Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15.

[39] Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., and Keutzer, K. (2022). A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision*, pages 291–326. Chapman and Hall/CRC.

[40] Giannikis, G., Alonso, G., and Kossmann, D. (2012). Shareddb: killing one thousand queries with one stone. *Proceedings of the VLDB Endowment*, 5(6):526–537.

[41] Goertzel, B. (2014). Artificial general intelligence: concept, state of the art, and future prospects. *Journal of Artificial General Intelligence*, 5(1):1–48.

[42] Goertzel, B. (2023). Generative ai vs. agi: The cognitive strengths and weaknesses of modern llms. *arXiv preprint arXiv:2309.10371*.

[43] Goertzel, B. and Pennachin, C. (2007). *Artificial general intelligence*, volume 2. Springer.

[44] Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., et al. (2014). Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*.

[45] Harizopoulos, S., Shkapenyuk, V., and Ailamaki, A. (2005). Qpipe: a simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 383–394. ACM.

[46] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[47] Holmes, C., Tanaka, M., Wyatt, M., Awan, A. A., Rasley, J., Rajbhandari, S., Aminabadi, R. Y., Qin, H., Bakhtiari, A., Kurilenko, L., et al. (2024). Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*.

[48] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. (2021). Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

[49] Huang, C.-C., Jin, G., and Li, J. (2020). SwapAdvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355.

[50] Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Gonzalez, J., Keutzer, K., and Stoica, I. (2020). Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511.

[51] Jeong, J., Baek, S., and Ahn, J. (2023). Fast and efficient model serving using multi-gpus with direct-host-access. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 249–265.

[52] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM.

[53] Korthikanti, V. A., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. (2023). Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5.

[54] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. (2023). Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[55] Leviathan, Y., Kalman, M., and Matias, Y. (2023). Fast inference from transformers via speculative decoding. In *ICML*.

[56] Li, C. (2020). Openai's gpt-3 language model: A technical overview. *Blog Post*.

[57] Liu, Y., Li, H., Du, K., Yao, J., Cheng, Y., Huang, Y., Lu, S., Maire, M., Hoffmann, H., Holtzman, A., et al. (2023). Cachegen: Fast context loading for language model applications. *arXiv preprint arXiv:2310.07240*.

[58] Looks, M., Herreshoff, M., Hutchins, D., and Norvig, P. (2017). Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*.

[59] Makreshanski, D., Giannikis, G., Alonso, G., and Kossmann, D. (2017). Many-query join: efficient shared execution of relational joins on modern hardware. *The VLDB Journal*, pages 1–24.

[60] Marelli, M., Bentivogli, L., Baroni, M., Bernardi, R., Menini, S., and Zamparelli, R. (2014). Semeval-2014 task 1: Evaluation of compositional distributional semantic models on full sentences through semantic relatedness and textual entailment. In *Proceedings of the 8th international workshop on semantic evaluation (SemEval 2014)*, pages 1–8.

[61] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. (2016). Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241.

[62] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., et al. (2017). Mixed precision training. *arXiv preprint arXiv:1710.03740*.

[63] Min, S. W., Mailthody, V. S., Qureshi, Z., Xiong, J., Ebrahimi, E., and Hwu, W.-m. (2020). Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus. *arXiv preprint arXiv:2006.06890*.

[64] Min, S. W., Wu, K., Huang, S., Hidayetoğlu, M., Xiong, J., Ebrahimi, E., Chen, D., and Hwu, W.-m. (2021). Large graph convolutional network training with gpu-oriented data communication architecture. *arXiv preprint arXiv:2103.03330*.

[65] Minin, A. and Tsyganov, K. (2023). On the way towards agi: Human intelligence through llms perspective. *Blog Post*.

[66] Nagel, M., Fournarakis, M., Amjad, R. A., Bondarenko, Y., Van Baalen, M., and Blankevoort, T. (2021). A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*.

[67] Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., et al. (2017a). Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*.

[68] Neubig, G., Goldberg, Y., and Dyer, C. (2017b). On-the-fly operation batching in dynamic computation graphs. In *Advances in Neural Information Processing Systems*, pages 3974–3984.

[69] NVIDIA (2017a). CUDA Unified Memory. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[70] NVIDIA (2017b). CUTLASS. https://github.com/NVIDIA/cutlass.

[71] NVIDIA (2018). Triton Inference Server. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html.

[72] NVIDIA (2021). FasterTransformer. https://github.com/NVIDIA/FasterTransformer.

[73] NVIDIA (2023). TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM.

[74] Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., Rajashekhar, V., Ramesh, S., and Soyke, J. (2017a). Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*.

[75] Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., Rajashekhar, V., Ramesh, S., and Soyke, J. (2017b). Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*.

[76] OpenAI (2022). ChatGPT. https://chat.openai.com/chat.

[77] Pang, B., Lee, L., and Vaithyanathan, S. (2002). Thumbs up?: sentiment classification using machine learning techniques. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 79–86. Association for Computational Linguistics.

[78] Paszke, A., Gross, S., Chintala, S., and Chanan, G. (2017). Pytorch. http://pytorch.org/.

[79] PATEL, D. and AHMAD, A. (2023). Peeling the onion's layers - large language models search architecture and cost. *Blog Post*.

[80] Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Heek, J., Xiao, K., Agrawal, S., and Dean, J. (2023). Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5.

[81] Qiao, L., Raman, V., Reiss, F., Haas, P. J., and Lohman, G. M. (2008). Main-memory scan sharing for multi-core cpus. *Proceedings of the VLDB Endowment*, 1(1):610–621.

[82] Rabe, M. N. and Staats, C. (2021). Self-attention does not need $O(n^2)$ memory. *arXiv preprint arXiv:2112.05682*.

[83] Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. (2018). Improving language understanding by generative pre-training.

[84] Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. (2021). ZeRO-Offload: Democratizing Billion-Scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564.

[85] Romero, F., Li, Q., Yadwadkar, N. J., and Kozyrakis, C. (2021). {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411.

[86] ShareGPT (2023). https://sharegpt.com/.

[87] Shen, H., Chen, L., Jin, Y., Zhao, L., Kong, B., Philipose, M., Krishnamurthy, A., and Sundaram, R. (2019). Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337.

[88] Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Fu, D. Y., Xie, Z., Chen, B., Barrett, C., Gonzalez, J. E., et al. (2023). High-throughput generative inference of large language models with a single gpu. *arXiv preprint arXiv:2303.06865*.

[89] Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. (2019). Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.

[90] Socher, R., Lin, C. C., Manning, C., and Ng, A. Y. (2011). Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 129–136.

[91] Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., and Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642.

[92] Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

[93] Tai, K. S., Socher, R., and Manning, C. D. (2015). Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*.

[94] Tan, M., Santos, C. d., Xiang, B., and Zhou, B. (2015). Lstm-based deep learning models for non-factoid answer selection. *arXiv preprint arXiv:1511.04108*.

[95] Thakur, N. (2023). The mind-boggling processing power and cost behind chat gpt: What it takes to build the ultimate ai chatbot? *Blog Post*.

[96] Tokui, S., Oono, K., Hido, S., and Clayton, J. (2015). Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5.

[97] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. (2023a). Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

[98] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. (2023b). Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

[99] TUG (2015). Wmt15 machine translation task. http://www.statmt.org/wmt15/translation-task.html.

[100] Unterbrunner, P., Giannikis, G., Alonso, G., Fauser, D., and Kossmann, D. (2009). Predictable performance for unpredictable workloads. *Proceedings of the VLDB Endowment*, 2(1):706–717.

[101] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

[102] Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2015). Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164.

[103] Wang, X., Xiong, Y., Wei, Y., Wang, M., and Li, L. (2021). LightSeq: A high performance inference library for transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers (NAACL-HLT)*, pages 113–120. Association for Computational Linguistics.

[104] Wang, Y., Chen, K., Tan, H., and Guo, K. (2023). Tabi: An efficient multi-level inference system for large language models. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 233–248.

[105] Welsh, M., Culler, D., and Brewer, E. (2001). Seda: An architecture for well-conditioned, scalable internet services. *ACM SIGOPS operating systems review*, 35(5):230–243.

[106] Wu, S., Irsoy, O., Lu, S., Dabravolski, V., Dredze, M., Gehrmann, S., Kambadur, P., Rosenberg, D., and Mann, G. (2023a). Bloomberggpt: A large language model for finance. *arXiv preprint arXiv:2303.17564*.

[107] Wu, T., He, S., Liu, J., Sun, S., Liu, K., Han, Q.-L., and Tang, Y. (2023b). A brief overview of chatgpt: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica*, 10(5):1122–1136.

[108] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

[109] Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. (2023). Smoothquant: Accurate and efficient post-training quantization for large language models. *International Conference on Machine Learning (ICML)*.

[110] Yin, J., Jiang, X., Lu, Z., Shang, L., Li, H., and Li, X. (2015). Neural generative question answering. *arXiv preprint arXiv:1512.01337*.

[111] Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. (2022). Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538.

[112] Zhang, B. and Sennrich, R. (2019). Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32.

[113] Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. (2022). OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*.