



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

Ph.D. DISSERTATION

# Designing Scalable Computer Systems to Accelerate Heterogeneous NLP Models

이종 자연어 처리 모델을 위한  
확장형 컴퓨터 시스템 설계

BY

Kim Joonsung

FEBRUARY 2021

DEPARTMENT OF ELECTRICAL ENGINEERING AND  
COMPUTER SCIENCE  
COLLEGE OF ENGINEERING  
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

# Designing Scalable Computer Systems to Accelerate Heterogeneous NLP Models

이종 자연어 처리 모델을 위한  
확장형 컴퓨터 시스템 설계

BY

Kim Joonsung

FEBRUARY 2021

DEPARTMENT OF ELECTRICAL ENGINEERING AND  
COMPUTER SCIENCE  
COLLEGE OF ENGINEERING  
SEOUL NATIONAL UNIVERSITY

# Designing Scalable Computer Systems to Accelerate Heterogeneous NLP Models

이종 자연어 처리 모델을 위한  
확장형 컴퓨터 시스템 설계

지도교수 김 장 우  
이 논문을 공학박사 학위논문으로 제출함

2020년 12월

서울대학교 대학원






전기 컴퓨터 공학부

김 준 성

김준성의 공학박사 학위 논문을 인준함

2020년 12월

위 원 장:	이 재 욱
부위원장:	김 장 우
위 원:	김 한 준
위 원:	김 재 준
위 원:	심 재 응

(인)   
(인)   
(인)   
(인)   
(인) 

# Abstract

Modern neural-network (NN) accelerators have been successful by accelerating a small number of basic operations (e.g., convolution, fully-connected, feedback) comprising the specific target neural-network models (e.g., CNN, RNN). However, this approach no longer works for the emerging full-scale natural language processing (NLP)-based neural network models (e.g., *Memory networks*, *Transformer*, *BERT*), which consist of different combinations of complex and heterogeneous operations (e.g., self-attention, multi-head attention, large-scale feed-forward). Existing acceleration proposals cover only the proposal-specific basic operations and/or customize them for specific models only, which leads to the low performance improvement and the narrow model coverage. Therefore, an ideal NLP accelerator should first identify all performance-critical operations required by different NLP models and support them as a single accelerator to achieve a high model coverage, and can adaptively optimize its architecture to achieve the best performance for the given model.

To address these scalability and model/config diversity issues, the dissertation introduces two novel projects (i.e., *MnnFast* and *NLP-Fast*) to efficiently accelerate a wide spectrum of full-scale NLP models. First, *MnnFast* proposes three novel optimizations to resolve three major performance problems (i.e., *high memory bandwidth*, *heavy computation*, and *cache contention*) in memory-augmented neural networks. Next, *NLP-Fast* adopts three optimization techniques to resolve the huge performance variation due to the model/config diversity in emerging NLP models. We implement both *MnnFast* and *NLP-Fast* on different hardware platforms (i.e., CPU, GPU, FPGA) and thoroughly evaluate their performance improvement on each platform.

**keywords:** Hardware-Software Co-Design, Natural Language Processing (NLP), AI Accelerator, Machine Learning, Computer Architecture

**student number:** 2017-36250

# Contents

<b>Abstract</b>	<b>i</b>
<b>Contents</b>	<b>ii</b>
<b>List of Tables</b>	<b>v</b>
<b>List of Figures</b>	<b>vi</b>
<b>1 INTRODUCTION</b>	<b>1</b>
<b>2 Background</b>	<b>6</b>
2.1 Memory Networks . . . . .	6
2.2 Deep Learning for NLP . . . . .	9
<b>3 A Fast and Scalable System Architecture for Memory-Augmented Neural Networks</b>	<b>14</b>
3.1 Motivation & Design Goals . . . . .	14
3.1.1 Performance Problems in MemNN – High Off-chip Memory Bandwidth Requirements . . . . .	15
3.1.2 Performance Problems in MemNN – High Computation . . . . .	16
3.1.3 Performance Problems in MemNN – Shared Cache Contention . . . . .	17
3.1.4 Design Goals . . . . .	18
3.2 MnnFast . . . . .	19

3.2.1	Column-Based Algorithm . . . . .	19
3.2.2	Zero Skipping . . . . .	22
3.2.3	Embedding Cache . . . . .	25
3.3	Implementation . . . . .	26
3.3.1	General-Purpose Architecture – CPU . . . . .	26
3.3.2	General-Purpose Architecture – GPU . . . . .	28
3.3.3	Custom Hardware (FPGA) . . . . .	29
3.4	Evaluation . . . . .	31
3.4.1	Experimental Setup . . . . .	31
3.4.2	CPU . . . . .	33
3.4.3	GPU . . . . .	35
3.4.4	FPGA . . . . .	37
3.4.5	Comparison Between CPU and FPGA . . . . .	39
3.5	Conclusion . . . . .	39

## **4 A Fast, Scalable, and Flexible System for Large-Scale Heterogeneous NLP**

<b>Models</b>		<b>40</b>
4.1	Motivation & Design Goals . . . . .	40
4.1.1	High Model Complexity . . . . .	40
4.1.2	High Memory Bandwidth . . . . .	41
4.1.3	Heavy Computation . . . . .	42
4.1.4	Huge Performance Variation . . . . .	43
4.1.5	Design Goals . . . . .	43
4.2	NLP-Fast . . . . .	44
4.2.1	Bottleneck Analysis of NLP Models . . . . .	44
4.2.2	Holistic Model Partitioning . . . . .	47
4.2.3	Cross-operation Zero Skipping . . . . .	51
4.2.4	Adaptive Hardware Reconfiguration . . . . .	54
4.3	NLP-Fast Toolkit . . . . .	56



4.4	Implementation . . . . .	59
4.4.1	General-Purpose Architecture – CPU . . . . .	59
4.4.2	General-Purpose Architecture – GPU . . . . .	61
4.4.3	Custom Hardware (FPGA) . . . . .	62
4.5	Evaluation . . . . .	64
4.5.1	Experimental Setup . . . . .	65
4.5.2	CPU . . . . .	65
4.5.3	GPU . . . . .	67
4.5.4	FPGA . . . . .	69
4.6	Conclusion . . . . .	72
<b>5</b>	<b>Related Work</b>	<b>73</b>
5.1	Various DNN Accelerators . . . . .	73
5.2	Various NLP Accelerators . . . . .	74
5.3	Model Partitioning . . . . .	75
5.4	Approximation . . . . .	76
5.5	Improving Flexibility . . . . .	78
5.6	Resource Optimization . . . . .	78
<b>6</b>	<b>Conclusion</b>	<b>80</b>
	<b>Abstract (In Korean)</b>	<b>106</b>

# List of Tables

2.1	Representative models in emerging NLP models and their key computational components. . . . .	10
3.1	Memory networks configuration for the evaluation. . . . .	32
4.1	Time and space complexity of each key operation commonly used in emerging NLP models. . . . .	45
4.2	Base configurations of each NLP model. . . . .	64

# List of Figures

1.1	(a) shows an example story with a question. (b) shows where the memory network stores the story, and how it processes the question to derive the answer. . . . .	2
2.1	Computational steps of memory networks (MemNN). MemNN consists of embedding, input memory representation, output memory representation and output calculation. $nw$ means the maximum number of words in a sentence. $nq$ and $ns$ are the number of questions and given story sentences, respectively. $ed$ is the embedding dimension. . . . .	7
2.2	Computational steps of key components used in state-of-the-art NLP models. . . . .	11
3.1	Limited scalability due to memory bandwidth. The speedup results of each channel configuration are normalized to the corresponding single-thread result. . . . .	15
3.2	Performance degradation due to co-executed embedding threads. The slowdown results are relative to the corresponding 1-embedding thread cases. Embedding threads contend with inferencing threads for shared memory system, thus reducing the performance of MemNN. . . . .	17
3.3	Dataflow comparison between the baseline and the column-based algorithm. . . . .	20

3.4	Probability value distribution. Each column represents the probability vector to each question. We use the Facebook bAbi dataset and test-set [135]. . . . .	22
3.5	Tradeoffs between accuracy loss and computation reduction according to the skip threshold. . . . .	24
3.6	A high-level architecture of FPGA-based MnnFast. . . . .	30
3.7	Performance of column-based algorithm on CPU. . . . .	33
3.8	Scalability of column-based algorithm on CPU. . . . .	35
3.9	The number of off-chip memory accesses on CPU. . . . .	36
3.10	Scalability of column-based algorithm on GPU. . . . .	37
3.11	Latency reduction of FPGA-based MnnFast. Each latency is normalized to the baseline. . . . .	38
3.12	Effectiveness of embedding cache in FPGA-based MnnFast. Each latency result is normalized to the No Cache. . . . .	38
4.1	Limited scalability due to memory bandwidth. The speedup results of each channel configuration are normalized to the corresponding single-thread result. . . . .	41
4.2	Performance breakdown of NLP models with various parameter configurations. <i>base</i> configurations are in the evaluation section (i.e., <i>experimental setup</i> ), and S/H/F-4x are configurations with a fourfold increase of $s$ , $d_H$ , and $d_{FF}$ , respectively. . . . .	46
4.3	Dataflow comparison between baseline and our optimizations: <i>partial-head update</i> (P), <i>column-based algorithm</i> (C), and <i>feed-forward splitting</i> (F). . . . .	48
4.4	Probability value distribution. Each column is the probability vector of each query. We randomly choose 512 queries in BERT during inference on SQuAD. . . . .	51

4.5	Tradeoffs between accuracy loss and computation reduction according to the skip threshold. . . . .	52
4.6	Stall-minimized resource rebalancing. C-n and M-n represent the execution time of compute and memory parts in $n^{th}$ operation, respectively. . . . .	56
4.7	The overview of our NLP-Fast toolkit. The left figure shows an example of applying NLP-Fast’s optimizations by using the pre-implemented NLP-Fast libraries. The right figure presents the overview of bottleneck analysis for given NLP models with various configurations. . . . .	57
4.8	Performance improvement on a single GPU with multiple CUDA streams. NLP-Fast’s model partitioning enables GPU to exploit CUDA streams. . . . .	61
4.9	The architecture of FPGA-based NLP-Fast. . . . .	63
4.10	Performance improvement of CPU-based NLP-Fast on various NLP models and configurations. MP means <i>model partitioning</i> . . . . .	66
4.11	Scalability on different memory bandwidth. . . . .	66
4.12	Normalized LLC MPKI for each partitioning optimization. P/C/F means partial-head update, column-based algorithm, and feed-forward splitting. . . . .	67
4.13	Single-GPU performance improvement of GPU-based NLP-Fast on various NLP models. . . . .	68
4.14	The overhead analysis of multi-GPU version of NLP-Fast (NE) and the expected speedup of NLP-Fast with high bandwidth (e.g., NVLink 2.0). . . . .	68
4.15	Performance of FPGA-based NLP-Fast on various models. MP means model partitioning. . . . .	69
4.16	Latency reduction of FPGA-based NLP-Fast on BERT. Each latency is normalized to baseline. P/C/Z/F represents partial-head update/column-based algorithm/zero skipping/feed-forward splitting, respectively. . . . .	70

4.17 Effectiveness of *adaptive hardware reconfiguration* on various configurations of BERT. S/H/F-4x are parameter configurations with a four-fold increase of  $s$ ,  $d_H$ , and  $d_{FF}$ , respectively. . . . . 71

# Chapter 1

## INTRODUCTION

Recently, the rapid advancement of natural language processing (NLP) attracts massive attention from both industry and research as the technology enhancement unlocks some new products and services. For example, big technology companies (e.g., Google, Facebook) continuously develop new types of NLP workloads, which leads to technological breakthroughs. Also, many researchers and open-source communities propose specific NLP workloads for their own purpose.

*Memory-augmented neural networks* (MemNN) are getting more attention from NLP researchers as they dramatically increase the accuracy of state-of-the-art NLP tasks [130, 101, 100]. In contrast to typical neural networks (e.g., DNNs, CNNs, RNNs), these memory-augmented neural networks can discretely read and write tokens (e.g., words, sentences) from and to an external memory, which provides the capability to make an inference with the previous history stored in memory. Also, the memory-augmented neural networks exploit the *attention mechanism* which allows a model to learn interdependence between input and output tokens. Thanks to these powerful context-aware information processing capabilities, the attention-based neural networks successfully become one of the most popular neural networks for NLP researchers.

Figure 1.1 shows how MemNN processes a story with a question and an answer. In

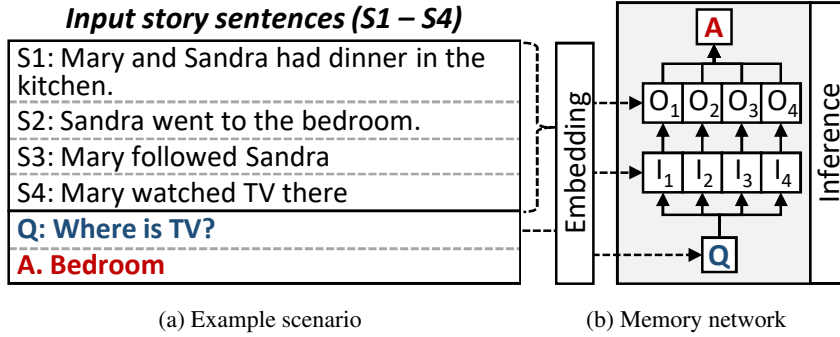


Figure 1.1: (a) shows an example story with a question. (b) shows where the memory network stores the story, and how it processes the question to derive the answer.

this example, MemNN first receives a four-sentence story and stores it in its memory. Next, it receives a question asking the location of TV, which can be answered only by understanding the story (e.g., the order of sentences, the relation of words across the sentences). To enable such context-aware information processing, MemNN performs an inference by utilizing the information stored in the memory against the question. In this process, the question and input sentences are converted into internal state values (i.e.,  $Q, I_i, O_i$ ), and these values are stored into memory components.

To improve the reasoning power, MemNN needs to increase the size of memory and train the network with a large-scale dataset. In fact, recent studies propose *large-scale memory networks* to support growing demands for large-scale question answering tasks, but performed within the target latency [17].

These increasing demands to perform a *large-scale Q/A* task within the target latency require a *fast* and *scalable* computer infrastructure; however, the current system architecture does not provide the expected scalability due to the following reasons. First, a large-scale MemNN can suffer from the increasing number of cache misses as the data do not fit into the cache. Second, when MemNN goes through a memory-intensive phase (i.e., the embedding operation), the increased number of DRAM accesses can degrade the overall performance significantly. Third, when MemNN goes



through a compute-intensive phase (i.e., the inference operation), the infrastructure can be short of the required computing resources. Lastly, a large-scale MemNN can suffer from a significant number of cache conflicts when different operations contend for the shared cache (e.g., embedding vs. inference).

In addition to this scalability issue, recent emerging NLP workloads have huge *model/config diversity* as researchers propose various types of network models according to different purposes. For example, Memory networks, developed by Facebook, is used in a simple language question answering system and language modeling tasks. Transformer, developed by Google, is proposed to solve many sequence transduction tasks (e.g., language translation). BERT, also proposed by Google, causes a stir in the NLP community by providing state-of-the-art results in a wide variety of NLP tasks (e.g., question answering, natural language inference, GLUE benchmarks).

This huge model/config diversity makes the current computer infrastructure difficult to provide enough scalability due to the following reasons. First, there is no work endeavoring to *holistically* optimize all operations in the NLP workloads. Second, a wide variety of parameter configurations incurs a huge *performance variation*, which makes the current system architecture difficult to find out an optimal design point. ***Therefore, we need a fast, scalable, and flexible system architecture for emerging heterogeneous NLP workloads.***

In this dissertation, we introduce two novel schemes (i.e., ***MnnFast, NLP-Fast***) to address aforementioned scalability and diversity problems in recent NLP models. We first present MnnFast a novel large-scale MemNN system architecture to achieve fast and scalable reasoning performance. MnnFast adopts three novel optimizations: *column-based algorithm with streaming, zero-skipping, and embedding cache*. To reduce the memory bandwidth overhead, MnnFast applies a modified memory-access algorithm (called *column-based algorithm*) to minimize the size of data spills and enable more efficient data chunking by transforming a large-scale memory access to many parallelized small-scale memory accesses. With the column-based algorithm ap-

plied, MnnFast can further improve its performance by performing data computation and prefetching data required for the next calculation in parallel (called *streaming optimization*). To reduce the computation overhead, MnnFast applies an optimization technique (called *zero-skipping*) to bypass computations dealing with zero or near-zero numbers stored in the memory. To solve the cache contention problem, MnnFast can make memory-intensive embedding operations either bypass the cache or stored in a dedicated memory (called *embedding cache*).

Next, we show NLP-Fast, a novel *fast, scalable, and adaptive* system architecture for the emerging NLP workloads. To cover various types of emerging NLP workloads, we extract common operations used in the NLP workloads by conducting extensive profiling and static analysis. Then, we propose three novel optimizations: *holistic model partitioning*, *cross-operation zero-skipping*, and *model/config-adaptive hardware reconfiguration*. To holistically reduce the memory accessing overhead, NLP-Fast applies three novel model partitioning techniques (i.e., *partial-head update*, *column-based algorithm*, *feed-forward splitting*), which can cover all types of operations in state-of-the-art NLP workloads. These model partitioning techniques minimize the size of data spills and enable the current infrastructure to hide most memory accessing overhead by significantly reducing the working set size. To reduce the computation overhead, NLP-Fast applies an optimization technique (called *cross-operation zero skipping*) to bypass computations dealing with zero or near-zero values stored in the memory. Also, NLP-Fast carefully manages an execution time skewness caused by the zero-skipping optimization to maximize resource utilization. To increase the system flexibility, we provide further optimization (called *model/config-adaptive hardware reconfiguration*) to fully leverage hardware accelerators. Our adaptive hardware reconfiguration helps an HW accelerator to achieve full potential performance by finding an optimal design point.

For the evaluation, we implement both MnnFast and NLP-Fast on top of various platforms: CPU, GPU, and FPGA. For MnnFast, we first present the results of CPU-

based MnnFast with extensive profiling and analysis: memory throttling test and cache statistics. Next, we implement GPU-based MnnFast and show that our optimizations can improve single-GPU performance as well as achieve the scalable performance in the multi-GPU environment. Lastly, we build FPGA-based MnnFast with the embedding cache and measure its performance and energy efficiency.

For NLP-Fast, we present the similar results for CPU-based and GPU-based NLP-Fast (i.e., extensive profiling and analysis for CPU, scalable performance improvement for GPU). Here, we highlight the results of FPGA-based NLP-Fast with the model/config-adaptive hardware reconfiguration and measure its performance.

The rest of the dissertation is organized as follows. Chapter 2 explains the characteristics of memory networks and the state-of-the-art deep learning approaches for NLP workloads. Chapter 3 describes how MnnFast solves the performance problems to achieve the scalable performance in memory networks. Chapter 4 shows how NLP-Fast provides enough scalability for heterogeneous NLP models on various hardware platforms. Finally, Section 5 and Section 6 provide related work and conclusion, respectively.

## Chapter 2

### Background

In this section, we introduce a representative memory-augmented neural network, *memory networks* developed by Facebook [125, 136] (Section 2.1). We then provide state-of-the-art neural networks (including memory networks) widely used in NLP field (Section 2.2).

#### 2.1 Memory Networks

Neural networks have shown high accuracy comparable to the human on image classification and speech recognition. Recurrent neural networks (RNNs), designed to work on sequence prediction problems, derive an answer to a question from the previous reasonings [86, 57]. However, RNNs cannot memorize the previous history for a long time [15] nor handle a large amount of history due to their small memory [136]. Therefore, they cannot perform sophisticated tasks requiring a large amount of memory, such as a task which comprehends a series of books to provide useful information to users.

*Memory networks* (MemNNs), developed by Facebook, solve the problem of RNNs by augmenting neural networks with external memory [136, 125]. The large-scale external memory allows MemNN to solve the sophisticated tasks. Nowadays, thanks to its huge reasoning power, MemNN is widely used in various fields from simple dialog

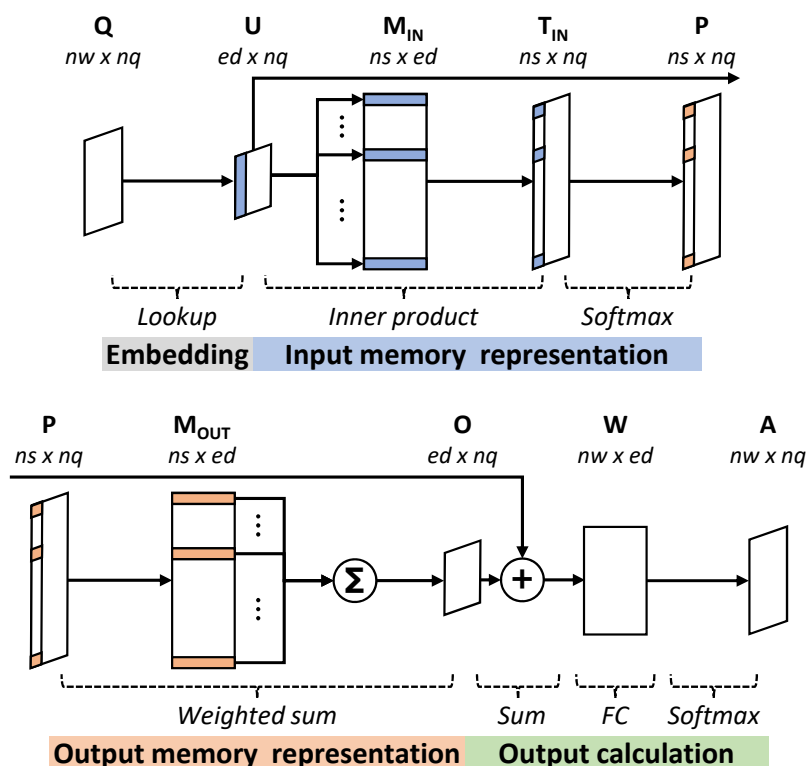


Figure 2.1: Computational steps of memory networks (MemNN). MemNN consists of embedding, input memory representation, output memory representation and output calculation.  $nw$  means the maximum number of words in a sentence.  $nq$  and  $ns$  are the number of questions and given story sentences, respectively.  $ed$  is the embedding dimension.

comprehension to large-scale question & answering system using a large-scale dataset (e.g., Wikipedia) [134, 125, 55, 34, 87]

Figure 2.1 shows a high-level overview of MemNN’s computational structure. MemNN consists of two major operations: *embedding* and *inference*. The embedding operation converts a given sentence into an internal state. MemNN first converts story and question sentences into internal states, and these states are stored into input/output memory ( $M_{IN}/M_{OUT}$ ) and question state memory ( $U$ ), respectively. The inference operation calculates answers to each question by going through multiple different types

of computational layers: *input memory representation*, *output memory representation*, and *output calculation*. By doing so, MemNN successfully reasons out answers by exploiting the large-scale memory components. The following paragraphs describe each operation and their characteristics in more details.

**Embedding operation.** The main purpose of the embedding operation is converting an input sentence into a representative internal state vector (of size  $1 \times ed$ ). First, MemNN extracts features from texts by using a bag-of-words (BoW) model [109]. By following the BoW model, MemNN embeds each word by looking up a vector from an embedding matrix (of size  $ed \times V$ ,  $V$  is the number of words in a dictionary) and sums the resulting vectors.<sup>1</sup> Internal state vectors of story sentences are stored into matrices  $M_{IN}$  and  $M_{OUT}$ , and question sentences are also embedded in a matrix  $U$  (Figure 2.1).

**Inference operation.** With the extracted internal states, MemNN calculates an answer during the inference operation which consists of three computational steps: input memory representation, output memory representation, and output calculation.

First, in the input memory representation step, MemNN calculates a probability vector, *p-vector*, which represents the correlation between the given question and each story sentence.

$$p = \text{Softmax}(u \times M_{IN}) \quad (2.1)$$

Equation (2.1) shows how to compute the p-vector. MemNN computes the p-vector by calculating dot product of the internal state vector of a question ( $u$ ) and each memory state vector in the input matrix ( $M_{IN}$ ) and applying a softmax function ( $\text{Softmax}(x_i) = e^{x_i} / \sum_j e^{x_j}$ ) to the results of the dot products. By doing so, MemNN can extract the correlation of questions and sentences.

Second, in the output memory representation step, MemNN calculates a weighted sum of an output memory. The output memory ( $M_{OUT}$ ) holds internal state vectors corresponding to given story sentences. MemNN computes a sum over these output

---

<sup>1</sup>Some studies multiply position weights to vectors before the sum of all vectors to preserve the order of words in the sentence.

vectors ( $m_i^{OUT}$ ) weighted by the probability value ( $p_i$ ) extracted from the previous step:

$$o = \sum_i p_i m_i^{OUT} \quad (2.2)$$

The resulting vector  $o$  is called a response vector, and the response vector is delivered to the output calculation step to make the final answer.

In the output calculation step, MemNN generates the final prediction for the given question. It computes the sum of the response vector  $o$  and the question vector  $u$ , and the outcome passes through the fully connected (FC) layer with a weight matrix ( $W$ ). Depending on the applications, the input memory and output memory representation steps iterate over several times for better results, followed by the FC layer and the softmax function.

## 2.2 Deep Learning for NLP

Recently, neural networks have risen as new information processing paradigms in natural language processing (NLP) as they break records on many NLP tasks. In the early period, *simple* neural networks (e.g., RNNs) are used for NLP tasks; however, they cannot perform sophisticated tasks due to their limited capability of memorizing the previous history. To resolve these limitations, big technology companies (e.g., Google, Facebook, Microsoft) and open-source research communities (e.g., OpenAI) actively propose new types of *complex* NLP models (e.g., memory-augmented neural networks [136, 125], attention-based neural networks [128, 32]). Nowadays, these complex emerging NLP models are widely used in various NLP tasks from simple dialog comprehension to large-scale question & answering system using a large-scale dataset (e.g., Wikipedia) [134, 55, 34, 87].

We group emerging NLP models into three representative models (i.e., Memory networks, Transformer, BERT) and classify their key computational components by conducting extensive profiling and static analysis. Table 2.1 shows the category of

Table 2.1: Representative models in emerging NLP models and their key computational components.

	Key Computational Components			
	Attention mechanism	Multi-head attention	Multi-head self-attention	Feed forward
Memory networks [136, 125, 17, 87, 72]	✓			
Transformer [128, 29, 141, 98, 12, 68]		✓ <sup>†</sup>	✓	✓
BERT [32, 73, 79, 149, 28, 66]			✓	✓

<sup>†</sup> Encoder-decoder attention layer in the transformer decoder.

recent NLP models and their key computational components: *attention mechanism*, *multi-head attention*, *multi-head self-attention*, and *feed-forward network*.

*Memory networks* consists of multiple consecutive attention mechanisms followed by a simple fully-connected layer with softmax. *Transformer* has an encoder-decoder structure. The encoder consists of two components (i.e., multi-head self-attention, feed-forward network), and the decoder comprises three components: two components in the encoder with an additional component (i.e., multi-head attention). *BERT* consists of multiple Transformer encoders. Different from the Transformer, BERT uses GELU (not ReLU) as an activation function in the feed-forward network.

In Figure 2.2, we illustrate how each computational component operates. Figure 2.2a shows the *attention mechanism*. There are three input matrices in the attention mechanism: a query matrix ( $\mathbf{Q} \in \mathbb{R}^{nq \times d}$ ), a key matrix ( $\mathbf{K} \in \mathbb{R}^{ns \times d}$ ), and a value matrix ( $\mathbf{V} \in \mathbb{R}^{ns \times d}$ ). Here,  $nq$  and  $ns$  represent the number of queries and key-value pairs respectively, and  $d$  is the dimension of internal states. The query matrix passes through



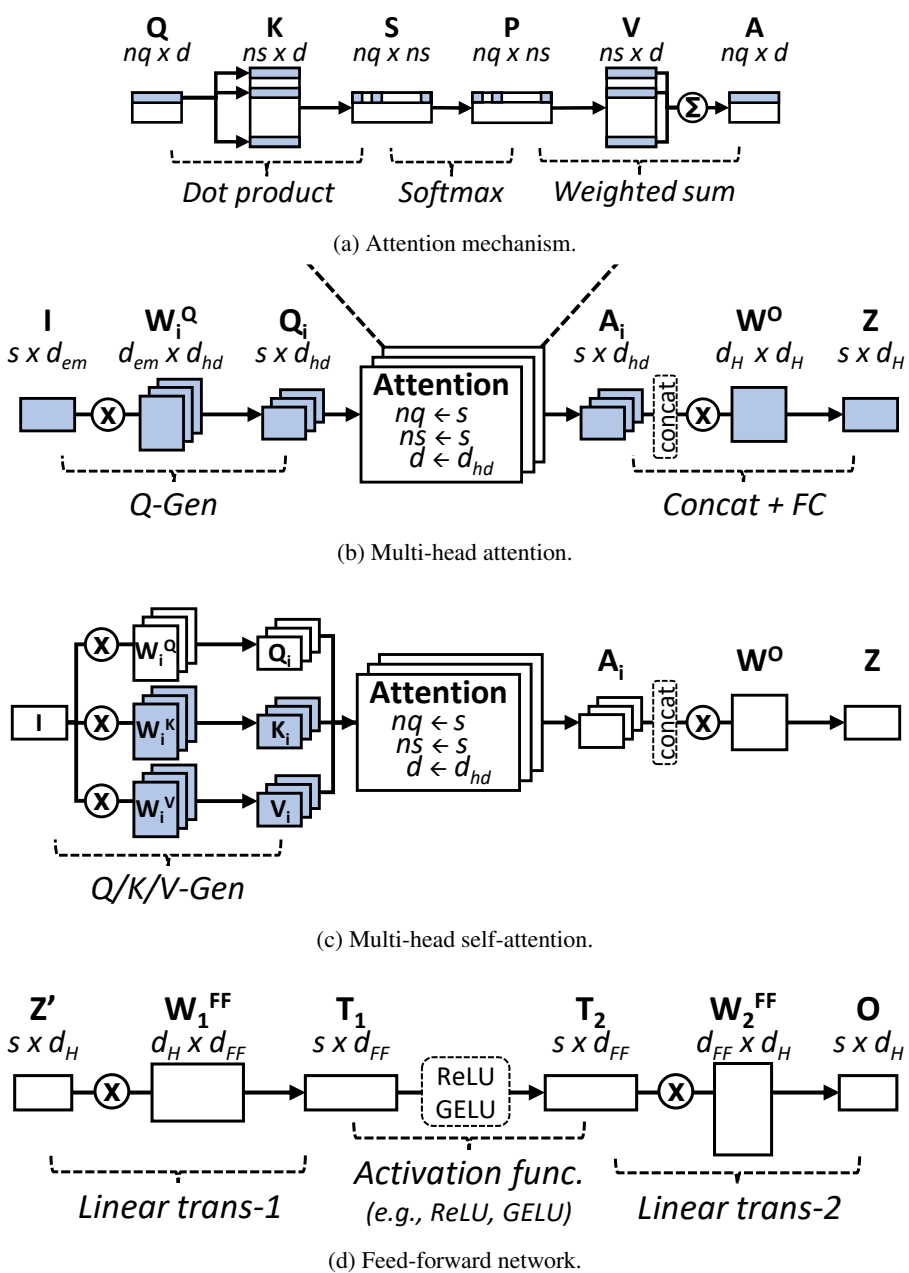


Figure 2.2: Computational steps of key components used in state-of-the-art NLP models.

three operations (i.e., *dot product*, *softmax*, *weighted sum*) to calculate an attention result for each query. Equation (2.3) shows how to compute the attention result (A).

$$A = \text{Attention}(Q, K, V) = \text{Softmax}(Q \times K^T) \times V \quad (2.3)$$

The attention mechanism first computes a score matrix (S) by computing dot products of each query with all keys. Next, it calculates a probability matrix (P) by applying a softmax function ( $\text{Softmax}(x_i) = e^{x_i} / \sum_j e^{x_j}$ ) to the score matrix. Then, the attention mechanism computes a sum of values weighted by these probabilities to calculate the attention result (A).

Figure 2.2b illustrates how the *multi-head attention* works. The multi-head attention computes different versions of attention results ( $A_i$ ) for each head, concatenates the attention results, and applies a fully-connected (FC) operation (called *attention FC*) to the concatenated results for calculating the final output (Z). Equation (2.4) shows the details.

$$Z = \text{MultiHead}(I, K, V) = \text{Concat}(\{\text{head}_i\}_{i=1}^h) \times W^O \quad (2.4)$$

$$\text{head}_i = \text{Attention}(I \times W_i^Q, k_i, v_i)$$

There are three input matrices in the multi-head attention: an input matrix ( $I \in \mathbb{R}^{s \times d_{em}}$ ), a key matrix ( $K \in \mathbb{R}^{s \times d_H}$ ), and a value matrix ( $V \in \mathbb{R}^{s \times d_H}$ ). Both key and value matrices are split into  $h$  sub-key/value matrices ( $k_i \in \mathbb{R}^{s \times d_{hd}}$ ,  $v_i \in \mathbb{R}^{s \times d_{hd}}$ ), respectively. Here,  $h$  is the number of heads,  $s$  means a sequence length in the model, and  $d_{em}$  and  $d_H$  are the dimensions of input and internal states, respectively. For each attention head,  $d_H$  is mapped into the lower dimension of size  $d_{hd}$ . Different from the attention mechanism, the multi-head attention conducts a query generation (*Q-Gen*) to calculate the query matrix from an input ( $I \times W_i^Q$ ).

Figure 2.2c shows the *multi-head self-attention*. Compared to the multi-head attention (which needs three input matrices), the multi-head self-attention requires only one input matrix (I). Instead, it generates query/key/value matrices from the input matrix (*Q/K/V-Gen*) as Equation (2.5).

$$Z = \text{MultiHeadSelf}(I) = \text{Concat}(\{\text{head}_i\}_{i=1}^h) \times W^O \quad (2.5)$$

$$\text{head}_i = \text{Attention}(I \times W_i^Q, I \times W_i^K, I \times W_i^V)$$

In addition to attention components, state-of-the-art NLP models also contain a *feed-forward network*. Figure 2.2d shows the high-level overview of the feed-forward network (FFN) commonly used in NLP models. There are two weight matrices for two linear transformations ( $W_1^{FF} \in \mathbb{R}^{d_H \times d_{FF}}$ ,  $W_2^{FF} \in \mathbb{R}^{d_{FF} \times d_H}$ ). Equation (2.6) shows the details.

$$O = \text{FFN}(Z') = \text{ActFunc}(Z' \times W_1^{FF}) \times W_2^{FF} \quad (2.6)$$

An activation function (*ActFunc*) may differ for each type of NLP models. For example, the Transformer uses *ReLU* as the activation function while BERT uses *GELU*.

## Chapter 3

# A Fast and Scalable System Architecture for Memory-Augmented Neural Networks

### 3.1 Motivation & Design Goals

Researchers start exploiting the huge reasoning power of MemNN to solve sophisticated problems such as large-scale question answering tasks. To solve such complex problems, MemNN becomes bigger and turns into a large-scale memory network, demanding larger embedding dimension ( $ed$ ) and more input sentences ( $ns$ ) [17, 18, 55, 34, 87, 76]. The large-scale memory networks require high-scalability to handle the increasing computation and memory demands. The current MemNN, however, cannot achieve scalability for three reasons: *high memory bandwidth*, *heavy computation*, and *cache contention*.

In this section, we show the major performance bottlenecks in the state-of-the-art MemNN [125]. We first explain how memory bandwidth affects the overall performance (Section 3.1.1). Next, we provide the characteristics of MemNN computation, which requires huge compute resources (Section 3.1.2). Lastly, we show cache contention between the inference and embedding operations and quantify its performance impacts (Section 3.1.3).

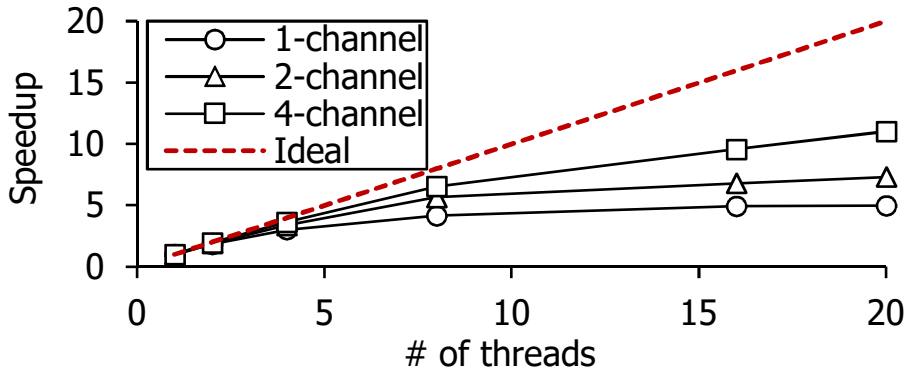


Figure 3.1: Limited scalability due to memory bandwidth. The speedup results of each channel configuration are normalized to the corresponding single-thread result.

### 3.1.1 Performance Problems in MemNN – High Off-chip Memory Bandwidth Requirements

MemNN requires a significant amount of memory bandwidth. For example, during the embedding operation, MemNN looks up the embedding matrix to convert input sentences into internal state vectors. Larger embedding dimension is good at solving complicated questions [17] but incurs higher memory pressure. Not only the embedding operation, but also the inference operation causes high memory traffic (e.g., input/output memory accesses, intermediate data spills). In the inference operation, MemNN has to load the whole input and output memory ( $M_{IN}$  and  $M_{OUT}$ , respectively) whose size is proportionate to the embedding dimension ( $ed$ ) and the number of story sentences ( $ns$ ). As the networks are getting larger, the size of these in/out memory is rapidly increasing. Furthermore, MemNN spills a large amount of intermediate data between each layer: *Inner product*, *Softmax*, and *Weighted sum* (Figure 2.1). These data spills are proportionate to  $ns$  as well. Therefore, their overheads will continuously increase.

Figure 3.1 shows how available memory bandwidth affects the scalability of MemNN. To prove the high memory bandwidth requirement is one of the key limiting factor, we measure the performance speedup with multiple threads, reducing memory bandwidth

(# of memory channels). Here, our experimental environment has enough CPU cores (i.e., Xeon E5-2650 v4 12C/24T 2x), so the computation does not be a performance bottleneck. We can observe that MemNN quickly reaches a performance saturation point as the bandwidth decreases; in other words, a large amount of memory bandwidth requirements prevent MemNN from achieving scalable performance.

To overcome the memory bandwidth problem, we need more efficient memory management mechanisms to achieve good scalability. Fortunately, we find out a novel method to reduce memory bandwidth requirements and hide memory accessing overheads in the background. We propose a new computation algorithm (called *column-based algorithm*) which minimizes the size of data spills, provides more efficient data chunking, and enable MemNN to hide most memory accessing overheads. We explain the algorithm in Section 3.2.1 for more details.

### 3.1.2 Performance Problems in MemNN – High Computation

MemNN is not only memory-intensive but also compute-intensive application. Analyzing the characteristics and types of computation, we find out MemNN consists of a few compute-intensive operations. For example, MemNN requires multiple matrix multiplication operations (i.e., *Inner product*, *Weighted sum*, *FC layer*) known as compute-intensive tasks. Also, the *Softmax* function uses exponentiation requiring a large number of integer multiplications.

Therefore, MemNN is challenging to achieve scalable performance due to its substantial computational overheads. The amount of computation in MemNN superlinearly increases as its time complexity is  $O(n^a)$  (where  $a \geq 2.375$ ) [137]. So, even if the number of CPU cores increases, the overall system will show sublinear performance. Also, we cannot easily scale-up CPU performance due to technology constraints; the overall performance will be quickly saturated.

To overcome the high computation problem, we need more powerful computing units (e.g., GPU, FPGA, ASIC) and optimization techniques to reduce the amount of

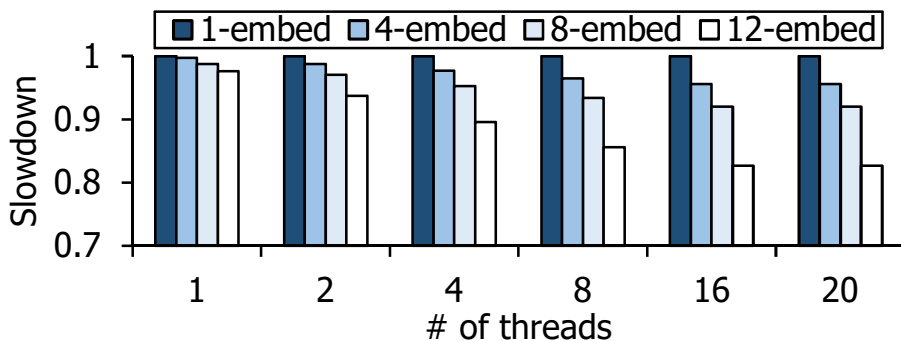


Figure 3.2: Performance degradation due to co-executed embedding threads. The slowdown results are relative to the corresponding 1-embedding thread cases. Embedding threads contend with inferencing threads for shared memory system, thus reducing the performance of MemNN.

computation. From an in-depth analysis, we find out the high potential for reducing the computation in the output memory representation step. The probability vector  $p$  represents the correlation between a question sentence and story sentences, and only a few story sentences are related to the given question; therefore, most values are close to zero. So, we propose a *zero-skipping* optimization to bypass a large amount of output computation. We explain the optimization in Section 3.2.2 for more details.

### 3.1.3 Performance Problems in MemNN – Shared Cache Contention

MemNN consists of the embedding and inference operations, and these two operations exhibit different characteristics: memory-intensive and compute-intensive, respectively. The inference operation has a large amount of computation; to efficiently handle such a large amount of computes with high throughput, they need to keep their necessary data in a shared cache as much as possible [49]. However, as multiple question answering tasks are executed simultaneously (multi-tenant setting), two operations contend for the shared cache, which results in the cache contention problem. The embedding operation accesses a large amount of data, polluting the shared cache.

Since the inference operation heavily uses the shared cache (e.g., OpenBLAS [49], Intel MKL [131]), the cache contention results in significant performance degradation.

Figure 3.2 shows the performance degradation due to the cache contention. We measure the cache contention’s impact on different scales of MemNN by varying the number of simultaneously-executed embedding operations. The impact increases with the scale of MemNN and the number of embedding operations, which indicates that we cannot simply scale up MemNN to meet the increasing demands due to the contention.

To overcome the cache contention problem, we should isolate memory accesses during the embedding operation from the entire memory accesses. We can simply apply cache bypassing techniques for CPU; however, it results in high latency overheads to the embedding operation and more memory pressure on off-chip DRAM. To minimize these overheads, we propose an *embedding cache* dedicated to efficiently cache the embedding matrix. We describe a caching policy and an architecture of the embedding cache in Section 3.2.3 for more details.

### 3.1.4 Design Goals

Based on the discussion of the performance problems so far, we set our key design goals and provide a brief description of our key ideas to achieve the scalable large-scale memNN.

- **Efficient memory management algorithm.** It should minimize the memory bandwidth requirements. We propose the *column-based algorithm* to eliminate the intermediate data spills.
- **Reduction of computation.** It should reduce the amount of computation. We propose the *zero-skipping* optimization to decrease the output computation by skipping output operations of near-zero probability values.
- **Shared cache isolation.** It should avoid cache contention between the embedding and inference operations. We propose the *embedding cache* dedicated for the embedding matrix.



## 3.2 MnnFast

### 3.2.1 Column-Based Algorithm

MemNN suffers from a large amount of off-chip memory bandwidth (Section 3.1.1), which results in poor scalability. The current algorithm (*baseline*) consecutively calculates each layer (i.e., in-memory dot product (*Inner product*), softmax for p-value (*Softmax*), weighted sum with out-memory (*Weighted sum*)), which generates a number of intermediate data spills between each layer. Since shared cache cannot afford to hold these intermediate data, the baseline MemNN necessarily flushes and re-reads those temporary data to and from off-chip DRAM. Not only intermediate data spills but the baseline MemNN also suffers from inefficient data chunking of current matrix multiplication libraries (e.g., OpenBLAS) as their data chunking mechanisms are not MemNN-friendly.

$$o = \sum_i \text{Softmax}(u \times m_i^{IN}) m_i^{OUT} = \sum_i \frac{e^{u \times m_i^{IN}} m_i^{OUT}}{\sum_j e^{u \times m_j^{IN}}} \quad (3.1)$$

Equation (3.1) shows how baseline MemNN computes the output vector. The baseline first calculates a probability vector ( $p$ ) by calculating the dot product between an input vector ( $u$ ) and each in-memory vector ( $m_i^{IN}$ ) followed by applying the softmax function. Next, the baseline computes the sum of weighted values multiplying each output-memory vector ( $m_i^{OUT}$ ) by a corresponding probability value ( $p_i$ ). Figure 3.3a describes dataflow of these computational steps. The baseline generates three temporary vectors (i.e.,  $T_{IN}$ ,  $P_{exp}$ ,  $P$ ) for each question, and the size of these vectors is proportionate to the number of story sentences ( $ns$ ) which continuously increases to support more complex question answering tasks. For example, when MemNN uses Wikipedia for training, the number of story sentences are around 200M [7]. In this case, the size of each intermediate vector is 800MB (use `float` data type) per each question, which easily exceeds the size of the typical shared cache (8MB – 40MB). Therefore, these temporary data are spilled to off-chip DRAM, incurring huge memory traffic and ex-

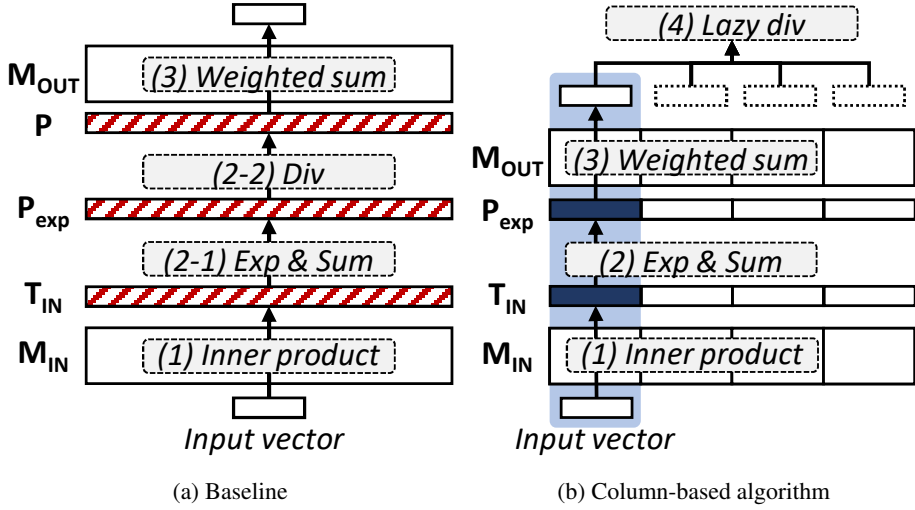


Figure 3.3: Dataflow comparison between the baseline and the column-based algorithm.

acerbating the overall performance.

To reduce the size of temporary data, we propose a *column-based algorithm* enabling MemNN to partially calculate output vectors. The key idea of the proposed algorithm is a *lazy softmax calculation*, which computes the *Softmax*'s division operation at last, not in the middle.

$$o = \frac{1}{\sum_j e^{u \times m_j^{IN}}} \sum_i e^{u \times m_i^{IN}} m_i^{OUT} \quad (3.2)$$

Equation (3.2) shows how the column-based algorithm calculates the output vector. Compared to the baseline, the column-based algorithm pulls the sum ( $\sum_j e^{u \times m_j^{IN}}$ ) out of the outer summation ( $\sum_i$ ). Since the sum does not depend on the index  $i$ , the column-based algorithm generates the same results to the baseline. By doing so, the column-based MemNN does not need to wait for the sum of entire values in the *Softmax* function and possible to calculate a part of the output vector.

Figure 3.3b describes the computational steps of the column-based algorithm and its dataflow. The column-based MemNN partitions input/output memory into multi-

ple *chunks*, and calculates partial output vectors for each chunk. The column-based MemNN computes dot products of the input vector with each in-memory vector and applies exponential function to the results, similar to the baseline. Here, in contrast to the baseline, the column-based MemNN directly calculates the weighted sum. The column-based MemNN iterates above operations over multiple chunks and accumulates each weighted sum into the output vector. After the end of the iteration, the column-based MemNN divides the output vector by the sum ( $\sum_j e^{u \times m_j^{IN}}$ ) calculated in the second step (*lazy softmax calculation*).

By doing so, the column-based MemNN can successfully reduce the size of temporary data to fit those into the cache. For example, when the chunk size is 1K, the total amount of intermediate vectors (i.e.,  $T_{IN}$ ,  $P_{exp}$ ) is 8KB per each chunk calculation; therefore, we can eliminate the entire off-chip DRAM accesses for intermediate vectors. In addition, the column-based MemNN facilitates input/output memory streaming to hide the memory access overheads. In contrast to the baseline, which cannot load input/output memory into caches due to their enormous size, the column-based MemNN can load those memory into the cache because it partially loads input/output memory per each chunk operation. We show the performance impacts of both temporary data reduction and input/output memory streaming in Section 3.4.2.

Also, the column-based MemNN can reduce the amount of computation (i.e., softmax’s division operation). In the baseline, the number of division operations is proportionate to the number of story sentences  $ns$  (step 2-2 in Figure 3.3a). However, the column-based MemNN requires the division operations in the size of the embedding dimension  $ed$  (step 4 in Figure 3.3b). Since the typical size of  $ed$  (32 – 256) is much smaller than  $ns$  ( $\approx 100M$ ), the column-based MemNN can significantly reduce the amount of computation.

Lastly, the column-based algorithm enables MemNN to achieve scale-out architecture. As the baseline computes each layer step-by-step, it cannot split each layer into multiple sub-layers due to enormous synchronization overheads. Therefore, to improve

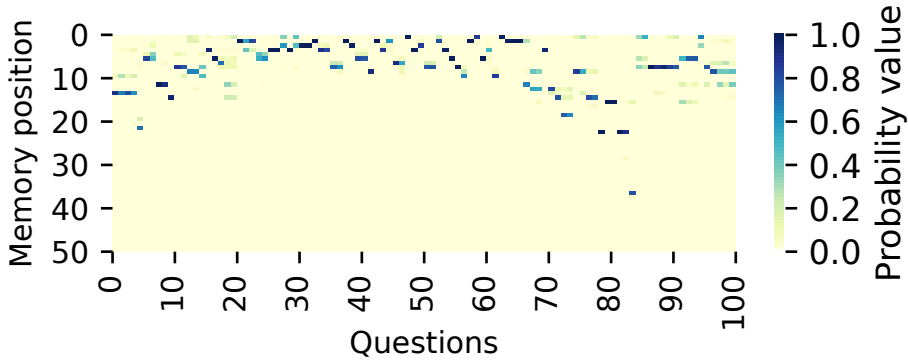


Figure 3.4: Probability value distribution. Each column represents the probability vector to each question. We use the Facebook bAbi dataset and testset [135].

the overall performance, the system should be scaled-up. Instead, the column-based MemNN can partition each layer into multiple sub-layers based on the chunk and merge the results at once. Here, synchronization overheads are negligible because the size of output results are proportionate to *ed*. Therefore, the column-based MemNN can distribute these sub-tasks into multiple compute units (e.g., CPU, GPU, FPGA) and fully utilize these resources. We evaluate this scale-out characteristic in Section 3.4.3.

### 3.2.2 Zero Skipping

Increasing demands for large-scale MemNN result in significant computation overheads because its computation algorithm shows super-linear complexity. As described in Section 3.1.2, MemNN’s compute-intensive phase (the inference operation) consists of three core computation steps: inner product, softmax, and weighted sum. For inner product and weighted sum, we need to calculate matrix multiplication known for super-linear time complexity. In this section, to reduce these computation overheads, we propose a *zero-skipping* technique and show its tradeoff between accuracy loss and the ratio of computation reduction.

The key observation for the zero-skipping technique is that the probability vector, calculated from the inner product between a question vector and in-memory matrix,

---

**Algorithm 1:** MnnFast’s zero-skipping algorithm.

---

**input** : The skip threshold  $th_{skip}$

**input** : The probability vector  $P$ .

**input** : The output memory  $M_{IN}$

**output:** The weighted sum  $O$ .

```
/* Calculate the weighted sum of the output memory with the
   probability values. */
1  $O = [0]$  /* Initialize the output vector. */
2 foreach  $i < ns$  do
   /*  $ns$  is the number of story sentences. */
3   if  $p_i > th_{skip}$  then
4      $O = O + p_i m_i^{OUT}$ 
5   end
6 return  $O$ 
```

---

shows a huge imbalance. Specifically, only a few values are non-zero and others are close to zero.<sup>1</sup> Figure 3.4 shows the probability value distribution. We use Facebook bAbi tasks and its dataset [135] and measure probability values to each question. In this evaluation, MemNN gets up to 50 story sentences followed by a question, and we show probability vectors for randomly chosen 100 questions. The results show that *only a few probability values are activated and most values are close to zero*. It is because the probability vector means the correlation of question sentence with story sentences, and only a few story sentences are related to the given question.

By using this characteristic, we propose the *zero-skipping* optimization to bypass a large amount of output computation. Algorithm 1 shows how MnnFast reduces the quantity of output memory computation by using the zero-skipping optimization. In contrast to the baseline which calculates all multiplications between probability values

---

<sup>1</sup>Note that the sum of all values in the probability vector is one because these values are normalized by the softmax function.

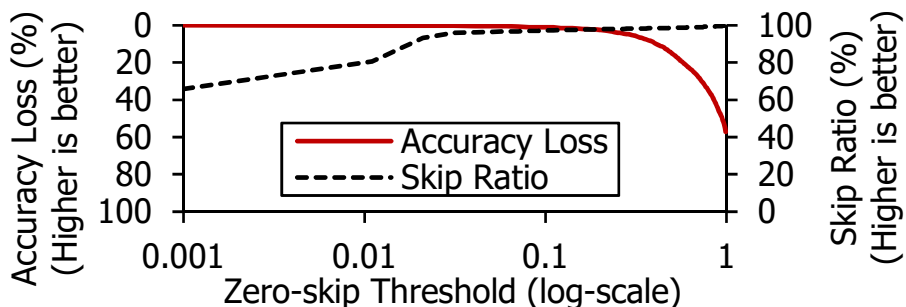


Figure 3.5: Tradeoffs between accuracy loss and computation reduction according to the skip threshold.

and output memory vectors, the zero-skipping algorithm only computes the multiplication when the probability value is larger than a threshold value ( $th_{skip}$ ) (line 3 and 4). By doing so, MnnFast can skip some portions of output memory computation.

The zero-skipping optimization can significantly reduce the amount of output memory computation; however, it also sacrifices the prediction accuracy. If the skip threshold is too high, we can skip an enormous amount of computation; however, its prediction accuracy will dramatically decrease. On the other hand, if the skip threshold is too low, we cannot get enough opportunity to reduce the quantity of computation.

To quantify tradeoffs between accuracy loss and computation reduction, we measure both the accuracy loss and the ratio of computation reduction according to the different skip thresholds. We use Facebook’s bAbi dataset and 20 QA tasks and get an average among the QA tasks. Both accuracy loss and computation reduction are measured by comparing with the baseline’s accuracy and the amount of computation respectively.

Figure 3.5 shows the results of the evaluation. The results show that the zero-skipping optimization can achieve 97% reduction of output computation while sacrificing 0.87% of accuracy when the skip threshold is 0.1. When the skip threshold is 0.01, our optimization shows 81% reduction without any accuracy loss. Note that probability values in MemNN represent the correlation between a question and story

sentences, and generally few story sentences are related to the given question; therefore, the zero-skipping optimization is highly promising for MemNN to reduce the amount of computation.

### 3.2.3 Embedding Cache

As described in Section 3.1.3, the large-scale MemNN suffers from huge cache contention between the embedding and inference operations. While the inference operation actively uses the shared cache to maximize CPU utilization, the embedding operation generates a large quantity of memory requests and pollutes the shared cache. This cache contention dramatically increases the number of cache misses for the inference operation, which results in huge performance degradation. In this section, we provide some techniques (i.e., cache bypassing, embedding cache) to solve the cache contention problem in various hardware architectures.

In the CPU environment, we can simply apply a cache bypassing technique for embedding’s memory requests. By using non-temporal memory instructions [9] for the embedding operation, we can achieve the memory isolation between the inference and embedding operations. However, the bypassing techniques has two major drawbacks. First, memory access overheads of the embedding operation are limited to DRAM access latencies, which increases the execution latency of the embedding operation significantly. Second, the technique raises the amount of memory pressure as the number of DRAM accesses increases.

To overcome these limitations, we propose an *embedding cache*. The embedding cache is a dedicated cache for storing internal state vectors during the embedding operation. As explained in Section 2.1, during the embedding operation, MemNN looks up a vector from the embedding matrix per each word in a sentence. Here, the embedding cache stores pairs of word ID (represented by the BoW model) and a corresponding internal state vector. Since each access loads the vectors whose size is the embedding dimension, we set the word size of our embedding cache as the embedding dimension.

With the embedding cache, we can perfectly eliminate the cache contention.

In addition, the embedding cache reduces the quantity of off-chip DRAM memory requests because of a high word locality. Linguistics researchers show that the high word locality exists in from daily conversations to literature [30]. Therefore, most lookup operations load corresponding internal vectors from the embedding cache, which leads to reducing the number of DRAM accesses.

### 3.3 Implementation

To show the effectiveness of our idea MnnFast across various platforms, we implement the baseline MemNN and MnnFast on CPU, GPU, and FPGA. While specialized architectures are getting lots of attention for machine learning, general-purpose CPU and GPU are still popular. Thus, we first validate our idea on CPU and GPU first, then we elaborate on the possible specialized hardware design, which is based on the analysis of the general-purpose architecture.

#### 3.3.1 General-Purpose Architecture – CPU

We first implement the baseline MemNN [125] and MnnFast on CPU. In Section 3.4.2, we elaborate on the potential effect of column-based MemNN based on this implementation.

**Baseline Implementation (MemNN).** We implement MemNN in C++ with open-source BLAS library, OpenBLAS [132, 140]. We implement each operation (described in Section 2.1) as a single function. Our implementation takes three input data: input memory, output memory, and question sentences. The input and output memory are usually provided by the system while users give the question sentences to the system. For interactive applications, users could provide necessary sentences to build the input and output memory containing user-specific contextual database (e.g., the book contents a user has read).



We assume that all the input and output memory have already been converted into the internal data format as the data would be prepared from an external database in advance. On the other hand, as questions are generated on-the-fly by users, we assume each question is in a raw format, Bag-of-Words.

First, we convert a given question into an internal representation. The operation consists of *lookup* operations. For each word in the question, we find the corresponding embedding vector from an embedding matrix (an embedding dictionary) and sum up the vectors into a single vector to represent the question. We implement the embedding matrix as an array to retrieve embedding vectors in  $O(1)$ .

Then, we inference the corresponding answer from the given knowledge database, input and output memory. The remaining computational steps (input and output memory representation and output calculation) consist of a series of *Inner product*, *Softmax*, *Weighted sum*, *Vector sum*, and *FC*. All operations excluding *Softmax* are represented in vector operations (Section 2.1). For example, *Inner product* and *Weighted sum* are implemented as matrix-vector (vector-matrix) multiplications. To implement them, we rely on OpenBLAS for efficient computation.

We parallelize each operation in a lock-step manner. To parallelize the BLAS-based operations, we exploit the multi-threading feature of BLAS library. To parallelize *Softmax*, which does not rely on BLAS operations, we divide it into three steps: (1) applying the natural exponential function on the elements of a vector, (2) calculating the sum of the exponential results, and (3) normalizing the exponential results with the summation. We exploit data-level parallelization of each step with PThread.

**Column-based algorithm & zero-skipping.** The column-based algorithm simultaneously applies the inference operation on sub-chunks of the given knowledge database. We divide the knowledge database into sub-chunks, each of which contains 1000 sentences, and then make multiple worker threads to process them independently. Unlike MemNN, each thread performs a series of inference operations on only a given sub-chunk, not on the entire data.

To perform inference on sub-chunks in parallel, we use *Partial softmax* we propose in Section 3.2.1. The other operations are implemented in a similar way to the baseline MemNN. Compared to the baseline MemNN, column-based algorithm parallelizes operations not only within a single operation but also across multiple operations.

As only a few sentences of the output memory are related to a given question, we apply zero-skipping (Section 3.2.2) to *Weight sum* operation. Our implementation skips adding up output sentences whose weight (or probability) is lower than 0.1.

### 3.3.2 General-Purpose Architecture – GPU

As an intermediate step before we move on to the FPGA implementation, we evaluate the proposed algorithm on GPUs. As discussed earlier, *column-based algorithm* changes the following four consecutive steps, namely `Inner product`, `Softmax`, `Weighted sum`, and `Sum`. Therefore, we make these four steps into GPU kernels and perform memory copy from/to GPUs between kernel invocations if needed.

**GPU kernel implementation.** We use cuBLAS [1] provided with CUDA Toolkit 10.0 to perform matrix-matrix multiplications. By using the state of the art GPU BLAS library, we try to avoid any inefficiency incurred by unoptimized kernel implementation. All steps except `Softmax` is implemented by calling a cuBLAS function, while `Softmax` is implemented as one custom kernel followed by a cuBLAS function. The custom kernel of `Softmax` calculates exponential value of each input value, which is too simple to be further optimized, as threads are regularly mapped to one input/output memory address. The other three steps are simply translated into cuBLAS calls; specifically, `Inner product` is multiplication of two matrices  $M_{IN}$  and  $U$ , `Weighted sum` is multiplication of  $P^T$  (transpose of  $T$ ) and  $M_{OUT}$ , and `Sum` is multiplication of a vector filled with ones and the result of the previous step.

**Column-based algorithm.** To apply column-wide data partition, we use multiple CUDA streams with one GPU, or multiple GPUs each with one stream. Each stream/GPU processes sub-matrices consisting of smaller number of sentences. Thanks

to the *column-based algorithm*, the first three steps are executed in parallel. Only the last step, which serves as reducing the partial results from different streams/GPUs, is executed by one stream/GPU. Fortunately, this step takes a negligible portion of the entire latency, as it sums up small matrices ( $ed \times nq$ ) whose count equals the number of streams/GPUs.

**Zero skipping.** A pruning scheme like zero skipping is ineffective or even harmful for GPUs [91, 56]. The reason is that a warp cannot complete early unless *all* 32 threads in the warp are zero skipped, which is very unlikely. To eliminate the poor utilization problem, we can compact the pruned matrix into a sparse matrix, but the transformation itself is costly again. We implement the transformation by following an example of the official cuSPARSE document, but the latency of transforming  $M_{OUT}$  is comparable to `Weighted sum` stage. This means that even with an unrealistic assumption such as 100% skip ratio cannot justify the use of the sparse matrix multiplication at `Weighted sum` step. Even worse, sparse matrices are basically much slowly processed due to indirect memory accesses.

Currently, we consider DeftNN [56], a recently proposed GPU compaction scheme for CNNs, as the most promising option, but it is effective only for an extreme case (small  $nq$ ). By adopting DeftNN, we have to eliminate the same number of rows from  $P$  and  $M_{OUT}$  from our `Weighted sum` stage. Meanwhile, we need sufficiently high  $nq$  to prevent the host-to-GPU *memcpys* from taking an excessive portion of the execution time (Section 3.4.3). As a result, we can hardly find all-zero rows, as a row of  $P$  has  $nq$  probabilities relying on *independent* questions.

### 3.3.3 Custom Hardware (FPGA)

We design and implement an FPGA-based accelerator for MnnFast by using Vivado High-Level Synthesis (HLS). We create MnnFast’s IP core from Vivado HLS and use the IP core in Xilinx Vivado Design Suite to generate a bitstream file for ZedBoard Zynq-7020 FPGA. In this section, we omit the baseline implementation because its

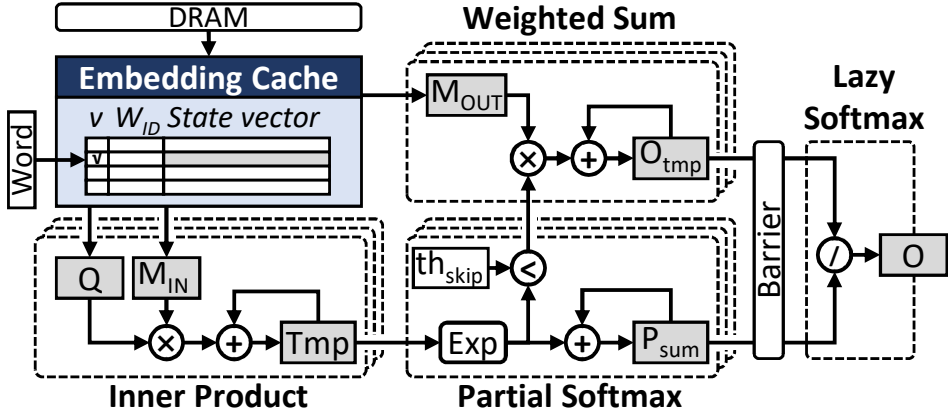


Figure 3.6: A high-level architecture of FPGA-based MnnFast.

design is straightforward.

**Column-based algorithm.** Figure 3.6 shows a high-level architecture of our FPGA-based accelerator design. First, each word in a sentence passes through the embedding cache to calculate a corresponding internal state vector. During embedding, MnnFast converts a question and new incoming story sentences into internal state vectors, and the state vectors of story sentences are appended to the input and output memory:  $M_{IN}$  and  $M_{OUT}$ , respectively.

Next, MnnFast partitions a number of story sentences  $ns$  into multiple chunks to reduce the size of intermediate data between each computational layer. For each chunk, MnnFast calculates the inner product between question vectors  $Q$  and each story vector  $m_i^{IN}$ . The resulting vector  $Tmp$ , whose size is same as  $ns$ , is delivered to the partial softmax.

In contrast to the baseline which is blocked until calculating the inner product over all chunks, MnnFast partially computes the softmax function followed by the weighted sum. During the partial softmax, MnnFast applies an exponential function to each value in the vector  $Tmp$  and accumulates the exponential results into  $P_{sum}$ . The exponential results are also delivered to the weighted sum to calculate a partial output vector  $O_{tmp}$ .

After the end of the iteration over all chunks, MnnFast applies a remaining part of the softmax function, lazy softmax. MnnFast divides each value in  $O_{tmp}$  by  $P_{sum}$  and returns the final output vector  $O$ .

**Zero skipping.** To implement the zero-skipping optimization, MnnFast compares the exponential results with the skip threshold  $th_{skip}$ . If the skip threshold is larger than an exponential result, MnnFast does not calculate the weighted sum and only adds the result into  $P_{sum}$ .

As multiple partial softmax units are parallelly executed at runtime, all exponential results may not be lower than the skip threshold. In this case, we calculate the weighted sum with those values although only a few values are higher than the skip threshold. We believe that this case does not frequently occur because most probability values are zero in general. As a result, MnnFast can reduce the amount of output computation significantly.

**Embedding cache.** We design the embedding cache as a direct mapped cache. Each entry in the embedding cache consists of three fields: a valid bit (1 bit), a word ID ( $\log_2(\# \text{ words in dictionary})$  bits), and a state vector ( $32 * ed$  bits). By using the embedding cache, MnnFast can achieve memory isolation between the inference and the embedding operation and reduce the number of DRAM accesses, which improves the performance of the embedding operation. We show the performance impact of the embedding cache according to different cache sizes in Section 3.4.4.

## 3.4 Evaluation

### 3.4.1 Experimental Setup

We implement the baseline, the baseline with each optimization (i.e., column-based algorithm, column-based with streaming, zero-skipping, embedding cache), and MnnFast in various hardware platforms: CPU, GPU, and FPGA.

**CPU configuration.** We compare the baseline with MnnFast on a 24-core dual-

Table 3.1: Memory networks configuration for the evaluation.

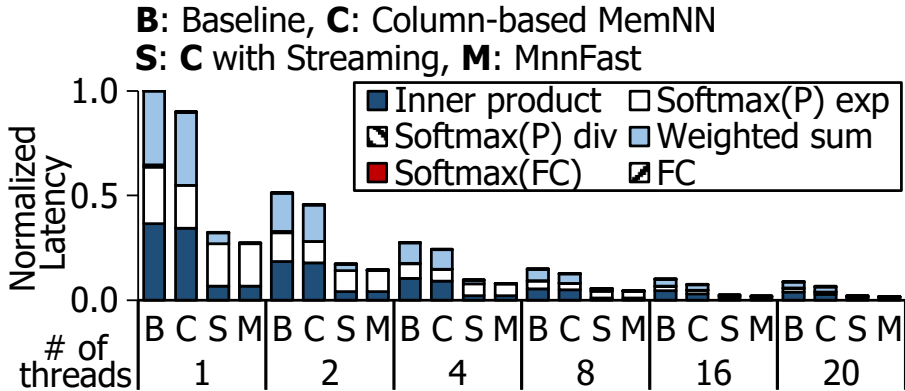
Entry	CPU	GPU	FPGA
Embedding dimension (# entry)	48	64	25
Database size (# sentences)	100M	100M	1000
Chunk-size (# sentences)	1000	Variable	25

socket Xeon CPU system with DDR4-2400MHz 256GB memory. We run our implementation on Ubuntu 16.04 LTS and use OpenBLAS [132, 140] for BLAS operations. For evaluation, we use the following network configuration, described in Table 3.1.

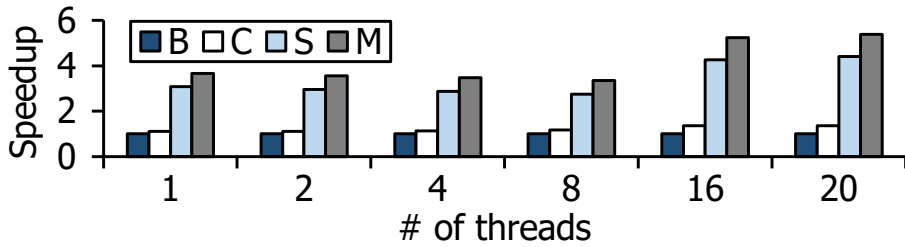
**GPU configuration.** We use a SUPERMICRO SuperServer 4028GR-TRT with two Intel Xeon CPU E5-2650 v4 and six Nvidia TITAN Xp GPUs. We measure the performance of GPU-based MnnFast with Linux kernel version (4.4.0-89-generic) and CUDA Toolkit version 10.0.

**FPGA configuration.** We implement FPGA-based MnnFast on ZedBoard featuring Xilinx Zynq-7020 Soc and DDR3 memory by using Vivado HLS. Our implementation on the programmable logic (PL) runs at 100MHz with DDR3 memory operating at 533MHz. The memory has 32-bit effective width. To control MnnFast implemented on the PL, we build a monitoring program executed on the ARM Cortex-A9 processor of Zynq SoC.

**Memory network configuration.** Table 3.1 shows the configuration parameters of MemNN for the evaluation. We use a similar configuration for CPU and GPU but scale it down for FPGA due to lack of available logic cells. The embedding dimension of GPU is different to that of CPU to fully utilize streaming multiprocessors (SMs) in GPUs.



(a) Execution latency breakdown



(b) Performance speedup

Figure 3.7: Performance of column-based algorithm on CPU.

### 3.4.2 CPU

This section shows the performance and its scalability of MnnFast compared to the baseline MemNN. To validate the key idea of our proposal, we compare the results from 1) the baseline MemNN, 2) Column-based MemNN, 3) Column-based MemNN with data streaming, and 4) MnnFast.

**Performance.** Figure 3.7 shows the performance of MnnFast and its comparison targets. Column-based algorithm brings two benefits, 1) improved cache uses from data chunking and 2) data streaming enabled by small-size data chunk. To analyze each benefit, we compare the performance of column-based algorithm without data streaming and with streaming against the baseline. Column-based algorithm achieves 1.21x speedup compared to the baseline. The speedup comes from the efficient data

chunking and intermediate data reuse of our algorithm, which reduces the execution latency of Softmax(P) exp (Figure 3.7a).

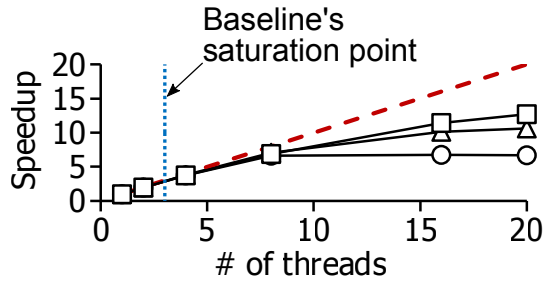
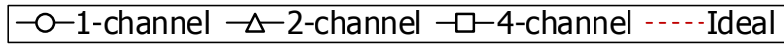
Data streaming significantly improves the performance against the baseline (3.33x on average). As data streaming brings all necessary data into computational units timely, we can avoid lots of cache misses and in turn improve the performance of *Inner product* and *Weighted sum* operation.

MnnFast achieves 4.02x average speedup compared the baseline thanks to our column-based algorithm and zero-skipping techniques. As MnnFast improves cache utilization, its effects increase with the number of working threads as shown in Figure 3.7b.

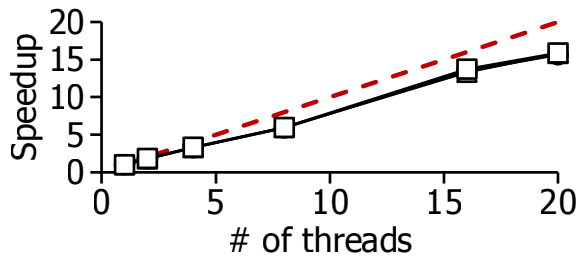
**Cache Efficiency.** Figure 3.8 shows the scalability of MnnFast. Ideally, the performance should be linearly proportionate to the number of threads. We measure the performance improvement of parallelization at different memory bandwidth configurations. As shown in Figure 3.8a, the performance of MemNN using column-based algorithm is saturated at 10-thread on 4-memory channel system and is more scalable than the baseline, whose saturation point is around 4-thread. Column-based algorithm improves the scalability, but the results from different memory bandwidth diverge and are still far from the ideal result.

Figure 3.8b and 3.8c show that MnnFast achieves highly scalable performance with data streaming-enabled column-based algorithm, reaching the ideal speedup. Such a scalability comes from the fact that column-based algorithm with data streaming reduces the number of accesses to the shared memory system including shared cache. Figure 3.9 shows the number of off-core memory accesses. The counts are normalized to the baseline result. Column-based algorithm makes off-core DRAM accesses of the baseline hit onto LLC, which in turn achieves higher scalability than the baseline (Figure 3.8a). As column-based algorithm with data streaming cuts off more than 60% of off-core accesses, MnnFast can achieve highly-scalable performance.

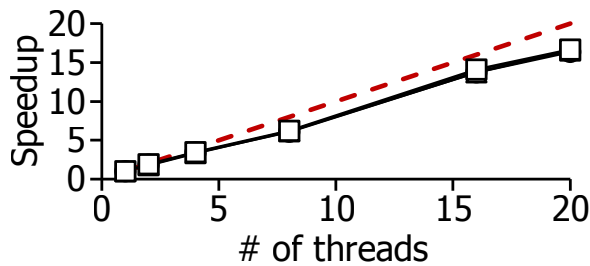




(a) Column



(b) Column-S



(c) MnnFast

Figure 3.8: Scalability of column-based algorithm on CPU.

### 3.4.3 GPU

Figure 3.10 shows the experiment results for CUDA stream and multiple GPUs. Note that `sum` and `device-to-host memcpy` take a small portion of the entire latency, so we did not show it here.

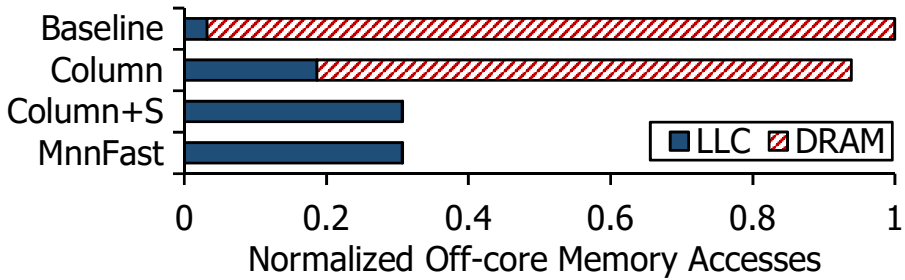
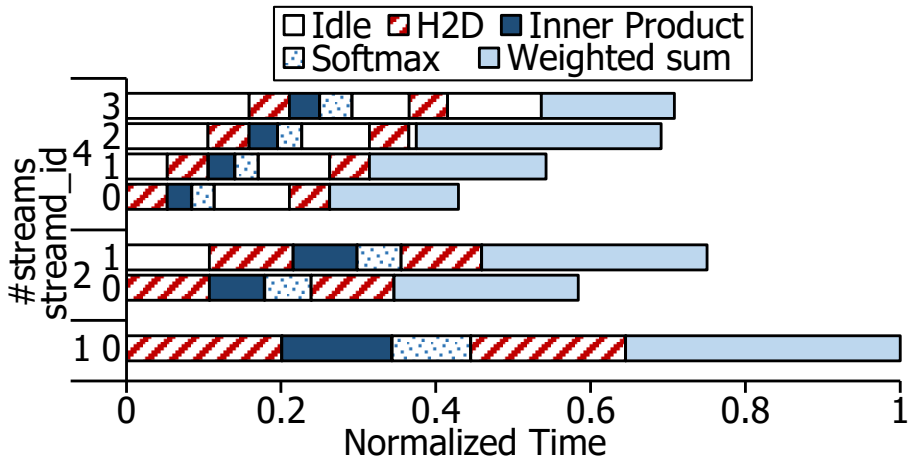


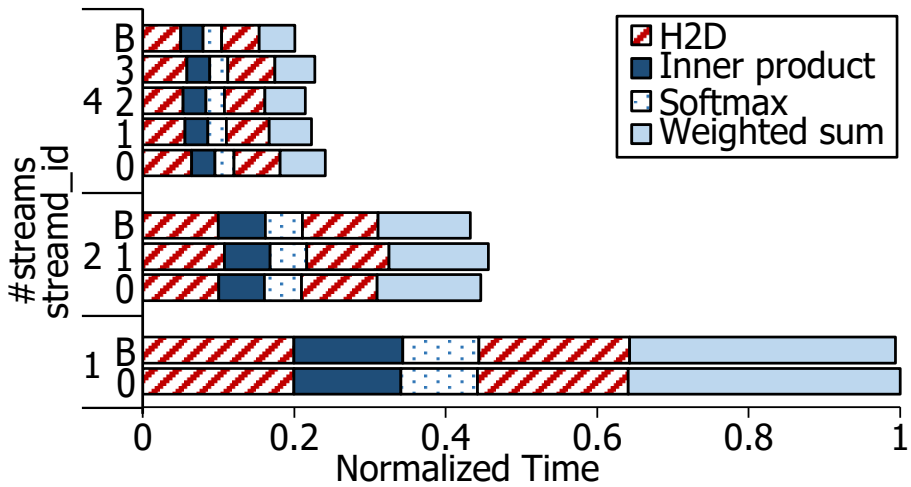
Figure 3.9: The number of off-chip memory accesses on CPU.

**GPU streams.** Figure 3.10a shows the latency of CUDA stream chunking. As we use only one GPU for the multiple streams case, there is a restriction that *memcpy* cannot overlap; other overlaps are achievable if the GPU has enough resources. The result resembles typical pipelined parallelization. Two streams almost perfectly achieve overlap of *memcpy* of one stream and kernels of the other stream. This simple change gives 1.33X speedup, confirming the importance of data transfer overheads between host memory and accelerator memory [35]. Increasing the number of streams does not reduce the latency much, as *memcpy*s form a critical path.

**Multiple GPUs.** Figure 3.10b shows a similar experiment with multiple GPUs. To show the best latency achievable without PCIe bandwidth limit, we additionally experiment with only one GPU running on its chunk, and present the results labeled as ‘B’. We achieve much better scalability than before, as using multiple GPUs eliminates the previous overlap constraint. However, the scalability is still limited, because of two reasons. First, kernel efficiency decreases as we reduce the size of input data. This is the fundamental limit of the GPU scale-out approach, and explain why the best ‘B’ latency does not scale perfectly. Second, PCIe bandwidth is limited for one node. This problem is solvable by using multiple nodes, each with a small number of GPUs. Note that the extra communication overheads for the last *Sum* in this case would be negligible as well due to small input data for *Sum*.



(a) Multiple streams



(b) Multiple GPUs (B: best run-alone)

Figure 3.10: Scalability of column-based algorithm on GPU.

### 3.4.4 FPGA

**Performance** In this experiment, we evaluate the effectiveness of each optimization on FPGA. We implement four versions: baseline, column (applying column-based algorithm only), column+S (applying both column-based and streaming), and MnnFast. Figure 3.11 shows the latency results of four versions of the FPGA implementation.

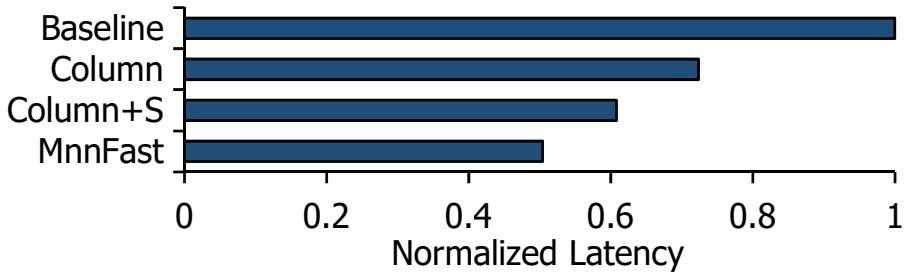


Figure 3.11: Latency reduction of FPGA-based MnnFast. Each latency is normalized to the baseline.

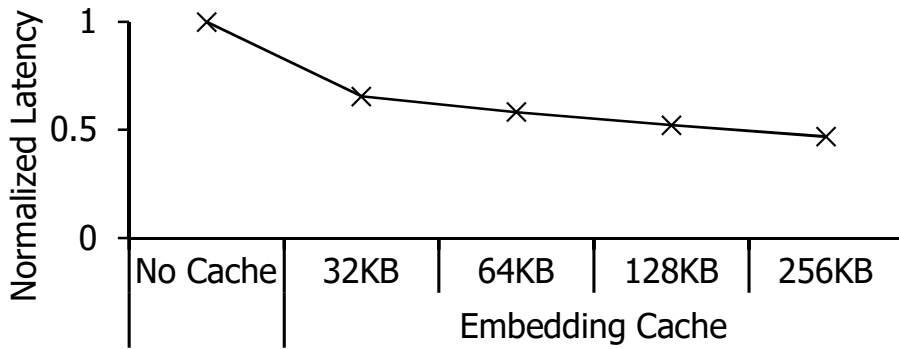


Figure 3.12: Effectiveness of embedding cache in FPGA-based MnnFast. Each latency result is normalized to the No Cache.

Each latency result is normalized to the baseline. The results show each optimization gradually reduces the execution latency. Compared with the baseline, the column-based algorithm and the streaming optimization reduce the latency by 27.6% and 39.2% respectively. With all the optimizations (i.e., column-based algorithm, streaming optimization, zero-skipping), MnnFast shows the performance improvement by up to  $2.01\times$ .

**Effectiveness of Embedding Cache.** To evaluate the effectiveness of the embedding cache, we use the word frequency of the Corpus of Contemporary American English dataset (COCA) [30], and the size of the embedding dimension is 256. We

measure the latency reduction according to different sizes of the embedding cache: 32KB, 64KB, 128KB, and 256KB. Figure 3.12 shows the latency reduction of the embedding cache among various sizes. Each latency is normalized to the “No Cache” version. The results show that the embedding cache effectively reduces latency overheads during the embedding operation. For each cache size (32KB, 64KB, 128KB, 256KB), the embedding cache reduces the latency by 34.5%, 41.7%, 47.7%, and 53.1%, respectively. Note that the required cache size is moderate thanks to the word locality.

### 3.4.5 Comparison Between CPU and FPGA

We compare energy efficiency between CPU-based MnnFast and FPGA-based MnnFast. For a fair comparison, we resize the network configuration for both platforms to process the same quantity of question answering tasks. We use `turbostat` to measure energy consumption for CPU-based MnnFast. For FPGA-based MnnFast, we use power stats provided by Xilinx Vivado Design Suite after generating the bitstream. The results show that FPGA-based MnnFast improves energy efficiency by up to  $6.54\times$ .

## 3.5 Conclusion

We propose MnnFast, a novel system architecture for large-scale memory networks to achieve fast and scalable reasoning performance. We identify three performance problems of the current architecture and propose three key optimizations (i.e., *column-based algorithm with streaming*, *zero-skipping*, and *embedding cache*). We show MnnFast outperforms the baseline on various hardware: CPU, GPU, and FPGA.

## Chapter 4

# A Fast, Scalable, and Flexible System for Large-Scale Heterogeneous NLP Models

### 4.1 Motivation & Design Goals

#### 4.1.1 High Model Complexity

As described in Section 2.2, emerging NLP models consist of various combinations of basic operations with different parameter configurations; therefore, an ideal NLP accelerator should cover a wide variety of the basic operations in NLP models. However, existing work *only* optimizes specific operations, which results in the narrow model coverage. To the best of our knowledge, there is no work aiming to holistically optimize whole operations in emerging NLP models. For example, some studies [62, 52] propose optimization techniques to accelerate the *attention mechanism* only, but the attention mechanism is not a major performance bottleneck in most models and configurations (e.g., 11.8% in Transformer, 8.5% in BERT). In this case, their end-to-end performance improvements are only 13.4% and 9.3%, respectively.

To overcome this high model complexity in NLP models, we need to identify performance-critical operations in different NLP models. We conduct static analysis and extensive profiling to identify all performance-critical operations required by most

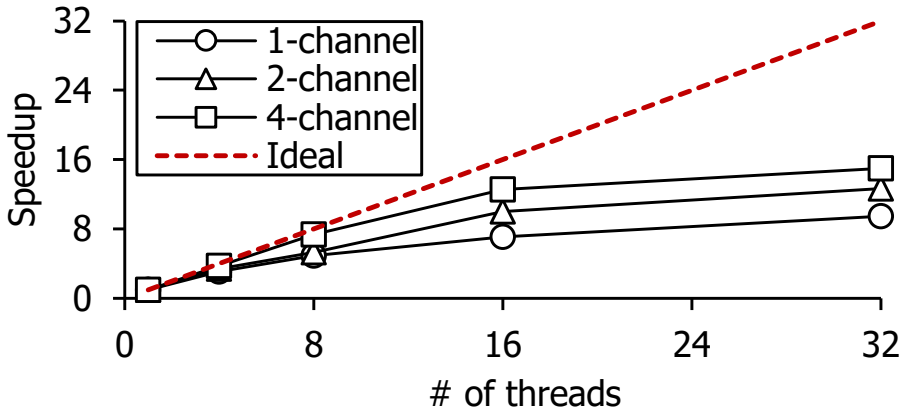


Figure 4.1: Limited scalability due to memory bandwidth. The speedup results of each channel configuration are normalized to the corresponding single-thread result.

NLP models. We explain analysis methods and results in Section 4.2.1 for more details.

#### 4.1.2 High Memory Bandwidth

All NLP models require a considerable amount of memory bandwidth, and this memory overhead continuously increases because of growing demands for high reasoning power. Each parameter (i.e.,  $s$ ,  $d_H$ ,  $d_{FF}$ ) can be scaled-up to enhance the accuracy of NLP models. For example, as sequence length ( $s$ ) increases, the models can learn more long-distance relationships among words and sentences stored in external memory. In addition, the larger dimensions ( $d_H$  and  $d_{FF}$ ) of internal states help to solve complicated questions. Therefore, the memory pressure problem becomes intensified, which makes the current system difficult to achieve scalable performance.

Figure 4.1 shows how memory bandwidth affects the scalability of NLP models. We measure the performance of BERT with the default configuration (see Section 4.5.1). To show the performance impact of memory bandwidth, we measure the speedup with multiple threads while reducing memory bandwidth (# of memory channels). In this evaluation, we use enough CPU cores (i.e., two Intel Xeon Gold 5220 18C/36T CPUs with turning-off hyperthreading) to prevent the computation from be-

coming a performance bottleneck. We can observe that BERT quickly reaches a performance saturation point as the bandwidth decreases; in other words, the high memory bandwidth requirements of BERT prevent the current system from achieving scalable performance.

To overcome the memory bandwidth problem, we need more efficient memory management mechanisms applicable to all kinds of operations in emerging NLP models. We propose a new computation algorithm (called *holistic model partitioning*) that (1) minimizes the size of data spills, (2) provides more efficient data chunking, and (3) enables the current system to hide most memory accessing overhead by significantly reducing the working set size. We explain this algorithm in Section 4.2.2 for more details.

### 4.1.3 Heavy Computation

Similar to the high memory bandwidth problem, the computation overhead in NLP models also increases as the size of recent NLP models continuously increases. By analyzing the characteristics of each operation, we find out that attention-related operations (e.g., *Q/K/V-Gen*, *dot product*, *softmax*, *weighted sum*) are the most compute-intensive parts in state-of-the-art NLP models. *Q/K/V-Gen*, *dot product*, and *weighted sum* require multiple dense matrix multiplications known for compute-intensive operations. Also, *softmax* uses exponential functions that are highly compute-intensive.

To overcome the heavy computation problem, we need an optimization technique to reduce the amount of computation. From an in-depth analysis, we find out the high potential for reducing the computation in *V-Gen*, *softmax*, and *weighted sum*. The key insight is that most probability values (representing the correlation between a query token and a key token) are close to zero because only a few key tokens are related to the given query token. Therefore, we propose *cross-operation zero skipping* to bypass a large amount of computation in attention-related operations. We explain this optimization in Section 4.2.3 for more details.



These near-zero approximations are beneficial to computation reduction; however, these optimizations incur a new problem: *execution time skewness*. As each partitioned block has a different skipping opportunity and these blocks should be synchronized at the last part of attention components, the overall execution time is delayed by the worst-case latency, which results in resource underutilization. To overcome this skewness problem, we propose a *dynamic scheduler* to maximize resource utilization in NLP accelerators (Section 4.2.3).

#### 4.1.4 Huge Performance Variation

Through extensive profiling, we find out various models and parameter configurations incur a huge *performance variation* in NLP models as shown in Figure 4.2. In other words, one operation can be a major performance bottleneck on a specific model, but its overhead can be negligible on other models. Therefore, this huge performance variation makes the current system architecture difficult to find out an optimal design point. For example, if one accelerator (optimized to a specific parameter configuration) executes the NLP model with a different configuration, it can only achieve 29.7% of the performance we can get from an accelerator optimized for that configuration. (Section 4.5.4).

To overcome the huge performance variation, we need an efficient resource rebalancing technique to achieve the best performance for a given parameter configuration. We propose an *adaptive hardware reconfiguration* enabling an HW accelerator to achieve full potential performance by finding an optimal design point. We explain the details in Section 4.2.4.

#### 4.1.5 Design Goals

We set our key design goals to achieve scalable performance on the wide variety of NLP models.

- **High coverage of various NLP models.** It should support diverse NLP models.

We extract key basic operations used in the NLP models and propose a holistic solution optimizing all these operations.

- **Efficient memory management algorithm.** It should minimize the memory bandwidth requirements. We propose *holistic model partitioning* to eliminate intermediate data spills by minimizing the working set size.
- **Reduction of computation.** It should reduce the computation amount. We propose the *cross-operation zero skipping* to reduce the computations of various operations by skipping operations of near-zero values.
- **Model/config-adaptive system architecture.** It should provide an optimal design point for a given configuration. We propose the *adaptive hardware reconfiguration* to fully leverage hardware accelerators.

## 4.2 NLP-Fast

### 4.2.1 Bottleneck Analysis of NLP Models

Aforementioned in Section 4.1.1, there are various kinds of NLP models, and each model is composed of diverse operations. Therefore, we conduct (1) static analysis to find the basic operations in NLP models and (2) extensive profiling to find the performance-critical operations.

Table 4.1 shows the result of our static analysis. We analyze the types of operations used in the key computational components (in Table 2.1) and calculate their time and space complexity according to various parameters. We observe that NLP models consist of some key operations (i.e., *Q/K/V-Gen*, *dot product*, *softmax*, *weighted sum*, *attention FC*, *linear trans-1*, *activation func*, *linear trans-2*).

Figure 4.2 shows the profiling results of representative NLP models (i.e., Memory networks (MemNet), Transformer (TF), BERT) with different parameter configurations. We scale a specific parameter (S:  $s$ , H:  $d_H$ , F:  $d_{FF}$ ) from base configurations to profile models with various configurations. We break the total execution time down in

Table 4.1: Time and space complexity of each key operation commonly used in emerging NLP models.

	Operations	Time complexity	Space complexity
Multi-head self-Att. <sup>†</sup>	Q-Gen	$O(s \cdot d_H^2)$	$O((s + d_H) \cdot d_H)$
	K/V-Gen	$O(s \cdot d_H^2)$	$O((s + d_H) \cdot d_H)$
	Dot product	$O(s^2 \cdot d_H)$	$O(s \cdot d_H)$
	Softmax	$O(s^2 \cdot h)$	$O(s^2 \cdot h)$
	Weighted sum	$O(s^2 \cdot d_H)$	$O(s \cdot (s + d_H))$
	Attention FC	$O(s \cdot d_H^2)$	$O((s + d_H) \cdot d_H)$
FFN	Linear trans-1	$O(s \cdot d_H \cdot d_{FF})$	$O((s + d_{FF}) \cdot d_H)$
	Activation func	$O(s \cdot d_{FF})$	$O(s \cdot d_{FF})$
	Linear trans-2	$O(s \cdot d_H \cdot d_{FF})$	$O((s + d_H) \cdot d_{FF})$

<sup>†</sup> *Multi-head self-attention* contains key/value generation overhead; otherwise, *multi-head attention* does not include the key/value generation overhead.

different types of operations described in Table 4.1. In Figure 4.2, we notice that the recent NLP models suffer from a huge performance variation due to the following reasons. First, a wide variety of models causes the performance variation as each model consists of different types of operations (*model diversity*). Furthermore, different parameter configurations incur different performance breakdown ratio (*config diversity*).

**Model diversity.** The performance breakdown shows different aspects according to each model. For example, the attention mechanism (i.e., *dot product*, *softmax*, *weighted sum*) dominates the total execution time in MemNet (*base*) (85.3%). However, these operations only take 11.8% and 8.5% in TF (*base*) and BERT (*base*), respectively.

TF (*base*) and BERT (*base*) show different performance aspects, although they

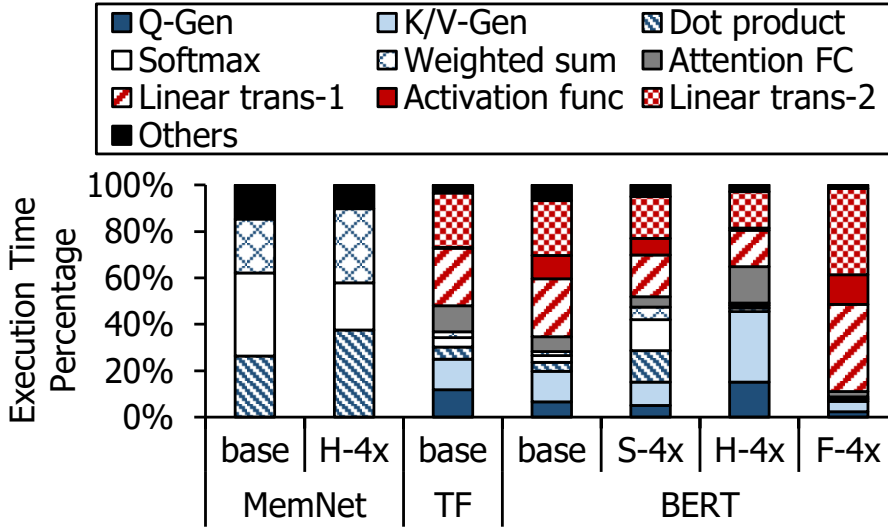


Figure 4.2: Performance breakdown of NLP models with various parameter configurations. *base* configurations are in the evaluation section (i.e., *experimental setup*), and S/H/F-4x are configurations with a fourfold increase of  $s$ ,  $d_H$ , and  $d_{FF}$ , respectively.

have the same parameter configuration. Instead of using ReLU as the activation function, BERT uses GELU, a more complex operation; therefore, the activation function takes 9.6% more in BERT. Also, in TF, a portion of *Q-Gen* overhead in *Q/K/V-Gen* is larger than BERT’s *Q-Gen* overhead because the multi-head attention of TF decoder does not contain *K/V-Gen*.

**Config diversity.** Not only model diversity, but various parameter configurations also cause a huge performance variation. Figure 4.2 shows the performance with a fourfold increase of each parameter (i.e.,  $s$ ,  $d_H$ ,  $d_{FF}$ ). As described in Table 4.1, if  $s$  increases, the overhead of the attention mechanism (i.e., *dot product*, *softmax*, and *weighted sum*) would be significantly higher as their time complexity is quadratic with respect to the  $s$ . Similarly, if  $d_H$  increases, the *Q/K/V-Gen* and *attention FC* takes a more portion of the total execution time. Lastly, the overhead of the feed-forward network (composed of the *linear transformation-1*, *activation function*, and *linear transformation-2*) linearly increases with  $d_{FF}$ .

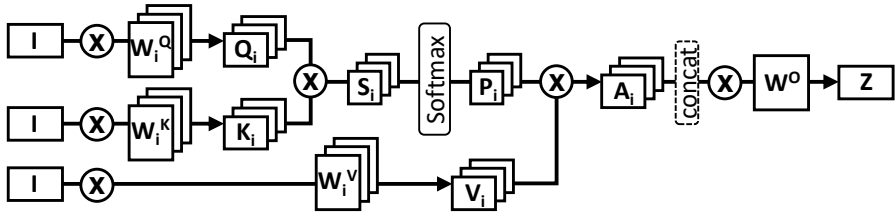
As shown in this section, NLP models are composed of some basic operations. Also, they have huge performance variation because of various models and parameter configurations. Therefore, *we propose a holistic solution that optimizes every operation in NLP models.*

## 4.2.2 Holistic Model Partitioning

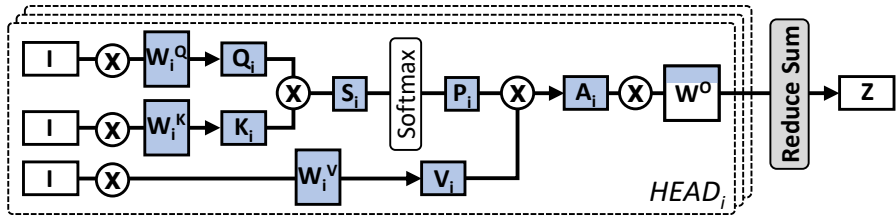
As the size of NLP models continuously increases, all NLP models suffer from a large amount of off-chip memory bandwidth (Section 4.1.2), which results in poor scalability. The current system (*baseline*) consecutively calculates each operation (listed in Table 4.1), which generates an immense amount of intermediate data between each operation. Figure 4.3a and Figure 4.3d describe the dataflow of baseline *multi-head self-attention* and *feed-forward network*, respectively. Note that we only show the case of multi-head self-attention as it includes all attention-related operations (i.e., *Q/K/V-Gen, dot product, softmax, weighted sum, attention FC*). As shown in Figure 4.3a, during the attention-related operations, the baseline system accesses multiple weight matrices (i.e.,  $W_i^Q, W_i^K, W_i^V, W^O$ ) and generates various intermediate data (i.e.,  $Q_i, K_i, V_i, S_i, P_i, A_i$ ). Similarly, as shown in Figure 4.3d, the baseline system uses multiple weights (i.e.,  $W_1^{FF}, W_2^{FF}$ ) and generates intermediate data (i.e.,  $T_1, T_2$ ) while passing through the feed-forward network.

To reduce the size of intermediate data, we propose three model partitioning optimizations: *partial-head update, column-based algorithm, and feed-forward splitting*. These optimizations can cover all types of operations in the NLP models. With these optimizations, NLP-Fast can significantly reduce the working set size from quadratic or cubic to linear space complexity, which enables the system to hide memory accessing overhead by prefetching data required in the next operation.

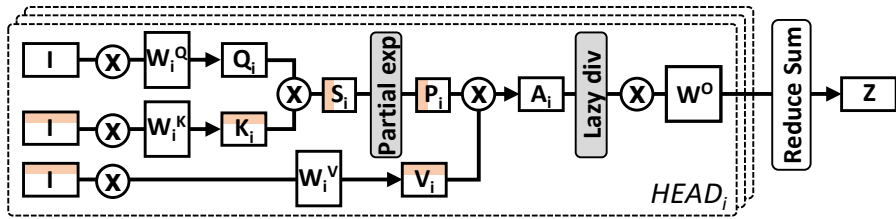
**Partial-head Update.** For the partial-head update optimization, we first exploit inherent parallelism in the multi-head attention component. As each head is independent of each other, we can separately execute each operation in head granularity. Then, we



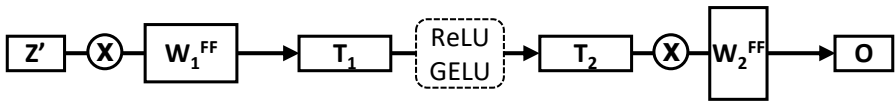
(a) Baseline dataflow of multi-head self-attention.



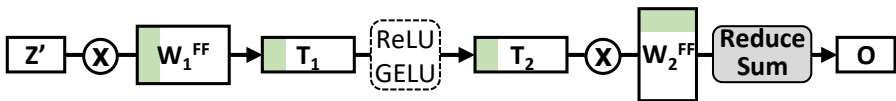
(b) Dataflow of baseline+P.



(c) Dataflow of baseline+P+C.



(d) Baseline dataflow of feed-forward network.



(e) Dataflow of baseline+F.

Figure 4.3: Dataflow comparison between baseline and our optimizations: *partial-head update* (P), *column-based algorithm* (C), and *feed-forward splitting* (F).

apply *partial attention FC* and *Reduce Sum* operations to partially calculate the final output ( $Z$ ) without waiting for attention results ( $A_i$ ) of all heads.

By doing so, different from the baseline system which needs to synchronize all attention results ( $A_i$ ) before the attention FC operation, NLP-Fast can directly compute the partial attention FC operation. Figure 4.3b shows modified computational steps and the dataflow after applying partial-head update. We highlight the reduced intermediate data (i.e.,  $Q_i, K_i, V_i, S_i, P_i, A_i$ ) and weights (i.e.,  $W_i^Q, W_i^K, W_i^V, W^O$ ) in the figure (color with light blue). All operations in each head use the reduced internal state vectors of which size is called *head size* ( $d_H$  divided by the number of heads,  $h$ ), and the head size is fixed in different NLP models [128, 32]. Therefore, the partial-head update optimization can successfully reduce the space complexity of weights and intermediate data used in attention-related operations.

**Column-based Algorithm.** Although the partial-head update optimization reduces the working set size of attention-related operations, some intermediate data still have high space complexity (e.g., score ( $S$ ), probability ( $P$ ), and other intermediate data<sup>1</sup>).

To reduce the size of the remaining intermediate data, we propose a *column-based algorithm* enabling NLP-Fast to partially generate  $K/V$  and calculate a partial attention result ( $A_i$ ) for these partial  $K/V$ . The key idea is a *lazy softmax calculation*, which computes the *softmax*'s division operation at last, not in the middle. Our advanced column-based algorithm is similar to MnnFast [62]. We apply a *lazy softmax calculation*, which computes the *softmax*'s division operation at last, not in the middle.

$$P = \sum_{\alpha} \frac{e^{q \times k_{\alpha}} \times v_{\alpha}}{\sum_{\beta} e^{q \times k_{\beta}}} = \frac{1}{\sum_{\beta} e^{q \times k_{\beta}}} \sum_{\alpha} e^{q \times k_{\alpha}} \times v_{\alpha} \quad (4.1)$$

Equation (4.1) shows the difference between the baseline (LHS) and the column-based algorithm (RHS) in the probability calculation process. Compared to the baseline, the column-based algorithm pulls the sum ( $\sum_{\beta} e^{q \times k_{\beta}}$ ) out of the outer summation ( $\sum_{\alpha}$ ). Since the sum does not depend on the index  $i$ , the column-based algorithm generates

---

<sup>1</sup>In addition to score and probability, many intermediate data are generated during softmax, we omit them for the sake of simplicity.

the same results to the baseline. By doing so, NLP-Fast does not need to wait for the sum of entire values in *softmax* and possible to calculate a partial attention result. In addition to partitioning the attention mechanism, our column-based algorithm also partitions input matrices ( $\mathbf{I}^K$  and  $\mathbf{I}^V$ ) used in K/V generation into multiple chunks. As the column-based algorithm only requires partial K/V to calculate the partial attention result, we generate these K/V chunks from corresponding input chunks. Figure 4.3c shows the dataflow of the modified computational steps when applying both partial-head update and column-based algorithm simultaneously. We highlight inputs (i.e.,  $\mathbf{I}^K$ ,  $\mathbf{I}^V$ ) and intermediate data (i.e.,  $\mathbf{K}_i$ ,  $\mathbf{V}_i$ ,  $\mathbf{S}_i$ ,  $\mathbf{P}_i$ ) reduced by the column-based algorithm (color with light orange).

**Feed-forward Splitting.** Now, we explain how our NLP-Fast can reduce the size of intermediate data generated in the feed-forward network. As explained in Section 2.2, the first part of the feed-forward network is the linear transformation (dimension of internal state vectors:  $d_H \rightarrow d_{FF}$ ) followed by an activation function (e.g., ReLU, GELU).

$$O = ActFunc(Z \times W) \quad (4.2)$$

There are two partitioning options to parallelize the first linear transformation: *row-based weight splitting* and *column-based weight splitting*. The first option splits the weight matrix and input matrix along its rows and columns respectively ( $[\mathbf{Z}_1, \mathbf{Z}_2] \times \begin{bmatrix} \mathbf{W}_1 \\ \mathbf{W}_2 \end{bmatrix}$ ). In this case, we cannot separately apply the activation function to each partial result of matrix multiplication because the activation function is a non-linear function. Therefore, this option requires an additional synchronization point before the activation function. However, another option (*column-based weight splitting*,  $[\mathbf{Z}] \times [\mathbf{W}_1, \mathbf{W}_2]$ ) does not require any synchronization point as each partial result is independent of each other. Therefore, we choose the column-based weight splitting to partition the first part of the feed-forward network. The partial output of the activation function directly passes through the second linear transformation with corresponding weights, and the outputs of the second linear transformation are reduced to calculate the final



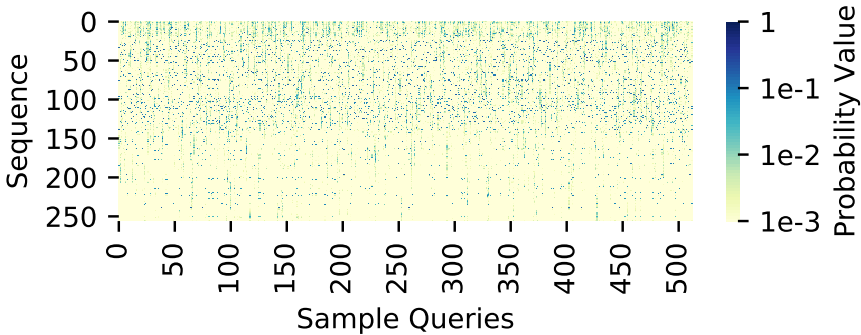


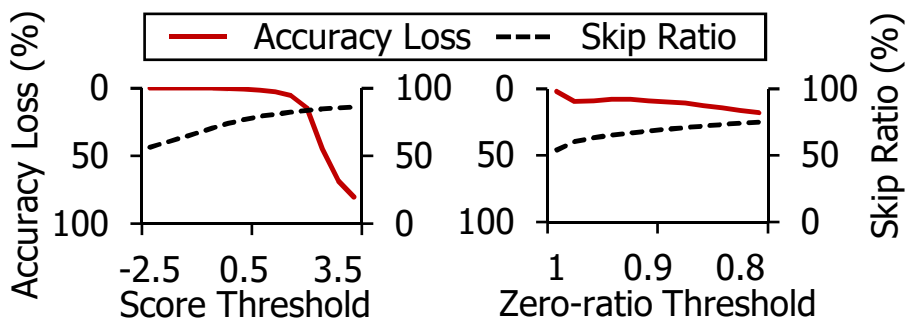
Figure 4.4: Probability value distribution. Each column is the probability vector of each query. We randomly choose 512 queries in BERT during inference on SQuAD.

output (O). Figure 4.3e illustrates how our *feed-forward splitting* optimization works by highlighting the partial chunks (i.e.,  $W_1^{FF}$ ,  $T_1$ ,  $T_2$ ,  $W_2^{FF}$ ) (color with light green).

### 4.2.3 Cross-operation Zero Skipping

The key observation for the proposed two zero-skipping optimizations (i.e., zero skipping, *V-Gen* skipping) is that the probability matrix has a huge imbalance (i.e., most probability values are close to zero). It is because the probability vector means the correlation between a query and keys, and only a few keys are related to the given query. Figure 4.4 shows the distribution of probability values in BERT. We use a pre-trained model [32] and conduct a fine-tuning for SQuAD dataset [101]. Then we measure probability values during inference. The results show that *only a few probability values are activated and most values are close to zero*.

**Zero Skipping in Various Operations – Zero skipping.** We first propose the *zero skipping* optimization to bypass a large amount of computation in *softmax* and *weighted sum*. In softmax, our zero skipping computes exponents of score values only when a score value is larger than a *score threshold*. If a score value is lower than the score threshold, a corresponding probability value is set to zero. In this case, the



(a) Zero skipping.

(b) *V-Gen* skipping.

Figure 4.5: Tradeoffs between accuracy loss and computation reduction according to the skip threshold.

number of zero values in a probability matrix increases proportionally to the number of skipped operations. In weighted sum, our zero skipping skips multiplications for zero probability values. By doing so, we can bypass both time-consuming exponentiation operations in softmax and multiplications in weighted sum.

This zero skipping optimization can significantly reduce the amount of computation in the attention mechanism; however, it also sacrifices the prediction accuracy. If the score threshold is too high, we can skip a large amount of computation; however, its prediction accuracy will dramatically decrease. On the other hands, if the score threshold is too low, we cannot get enough opportunity to reduce the quantity of computation. To quantify tradeoffs between accuracy loss and computation reduction, we measure both the accuracy loss and the ratio of computation reduction according to different score thresholds. We use SQuAD dataset for the inference test. Both accuracy loss and computation reduction are measured by comparing with the baseline’s accuracy and the amount of computation respectively. Figure 4.5a shows the results of the evaluation. The results show that zero skipping can achieve 79.0% computation reduction in both softmax and weighted sum while sacrificing only 1.4% of accuracy when the score threshold is 1.0, which is a negligible accuracy loss.

**Zero Skipping in Various Operations – V-Gen skipping.** In addition to the zero skipping optimization, we propose a more aggressive skipping optimization: *V-Gen skipping*. The key observation of V-Gen skipping is that, in the probability matrix, there are lots of columns in which almost all elements are zero. It is because each column represents the correlation between queries and a key, and some unimportant keys are not related to any queries.

As weighted sum is the matrix multiplication ( $P \times V$ ), if all elements of a certain column in the probability matrix (P) are zero, we do not need to generate a corresponding row in the value matrix (V). In other words, our *V-Gen skipping* skips generating the  $i^{th}$  row in the value matrix when most elements in the  $i^{th}$  column in the probability matrix are zero. We adopt a *zero-ratio threshold* to determine whether a row in the value matrix would be a skip target or not. For example, if the ratio of zeros in the  $j^{th}$  column is larger than the zero-ratio threshold, we do not generate the  $j^{th}$  row in the value matrix during *V-Gen* operation.

To quantify tradeoffs between accuracy loss and computation reduction, we conduct a similar measurement (Figure 4.5a) according to different zero-ratio thresholds. In this case, score threshold is 1.0. Figure 4.5b shows the results of the evaluation. The results show that *V-Gen* skipping can achieve 54.0% computation reduction in *V-Gen* operation while sacrificing only 1.8% of accuracy when the zero-ratio threshold is 1.0, which is negligible. If we apply more aggressive *V-Gen* skipping with lower threshold (0.9), we can achieve 68.2% computation reduction with 8.9% of accuracy loss.

**Skewness-minimized Dynamic Scheduler** Zero skipping and *V-Gen* skipping are beneficial to computation reduction; however, they incur a new problem: *execution time skewness*. This is because the amount of skipped computation is different according to each independent chunk. In this case, as a synchronization point exists at the last part of attention components, the overall execution time is delayed by the worst-case latency.

To quantify the impact of the time skewness problem, we measure the resource

utilization when we statically distribute chunks among threads. In this evaluation, we assume that 1024 chunks (64 chunks per head  $\times$  16 heads) are statically allocated across 32 threads, and each thread executes 32 chunks. We evaluate three different static schedulers: *head-first*, *chunk-first*, and *random*. The head-first and chunk-first schedulers preferentially allocate different heads and chunks to the same thread, respectively. The random scheduler randomly assigns 32 chunks to each thread. The results show that all static schedulers incur low resource utilization: head-first (41.6%), chunk-first (41.9%), and random (66.2%).

To maximize resource utilization, we propose a *dynamic scheduler* that assigns a chunk to an idle thread dynamically. By doing so, we can achieve high resource utilization (91.2%). Here, our dynamic scheduler needs an additional queue to hold chunk indices; however, the queue size overhead is negligible as the queue only needs to store index values of chunks, not the actual data.

#### 4.2.4 Adaptive Hardware Reconfiguration

To further improve the performance by reconfiguring the architecture to the target model, we propose *adaptive hardware reconfiguration*, applicable to FPGA. We find an optimal design point by taking into account the change of performance bottlenecks depending on models and their configurations,

Algorithm 2 shows how our proposed scheme finds the optimal design point. For the given model and its configuration ( $model_{config}$ ), we extract the number of key operations ( $k$ ) and resource usage ( $LUT_i, FF_i, DSP_i$ ) of a basic compute unit (e.g. MAC, Adder) for each operation. Then, we conduct a performance simulation which calculates expected latencies of each operation based on our static analysis in Section 4.2.1. With the extracted latencies, we solve a nonlinear programming for the objective function (line 2) with three constraints (line 3–5) by using sequential quadratic programming (SQP) methods. Here, the objective function is set to minimize an expected execution latency while meeting the resource constraints in FPGA. We assume that the

---

**Algorithm 2:** Adaptive design space exploration.

---

```
input   : A model configuration,  $model_{config}$  (i.e., model structure & parameter configurations).
input   : The number of operations in a target model,  $k$ .
input   : A list of tuples of resource usage of basic units for each operation,  $[<LUT_i, FF_i, DSP_i >]$ .
output  : A tuple of the degree of parallelism for each operation,  $<n_1, n_2, \dots, n_k >$ .

/* Conduct a performance simulation to get expected latencies of each
   operation. */
1  $<lat_1, lat_2, \dots, lat_k > = perf\_simulation(model_{config})$ 
/* Find optimized degrees of parallelism for each
   operation:  $<n_1, n_2, \dots, n_k >$ . */
2 minimize  $\sum_{i=0}^k \frac{lat_i}{n_i}$  subject to
3   (constraint-1)  $\sum_{i=0}^k LUT_i \times n_i < LUT_{total}$ 
4   (constraint-2)  $\sum_{i=0}^k FF_i \times n_i < FF_{total}$ 
5   (constraint-3)  $\sum_{i=0}^k DSP_i \times n_i < DSP_{total}$ 
/* Do iterative rebalancing between memory & compute to find the
   optimal  $n_i$  minimizing performance gap. */
6 while true do
7    $pf\_res = FPGA\_profiling(model_{config}, <n_1, \dots, n_k >)$ 
8    $<n'_1, \dots, n'_k > = resource\_rebalancing(pf\_res)$ 
9   if  $<n_1, \dots, n_k > \neq <n'_1, \dots, n'_k >$  then
10     $<n_1, \dots, n_k > = <n'_1, \dots, n'_k >$ 
11  else
12    break
13  end
14 end
15 return  $<n_1, \dots, n_k >$ 
```

---

latency of each operation is inversely proportionate to the amount of resources. By solving the problem, we can find the optimized degrees of parallelism for each operation ( $n_i$ ).

Next, we apply an iterative resource rebalancing technique to minimize the processing stall time caused by waiting for long memory access latency (line 6–14). We synthesize the current design and conduct performance profiling to quantify the processing stall time (line 7). Then we conduct the resource redistribution to minimize the stall overhead (line 8). This resource redistribution rebalances computations in stall minimized manner, as shown in Figure 4.6. First, we find an operation that has ex-

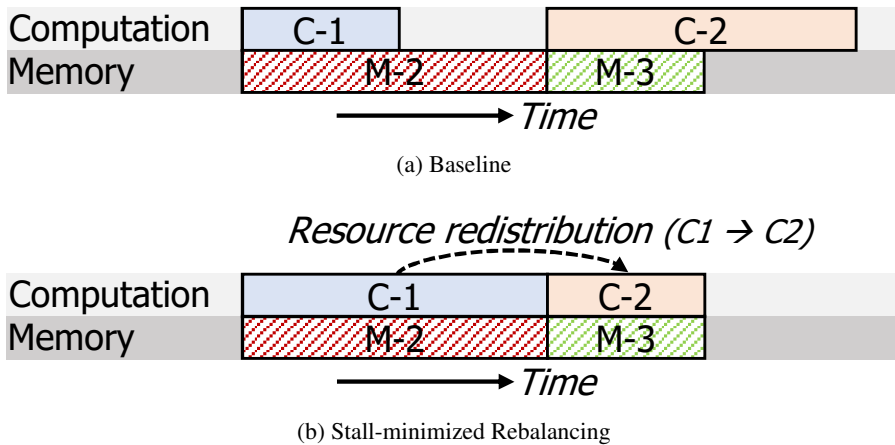


Figure 4.6: Stall-minimized resource rebalancing. C- $n$  and M- $n$  represent the execution time of compute and memory parts in  $n^{th}$  operation, respectively.

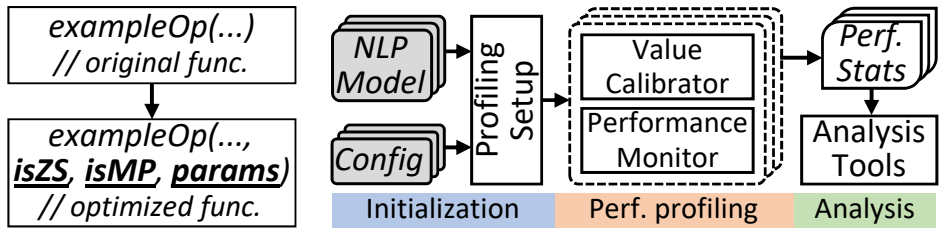
cessive computing resources (C-1 in Figure 4.6a). Next, we deallocate some resources from this operation by reducing the degree of parallelism and allocate the extracted resources to other operations (C-2 in Figure 4.6b). We iteratively perform this resource rebalancing until there is no further optimization opportunity. Finally, we return the optimal design point (line 15).

Note that *adaptive hardware reconfiguration* optimization can be only applicable to FPGA as typical hardware architecture (e.g., CPU, GPU, ASIC) have fixed designs for a certain purpose.

### 4.3 NLP-Fast Toolkit

We make the NLP-Fast toolkit that consists of the proposed three optimizations as well as our performance analysis tools. With the NLP-Fast toolkit, we can apply our optimization methodology to various NLP models with different parameter configurations. In this section, we highlight the usefulness and effectiveness of our NLP-Fast toolkit by explaining how to use the NLP-Fast toolkit for each NLP model.

Fig. 4.7 shows the high-level overview of the NLP-Fast toolkit. The NLP-Fast



(a) The pre-implemented NLP-Fast libraries. (b) The overview of our analysis tool (i.e., *NLP-Perf*) for analyzing various NLP models/configs.

Figure 4.7: The overview of our NLP-Fast toolkit. The left figure shows an example of applying NLP-Fast’s optimizations by using the pre-implemented NLP-Fast libraries. The right figure presents the overview of bottleneck analysis for given NLP models with various configurations.

toolkit mainly consists of two parts: pre-implemented libraries (Fig. 4.7a) and performance analysis tools (Fig. 4.7b).

First, we provide the pre-implemented NLP-Fast libraries with the proposed optimizations applied (i.e., *holistic model partitioning*, *cross-operation zero skipping*). With these pre-implemented libraries, the users can easily apply the proposed optimizations to their target NLP models by simply replacing original libraries with the NLP-Fast libraries.

Fig. 4.7a shows an example of applying NLP-Fast’s optimizations. For each operation, we add three additional parameters to original functions: *isZS*, *isMP*, and *params*. The boolean type *isZS* and *isMP* parameters determine whether the corresponding optimizations (e.g., *isZS*: cross-operation zero skipping, *isMP*: holistic model partitioning) turn on/off. The last parameter, *params*, is a tuple of various thresholds (e.g., score threshold, zero-ratio threshold, the degree of parallelism) used by the proposed optimizations.

By using the three parameters, the users can easily turn on/off each optimization with different thresholds. Note that, to apply NLP-Fast’s optimizations, the users do not need to modify their target NLP models, but simply set the three parameters (*good*

usability). We believe that our approach, preparing the pre-implemented NLP-Fast libraries, is easily applicable to other ML frameworks (e.g., PyTorch).

Second, we provide *NLP-Perf*, the performance analysis tool, to profile and analyze NLP models with various configurations. With *NLP-Perf*, the users can easily obtain various types of performance analysis results (e.g., performance breakdown in Fig. 4.2, cache hit/miss statistics in Fig. 4.12).

Fig. 4.7b illustrates how *NLP-Perf* profiles various NLP models with different configurations. *NLP-Perf* consists of three phases: *initialization*, *profiling*, and *analysis*. In the initialization phase, the users provide specific NLP models with target configurations to *NLP-Perf*. Based on the given NLP models/configs, *NLP-Perf* initializes inputs and weights with random values. The users can also set the profiling mode: *default*, *cachestat*, and *custom*. The *default* and *cachestat* modes measure each operation’s latency and LLC misses, respectively. For the *custom* mode, the users can select hardware performance events (e.g., core cycles, TLB-related events, I/D cache-related events) for their own purpose.

In the profiling phase, *NLP-Perf* executes every combination of given NLP models and corresponding configurations. At the runtime, a *value calibrator* dynamically adjusts the intermediate values to avoid operations on denormalized floating-point which are hundreds of times slower than on normalized floating-point on CPUs<sup>2</sup>. In addition to the value calibrator, a *performance monitor* collects the hardware performance events according to the profiling mode (i.e., *default*, *cachestat*, *custom*) configured in the initialization phase.

In the analysis phase, *NLP-Perf* collects the performance statistics from the profiling phase. The users can utilize pre-defined analysis tools (e.g., latency breakdown, cache analysis). With *NLP-Perf*, we believe that the researchers can gain insights for further optimizations.

---

<sup>2</sup>This performance degradation issue does not exist on GPUs as they can support denormals at full speed [138]



## 4.4 Implementation

To show the effectiveness of our NLP-Fast across various platforms, we implement the three representative models (i.e., memory networks, Transformer, BERT) in two versions (`Baseline` and `NLP-Fast`) on CPU, GPU, and FPGA. We first validate our idea on CPU and GPU, then we elaborate on the specialized hardware design based on the analysis of the general-purpose architecture.

### 4.4.1 General-Purpose Architecture – CPU

We implement both baseline and NLP-Fast on CPU for each model: memory networks, Transformer, and BERT). In the evaluation, we show the performance improvement of NLP-Fast and detail analysis by using CPU hardware performance counters (Section 4.5.2).

**Baseline CPU implementation.** We use C++ to implement each computational component and their sub-operations (described in Section 2.2) as a separate module. Then, we build each NLP model by combining these modules. In addition to these key components, we also implement other parts (e.g., final answer labeling in memory networks and residual connection, masking, and layer normalization in Transformer and BERT) according to the papers [125, 128, 32]. We verify our three different baseline models by comparing the final outputs of the baseline with the results of open-source projects provided by Facebook and Google [3, 8].

Now we explain the implementation details of each operation. Most operations (i.e., *Q/K/V-Gen*, *dot product*, *weighted sum*, *attention FC*, *linear trans-1/2*) are general matrix-matrix multiplication (GEMM). For the efficient GEMM computation, we use Intel Math Kernel Library (Intel MKL) [131] to fully utilize CPU resources. In addition to GEMM-based operations, we also implement other operations (i.e., *softmax*, *activation function*). For softmax, we apply the natural exponential function on the elements of a vector and normalize the exponential results with the summation.

For the activation function, we implement ReLU and GELU according to the BERT model [32].

We parallelize each operation as follows. For the GEMM-based operations, we exploit the multi-threading feature of MKL library. For other operations, we exploit data-level parallelization of each step with Pthreads.

**Holistic model partitioning.** As described in Section 4.2.2, we implement three model partitioning techniques: *partial-head update*, *column-based algorithm*, and *feed-forward splitting*. First, we partition the multi-head attention into separate heads by exploiting inherent parallelism and partial attention FC. Note that the dimension of internal states in each head is fixed; therefore, this partial-head update can significantly reduce the working set size of multi-head attention. Second, we split the attention mechanism into multiple sub-chunks by applying the column-based algorithm. This chunking also can be applied to *K/V-Gen* operations. Here, we can set the chunk size with no restrictions. Lastly, we partition the feed-forward network into multiple sub-chunks by applying the column-based weight splitting. Note that we can freely choose the size of each sub-chunk in both attention and feed-forward components. To hide memory accessing overhead, we use the built-in prefetch function provided by GCC 7.4.0.

**Cross-operation zero skipping.** As described in Section 4.2.3, we implement two zero-skipping optimizations in NLP-Fast: *zero skipping* and *V-Gen skipping*. First, we apply zero skipping for the attention mechanism. We perform a sensitivity analysis to find a proper score threshold for each model (e.g., we use 1 on BERT). Next, we implement *V-Gen* skipping to reduce the computation overhead of *V-Gen* operation. Here, the zero-ratio threshold is 0.9. In addition to these zero-skipping optimizations, we implement a *software-version dynamic scheduler* to minimize the execution time skewness.

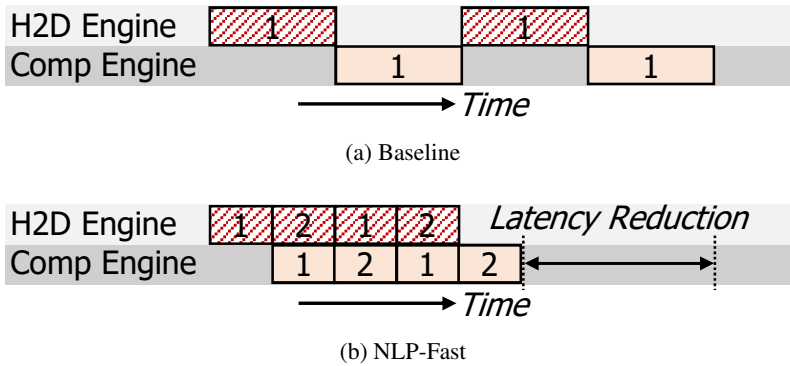


Figure 4.8: Performance improvement on a single GPU with multiple CUDA streams. NLP-Fast’s model partitioning enables GPU to exploit CUDA streams.

#### 4.4.2 General-Purpose Architecture – GPU

We also implement GPU-version baseline and NLP-Fast for each NLP workload supporting various configurations. In the evaluation, we present single-GPU performance improvement as well as multi-GPU scalability (Section 4.5.3).

**Baseline GPU kernel implementation.** Similar to the CPU-version baseline, we implement each operation as separate GPU kernels and combine these operations into a specific NLP model. We verify our three different GPU baseline models by comparing their results to the corresponding CPU-version baseline results.

We use cuBLAS [1] provided with CUDA Toolkit 10.2 to perform GEMM-based operations. By using the state-of-the-art GPU BLAS library, we aim to avoid any inefficiency incurred by unoptimized kernel implementation. Different from GEMM-based operations, we implement custom kernels for other operations (e.g., softmax, activation function).

**Holistic model partitioning.** We implement three model partitioning techniques for GPU-version NLP-Fast. For *partial-head update* and *feed-forward splitting*, we add a custom GPU kernel (called reduce sum) to reduce the results within each partitioned block (e.g., heads, chunks). For *column-based algorithm*, we split the softmax

kernel into two kernels: partial exponentiation and lazy division. By doing so, we can holistically partition all NLP models into small-chunk granularity.

This data chunking capability of NLP-Fast enables GPU to exploit CUDA streams to overlap kernel executions by data transfers between the host and GPU. Figure 4.8 illustrates how NLP-Fast hides the data transfer overhead with CUDA streams. With CUDA streams, NLP-Fast can successfully improve a single-GPU performance by minimizing the data transfer overhead.

Also, we implement a multi-GPU version of NLP-Fast capable of running an extremely large NLP model which cannot be executed on the single-GPU environment. As NLP-Fast requires only a few synchronization points, our multi-GPU version of NLP-Fast achieves scalable performance.

**Cross-operation zero skipping.** Zero skipping is ineffective or even harmful for GPUs [91, 56, 62]. As GPU executes groups of threads known as warps (32 threads per warp), zero skipping is effective only if *all* threads in the warp are zero skipped, which is very unlikely. To eliminate this poor utilization problem, we can transform a dense matrix into a sparse matrix, but the transforming overhead is too high. We implement the transformation by following an example of the official cuSPARSE document [2], but the transforming overhead is even comparable to *weighted sum*. Therefore, we do not apply zero skipping to GPU-version NLP-Fast.

### 4.4.3 Custom Hardware (FPGA)

We design and implement each NLP model using Vivado High-Level Synthesis (HLS). Figure 4.9 shows the high-level architecture of FPGA-version NLP-Fast for BERT. We omit the baseline design as it is too straightforward. Note that other models (e.g., Transformer, Memory networks) can be implemented by removing unnecessary operations.

**Baseline FPGA implementation.** We first implement all basic compute units required by operations in NLP models. Basic compute units are listed as follows: MAC,

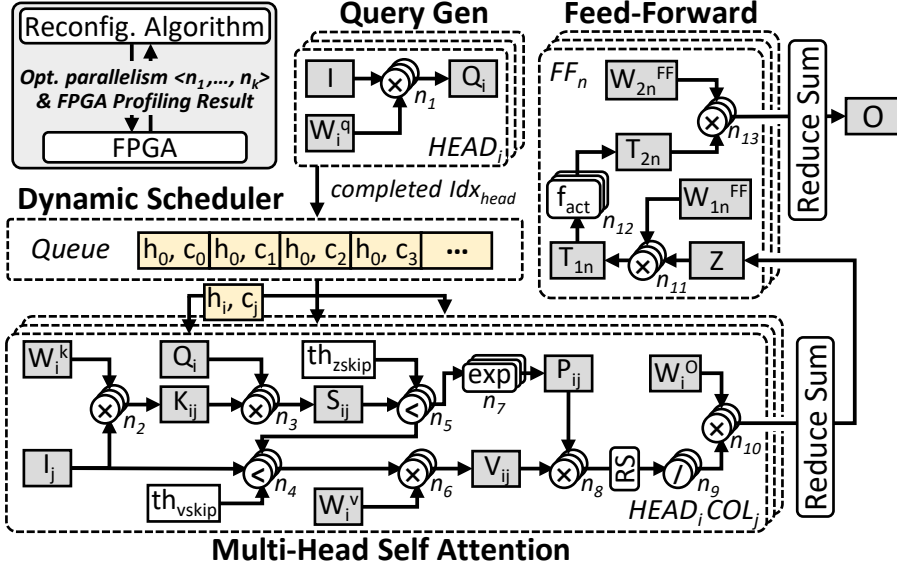


Figure 4.9: The architecture of FPGA-based NLP-Fast.

comparator, exponentiator, divider, adder, multiplier, and square root. We use *exp* and *sqrt* function of HLS math library for exponentiator and square root. In addition, we implement other non-performance critical parts (i.e. layernorm, residual sum).

**Holistic model partitioning.** Our FPGA-version NLP-Fast consists of three execution units (i.e., *query generator*, *multi-head self-attention*, *feed-forward*), and the dataflow of each execution unit follows Figure 4.3c and Figure 4.3e. For each operation, NLP-Fast fetches required data from DRAM to BRAM. We use AXI4 for data transfer. To overlap memory prefetching and computation, we explicitly remove dependencies of each BRAM block by using HLS directives.

**Cross-operation zero skipping.** We set each threshold value through AXI port at the beginning of execution. We choose the same threshold values as CPU (1 for score threshold,  $th_{zskip}$  and 0.9 for zero-ratio threshold,  $th_{vskip}$ ). For zero skipping, we use comparator units to compare a score value with  $th_{zskip}$ . For *V-Gen* skipping, we use comparator and adder units to count the number of zeros. In this case, we need an extra counter ( $\log_2(\text{sequence length})$  bits) and check bit (1 bit) per each column, which

Table 4.2: Base configurations of each NLP model.

Entry	CPU			GPU			FPGA		
	MemNet	TF	BERT	MemNet	TF	BERT	MemNet	TF	BERT
$s(ns)$	100M	256	256	100M	256	256	1000	256	256
$d_H$	64	1024	1024	64	1024	1024	64	1024	1024
$d_{FF}$	-	4096	4096	-	4096	4096	-	4096	4096

is under 1KB overhead in total. Overall, the logic overhead is negligible, as it is just a simple adder logic.

**Dynamic scheduler.** We implement the dynamic scheduler with a FIFO queue. When  $Q$ -Gen is finished for any head, the index of the head is delivered to the dynamic scheduler. Then, the dynamic scheduler pushes all  $\langle Idx_{head}, Idx_{chunk} \rangle$  pairs of the corresponding head to the queue. If there is any idle execution unit, the scheduler assigns an index pair to the unit. We choose the queue size as twice the number of multi-head self-attention units. Note that the queue size is negligible as the queue only stores index pairs which are  $\langle \log_2(\#head) \text{ bits}, \log_2(\#chunk) \text{ bits} \rangle$ .

**Adaptive Hardware Reconfiguration.** We use a *scipy* python module with the SLSQP method to resolve the objective function described in Section 4.2.4. We apply the degrees of parallelism to the FPGA by setting the number of each basic compute unit in FPGA (e.g., MAC, comparator) using HLS directives. For the resource rebalancing function described in Section 4.2.4, we find the target operations for resource reallocation by the sequential search.

## 4.5 Evaluation

We achieve the *high model coverage* by accelerating three representative models that existing proposals cannot support.

### 4.5.1 Experimental Setup

We implement the three representative models (i.e., memory networks, Transformer, BERT) in several versions (i.e., baseline, baseline with each optimization, NLP-Fast) on various hardware platforms: CPU, GPU, and FPGA.

**CPU eval.** We compare baseline with NLP-Fast on two Xeon CPUs (Intel Xeon Gold 5220 18C/36T) system with DDR4-2400MHz 256GB memory. We run NLP models on Ubuntu 16.04 LTS and use Intel MKL [131] for GEMM.

**GPU eval.** We use a Supermicro SuperServer 4028GR-TRT with two Intel Xeon CPU E5-2650 v4 and four Nvidia TITAN Xp GPUs attached via PCIe Gen 3. We measure the performance of GPU-based NLP-Fast with Linux kernel version (4.4.0-142-generic) and CUDA Toolkit version 10.2.

**FPGA eval.** We use Xilinx Virtex Ultrascale+ FPGA VCU118 and DDR4 memory. The programmable logic (PL) runs at 100MHz. The memory has 32-bit effective width. To control NLP-Fast, we build a monitoring program executed on MicroBlaze.

**NLP model config.** Table 4.2 shows the types of NLP models and their *base* parameter configurations for the evaluation. For *base* configurations, we get the parameters from the Transformer and BERT paper for a fair evaluation [128, 32]. For S/H/F-4x configurations, we increase each parameter (i.e.,  $s$ ,  $d_H$ ,  $d_{FF}$ ) in four times, respectively.

### 4.5.2 CPU

**Performance.** Figure 4.10 shows the performance of NLP-Fast on various NLP models and different configurations. For each model, we measure latencies of three implementations: *baseline*, *baseline+MP* (i.e., baseline with optimization called *holistic model partitioning*), and *NLP-Fast*. Our model partitioning achieves  $1.73\times$  average speedup on various NLP models. With all optimizations, NLP-Fast achieves  $2.15\times$  average speedup. Memory networks gets huge performance improvement from *cross-operation zero skipping* because zero skipping significantly reduces the computation overhead in attention-related operations (Note that attention mechanism is dom-

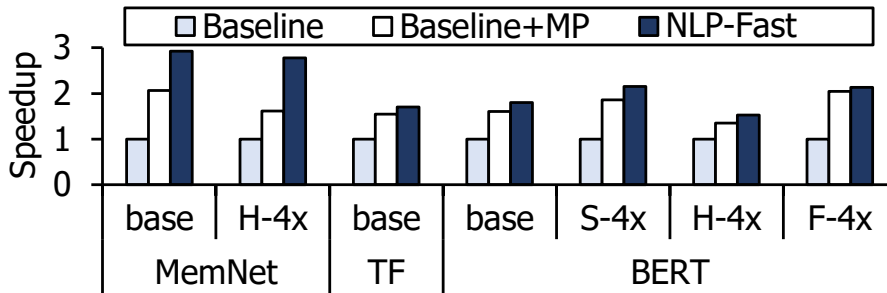


Figure 4.10: Performance improvement of CPU-based NLP-Fast on various NLP models and configurations. MP means *model partitioning*.

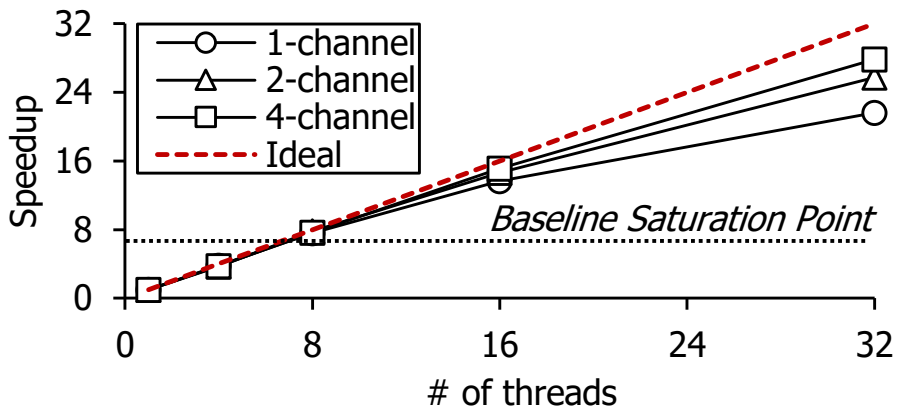


Figure 4.11: Scalability on different memory bandwidth.

inant in memory networks as early illustrated in Figure 4.2). In this case, NLP-Fast achieves performance improvement by up to  $2.92\times$ .

**Scalability: Cache efficiency.** We measure the performance improvement of parallelization at different memory bandwidth configurations (# of memory channels) to evaluate the scalability of NLP-Fast on CPU. Figure 4.11 shows speedup normalized to the 1-thread case on each memory configuration. Ideally, the performance should be linearly proportionate to the number of threads. Different from the baseline whose performance is saturated at 8-thread, our NLP-Fast shows highly-scalable performance on



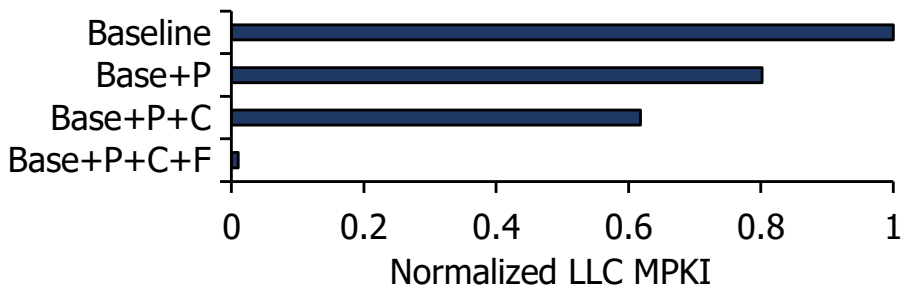


Figure 4.12: Normalized LLC MPKI for each partitioning optimization. P/C/F means partial-head update, column-based algorithm, and feed-forward splitting.

various memory bandwidth configurations. This is because NLP-Fast’s holistic model partitioning significantly reduces the working set size, which aids CPU in using cache more efficiently.

To quantify the impacts of our model partitioning techniques (i.e., *partial-head update*, *column-based algorithm*, *feed-forward splitting*), we measure LLC misses per kilo instructions while applying each model partitioning optimization step-by-step. We use the Linux performance monitoring library [37] to get LLC misses. Figure 4.12 shows the LLC MPKI normalized to the baseline. Partial-head update and column-based algorithm reduce off-chip DRAM accesses by 20% and 18%, respectively. With all three model partitioning techniques, we eliminate almost all off-chip DRAM accesses.

### 4.5.3 GPU

**Performance.** We evaluate the performance improvement of NLP-Fast on the single-GPU environment. As explained in Section 4.4.2, we do not apply the zero skipping optimization to GPU-version NLP-Fast. Figure 4.13 shows the performance improvement of NLP-Fast on various NLP models. As described in Section 4.4.2, our holistic model partitioning technique enables NLP-Fast to exploit CUDA streams to overlap kernel executions by data transfers between host and device. By doing so, NLP-Fast

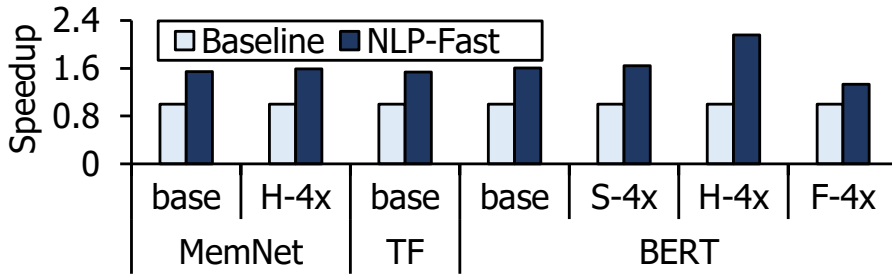


Figure 4.13: Single-GPU performance improvement of GPU-based NLP-Fast on various NLP models.

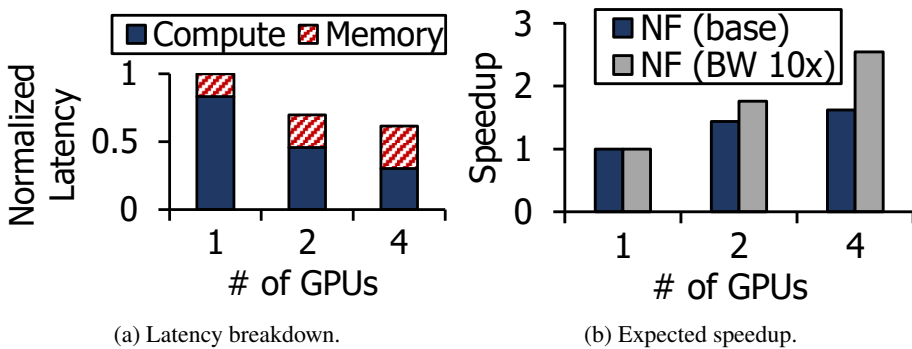


Figure 4.14: The overhead analysis of multi-GPU version of NLP-Fast (NE) and the expected speedup of NLP-Fast with high bandwidth (e.g., NVLink 2.0).

achieves  $1.63\times$  average speedup on various NLP models. In addition to optimizing data transfers, CUDA streams aid in increasing the computational efficiency of GPU; therefore, BERT with H-4x configuration achieves more performance improvement than other cases ( $2.16\times$  speedup).

**Scalability: scale-out multi-GPU.** NLP-Fast utilizes multiple GPUs more efficiently than the baseline system because our holistic model partitioning optimization reduces the working set size. Note that our extra communication overhead for supporting multi-GPU version NLP-Fast is negligible as NLP-Fast does not require many synchronization points.

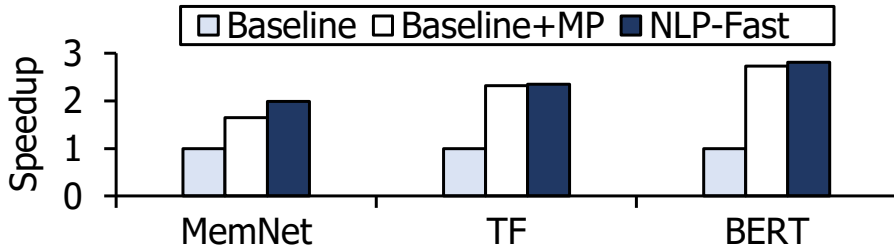


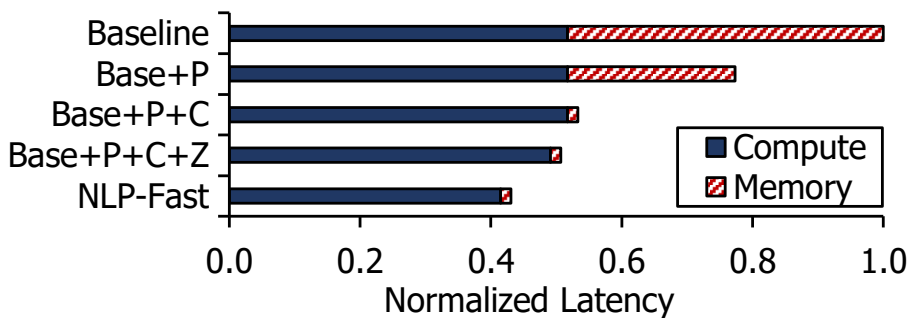
Figure 4.15: Performance of FPGA-based NLP-Fast on various models. MP means model partitioning.

By exploiting multiple GPUs, NLP-Fast can run an extremely large NLP model which cannot be executed on the single-GPU environment. Figure 4.14 shows the results of the multi-GPU evaluation. Figure 4.14a shows the normalized latency of NLP-Fast on different numbers of GPUs (normalized to the single-GPU case) and the overhead breakdown (compute vs. memory). In this case, NLP-Fast achieves  $1.44\times$  and  $1.63\times$  speedup on two GPUs and four GPUs, respectively.

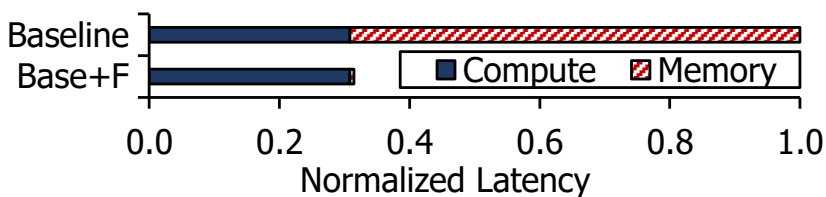
Currently, NLP-Fast cannot achieve scalable performance due to high memory accessing overhead. The latency breakdown (Figure 4.14a) shows that the memory overhead takes 16.9%, 34.3%, and 51.0% on 1, 2, and 4 GPUs, respectively. Fortunately, this high memory accessing overhead is continuously alleviated as interconnect technology scales. For example, NVIDIA NVLink 2.0 [4] achieves  $10\times$  more bandwidth than PCIe Gen 3. Figure 4.14b shows the expected speedup of NLP-Fast with high-bandwidth interconnect technology. Here, we assume NVLink 2.0 ( $10\times$  more bandwidth). In this case, NLP-Fast can achieve  $1.76\times$  and  $2.55\times$  speedup on two and four GPUs, respectively.

#### 4.5.4 FPGA

**Performance.** Figure 4.15 shows the performance of NLP-Fast on various NLP models. The results illustrate that NLP-Fast achieves gradual speedup as we apply the holistic partitioning and cross-operation skipping. First, our holistic model partitioning



(a) Multi-head self-attention.



(b) Feed-forward network.

Figure 4.16: Latency reduction of FPGA-based NLP-Fast on BERT. Each latency is normalized to baseline. *P/C/Z/F* represents partial-head update/column-based algorithm/zero skipping/feed-forward splitting, respectively.

provides  $2.72\times$  speedup by minimizing the data spilling and enabling data prefetching. Also, cross-operation zero skipping with dynamic scheduler achieves an additional  $1.20\times$  speedup by skipping unnecessary computations and removing skewness. With all schemes applied, NLP-Fast accomplishes total speedup by up to  $2.81\times$ .

We quantify the impacts of each optimization on BERT. We implement five versions on multi-head attention (i.e., baseline, partial-head update, column-based algorithm, zero skipping, V-Gen skipping), and two versions on feed-forward (i.e., baseline, feed-forward splitting). Figure 4.16a shows the latency breakdown of the multi-head self-attention part, which is normalized to the baseline. First, *partial-head update* (P) and *column-based algorithm* (C) gradually reduce the execution latency by removing memory overhead. Next, our zero skipping (Z) reduces computation overhead. Lastly, with *V-Gen* skipping, NLP-Fast achieves latency reduction by 57.0% compared

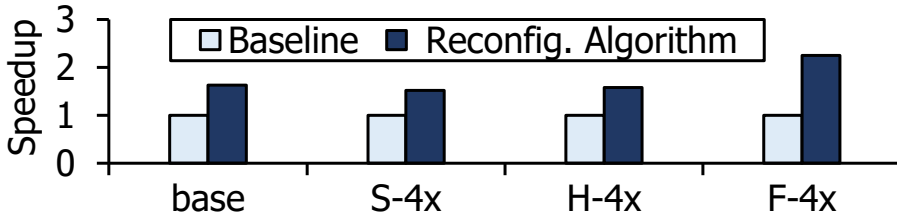


Figure 4.17: Effectiveness of *adaptive hardware reconfiguration* on various configurations of BERT. S/H/F-4x are parameter configurations with a fourfold increase of  $s$ ,  $d_H$ , and  $d_{FF}$ , respectively.

to the baseline. Figure 4.16b is the latency breakdown of the feed-forward part. Each Latency is normalized to the baseline. The results show that *feed-forward splitting* (F) achieves latency reduction by 68.5%.

**Adaptive hardware reconfiguration.** To show the effectiveness of our *adaptive hardware reconfiguration*, we implement NLP-Fast with various configurations of BERT on FPGA. Figure 4.17 is the speedup results of each configuration with an optimized design point. The speedup is normalized to the execution latency of each configuration with the baseline design point. As the latency of each operation significantly varies with parameters, we assume the baseline design point that all operations have a similar amount of resources. The results show that our scheme achieves more than  $1.5\times$  speedup on every configuration. This improvement occurs as our scheme redistributes resources from non-performance critical operations to other operations. Especially, for F-4x configuration, *linear trans* operations take 74.7% of latency while other operations take 25.3% in total. Therefore, F-4x configuration achieves a higher speedup than other cases ( $2.25\times$ ).

We observe that *adaptive hardware reconfiguration* is essential because the specific design point, optimized for certain configuration, works poorly on other configurations. For example, if we run BERT with H-4x configuration on the FPGA accelerator designed for F-4x configuration, we can only get 76.2% of the performance we get from an optimal design for H-4x configuration. Moreover, this performance loss is

aggravated as the model size increases (e.g., 46.7% in an eightfold increase, 29.7% in a sixteen-fold increase).

## **4.6 Conclusion**

We propose NLP-Fast, a novel system solution for heterogeneous NLP models to achieve fast and scalable performance. We identify performance-critical operations in various NLP models and their performance problems. To resolve the problems, we propose three techniques to optimize various NLP models on any hardware platforms: CPU, GPU, and FPGA.

## Chapter 5

### Related Work

#### 5.1 Various DNN Accelerators

A DNN is basically composed of a series of linear algebra operations. Variants of existing DNNs usually have different requirements including new operations, which can motivate researchers to develop new types of DNN accelerators. For example, Lin et al. [78] investigate various neural networks used in autonomous driving systems, identify key computational bottlenecks, and provide acceleration techniques applicable to different hardware platforms: GPU, FPGA, and ASIC. Kung et al. [70] provide optimization techniques for specific neural networks (e.g., multi-layer cellular nonlinear network) to improve GPU's energy efficiency. Some prior work proposes both software and hardware optimizations for low-precision SGD [31]. Also, there are several attempts to optimize GAN-based deep learning [118, 142].

In addition to the software optimizations, some prior work proposes hardware-specific techniques for different types of DNNs and CNNs [20, 40, 112, 117]. Some studies try to accelerate DNNs with respect to memory [26, 44], while others present acceleration with low power [25].

As these model-specific optimizations are challenging, some researchers propose systematic accelerator performance analysis and optimization methodology [13, 104,

113, 105, 123, 82, 122, 41]. In the dissertation, I focus on addressing the performance problems of various NLP models (e.g., Memory networks, Transformer, BERT). Different from typical prior NN acceleration work, our approach is a generic solution applicable to various NLP models, thanks to our in-depth model analysis and general software optimizations.

## 5.2 Various NLP Accelerators

Recently, neural networks have risen as new information processing paradigms in natural language processing (NLP) as they break records on many NLP tasks. Recurrent neural networks (RNNs), designed to work on sequence prediction problems, derive an answer to a question from the previous reasonings [86, 57]. There are some studies with different accelerating approaches for reconfigurable hardware to accelerate RNNs [22, 50, 74, 43] and LSTMs [133].

However, RNNs and LSTMs cannot memorize the previous history for a long time [15] nor handle a large amount of history due to their small memory [136]. Therefore, they cannot perform sophisticated tasks requiring a large amount of memory, such as a task which comprehends a series of books to provide useful information to users. To resolve these limitations, big technology companies (e.g., Google, Facebook, Microsoft) and open-source research communities (e.g., OpenAI) actively propose new types of *complex* NLP models (e.g., memory-augmented neural networks [136, 125], attention-based neural networks [128, 32]). Nowadays, these complex emerging NLP models are widely used in various NLP tasks from simple dialog comprehension to large-scale question & answering systems using a large-scale dataset (e.g., Wikipedia) [134, 55, 34, 87].

Unfortunately, there are only a few proposals aiming to accelerate these new types of complex NLP models. Ham et al. [52] propose a specific architecture conducting approximation on the attention mechanism. They exploit the inherent sparsity in the



attention mechanism to reduce a large portion of computations with near-zero values. Also, they provide the hardware architecture specialized for those approximation-enabled attention networks. Jang et al. [62] present three general optimizations to reduce working set size in the attention mechanism, minimize the overall computations, and avoid a cache conflict between inference and embedding operations. Then, they apply the three optimizations on different hardware platforms (i.e., CPU, GPU, FPGA) to show the effectiveness of the proposed techniques.

However, these studies cannot always be effective due to the limited range of acceleration (only for the attention mechanism). As described in the dissertation, the state-of-the-art NLP models (e.g., Transformer, BERT) consist of various combinations of complex operations, and the attention mechanism is not a major performance bottleneck in most models/configurations. Therefore, those attention-specific optimizations are not highly efficient for the emerging NLP models. In the dissertation, I focus on accelerating the whole types of NLP models as well as upcoming new types of models to cover various NLP models.

Stevens et al. [124] present a memory-centric hardware architecture for attention-based networks, which is also beneficial for NLP-Fast. TensorRT [6] and ONNX Runtime [5] propose sub-graph fusion to reduce memory accesses and more self-attention heads to enhance parallelism, which is orthogonal to our optimizations: holistic model partitioning and cross-operation zero skipping.

### **5.3 Model Partitioning**

To achieve higher accuracy, neural networks are becoming bigger and more complex. Not only DNNs and CNNs [45, 59, 118], but also various *large-scale* NLP models are actively proposed in both industry and academia [116, 17, 72]. This large-scale NLP models increase the working set size, which incurs extremely low cache efficiency and increases high off-chip memory bandwidth requirements.

To resolve this memory accessing overhead, some studies propose model partitioning techniques to handle the growing size of DNNs, CNNs, and NLP models. Gao et al. [45] propose dataflow optimizations exploiting both intra-layer parallelism and inter-layer pipelining. Lu et al. [80] introduce a flexible dataflow architecture for various CNN workloads. Jia et al. [65] use a deep learning engine to find a customized parallelization strategy. For NLP models, Jang et al. [62] propose model partitioning for the attention mechanism. With their novel column-based algorithm, they successfully reduce the working set size and achieve high cache efficiency in each hardware platform: CPU, GPU, and FPGA. Some groups exploit the inherent parallelism in the multi-head attention to reduce the working set size in Transformer-based models [116].

Note that these proposals cannot support a wide spectrum of NLP models as each work only focuses on a specific operation or NLP model. In recent large-scale NLP models, the most critical characteristic is that different parts can be scaled-up individually. For example, each parameter in Transformer-based models can be scaled-up to enhance the model accuracy, which results in high model/config diversity. In the dissertation, our holistic model partitioning provides a high model coverage by optimizing all types of operations used in recent NLP models.

## 5.4 Approximation

Fault-tolerance of NNs [127, 126] allows NNs to adopt approximation with small accuracy loss while improving energy efficiency [110]. Fault-tolerance varies by NNs [103]. Some studies help architects properly use the approximation [11, 19, 42]. The excessive approximation can threaten our safety [151], and also degrade performance and energy if subsequent tasks are inefficient on inaccurate data [129, 145, 146].

**Network pruning** is an effective optimization, especially for constrained environments [47, 96, 27]. Static pruning is usually independent of hardware, while dynamic pruning relies on software-hardware co-designs. Static pruning can greatly reduce NN

size with little loss of accuracy [60] and domain adaptation ability [139]. Song et al. [120] improve the accuracy of overly-pruned NNs by incremental updates. Dynamic pruning can reduce the working set size [120, 144] using runtime information. Value prediction [95, 85, 143] can increase pruning candidates [10, 119]. Dynamic pruning can produce a sparse matrix whose irregular data access patterns can negate the pruning benefits; thus, accelerators specialized for sparse matrices [148, 53, 93, 54, 63, 75] are proposed. Generative adversarial networks (GANs) *populate* zeros as opposed to pruning, but require sparse DNN accelerators [118, 142].

Some studies propose effective pruning techniques for Transformer-based NLP models [84, 48]. They focus on pruning Query/Key/Value generation parts, attention FC, and linear transformations in the feed-forward network. In addition to network pruning, there are different approaches to compress BERT by leveraging knowledge distillation [111]. Note that these network pruning approaches are orthogonal to our zero skipping optimizations as the zero skipping optimizations are used in attention-related operations. I believe combining these pruning techniques with the zero skipping optimizations can significantly reduce both memory bandwidth and computation requirements.

**Precision** can also be adjusted. Low-voltage SRAM [16] is universally applicable to accelerators. Brandon et al. [102] propose a novel lossy weight encoding scheme. Processing in memory (PIM) performs read and computation simultaneously while losing reliability [39], which is appealing features for DNN accelerators [36]. The PIM reliability issue is handled by [39, 38, 77]. Quantization decreases overall bitwidths [81, 51, 96]. Park et al. [94] use two different bitwidths to handle outliers. Ding et al. [33] replace expensive multipliers with cheaper shifters and adders.

Note that the above various approximation techniques (e.g., network pruning, low precision, quantization) can be independently applied with the proposed zero-skipping optimizations in this dissertation.

## 5.5 Improving Flexibility

In neural networks, the strong binding of fast-evolving software and optimized hardware discourages actual chip development [71]; therefore, some accelerators try to exploit the reconfigurable hardware to satisfy this rapid advancement in recent neural networks. For example, some studies utilize FPGA’s flexibility by reconfiguring interconnects or rebalancing the hardware resources in the most efficient way for DNNs [71, 97] and CNNs [115]. Some prior work exploits the flexibility by concentrating on most compute-intensive code regions to maximize resource utilization [90, 88].

The FPGA’s flexibility is highly beneficial for neural networks; however, users cannot easily adopt FPGA for a target accelerator because of its programming difficulty (e.g., Verilog). To ease the programming difficulty, some studies use high-level synthesis to design their accelerator rather than using a hardware description language [46, 23, 24]. Others propose ISA-driven spatial accelerators, using autonomous fixed-ISA processing elements [106]. In addition to the programming language, some studies aim to automate the design space exploration by exploiting compiler techniques [64, 121]. Virtual memory [89] eliminates the limit on runnable NNs but also degrades performance unless carefully designed [107, 61].

## 5.6 Resource Optimization

In addition to exploiting the FPGA’s flexibility by proposing a target-specific design, finding an optimal design point is also critical in FPGA-based accelerators. For this purpose, some studies provide frameworks to automatically explore various design points for various NN models [14, 83, 21, 108, 58, 67, 114]. In resource optimization, there are different goals (e.g., performance, power, throughput, resource utilization). Among these goals, some groups aim to maximize resource utilization [99, 150, 147]. Palesi et al. [92] propose a technique for Pareto-optimal configurations on parameterized system-on-a-chip architecture. Krishnan et al. [69] introduce a framework for

efficient design space exploration using a genetic algorithm.

In the dissertation, NLP-Fast leverages the FPGA's flexibility to find optimal design points for each NLP model and configuration by dynamically rebalancing the hardware resources. NLP-Fast allocates more resources on compute-intensive operations identified by our in-depth performance analysis while minimizing stalls due to memory overheads.

## Chapter 6

### Conclusion

Emerging natural language processing (NLP) models have become more complex and bigger to provide more sophisticated NLP services. Accordingly, there is also a strong demand for scalable and flexible computer infrastructure to support these large-scale, complex, and diverse NLP models. However, existing proposals cannot provide enough scalability and flexibility as they neither identify nor optimize a wide spectrum of performance-critical operations appearing in recent NLP models and only focus on optimizing specific operations.

The dissertation proposes two novel projects for large-scale heterogeneous NLP models to achieve fast and scalable reasoning performance. The first project is MnnFast that accelerates large-scale memory-augmented neural networks (e.g., memory networks). In this project, the attention mechanism (commonly used operation in recent NLP models) is the major performance bottleneck of the target memory-augmented neural networks. MnnFast conducts extensive performance bottleneck analysis and identifies that the current architecture suffers from three major performance problems in the large-scale attention mechanism: high memory bandwidth consumption, heavy computation, and cache contention. To overcome these performance problems, MnnFast proposes three novel optimizations.

First, to minimize the off-chip memory accessing overhead, MnnFast presents a

new column-based algorithm with streaming which minimizes the size of data spills and hides most of the off-chip memory accessing overhead. Second, to reduce the computational overhead, MnnFast proposes a zero-skipping optimization to bypass a large amount of output computation by exploiting high sparsity in the attention mechanism. Lastly, to eliminate the cache contention, we propose an embedding cache dedicated for the embedding operations.

Evaluation results show that MnnFast is significantly effective in various hardware platforms: CPU, GPU, and FPGA. MnnFast improves the overall throughput by up to  $5.38\times$ ,  $4.34\times$ , and  $2.01\times$  on CPU, GPU, and FPGA respectively.

The second project is NLP-Fast that aims to design a fast, scalable, and flexible system architecture for heterogeneous NLP models. As aforementioned, the state-of-the-art NLP models are becoming diversified, which results in huge performance variation. For example, the attention mechanism is not a major performance bottleneck in recent NLP models anymore. To support a wide spectrum of NLP models, NLP-Fast builds the performance analysis tool for heterogeneous NLP models (e.g., memory networks, Transformer, BERT) and conducts extensive profiling on different configurations. By doing so, NLP-Fast identifies the performance-critical operations and their performance bottlenecks in the NLP models.

Then, to remove the identified performance bottlenecks, NLP-Fast proposes two general-purpose optimizations applicable to any hardware platforms and one hardware-specific optimization for the reconfigurable hardware platform. For the general-purpose optimizations, one optimization is holistic model partitioning which combines three model partitioning techniques (i.e., partial-head update, column-based algorithm, feed-forward splitting) to enable end-to-end model partitioning. Another optimization is cross-operation zero skipping which skips computations with zero/near-zero values over multiple operations. With these optimizations, NLP-Fast can holistically reduce both memory and computation overheads in various NLP models on any hardware platforms. To further improve the performance by reconfiguring the architecture to the

target model, we apply a workload-specific method, model/config adaptive hardware reconfiguration, applicable to FPGA. By taking into account the change of performance bottlenecks depending on models and their configurations, the proposed optimization reallocates the FPGA's resources to prioritize the most critical bottlenecks.

For the evaluation, we apply two general-purpose optimizations to TensorFlow-based NLP models for CPU and GPU platforms. We also implement our baseline models for FPGA and apply three optimizations: two general-purpose and one hardware-specific optimization. The evaluation results show that CPU-based NLP-Fast achieves up to  $2.92\times$  speedup ( $2.00\times$  on average) and shows scalable performance using more cores. GPU-based NLPFast achieves up to  $1.59\times$  speedup ( $1.44\times$  on average) in the single-GPU environment and shows scalable performance using more GPUs. FPGA-based NLP-Fast achieves up to  $2.89\times$  speedup ( $2.59\times$  on average), and up to  $4.47\times$  speed up with its adaptive hardware reconfiguration enabled.

In summary, the dissertation shows that the proposed optimizations and system design methodology successfully support a wide spectrum of emerging NLP models on various hardware platforms.



# Bibliography

- [1] “cuBLAS,” Available online at <https://docs.nvidia.com/cuda/cublas/index.html>, accessed: 2020-04.
- [2] “cuSPARSE,” Available online at <https://docs.nvidia.com/cuda/cusparse/index.html>, accessed: 2020-04.
- [3] “Memory-augmented neural networks. facebook open-source projects,” Available online at <https://github.com/facebook/MemNN>, accessed: 2020-04.
- [4] “Nvidia nvlake and nvswitch,” Available online at <https://www.nvidia.com/en-us/data-center/nvlink/>, accessed: 2020-04.
- [5] “Onnx runtime: Microsoft open sources breakthrough optimizations for transformer inference on gpu and cpu,” Available online at <https://github.com/microsoft/onnxruntime>, accessed: 2020-08.
- [6] “Real-time natural language understanding with bert using tensorrt,” Available online at <https://developer.nvidia.com/blog/nlu-with-tensorrt-bert/>, accessed: 2020-08.
- [7] “The size of wikipedia,” Available online at [https://en.wikipedia.org/wiki/Wikipedia:Size\\_in\\_volumes](https://en.wikipedia.org/wiki/Wikipedia:Size_in_volumes), accessed: 2018-12.
- [8] “Tensor2tensor. google open-source projects,” Available online at <https://github.com/tensorflow/tensor2tensor>, accessed: 2020-04.

- [9] *Intel 64 and IA-32 Architectures Software Developer’s Manual Vol. 3A*. Intel, 2018, ch. Memory Cache Control.
- [10] V. Aklaghi, A. Yazdanbakhsh, K. Samadi, H. Esmailzadeh, and R. Gupta, “SnaPEA: Predictive early activation for reducing computation in deep convolutional neural networks,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 662–673.
- [11] R. Akram and A. Muzahid, “Approximeter: Automatically finding and quantifying code sections for approximation,” in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2017, pp. 116–117.
- [12] R. Al-Rfou, D. Choe, N. Constant, M. Guo, and L. Jones, “Character-level language modeling with deeper self-attention,” 2018.
- [13] M. S. B. Altaf and D. A. Wood, “LogCA: A high-level performance model for hardware accelerators,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA ’17. New York, NY, USA: ACM, 2017, pp. 375–388. [Online]. Available: <http://doi.acm.org/10.1145/3079856.3080216>
- [14] G. Ascia, V. Catania, and M. Palesi, “A ga-based design space exploration framework for parameterized system-on-a-chip platforms,” *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 4, pp. 329–346, 2004.
- [15] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, March 1994.
- [16] R. Bertran, P. Bose, D. Brooks, J. Burns, A. Buyuktosunoglu, N. Chandramoorthy, E. Cheng, M. Cochet, S. Eldridge, D. Friedman, H. Jacobson, R. Joshi, S. Mitra, R. Montoye, A. Paidimarri, P. Parida, K. Skadron, M. Stan, K. Swaminathan, A. Vega, S. Venkataramani, C. Vezyrtzis, G.-Y. Wei, J.-D. Wellman,

- and M. Ziegler, “Very low voltage (VLV) design,” in *2017 IEEE International Conference on Computer Design (ICCD)*, Nov 2017, pp. 601–604.
- [17] A. Bordes, N. Usunier, S. Chopra, and J. Weston, “Large-scale simple question answering with memory networks,” *arXiv preprint arXiv:1506.02075*, 2015.
- [18] A. Bordes and J. Weston, “Learning end-to-end goal-oriented dialog,” in *7th International Conference on Learning Representations*, ser. ICLR ’17, 2017.
- [19] B. Boston, A. Sampson, D. Grossman, and L. Ceze, “Probability type inference for flexible approximate programming,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 470–487. [Online]. Available: <http://doi.acm.org/10.1145/2814270.2814301>
- [20] R. Cai, A. Ren, N. Liu, C. Ding, L. Wang, X. Qian, M. Pedram, and Y. Wang, “Vibnn: Hardware acceleration of bayesian neural networks,” *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 476–488, 2018.
- [21] H. Calborean and L. Vințan, “An automatic design space exploration framework for multicore architecture optimizations,” in *9th RoEduNet IEEE International Conference*. IEEE, 2010, pp. 202–207.
- [22] A. X. M. Chang and E. Culurciello, “Hardware accelerators for recurrent neural networks on fpga,” in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2017, pp. 1–4.
- [23] G. Charitopoulos, C. Vatsolakis, G. Chrysos, and D. N. Pnevmatikatos, “A decoupled access-execute architecture for reconfigurable accelerators,” in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, ser. CF ’18. New York, NY, USA: ACM, 2018, pp. 244–247. [Online]. Available: <http://doi.acm.org/10.1145/3203217.3203267>

- [24] T. Chen and G. E. Suh, “Efficient data supply for hardware accelerators with prefetching and access/execute decoupling,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.
- [25] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, 2014.
- [26] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, “Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 27–39, 2016.
- [27] T.-W. Chin, C. Zhang, and D. Marculescu, “Layer-compensated pruning for resource-constrained convolutional neural networks,” *arXiv preprint arXiv:1810.00518*, 2018.
- [28] A. Conneau and G. Lample, “Cross-lingual language model pretraining,” in *Advances in Neural Information Processing Systems*, 2019, pp. 7057–7067.
- [29] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. Le, and R. Salakhutdinov, “Transformer-XL: Attentive language models beyond a fixed-length context,” in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 2978–2988. [Online]. Available: <https://www.aclweb.org/anthology/P19-1285>
- [30] M. Davies, “The Corpus of Contemporary American English (COCA): 560 million words, 1990-present,” Available online at <https://corpus.byu.edu/coca/>, 2008-.

- [31] C. De Sa, M. Feldman, C. Ré, and K. Olukotun, “Understanding and optimizing asynchronous low-precision stochastic gradient descent,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA ’17. New York, NY, USA: ACM, 2017, pp. 561–574. [Online]. Available: <http://doi.acm.org/10.1145/3079856.3080248>
- [32] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.
- [33] R. Ding, Z. Liu, R. Shi, D. Marculescu, and R. S. Blanton, “LightNN: Filling the gap between conventional deep neural networks and binarized networks,” in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, ser. GLSVLSI ’17. New York, NY, USA: ACM, 2017, pp. 35–40. [Online]. Available: <http://doi.acm.org/10.1145/3060403.3060465>
- [34] J. Dodge, A. Gane, X. Zhang, A. Bordes, S. Chopra, A. H. Miller, A. Szlam, and J. Weston, “Evaluating prerequisite qualities for learning end-to-end dialog systems,” in *6th International Conference on Learning Representations*, ser. ICLR ’16, 2016.
- [35] M. Donato, B. Reagen, L. Pentecost, U. Gupta, D. Brooks, and G.-Y. Wei, “On-chip deep neural network storage with multi-level eNVM,” in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC ’18. New York, NY, USA: ACM, 2018, pp. 169:1–169:6. [Online]. Available: <http://doi.acm.org/10.1145/3195970.3196083>
- [36] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, “Neural Cache: Bit-serial in-cache acceleration of deep neural networks,” in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA ’18. Piscataway, NJ, USA: IEEE Press,

2018, pp. 383–396. [Online]. Available: <https://doi.org/10.1109/ISCA.2018.00040>

- [37] S. Eranian, “Perfmon2: a flexible performance monitoring interface for linux,” in *Proc. of the 2006 Ottawa Linux Symposium*. Citeseer, 2006, pp. 269–288.
- [38] B. Feinberg, U. K. R. Vengalam, N. Whitehair, S. Wang, and E. Ipek, “Enabling scientific computing on memristive accelerators,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 367–382.
- [39] B. Feinberg, S. Wang, and E. Ipek, “Making memristive neural network accelerators reliable,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 52–65. [Online]. Available: [doi.ieeecomputersociety.org/10.1109/HPCA.2018.00015](https://doi.ieeecomputersociety.org/10.1109/HPCA.2018.00015)
- [40] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, “A configurable cloud-scale dnn processor for real-time ai,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 1–14.
- [41] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. Moura, “SPIRAL: Extreme performance portability,” *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, Nov 2018.
- [42] D. Gadioli, E. Vitali, G. Palermo, and C. Silvano, “mARGOt: a dynamic auto-tuning framework for self-aware approximate computing,” *IEEE Transactions on Computers*, pp. 1–1, 2018.

- [43] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, “Deltarnn: A power-efficient recurrent neural network accelerator,” in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 21–30.
- [44] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, “Tetris: Scalable and efficient neural network acceleration with 3d memory,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 751–764.
- [45] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, “Tangram: Optimized coarse-grained dataflow for scalable nn accelerators,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 807–820.
- [46] R. Garibotti, B. Reagen, Y. S. Shao, G.-Y. Wei, and D. Brooks, “Assisting high-level synthesis improve SpMV benchmark through dynamic dependence analysis,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 10, pp. 1440–1444, Oct 2018.
- [47] G. Gobieski, N. Beckmann, and B. Lucia, “Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems,” *ArXiv e-prints*, p. arXiv:1810.07751, Sep. 2018.
- [48] M. A. Gordon, K. Duh, and N. Andrews, “Compressing bert: Studying the effects of weight pruning on transfer learning,” *arXiv preprint arXiv:2002.08307*, 2020.
- [49] K. Goto and R. A. v. d. Geijn, “Anatomy of high-performance matrix multiplication,” *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 12:1–12:25, May 2008. [Online]. Available: <http://doi.acm.org/10.1145/1356052.1356053>

- [50] Y. Guan, Z. Yuan, G. Sun, and J. Cong, “Fpga-based accelerator for long short-term memory recurrent neural networks,” in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 629–634.
- [51] P. Gysel, M. Motamedi, and S. Ghiasi, “Hardware-oriented approximation of convolutional neural networks,” *arXiv preprint arXiv:1604.03168*, 2016.
- [52] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J. Park, S.-H. Lee, K. M. Park, J. W. Lee, and D.-K. Jeong, “A3: Accelerating attention mechanisms in neural networks with approximation,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.
- [53] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “EIE: Efficient inference engine on compressed deep neural network,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 243–254.
- [54] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher, “UCNN: Exploiting computational reuse in deep neural networks via weight repetition,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 674–687.
- [55] F. Hill, A. Bordes, S. Chopra, and J. Weston, “The goldilocks principle: Reading children’s books with explicit memory representations,” in *6th International Conference on Learning Representations*, ser. ICLR ’16, 2016.
- [56] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, “DeftNN: Addressing bottlenecks for DNN execution on GPUs via synapse vector elimination and near-compute data fission,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 ’17. New York, NY, USA: ACM, 2017, pp. 786–799. [Online]. Available: <http://doi.acm.org/10.1145/3123939.3123970>



- [57] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: <http://dx.doi.org/10.1162/neco.1997.9.8.1735>
- [58] P.-A. Hsiung, C.-S. Lin, and C.-F. Liao, “Perfecto: A systemc-based design-space exploration framework for dynamically reconfigurable architectures,” *ACM Transactions on Reconfigurable Technology and Systems (TRETTS)*, vol. 1, no. 3, pp. 1–30, 2008.
- [59] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, “Gpipe: Efficient training of giant neural networks using pipeline parallelism,” in *Advances in Neural Information Processing Systems*, 2019, pp. 103–112.
- [60] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, “SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size,” *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: <http://arxiv.org/abs/1602.07360>
- [61] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, “Gist: Efficient data encoding for deep neural network training,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 776–789.
- [62] H. Jang, J. Kim, J.-E. Jo, J. Lee, and J. Kim, “Mnnfast: a fast and scalable system architecture for memory-augmented neural networks,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 250–263.
- [63] H. Ji, L. Song, L. Jiang, H. H. Li, and Y. Chen, “ReCom: An efficient resistive accelerator for compressed deep neural networks,” in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 237–240.

- [64] Y. Ji, Y. Zhang, W. Chen, and Y. Xie, “Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’18. New York, NY, USA: ACM, 2018, pp. 448–460. [Online]. Available: <http://doi.acm.org/10.1145/3173162.3173205>
- [65] Z. Jia, M. Zaharia, and A. Aiken, “Beyond data and model parallelism for deep neural networks,” 2018.
- [66] M. Joshi, D. Chen, Y. Liu, D. S. Weld, L. Zettlemoyer, and O. Levy, “Spanbert: Improving pre-training by representing and predicting spans,” *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 64–77, 2020.
- [67] K. Keutzer, K. Ravindran, N. Satish, and Y. Jin, “An automated exploration framework for fpga-based soft multiprocessor systems,” in *2005 Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS’05)*. IEEE, 2005, pp. 273–278.
- [68] N. Kitaev, Ł. Kaiser, and A. Levskaya, “Reformer: The efficient transformer,” *arXiv preprint arXiv:2001.04451*, 2020.
- [69] V. Krishnan and S. Katkooi, “A genetic algorithm for the design space exploration of datapaths during high-level synthesis,” *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 3, pp. 213–229, 2006.
- [70] J. Kung, Y. Long, D. Kim, and S. Mukhopadhyay, “A programmable hardware accelerator for simulating dynamical systems,” in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 403–415.
- [71] H. Kwon, A. Samajdar, and T. Krishna, “MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects,”

- in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 461–475. [Online]. Available: <http://doi.acm.org/10.1145/3173162.3173176>
- [72] G. Lample, A. Sablayrolles, M. Ranzato, L. Denoyer, and H. Jégou, “Large memory layers with product keys,” in *Advances in Neural Information Processing Systems*, 2019, pp. 8546–8557.
- [73] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, “Albert: A lite bert for self-supervised learning of language representations,” in *International Conference on Learning Representations*, 2020. [Online]. Available: <https://openreview.net/forum?id=H1eA7AEtvS>
- [74] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, “Unpu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision,” in *2018 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2018, pp. 218–220.
- [75] J. H. Lee and H. Kim, “StaleLearn: Learning acceleration with asynchronous synchronization between model replicas on PIM,” *IEEE Transactions on Computers*, vol. 67, no. 6, pp. 861–873, June 2018.
- [76] J. Li, A. H. Miller, S. Chopra, M. Ranzato, and J. Weston, “Dialogue learning with human-in-the-loop,” *CoRR*, vol. abs/1611.09823, 2016. [Online]. Available: <http://arxiv.org/abs/1611.09823>
- [77] M.-Y. Lin, H.-Y. Cheng, W.-T. Lin, T.-H. Yang, I.-C. Tseng, C.-L. Yang, H.-W. Hu, H.-S. Chang, H.-P. Li, and M.-F. Chang, “DL-RSIM: A simulation framework to enable reliable ReRAM-based accelerators for deep learning,” in *Proceedings of the International Conference on Computer-Aided Design*,

- ser. ICCAD '18. New York, NY, USA: ACM, 2018, pp. 31:1–31:8. [Online]. Available: <http://doi.acm.org/10.1145/3240765.3240800>
- [78] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, “The architectural implications of autonomous driving: Constraints and acceleration,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 751–766. [Online]. Available: <http://doi.acm.org/10.1145/3173162.3173191>
- [79] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019.
- [80] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, “Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 553–564.
- [81] Y. Ma, N. Suda, Y. Cao, J.-s. Seo, and S. Vrudhula, “Scalable and modularized RTL compilation of convolutional neural networks onto FPGA,” in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–8.
- [82] D. Marculescu, D. Stamoulis, and E. Cai, “Hardware-aware machine learning: Modeling and optimization,” in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '18. New York, NY, USA: ACM, 2018, pp. 137:1–137:8. [Online]. Available: <http://doi.acm.org/10.1145/3240765.3243479>
- [83] G. Mariani, P. Avasare, G. Vanmeerbeeck, C. Ykman-Couvreur, G. Palermo, C. Silvano, and V. Zaccaria, “An industrial design space exploration framework

- for supporting run-time resource management on multi-core systems,” in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 2010, pp. 196–201.
- [84] J. McCarley, R. Chakravarti, and A. Sil, “Structured pruning of a bert-based question answering model,” *arXiv preprint arXiv:1910.06360*, 2019.
- [85] J. S. Miguel, M. Badr, and N. E. Jerger, “Load value approximation,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 127–139. [Online]. Available: <http://dx.doi.org/10.1109/MICRO.2014.22>
- [86] T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur, “Recurrent neural network based language model,” vol. 2, 01 2010, pp. 1045–1048.
- [87] A. Miller, A. Fisch, J. Dodge, A.-H. Karimi, A. Bordes, and J. Weston, “Key-value memory networks for directly reading documents,” in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2016, pp. 1400–1409. [Online]. Available: <http://aclweb.org/anthology/D16-1147>
- [88] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmailzadeh, L. Ceze, and M. Oskin, “SNNAP: Approximate computing on programmable SoCs via neural acceleration,” in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 603–614.
- [89] M. J. Nielsen and Z. S. Hussain, “Unified memory computer architecture with dynamic graphics memory allocation,” Aug. 15 2000, uS Patent 6,104,417.
- [90] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, “Stream-dataflow acceleration,” in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 416–429.

- [91] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, “Can FPGAs beat GPUs in accelerating next-generation deep neural networks?” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’17. New York, NY, USA: ACM, 2017, pp. 5–14. [Online]. Available: <http://doi.acm.org/10.1145/3020078.3021740>
- [92] M. Palesi and T. Givargis, “Multi-objective design space exploration using genetic algorithms,” in *Proceedings of the tenth international symposium on Hardware/software codesign*, 2002, pp. 67–72.
- [93] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, “SCNN: An accelerator for compressed-sparse convolutional neural networks,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA ’17. New York, NY, USA: ACM, 2017, pp. 27–40. [Online]. Available: <http://doi.acm.org/10.1145/3079856.3080254>
- [94] E. Park, D. Kim, and S. Yoo, “Energy-efficient neural network accelerator based on outlier-aware low-precision computation,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 688–698.
- [95] A. Perais and A. Sez nec, “Practical data value speculation for future high-end processors,” in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 428–439.
- [96] A. Polino, R. Pascanu, and D. Alistarh, “Model compression via distillation and quantization,” *CoRR*, vol. abs/1802.05668, 2018. [Online]. Available: <http://arxiv.org/abs/1802.05668>

- [97] M. Putic, A. Buyuktosunoglu, S. Venkataramani, P. Bose, S. Eldridge, and M. Stan, “DyHard-DNN: Even more DNN acceleration with dynamic hardware reconfiguration,” in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, June 2018, pp. 1–6.
- [98] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019.
- [99] A. Rahman, S. Oh, J. Lee, and K. Choi, “Design space exploration of fpga accelerators for convolutional neural networks,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1147–1152.
- [100] P. Rajpurkar, R. Jia, and P. Liang, “Know what you don’t know: Unanswerable questions for squad,” *CoRR*, vol. abs/1806.03822, 2018. [Online]. Available: <http://arxiv.org/abs/1806.03822>
- [101] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, “Squad: 100, 000+ questions for machine comprehension of text,” *CoRR*, vol. abs/1606.05250, 2016. [Online]. Available: <http://arxiv.org/abs/1606.05250>
- [102] B. Reagen, U. Gupta, R. Adolf, M. M. Mitzenmacher, A. M. Rush, G.-Y. Wei, and D. Brooks, “Weightless: Lossy weight encoding for deep neural network compression,” *arXiv preprint arXiv:1711.04686*, 2017.
- [103] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, “Ares: A framework for quantifying the resilience of deep neural networks,” in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC ’18. New York, NY, USA: ACM, 2018, pp. 17:1–17:6. [Online]. Available: <http://doi.acm.org/10.1145/3195970.3195997>

- [104] B. Reagen, J. M. Hernández-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G.-Y. Wei, and D. Brooks, “A case for efficient accelerator design space exploration via bayesian optimization,” in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, July 2017, pp. 1–6.
- [105] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, “Minerva: Enabling low-power, highly-accurate deep neural network accelerators,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 267–278. [Online]. Available: <https://doi.org/10.1109/ISCA.2016.32>
- [106] T. J. Repetti, J. a. P. Cerqueira, M. A. Kim, and M. Seok, “Pipelining a triggered processing element,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 96–108. [Online]. Available: <http://doi.acm.org/10.1145/3123939.3124551>
- [107] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, “vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design,” in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 18:1–18:13. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3195638.3195660>
- [108] K. Rosvall and I. Sander, “A constraint-based design space exploration framework for real-time applications on mpsoCs,” in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–6.
- [109] Z. S. Harris, “Distributional structure,” *Word*, vol. 10, pp. 146–162, 08 1954.



- [110] A. Sampson, J. Bornholt, and L. Ceze, “Hardware-Software Co-Design: Not Just a Cliché,” in *1st Summit on Advances in Programming Languages (SNAPL 2015)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, Eds., vol. 32. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 262–273. [Online]. Available: <http://drops.dagstuhl.de/opus/volltexte/2015/5030>
- [111] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, “Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter,” *arXiv preprint arXiv:1910.01108*, 2019.
- [112] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, “Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.
- [113] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, “The aladdin approach to accelerator design and modeling,” *IEEE Micro*, vol. 35, no. 3, pp. 58–70, May 2015.
- [114] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, “From high-level deep neural models to fpgas,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.
- [115] Y. Shen, M. Ferdman, and P. Milder, “Maximizing CNN accelerator efficiency through resource partitioning,” in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 535–547.

- [116] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, “Megatron-lm: Training multi-billion parameter language models using gpu model parallelism,” *arXiv preprint arXiv:1909.08053*, 2019.
- [117] L. Song, X. Qian, H. Li, and Y. Chen, “Pipelayer: A pipelined reram-based accelerator for deep learning,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 541–552.
- [118] M. Song, J. Zhang, H. Chen, and T. Li, “Towards efficient microarchitectural design for accelerating unsupervised gan-based deep learning,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 66–77.
- [119] M. Song, J. Zhao, Y. Hu, J. Zhang, and T. Li, “Prediction based execution on deep neural networks,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 752–763.
- [120] M. Song, K. Zhong, J. Zhang, Y. Hu, D. Liu, W. Zhang, J. Wang, and T. Li, “In-Situ AI: Towards autonomous and incremental deep learning for IoT systems,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 92–103. [Online]. Available: [doi.ieeecomputersociety.org/10.1109/HPCA.2018.00018](https://doi.ieeecomputersociety.org/10.1109/HPCA.2018.00018)
- [121] P. Srivastava, M. Kang, S. K. Gonugondla, S. Lim, J. Choi, V. Adve, N. S. Kim, and N. Shanbhag, “PROMISE: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms,” in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA ’18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 43–56. [Online]. Available: <https://doi.org/10.1109/ISCA.2018.00015>
- [122] D. Stamoulis, E. Cai, D.-C. Juan, and D. Marculescu, “HyperPower: Power- and memory-constrained hyper-parameter optimization for neural networks,” in

2018 Design, Automation Test in Europe Conference Exhibition (DATE), March 2018, pp. 19–24.

- [123] D. Stamoulis, T.-W. R. Chin, A. K. Prakash, H. Fang, S. Sajja, M. Bogner, and D. Marculescu, “Designing adaptive neural networks for energy-constrained image classification,” in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '18. New York, NY, USA: ACM, 2018, pp. 23:1–23:8. [Online]. Available: <http://doi.acm.org/10.1145/3240765.3240796>
- [124] J. R. Stevens, A. Ranjan, D. Das, B. Kaul, and A. Raghunathan, “Manna: An accelerator for memory-augmented neural networks,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 794–806.
- [125] S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus, “End-to-end memory networks,” *CoRR*, vol. abs/1503.08895, 2015. [Online]. Available: <http://arxiv.org/abs/1503.08895>
- [126] H. Tang, C. Yu, C. Renggli, S. Kassing, A. Singla, D. Alistarh, J. Liu, and C. Zhang, “Distributed learning over unreliable networks,” *arXiv preprint arXiv:1810.07766*, 2018.
- [127] O. Temam, “A defect-tolerant accelerator for emerging high-performance applications,” in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 356–367. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2337159.2337200>
- [128] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

- [129] R. Venkatagiri, K. Swaminathan, C.-C. Lin, L. Wang, A. Buyuktosunoglu, P. Bose, and S. Adve, “Impact of software approximations on the resiliency of a video summarization system,” in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018, pp. 598–609.
- [130] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, “Glue: A multi-task benchmark and analysis platform for natural language understanding,” *arXiv preprint arXiv:1804.07461*, 2018.
- [131] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, “Intel math kernel library,” in *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.
- [132] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, “AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’13. New York, NY, USA: ACM, 2013, pp. 25:1–25:12. [Online]. Available: <http://doi.acm.org/10.1145/2503210.2503219>
- [133] S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang, “C-lstm: Enabling efficient lstm using structured compression techniques on fpgas,” in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 11–20.
- [134] J. Weston, “Dialog-based language learning,” *CoRR*, vol. abs/1604.06045, 2016. [Online]. Available: <http://arxiv.org/abs/1604.06045>
- [135] J. Weston, A. Bordes, S. Chopra, and T. Mikolov, “Towards AI-complete question answering: A set of prerequisite toy tasks,” *CoRR*, vol. abs/1502.05698, 2015. [Online]. Available: <http://arxiv.org/abs/1502.05698>

- [136] J. Weston, S. Chopra, and A. Bordes, “Memory networks,” *CoRR*, vol. abs/1410.3916, 2014. [Online]. Available: <http://arxiv.org/abs/1410.3916>
- [137] V. V. Williams, “Breaking the coppersmith-winograd barrier,” 2011.
- [138] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.
- [139] C. Wu, W. Wen, T. Afzal, Y. Zhang, Y. Chen, and H. Li, “A compact DNN: Approaching GoogLeNet-level accuracy of classification and domain adaptation,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017, pp. 761–770.
- [140] Z. Xianyi, W. Qian, and Z. Yunquan, “Model-driven level 3 BLAS performance optimization on loongson 3A processor,” in *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*, ser. ICPADS ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 684–691. [Online]. Available: <http://dx.doi.org/10.1109/ICPADS.2012.97>
- [141] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, “Xlnet: Generalized autoregressive pretraining for language understanding,” in *Advances in neural information processing systems*, 2019, pp. 5754–5764.
- [142] A. Yazdanbakhsh, H. Falahati, P. J. Wolfe, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, “GANAX: A unified mimd-simd acceleration for generative adversarial networks,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 650–661.
- [143] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, “RFVP: Rollback-free value prediction with safe-to-approximate loads,” vol. 12, no. 4. New York, NY, USA: ACM, Jan. 2016, pp. 62:1–62:26. [Online]. Available: <http://doi.acm.org/10.1145/2836168>

- [144] R. Yazdani, J.-M. Arnau, and A. González, “UNFOLD: A memory-efficient speech recognizer using on-the-fly WFST composition,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 69–81. [Online]. Available: <http://doi.acm.org/10.1145/3123939.3124542>
- [145] R. Yazdani, M. Riera, J.-M. Arnau, and A. González, “The dark side of DNN pruning,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 790–801.
- [146] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, “Scalpel: Customizing DNN pruning to the underlying hardware parallelism,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 548–560. [Online]. Available: <http://doi.acm.org/10.1145/3079856.3080215>
- [147] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing fpga-based accelerator design for deep convolutional neural networks,” in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170.
- [148] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, “Cambricon-x: An accelerator for sparse neural networks,” in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 20:1–20:12. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3195638.3195662>
- [149] Z. Zhang, X. Han, Z. Liu, X. Jiang, M. Sun, and Q. Liu, “ERNIE: Enhanced language representation with informative entities,” in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence,

Italy: Association for Computational Linguistics, Jul. 2019, pp. 1441–1451.  
[Online]. Available: <https://www.aclweb.org/anthology/P19-1139>

- [150] G. Zhong, A. Prakash, S. Wang, Y. Liang, T. Mitra, and S. Niar, “Design space exploration of fpga-based accelerators with multi-level parallelism,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1141–1146.
- [151] Y. Zhu, V. J. Reddi, R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks, “Cognitive computing safety: The new horizon for reliability / the design and evolution of deep learning workloads,” *IEEE Micro*, vol. 37, no. 1, pp. 15–21, Jan.-Feb. 2017. [Online]. Available: [doi.ieeecomputersociety.org/10.1109/MM.2017.2](https://doi.ieeecomputersociety.org/10.1109/MM.2017.2)

# 초 록

자연어 처리의 중요성이 대두됨에 따라 여러 기업 및 연구진들은 다양하고 복잡한 종류의 자연어 처리 모델들을 제시하고 있다. 즉 자연어 처리 모델들은 형태가 복잡해지고, 로규모가 커지며, 종류가 다양해지는 양상을 보여준다. 본 학위논문은 이러한 자연어 처리 모델의 복잡성, 확장성, 다양성을 해결하기 위해 여러 핵심 아이디어를 제시하였다. 각각의 핵심 아이디어들은 다음과 같다. (1) 다양한 종류의 자연어 처리 모델의 성능 오버헤드 분포도를 알아내기 위한 정적/동적 분석을 수행한다. (2) 성능 분석을 통해 알아낸 주된 성능 병목 요소들의 메모리 사용을 최적화하기 위한 전체론적 모델 병렬화 기술을 제시한다. (3) 여러 연산들의 연산량을 감소하는 기술과 연산량 감소로 인한 skewness 문제를 해결하기 위한 dynamic scheduler 기술을 제시한다. (4) 현 자연어 처리 모델의 성능 다양성을 해결하기 위해 각 모델에 최적화된 디자인을 제시하는 기술을 제시한다. 이러한 핵심 기술들은 여러 종류의 하드웨어 가속기 (예: CPU, GPU, FPGA, ASIC) 에도 범용적으로 사용될 수 있기 때문에 매우 효과적이므로, 제시된 기술들은 자연어 처리 모델을 위한 컴퓨터 시스템 설계 분야에 광범위하게 적용될 수 있다. 본 논문에서는 해당 기술들을 적용하여 CPU, GPU, FPGA 각각의 환경에서, 제시된 기술들이 모두 유의미한 성능향상을 달성함을 보여준다. 처처

**주요어:** 인공지능 가속기 설계/디자인, 자연어 처리 모델, 머신러닝, 하드웨어 아키텍처

**학번:** 2017-36250