A Fast and Flexible FPGA-based Accelerator for Natural Language Processing Neural Networks

SUYEONC HUR, SEONGMIN NA, DONGUP KWON, and JOONSUNG KIM, Seoul National University, Republic of Korea
ANDREW BOUTROS and ERIKO NURVITADHI, MangoBoost Inc., United States
JANGWOO KIM, Seoul National University, Republic of Korea

Deep neural networks (DNNs) have become key solutions in the natural language processing (NLP) domain. However, the existing accelerators customized for their narrow target models cannot support diverse NLP models. Therefore, naively running complex NLP models on the existing accelerators often leads to very marginal performance improvements. For these reasons, architects are now in dire need of a new accelerator that can run various NLP models while taking its full performance potential. In this article, we propose FlexRun, an FPGA-based modular accelerator to efficiently support diverse and complex NLP models. First, we identify key components commonly used by NLP models and implement them on top of a current state-of-the-art FPGA-based accelerator. Next, FlexRun conducts an in-depth design space exploration to find the best accelerator architecture for a target NLP model. Last, FlexRun automatically reconfigures the accelerator based on the exploration results. Our FlexRun design outperforms the current state-of-the-art FPGA-based accelerator by 1.21×–2.73× and 1.15×–1.50× for BERT and GPT2, respectively. Compared to Nvidia's V100 GPU, FlexRun achieves 2.69× higher performance on average for various BERT and GPT2 models.

CCS Concepts: • **Hardware** → **Emerging technologies**; **Emerging architectures;**

Additional Key Words and Phrases: Neural networks, Natural langauge processing, modular architecture, FPGA

ACM Reference format:
Suyeonc Hur, Seongmin Na, Dongup Kwon, Joonsung Kim, Andrew Boutros, Eriko Nurvitadhi, and Jangwoo Kim. 2023. A Fast and Flexible FPGA-based Accelerator for Natural Language Processing Neural Networks. *ACM Trans. Arch. Code Optim.* 20, 1, Article 11 (February 2023), 24 pages.
https://doi.org/10.1145/3564606

Andrew Boutros and Eriko Nurvitadhi author's affiliation was Programmable Solutions Group, Intel Corporation, when writing the article.
This work was supported in part by SNU-SK Hynix Solution Research Center (S3RC), National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) (NRF-2021-M3F3A2A02037893, NRF-2021R1F1A1062902), Institute of Information & Communications Technology Planning & Evaluation funded by the Korean Government (MSIT) (No. 1711080972, No. 2021000853), and Creative Pioneering Researchers Program through Seoul National University. We also appreciate the support from Automation and Systems Research Institute (ASRI) and Inter-university Semiconductor Research Center (ISRC) at Seoul National University.
Authors' addresses: S. Hur, S. Na, D. Kwon, J. Kim, and J. Kim (corresponding author), Seoul National University, Gwanak-ro 1 gil, Gwanak-gu, Seoul, Republic of Korea, 08826; emails: {suyeon339, seongmin.na, dongup, joonsung90, jangwoo}@snu.ac.kr; A. Boutros and E. Nurvitadhi, MangoBoost Inc.; emails: eriko@mangoboost.io, andrew.mboutros@gmail.com.

1544-3566/2023/02-ART11 $15.00
https://doi.org/10.1145/3564606

1 INTRODUCTION

Recently, **deep neural network (DNN)**-based **natural language processing (NLP)** models are rapidly developing, bringing a breakthrough in the field of NLP. There are two types of DNN-based NLP models, and both types show high accuracy in various NLP tasks. For example, recurrent neural network-based [30] models are good at speech recognition [16]. Meanwhile, attention-based models show good performance for tasks such as question answering and language modeling.

There are various types of services that use NLP models. Some of these NLP tasks have strict real-time constraints (i.e., real-time interactive services [11]) while others are performed offline with relaxed latency requirements (e.g., text summarization). For the real-time NLP tasks, it is necessary to support a fast batch-1 inference for the immediate responses. However, it is challenging to accelerate NLP models in a single batch due to their following characteristics: (1) diverse and complex operations, (2) various ranges of dimensions, (3) various parameter configurations, and (4) heterogeneous vector operations. In this article, we focus on accelerating NLP tasks for real-time interactive NLP services.

These characteristics of NLP models incur three challenges (i.e., non-negligible vector operations' latency, a wide range of dimensions and irregular matrix operations, heterogeneity of vector operations). First, challenge 1: We need to reduce the overhead of vector operations. According to characteristic (1), NLP models have a number of complex vector operations, resulting in non-negligible overhead. Second, challenge 2: We should cover a wide range of dimensions and deal with irregular matrix operations in the models. Due to characteristics (2) and (3), NLP models' sizes and dimensions span a very wide range, and there are many irregular matrix operations. Last, challange 3: We should handle the heterogeneity of vector operations because in characteristic (4), NLP models consist of vector operations of different types, orders, and lengths.

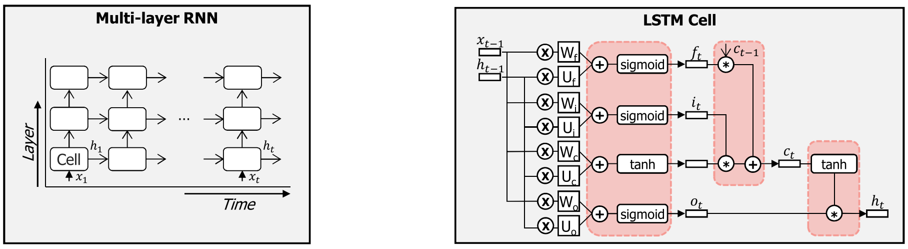
However, existing accelerators [1, 11–13, 26] cannot solve these three challenges. For example, GPUs show low utilization when running NLP models in a single batch or running models with small dimensions because they are throughput-oriented. Also, some works design ASICs for target models. But ASICs made for a particular model and configuration [13] perform poorly on different models or the same model with different configurations. Therefore, prior approaches degrade the performance of NLP models that they are not originally targeting. Also, they fail to support various NLP models due to the absence of functional units required by some NLP models.

In this article, we propose FlexRun, an FPGA-based modular architecture approach to solve the three challenges of accelerating NLP models. FlexRun exploits the high reconfigurability of FPGAs to dynamically adapt the architecture to the target model and its configuration. FlexRun includes three main schemes, *FlexRun:Architecture, FlexRun:Algorithm, and FlexRun:Automation.*

First, FlexRun:Architecture is an FPGA-based flexible base architecture template. Our base architecture template alleviates the overhead of vector operations by adopting a deeply pipelined architecture, resolving challenge 1. Most importantly, it consists of parameterized pre-defined basic modules so we can configure the architecture to fit the input model and its configuration.

Next, we suggest FlexRun:Algorithm, design space exploration algorithms to get the optimal compute unit (i.e., matrix unit, vector unit) design by finding the best modules and parameters set for the input models, resolving challenge 2 and challenge 3. For FlexRun:Algorithm, we define the design space of the base architecture template (i.e, matrix unit: three dimensions of matrix multiplication unit, vector unit: vector operators' types, order, and number).

Last, we propose an automatic tool, FlexRun:Automation, which automates the entire flow to find the best architecture and implement it. FlexRun:Automation reconfigures compute units, memory units, and interconnects according to the results of FlexRun:Algorithm. Also, it generates a new decoder so instructions can be properly decoded to the modified architecture.



(a) The architecture of multi-layer RNNs. (b) Operations of LSTM's cell.

Fig. 1. Architecture of RNN-based models 1(a) and detailed cell operations of an LSTM 1(b).

For evaluation, we compare FlexRun with Intel's Brainwave-like architecture [26] on FPGA (Stratix10 GX and MX) and equivalent GPU (Tesla V100) with tensor cores enabled. First, compared to the FPGA baseline, FlexRun achieves an average speedup of 1.59× on various configurations of BERT. For GPT2, FlexRun gets 1.31× average speedup. Next, when comparing to the GPU implementation, FlexRun improves the performance by 2.79× and 2.59× for BERT and GPT2, respectively. Last, we evaluate the scalability of FlexRun by doubling the compute and memory resources of FPGAs, modeling hypothetical next-generation FPGAs. The results show that FlexRun is able to get the scalable performance improvement, showing 1.57× additional speedup compared to current generation FPGAs.

2 BACKGROUND

2.1 Neural Network-based NLP Models

There are two types of DNN-based NLP models. The first is **Recurrent Neural Networks (RNNs)** models such as SRNN, **long short term memory (LSTM)** [17], and **gated recurrent neural networks (GRU)** [30]. The other is attention-based NLP models, which include Transformer [35], BERT [10], and GPT2 [27]. Both types are used for various NLP tasks such as speech recognition and question-answering. For example, RNN-based models are mainly for speech recognition [16]. Meanwhile, attention-based models exhibit high accuracy in question-answering tasks [28].

2.1.1 RNN-based NLP Models. First, we introduce RNN-based NLP models (e.g., LSTM, GRU). In Figure 1(a), RNNs are structures in which cells of the same operations are repeated with incoming inputs over time. The operations in the cell differ, depending on the type of RNNs, but they usually consist of matrix-vector multiplications and vector operations. The cells are stacked for higher accuracy, which is called a multi-layer model. As an example, we describe the cell operations of LSTM [17]. An LSTM's cell consists of four gates (i.e., forget, input, cell, output) and each gate has two weight matrices and one bias vector ($W_f, U_f, b_f, W_i, U_i, b_i, W_c, U_c, b_c, W_o, U_o, b_o$) of the same dimensions ($W \in \mathbb{R}^{d_{hidden} \times d_{hidden}}$, $b \in \mathbb{R}^{1 \times d_{hidden}}$). Figure 1(b) illustrates each gate's operations. In Figure 1(b), x_t, h_t, i_t, f_t, c_t, and o_t indicate the input, state vector, input gate, forget gate, cell state, and output gate of time t, respectively. They are vectors of the same dimension, $\mathbb{R}^{1 \times d_{hidden}}$. Also, "$*, +, sigmoid, tanh$ ($hyperbolic\ tangent$)" are element-wise vector operations, while "×" is a matrix-vector multiplication. In Figure 1(b), we highlight the vector operations with red boxes, and we omit bias add operations for simplicity.

2.1.2 Attention-based NLP Models. Next, attention-based NLP models such as BERT [10] and GPT2 [27] achieve high accuracy, thanks to attention operations [5] that can catch the relationship between words. In addition to the attention operations, they include many other complex operations.

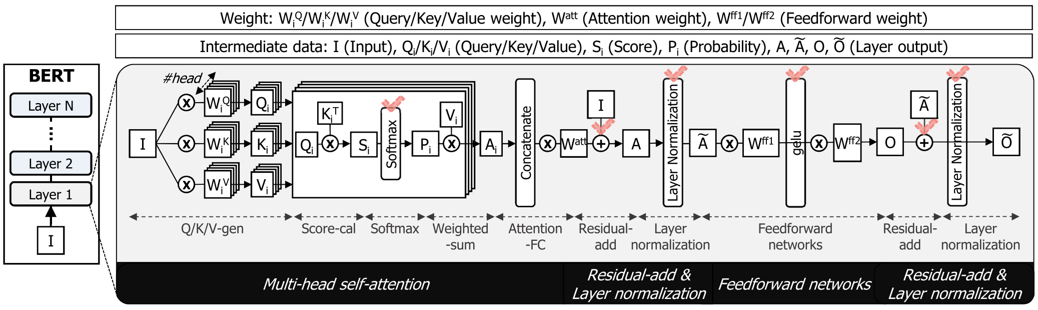

Weight: $W_i^Q/W_i^K/W_i^V$ (Query/Key/Value weight), W^{att} (Attention weight), W^{ff1}/W^{ff2} (Feedforward weight)
Intermediate data: I (Input), $Q_i/K_i/V_i$ (Query/Key/Value), S_i (Score), P_i (Probability), A, $\widetilde{A}$, O, $\widetilde{O}$ (Layer output)
BERT
Layer N
Layer 2
Layer 1
I
#head
Softmax
Concatenate
Layer Normalization
gelu
Q/K/V-gen
Score-cal
Softmax
Weighted -sum
Attention -FC
Residual-add
Layer normalization
Feedforward networks
Residual-add
Layer normalization
Multi-head self-attention
Residual-add & Layer normalization
Feedforward networks
Residual-add & Layer normalization

Fig. 2. Operations for a one layer of BERT.

We describe BERT as an example of the attention-based NLP models. As shown in Figure 2, BERT consists of multiple layers, and each layer has weights of diverse dimensions (e.g., $W_i^Q/W_i^K/W_i^V \in \mathbb{R}^{d_{hidden} \times d_{head}}$, $b_i^Q/b_i^K/b_i^V \in \mathbb{R}^{1 \times d_{head}}$, $W^{att} \in \mathbb{R}^{d_{hidden} \times d_{hidden}}$, $W^{ff1} \in \mathbb{R}^{d_{hidden} \times d_{ff}}$, $b^{ff1} \in \mathbb{R}^{1 \times d_{ff}}$, $W^{ff2} \in \mathbb{R}^{d_{ff} \times d_{hidden}}$, $b^{ff2}/gamma^{1,2}/beta^{1,2}, \in \mathbb{R}^{1 \times d_{hidden}}$ where $i = 1 \ldots \#head$). When BERT receives input I ($I \in \mathbb{R}^{S \times d_{hidden}}$), the operations of Figure 2 are repeated as many as the number of layers. We highlight the vector operations (i.e., *Softmax*, +, *Layer Normalization*, *gelu*) with red check marks. *Layer Normalization* is complex operations that include several basic vector operations such as addition or multiplication. Note that we omit the bias operations in Figure 2 for simplicity, too.

Other attention-based NLP models are similar to BERT, but they have some differences. For example, GPT2 [27] does the same operations as BERT, but it has dependencies between incoming input vectors. The next input vector cannot start processing until the operations on the previous input vector are complete. Such structures with dependencies between the inputs are called decoder structures. However, BERT has no dependency between its inputs. The structures like BERT are called encoder structures. In the case of a Transformer [35], it includes both an encoder and a decoder structure, and ReLU is used as an activation function instead of gelu.

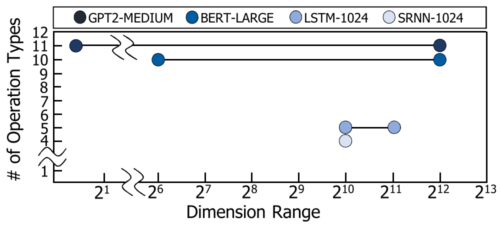
2.2 Fast Inference Support for NLP Tasks

For NLP tasks, fast inference in a single batch is very important [11]. This is because there are many real-time interactive services (e.g., speech recognition, translation) that use NLP models. Interactive services require immediate responses; otherwise, their **quality of service (QoS)** will be seriously compromised. For immediate responses, we cannot deploy batch processing that collects multiple inputs and processes them at once. Instead, the input should be processed as soon as it arrives. For example, two inference scenarios of MLPerf [29] that target responsiveness-critical applications (e.g., online translation) set their batch size as 1 and use latency as the key evaluation metric.

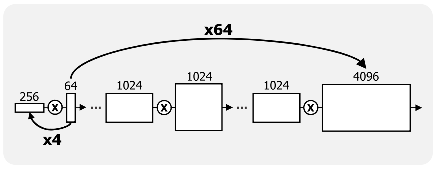
3 MOTIVATION

3.1 Characteristics of NLP Models

3.1.1 Diverse and Complex Operations. **The NLP models consist of diverse and complex operations.** In Figure 3(a), the y-axis indicates how many different types of operations each NLP model has (e.g., gemv, gemm, transpose, exponent, sigmoid, tanh, ReLU, gelu, add/sub, multiplication, reduction, square, sqrt, reciprocal). In the figure, NLP models include several different operations. For example, BERT-LARGE has 10 different types of operations. In detail, we list the operations each NLP model contains in Table 1. As shown in Table 1, NLP models, especially



(a) The number of operation types and range of dimensions for NLP models



(b) Some of gemm operations in BERT-LARGE and operations' dimensions.

Fig. 3. Figure 3(a) shows the number of operation types (y-axis) and ranges of dimensions (x-axis) for GPT2-MEDIUM, BERT-LARGE, LSTM-1024, and SRNN-1024. The word and number after the hyphen represents the parameter scales. Figure 3(b) shows some gemm (general matrix multiplications) operations in BERT-LARGE.

Table 1. Matrix and Vector Operation Types in NLP Models

NLP model	Matrix Operations			Vector Operations					
	gemv	gemm	transpose	activation	exp	add/sub	mul	reduction	square/sqrt/div
SRNN	✓	-	-	tanh	-	✓	✓	-	-
LSTM	✓	-	-	sig/tanh	-	✓	✓	-	-
GRU	✓	-	-	sig/tanh	-	✓	✓	-	-
Transformer	✓	✓	✓	ReLU	✓	✓	✓	✓	✓
BERT	-	✓	✓	gelu	✓	✓	✓	✓	✓
GPT2	✓	✓	✓	gelu	✓	✓	✓	✓	✓

gemv and *gemm* are general matrix-vector and matrix-matrix multiplications, respectively. In case of the vector operations, all operations except reduction are element-wise operations. *tanh*, *sig*, *exp*, *mul*, and *div* stand for tangent hyperbolic, sigmoid, exponential, multiplication, and division operations, respectively.

attention-based NLP models (i.e., BERT, GPT2) have a number of complex operations such as exponent or reduction.

3.1.2 Various Range of Dimensions. **NLP models have operations of various dimensions**. First, NLP models have to deal with a wide range of dimensions. In Figure 3(a), the x-axis is the dimension range of different NLP models. Dimension range is the minimum and maximum values of the operations' dimensions in the models. The figure shows that attention-based NLP models such as BERT and GPT2 consist of operations of varying dimensions. For example, BERT-LARGE has dimension sizes ranging from 64 to 4,096.

Also, gemm operations in attention-based NLP models have very irregular dimensions rather than a square. For example, in Figure 3(b), we can observe that the first gemm operation of BERT-LARGE has two dimensions, 64 and 256, that differ by four times.

3.1.3 Various Parameter Configurations. **For each NLP model, there are many different parameter configurations.** Table 4 shows some parameter configurations for BERT. In the table, there are many versions of BERT according to parameter scales, from TINY to **MG3 (Megatron3)** [33]. These different versions do the same operations but with totally different parameter scales. For example, BERT-MG3 uses 24× larger d_{hidden} dimensions (3,072) than BERT-TINY (128). Other NLP models also have various parameter configurations.

3.1.4 Heterogeneous Vector Operations. **NLP models have heterogeneous vector operations.** Figure 4 visualizes the lists of vector operations that are executed sequentially (i.e.,

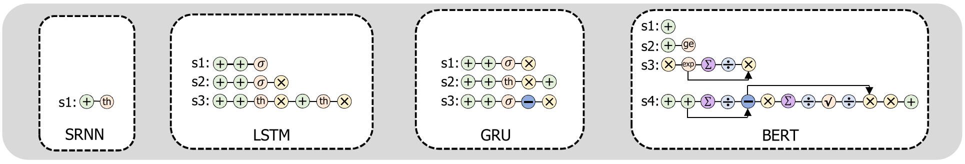


Fig. 4. The lists of sequential-vector-operations in SRNN, LSTM, GRU, and BERT. Sequential-vector-operations are marked with s# in the figure.

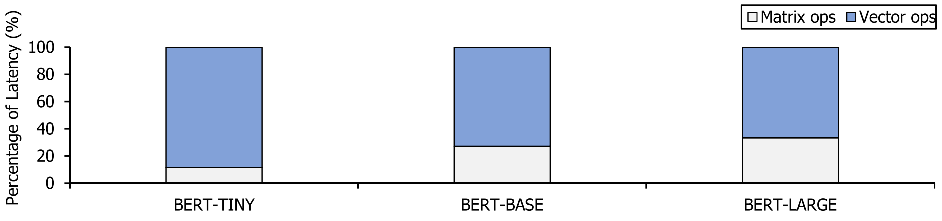


Fig. 5. The latency breakdown of BERT-TINY, BERT-BASE, and BERT-LARGE on Tesla V100. We use XLA [8], the compiler optimization provided by TensorFlow [4]. Refer to Section 5.2.2 for a detailed experimental setup.

sequential-vector-operation, marked as *s* in the figure) in SRNN, LSTM, GRU, and BERT. From the figure, we can observe that most NLP models have quite heterogeneous sequential-vector-operations. For example, there is no overlap between BERT's sequential-vector-operations (i.e., *s1* ~ *s4*) and GRU's sequential-vector-operations (i.e., *s1* ~ *s3*). Also, *s2* and *s4* of BERT greatly differ in both lengths and types of vector operations.

We summarize the characteristics of NLP models as follows:

- NLP models have various and complex operations. Specifically, attention-based NLP models have much more complex and diverse operations.
- NLP models consist of a wide range of dimensions and irregular matrix operations.
- NLP models have a large diversity in terms of parameter configurations.
- The sequential-vector-operations of NLP models are heterogeneous.

3.2 Challenges of NLP Models

From the characteristics of NLP models, we derive challenges that make fast inference support difficult for NLP models in a single batch environment.

3.2.1 Challenge 1: Non-negligible Vector Operations' Latency. First, complex vector operations of NLP models have non-negligible overhead. They take a dominant portion of total latency in a single batch environment. Figure 5 is a latency breakdown of BERT-TINY, BERT-BASE, and BERT-LARGE on GPU Tesla V100 [1]. For BERT-LARGE, vector operations take 66.7% of total latency, which is comparable to twice the matrix operations. Also, as the size of the model shrinks, the portion taken by vector operations grows. In BERT-BASE and BERT-TINY, vector operations take 72.9% and 88.5% of total latency, respectively. Note that we apply XLA [8], the compiler optimization of TensorFlow [4] that minimizes the vector operations' latency by applying layer fusion. Therefore, to achieve high performance for NLP models, we should further reduce the overhead of vector operations.

3.2.2 Challenge 2: Wide Range of Dimensions and Irregular Matrix Operations. Next, NLP models cover a very wide range of dimensions, as shown in Figure 3(a). Also, in attention-based NLP models, most matrix operations are irregular (Figure 3(b)). In addition, there are diverse parameter configurations in the same model (Table 4). Therefore, the NLP accelerator should be able to run both small and big models quickly while dealing with various dimensions of irregular matrix operations. However, the conventional accelerators (e.g., big square systolic array) exhibit slow inference due to the low utilization of matrix operations, as they cannot deal with diversity and irregularity of dimensions.

3.2.3 Challenge 3: Heterogeneity of Vector Operations. As shown in the Figure 4, the NLP models consist of vector operations of different types, orders, and lengths. As previously mentioned, there is no overlap between the models' sequential-vector-operations (i.e., BERT: *s1* ~ *s4* and GRU: *s1* ~ *s3*). Also, sequential-vector-operations vary within the same model, too. In Figure 4, BERT's *s3* and *s4* are different in the types, orders, and lengths of the vector operations. So, to accelerate the NLP models, it is essential to efficiently support all models' vector operations.

3.3 Limitations of Previous Works

There have been many works to accelerate the NLP models. They generally take the following approaches: using a general-purpose accelerator (i.e., GPU) or designing the specialized architecture for the specific models (i.e., ASICs). Some works [14, 22, 23, 34, 37, 39] propose techniques (i.e., pruning, compression, and approximation) to make the models lightweight and design architecture exploiting their techniques as well. However, those approaches cannot properly deal with the three challenges of NLP models.

3.3.1 GPU (General-purpose Accelerator). GPUs are the most commonly used accelerators for various DNN models. Thanks to a convenient framework and highly parallel architecture, GPUs can support various models and achieve high performance for big models or models with a large batch size [24]. However, GPUs cannot well handle the NLP models in the single batch environment. Figure 6 shows the average utilization of tensor cores (matrix compute units) and CUDA Cores (vector compute units) in V100 for different versions of BERT, assuming a single batch environment. We observe that GPUs are severely underutilized in a single batch environment even for a large model (Tensor cores: 5.55%, CUDA Cores: 0.21% for BERT-LARGE). In addition, utilization dramatically decreases as the model's size diminishes. For example, BERT-BASE has 4.23% and 0.22% effective utilization for tensor cores and CUDA Cores, respectively, while BERT-TINY has only 0.15% and 0.05%.

Also, GPUs cannot run vector operations efficiently, as they mainly focus on matrix operations. Figure 7 is the change in utilizations over time when running single batch BERT-LARGE on V100. In Figure 7, the latency of vector operations (CUDA Cores) is exposed and takes a larger portion than the gemm operations (Tensor cores). Also, utilization of vector operations is very low (0.78% at maximum). This severe underutilization is due to the working mechanism of GPU, where data go down into memory between every gemm and vector operation, resulting in frequent memory access. In conclusion, GPUs are not adequate for running NLP models in a single batch environment.

3.3.2 ASICs and FPGA. Previous works [12, 13] implement ASICs for specific models. ASICs usually show a great performance for a target model and configuration. They are exploiting unique characteristics of the target model and have the best setting for the target configurations. However, ASICs fail to resolve challenge 2 (i.e., diversity and irregularity of dimensions) and challenge 3 (i.e., heterogeneous vector operations) of NLP models, as they are fixed architectures.

First, ASICs optimized for a specific model and configuration may not show good performance for the same models with different parameter configurations. However, there are diverse parameter configurations for the same NLP models. Also, we can arbitrarily change parameters as in previous

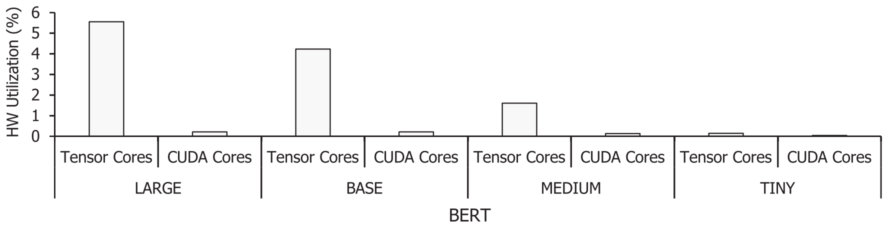


Fig. 6. Tesla V100's (Tensor cores and CUDA Cores) utilization on different versions of BERT.

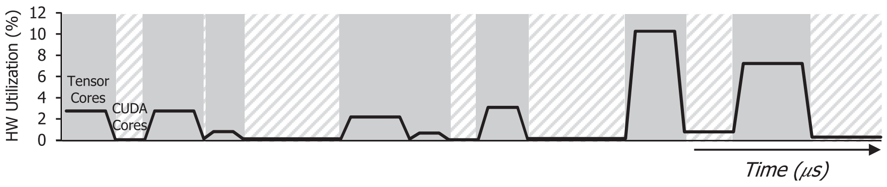


Fig. 7. The utilization of V100 on BERT-LARGE over time. The gray boxes are gemm operations using tensor cores, and the shaded boxes are vector operations using CUDA Cores.

studies [32, 33]. For example, A^3 [13] focuses on the attention operations of NLP models, making attention-specialized units. A^3 sets one of its specialized units' size as the same as d_{head} size (64) of BERT-LARGE/BASE. However, if we change d_{head} size to 32 [33], then utilization will be cut in half.

Also, ASICs may fail to run NLP models that they do not cover. For example, ASICs optimized for LSTM cannot run the attention-based NLP models like BERT due to the absence of required units (e.g., transpose, reduction, sqrt). They can run the models by adding a few units, but it will be inefficient without optimization for new models. However, changing existing architecture or designing new architecture for every upcoming model is impossible, because new models are released at such a rapid pace, resulting in excessive engineering costs.

There are some previous works that exploit FPGAs for accelerating NLP workloads [11, 26]. But they also focus on accelerating specific NLP models (e.g., LSTM) and fail to solve the challenges of NLP models. However, FPGAs have many advantages in running DNN workloads. First, FPGAs have high reconfigurability. Also, FPGAs support various data precision (e.g., FP32, INT8) and some products have HBM on chip [7]. New FPGA products with many operators and large on-chip memory for DNN workloads are actively entering the market recently [7].

3.3.3 Hardware-software Co-Design. There are works that propose software techniques and design the hadrware exploiting their techniques. References [22, 23, 34, 37] make the models lightweight by proposing some techniques such as pruning, compression, and approximation. Meanwhile, Reference [14] exploits sparsity in the model. Those works also design hardware that exploits their techniques. However, since they are ASICs, they cannot solve challenge 2 and challenge 3 as mentioned above. Also, their hardware design loses generality, because it is dependent on the techniques that they propose. For example, SpAtten [37] applies on-fly pruning to matrix operations in the models and designs hardware to support on-fly pruning. Therefore, its hardware is dependent on its pruning technique. Also, there is an overhead to support on-fly pruning. ELSA [14] adopts approximate self-attention. Its hardware is also dependent on its technique. In addition, since its approximation technique only targets self-attention, it cannot be applied to an NLP model without self-attention (e.g., LSTM).

Table 2. Previous Works and Their Limitation in Solving the Challenges of NLP Models

	Previous Works	Challenge 1	Challenge 2	Challenge 3
Acclerator for NLP	Brainwave [11], NPU [26]	O	X	X
	A^3 [13], SpAtten [37], ELSA [14, 22, 23, 34])	△	X	X
	ATT [12]	O	△	X

3.4 Solutions

We summarize the previous works and their weaknesses in Table 2. As we have already mentioned, some previous works [13] cannot efficiently handle the diversity in dimensions (challenge 2), neither do they deal with overhead and heterogeneity of vector operations (challenge 1, challenge 3) in NLP models. Meanwhile, other works [11, 12, 14, 22, 23, 26, 34, 37, 39] relieve the overhead of vector operations by pipelining, solving challenge 1. However, they cannot still cope with challenge 2 and challenge 3.

The main reason these works cannot solve the challenge is that they are fixed architectures that are limited to specific models. **Therefore, to solve the challenges, we propose FlexRun, the end-to-end solution that implements efficient modular architectures. In this modular architecture approach, the architecture composes of pre-defined basic modules so we can flexibly reconfigure the architecture adaptively to the target model.** To this end, we choose FPGAs as our HW platform, because they have high reconfigurability.

FlexRun includes three main features. First feature is flexible base architecture template that consists of pre-defined parameterized modules (i.e., FlexRun:Architecture). The next feature is a design space and design space exploration algorithm to find the best modules and parameter set for the target models (i.e., FlexRun:Algorithm). Finally, the last feature is an automatic tool that automates the two steps, finding the best architecture according to the inputs and implementing architecture (i.e., FlexRun:Automation). We will explain our FlexRun in detail in Section 4. For FlexRun to achieve high performance for NLP models, we set the design goals as follows:

Design Goals

- Devise flexible architecture template that can solve the three challenges discussed previously (FlexRun:Architecture).
- Define design space for handling challenges 1, 3 and devise an algorithm to efficiently search the design space for the given inputs (FlexRun:Algorithm).
- Automate the whole process, from searching design space to implementing architecture, for ease of use (FlexRun:Automation).

4 FLEXRUN

4.1 Overview

In this subsection, we will briefly explain how FlexRun works and how it solves the challenges of accelerating the low-batch NLP inference. FlexRun is an end-to-end solution that implements a fast and flexible modular architecture for NLP models. Figure 8 shows the workflow of FlexRun. FlexRun has three main schemes, FlexRun:Architecture, FlexRun:Algorithm, and FlexRun:Automation. First, FlexRun:Architecture is a base architecture template that consists of parameterized basic modules (FlexRun:Library). Next, FlexRun:Algorithm finds the best set of modules and their parameters for the given model and FPGA spec (① in Figure 8). Finally, a FlexRun:Automation reconfigures and implements the architecture template given by FlexRun:Architecture according to the FlexRun:Algorithm's results (② & ③ in Figure 8). Also,

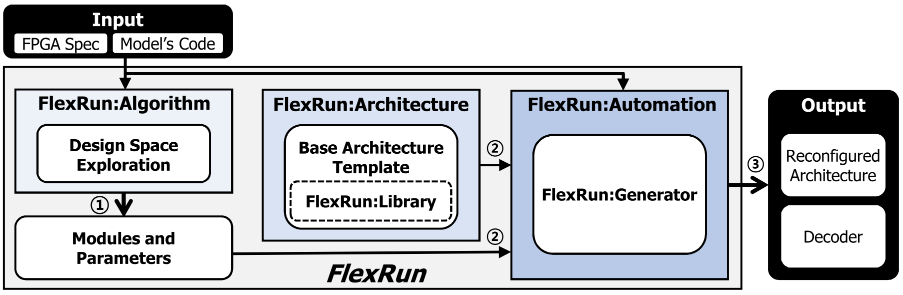


Fig. 8. End-to-end workflow of FlexRun.

the FlexRun:Automation generates the new decoder codes so the model's code is decoded for the reconfigured architecture.

FlexRun can solve the three challenges of accelerating NLP models. First, to solve challenge 1 (i.e., overhead of vector operations), FlexRun has a pipelined structure and a reconfigurable vector unit. The operators in FlexRun (between matrix unit and vector unit, between vector operators) are pipelined. Therefore, it is suitable for a model with a large overhead of vector operations. In addition, since the vector unit of FlexRun can be reconfigured according to the model, FlexRun can further reduce the overhead. Second, FlexRun is suitable for solving challenge 2 (i.e., diversity and irregularity of dimensions) because it is a structure that can tune the parameters of the matrix unit to dimensions of the matrix operations. Last, for challenge 3 (i.e., heterogeneous vector operations), FlexRun has a reconfigurable vector unit.

4.2 FlexRun:Architecture

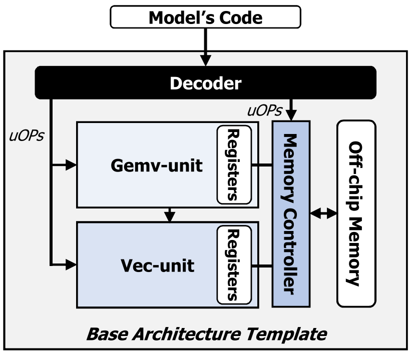
FlexRun:Architecture is our base architecture template made of parameterized basic modules. We devise a FlexRun:Architecture, considering the three challenges of NLP models.

In making our base architecture template, we refer to NPU [26]. We adopt the structure of Gemv-unit from NPU. However, the existing structure cannot handle various dimensions efficiently, so we change the three parameters of Gemv-unit to be reconfigurable. In addition, we use the direct data paths between Gemv-unit and Vec-unit like NPU. However, since the Vec-unit is fixed in the existing structure, the overhead of vector operations occurs, and heterogeneity cannot be handled. To solve this, we make Vec-unit reconfigurable. Figure 9(a) illustrates FlexRun's base architecture template. The base architecture template consists of three parts. First, there are two compute units: Gemv-unit and Vec-unit. Gemv-unit is a highly parallel compute unit for gemv operations. Vec-unit is a compute unit for vector operations, which can be made of any combination of basic vector operators (e.g., add, exp). Gemv-unit and Vec-unit have their own registers to store the weights and intermediate data. Second, there is a memory controller that prefetches required weights from off-chip memory to registers. Last, there is a decoder that decodes the model's code to micro-operations (uOps) for the memory controller and compute units. The detailed structure and working mechanism of the base architecture template are explained in the following subsections.

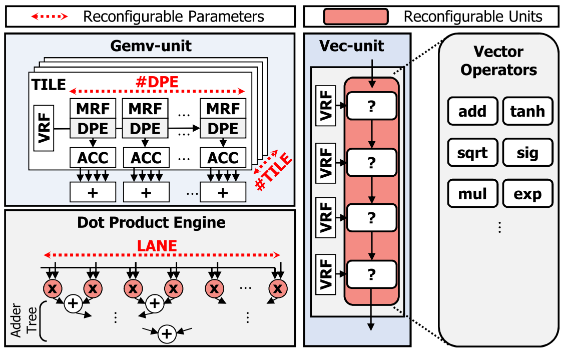
4.2.1 *Structure of FlexRun:Architecture.*

Figure 9(b) is a detailed figure for the compute units of FlexRun's base architecture template.

Gemv-unit: Gemv-unit composes of multiple SIMD arithmetic units for vector-matrix multiplications as in Figure 9(b). Gemv-unit is a highly parallel architecture so it fits for NLP workloads that have many matrix operations. Also, Gemv-unit computes in vector-matrix granularity, which



(a) FlexRun's base architecture template.



(b) Compute units of FlexRun's base architecture template.

Fig. 9. FlexRun's base architecture template and details of compute units (Gemv-unit and Vec-unit).

is good for NLP workloads that have dependencies between the inputs (decoder structures such as LSTM and GPT2) in a single batch environment.

In Figure 9(b), Gemv-unit has multiple TILEs that split a matrix into sub-column blocks, and each TILE has several **DPEs (Dot Product Engines)** and **ACCs (Accumulators)**. Each DPE executes the same number of element-wise multiplications as the size of LANEs. In our architecture template, the number of TILE, DPE, and size of LANE is reconfigurable. Therefore, through design space exploration, we can reconfigure these parameters adaptively to the target operations' sizes, solving challenge 2 (i.e., wide ranges of dimensions and irregular matrix operations).

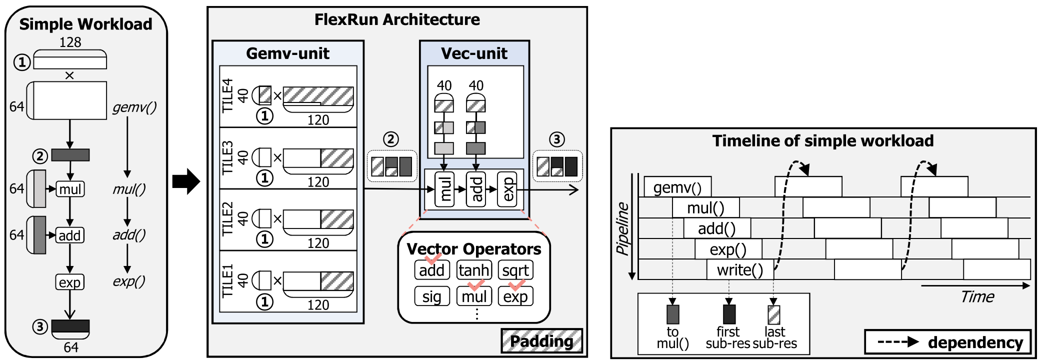
Also, Gemv-unit and Vec-unit as well as each vector operator are deeply pipelined. Therefore, most vector operations' latency is hidden by gemv and other vector operations. In addition, thanks to the direct path between Gemv-unit and Vec-unit, there is no unnecessary memory access between gemv and vector operations (Figure 10(b)), mitigating challenge 1 (i.e., vector operations' overhead).

Vec-unit: Vec-unit executes vector operations of size VEC_LANE at once. In our template, we set the size of VEC_LANE equal to the Gemv-unit's LANE size. Also, there are some additional operators (i.e., reduction, exp, gelu) compared to Reference [26] to support attention-based NLP models. Through the direct data path between Gemv-unit and Vec-unit, the Vec-unit can start independent instruction upon receiving the first sub-block results from the Gemv-unit.

Previous works [26] have fixed types, numbers, and order of vector operators in their vector compute units. However, as shown in Figure 9(b), we remain Vec-unit as an empty box that can be made of any combination of basic vector operators in FlexRun:Library. In this way, we can efficiently deal with challenge 3 (i.e., heterogeneity of vector operations).

Memory and Datapath: Gemv-unit and Vec-unit have separate register files, **MRF (Matrix register file)** and **VRF (Vector register file)**, for decoupled execution. Previous works [11, 26] that utilize the persistent-AI approach keep all weights in on-chip memory (MRFs). However, some NLP models (e.g., BERT-LARGE) cannot hold their whole weights in on-chip memory due to their excessive size. Therefore, we add new datapaths that connect memory controller and MRFs to fetch weights from off-chip memory to MRFs. These datapaths are used when we use the results of Gemv-unit or Vec-unit as the vector input of Gemv-unit, too. Also, we place a matrix transpose unit in the MRF/VRF write-back path.

ISA and Decoder: FlexRun generates an instruction-based accelerator. We adopt an instruction-based accelerator for generality so the user can use FlexRun for their own customized models (e.g., changing parameters or vector operations). FlexRun's ISA extends the NPU's ISA [26] to support



(a) Simple workload and execution of workload in FlexRun. (b) Timeline of simple workload's execution.

Fig. 10. (a) shows a simple workload and FlexRun's executions of the workload. (b) is the timeline graph of the workload when it is repeated three times with dependencies.

new operations of attention-based NLP models (i.e., reduction, exp, gelu, transpose). FlexRun's ISA is architecture-independent for programmability. Instead, the decoder decodes the model's codes into multiple uOps adaptively to the reconfigured architecture. When we reconfigure the architecture, we remake the decoder to fit the new architecture. We will provide decoder examples in Section 4.5.

FlexRun:Library: In FlexRun:Library, there are basic modules of our template; TILE of Gemv-unit and vector operators for Vec-unit. These basic modules are parameterized and modular so we can configure and merge them adaptively to the target model.

4.2.2 Working Mechanism of FlexRun:Architecture. As an example, we assume a simple workload as Figure 10(a). The workload consists of one gemv operation and following vector operations; gemv()-mul()-add()-exp(). The size of gemv operation is $(1 \times 128) \times (64 \times 128)^T$. For the example workload, we configure the FlexRun's architecture as follows: In the case of Gemv-unit, we assume (#TILE, #DPE, LANE size) as (4, 120, 40). Also, we set the Vec-unit's structure as [mul-add-exp].

Now, we explain how our architecture handles the example. First, in Figure 10(a), we add paddings to the input vector (①) and the column of weight matrix so it would be the multiple of LANE size (128→160). Then, the input vector is distributed to each TILE (①→①'). All TILEs have the same size of vectors, 40. But, 32 elements of TILE 4's input vector are zero due to the padding. Also, we add paddings to the row of weight matrix to be the multiple of #DPE (64→120). Next, the DPEs and ACCs perform matrix-vector multiplications of LANE size. Since the size of Gemv-unit's ouput vector (②) is 120, Gemv-unit produces three (120/40) sub-vectors of LANE size (②→②'). These sub-vectors (②') are fed into the Vec-unit as soon as they come out from Gemv-unit. Finally, the result vector (③, ③') comes out from the Vec-unit. As shown in the figure, a lot of fragmentation occurs, as the size of matrix and vector does not match the (#TILE, #DPE, LANE size).

Figure 10(b) shows the timeline and pipeline graph of the simple workload when it is repeated three times with dependencies. Gemv-unit and Vec-unit are pipelined so the output sub-vectors of Gemv-unit directly go to the Vec-unit even though Gemv-unit does not complete its executions. These pipelined execution helps hide the latency of vector operations with gemv operations. Each vector operator is also deeply pipelined, hiding each other's latency. However, due to the dependencies, the next input's execution cannot start until the previous one is finished. If there is no dependency, then the next input can start processing as soon as the compute unit becomes free.

Table 3. Design Space of FlexRun

Unit	Design Space	Explanation
Gemv-unit	#TILE	The number of TILE
	#DPE	The number of DPE
	LANE size	The size of LANE
Vec-unit	Type	Vector operators to choose
	Order	Vector operators' placement order
	Number	The number of same vector operators to choose

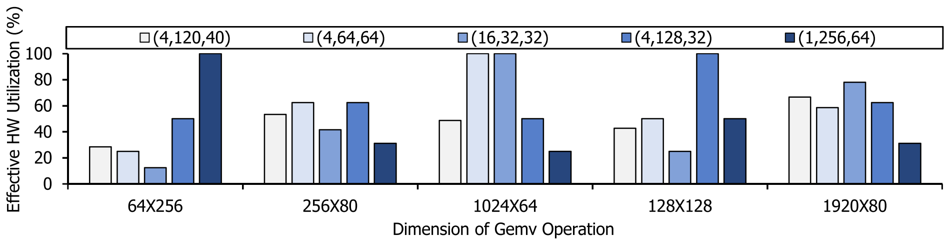


Fig. 11. The effective HW utilization of Gemv-unit for various dimensions of gemv operations. The legend is the reconfigurable parameters of Gemv-unit: (#TILE, #DPE, LANE size). To be specific, (64 × 256) in the x-axis indicates the gemv operation that the vector size is (1 × 64) and the matrix size is (64 × 256).

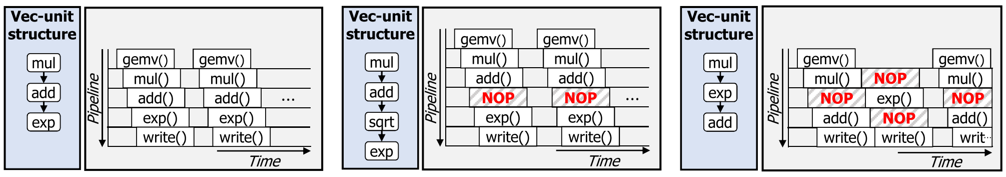
4.3 FlexRun:Algorithm - Design Space

Table 3 shows FlexRun's design space. FlexRun aims to find the best configuration in the design space for the given model and FPGA spec. In case of the Gemv-unit, three parameters (#TILE, #DPE, LANE size) constitute our design space. These parameters cover all the gemv dimensions, (1 × N) × (N × K).

For the Vec-unit, the types, order, and number of the basic vector operators constitute design space. Some may think that the flexible vector compute units are unnecessary, as pipelining alleviates the overhead of vector operations. However, if the vector compute units are not properly configured, the overhead of vector operations is exposed even with the pipelining.

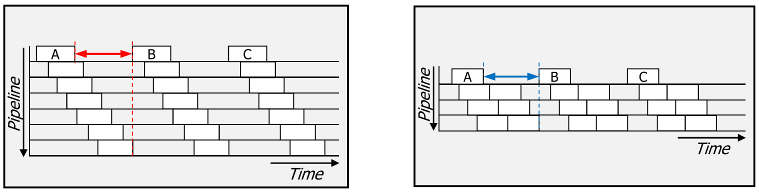
In the following contents, we will show how our design parameters choice affects the performance of the NLP models and solves the three challenges.

4.3.1 Design Space of Gemv-unit: (#TILE, #DPE, LANE Size). Figure 11 shows the HW utilization for gemv operations with various dimensions, changing the three Gemv-unit's parameters. The x-axis is the dimension of gemv operation, and the y-axis is the effective utilization of Gemv-unit. Also, the legend is the combination of (#TILE, #DPE, LANE size). All combinations satisfy the same maximum resource limitation. In the figure, each gemv operation has different utilization according to the (#TILE, #DPE, LANE size). For example, in the case of (64 × 256), it achieves 100% effective utilization when the (#TILE, #DPE, LANE size) is (1, 256, 64), while (16, 32, 32) reduces the utilization by 87.5%. Also, there is no combination of (#TILE, #DPE, LANE size) that works best for all gemv operations. For example, (1, 256, 64) works best for (64 × 256), but has the worst utilization for (1,024 × 64). From the experiments, we conclude that fixed-size matrix compute units cannot handle a wide range of dimensions as well as irregular matrix operations. Therefore, by configuring (#TILE, #DPE, LANE size) adaptively to the model, we can get the highest possible utilization, resolving challenge 2.



(a) Right types & order. (b) Wrong types but right order. (c) Right types but wrong order.

Fig. 12. Timeline and pipeline graphs of simple workload's executions according to Vec-unit's structures.



(a) #operators ↑ (pipeline's depth ↑). (b) #operators ↓ (pipeline's depth ↓).

Fig. 13. Tradeoff according to the number of vector operators. Arrows indicate the exposed pipeline's depth.

4.3.2 Design Space of Vec-unit: Types, Order, and Number of Basic Vector Operators. When we design vector compute units, we need to consider the types, order, and the number of vector operators. We will explain how Vec-unit's design affects the performance by revisiting the simple workload, gemv()-mul()-add()-exp(). The simple workload needs three operators: mul, add, and exp. Figure 12 shows the timeline graphs of simple workload for the three cases: right types & order (Figure 12(a)), wrong types but right order (Figure 12(b)), and right types but wrong order (Figure 12(c)).

First, we show how the types and order affect the performance. Figure 12(a) is when the types and order are perfectly matched to the given model. However, if there are any unnecessary operators, then it occurs underutilization like Figure 12(b). Also, in the case when the order is not optimal, the data may go through the pipeline multiple times to complete the operations, as in Figure 12(c).

Next, we show the impact of the number of operators. Depending on the number of vector operators, which determines the pipeline's depth, a tradeoff occurs like Figure 13. In the case where Vec-unit has many vector operators, the input data can be processed in one execution, as shown in Figure 13(a). However, in Figure 13(a), the pipeline gets deeper and the overhead of the exposed pipeline's depth grows. Otherwise, if Vec-unit has fewer vector operators like Figure 13(b), then the data has to go through the pipeline several times to finish execution. But the pipeline gets shorter and the overhead of the exposed pipeline's depth is reduced.

Figure 14 shows the performance of GPT2-MEDIUM on various Vec-unit's structures. The x-axis is the Vec-unit's structure and the y-axis is speedup normalized to the last structure. The last structure in Figure 14, the one with the random ordering, has the worst performance. The first and third structures have proper ordering, but they show lower performance than the second one, as they have too many operators. The second structure, which has optimal order and number of operators, has the best performance. From the result, we conclude that by reconfiguring the Vec-unit adaptively to the given model, we can deal with challenge 1 and challenge 3.

Some may think it is more efficient to put all the vector operators needed for the NLP models in Vec-unit and make all-to-all connections. However, there are two problems. First, interconnect overheads will be too expensive, as complex models need many different operators. Second, simple

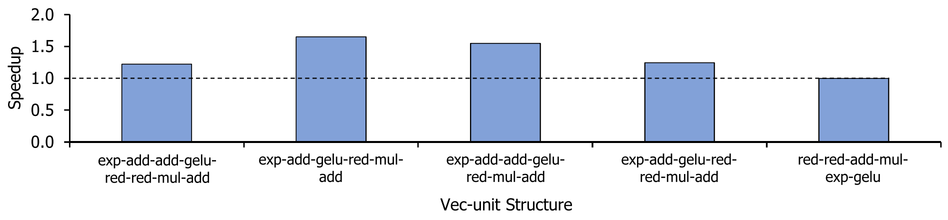


Fig. 14. The GPT2-MEDIUM's relative performance improvement according to various Vec-unit's structures. The performance improvement is normalized to the last feature. Also, *red* means reduction operations. We get the results using our analytical model for FlexRun. Refer to Section 4.4 for details.

ALGORITHM 1: Vec-unit Reconstruction Algorithm.

```
input     : $model_{code}$, ($\#TILE$, $\#DPE$, $LANE\ size$).
/* Get vector-operation-sequences in the model.                                   */
{$vec\_seq_0, \ldots, vec\_seq_k$} = $get\_vector\_op\_sequences(model_{code})$
/* Find Shortest Common Sequence (SCS) for given vector sequences, {$vec\_seq_0, vec\_seq_1, \ldots, vec\_seq_k$}.   */
{$SCS_0, \ldots, SCS_m$} = $Find\_SCS(vec\_seq_0, vec\_seq_1, \ldots, vec\_seq_k)$
$min\_lat = total\_lat(SCS_0, model_{code}, (\#TILE, \#DPE, LANE\ size))$
$optimal\_structure = SCS_0$
for $SCS$ in {$SCS_1, \ldots, SCS_m$} do
    /* Get all possible subsequences of SCS by removing duplicate operators.      */
    {$SCS\_subseq_0, \ldots, SCS\_subseq_n$} = $Find\_SCS\_subsequences(SCS)$
    /* Find SCS_subsequence that gives the minimum total latency for the inputs.  */
    for $arch$ in {$SCS\_subseq_0, \ldots, SCS\_subseq_n$} do
        $temp = total\_lat(arch, model_{code}, (\#TILE, \#DPE, LANE\ size))$
        if $temp < min\_lat$ then
            $min\_lat = temp$
            $optimal\_structure = arch$
    end
end
return $optimal\_structure$
```

models like SRNNs suffer from underutilization, because they do not use most vector operators. Also, future models may require new types of operators, which further increases the overhead.

4.4 FlexRun:Algorithm - Design Space Exploration

To find the best design for the model, we take a greedy approach. First, we find the most optimal Gemv-unit structure. After that, we find the Vec-unit structure, which shows the highest performance for the optimal Gemv-unit structure found. To this end, we introduce two algorithms: Gemv-unit Rearrangement for Gemv-unit and Vec-unit Reconstruction for Vec-unit. For design space exploration, we build our own analytical model for FlexRun like Reference [31]. The model gets the FlexRun's parameters and model's code as inputs and measures the execution latency.

4.4.1 Gemv-unit Rearrangement. Gemv-unit Rearrangement gets the model's code and FPGA spec (i.e., BRAM, DSP, LUT, Memory BW) as inputs. Then, it searches all combinations of (#TILE, #DPE, LANE size) that satisfy the resource limitation. Among the found combinations, the algorithm selects the top-k sets that have the smallest total gemv/gemm latency of the model. In experiments, results of Gemv-unit Rearrangement for FPGAs on the current market are available in seconds.

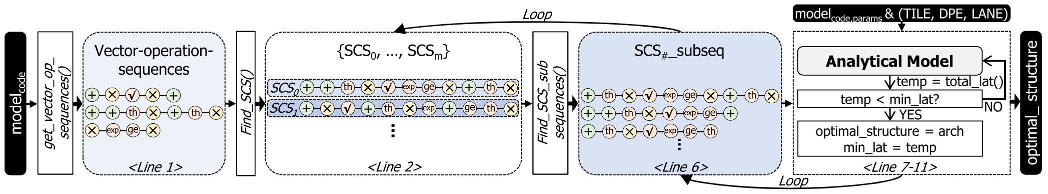
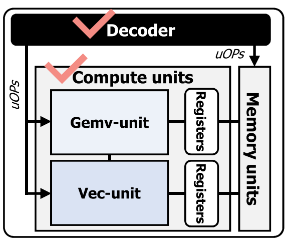
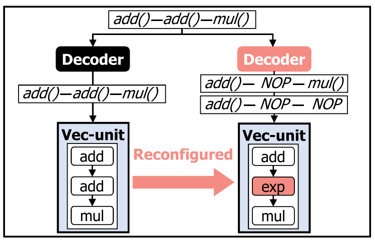


Fig. 15. Visualization for each step of Vec-unit Reconstruction algorithm.



(a) Components to be reconfigured in the base architecture template.



(b) Example for how decoder works according to the architectures.

Fig. 16. FlexRun: Generator.

4.4.2 Vec-unit Reconstruction. Vec-unit Reconstruction gets two inputs, the model's code and results of Gemv-unit Rearrangement. For target model and configurations, Vec-unit Reconstruction finds a Vec-unit's structure with minimum total latency. Vec-unit Reconstruction uses the **Shortest Common Sequence (SCS)** algorithm. SCS is the shortest sequence that includes all the vector-operation-sequences in the model while keeping the original order of each sequence.

Algorithm 1 shows each step of Vec-unit Reconstruction. In Algorithm 1, first, we extract the list of vector-operation-sequences (vec_seq_i) from the model's code: line1. Then, we find all SCSs for given vector-operation-sequences: line2. Next, for every SCS in the list, we repeat lines 6–12. In line 6, we derive all possible sub-sequences of the SCS (SCS_subseq_i) by removing duplicated elements. Last, we find the sub-sequence that gives the minimum total execution latency for the model ($optimal\_structure$), using an analytical model, $total\_lat()$: lines 7–11. For easy understanding, we visualize each step of the algorithm in Figure 15.

4.5 FlexRun:Automation

FlexRun:Automation is an automatic tool of FlexRun to make the reconfiguration process automatic. As in Figure 8, FlexRun:Automation receives the model's code and FPGA spec. Then, it finds the optimal configurations (i.e., (#TILE, #DPE, LANE size) and Vec-unit's structure) for the given model using the two functions in FlexRun:Algorithm. Finally, FlexRun:Automation reconfigures the base architecture template according to the configuration.

4.5.1 FlexRun: Generators. FlexRun:Automation contains two additional features, compute units generators and decoder generators (FlexRun:Generator). When we reconfigure our base architecture template, two components marked in Figure 16(a) should be modified. One is the whole compute processor, including compute units, registers, and interconnects. The other is the decoder, which decodes model's codes into multiple uOPs for the new architecture.

Figure 16(b) shows how the decoder decodes the same codes when the architecture changes. For simplicity, we change the structure of Vec-unit only. In the example, we assume the model performs addition, addition, and multiplication on a vector in sequence. Then, we can write the model's code as add()-add()-mul() using FlexRun's ISA. As shown in Figure 16(b), the first architecture has two adder operators and multiplication operators in order. Therefore, the model's code is decoded into following uOps: add()-add()-mul(). However, the second architecture has an exponentiation operator in the middle. So, to execute the code, the vector goes through Vec-unit twice. In the first pass, no operation occurs in the second (exp) and third (mul) operators. And in the second pass, no operation is performed on the second (exp) operator. So the model's code is decoded into the following uOps: add()-NOP-NOP for the first pass and add()-NOP-mul() for the second pass.

5 IMPLEMENTATION

5.1 FlexRun

5.1.1 FlexRun. To implement basic modules in FlexRun:Library (TILE of Gemv-unit and vector operators in Vec-unit), decoder, and memory units, we use C-based Vivado **High-Level Synthesis (HLS)**. First, in TILE, we make the number of DPEs, LANE size, and accumulators as reconfigurable. Also, we set the registers' size adaptively to the input model and configurations. Next, we code each vector operator separately so we can make any combination of operators. Each vector operator has a parameter named VEC_LANE (basic unit of processing), which has the same size as the Gemv-unit's LANE size.

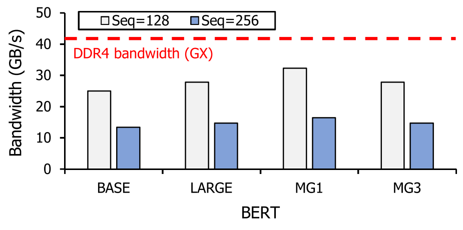
We use Python to implement two algorithms in FlexRun:Algorithm. First, Gemv-unit Rearrangement gets the model's code, model descriptions (e.g., parameters, vector operations), and FPGA spec as inputs. Then it lists out all combinations of (#TILE, #DPE, LANE's size) satisfying the resource limitation. The analytical model gets these lists and finds the top-k combinations that have the smallest matrix operation's latency for the target model. In the evaluation, we use 3 as the value of k. Second, Vec-unit Reconstruction receives outputs of Gemv-unit Rearrangement as well as model's code and descriptions. Following Algorithm 1, the function finds the best set of Vec-unit's structure and (#TILE, #DPE, LANE size). Put all together, FlexRun:Automation gets the model's code and FPGA spec. Then, it finds the optimal configurations using FlexRun:Algorithm. Finally, two generator fucntions make the compute processor and decoder according to the configuration.

5.1.2 Memory. Since data reuse is limited in a single batch environment, high memory bandwidth is required to avoid the memory bottleneck. When the model's size is small enough to be stored in on-chip memory, the persistent AI approach used by previous works [11, 26] can address the memory bottleneck issues. However, this approach cannot be applied to NLP models with large parameters. In this work, we implement layer-wise double buffering in on-board DDR4 and HBM to hide the memory overhead. With this technique, the memory controller prefetches the weight matrices of the next layer while the current layer is computed.

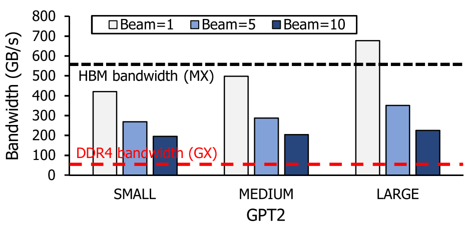
Figure 17 shows the maximum memory bandwidth requirement of our target NLP models when adopting the layer-wise double-buffering scheme. In the figure, we compare the bandwidth requirement of the models with the off-chip memory's bandwidth of two FPGAs. We target Intel's Stratix 10 GX with DDR4 and Stratix 10 MX with HBM. In Figure 17(a), BERT with large parameters can be executed on both GX and MX. However, GPT2 should be executed on an MX with HBM to avoid memory bottleneck, like Figure 17(b). Therefore, we use GX for BERT and MX for GPT2 in evaluation.

5.2 Workloads and Experimental Setup

5.2.1 Workloads. Our target workloads and their configurations are in Table 4. We choose the workloads by the following criteria: First, we evaluate the attention-based NLP models that have



(a) Comparison between bandwidth requirement of BERT and DDR4's bandwidth.



(b) Comparison between bandwidth requirement of GPT2 and HBM's bandwidth.

Fig. 17. Comparison of models' bandwidth requirement and off-chip memory bandwidth of DDR4 and HBM.

Table 4. NLP Models and Their Configurations Included in the Workloads

Model	Scale	d_{hidden}	d_{head}	$\#head$	d_{ff}	$beamwidth$
LSTM	-	1,024	-	-	-	-
BERT	TINY	128	64	2	512	-
	MEDIUM	512	64	8	2,048	-
	BASE	768	64	12	3,072	-
	LARGE	1,024	64	16	4,096	-
	MG1	1,280	64	16	5,120	-
	MG3	3,072	64	24	3,072	-
GPT2	TINY	768	64	12	3,072	5, 10, 40
	MEDIUM	1,024	64	16	4,096	5, 10, 40
	LARGE	1,280	80	16	5,120	5, 10, 40

complex vector operations so we can show how FlexRun reduces the overhead of vector operations (challenge 1). Next, we show how FlexRun deals with the heterogeneity of vector operations by comparing the performance trends of the three models (challenge 3). Last, by covering a wide variety of parameter scales, we show that FlexRun can cope with a wide range of dimensions and irregular matrix operations (challenge 2).

5.2.2 *Experimental Setup.* As the baseline, we use GPU and our base architecture template. For GPU, we use a Tesla V100, the biggest available AI-targeted GPU using the same process technology as S10 GS and MX. For V100, we enable tensor cores [1] and set frquency to 1,350 MHz. We use official TensorFlow implementations of NLP models from NVIDIA and OpenAI [2]. We use FP16 for GPU. When analyzing the results of GPU, we exploit the Nsight Systems [3], which is an official performance analysis tool of NVIDIA. In the case of the second baseline, we refer to NPU [26] for setting the design space parameters ((#TILE, #DPE, LANE size) and Vec-unit's structure). The detailed configurations are specified in Table 5. For the rest of the article, we will simply call the second baseline the Baseline. For FlexRun's evaluation, we use two FPGAs, Intel's Stratix 10 GX and MX [9]. Also, FlexRun supports an 8-bit integer data type. We refer to Q8BERT [40], which proposes 8-bit quantization techniques that maintain 99% accuracy compared to the FP32 version of BERT.

6 EVALUATION

We evaluate our schemes using a cycle-accurate simulator to measure the scalability of FlexRun, considering the trends of FPGAs with increasing resources. We carefully validate our simulator as

Table 5. Configurations of Baseline

FPGA	(#TILE, #DPE, LANE size)	Vec-unit's structure	Frequency
Stratix 10 GX	(4, 120, 40)	red-add-act-mul	275 MHz
Stratix 10 MX	(4, 80, 40)	red-add-act-mul	290 MHz

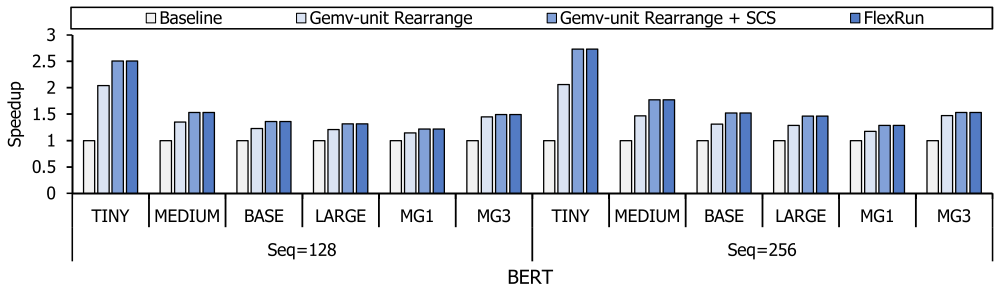


Fig. 18. Speedup of FlexRun for BERT normalized to the Baseline.

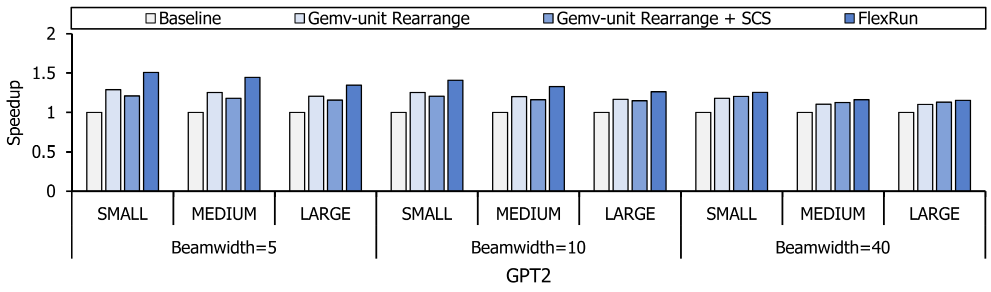


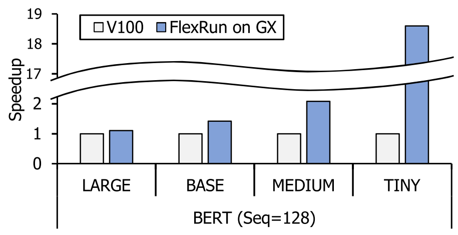
Fig. 19. Speedup of FlexRun for GPT2 normalized to the Baseline.

follows: First, we implement SW-based NPU-like architecture and compare its RNN/LSTM performance against Intel's pre-validated Stratix 10 GX and MX FPGA implementation in Reference [26]. The errors between our simulator and FPGA implementation are under 0.1% for various parameter settings. Then, we add FlexRun's specific features (e.g., transpose unit) on top of SW architecture to make our FlexRun base architecture template.

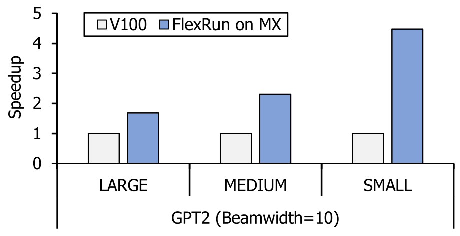
6.1 Performance Improvement of FlexRun Compared to the Baseline

We first measure the speedup of FlexRun compared to the Baseline. Figures 18 and 19 show the comparison results for BERT and GPT2, respectively. The x-axis is the parameter configuration and the y-axis is the speedup. The legend is the applied optimization schemes. We apply three optimizations of FlexRun one-by-one. The first legend is the Baseline. The second legend (Gemv-unit Rearrange) indicates the case that we only apply Gemv-unit Rearrangement. The third legend (Gemv-unit Rearrange + SCS) is the result of changing the Vec-unit's structure to SCS. The last legend (FlexRun) is the case in which we apply both Gemv-unit Rearrangement and Vec-unit Reconstruction. Note that FlexRun does not use SCS, but uses the optimal structure found while removing redundant operators from SCS through Vec-unit Reconstruction. Also, for Gemv-unit Rearrange + SCS baseline, the one showing the best performance improvement among SCS was used.

Figure 18 is the results for BERT with two input sequence sizes: 128, and 256. First, Gemv-unit Rearrangement (second legend) achieves 1.36× speedup, on average. The scheme gives more benefit



(a) Speedup of FlexRun on GX compared to V100 for BERT.



(b) Speedup of FlexRun on MX compared to V100 for GPT2.

Fig. 20. Comparison of FlexRun and V100 for BERT and GPT2.

to the small models like BERT-TINY (2.05× speedup), as they have a higher chance of underutilization. Also, it gets higher speedup, 1.46× for BERT-MG3, which has the largest irregularity in matrix operations. Next, Vec-unit Reconstruction (last legend) brings 1.17× additional speedup, on average. Note that for BERT, whether Vec-unit has an SCS structure (third legend) or an optimal structure (last legend) does not make a difference in performance. The overall average speedup is 1.59×.

Figure 19 shows the FlexRun's performance improvement for GPT2 with three beamwidths, 5, 10, and 40. In the case of GPT2, Gemv-unit Rearrangement brings 1.19× speedup, on average. Also, Vec-unit Reconstruction gets 1.1× additional speedup, so the overall average speedup is 1.31×. Unlike BERT, SCS structure degrades the performance in GPT2. These differences arise from the presence of dependencies between the inputs. If there are dependencies between the inputs, the pipeline's depth affects the performance. Therefore, the SCS structure harms the performance of GPT2. Also, when we increase the beamwidth of GPT2, the performance degradation of SCS decreases as the inputs without dependencies increase.

6.2 Comparison of FlexRun and GPU

In Figure 20, we compare the results of FlexRun with V100 for BERT and GPT2. The x-axis is a scale of the NLP models, and the y-axis is the speedup normalized to the latency of V100. FlexRun achieves 2.79× and 2.59× average performance improvements over V100 for BERT and GPT2, respectively. Especially, FlexRun shows higher performance for the small models, as small models suffer severe underutilization in GPU. Also, FlexRun usually gets higher speedup for BERT, because pipelining and Vec-unit Reconstruction give more benefits to the encoder structure. As the encoder structure does not have dependencies between the inputs, the pipeline depths are almost hidden.

6.3 Scalability of FlexRun

We check the scalability of FlexRun by doubling the compute and memory resources of FPGAs. We assume that the future generation of FPGAs has twice more compute and bandwidth resources than current FPGAs, GX, and MX. Figure 21 is the speedup of FlexRun on the future generation of FPGAs (second legend, 2× FPGA) and current FPGAs (first legend, GX/MX). The speedup of FlexRun on GX/MX is normalized to the same baseline of Figure 18. For FlexRun on 2× FPGA, we assume a new baseline with twice the #TILE and twice faster memory than the baseline of Figure 18.

The results show that FlexRun achieves scalable performance improvements as FPGA resources increase. Increasing the FPGA resources shows good performance gains in large models. In the case of BERT-LARGE, FlexRun attains 1.85× speedup on 2× FPGA while achieving 1.32× speedup on GX. On average, with twice the resources, FlexRun gets 2.2× and 1.99× speedup for BERT and GPT2, respectively. This is 1.46× and 1.69× additional speedup for FlexRun on GX and MX,

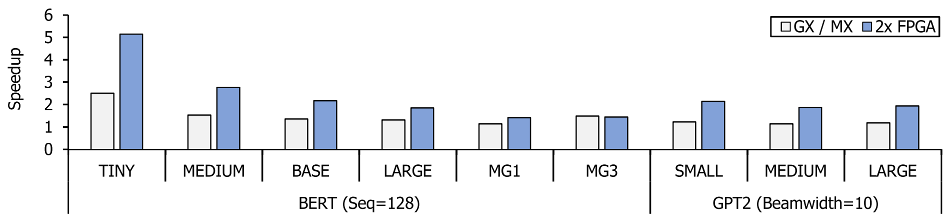


Fig. 21. Speedup of FlexRun for BERT and GPT2 on GX, MX, and FPGA with twice as many resources.

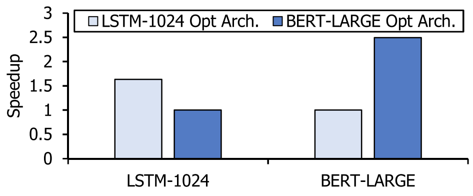


Fig. 22. Impact of FlexRun.

respectively. The FlexRun secures scalability, thanks to Gemv-unit Rearrangement. When the resources increase, the chances of underutilization in gemv compute unit grows due to fragmentation. So, for the future FPGAs, it is important to find the proper dimension of the gemv compute units, which is done by Gemv-unit Rearrangement in FlexRun.

6.4 Impact of FlexRun's Reconfigurability

Last, we show the impact of FlexRun's reconfigurability in Figure 22. We run BERT-LARGE and LSTM-1024 on architectures optimized for each model using FlexRun. In the legend, LSTM-1024 Opt Arch is the architecture optimized for LSTM-1024, and BERT-LARGE Opt Arch is the architecture for BERT-LARGE. The performance of the models is normalized to the slower one. In Figure 22, the performance is severely compromised when the model is executed on the architecture optimized for other model. In the BERT-LARGE case, the performance improves 2.83× when running on BERT-LARGE Opt Arch than on LSTM-1024 Opt Arch. For LSTM-1024, the performace improves by 1.63× on its optimized architecture.

7 RELATED WORK

There are works that accelerate NLP models, and they exploit different methods.

First, there are studies using the quantization method to accelerate NLP models and reduce models' sizes [39–41]. References [39, 41] suggest new quantization methods, expressing parameters of BERT with 3 bits. Also, Reference [40] presents BERT's parameters with eight bits, targeting INT8.

Similar to quantization, many studies apply pruning to NLP models. Reference [15] uses the weight pruning to reduce the size of LSTM and designs architecture for sparse LSTM. Also, Reference [38] proposes block-circulant matrices for weight matrices to resolve irregularities in the neural network in addition to pruning. For attention-based NLP models like BERT, Reference [25] proposes a structured pruning, while Reference [25] uses structured dropout.

References [18–21] utilize model partitioning for acceleration. Reference [19] defines parallelizable dimensions in DNNs and finds the best parallelization strategies for the target model. Reference [20] applies holistic model partitioning to all operations across attention-based NLP models. Reference [18] exploits model partitioning to accelerate large RNN models by enabling multi-FPGA executions.

Also, some works design accelerators for the NLP models. Reference [13] targets attention operations in NLP models and makes attention-specialized units. Reference [12] exploits PIM technologies to minimize the memory overhead of the NLP models. References [6, 11, 26] exploit FPGAs as their HW platforms to accelerate NLP models. However, none of those works can address three challenges of NLP models.

Last, References [36, 42, 43] take modular approach for accelerating DNNs. However, these works focus on Convolutional Neural Networks, rather than NLP models. Reference [36] suggests a modular accelerator generator for CNNs. References [42, 43] use FPGAs to build accelerators through their design space exploration tool in the cloud and edge-computing environments.

8 CONCLUSION

In this article, we propose FlexRun, an FPGA-based modular architecture approach to accelerate NLP models. When receiving input models, FlexRun reconfigures the architecture adaptively to the models. In evaluation, we get 2.69× and 1.44× performance improvement compared to V100 and Brainwave-like FPGA baseline, respectively.

REFERENCES

[1] 2017. Nvidia Tesla V100 GPU Architecture, The World's Most Advanced Data Center GPU. Retrieved from https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[2] 2021. DeepLearningExamples. Retrieved from https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow.

[3] 2021. Nsight Systems Release Notes. Retrieved from https://docs.nvidia.com/nsight-systems/ReleaseNotes/index.html.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 265–283.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[6] Andrew Boutros, Eriko Nurvitadhi, and Vaughn Betz. 2021. Specializing for efficiency: Customizing AI inference processors on FPGAs. In *International Conference on Microelectronics (ICM)*. 62–65. DOI: https://doi.org/10.1109/ICM52667.2021.9664938

[7] Andrew Boutros, Eriko Nurvitadhi, Rui Ma, Sergey Gribok, Zhipeng Zhao, James C. Hoe, Vaughn Betz, and Martin Langhammer. 2020. Beyond peak performance: Comparing the real performance of AI-optimized FPGAs and GPUs. In *International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 10–19.

[8] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. Retrieved from https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html.

[9] Lance Brown, Manish Deo, and Jeffrey Schulz. 2019. Intel® Stratix® 10 MX Devices with Samsung* HBM2 Solve the Memory Bandwidth Challenge.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805* (2018).

[11] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *International Symposium on Computer Architecture*.

[12] Haoqiang Guo, Lu Peng, Jian Zhang, Qing Chen, and Travis D. LeCompte. 2020. ATT: A fault-tolerant ReRAM accelerator for attention-based neural networks. In *IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 213–221.

[13] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H. Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W. Lee, et al. 2020. A^ 3: Accelerating attention mechanisms in neural networks with approximation. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 328–341.

[14] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W. Lee. 2021. ELSA: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 692–705. DOI: https://doi.org/10.1109/ISCA52012.2021.00060

[15] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. 2017. ESE: Efficient speech recognition engine with sparse LSTM on FPGA. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 75–84.

[16] Yanzhang He, Tara N. Sainath, Rohit Prabhavalkar, Ian McGraw, Raziel Alvarez, Ding Zhao, David Rybach, Anjuli Kannan, Yonghui Wu, Ruoming Pang, et al. 2019. Streaming end-to-end speech recognition for mobile devices. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 6381–6385.

[17] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computat.* 9, 8 (1997), 1735–1780.

[18] Hanhwi Jang, Joonsung Kim, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. 2019. MnnFast: A fast and scalable system architecture for memory-augmented neural networks. In *46th International Symposium on Computer Architecture*. 250–263.

[19] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358* (2018).

[20] Joonsung Kim, Suyeon Hur, Eunbok Lee, Seungho Lee, and Jangwoo Kim. 2021. NLP-Fast: A fast, scalable, and flexible system to accelerate large-scale heterogeneous NLP models. In *30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 75–89.

[21] Dongup Kwon, Suyeon Hur, Hamin Jang, Eriko Nurvitadhi, and Jangwoo Kim. 2020. Scalable multi-FPGA acceleration for large RNNs with full parallelism levels. In *57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[22] Bingbing Li, Santosh Pandey, Haowen Fang, Yanjun Lyv, Ji Li, Jieyang Chen, Mimi Xie, Lipeng Wan, Hang Liu, and Caiwen Ding. 2020. FTRANS: Energy-efficient acceleration of transformers using FPGA. In *ACM/IEEE International Symposium on Low Power Electronics and Design*. Association for Computing Machinery, New York, NY, 175–180. DOI: https://doi.org/10.1145/3370748.3406567

[23] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21)*. Association for Computing Machinery, New York, NY, 977–991. DOI: https://doi.org/10.1145/3466752.3480125

[24] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 522–531.

[25] J. S. McCarley, Rishav Chakravarti, and Avirup Sil. 2019. Structured pruning of a BERT-based question answering model. *arXiv preprint arXiv:1910.06360* (2019).

[26] Eriko Nurvitadhi, Dongup Kwon, Ali Jafari, Andrew Boutros, Jaewoong Sim, Phillip Tomson, Huseyin Sumbul, Gregory Chen, Phil Knag, Raghavan Kumar, et al. 2019. Why compete when you can work together: FPGA-ASIC integration for persistent RNNs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*.

[27] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. (2019).

[28] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[29] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2020. MLPerf inference benchmark. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 446–459.

[30] David E. Rumelhart and James L. McClelland. 1987. *Learning Internal Representations by Error Propagation*. 318–362.

[31] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2018. SCALE-Sim: Systolic CNN accelerator simulator. *arXiv preprint arXiv:1811.02883* (2018).

[32] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).

[33] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[34] Thierry Tambe, Coleman Hooper, Lillian Pentecost, Tianyu Jia, En-Yu Yang, Marco Donato, Victor Sanh, Paul Whatmough, Alexander M. Rush, David Brooks, and Gu-Yeon Wei. 2021. EdgeBERT: Sentence-level energy optimizations for latency-aware multi-task NLP inference. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21)*. Association for Computing Machinery, New York, NY, 830–844. DOI: https://doi.org/10.1145/3466752.3480095

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *International Conference on Advances in Neural Information Processing Systems*.

[36] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. 2019. MAGNet: A modular accelerator generator for neural networks. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[37] Hanrui Wang, Zhekai Zhang, and Song Han. 2020. SpAtten: Efficient sparse attention architecture with cascade token and head pruning. *CoRR* abs/2012.09852 (2020).

[38] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. 2018. C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs. In *ACM/SIGDA International Symposium on Field-programmable Gate Arrays*. 11–20.

[39] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. 2020. GOBO: Quantizing attention-based NLP models for low latency and energy efficient inference. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 811–824.

[40] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. 2019. Q8BERT: Quantized 8bit BERT. *arXiv preprint arXiv:1910.06188* (2019).

[41] Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. 2020. TernaryBERT: Distillation-aware ultra-low bit BERT. *arXiv preprint arXiv:2009.12812* (2020).

[42] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[43] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2020. DNNExplorer: A framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator. In *39th International Conference on Computer-Aided Design*. 1–9.

Received 4 March 2022; revised 8 August 2022; accepted 5 September 2022