2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT) | 978-1-6654-4278-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/PACT52795.2021.00013

NLP-Fast: A Fast, Scalable, and Flexible System to Accelerate Large-Scale Heterogeneous NLP Models

Joonsung Kim, Suyeon Hur, Eunbok Lee, Seungho Lee, and Jangwoo Kim*
Department of Electrical and Computer Engineering, Seoul National University
Emails: {joonsung90, suyeon339, eunbok, seungho.lee, jangwoo}@snu.ac.kr

Abstract—**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.**

In this paper, we propose *NLP-Fast*, a novel system solution to accelerate a wide spectrum of large-scale NLP models. NLP-Fast mainly consists of two parts: (1) *NLP-Perf*: an in-depth performance analysis tool to identify critical operations in emerging NLP models and (2) *NLP-Opt*: three end-to-end optimization techniques to accelerate the identified performance-critical operations on various hardware platforms (e.g., CPU, GPU, FPGA). In this way, NLP-Fast can accelerate various types of NLP models on different hardware platforms by identifying their critical operations through NLP-Perf and applying the NLP-Opt's holistic optimizations. We evaluate NLP-Fast on CPU, GPU, and FPGA, and the overall throughputs are increased by up to 2.92×, 1.59×, and 4.47× over each platform's baseline.

We release NLP-Fast to the community so that users are easily able to conduct the NLP-Fast's analysis and apply NLP-Fast's optimizations for their own NLP applications.

Index Terms—**Architecture; Computation/Dataflow Optimization; Natural Language Processing (NLP); Parallel Algorithm**

I. Introduction

Recent research breakthroughs in natural language processing (NLP) are strongly driving major IT companies to introduce various high-impact NLP models (e.g., Memory networks [1], Transformer [2], BERT [3]). With these emerging models available, researchers are actively developing various NLP-inspired services by customizing the models for their own purposes. This trend will obviously continue as the industry aims to provide more sophisticated NLP services.

As such NLP models are actively adopted, there is also a strong demand for a scalable and flexible system architecture for emerging NLP models. However, existing work cannot provide enough scalability and flexibility due to their limited focus on specific operations in NLP models. For example, some studies (e.g., MnnFast [4], A3 [5]) aim to accelerate the *attention mechanism* only, but the attention mechanism is not a major performance bottleneck in recent NLP models (e.g., 11.8% in Transformer, 8.5% in BERT). In this case, they can provide marginal end-to-end performance improvements, 13.4% and 9.3% in Transformer and BERT, respectively.

*Corresponding author.

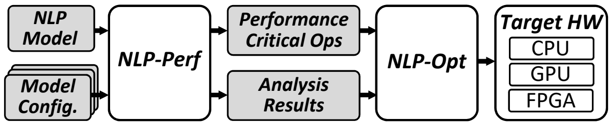


Fig. 1. The high-level workflow of NLP-Fast.

This limited performance improvement of existing proposals is mainly due to the following reasons. First, they do not consider a wide spectrum of performance-critical operations used in recent NLP models. As the performance bottlenecks in a large-scale NLP model span over many operations, they must identify these critical operations. Second, the existing optimizations are not end-to-end solutions to cover all types of performance-critical operations.

In this paper, we propose NLP-Fast, a *fast*, *scalable*, and *flexible* system solution for emerging NLP models. Fig. 1 shows the high-level workflow of NLP-Fast composed of two parts: *NLP-Perf* and *NLP-Opt*.

First, we build NLP-Perf, the performance analysis tool for emerging NLP models, to analyze and profile the NLP models. For a given NLP model and its configurations, NLP-Perf provides the analysis results (e.g., latency breakdown, cache statistics) through extensive profiling of all combinations of the NLP model and configurations. Based on the analysis results, NLP-Perf identifies the performance-critical operations. In this work, we conduct a thorough analysis on three representative NLP models: *Memory networks*, *Transformer*, and *BERT*.

Next, to remove the identified performance bottlenecks, we propose NLP-Opt consisting of 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.

For the hardware-specific optimization, we propose *model/config adaptive hardware reconfiguration*, the method to adaptively reconfigure the architecture for further performance improvements. In this work, we use FPGA as a reconfigurable platform. 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 optimizations. As we choose the standard ML libraries used in TensorFlow as the baseline, NLP-Fast is also be applicable to other various AI/ML frameworks (e.g., PyTorch, MxNet), as long as NLP-Fast's three optimization techniques are applied.

The evaluation results show that CPU-based NLP-Fast achieves up to 2.92× speedup (2.00× on average) and shows scalable performance using more cores. GPU-based NLP-Fast achieves up to 1.59× speedup (1.44× on average) in the single-GPU environment and shows scalable performance using more GPUs. FPGA-based NLP-Fast achieves up to 2.89× speedup (2.59× on average), and up to 4.47× speed up with its adaptive hardware reconfiguration enabled.

We make NLP-Fast as an open-source project: *NLP-Fast toolkit*). To show usefulness and effectiveness, we illustrate how to use the NLP-Fast toolkit for performance analysis (NLP-Perf) and optimizations (NLP-Opt) in Section IV.

We make the following contributions:

- **Critical problem identification.** We identify critical issues in supporting various commodity NLP models and their model/config-dependent performance bottlenecks.
- **High model coverage.** We can execute a wide spectrum of commodity NLP models, and NLP-Fast can easily extend its coverage by analyzing the model, adding the required operations, and applying the holistic optimizations as done in this work.
- **High performance with novel solutions.** The proposed scheme can achieve significant performance improvements by applying three holistic optimizations (i.e., *holistic model partitioning*, *cross-operation zero skipping*, *adaptive hardware reconfiguration*).
- **Easy implementation on various platforms.** The proposed scheme can be easily applied on any platforms: CPU, GPU, and FPGA.
- **Software release.** We release our NLP-Fast toolkit to the community for future researchers (our project github link: https://github.com/SNU-HPCS/NLP-Fast).

The rest of the paper is organized as follows. Section II explains the characteristics of emerging NLP models and motivates our work with its key design goals. Section III describes how NLP-Fast solves the performance problems to achieve fast and scalable performance. Section IV and Section V show our NLP-Fast toolkit and NLP-Fast's implementation details, respectively. We provide evaluation results for NLP-Fast on Section VI. Finally, Section VII and Section VIII provide related work and conclusion, respectively.

TABLE I
THREE 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 [1], [9]–[12]	✓			
Transformer [2], [13]–[15]		✓[†]	✓	✓
BERT [3], [16]–[19]			✓	✓

[†] Encoder-decoder attention layer in the transformer decoder.

II. Background & Motivation

A. Deep Learning for NLP

NLP models are widely used from simple dialog comprehension to large-scale question & answering system using a large-scale dataset (e.g., Wikipedia) [6]–[9].

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. Other emerging NLP models are slight variations of these three models. Table I shows the category of recent NLP models and their key computational components: *attention mechanism*, *multi-head attention*, *multi-head self-attention*, and *feed-forward network*.

First, *Memory networks* consists of multiple consecutive attention mechanisms followed by a simple fully-connected layer with softmax. Second, *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). Lastly, *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 Fig. 2, we illustrate how each computational component operates in more detail. Fig. 2a shows the *attention mechanism*. There are three input matrices in the attention mechanism: a query matrix ($Q \in \mathbb{R}^{nq \times d}$), a key matrix ($K \in \mathbb{R}^{ns \times d}$), and a value matrix ($V \in \mathbb{R}^{ns \times d}$). 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 three operations (i.e., *dot product*, *softmax*, *weighted sum*) to calculate an attention result for each query. Equation (1) shows how to compute the attention result (A).

$$A = Attention(Q,K,V) = Softmax(Q \times K^T) \times V \tag{1}$$

The attention mechanism first computes a score matrix (S) by computing dot products of each query with all keys (*dot product*). Next, it calculates a probability matrix (P) by applying a softmax function ($Softmax(x_i) = e^{x_i}/\sum_j e^{x_j}$) to the score matrix (*softmax*). Here, each row in the probability matrix represents the correlation between a corresponding query and all keys. Then, the attention mechanism computes a sum of values (V) weighted by these probabilities to get the attention result (A) (*weighted sum*).

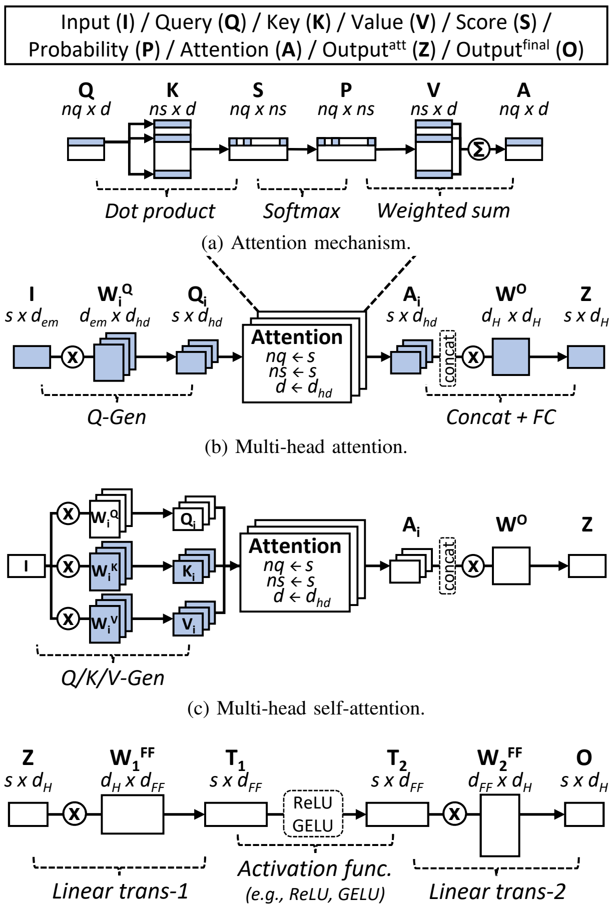


Fig. 2. Computational steps of key components used in recent NLP models.

Fig. 2b and Fig. 2c illustrate how the *multi-head attention* and *multi-head self-attention* work, respectively. 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 multi-head attention output (Z). Equation (2) shows the details.

$$Z = MultiHead(I,K,V) = Concat(\{head_i\}_{i=1}^{h}) \times W^O$$
$$head_i = Attention(I \times W_i^Q, K_i, V_i) \quad (2)$$
$$Z = MultiHeadSelf(I) = MultiHead(I, I \times W^K, I \times W^V)$$

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 ($I \times W_i^Q$).

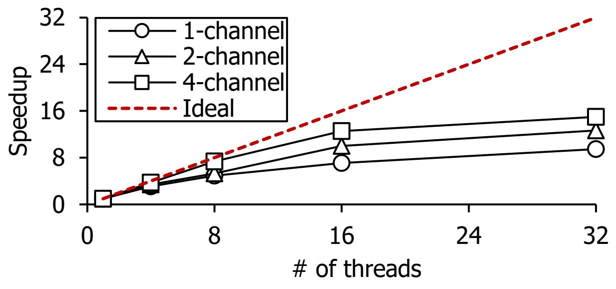


Fig. 3. Limited scalability due to memory bandwidth. The speedup of each channel configuration is normalized to the corresponding single-thread result.

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*).

In addition to attention components, state-of-the-art NLP models also contain a *feed-forward network*. Fig. 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 (3) shows the details.

$$O = FFN(Z) = ActFunc(Z \times W_1^{FF}) \times W_2^{FF} \quad (3)$$

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*.

B. Limitations of Previous NLP Acceleration Studies

As described in Section II-A, emerging NLP models consist of various combinations of heterogeneous operations with different configurations. Therefore, an ideal NLP acceleration system should optimize a wide spectrum of these operations in NLP models. However, existing studies do not identify performance-critical operations and *only* focus on optimizing specific operations, which results in the limited performance improvement. For example, some studies (e.g., MnnFast [4], A3 [5]) propose acceleration methods of the *attention mechanism* (Fig. 2a) only, but the attention mechanism is not a major performance bottleneck in recent NLP models (e.g., 11.8% in Transformer, 8.5% in BERT). In this case, they can only provide marginal end-to-end performance improvements, 13.4% and 9.3% in Transformer and BERT, respectively. To the best of our knowledge, there is no work aiming to holistically optimize all operations in emerging NLP models.

Therefore, we need an in-depth analysis to figure out performance-critical operations in recent NLP models. To do so, we make NLP-Perf and conduct extensive profiling to identify these performance-critical operations (Section III-A).

C. Performance Problems in NLP Models

Through the NLP-Perf's in-depth analysis, we find out three performance problems in NLP models: *high memory bandwidth*, *heavy computation*, and *huge performance variation*.

1) High Memory Bandwidth: Fig. 3 shows how memory bandwidth affects the scalability of NLP models. We measure the performance of BERT with the default configuration (see Section VI-A). 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 to prevent the computation from becoming a performance bottleneck. We can observe that BERT quickly reaches a performance saturation point as the bandwidth decreases; in other words, BERT's high memory bandwidth requirements prevent the current system from achieving scalable performance.

To overcome the bandwidth problem, we propose *holistic model partitioning* enabling to hide most off-chip DRAM accesses by reducing the working set size (*NLP-Opt #1*). We explain this technique in Section III-B for more details.

2) Heavy Computation: Similar to the high memory bandwidth problem, the computation overhead 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 require a large number of integer multiplications, which is highly compute-intensive.

To overcome the heavy computation problem, we propose *cross-operation zero skipping* that bypasses near-zero computation in attention-related operations (*NLP-Opt #2*).

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 (*NLP-Opt #2*). We explain this optimization in Section III-C for more details.

3) Huge Performance Variation: We find out various models and parameter configurations incur a huge *performance variation* as shown in Fig. 4. For example, one operation can be a major 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 11.5% of the performance we can get from an accelerator optimized for that configuration (Section VI-D).

To overcome the huge performance variation, we propose an *adaptive hardware reconfiguration* enabling an HW accelerator to achieve full potential performance by finding an optimal design point with resource rebalancing (*NLP-Opt #3*). We explain the details in Section III-D.

TABLE II
TIME AND SPACE COMPLEXITY OF EACH KEY OPERATION COMMONLY USED IN EMERGING NLP MODELS.

		Operations	**Time complexity**	**Space complexity**
Multi-head self-Att.†		Q-Gen	$O(s \cdot d_H^2)$	$O((s+d_H) \cdot d_H)$
		K/V-Gen	$O(s \cdot d_H^2)$	$O((s+d_H) \cdot d_H)$
	Attention	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	$O(s \cdot d_H \cdot d_{FF})$	$O((s+d_H) \cdot d_{FF})$

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

D. Design Goals & Key Ideas

To accelerate emerging heterogeneous NLP models, we set design goals as follows.

- **High coverage of NLP models.** It should support diverse NLP models. We extract key basic operations in the models and propose a holistic optimization for these operations.
- **Efficient memory management algorithm.** It should minimize the off-chip 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 *cross-operation zero skipping* to skip operations of near-zero values.
- **Model/config-adaptive system.** It should provide an optimal design for a given model/config. For FPGA, we propose *adaptive hardware reconfiguration* to fully utilize resources.

III. NLP-FAST

A. NLP-Perf: Bottleneck Analysis of NLP Models

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

Table II shows the result of our static analysis. We analyze operations used in the key computational components and calculate their time and space complexity according to various parameters. We observe that NLP models consist of some key operations: *Q/K/V-Gen*, *dot product*, *softmax*, *weighted sum*, *attention FC*, *linear trans-1*, *activation func*, *linear trans-2*. Here, s means the sequence length and d_H and d_{ff} represent the dimensions of hidden vectors and internal vectors used in the attention mechanism and feed-forward network, respectively.

Fig. 4 shows the profiling results of representative NLP models with different parameter configurations. We scale a specific parameter (S: sequence length (s), H: hidden dimension (d_H), F: feed-forward dimension (d_{FF})) from base configurations to profile models with various configurations. We break the total execution time down in different operations

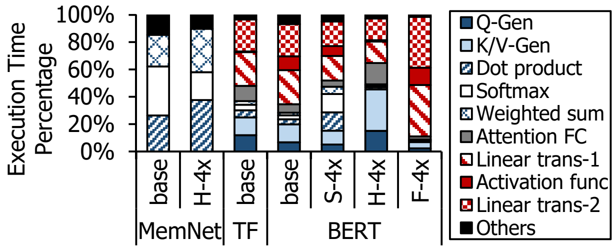


Fig. 4. Performance breakdown of NLP models with various configurations (experimental details are in Section VI-A). MemNet and TF mean Memory networks and Transformer, and S/H/F-4x are configurations with a fourfold increase of s, d_H, and d_{FF}, respectively.

listed in Table II. 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. This is because that TF and BERT have more complex structures than MemNet consisting of multiple attention layers and a simple answer labeling layer.

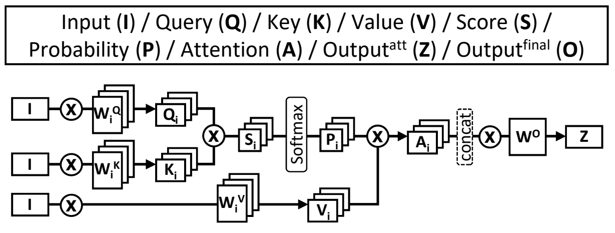
TF (*base*) and BERT (*base*) also show different performance aspects, although they have the same configuration. Instead of using ReLU as the activation function, BERT uses GELU, a more complex operation; therefore, the activation function takes 9.6% more in BERT. Also, in TF, a portion of *Q-Gen* overhead is larger than BERT's *Q-Gen* overhead because the multi-head attention of TF decoder does not contain *K/V-Gen*

Config diversity. Various parameter configurations also incur a huge performance variation. Fig. 4 shows the performance with a fourfold increase of each parameter: s, d_H, and d_{FF}. As shown in Table II, 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* take more portion of the total execution time. Lastly, the overhead of the feed-forward network (i.e., *linear transformation-1*, *activation function*, and *linear transformation-2*) linearly increases with d_{FF}.

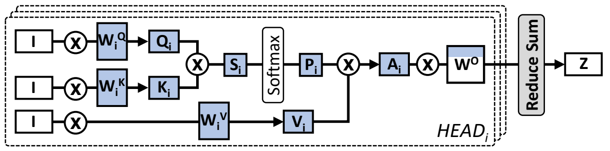
As aforementioned, emerging NLP models suffer from huge performance variation because of various models and parameter configurations. Therefore, ***we propose a holistic solution that optimizes every operation in NLP models***.

B. NLP-Opt #1: Holistic Model Partitioning

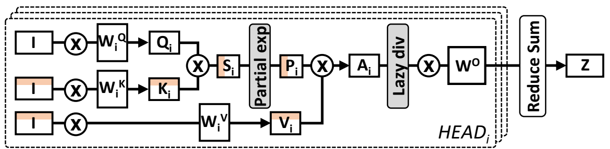
Fig. 5a and Fig. 5d describe the dataflow of baseline *multi-head self-attention* and *feed-forward network*, respectively.


Input (I) / Query (Q) / Key (K) / Value (V) / Score (S) / Probability (P) / Attention (A) / Output^att (Z) / Output^final (O)
Softmax
concat

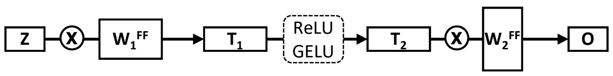
(a) Baseline dataflow of multi-head self-attention.



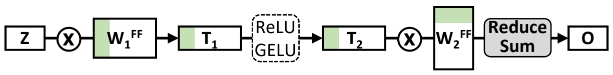
(b) Dataflow of baseline with applying *partial-head update* (P).



(c) Dataflow of baseline with applying both *partial-head update* (P) and *column-based algorithm* (C).



(d) Baseline dataflow of feed-forward network.



(e) Dataflow of baseline with applying *feed-forward splitting* (F).

Fig. 5. Dataflow comparison between baseline and our optimizations: *partial-head update* (P), *column-based algorithm* (C), and *feed-forward splitting* (F).

Note that we describe the case of multi-head self-attention because it includes all attention-related operations (i.e., *Q/K/V-Gen*, *dot product*, *softmax*, *weighted sum*, *attention FC*). As shown in Fig. 5a, during the attention-related operations, the baseline system uses multiple weights (i.e., W_i^Q, W_i^K, W_i^V, W^O) and generates intermediate data (i.e., Q_i, K_i, V_i, S_i, P_i, A_i). Similarly, as shown in Fig. 5d, the 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.

Table III shows the summary of these weights and intermediate data and their memory footprint per each scenario. In the baseline, all variables have quadratic or cubic space complexity, which incurs considerable off-chip memory accessing overhead. To reduce these off-chip memory bandwidth requirements, we need to reduce the working set size. For the purpose, we propose three model partitioning optimizations: *partial-head update*, *column-based algorithm*, and *feed-forward splitting*.

1) Partial-head Update: We exploit inherent parallelism in the multi-head attention. As each head is independent of each other, we can execute each operation in head granularity. To do

TABLE III
ALL WEIGHTS AND INTERMEDIATE DATA WITH THEIR SPACE COMPLEXITY IN NLP MODELS. WE APPLY THREE OPTIMIZATIONS ON THE BASELINE SYSTEM: *partial-head update* (P), *column-based algorithm* (C), AND *feed-forward splitting* (F).

	$\mathbf{I}^Q$	$\mathbf{W}^Q$	**Q**	$\mathbf{I}^K$ / $\mathbf{I}^V$	$\mathbf{W}^K$ / $\mathbf{W}^V$	**K / V**	**S**	**P**	**A**	$\mathbf{W}^O$	$\mathbf{W}_1^{FF}$	$\mathbf{T}_1$ / $\mathbf{T}_2$	$\mathbf{W}_2^{FF}$
Baseline	$O(s \cdot d_H)$	$O(d_H \cdot d_H)$	$O(s \cdot d_H)$	$O(s \cdot d_H)$	$O(d_H \cdot d_H)$	$O(s \cdot d_H)$	$O(s \cdot s \cdot h)$	$O(s \cdot s \cdot h)$	$O(s \cdot d_H)$	$O(d_H \cdot d_H)$	$O(d_H \cdot d_{FF})$	$O(s \cdot d_{FF})$	$O(d_H \cdot d_{FF})$
Base+P	$O(s \cdot d_H)$	$O(d_H)$	$O(s)$	$O(s \cdot d_H)$	$O(d_H)$	$O(s)$	$O(s \cdot s)$	$O(s \cdot s)$	$O(s)$	$O(d_H)$	$O(d_H \cdot d_{FF})$	$O(s \cdot d_{FF})$	$O(d_H \cdot d_{FF})$
Base+P+C	$O(s \cdot d_H)$	$O(d_H)$	$O(s)$	$O(d_H)$	$O(d_H)$	$O(1)$	$O(s)$	$O(s)$	$O(s)$	$O(d_H)$	$O(d_H \cdot d_{FF})$	$O(s \cdot d_{FF})$	$O(d_H \cdot d_{FF})$
Base+P+C+F	$O(s \cdot d_H)$	$O(d_H)$	$O(s)$	$O(d_H)$	$O(d_H)$	$O(1)$	$O(s)$	$O(s)$	$O(s)$	$O(d_H)$	$O(d_H)$	$O(s)$	$O(d_H)$

so, we add *partial attention FC* and *Reduce Sum* operations to partially calculate the multi-head attention 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. Fig. 5b shows the dataflow of modified computational steps after applying partial-head update. We highlight the reduced intermediate data and weights in the figure (color with light blue). Table III summarizes the impact of the partial-head update optimization (Base+P: color with light blue). In each head, all operations use the reduced internal state vectors whose size is *head_size* (d_H/h, h: the number of head), and *head_size* is fixed in different NLP models [2], [3]. Therefore, the partial-head update optimization can successfully reduce the memory footprint of attention-related operations.

2) Column-based Algorithm: Although partial-head update quite reduces the memory footprint, some intermediate data in the attention mechanism still have high space complexity (i.e., score (S), probability (P), and other intermediate data[1]). To reduce the size of these data, we propose *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* that computes the *softmax*'s division operation at last, not in the middle.

$$P = \sum_{\alpha} \frac{e^{q \times k_\alpha} \times v_\alpha}{\sum_\beta e^{q \times k_\beta}} = \frac{1}{\sum_\beta e^{q \times k_\beta}} \sum_{\alpha} e^{q \times k_\alpha} \times v_\alpha \qquad (4)$$

Equation (4) shows the difference between the baseline (LHS) and column-based algorithm (RHS) in the probability calculation process. Compared to the baseline, column-based algorithm pulls the sum ($\sum_\beta e^{q \times k_\beta}$) out of the outer summation ($\sum_\alpha$). 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. Fig. 5c shows the modified dataflow when applying both partial-head update and column-based algorithm simultaneously. We highlight inputs and intermediate data reduced by column-based algorithm (color with light orange). In Table III, we summarize the reduced working set size (Base+P+C: color with light orange).

3) 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 II-A, the first part of the feed-forward network is the linear transformation

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

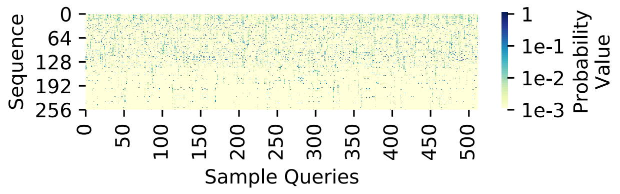


Fig. 6. Probability distribution. Each column is a probability vector of each query. We randomly choose 512 queries in BERT during inference on SQuAD.

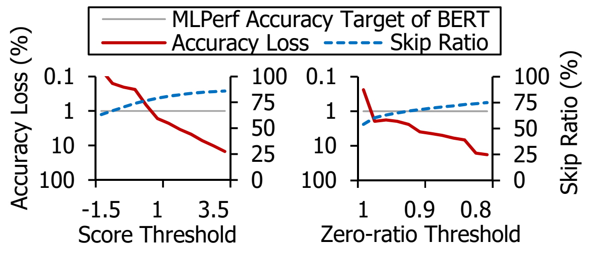
(dimension of internal state vectors: $d_H \rightarrow d_{FF}$) followed by an activation function (i.e., $ActFunc(Z \times W)$). Here, we vertically split the weight matrix in the first linear transformation ($[Z] \times [W_1, W_2]$). In this case, we can separately apply the activation function to each partial result of matrix multiplication. 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 output (O). Fig. 5e illustrates how our *feed-forward splitting* optimization works by highlighting the partial chunks, and Table III summarizes its space complexity (Base+P+C+F: color with light green).

C. NLP-Opt #2: Cross-operation Zero Skipping

We observe that the probability matrix (P in Fig. 5) has a huge sparsity (i.e., most probability values are close to zero). Fig. 6 shows the distribution of probability values in BERT. We use a pre-trained model [3] and conduct a fine-tuning for SQuAD dataset [20]. 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.*

1) Zero Skipping in Various Operations: We propose two skipping techniques: *zero skipping* and *V-Gen skipping*. First, we propose the *zero skipping* optimization to bypass computation in *softmax* and *weighted sum*. In softmax, we only compute exponents of score values when a score value is larger than a *score threshold*. If a score value is lower than the threshold, a corresponding probability value is set to zero. In weighted sum, we skip multiplications for zero probabilities. 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. To quantify trade-offs between accuracy loss and computation reduction, we measure both the accuracy loss and the ratio of computation



(a) Zero skipping. (b) *V-Gen* skipping.

Fig. 7. Tradeoffs between accuracy loss and computation reduction according to each skip threshold (i.e., *score threshold*, *zero-ratio threshold*).

reduction according to different score thresholds (Fig. 7). We use SQuAD dataset for the inference test. Fig. 7a shows the results of the evaluation. The results show that zero skipping can achieve 73.9% computation reduction in both softmax and weighted sum while sacrificing only 0.23% of accuracy when the score threshold is 0.0, which is a negligible accuracy loss.

In addition to *zero skipping*, we propose a more aggressive skipping: *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.

Therefore, in weighted sum ($P \times V$), 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). 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 in the probability matrix 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 (Fig. 7a) according to different zero-ratio thresholds. We set the score threshold as 0.0. Fig. 7b shows the results of the evaluation. The results show that *V-Gen* skipping can achieve 57.0% computation reduction in *V-Gen* operation while sacrificing only 0.85% of accuracy when the zero-ratio threshold is 0.99.

Note that our scheme incurs *tolerable* accuracy loss because the loss does not exceed the target accuracy suggested by MLPerf [21] allowing 1.0% loss in BERT with SQuAD.

2) 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 for each chunk. 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 chunks to idle threads dynamically (*The detailed implementation is in Section V-B*). By doing so, we can achieve high resource utilization (91.2%).

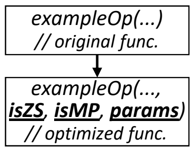
Algorithm 1: Adaptive design space exploration.

```
input  : A model configuration, $model_{config}$ (i.e., model
         structure & parameter configurations).
input  : The number of operations in a target model, $k$.
output : A tuple of the degree of parallelism for each
         operation, $< n_1, n_2, ..., n_k >$.
/* Conduct a performance simulation to get expected
   latencies and resource usage of each operation. */
1  $[< lat_1, LUT_1, FF_1, DSP_1 >, ..., < lat_k, LUT_k, FF_k, DSP_k >] =$
   $perf\_simulation(model_{config})$
/* Find optimized degrees of parallelism for each
   operation:$< n_1, n_2, ..., n_k >$.  */
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 optimal $n_i$ minimizing performance gap. */
6  while true do
7      $pf\_res = FPGA\_profiling(model_{config}, < n_1, ..., n_k >)$
8      $< n'_1, ..., n'_k > = resource\_rebalancing(pf\_res)$
9      if $< n_1, ..., n_k > \neq < n'_1, ..., n'_k >$ then
10         $< n_1, ..., n_k > = < n'_1, ..., n'_k >$
11     else
12         break
13     end
14 end
15 return $< n_1, ..., n_k >$
```

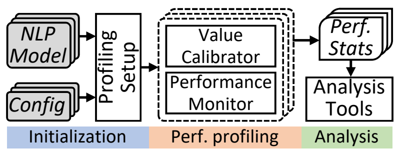
D. NLP-Opt #3: Adaptive Hardware Reconfiguration

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

For this purpose, we optimally distribute the FPGA resources to each operation, considering the computation amount of operations and memory prefetching latency. Algorithm 1 shows how our proposed scheme finds the optimal design point. For the given model and its configuration ($model_{config}$), we extract the expected latencies (lat_i) and resource usage (LUT_i, FF_i, DSP_i) of each operation by conducting a *performance simulation* (line 1). We use the estimated latencies and resource utilization provided by HLS synthesis reports. With the extracted latencies, we solve a nonlinear programming problem for the objective function (line 2) with three



(a) The pre-implemented NLP-Fast libraries.



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

Fig. 8. The NLP-Fast toolkit. Fig. 8a shows an example of applying NLP-Fast's optimizations with the pre-implemented libraries. Fig. 8b illustrates the bottleneck analysis for given NLP models with various configurations.

constraints (line 3–5) by using a *sequential least squares programming* (SLSQP) method. Here, the objective function is set to minimize an expected end-to-end execution latency while meeting the resource constraints in FPGA. We assume that the latency of each operation is inversely proportionate to the amount of resources. Therefore, by solving the problem, we can find the optimal degrees of parallelism for each operation (n_i).

Next, we apply an iterative resource rebalancing technique to minimize the stall time caused by waiting for the long memory access latency (line 6–14). As each operation is concurrently executed with data prefetching for the next operation, some operations (whose prefetching overhead is longer than the execution latency) have to wait until data prefetching is finished. To minimize this stall time, we first synthesize the target design and conduct profiling to quantify the processing stall time (line 7). Then, we redistribute the FPGA resources to minimize the stall overhead (line 8). To be specific, we find the most memory-bound operation whose gap between prefetching and computation is the largest. Next, we deallocate resources from the selected operation by reducing the degree of parallelism and reallocate the extracted resources to the other operations. We exclude the selected operation from the operation list and iteratively perform this resource rebalancing until no further memory-bound operations. Finally, we return the optimal design point (line 15). Here, the maximum number of iterations is limited to the number of operations as we mask out the deallocated operations in each iteration. In this work, it mostly takes less than five iterations for BERT.

IV. NLP-Fast Toolkit

In this section, we explain how to use the NLP-Fast toolkit for improving the overall throughput and performing in-depth performance analysis for each NLP model.

Fig. 8 shows the high-level overview of the NLP-Fast toolkit. The NLP-Fast toolkit mainly consists of two parts: pre-implemented libraries and performance analysis tools.

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. 8a 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).

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, cache hit/miss statistics in Fig. 12).

Fig. 8b 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 operations on normalized floating-point on CPUs[2]. 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 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.

V. Implementation

To show the effectiveness of our NLP-Fast, we implement the three representative models (i.e., memory networks, Transformer, BERT) on CPU, GPU, and FPGA.

A. General-Purpose Architecture (CPU & GPU)

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,

[2]This performance degradation issue does not exist on GPUs as they can support denoramls at full speed [22]

then we elaborate on the specialized hardware design based on the analysis of the general-purpose architecture.

For CPU and GPU results, we choose a standard deep learning library (TensorFlow) as our baseline implementation to show the real impacts of our optimizations. To implement the representative baseline, we refer the TensorFlow (v1.11.0) code for Transformer and BERT models [2], [3], [23]. For Memory networks, unlike Transformer and BERT, there is no standard ML library; therefore, we implement the baseline similar to MnnFast [4]. Similarly, we use Intel Math Kernel Library [24] and cuBLAS [25] (CUDA Toolkit 10.2) to implement general matrix-matrix multiplication (GEMM) operations for CPU and GPU, respectively. For GPU implementation, we apply the same layer fusion optimizations used in TensorRT-version BERT [26].

Then, we implement the CPU/GPU-based NLP-Fast on top of our baseline implementation for each NLP model. We apply two general-purpose optimizations (i.e., *holistic model partitioning*, *cross-operation zero skipping*) to show our NLP-Fast's general applicability as a system solution.

Holistic model partitioning. We implement three model partitioning techniques: *partial-head update*, *column-based algorithm*, and *feed-forward splitting*. For *partial-head update* and *feed-forward splitting*, we add custom functions/kernels to reduce the intermediate data within each partitioned block (e.g., heads, chunks). For the *column-based algorithm*, we split the softmax function/kernel into two parts: partial exponentiation and lazy division. By doing so, we can holistically partition all NLP models into small-chunk granularity.

For the GPU-based baseline, our holistic model partitioning enables GPU to exploit CUDA streams to overlap kernel executions by data transfers between the host and GPU. 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. For CPU, 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 0.0 on BERT). Next, we implement *V-Gen* skipping to reduce the computation overhead of *V-Gen* operation. Here, the zero-ratio threshold is 0.99. In addition to these zero-skipping optimizations, we implement a *software-version dynamic scheduler* to minimize the execution time skewness.

On the other hand, we do not apply the zero-skipping optimizations on the GPU-based NLP-Fast because zero skipping is ineffective or even harmful for GPUs [4], [27], [28]. We modify some operations to use cuSPARSE [29] to measure the performance impacts, and we find out that the matrix transformation (dense-to-sparse, sparse-to-dense) overhead is too large, which results in performance degradation.

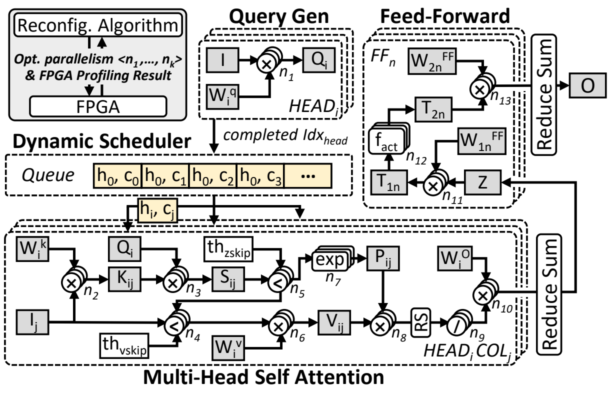


Fig. 9. The architecture of FPGA-based NLP-Fast.

B. Custom Hardware (FPGA)

For our baseline implementation, We use C-based Vivado High-Level Synthesis (HLS) to implement each NLP model (*custom design*). We first implement all basic operations (i.e., MAC, comparator, exponentiator, divider, adder, multiplier, and square root) used in NLP models. For square root and exponentiation, we use HLS math library. Also, we implement other parts (i.e., layernorm, residual sum).

Then, we implement FPGA-based NLP-Fast on top of our baseline implementation. Fig. 9 shows the high-level architecture of the FPGA-based NLP-Fast for BERT. (We omit the baseline design as it is straightforward). Transformer and Memory networks can be implemented in a similar way.

Holistic model partitioning. Our FPGA-based 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 Fig. 5c and Fig. 5e. 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 the dependencies of each BRAM block by using HLS directives.

Cross-operation zero skipping. We set each threshold value via AXI4-lite at the beginning of execution. We use the same threshold values as CPU (0.0 for score threshold, th_{zskip} and 0.99 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(*sequence length*) bits) and check bit (1 bit) per each column, which is under 1KB overhead in total. Note that the logic (e.g., adder, comparator) overhead is negligible.

Dynamic scheduler. For the dynamic scheduler, we use a FIFO queue. When *Q-Gen* is finished for any head, its index (Idx_{head}) is sent to the dynamic scheduler. The scheduler pushes all $\langle Idx_{head}, Idx_{chunk} \rangle$ pairs of the corresponding head to the queue. Then, the scheduler assigns an index pair to any idle execution unit. Note that the queue size is negligible as the queue only stores index pairs which are $\langle log_2(\#head)$ bits, $log_2(\#chunk)$ bits$\rangle$; therefore, the total overhead is under 1KB.

TABLE IV
BASE CONFIGURATIONS OF EACH NLP MODEL.

Entry	MemNet	Transformer	BERT
$s(ns)$	100M	256	256
d_H	64	1024	1024
d_{FF}	-	4096	4096

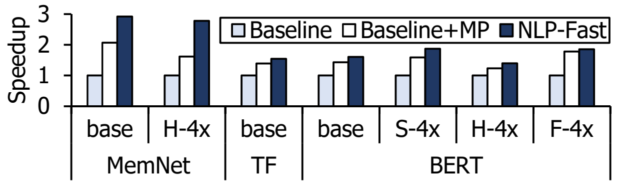


Fig. 10. Speedup results of CPU-based NLP-Fast on different models and configurations. MP means *model partitioning*.

Adaptive Hardware Reconfiguration. To resolve the objective function (described in Section III-D), we use a *sequential least squares programming* (SLSQP) algorithm from the *SciPy* python library. By doing so, we obtain the optimal degree of parallelism for each operation. Then, we use HLS directives (e.g., unroll) to meet the level of parallelism.

VI. EVALUATION

A. Experimental Setup

The hardware (CPU/GPU/FPGA) specifications and NLP models/configs used in the evaluation are as follows.

- **CPU eval.** We use two Xeon CPUs (Intel Xeon Gold 5220 18C/36T) system with DDR4-2400MHz 256GB memory and run NLP models on Ubuntu 16.04 LTS.
- **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.
- **FPGA eval.** We use Xilinx Virtex UltraScale+ FPGA VCU118, and its programmable logic (PL) runs at 100MHz.
- **NLP model config.** Table IV shows *base* parameter configurations (from Transformer and BERT papers [2], [3]) for the evaluation. For *S/H/F-4x* configurations, we increase each parameter (i.e., s, d_H, d_{FF}) in four times, respectively.

B. CPU

Performance. Fig. 10 shows the performance of NLP-Fast on various NLP models with different configurations. We measure execution latencies of three implementations: baseline, baseline+MP (i.e., baseline with *holistic model partitioning*), and NLP-Fast. Our model partitioning achieves 1.59× average speedup on various NLP models. With all optimizations, NLP-Fast achieves 2.00× average speedup. Memory networks gets huge performance improvement from *cross-operation zero skipping* because zero skipping significantly reduces the computation overhead in the attention mechanism (note that the attention mechanism is dominant in memory networks). In this case, NLP-Fast achieves speedup by up to 2.92×.

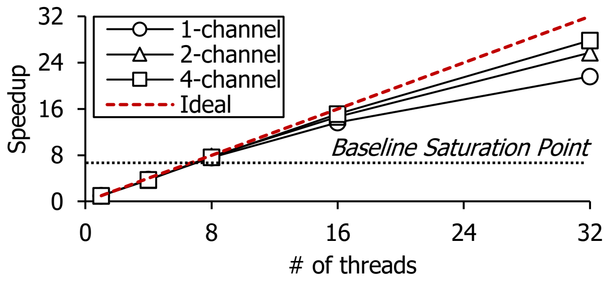


Fig. 11. Scalability on different memory bandwidth.

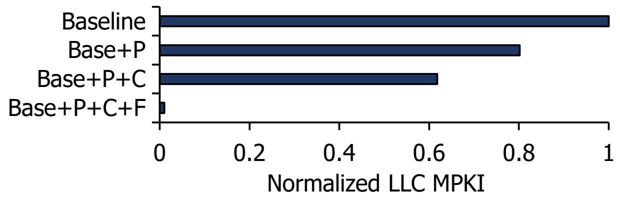


Fig. 12. LLC MPKI normalized to the baseline. P/C/F mean *partial-head update*, *column-based algorithm*, and *feed-forward splitting*, respectively.

Scalability: Cache efficiency. We evaluate the scalability of NLP-Fast on CPU by measuring the execution latency at different memory bandwidth configurations (# of memory channels). Fig. 11 shows speedup normalized to the 1-thread case on each memory configuration. Different from the baseline whose performance is saturated at 8-thread, our NLP-Fast shows highly-scalable performance on 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 [30] while applying each model partitioning optimization one by one. Fig. 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. We eliminate almost all off-chip DRAM accesses with all three optimizations because the reduced working set size fits into the LLC in this evaluation setup.

C. GPU

Performance. We evaluate the performance improvement of NLP-Fast on the single-GPU environment. For GPU, we do not apply the zero skipping optimization (Section V-A). Fig. 13 shows the speedup results of NLP-Fast on various NLP models. As shown in Section V-A, our holistic model partitioning technique enables NLP-Fast to exploit CUDA streams to overlap kernel executions by data transfers between host and device. Also, CUDA streams aid in increasing the computational efficiency of GPU. By doing so, NLP-Fast achieves 1.44× average speedup on various NLP models.

We also evaluate NLP-Fast on different generations of GPUs: Titan Xp and Tesla V100. Fig. 14 shows the speedup

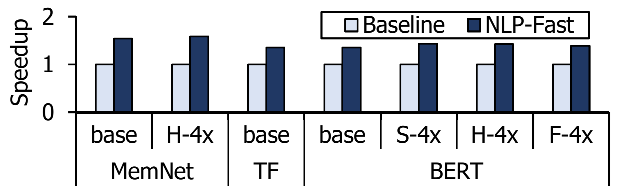


Fig. 13. Single-GPU performance improvement of GPU-based NLP-Fast on various NLP models.

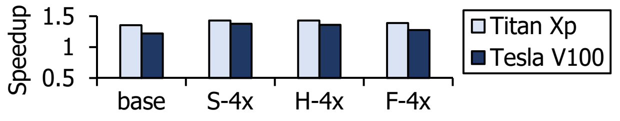
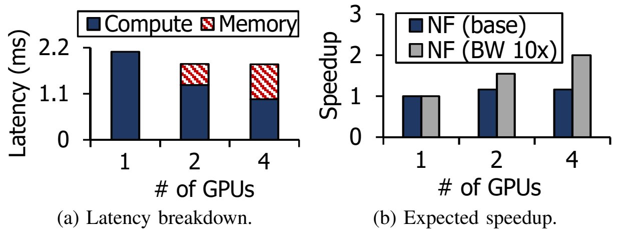


Fig. 14. Performance of GPU-based NLP-Fast on various configurations of BERT for different generations of GPUs: Titan Xp and Tesla V100.



(a) Latency breakdown. (b) Expected speedup.

Fig. 15. The overhead analysis of multi-GPU version of NLP-Fast (NF) and the expected speedup with high-end interconnect (e.g., NVLink 2.0).

results of NLP-Fast on BERT with different configurations. On the high-end GPU (Tesla V100), NLP-Fast can achieve 1.31× average speedup. The results show that NLP-Fast's optimizations are effective to any types of GPUs.

Scalability: scale-out multi-GPU. With multiple GPUs, NLP-Fast can run an extremely large NLP model which cannot be executed on the single-GPU environment. Note that our extra communication overhead for supporting multi-GPU version NLP-Fast is negligible as NLP-Fast does not require many synchronization points. Fig. 15 shows the results of the multi-GPU evaluation. We run BERT with the base configuration (*batch_size* is 1) and measure single-layer execution latency. Fig. 15a shows the execution latency of NLP-Fast on different numbers of GPUs and the overhead breakdown (compute vs. memory). In this case, NLP-Fast achieves 1.16× and 1.17× speedup on two GPUs and four GPUs, respectively.

It seems that NLP-Fast cannot achieve scalable performance due to high memory accessing overhead during synchronization (i.e., *layernorm*). The latency breakdown (Fig. 15a) shows that the synchronization memory overhead takes 27.6%, and 46.4% on two and four GPUs, respectively. Fortunately, this high memory accessing overhead is continuously alleviated as interconnect technology scales. For example, NVIDIA NVLink 2.0 [31] achieves 10× more bandwidth than PCIe Gen 3. Fig. 15b shows the expected speedup of NLP-Fast with NVLink 2.0. In this case, NLP-Fast can achieve 1.55× and 2.00× speedup on two and four GPUs, respectively. In addition to the interconnect technology advancement, the hardware-assisted interconnect can also reduce the synchronization overhead [32]. Therefore, with NLP-Fast's holistic model partitioning, we can achieve scalable performance.

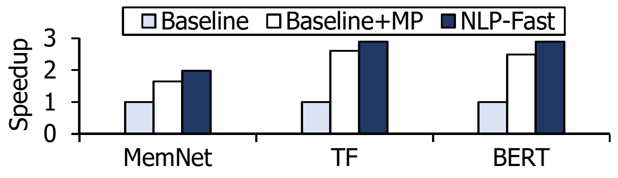
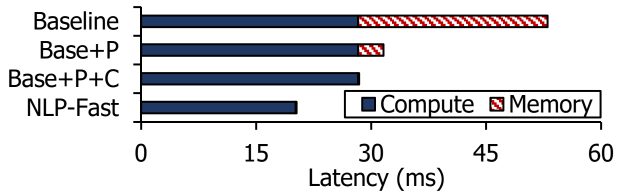
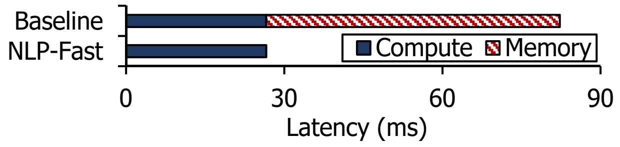


Fig. 16. Performance of FPGA-based NLP-Fast on various models. MP means model partitioning.



(a) Multi-head self-attention.



(b) Feed-forward network.

Fig. 17. Latency of FPGA-based NLP-Fast on BERT. P and C mean *partial-head update* and *column-based algorithm*.

D. FPGA

Performance. Fig. 16 shows the performance of FPGA-based NLP-Fast on various NLP models: Memory networks, Transformer, and BERT. The results illustrate that NLP-Fast achieves gradual speedup as we apply the proposed optimizations. Our *holistic model partitioning* (MP) achieves by up to 2.60× speedup, and our *cross-operation zero skipping* provides an additional 1.20× speedup. With all schemes applied, NLP-Fast accomplishes a total speedup by up to 2.89×.

We quantify the impacts of each optimization on BERT (Fig. 17). Fig. 17a 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 minimizing the data spilling and enabling data prefetching. Next, with *cross-operation zero skipping*, NLP-Fast achieves latency reduction by 61.8% compared to the baseline. Similar to the attention part, Fig. 17b shows that *feed-forward splitting* (F) achieves latency reduction by 67.7%.

Adaptive hardware reconfiguration. To show the effectiveness of our *adaptive hardware reconfiguration*, we evaluate NLP-Fast with various configurations. We assume the baseline design has the same parallelism degree for each operation. Fig. 18 shows that our scheme achieves more than 1.5× speedup on every configuration. This is because NLP-Fast redistributes the resources from non-performance

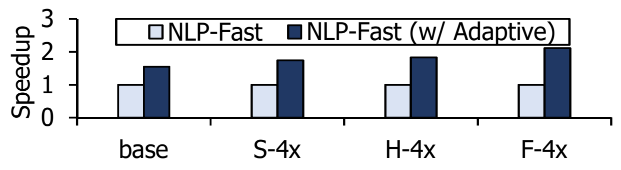


Fig. 18. Effectiveness of *adaptive hardware reconfiguration* on various configurations of BERT.

critical operations to other critical operations. Especially, for F-4x configuration, *linear trans* operations take 74.7% of latency while others take 25.3%. Therefore, F-4x configuration achieves a higher speedup than other cases (2.11×).

We observe that *adaptive hardware reconfiguration* is essential because the specific design point works poorly on other configurations. For example, if we run BERT with F-4x configuration on the FPGA accelerator designed for H-4x configuration, we can only get 0.47× performance of the optimal design for the F-4x configuration. Moreover, this performance loss aggravates as the model size increases (e.g., 0.26× at eightfold, 0.12× at sixteenfold).

Comparison between CPU and FPGA. We compare energy efficiency between CPU-based NLP-Fast and FPGA-based NLP-Fast on different NLP models. We use *powerstat* for CPU and Xilinx Vivado Design Suite for FPGA on average power measurement. The results show that FPGA-based NLP-Fast improves energy efficiency by 6.54× and 9.82× on MemNet and BERT, respectively.

VII. Related Work

Various NLP Accelerators. There are many attempts to accelerate DNNs [33]–[38]. Some studies try to accelerate DNNs concerning memory [36], [39], while others present acceleration with configurable hardware [40]–[42]. For RNNs, there are some works with different accelerating approaches [43]–[48]. However, only a few proposals are aiming to accelerate attention-based networks. Ham et al. [5] propose a specific architecture conducting approximation on the attention mechanism. However, these studies cannot be always effective due to the limited range of acceleration. Stevens et al. [49] present a memory-centric hardware architecture for attention-based networks, which is also beneficial for NLP-Fast. TensorRT [26] and ONNX Runtime [50] 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.

Model partitioning. To achieve higher accuracy, DNNs are becoming bigger and more complex [51]. Model partitioning provides scalability to handle the growing size of DNNs. Gao et al. [51] propose dataflow optimizations exploiting both intra-layer parallelism and inter-layer pipelining. Lu et al. [52] introduce a flexible dataflow architecture for various CNN workloads. Jia et al. [53] use a deep learning engine to find a customized parallelization strategy. Parashar et al. [54] propose the dataflow-specific data mapping on the memory hierarchy, which seems their key idea is also useful for our work. Kim et al. [55] exploit model partitioning to use CPU and GPU simultaneously. Some studies [4], [56] try to reduce working set size by applying model partitioning; however, no work holistically partitions all operations in emerging NLP models.

Approximation. Some studies exploit sparsity in NNs by applying approximation on near-zero values [57], [58]. Shi et al. [59] achieve speedup by designing an algorithm that removes zero operands not used in later operations. Parashar et al. [60] propose an innovative dataflow to eliminate multiplications with zero operands during the calculation.

Weight Pruning and quantization are well-known approximation methods on DNNs. Thanks to fault tolerance of DNN [61]–[63], weight pruning and quantization reduce the storage requirement with negligible accuracy loss [64], [65]. Some studies propose the network compression technique by leveraging the sparsity in LSTMs and CNNs [66], [67]. Also, [68] proposes a hardware framework to reduce data communication overhead in NoCs by approximation.

Resource Optimization. Some studies provide frameworks for automatic design space exploration on FPGA [69]–[75], and others try to maximize resource utilization [76], [76], [77]. Some works aim to find an optimal design point for various traditional neural networks (e.g., DNN, CNN, RNN, MLP) [42], [78], [79]. To the best of our knowledge, our work is the first attempt to find an optimal design for NLP models.

VIII. 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 two general-purpose optimizations and one hardware-specific optimization to optimize various NLP models on any hardware platform: CPU, GPU, and FPGA.

Acknowledgment

This work was partly supported by Samsung Electronics, National Research Foundation of Korea (NRF) grant funded by the Korean Government (NRF-2019R1A5A1027055, NRF-2021R1A2C3014131, NRF-2021M3F3A2A02037893), Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean Government (No.1711080972, No.2021000853). We also appreciate the support from Artificial Intelligence Institute (AIIS), Automation and Systems Research Institute (ASRI) and Inter-university Semiconductor Research Center (ISRC) at Seoul National University.

References

[1] 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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] J. Weston, "Dialog-based language learning," *CoRR*, vol. abs/1604.06045, 2016. [Online]. Available: http://arxiv.org/abs/1604.06045

[7] 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.

[8] 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.

[9] 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

[10] J. Weston, S. Chopra, and A. Bordes, "Memory networks," *CoRR*, vol. abs/1410.3916, 2014. [Online]. Available: http://arxiv.org/abs/1410.3916

[11] A. Bordes, N. Usunier, S. Chopra, and J. Weston, "Large-scale simple question answering with memory networks," *arXiv preprint arXiv:1506.02075*, 2015.

[12] 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.

[13] 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

[14] 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.

[15] N. Kitaev, Ł. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," *arXiv preprint arXiv:2001.04451*, 2020.

[16] 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.

[17] 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.

[18] A. Conneau and G. Lample, "Cross-lingual language model pretraining," in *Advances in Neural Information Processing Systems*, 2019, pp. 7057–7067.

[19] 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.

[20] 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

[21] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 446–459.

[22] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[23] "Tensor2tensor. google open-source projects," Available online at https://github.com/tensorflow/tensor2tensor, accessed: 2020-04.

[24] 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.

[25] "cuBLAS," Available online at https://docs.nvidia.com/cuda/cublas/index.html, accessed: 2020-04.

[26] "Real-time natural language understanding with bert using tensorrt," Available online at https://developer.nvidia.com/blog/nlu-with-tensorrt-bert/, accessed: 2020-08.

[27] 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

[28] 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

[29] "cuSPARSE," Available online at https://docs.nvidia.com/cuda/cusparse/index.html, accessed: 2020-04.

[30] S. Eranian, "Perfmon2: a flexible performance monitoring interface for linux," in *Proc. of the 2006 Ottawa Linux Symposium*. Citeseer, 2006, pp. 269–288.

[31] "Nvidia nvlink and nvswitch," Available online at https://www.nvidia.com/en-us/data-center/nvlink/, accessed: 2020-04.

[32] B. Klenk, N. Jiang, G. Thorson, and L. Dennison, "An in-network architecture for accelerating shared-memory multiprocessor collectives," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 996–1009.

[33] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Neural acceleration for gpu throughput processors," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 482–493.

[34] 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.

[35] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *International conference on artificial neural networks*. Springer, 2014, pp. 281–290.

[36] 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.

[37] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 14–27.

[38] C.-E. Lee, Y. S. Shao, J.-F. Zhang, A. Parashar, J. Emer, S. W. Keckler, and Z. Zhang, "Stitch-x: An accelerator architecture for exploiting unstructured sparsity in deep neural networks," in *SysML Conference*, 2018.

[39] 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.

[40] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2018.

[41] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera,

L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] B. Zheng, N. Vijaykumar, and G. Pekhimenko, "Echo: Compiler-based gpu memory footprint reduction for lstm rnn training," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 1089–1102.

[49] 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.

[50] "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.

[51] 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.

[52] 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.

[53] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," 2018.

[54] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2019, pp. 304–315.

[55] Y. Kim, J. Kim, D. Chae, D. Kim, and J. Kim, "μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–15.

[56] 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.

[57] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 662–673.

[58] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. M. Rush, G.-Y. Wei, and D. Brooks, "Masr: A modular accelerator for sparse rnns," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 1–14.

[59] S. Shi and X. Chu, "Speeding up convolutional neural networks by exploiting the sparsity of rectifier units," 2017.

[60] 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

[61] C. Yu, H. Tang, C. Renggli, S. Kassing, A. Singla, D. Alistarh, C. Zhang, and J. Liu, "Distributed learning over unreliable networks," 2018.

[62] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 356–367.

[63] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. Moshovos, "Proteus: Exploiting numerical precision variability in deep neural networks," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–12.

[64] H. Wang, Z. Wu, Z. Liu, H. Cai, L. Zhu, C. Gan, and S. Han, "Hat: Hardware-aware transformers for efficient natural language processing," in *Annual Conference of the Association for Computational Linguistics*, 2020.

[65] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2015.

[66] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 75–84. [Online]. Available: https://doi.org/10.1145/3020078.3021745

[67] D. Zhang, J. Yang, D. Ye, and G. Hua, "Lq-nets: Learned quantization for highly accurate and compact deep neural networks," in *The European Conference on Computer Vision (ECCV)*, September 2018.

[68] R. Boyapati, J. Huang, P. Majumder, K. H. Yum, and E. J. Kim, "Approx-noc: A data approximation framework for network-on-chip architectures," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 666–677.

[69] 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.

[70] 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.

[71] H. Calborean and L. Vinţan, "An automatic design space exploration framework for multicore architecture optimizations," in *9th RoEduNet IEEE International Conference*. IEEE, 2010, pp. 202–207.

[72] 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.

[73] 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 (TRETS)*, vol. 1, no. 3, pp. 1–30, 2008.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 14–26.

[79] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, Y. Zhang, B. Zimmer, W. J. Dally, J. Emer, S. W. Keckler, and B. Khailany, "Magnet: A modular accelerator generator for neural networks." in *ICCAD*, 2019, pp. 1–8.