Understanding the Potential of FPGA-based Spatial Acceleration for Large Language Model Inference

HONGZHENG CHEN, Cornell University, Ithaca, United States
JIAHAO ZHANG, Tsinghua University, Beijing, China
YIXIAO DU, Cornell University, Ithaca, United States
SHAOJIE XIANG, Cornell University, Ithaca, United States
ZICHAO YUE, Cornell University, Ithaca, United States
NIANSONG ZHANG, Cornell University, Ithaca, United States
YAOHUI CAI, Cornell University, Ithaca, United States
ZHIRU ZHANG, Cornell University, Ithaca, United States

Recent advancements in large language models (LLMs) boasting billions of parameters have generated a significant demand for efficient deployment in inference workloads. While hardware accelerators for Transformer-based models have been extensively studied, the majority of existing approaches rely on temporal architectures that reuse hardware units for different network layers and operators. However, these methods often encounter challenges in achieving low latency due to considerable memory access overhead.

This article investigates the feasibility and potential of model-specific spatial acceleration for LLM inference on field-programmable gate arrays (FPGAs). Our approach involves the specialization of distinct hardware units for specific operators or layers, facilitating direct communication between them through a dataflow architecture while minimizing off-chip memory accesses. We introduce a comprehensive analytical model for estimating the performance of a spatial LLM accelerator, taking into account the on-chip compute and memory resources available on an FPGA. This model can be extended to multi-FPGA settings for distributed inference. Through our analysis, we can identify the most effective parallelization and buffering schemes for the accelerator and, crucially, determine the scenarios in which FPGA-based spatial acceleration can outperform its GPU-based counterpart.

To enable more productive implementations of an LLM model on FPGAs, we further provide a library of high-level synthesis (HLS) kernels that are composable and reusable. This library will be made available as open-source. To validate the effectiveness of both our analytical model and HLS library, we have implemented Bidirectional Encoder Representations from Transformers (BERT) and Generative Pre-trained Transformers (GPT2) on an AMD Xilinx Alveo U280 FPGA device. Experimental results demonstrate our approach can achieve up to 13.4× speedup when compared to previous FPGA-based accelerators for the BERT model. For GPT generative inference, we attain a 2.2× speedup compared to Design for Excellence, an FPGA overlay, in

J. Zhang's work was done while interning at Cornell.

This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and NSF Awards No. 2007832, No. 2019306, and No. 2118709.

Authors' addresses: H. Chen, Y. Du, S. Xiang, Z. Yue, N. Zhang, Y. Cai, and Z. Zhang, Cornell University, 136 Hoy Rd, Ithaca, NY, 14850, United States; e-mails: hzchen@cs.cornell.edu, yd383@cornell.edu, sx233@cornell.edu, zy383@cornell.edu, nz264@cornell.edu, yc2632@cornell.edu, zhiruz@cornell.edu; J. Zhang, Tsinghua University, 30 Shuangqing Rd, Beijing, 100190, China; e-mail: jiahao-z19@mails.tsinghua.edu.cn.

ACM 1936-7406/2024/12-ART5
https://doi.org/10.1145/3656177

the prefill stage, while achieving a 1.9× speedup and a 5.7× improvement in energy efficiency compared to the NVIDIA A100 GPU in the decode stage.

CCS Concepts: • **Hardware → Hardware-software codesign**; • **Computing methodologies → Neural networks;**

Additional Key Words and Phrases: FPGA, high-level synthesis, large language models, hardware acceleration

ACM Reference Format:
Hongzheng Chen, Jiahao Zhang, Yixiao Du, Shaojie Xiang, Zichao Yue, Niansong Zhang, Yaohui Cai, and Zhiru Zhang. 2024. Understanding the Potential of FPGA-based Spatial Acceleration for Large Language Model Inference. *ACM Trans. Reconfig. Technol. Syst.* 18, 1, Article 5 (December 2024), 29 pages. https://doi.org/10.1145/3656177

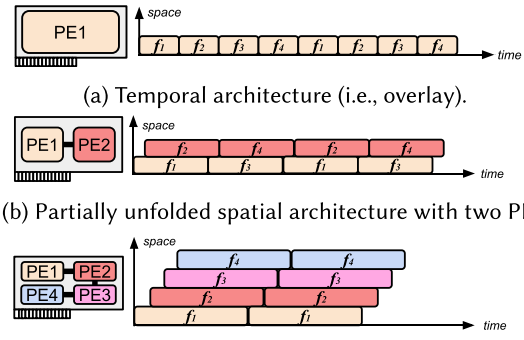
1 INTRODUCTION

The rapid advancement of Transformer-based **large language models (LLMs)** [5, 74] has sparked a revolution across a wide range of natural language processing tasks, such as conversational AI [13, 54, 104] and code generation [10, 42, 52]. Recent research has brought to light the phenomenon of "emergence" in LLMs, where advanced capabilities become evident as the models scale up to billions of parameters [77, 78]. However, supporting this unprecedented scale poses significant challenges, particularly in terms of computational and memory resources. At the same time, the increasing use of LLMs in interactive applications like voice assistants and autonomous systems requires hardware accelerators capable of providing both low latency and high energy efficiency [17, 54, 62].

Recent efforts have primarily focused on improving the performance of LLM inference on **graphic processing units (GPUs)** [2, 53], although GPUs are known for their high power consumption and are less suitable for latency-sensitive workloads [32, 62]. There is also an active body of research dedicated to developing specialized hardware accelerators tailored for Transformer models, with several of these efforts using **field-programmable gate arrays (FPGAs)** as the target platforms [23, 26, 39, 46, 59, 63].

FPGA-based LLM accelerators can be broadly categorized into two architectural paradigms: *temporal architecture* and *spatial architecture*. In a temporal architecture, a **processing engine (PE)** capable of performing various tasks is constructed and reused across different layers and models, as shown in Figure 1(a). For flexibility, these accelerators typically employ an overlay approach [23, 28, 39], where a virtual hardware architecture that executes instructions is "laid" on top of the physical FPGA fabric. Overlays provide a more restricted configuration space, allowing for quicker compilation with bitstream reuse across multiple models. However, the use of such temporal architecture requires more frequent off-chip memory access, as intermediate results must be written back to memory. This incurs a cost in terms of both latency and energy consumption that is significantly higher than direct on-chip memory access. Additionally, one could argue that an FPGA overlay will inherently be less efficient than its hardened ASIC counterpart.

In contrast, an FPGA-based spatial architecture typically involves the specialization of distinct PEs for specific operators or layers, facilitating direct communication between them using streaming buffers (e.g., **first in, first out (FIFOs)** or multi-buffers) [60, 72, 75, 80], as depicted in Figures 1(b) and 1(c). This dataflow-style execution substantially reduces off-chip memory accesses and enables the concurrent processing of multiple PEs in a pipelined manner. Moreover, the fine-grained programmability of FPGAs allows efficient support of model-specific spatial architectures, which can further leverage efficiency optimizations such as low-bitwidth quantization, custom numerical types, and sparsity [58, 69, 93, 102]. These capabilities can potentially enable highly



(a) Temporal architecture (i.e., overlay).

(b) Partially unfolded spatial architecture with two PEs.

(c) Fully unfolded spatial architecture with four PEs.

Fig. 1. Temporal and spatial architectures—PE stands for processing engine; f_1–f_4 represent different operators in the model.

efficient LLM inference implementations that surpass GPUs, especially in small-batch low-latency scenarios.

However, implementing a spatial architecture for LLM inference presents significant challenges. **Challenge 1: Navigating diverse parallelism in LLMs.** The generative inference process of LLMs typically consists of two distinct stages: (1) simultaneously processing user prompts and (2) sequentially generating new tokens in an autoregressive manner. These two stages exhibit significantly different computational and memory characteristics (detailed in Section 3), making it necessary to tailor hardware accelerators for their specific needs. This challenge cannot be directly addressed by leveraging techniques from the traditional **convolutional neural network (CNN)** designs [32, 97]. The large number of parameters and intermediate tensors further complicates the choice between on-chip and off-chip storage. Additionally, harnessing multiple accelerators for distributed LLM inference adds complexity, particularly when dealing with intricate parallelization schemes [23, 49, 68].

Challenge 2: Lack of standard LLM building blocks in hardware accelerators. The rapid evolution of LLM architectures [5, 54, 70] contrasts with the comparatively slow pace of hardware development. While a plethora of building blocks for Transformers have been proposed in the software domain [14, 19, 38], the absence of reusable blocks for hardware accelerator design hampers development progress. Many frameworks have been designed to automatically map deep learning models to FPGAs [3, 20, 72, 98, 100], but they are constrained to small CNN designs and lack support for complicated Transformer models. It is also hard to scale their designs to accommodate large models and multi-die FPGAs.

To tackle these challenges, this article is to provide a comprehensive set of hardware design considerations for LLMs and try to answer the following question: *What role can FPGA-based spatial accelerators play in enabling efficient LLM inference?* We start by conducting an in-depth analysis of the computational and memory requirements associated with each operator within Transformer models across two distinct stages of LLM generative inference—prefill and decode. Subsequently, we extend our analysis to reveal the potential benefits of distributed inference using multiple FPGAs. **We believe that providing such an analysis, rather than presenting only positive results in selectively chosen settings for an FPGA LLM accelerator, offers more valuable insights to the community.** To validate the feasibility of our analytical framework, we implement a specific design point and demonstrate its viability. Leveraging this analytical framework, we employ specific optimizations in HLS to craft each kernel and compose them into a hardware

accelerator that achieves the expected performance. While our primary focus is not to propose a new LLM accelerator architecture, we demonstrate that by using the analytical model, we can create a high-performance design that surpasses previous efforts. Our major contributions are as follows:

- We introduce an analytical framework that presents the first in-depth analysis of both the advantages and limitations of FPGA-based LLM spatial acceleration. This framework not only allows us to estimate the performance of a specific accelerator configuration on a given FPGA device but also provides guidance for designing accelerators for LLM inference.
- We create a suite of modular and reusable HLS kernels designed for building FPGA-based spatial accelerators for different Transformer models. We plan to open-source this kernel library[1] and expect it to serve as a valuable resource for benchmarking HLS and FPGA acceleration more broadly.
- Leveraging our kernel library, we design and implement a range of high-performance FPGA-based LLM accelerators that achieve speedups comparable to previous GPU and FPGA-based accelerators. Specifically, for the BERT model, we achieve a 13.4× speedup over prior FPGA-based accelerators. For GPT generative inference, we achieve speedups of 2.2× and 1.1× in prefill and decode stages, respectively, when compared to DFX, an FPGA-based overlay architecture. Additionally, our accelerator is 1.9× faster and 5.7× more energy-efficient than the A100 GPU in the decode stage.

2 BACKGROUND

This section provides backgrounds on Transformer models and introduces parallelization schemes for LLM inference.

2.1 Transformer Models

The Transformer model consists of both encoder and decoder blocks [74]. Recent employment on LLMs mostly uses decoder-only models, which leverage an auto-regressive approach for text generation [54, 65, 70]. We will mainly discuss decoder-only models in this article, but since encoders and decoders share the core building blocks with subtle architectural variances, our approach can also be extended for encoder-only models [16, 36, 45].

As illustrated in Figure 2, generative inference of LLMs has two stages: prefill stage and decode stage [62]. In the prefill stage, the model takes in user prompts, normally a long sequence with l_{input} tokens, goes through the whole Transformer model, and generates the first token. In the decode stage, the model takes in the previously generated token and generates l_{gen} new tokens one at a time in an auto-regressive way. Since each token depends on the previously generated tokens, the decode stage is purely sequential.

We then go through the detailed model architecture. The input tokens are first passed into an embedding layer that maps the discrete tokens into high-dimensional continuous representations while incorporating positional encoding for each token. Subsequently, it generates a tensor (i.e., hidden states) of shape (l, d), where l represents sequence length, and d is the size of hidden dimensions. We omit the batch dimension to simplify the analysis, focusing solely on single-batch inference in this article, but our approach can be easily extended to different batch sizes for LLM serving by adding an additional batch dimension [34, 43].

The hidden states then pass through a series of N Transformer blocks. Each Transformer block consists of two sublayers: a **multi-head attention (MHA)** module and a **feed-forward network**

[1] https://github.com/cornell-zhang/allo/tree/main/examples

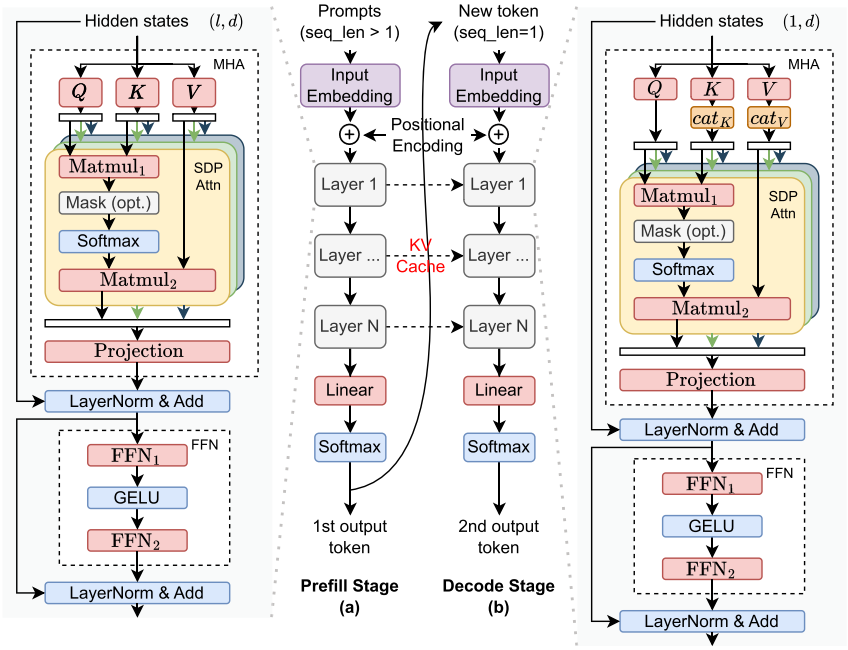


Fig. 2. Transformer model. Red blocks represent linear operators, and blue blocks signify non-linear operators.

(FFN). Residual connections and **layer normalization (LayerNorm)** functions are applied between these sublayers, although the specific order and application may vary across different models [91]. The MHA module plays a crucial role in capturing token relationships within the input sequence. The input is initially partitioned into h segments, where h corresponds to the number of attention heads. To compute the attention scores for each head, the input sequence of length l undergoes three linear projections: query, key, and value. These projections, which are trainable, yield matrices Q, K, and V, respectively. Attention scores are then computed using a **scaled dot-product (SDP)** operator between Q, K, and V, as specified by the formula

$$\text{Attention}(Q, K, V) = \text{softmax}\left(QK^T \Big/ \sqrt{d_k}\right) V\,, \tag{1}$$

where d_k is the size of the hidden dimension. The output from this operator involves h outputs, which are subsequently concatenated and processed through an additional linear projection. In the prefill stage, the generated K and V tensors will be stored as *KV cache* and later be concatenated before SDP during the decode stage [62].

The FFN module comprises a linear layer followed by a non-linear activation function and another linear layer. This module transforms the outputs of MHA into embedding matrices, which are then further processed by subsequent Transformer layers.

Finally, the output tensor will go through a softmax function to obtain a distribution. The model will sample a token from this distribution and feed it into the decode stage. For encoder-only models, there is only a prefill stage involved, and the distribution will be directly used for different downstream tasks like text classification [16, 36, 45].

In this article, we only focus on analyzing the core Transformer blocks and accelerating them on FPGAs. Embedding layers and output sampling [8, 25] require extensive random memory accesses, which may not be suitable for FPGA acceleration. Also, they only take a small fraction of overall

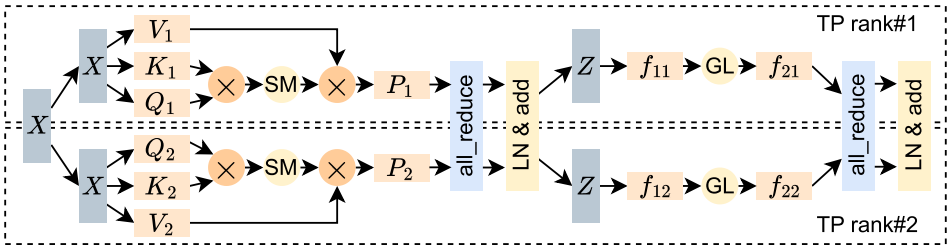


Fig. 3. Example of tensor parallelism of a Transformer layer with two devices. TP rank is the unique identifier given to a device within a TP group. SM is the softmax function, LN is LayerNorm, and GL is the GeLU function.

compute that does not affect the overall latency [32], so we leave them to execute on CPUs or GPUs as usual.

2.2 Parallelization Schemes

As model sizes continue to expand, it becomes increasingly common for a single device to be insufficient for accommodating the entire model. Consequently, the exploration of diverse parallelization schemes within a device and across devices becomes necessary. Parallelism techniques in deep learning can be roughly classified into data, tensor, and pipeline parallelism, together known as 3D parallelism [9, 33, 51, 68]. During the era of CNNs, data parallelism was the norm, involving the partitioning of input tensors along the batch dimension and their distribution across multiple devices [1, 41]. DeepSpeed ZeRO [66] and FSDP [47] extended data parallelism by proposing a three-stage parallelism strategy that partitions optimizer states, gradients, and parameters to minimize memory usage. However, this approach may incur high communication overheads during inference.

A more recent parallelism scheme for LLM inference is **tensor parallelism (TP)** [2, 43, 51, 62, 68], which distributes model parameters across multiple devices and conducts explicit collective operations to ensure model correctness. Megatron-LM [68] is the first to explore tensor parallelism for Transformer-based models, proving to be efficient in both training and inference due to relatively low communication costs. As shown in Figure 3, tensor parallelism requires two `all_reduce` operations inside a Transformer layer to ensure the results are correct. Our accelerator design also explores tensor parallelism, as detailed in Section 3.4.2.

Last, pipeline parallelism [49, 50, 92] divides the model across network layers. Multiple layers are grouped into a pipeline stage, and different stages are assigned to different devices. Pipeline parallelism is typically employed across multiple nodes. Since both tensor parallelism and pipeline parallelism handle only portions of the network model, they are collectively referred to as model parallelism. We revisit these parallelization schemes in Section 3.4.2.

3 ANALYTICAL MODELING FRAMEWORK

In this section, we propose a comprehensive analytical modeling framework aimed at understanding the computational requirements of a Transformer layer. Our investigation begins by analyzing the compute demands and resource constraints on a single device. We base our estimations on these constraints. Finally, we extend the framework to the analysis of multiple devices.

3.1 Computational Demands

Our first imperative is to calculate the computational demands of the model. Given that the predominant computation in the Transformer model is **general matrix-matrix multiplication (GEMM**

Table 1. MACs of the Prefill and Decode Stages of the Linear Layers in the Transformer Model in Figure 2

Linear Layer	Abbreviations	Input Matrices	Prefill	Decode
Q/K/V linear	q, k, v	XW_Q, XW_K, XW_V	$3ld^2$	$3d^2$
Matmul_1	a_1	QK^{T}	l^2d	$(l+1)d$
Matmul_2	a_2	$X_{\text{sm}}V$	l^2d	$(l+1)d$
Projection	p	$X_{\text{sdp}}W_{\text{Proj}}$	ld^2	d^2
FFN_1	f_1	$X_{\text{mha}}W_{\text{FFN}_1}$	ldd_{FFN}	dd_{FFN}
FFN_2	f_2	$X_{\text{act}}W_{\text{FFN}_2}$	ldd_{FFN}	dd_{FFN}

l denotes input sequence length, d denotes input feature dimension size, and d_{FFN} denotes FFN hidden dimension size.

or Matmul) or **general matrix-vector multiplication (GEMV)** [32, 62], we employ the number of **multiply–accumulates (MACs)** as the proxy metric for quantifying compute requirements of the linear layers, as depicted in Table 1. For non-linear layers such as softmax and GeLU functions, they are elementwise operators that can be easily fused with the GEMM kernels in a pipeline design without affecting the final performance. More experimental results are provided in Section 6.3.

We denote $X_{(\cdot)}$ as the output tensors from preceding layers, and $W_{(\cdot)}$ represents the weights of corresponding linear layers. For example, X_{sm} is the output of the softmax operator. Our analysis here is restricted to a single batch; hence the tensors only have two dimensions. We can observe that the computational demand during the prefill stage far surpasses that of the decode stage. In the prefill stage, the required MACs of the two `Matmul`s within the SDP are quadratic to the sequence length (i.e., l^2d). Consequently, when input sequences exhibit substantial length, attention layers may extend computation time significantly. On the contrary, in the decode stage, each operator processes a single token at a time, making the MACs independent of the sequence length except for SDP.

3.2 Resource Constraints

We then model the compute and memory resource constraints on an FPGA. In this section, we assume that one FPGA device can effectively compute at least a single Transformer layer, but our framework can be easily extended to more resource-constrained cases using a similar analysis proposed in Section 3.4.

3.2.1 Compute Resource Constraints. The core computational element for linear operators is the MAC unit. Let M_i denote the compute power, in terms of the number of MACs per cycle allocated to each matrix multiplication kernel, where i ranges over q, k, v, a_1, a_2, p, f_1, and f_2, based on the notation in Table 1. We quantize the matrix multiplication to integer inputs for maximum efficiency, which has been proven to be effective by many recent studies [15, 30, 67, 81]. Quantization enables single-cycle accumulation. As a result, one **multiply-accumulator (MAC)** unit can provide a 1 MAC/cycle throughput with a properly pipelined multiplier. Therefore, the latency for the Q projection can be calculated as ld^2/M_q cycles, considering that the total number of MACs computed in this operator is ld^2.

Suppose we want to deploy C Transformer model layers on an FPGA. The total MAC units must not exceed the capacity of the device. Since we employ a dataflow design that unfolds all the layers on-board, the required MAC units are simply the sum of the MAC units for each layer. This requirement can be expressed as

$$\sum M_i C < M_{\text{tot}}, i \in \{q, k, v, a_1, a_2, p, f_1, f_2\}, \tag{2}$$

where M_{tot} represents the total available compute power of an FPGA in terms of MACs per cycle, which can be obtained from the official data sheets. For FPGAs with specialized compute blocks (e.g., AI Engine [86] and AI Tensor Blocks [37]), we can convert their compute power to match the frequency of the programming logic, thus obtaining an overall value M_{tot} for the entire FPGA. For example, the VCK5000 FPGA [86] has 400 AI Engines, each of which can compute 128 MACs/cycle at 1 GHz. Therefore, the equivalent compute power at 250 MHz is 128×400×1 GHz/250 MHz, which is 204,800 MACs/cycle.

3.2.2 Memory Capacity Constraints. The demand for memory capacity stems from a variety of on-chip buffers, including weight buffers for parameters, buffers for K and V matrices, and FIFOs interconnecting different stages.

Parameter buffers. To optimize an FPGA-based dataflow design, we assume that all the quantized parameters can be accommodated in on-chip or off-chip memory. Suppose all the linear weights are quantized to b_W bits, and the size of the linear operator i is s_i. The total size of the buffers is $S_{\text{param}} = \sum_{i \in \{q,k,v,p,f_1,f_2\}} s_i b_W = (4d^2 + 2dd_{\text{FFN}})b_W$ if storing on-chip. If the parameters are too large to fit in on-chip memory, then we can store the parameters in DRAM and tile the parameters with size M_i on-chip, then the total tiled buffer size is $S_{\text{tile}} = \sum_{i \in \{q,k,v,p,f_1,f_2\}} M_i b_W$. To hide the memory access latency, we need to double buffer those parameters, so the final buffer size of the ith linear operator is $2S_{\text{tile}}$.

KV Cache. When conducting matrix multiplication, at least one of the matrices' elements must be accessed repeatedly so that a buffer is required. Given that parameters are already buffered, only the SDP requires buffering for at least one of the input matrices. In our case, we choose to buffer K and V, which will be later passed to the decode stage as the KV cache. We also double buffer K and V matrices to improve throughput. The final buffer size is $S_{\text{KV}} = 4l_{\max}db_A$, where b_A is the bitwidth of the activation and $l_{\max}$ is the maximum sequence length supported by the model. Notice KV cache can also be tiled on-chip, which can leverage a similar analysis above.

FIFOs. The intermediate results between linear operators flow in FIFOs, since the linear operators sequentially access them. For the initial residual connection, we assume that the input tensors are fetched from off-chip memory to obviate the need for additional buffering. However, for the second residual connection related to the FFN, it is necessary to use an intermediate buffer to store the projection's activation X_{act} before the FFN. This buffer simultaneously serves as a bypass path. To avoid deadlock, the buffer must possess sufficient capacity to store X_{act}. We simply create a FIFO of size ldb_A to store it. For other FIFO connections, we assume a FIFO depth of s and one FIFO connecting each layer in Figure 2, so the total FIFO size is equal to $S_{\text{FIFO}} = 16sb_A + ldb_A$.

In summary, the memory capacity constraint is expressed as

$$\begin{aligned} &S_{\text{param}}C < DRAM_{\text{tot}}\,, \\ &\sum S_i C < SRAM_{\text{tot}}\,, i \in \{\text{tile}, \text{KV}, \text{FIFO}\}\,, \end{aligned} \tag{3}$$

if the parameters are stored off-chip. $DRAM_{\text{tot}}$ and $SRAM_{\text{tot}}$ are the total available off-chip and on-chip memory.

3.2.3 Memory Port Constraints. Besides memory capacity, we also need to consider constraints on memory ports in a highly paralleled design. For matrix multiplication, if different MAC units work in parallel, they will visit the weight/result buffers simultaneously, hence contending for memory ports. This issue can be addressed by either partitioning the buffer, effectively offering more memory ports; or packing data to create wider elements, subsequently reducing the number of memory ports required.

SRAM resources. The on-chip SRAM resources of FPGAs are typically organized as blocks. Each block has a fixed capacity and may support configurable bitwidth. For example, on AMD UltraScale+ FPGAs, there are two types of SRAM resources: ***Block RAM (BRAM)*** and ***Ultra RAM (URAM)***. BRAM blocks can be configured to 1×36 Kb block or 2×18 Kb blocks, with two read and write ports each. URAM blocks are 288 Kb with one read and one write port. The port width of the BRAM block is flexible; it can be configured to 1, 2, 4, 9, 18, 36, or 72 (in 36 Kb mode) bits, while the port width of the URAM block is fixed at 72 bits. Similar to BRAM and URAM, Intel FPGAs have M20K and eSRAM with different configurable port widths.

Memory blocks needed without data packing. To begin with, we analyze the port constraints without data packing. In this case, to eliminate the port contention, different MAC units may need different memory ports. Consider the linear operator i with the size of s_i with M_i MAC units working in parallel, each loaded weight may feed multiple MAC units due to intrinsic data reuse in GEMM. We use r_i to represent the data reuse factor (number of MAC units sharing the loaded weight). Therefore, the weight buffer needs to be partitioned into M_i/r_i parts. If we store all the weight buffers on-chip, then the number of b_W-bit elements in each partition is $s_i/(M_i/r_i)$. However, b_W may not fully occupy one memory word as the memory bitwidth can only take limited options. We introduce the effective bit width, b_{BRAM}, to be the smallest memory bitwidth larger than b_W. Let S_{BRAM} be the total capacity (in bits) of one memory block, we can deduce the total number of memory blocks for one linear operator:

$$R_i = \left\lceil \frac{s_i b_{BRAM}}{M_i/r_i \times S_{BRAM}} \right\rceil \times M_i/r_i \,. \tag{4}$$

If the parameters are loaded from off-chip memory and we only store a tile of the weight on-chip, then s_i is simply M_i, and R_i also becomes M_i as $b_{BRAM} \ll S_{BRAM}$. Since we need to double buffer those parameters, the final buffer size of the ith linear operator is $2M_i$. Notice the k and v layers need to be double-buffered, so the required BRAM also doubles in these two layers. We can obtain the total required BRAM as follows:

$$\sum_{i \in \{q,k,v,p,f_1,f_2\}} CR_i + 2C(R_{a_1} + R_{a_2}) < Mem_{\text{tot}} \,. \tag{5}$$

Memory blocks needed with data packing. Data packing can alleviate the strain on memory port contention by consolidating multiple narrow data into a single, wider data element. This process allows multiple MAC units to access data from the same memory port. We consider packing data into b_{pack} bits for the linear weights, and we have $b_{pack} = kb_W$. Again, we denote b_{BRAM} as the smallest memory bitwidth larger than b_{pack}. We need to partition M_i/r_i MAC units to $M_i/r_i/k$ parts, and each partition has $\lceil s_i/k \times b_{BRAM}/(M_i/r_i/k) \rceil$ bits. Therefore, the total number of memory blocks needed is

$$R_i = \left\lceil \frac{s_i b_{BRAM}}{M_i/r_i \times S_{BRAM}} \right\rceil \times \frac{M_i/r_i}{k} \,. \tag{6}$$

3.2.4 Memory Bandwidth Constraints. If the parameters are stored off-chip, then we need to consider the impact of off-chip memory bandwidth. Similar to Section 4.2.2, we use r_i to denote the data reuse factor of a linear operator with M_i MAC units. Effectively, M_i/r_i weights must be loaded from off-chip memory per cycle to feed the MAC units, requiring a bandwidth of

$$B_i = b_W \times M_i/r_i \times freq \,, \tag{7}$$

where $freq$ is the achieved frequency of FPGA. If the total required bandwidth, $\sum_i CB_i (i \in \{q, k, v, p, f_1, f_2\})$, exceeds the maximum device bandwidth, then the inference becomes bandwidth

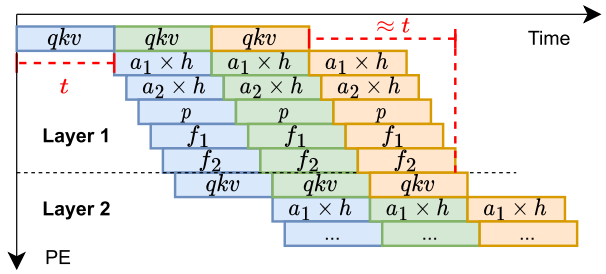


Fig. 4. Pipeline diagram. Different colors stand for different input samples. Different blocks stand for different linear operators, which also constitute the pipeline stages. h is the number of attention heads.

bound. Notice this bandwidth requirement needs to be analyzed for each operator individually if the data loading requires accessing multiple DDR or HBM channels.

3.3 Performance Estimation

In this section, we estimate the overall latency based on the constraints and conduct work balancing for the dataflow.

3.3.1 Latency Estimation. We construct the pipeline diagram as shown in Figure 4. As mentioned in Section 3.2.2, since we need to store the K and V values after the linear operators, there is an implicit synchronization point between the $q/k/v$ operator and the latter SDP and FFN parts. The computation of them cannot be overlapped. Notice the $q/k/v$ operator can be performed in parallel, since they do not have any dependencies. After k and v have been fully calculated, the subsequent computations of SDP and FFN can be greatly overlapped. This is because these operations do not need to wait for all the results to perform the next operation. The results of the previous operation can be directly streamed into the next operation as input. Moreover, since different Transformer layers share the same architecture, their computation can also be overlapped without waiting for the result of the previous layer.

Suppose the Transformer model has N layers in total. Since we have C layers on one FPGA, it needs to iterate N/C times to process the whole model. We can calculate the latency of different stages, and the overall latency is the maximum latency of these stages (which defines the initiation interval of the pipeline) times the number of iterations, i.e.,

$$T_{\text{prefill}} = \frac{1}{freq}\frac{N}{C}\left(\frac{ld^2}{M_k} + C\max\left(\frac{ld^2}{M_k}, \frac{l^2d}{M_{a_1}}, \frac{ldd_{\text{FFN}}}{M_{f_1}}, T_{\text{mem}}\right)\right), \tag{8}$$

$$T_{\text{decode}} = \frac{1}{freq}\frac{N}{C}\left(\frac{d^2}{M_k} + C\max\left(\frac{d^2}{M_k}, \frac{(l_{\max}+1)d}{M_{a_1}}, \frac{dd_{\text{FFN}}}{M_{f_1}}, T_{\text{mem}}\right)\right), \tag{9}$$

where the first term inside the parentheses is the latency of the $q/k/v$ linear operator (i.e., t in Figure 4). T_{mem} is the off-chip memory access latency, which can be calculated based on Equation (7).

3.3.2 Work Balancing. As the overall latency is determined by the slowest stage in the dataflow, we can balance the execution time of each stage; hence, we have

$$\frac{ld^2}{M_{q,k,v,p}} = \frac{l^2d/h}{M_{a_1,a_2}}h = \frac{ldd_{\text{FFN}}}{M_{f_1,f_2}} \tag{10}$$

$$\Longrightarrow M = M_{q,k,v,p} = d/lM_{a_1,a_2} = d/d_{\text{FFN}}M_{f_1,f_2}, \tag{11}$$

where M is defined as the global compute power in MACs/cycle. Finally, Equation (8) can be simplified to

$$T_{\text{prefill}} = \frac{1}{freq} N \left(1 + \frac{1}{C}\right) \frac{ld^2}{M}, \tag{12}$$

which shows the overall latency with work balancing. We can obtain the latency for the decode stage using a similar analysis.

To derive the optimal M for a given model, we devise a linear search algorithm to identify the maximum available M based on the constraints in Equations (2), (3), and (6). Notice the optimal M represents an upper bound of the compute power. In practice, we also need to consider the routing issue to adjust the actual achievable M as discussed in Section 5.2.

3.4 Distributed Inference

As a single FPGA may not be sufficient to process some extremely large models, we next extend our modeling to multiple FPGAs. We first characterize the communication cost between two FPGAs and discuss the impact of different parallelism schemes.

3.4.1 Communication. Various methods exist for facilitating inter-FPGA communication, including communication through the host, PCI-E **Peer-to-Peer (P2P)**, and on-device Ethernet. We mainly consider the third approach, since it does not necessitate orchestration from the host and provides higher bandwidth compared to other alternatives. For example, the AMD Alveo U280 FPGA provides two QSFP ports [85], each capable of carrying 100 Gb/s Ethernet data over optical fibers, which ensures robust and high-speed inter-FPGA communication. Most of the time, we cannot fully utilize the network bandwidth and need to pay for the package header overheads. Suppose the theoretical network bandwidth between two FPGA devices is B bits per second (bps), and the efficiency of the network is α, so we can have the effective bandwidth as αB, where α can be obtained through network benchmarking.

3.4.2 Parallelization Schemes. As mentioned in Section 2.2, we have various parallelization schemes when considering multiple devices. We first analyze **tensor parallelism (TP)**. As shown in Figure 3, the parameters of the linear operations are partitioned across different devices. For example, suppose the weight parameters of the two FFN layers f_1 and f_2 are A and B, then we can partition A along its column and partition B along its row, and obtain

$$\sigma(ZA)B = \sigma\left(Z\begin{bmatrix}A_1 & A_2\end{bmatrix}\right)\begin{bmatrix}B_1\\B_2\end{bmatrix} = \sigma(ZA_1)B_1 + \sigma(ZA_2)B_2,$$

where σ is the GeLU function. Therefore, apart from partitioning A and B, we need to insert an all-reduce operation to aggregate the partial results on each device to ensure correctness. The partitioned parameters will be stored on different devices. For example, A_1 will be on the first FPGA, and A_2 will be on the second FPGA. A similar partition scheme can be applied for MHA, and we refer the readers to Reference [68] for more details.

Based on this partition scheme, TP requires two all-reduce operations within one Transformer layer. However, these communicative operations are implemented in a blocking way. Figure 5(a) shows the subsequent FFN module needs to wait for the completion of the all-reduce process before it can conduct computation [76]. Notice that the all-reduce operation only involves fetching results from other devices and adding the result to its local tensor. Given that the output of MHA is a sequential stream, we can perform elementwise addition in a non-blocking manner. As soon as the kernel receives enough data, it can initiate data transfer to other devices without waiting for the remaining data to be computed. This leads to substantial synchronization time savings as shown in Figure 5(b).

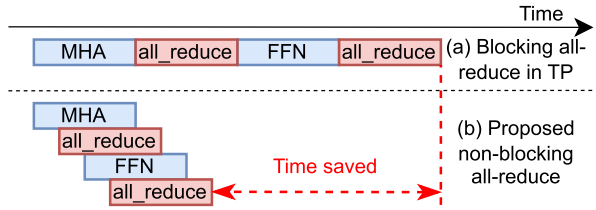


Fig. 5. Blocking and non-blocking all-reduce in TP. The latency of different stages is not drawn to scale.

Table 2. Models Used in Sections 4 and 6

Model	Type	# of params	# Layers N	# Heads h	Hidden Size d	FFN size d_{FFN}
BERT [16]	Encoder	110M	12	12	768	3,072
GPT2 [65]	Decoder	355M	24	16	1,024	4,096
LLaMA2 [71]	Decoder	7B	32	32	4,096	11,008
Vicuna [11]	Decoder	13B	40	40	5,120	13,824

Since the size of the output tensor of MHA and FFN are both ld, the communication time for one all-reduce is

$$T_{\text{comm}} = ldb_A/(\alpha B)\,. \tag{13}$$

As we have already implemented dataflow inside a device, pipeline parallelism (PP) essentially extends the dataflow to p_2 devices with a tensor of size ld communicated in between. Here, we only split the pipeline between two Transformer layers so the results of the previous device can be directly streamed to the next device in the same PP group. Notice TP and PP can be combined to conduct model inference [51], and the latency of Equation (8) becomes

$$T_{\text{prefill}} = \frac{1}{freq}\frac{N}{p_2 C}\left(\frac{ld^2}{p_1 M_k} + p_2 C \max\left(\frac{ld^2}{p_1 M_k}, \frac{l^2 d}{p_1 M_{a_1}}, \frac{ldd_{\text{FFN}}}{p_1 M_{f_1}}, T_{\text{mem}}, T_{\text{comm}}\right)\right), \tag{14}$$

where p_1 and p_2 are the size of a TP group and a PP group [68]. Additionally, the memory requirements of Equations (3) and (5) need to be divided by p_1 to satisfy the constraints of multiple devices.

Notice, we only discuss two basic parallelism schemes for Transformer models. Some recent works may partition the sequence dimension and leverage reduce-scatter and all-gather to reduce the overheads of all-reduce [33, 51]. The communication time can be similarly analyzed, and we will not discuss them here. The optimal parallelism scheme on multiple devices [48, 62, 73, 82, 105] is out of the scope of this article, and we will leave it as future works.

4 CASE STUDY

In this section, we leverage actual hardware configurations to estimate the model performance using our analytical framework and provide insights for LLM accelerator design.

4.1 Overview of Workloads and Hardware

Table 2 lists several widely used models that we choose for performance estimation. Such models include BERT-base [16], a representative encoder-only model, GPT2 [65], the only open-sourced model in the GPT family, LLaMA-7B [70, 71], an open-sourced model trained by Meta, and Vicuna-13B [11], the best non-commercial model on Chatbot Arena [104].

Table 3. Summary of FPGA and GPU Devices

	AMD Xilinx FPGA			Intel FPGA		Nvidia GPU	
	Alveo U280 [83]	**Versal VCK5000 [86]**	**Versal VHK158 [90]**	**Stratix 10 NX 2100 [37]**	**Agilex 7 AGM039 [27]**	**GeForce RTX 2080 Ti**	**Tesla A100**
Process Node	TSMC 16nm	TSMC 7nm	TSMC 7nm	Intel 14nm	Intel 7nm	TSMC 12nm	TSMC 7nm
Release Date	2018	2022	2023	2020	2022	2018	2021
Thermal Design Power	225 W	225 W	180 W	225 W	225 W	250 W	300 W
Peak Throughput	24.5 INT8 TOPS	145 INT8 TOPS	56 INT8 TOPS	143 INT8 TOPS	88.6 INT8 TOPS	14.2 TFLOPS	312 TFLOPS
Specialized Blocks	–	400× AI Engine	–	3,960× AI Tensor Block	–	544× Tensor Cores	432× Tensor Cores
DSP/CUDA Cores	9,024	1,968	7,392	–	12,300	4,352	6,912
BRAM18K/M20K	4,032	967	5,063	6,847	18,960	–	–
URAM/eSRAM	960	463	1,301	2	–	–	–
On-chip Memory Capacity	41 MB	24 MB	63.62 MB	30 MB	46.25 MB	5.5 MB	40 MB
Off-chip Memory Capacity	8 GB HBM2 & 32 GB DDR	16 GB DDR	32 GB HBM2e & 32 GB DDR	16 GB HBM2	32 GB HBM2e	11 GB DDR	80 GB HBM2e
On-chip Memory Bandwidth	460 GB/s & 38 GB/s	102.4 GB/s	819.2 GB/s & 102.4 GB/s	512 GB/s	820 GB/s	616 GB/s	1,935 GB/s

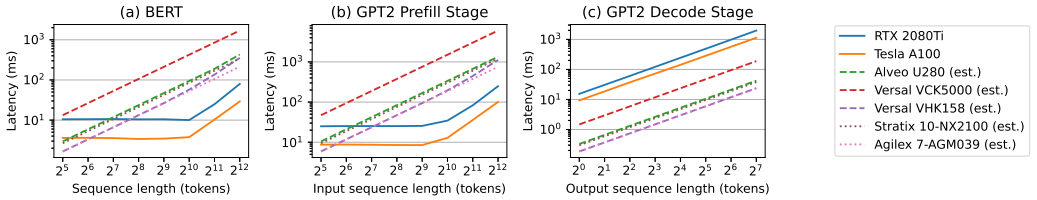


Fig. 6. Latency estimation of BERT and GPT2 on different FPGAs. GPU results are obtained from actual profiling.

To see the performance differences between FPGAs and GPUs, we pick and list several representative devices in Table 3. For FPGAs, Alveo U280 and Agilex 7 are two FPGAs that are widely used in cloud servers but not specially optimized for AI workloads; Versal VCK5000 and Stratix 10 NX FPGAs are designed for accelerating AI applications with specialized hardware units such as **AI Engine (AIE)** [84] or Tensor Blocks [37]. Versal VHK158 is the latest FPGA with HBM2e released by AMD in 2023. For GPUs, RTX 2080Ti is a high-end GPU designed for personal usage with a similar process node and release date to U280; A100 is the most deployed GPU in data centers to conduct LLM training and inference [54].

4.2 Single-device Performance Estimation

We first conduct experiments on a single device with pre-trained BERT-base and GPT2 models from HuggingFace Hub [79]. For GPU baselines, we run the models with PyTorch 2.0 [57] and CUDA 11.7, and measure the performance on both RTX 2080Ti and A100 GPUs. The host machine runs an Intel Xeon Silver 4114 CPU at 2.20 GHz with 40 cores. We follow the common practice to measure the out-of-the-box FP16 performance [2, 53, 65].

In this section, we only estimate the performance of FPGAs using the proposed framework in Section 3, and in Section 6, we will evaluate the actual performance on FPGAs. The quantization scheme is **4-bit weight and 8-bit activation (W4A8)** configurations for FPGA estimations, unless otherwise noted.

4.2.1 Latency of Different Stages. Based on the constraints of Equations (2), (3), and (6), we can calculate the latency one device can achieve at the maximum compute power M (defined in Equation (11)).

From Figures 6(a) and 6(b), we observe that the 2080Ti and A100 GPUs maintain an almost constant curve when the sequence length is less than 1,024. This behavior is primarily attributed to the

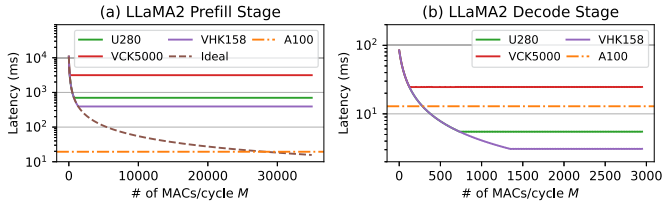


Fig. 7. Latency estimation of LLaMA2 model. The sequence length is set as 128, and the W4A8 quantization scheme is used in this experiment. GPU results are obtained from actual profiling.

high kernel launch overheads for hundreds of CUDA kernels in PyTorch, which overshadow the computation time. However, when the sequence length exceeds 1,024, GPUs become computation-bound, resulting in higher latency. In contrast, the latency of FPGAs increases linearly with the sequence length, as described in Equation (8), and demonstrates significantly longer latency when the sequence length is large (e.g., 512). It is worth noting that even when making use of state-of-the-art AI-optimized FPGAs like the Stratix 10 NX and Versal VCK5000, the situation does not improve significantly. This is because these modern FPGAs are equipped with DDR or older versions of HBM, which makes parameter loading from off-chip memory the bottleneck. Even though these FPGAs have highly efficient computation blocks, the compute units have to wait for the memory to fetch data, resulting in suboptimal performance. Moreover, many FPGAs struggle to achieve both high memory bandwidth and compute power simultaneously. Consequently, for VHK158, performance deteriorates when the sequence length reaches 4,096, as it shifts from being memory-bandwidth bound to compute-bound. Further scaling the sequence length larger than 4096 may lead to out-of-memory for both FPGAs and GPUs, and only A100 can handle such large sequence lengths, so the latency is not plotted on the figure.

The decode stage is on the contrary, as shown in Figure 6(c). When the sequence length is small, GPUs suffer from underutilizing the compute resources, and FPGAs can achieve a significantly lower latency compared to GPUs. The latency of RTX2080 and A100 GPUs increases, since the memory access takes control compared to computation in the decode stage. Although FPGAs are also bounded by off-chip memory bandwidth, they can always perform better than GPUs, since the computation is small when the sequence length is equal to one. Therefore, FPGAs can easily achieve GPU-level performance even with a small M.

Insight I: Existing FPGAs are inferior in the compute-intensive prefill stage but can outperform GPUs in the memory-intensive decode stage.

To further investigate what constrains the performance of FPGAs, we conduct an analysis on the LLaMA2 model by varying different M and observing the changes in latency. As shown in Figure 7, the VCK5000 FPGA exhibits the smallest off-chip memory bandwidth, which leads it to reach a latency plateau rather quickly. Conversely, the VHK158 FPGA has the largest off-chip memory bandwidth, so it can achieve the lowest latency in both prefill and decode stages. Moreover, we include the curve of ideal FPGA performance in Figure 7 to assess the compute power required to attain A100-level performance. Based on this estimation, we need around 30,000 MACs/cycle to achieve the A100-level performance in the prefill stage, assuming no memory bandwidth constraints. This is achievable by those AI-optimized FPGAs, which can conduct a large number of MACs efficiently. On the contrary, for the decode stage, once an FPGA has enough memory bandwidth, such as U280, it can reach the A100-level performance easily.

Insight II: The prefill stage requires large compute power M to achieve the GPU-level performance, while the decode stage only requires a small M.

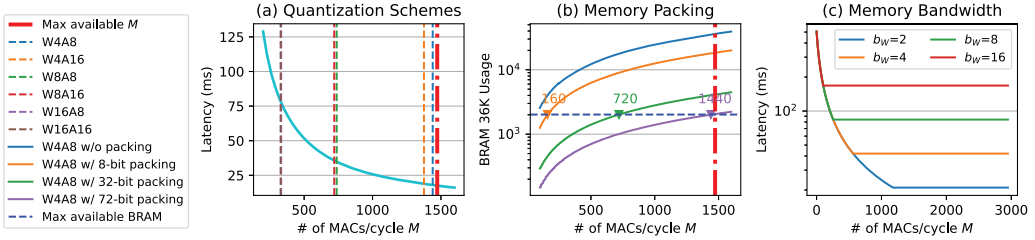


Fig. 8. (a) Impact of different quantization techniques on GPT2 prefilling stage on U280. The sequence length is set as 128. The cyan line shows the theoretical latency under different M without memory bandwidth constraints. Thin dashed lines depict the maximum M constrained by available BRAM resources. (b) Impact of memory packing in the W4A8 setting. (c) Impact of different weight quantization schemes on memory bandwidth and overall latency.

4.2.2 Quantization Schemes. We then investigate the impact of different quantization schemes and memory packing. We consider quantizing the weight parameters to x bits and the activation to y bits (abbreviated as W$\{x\}$A$\{y\}$). As shown in Figure 8(a), the red dashed line depicts the maximum available MACs/cycle on-board, which is calculated based on Equation (2). Different quantization schemes may have different requirements on BRAM usage constrained by Equation (3). W4A8 is the scheme that can almost fully utilize the compute resources. W8A8 and W16A16 require more memory resources, resulting in lower performance, since the computation is bound by the limited BRAM resources on-board. Also, we can see quantizing the weights gives the most benefits, but quantizing activation only gives little benefit (M does not change a lot under the same weight bitwidth), which is due to the fact that we employ a dataflow architecture and do not require large buffers to store the intermediate tensors on-board.

Insight III: Weight quantization is necessary for reducing memory usage, while activation quantization only has limited benefit.

4.2.3 Memory Packing. Next, we further consider the impact of memory packing under the W4A8 setting. As shown in Figure 8(b), if we do not conduct memory packing, then it even cannot satisfy the memory port constraint (Equation (5)) when M is small (blue curve). This is because a large number of partitioned arrays require more BRAMs, and many BRAMs are not fully utilized causing a large waste of resources. The orange curve shows packing two `int4` elements to `int8`, and we can achieve a small M under the resource constraint, since the number of partitioned arrays is reduced. The green curve packs 9×`int4` elements to `int36`, and it can achieve more than four times of M compared to the `int8` packing. The purple curve packs 18×`int4` elements to `int72`, and the curve can almost intersect with the red line before intersecting with the blue line, which means it reaches the maximum DSP constraint on-board (Equation (2)). This study shows that it is important to pack the parameters to reduce on-chip memory usage.

Insight IV: Memory packing can efficiently reduce the required BRAMs to store the tensors.

4.2.4 Memory Bandwidth. Last, we investigate how quantization impacts the required memory bandwidth. As shown in Figure 8(c), the low-bit weight quantization can significantly alleviate the demands of off-chip memory access. By reducing the volume of data needed in each cycle, it can achieve a larger compute power M, thus leading to a better performance. In particular, quantizing the model to a 2-bit representation yields a performance boost exceeding an order of magnitude when compared to a 16-bit weight quantization scheme. Recent research [7, 103] has demonstrated that 4-bit or even 2-bit quantization can be implemented without compromising model accuracy, which makes efficient LLM deployment on FPGAs possible.

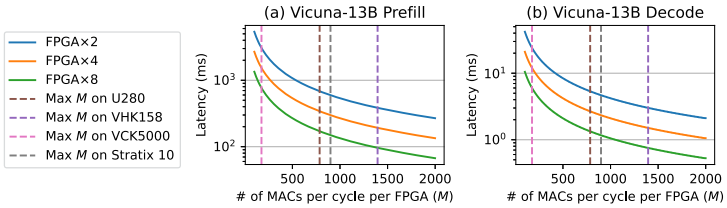


Fig. 9. Latency estimations of Vicuna-13B model on multiple FPGAs.

Insight V: Low-bit weight quantization can further help alleviate the demands of off-chip memory access.

4.3 Multi-device Performance Estimation

For multiple devices, we use the Vicuna-13B model to estimate the performance of 2, 4, and 8 FPGAs based on our analytical model. As shown in Figure 9, the latency can scale well when the number of devices increases. Since we employ a non-blocking communication scheme in our dataflow design as discussed in Section 3.4, communication will not be the bottleneck of the design. Multiple FPGAs can reduce the number of required MACs on each device, but cannot increase the number of available MACs on an FPGA, so the performance is still limited by the maximum available resources on-board and the off-chip memory bandwidth. For the decode stage, leveraging two FPGAs can already reduce the inference latency of the Vicuna-13B model to less than 10 ms based on the estimation.

Insight VI: Multiple FPGAs help reduce overall latency under the same M on each device.

5 IMPLEMENTATIONS

In this section, we describe the kernel implementation and accelerator design to show how to efficiently achieve the design points in the analytical framework.

5.1 HLS Kernels

This section is not intended to propose new HLS kernels. Instead, we explore the efficient ways to reach the maximum available M on FPGAs and implement the standard kernels as a library with parameterized quantization support, which is reusable across different models (e.g., BERT and GPT models implemented in Section 6).

Linear Operators. Linear operators are the key operators in Transformer models, because they are ubiquitous and compute-intensive. There are two types of linear operators in the Transformer layers, **activation-weight matrix multiply (A-W GEMM)** with bias, and **activation-activation (A-A GEMM)** matrix multiply. A-W GEMM includes the projection layers and the linear layers in the FFN, with weights buffered on-chip and activations streamed from an input FIFO; A-A GEMM includes the two GEMM operators in the SDP attention, with one of the input activations stored in a double buffer and the other one streamed.

We adopt an output-stationary systolic array architecture to implement the A-W and A-A GEMM process engines. As shown in Figure 10(a), the systolic array is a two-dimensional array of MAC units of shape $m_1 \times m_2$ with FIFOs connecting different MAC units. The number of MAC units of linear operator i is actually M_i defined in Section 3.1. We mainly discuss the A-W GEMM in the following, and the idea can also be applied to the A-A GEMM. The input activation from the previous layer will be first buffered in an activation buffer. After m_1 elements are buffered,

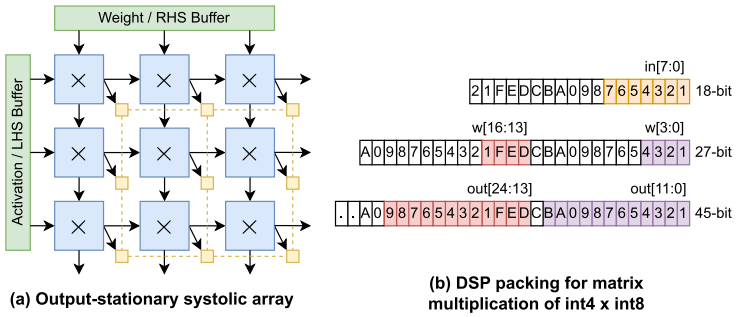


Fig. 10. Systolic array and DSP packing. The yellow blocks in the systolic array represent output buffers.

they will be streamed into the systolic array together with the m_2 weights. There is also a fully partitioned output buffer on-chip that allows the outputs to directly write back.

Each MAC unit can be implemented with a single DSP block and can provide one-MAC-per-cycle throughput. Based on the discussion in Section 4.2.2, we adopt the W4A8 quantization scheme for our accelerator design, which maximizes the utilization of available resources. As a result, the matrix multiplications involve either `int4` by `int8` or `int8` by `int8` operations, which are too large for LUTs, thus relying primarily on DSPs or specialized compute blocks (e.g., AIE [84]). In AMD FPGAs, the DSP48E2 hard blocks can support 18-bit by 27-bit multiplication and accumulation [83], enabling the packing of two multiplications into one slice for a W4A8 quantized model to save DSP resources and achieve a larger M. Figure 10(b) shows the bit packing method for 4-bit by 8-bit integer multiplications. One activation is filled into the lower 8 bits of the 18-bit DSP input, and two weights are filled into 0-to-3 and 13-to-16 bit positions of the 27-bit DSP input to avoid overlapping results. Finally, the two multiplication results are extracted by bit-slicing the 45-bit DSP result. Notice that since the DSP output is wide enough, we can also pack two 8-bit by 8-bit integer multiplications into one DSP slice by further offsetting the second weight and output. With DSP packing, we can easily double M to achieve higher performance but with much fewer DSPs.

Non-linear Operators. Since quantizing non-linear operators can lead to a significant degradation in model accuracy [67, 81], and these non-linear operators are not the bottleneck of the design, we directly implement the floating-point version of the operators in HLS. Specifically, we buffer a row of elements for softmax and LayerNorm functions, which requires conducting reduction along the last dimension. Consequently, this approach eliminates the need to wait for the complete results for the computation of these non-linear operators and effectively prevents dataflow stalling.

5.2 Accelerator Design

We integrate the proposed kernels in Section 5.1 to create a high-performance hardware accelerator. The overall architecture of our proposed accelerator is depicted in Figure 11. Different operators including linear and non-linear operators are connected with FIFOs, except that the KV cache discussed in Section 3.2.2 leverages double buffers. This design reads input tensors from off-chip memory and stores the results back to memory after each layer. Intermediate activation values are directly streamed to the next operator. Initially, all parameters are stored in DRAM, and data loaders $L_{(\cdot)}$ are responsible for loading data from DRAM or streaming buffers. Storing parameters off-chip reduces the number of FPGA devices needed and avoids the need to build different bitstreams for different network layers. After completing a layer, the accelerator fetches new parameters from the host to the device for the subsequent layer. Since data loading is hidden from

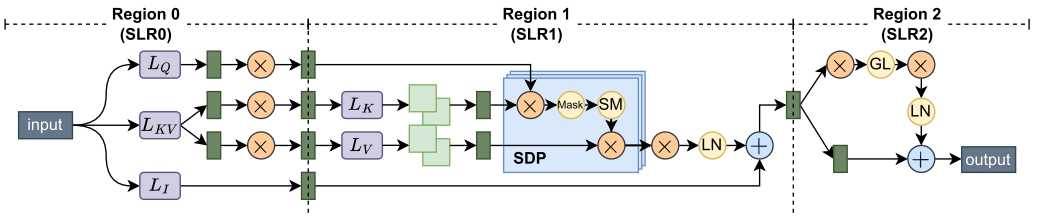


Fig. 11. Overall dataflow architecture of a single Transformer layer that uses post-LayerNorm scheme [65]. SDP denotes scaled dot-product. Orange nodes denote the GEMM kernels. Yellow nodes are the non-linear kernels, including softmax (SM), LayerNorm (LN), and GELU (GL). Green rectangles represent the FIFOs between kernels, and purple rectangles are the data loaders.

the computation, overall latency remains unaffected. Given the contemporary trend of multi-die FPGA designs [18, 22], explicit dataflow partitioning becomes necessary to meet timing requirements. Our target device, Alveo U280 FPGA [83], has three chip dies called **super logic regions (SLRs)**. Consequently, we partition the dataflow into three regions to confine each region fully inside SLR. According to our placement constraints, AMD Vitis toolchain will automatically insert AXI Register Slice IPs to pipeline SLR crossing.

We leverage the proposed analytical framework to guide our accelerator design. Since typical Transformer models have $d_{\text{FFN}} = 4d$ [16, 65] and $l < d$, according to work balancing of Equation (11), we have $M_{q,k,v,p} = M$, $M_{a_1,a_2} < M$, and $M_{f1,f2} = 4M$. A straight-forward division is to put the PEs for q, k, v, SDP, and p on SLR0, $f1$ on SLR1, and $f2$ on SLR2 so that each SLR roughly contains $4M$ MAC units. However, we observe that scaling up the linear operators in FFN poses significant challenges to timing closure. Among various configurations of systolic arrays we tested, the maximum capacity of one SLR at 250 MHz is three of 8×16 systolic arrays; a single 16×16 one fails timing. Therefore, we only leverage 8×8 and 8×16 systolic arrays for simplicity. We also explore using LUT-based multipliers as they provide greater flexibility for placement compared to DSPs. However, the presence of additional inter-LUT wires results in a much lower frequency (191 MHz) compared to the DSP-based multipliers. To minimize the number of SLR crossings, we put q, k, and v on SLR0 and use 8×16 systolic arrays, which also ensures a relatively low latency for the first stage based on Equation (8). MHA and the p projection are on SLR1, with a_1 and a_2 using 8×8, and p using 8×16 systolic arrays. f_1 and f_2 operators on SLR2 using 8×16 systolic arrays. Therefore, it can still form a relatively balanced 3:2:2 resource utilization ratio for linear operators.

6 EVALUATION ON FPGAS

In this section, we implement two design points studied in Section 4 to validate the feasibility of our framework. We first describe our experimental setup and perform evaluation on a single FPGA.

6.1 Experiment Setup

We test the publicly available BERT and GPT2 models listed in Table 2. We conduct post-training quantization and use the W4A8 quantization scheme for BERT and the W8A8 scheme for GPT, which are prevalent settings in nowadays LLM inference [81, 103].

We implement and run the actual design on an Alveo U280 FPGA [83] with $M = 256$ for the kernels. This FPGA has 4,032 BRAM18K blocks, 9024 DSP slices, 2.6M flip-flops, 1.3M LUTs, and 960 URAM blocks. It has three SLRs with an almost equal amount of resources. All the kernels are implemented in C++ with Vitis HLS v2022.1 [89], and synthesized with Vivado backend toolchains.

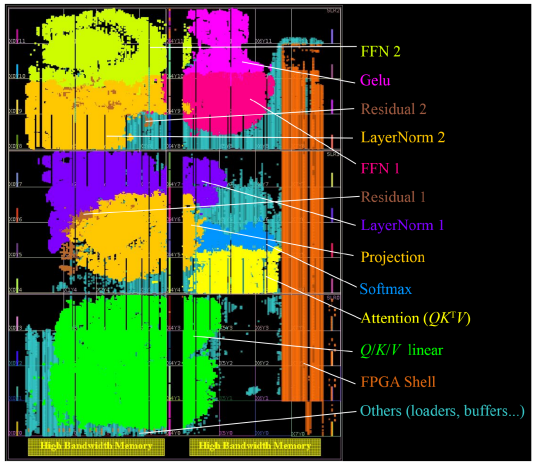


Fig. 12. Physical layout of the implemented spatial accelerator.

Table 4. Experimental Results Compared with Other FPGA-based Accelerators

Name	Device	Freq (MHz)	Quantization	Latency (ms) [Est. (Section 3)]	Throughput (samples/sec)	Speedup	BRAM	DSP	FF	LUT	URAM
Ours	U280	245	W4A8	26.01 [24.07]	**38.45**	–	389 (19%)	1780 (20%)	653K (25%)	569K (44%)	111 (12%)
- SLR0	–	–	–	4.86	–	–	130 (19%)	482 (17%)	200K (23%)	167K (38%)	3 (1%)
- SLR1	–	–	–	14.63	–	–	136 (20%)	590 (19%)	240K (28%)	212K (49%)	50 (16%)
- SLR2	–	–	–	19.81	–	–	123 (18%)	708 (23%)	213K (25%)	191K (44%)	58 (18%)
FQ-BERT [46]	ZCU111	214	W4A8	95.16	10.51	**3.66**×	679 (31%)	3287 (77%)	201K (24%)	190K (45%)	N/A
TRAC [61]	ZCU106	200	Fixed8	347.48	2.88	**13.36**×	181 (29%)	1379 (80%)	128K (28%)	126K (55%)	N/A

Sequence lengths are set as 512.

Figure 12 shows the final device layout of the implemented accelerator. We use OpenCL with **Xilinx RunTime (XRT)** for hardware execution and **Xilinx Board Utility (xbutil)** for computing power measurements. The environment for GPU experiments is listed in Section 4.1, and **NVIDIA system management interface (nvidia-smi)** is used for measuring GPU power. Notice the quantized models on GPUs are slower than the FP16 models, as the quantization methods normally leverage fake quantization and lack high-performance GPU kernels to support efficient inference. Therefore, we directly compare our accelerator with the best FP16 GPU results. The FPGA on-board results match the outputs from the quantized model in PyTorch and are able to achieve the same accuracy. The latency results are the average across 50 runs.

6.2 On-board Evaluation

We first compare our BERT accelerator with FQ-BERT [46] and TRAC [61], two FPGA-based accelerators for the BERT model. To make a fair comparison, we employ the same W4A8 quantization precision with FQ-BERT and assess the model accuracy using the CoLA dataset [16], a widely used language understanding task. The fp16 model achieves an accuracy of 56.84%, while our W4A8 quantized model attains 56.76% accuracy. We only compare the performance of the core encoder layers, and report the best on-board latency results of the baselines obtained from the original papers. Both baselines use a sequence length of 128, so we scale their latency results to match our sequence length of 512. FTrans [39] also targets BERT-variant models, but it does not provide frequency and sequence length in the paper, so we cannot make a direct comparison. From Table 4, we can see our spatial architecture delivers a much lower latency and a higher throughput with a much lower DSP usage. Even though our evaluation device is not exactly the same as

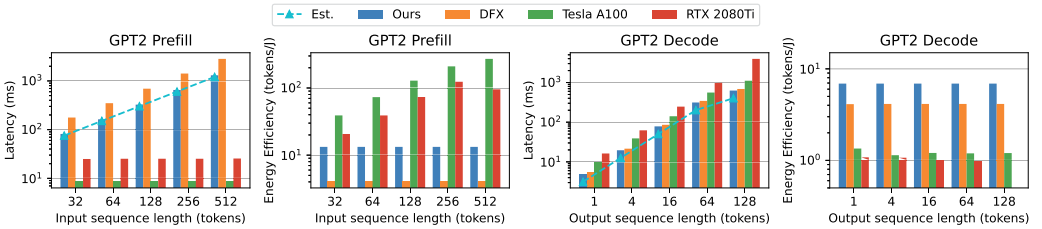


Fig. 13. Latency and energy efficiency of GPT2 model on different devices. The GPU results are obtained following the same setting in Section 4.

the baselines, our throughput improvement still surpasses the resource increment of FF and LUT. Specifically, our accelerator is 3.66× faster than FQ-BERT [46] and 13.36× faster than TRAC [61]. Compared to temporal architectures, our spatial architecture can efficiently overlap the execution of the operators in the model and eliminate most of the data movement overhead. In our design, one layer can start operation once the previous layer finishes computing one tile of the feature map, which typically takes only tens of cycles. In contrast, temporal architectures need to wait for the entire tensor to be produced, which may take hundreds of cycles. Therefore, even if the per-layer latency of spatial architectures is longer, the end-to-end latency can still be significantly lower than the temporal architectures employed by FQ-BERT and TRAC. Furthermore, our analytical framework precisely predicts the performance of the accelerator with less than 2ms of differences, showing the practicality of our approach.

We next design an accelerator for the GPT2 model. We support importing quantized models from different quantization frameworks [7, 81, 103]. Specifically, we export the W8A8 model from SmoothQuant [81] and achieve 62.2% on the LAMBADA dataset [56], whereas the FP16 model demonstrates an accuracy of 65.0%. We compare our GPT accelerator with the state-of-the-art GPT accelerator, DFX [23], which employs a temporal architecture with an instruction set and uses the same U280 FPGA device for on-board evaluation. On average, we are 2.16× and 1.10× faster than DFX in the prefill and decode stage, respectively. This is because our spatial architecture overlaps the computation and greatly eliminates off-chip memory access. We can also see our estimations in Section 3 align closely with the actual performance, achieving a 92% accuracy for the prefill stage. For the decode stage, the estimated latencies are lower than the actual results, which is mainly because the initial interval between two operators is not significantly smaller than the execution time of one stage, contributing to a notable increase in latency.

We also include the GPU results in Section 4 for a more comprehensive evaluation. As shown in Figure 13, neither DFX nor our design performs well during the prefill stage compared to GPUs that have more compute resources to exploit the abundant parallelism. Notably, the latency of FPGAs in the prefill stage increases linearly, while the GPU ones almost remain constant as the model does not fully utilize GPUs. For the decode stage, the situation is reversed. FPGA-based accelerators are more efficient than GPUs, and our accelerator can achieve a 1.85× speedup and is 5.69× more energy efficient compared to the A100 GPU. This is because the generation of each token is fully sequential, and GPUs cannot leverage their abundant parallelism, and suffer from extensive memory access overheads. On the contrary, our dataflow accelerator eliminates most of the off-chip memory accesses and overlaps the compute as much as possible. Thus, we can achieve a better performance compared to GPUs, aligning with our estimation results in Section 4. Notice the U280 FPGA only uses a 16 nm process while the A100 GPU has a more advanced 7nm process node based on the data in Table 3, but we can still achieve higher speedup, demonstrating the efficiency of our spatial accelerators. It also indicates the potential of further optimizing our HLS design and scaling it up to achieve even higher performance.

Table 5. Latency and Resource Usage of Our Systolic Array Library Function

	Latency (ms)	Effective M	BRAM	DSP	FF	LUT
Ours (w/o DSP packing)	15.73	256	0 (0%)	256 (2%)	88,284 (3%)	168,190 (12%)
Ours (w/ DSP packing)	15.73	128	0 (0%)	**128 (1%)**	79,969 (3%)	244,439 (18%)
AutoSA [75]	15.71	N/A	514 (12%)	256 (2%)	100,138 (3%)	244,032 (18%)

Results are directly derived from the HLS report in 300 MHz. The GEMM kernel is extracted from the first FFN layer in the BERT-base model with size $(512, 768) \times (768, 3072)$. We use a 16×16 systolic array to calculate the `int8` GEMM. The theoretical peak performance without DSP packing is $(512 \times 768 \times 3072)/(16 \times 16)$ cycles $\times 3.33$ ns/cycle = 15.71 ms.

Table 6. Performance and Resource Usage of Non-linear Operators in Our Kernel Library

Operator	Latency (ms)	BRAM	DSP	FF	LUT
Softmax	6.67	8	38	4,835	7,447
LayerNorm	0.85	20	80	18,751	12,746
GeLU	0.67	0	256	26,193	16,472

Kernel sizes are set to match those of the BERT model in Table 4. Results are directly derived from the HLS report in 300 MHz.

6.3 Ablation Study

We begin by examining the latency of different SLRs. As shown in rows 3–5 in Table 4, the overall latency of the BERT accelerator is nearly the sum of the latency of SLR0 and SLR2, aligning with the pipeline diagram in Figure 4. Moreover, the computation in SLR1 largely overlaps with that of SLR2 due to the fully pipelined design. The resource usage across different SLRs is also similar, resulting in a balanced design.

We further investigate the efficiency of our kernel functions. We conduct experiments on our template systolic array function with the AutoSA-generated systolic array, which is a highly optimized state-of-the-art systolic array implementation [75]. From Table 5, we can see our implementation achieves the same level of performance compared to AutoSA. While maintaining the same DSP usage, the resource usage of our kernel function is much smaller than AutoSA. Since the prediction of a single GEMM kernel is accurate and can achieve the theoretical performance, our analytical model can precisely predict the performance of spatial accelerators when combining multiple linear operators. The presence of latency-predictable kernels as foundational components plays a pivotal role in this predictability. Additionally, our function offers enhanced customizability, accommodating varying sizes and the choice of different quantization schemes.

Moreover, employing DSP packing further reduces the DSP usage, allowing one DSP to handle two MAC operations within a single cycle, a feature not supported in AutoSA. This experiment shows the efficiency of our kernels, facilitating the development of high-performance Transformer accelerators.

Last, we analyze the performance of the non-linear operators. As shown in Table 6, we observe that the softmax operator in the MHA module incurs the highest latency, primarily due to the need to compute the exponential function. Since these operators are elementwise and only require a row of data to start the computation, they can be easily fused with the preceding linear operators in the pipeline, thereby not significantly impacting the overall latency. For instance, the combined latency of a GEMM kernel (10.77 ms) and the softmax operator (6.67 ms) greatly exceeds the latency of SLR1 in Table 4 (14.63 ms/300 MHz×245 MHz), indicating substantial overlap between the softmax operator and other operators. Again, these ablation studies show that considering only the linear operators in the analytical framework is sufficient to achieve an accurate latency estimation.

7 DISCUSSION

In the previous sections, we provide details of the analytical framework and prove that it can achieve high accuracy compared to the latency of actual implementation. However, our framework may have limitations when analyzing the overlay designs or compressed models with sparsity, which requires changes in the resource and latency estimation. In this section, we will delve into several unanswered questions and open challenges.

AI-optimized FPGAs. In Section 4, we demonstrate the potential of leveraging FPGAs with specialized compute engines to accelerate LLMs. Although AIEs and tensor blocks provide massive compute power [37, 84], the memory layout and bandwidth requirements remain undiscovered. Future FPGAs for AI workloads should provide enough memory bandwidth and efficient on-chip interconnect to facilitate the local data movements in a spatial architecture. Moreover, these specialized hardware blocks usually adopt a unique programming model with custom compilation flows. It is still an open question whether existing practices for programming those hardware blocks enable efficient execution of Transformer models.

Timing Closure on Multi-die FPGAs. We encounter timing problems in partitioning and scaling our design in Section 5.2. In general, it is hard to adequately explore the design space of multi-die partitioning and scaling. There are automated frameworks [22, 29] to generate floorplanning constraints, but they are currently not expressive enough to capture the various data movement schemes (e.g., residual connection, multi-head splitting) within Transformer models. We hope similar tools for Transformers could be derived from our analytical framework to speed up the design closure.

Heterogeneous Deployment. Nowadays, data centers are increasingly heterogeneous, with CPUs, GPUs, and FPGAs available at scale [6, 12, 51]. Therefore, it is possible to leverage the advantages of different hardware to accelerate Transformer models. For example, GPUs are good for the GPT prefill stage due to their high compute power; FPGAs can achieve low-latency decode stage with customized spatial architecture. The key challenge is to build a distributed system that efficiently manages hundreds of heterogeneous devices. We hope our analysis on resource constraints, latency, and scaling could assist future deployment and evaluation of LLMs in a heterogeneous and distributed environment.

8 RELATED WORK

FPGA-based Transformer Accelerators. Most of the prior works on hardware accelerators leverage temporal or overlay architecture with one FPGA [26, 28, 39, 40, 46, 59, 63, 64]. Their performance usually suffers from frequent data movements of intermediate results. DFX [23] explores using multiple FPGAs to accelerate GPT2 inference, but it is still an overlay design. Some research has delved into software-hardware co-design to optimize the attention kernel [99]. These endeavors often lack in-depth analysis on resource utilization and cannot be easily generalized to other kernels.

Quantization on LLMs. Initial investigations [15, 81, 94] demonstrate lossless 8-bit quantization for LLMs. Subsequent studies [21, 31, 44, 94, 96, 103] keep lowering the bit width; the latest advancements reveal that 2-bit [7] and even 1-bit (binary) quantization [101] are adequate for an accurate LLM. While these approaches offer valuable insights, our focus remains orthogonal to quantization, as we illustrate optimization techniques and provide high-performance building blocks for deploying quantized LLMs on FPGAs.

HLS Kernel Libraries. Despite the existence of kernel libraries for accelerating Transformer models on GPUs [14, 38, 79], the hardware domain has seen only a handful of initiatives in this regard. AMD provides Vitis HLS library [87, 88] that only has basic kernel-level examples without

comprehensive designs tailored for Transformer models. TRAC [61] attempts to provide an HLS-based Transformer library, but its kernel performance is unpredictable, and it exclusively focuses on the BERT model using a temporal architecture. Some frameworks map deep learning frameworks to FPGAs [4, 20, 72, 98, 100], but can only handle small CNN designs and do not cater to LLMs. More recent tools allow hardware design using Python [24, 35, 55, 80, 95], but are still general-purpose and require hardware engineers to construct and optimize kernels from scratch. Our work provides a Transformer kernel library designed for dataflow implementations and demonstrates their composability in constructing high-performance hardware accelerators.

9 CONCLUSION

In this article, we propose an analytical framework for large language models and point out the bottlenecks and potential optimizations across the prefill and decode stages in the generative inference. To verify the feasibility of our framework, we provide a reusable HLS kernel library to quickly compose Transformer kernels into different LLMs that can achieve the expected performance. Based on these proposed kernels, we design FPGA-based spatial accelerators for both BERT and GPT models and achieve high performance and energy efficiency on par with high-end GPUs. By offering insights into performance bottlenecks, a suite of reusable kernels, and a high-performance accelerator, we propel the deployment of LLMs for real-world applications while pushing the boundaries of hardware innovation.

ACKNOWLEDGMENTS

We thank anonymous reviewers, Keisuke Kamahori, and Zihao Ye for providing insightful feedback. We also thank Jiajie Li, Jie Liu, and Zhanqiu Hu for their contributions to the initial LLM modeling and benchmarking.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*.

[2] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-inference: Enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*.

[3] Suhail Basalama, Atefeh Sohrabizadeh, Jie Wang, Licheng Guo, and Jason Cong. 2023. FlexCNN: An end-to-end framework for composing CNN accelerators on FPGA. *ACM Trans. Reconfig. Technol. Syst.* 16, 2, Article 23 (Mar. 2023), 32 pages. https://doi.org/10.1145/3570928

[4] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. 2018. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Trans. Reconfig. Technol. Syst.* 11, 3 (2018), 1–23.

[5] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill et al. 2021. On the opportunities and risks of foundation models. Retrieved from https://arXiv:2108.07258

[6] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783710

[7] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. 2023. QuIP: 2-bit quantization of large language models with guarantees. Retrieved from https://arXiv:2307.13304

[8] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating large language model decoding with speculative sampling. Retrieved from https://arXiv:2302.01318

[9] Hongzheng Chen, Cody Hao Yu, Shuai Zheng, Zhen Zhang, Zhiru Zhang, and Yida Wang. 2024. Slapo: A schedule language for progressive optimization of large deep learning model training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. Retrieved from ttps://arXiv:2107.03374

[11] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. Retrieved from https://lmsys.org/blog/2023-03-30-vicuna/

[12] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ahmad El Husseini, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Rapsang, Steven Reinhardt, Bita Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug Burger. 2018. Serving DNNs in real time at datacenter scale with project brainwave. *IEEE Micro* 38, 2 (2018), 8–20. https://doi.org/10.1109/MM.2018.022071131

[13] Aaron Daniel Cohen, Adam Roberts, Alejandra Molina, Alena Butryna, Alicia Jin, Apoorv Kulshreshtha, Ben Hutchinson, Ben Zevenbergen, Blaise Hilary Aguera-Arcas, Chung ching Chang, Claire Cui, Cosmo Du, Daniel De Freitas Adiwardana, Dehao Chen, Dmitry (Dima) Lepikhin, Ed H. Chi, Erin Hoffman-John, Heng-Tze Cheng, Hongrae Lee, Igor Krivokon, James Qin, Jamie Hall, Joe Fenton, Johnny Soraker, Kathy Meier-Hellstern, Kristen Olson, Lora Mois Aroyo, Maarten Paul Bosma, Marc Joseph Pickett, Marcelo Amorim Menegali, Marian Croak, Mark Díaz, Matthew Lamm, Maxim Krikun, Meredith Ringel Morris, Noam Shazeer, Quoc V. Le, Rachel Bernstein, Ravi Rajakumar, Ray Kurzweil, Romal Thoppilan, Steven Zheng, Taylor Bos, Toju Duke, Tulsee Doshi, Vincent Y. Zhao, Vinodkumar Prabhakaran, Will Rusch, YaGuang Li, Yanping Huang, Yanqi Zhou, Yuanzhong Xu, and Zhifeng Chen. 2022. LaMDA: Language models for dialog applications. Retrieved from https://arXiv:2201.08239

[14] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. Retrieved from https://arXiv:2205.14135

[15] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.).

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. Retrieved from https://arXiv:1810.04805

[17] Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, Yevgen Chebotar, Pierre Sermanet, Daniel Duckworth, Sergey Levine, Vincent Vanhoucke, Karol Hausman, Marc Toussaint, Klaus Greff, Andy Zeng, Igor Mordatch, and Pete Florence. 2023. PaLM-E: An embodied multimodal language model. In Retrieved from https://arXiv:2303.03378

[18] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. 2022. High-performance sparse linear algebra on HBM-equipped FPGAs using HLS: A case study on SpMV. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'22)*.

[19] ELS-RD. 2022. kernl.ai. Retrieved from https://github.com/ELS-RD/kernl

[20] Farah Fahim, Benjamin Hawks, Christian Herwig, James Hirschauer, Sergo Jindariani, Nhan Tran, Luca P. Carloni, Giuseppe Di Guglielmo, Philip Harris, Jeffrey Krupa, Dylan Rankin, Manuel Blanco Valentin, Josiah Hester, Yingyi Luo, John Mamish, Seda Orgrenci-Memik, Thea Aarrestad, Hamza Javed, Vladimir Loncar, Maurizio Pierini, Adrian Alan Pol, Sioni Summers, Javier Duarte, Scott Hauck, Shih-Chieh Hsu, Jennifer Ngadiuba, Mia Liu, Duc Hoang, Edward Kreinar, and Zhenbin Wu. 2021. hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices. https://arxiv.org/abs/2103.05579

[21] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate post-training compression for generative pretrained transformers. Retrieved from https://arXiv:2210.17323

[22] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'21)*. ACM, New York, NY, 81–92. https://doi.org/10.1145/3431920.3439289

[23] Seongmin Hong, Seungjae Moon, Junsoo Kim, Sungjae Lee, Minsub Kim, Dongsoo Lee, and Joo-Young Kim. 2022. DFX: A low-latency multi-FPGA appliance for accelerating transformer-based text generation. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO'22)*. 616–630. https://doi.org/10.1109/MICRO56248.2022.00051

[24] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, and Wen-Mei Hwu. 2021. PyLog: An algorithm-centric python-based FPGA programming and synthesis flow. *IEEE Trans. Comput.* 70, 12 (2021), 2015–2028. https://doi.org/10.1109/TC.2021.3123465

[25] HuggingFace. 2023. Text Generation Strategies. Retrieved from https://huggingface.co/docs/transformers/generation_strategies

[26] Suyeon 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. Archit. Code Optim.* 20, 1, Article 11 (Feb. 2023), 24 pages. https://doi.org/10.1145/3564606

[27] Intel. 2022. Intel Agilex 7 FPGA and SoC FPGA. Retrieved from https://www.intel.com/content/www/us/en/products/details/fpga/agilex/7.html

[28] Hamza Khan, Asma Khan, Zainab Khan, Lun Bin Huang, Kun Wang, and Lei He. 2021. NPE: An FPGA-based overlay processor for natural language processing. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'21)*. ACM, New York, NY, 227. https://doi.org/10.1145/3431920.3439477

[29] Moazin Khatti, Xingyu Tian, Yuze Chi, Licheng Guo, Jason Cong, and Zhenman Fang. 2023. PASTA: Programming and automation support for scalable task-parallel HLS programs on modern multi-die FPGAs. In *Proceedings of the IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'23)*. 12–22. https://doi.org/10.1109/FCCM57271.2023.00011

[30] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. I-BERT: Integer-only BERT quantization. In *Proceedings of the International Conference on Machine Learning (ICML'21)*.

[31] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael Mahoney, and Kurt Keutzer. 2023. SqueezeLLM: Dense-and-sparse quantization. Retrieved from https://arxiv.org/abs/2306.07629

[32] Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W. Mahoney, Yakun Sophia Shao, and Amir Gholami. 2023. Full stack optimization of transformer inference: A survey. Retrieved from https://arXiv:2302.14017

[33] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Reducing activation recomputation in large transformer models. Retrieved from https://arXiv:2205.05198

[34] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[35] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[36] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A lite BERT for self-supervised learning of language representations. In *Proceedings of the International Conference on Learning Representations*.

[37] Martin Langhammer, Eriko Nurvitadhi, Bogdan Pasca, and Sergey Gribok. 2021. Stratix 10 NX architecture and applications. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'21)*. ACM, New York, NY, 57–67. https://doi.org/10.1145/3431920.3439293

[38] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, and Daniel Haziza. 2022. xFormers: A Modular and Hackable Transformer Modelling Library. Retrieved from https://github.com/facebookresearch/xformers

[39] 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 *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED'20)*. ACM, New York, NY, 175–180. https://doi.org/10.1145/3370748.3406567

[40] Qin Li, Xiaofan Zhang, Jinjun Xiong, Wen-Mei Hwu, and Deming Chen. 2021. Efficient methods for mapping neural machine translator on FPGAs. *IEEE Trans. Parallel Distrib. Syst.* 32, 7 (2021), 1866–1877. https://doi.org/10.1109/TPDS.2020.3047371

[41] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.* (2020). https://dl.acm.org/doi/10.14778/3415478.3415530

[42] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158

[43] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*. USENIX Association, Boston, MA, 663–679. Retrieved from https://www.usenix.org/conference/osdi23/presentation/li-zhouhan

[44] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. 2023. AWQ: Activation-aware weight quantization for LLM compression and acceleration. Retrieved from https://arXiv:2306.00978

[45] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. Retrieved from https://arXiv:1907.11692

[46] Zejian Liu, Gang Li, and Jian Cheng. 2021. Hardware acceleration of fully quantized BERT for efficient natural language processing. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'21)*. 513–516.

[47] Meta. 2021. Fully Sharded Data Parallel: Faster AI Training with Fewer GPUs. Retrieved from https://engineering.fb.com/2021/07/15/open-source/fsdp/

[48] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. 2022. Galvatron: Efficient transformer training over multiple GPUs using automatic parallelism. *Proc. VLDB Endow.* 16, 3 (Nov. 2022), 470–479. https://doi.org/10.14778/3570690.3570697

[49] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*.

[50] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-efficient pipeline-parallel DNN training. In *Proceedings of the 38th International Conference on Machine Learning*.

[51] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on GPU clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[52] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An open large language model for code with multi-turn program synthesis. In *Proceedings of the 11th International Conference on Learning Representations*.

[53] Nvidia. 2022. FasterTransformer. Retrieved from https://github.com/NVIDIA/FasterTransformer

[54] OpenAI. 2023. GPT-4 technical report. Retrieved from https://arXiv:2303.08774

[55] Debjit Pal, Yi-Hsiang Lai, Shaojie Xiang, Niansong Zhang, Hongzheng Chen, Jeremy Casas, Pasquale Cocchini, Zhenkun Yang, Jin Yang, Louis-Noël Pouchet, and Zhiru Zhang. 2022. Accelerator design with decoupled hardware customizations: Benefits and challenges: Invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC '22)*. ACM, New York, NY, 1351–1354. https://doi.org/10.1145/3489517.3530681

[56] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and R. Fernández. 2016. The LAMBADA dataset: Word prediction requiring a broad discourse context. Retrieved from https://arXiv:1606.06031

[57] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*.

[58] Hongwu Peng, Shaoyi Huang, Shiyang Chen, Bingbing Li, Tong Geng, Ang Li, Weiwen Jiang, Wujie Wen, Jinbo Bi, Hang Liu, and Caiwen Ding. 2022. A length adaptive algorithm-hardware co-design of transformer on FPGA through sparse attention and dynamic pipelining. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC'22)*. ACM, New York, NY, 1135–1140. https://doi.org/10.1145/3489517.3530585

[59] Hongwu Peng, Shaoyi Huang, Tong Geng, Ang Li, Weiwen Jiang, Hang Liu, Shusen Wang, and Caiwen Ding. 2021. Accelerating transformer-based deep learning models on FPGAs using column balanced block pruning. In *Proceedings of the 22nd International Symposium on Quality Electronic Design (ISQED'21)*. 142–148. https://doi.org/10.1109/ISQED51717.2021.9424344

[60] Lucian Petrica, Tobías Alonso, Mairin Kroes, Nicholas J. Fraser, Sorin Dan Cotofana, and Michaela Blott. 2020. Memory-efficient dataflow inference for deep CNNs on FPGA. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT'20)*. 48–55.

[61] Patrick Plagwitz, Frank Hannig, and Jürgen Teich. 2022. TRAC: Compilation-based design of transformer accelerators for FPGAs. In *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications (FPL'22)*.

[62] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems*.

[63] Panjie Qi, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Hongwu Peng, Shaoyi Huang, Zhenglun Kong, Yuhong Song, and Bingbing Li. 2021. Accelerating framework of transformer by hardware design and model compression co-optimization. In *Proceedings of the IEEE/ACM International Conference On Computer Aided Design (ICCAD'21)*. IEEE Press, 1–9. https://doi.org/10.1109/ICCAD51958.2021.9643586

[64] Panjie Qi, Yuhong Song, Hongwu Peng, Shaoyi Huang, Qingfeng Zhuge, and Edwin Hsing-Mean Sha. 2021. Accommodating transformer onto FPGA: Coupling the balanced model compression and FPGA-implementation optimization. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI'21)*. ACM, New York, NY, 163–168. https://doi.org/10.1145/3453688.3461739

[65] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI Blog* 1, 8 (2019), 9.

[66] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20)*.

[67] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. 2020. Q-BERT: Hessian based ultra low precision quantization of BERT. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*. AAAI Press, 8815–8821.

[68] 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. Retrieved from https://arXiv:1909.08053

[69] Mengshu Sun, Zhengang Li, Alec Lu, Haoyu Ma, Geng Yuan, Yanyue Xie, Hao Tang, Yanyu Li, Miriam Leeser, Zhangyang Wang, Xue Lin, and Zhenman Fang. 2022. FPGA-Aware automatic acceleration framework for vision transformer with mixed-scheme quantization: Late breaking results. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC'22)*. ACM, New York, NY, 1394–1395. https://doi.org/10.1145/3489517.3530618

[70] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and efficient foundation language models. Retrieved from https://arXiv:2302.13971.

[71] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. Retrieved from https://arXiv:2307.09288

[72] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. ACM, 65–74.

[73] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. 2022. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and*

Implementation (OSDI'22). USENIX Association, Carlsbad, CA, 267–284. Retrieved from https://www.usenix.org/conference/osdi22/presentation/unger

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

[75] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'21)*. ACM, New York, NY, 93–104. https://doi.org/10.1145/3431920.3439292

[76] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. 2023. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*. ACM, New York, NY, 93–106. https://doi.org/10.1145/3567955.3567959

[77] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent abilities of large language models. *Trans. Mach. Learn. Res.* (2022). Retrieved from https://openreview.net/forum?id=yzkSU5zdwD

[78] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, 24824–24837.

[79] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. Retrieved from https://arXiv:1910.03771

[80] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. 2022. HeteroFlow: An accelerator programming model with decoupled data placement for software-defined FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[81] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*.

[82] Ningning Xie, Tamara Norman, Dominik Grewe, and Dimitrios Vytiniotis. 2022. Synthesizing optimal parallelism placement and reduction strategies on hierarchical systems for deep learning. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 548–566. Retrieved from https://proceedings.mlsys.org/paper_files/paper/2022/file/f0f9e98bc2e2f0abc3e315eaa0d808fc-Paper.pdf

[83] AMD Xilinx. 2021. Alveo U280 Data Center Accelerator Card. Retrieved from https://www.xilinx.com/products/boards-and-kits/alveo/u280.html#specifications

[84] AMD Xilinx. 2022. *AI Engines and Their Applications*. White Paper. AMD Xilinx.

[85] AMD Xilinx. 2022. QSFP Module Connector. Retrieved from https://docs.xilinx.com/r/en-US/ug1411-vmk180-eval-bd/QSFP-Module-Connector

[86] AMD Xilinx. 2022. VCK5000 Versal Development Card. Retrieved from https://www.xilinx.com/products/boards-and-kits/vck5000.html#specs

[87] AMD Xilinx. 2022. Vitis Accelerated Libraries. Retrieved from https://github.com/Xilinx/Vitis_Libraries

[88] AMD Xilinx. 2022. Vitis AI: Adaptable & Real-Time AI Inference Acceleration. Retrieved from https://github.com/Xilinx/Vitis-AI

[89] AMD Xilinx. 2022. Vitis HLS v2022.1. Retrieved from https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html

[90] AMD Xilinx. 2023. Versal VHK158. Retrieved from https://www.xilinx.com/products/boards-and-kits/vhk158.html

[91] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. 2020. On layer normalization in the transformer architecture. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. JMLR.org, Article 975, 10 pages.

[92] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Re, Christopher Aberger, and Christopher De Sa. 2021. PipeMare: Asynchronous pipeline parallel DNN training. In *Proceedings of Machine Learning and Systems*.

[93] Zhuoping Yang, Jinming Zhuang, Jiaqi Yin, Cunxi Yu, Alex K. Jones, and Peipei Zhou. 2023. AIM: Accelerating arbitrary-precision integer multiplication on heterogeneous reconfigurable computing platform versal ACAP. In *Proceedings of the IEEE/ACM International Conference On Computer Aided Design (ICCAD'23)*.

[94] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. 2022. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Adv. Neural Info. Process. Syst.* 35 (2022), 27168–27183.

[95] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A new scalable high-level synthesis framework on multi-level intermediate representation. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA'22)*.

[96] Zhihang Yuan, Lin Niu, Jiawei Liu, Wenyu Liu, Xinggang Wang, Yuzhang Shang, Guangyu Sun, Qiang Wu, Jiaxiang Wu, and Bingzhe Wu. 2023. RPTQ: Reorder-based post-training quantization for large language models. Retrieved from https://arXiv:2304.01089

[97] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*. ACM, New York, NY, 161–170. https://doi.org/10.1145/2684746.2689060

[98] 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 *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)*. 1–8. https://doi.org/10.1145/3240765.3240801

[99] Xinyi Zhang, Yawen Wu, Peipei Zhou, Xulong Tang, and Jingtong Hu. 2021. Algorithm-hardware co-design of attention mechanism on FPGA devices. *ACM Trans. Embed. Comput. Syst.* 20, 5s, Article 71 (Sep. 2021), 24 pages. https://doi.org/10.1145/3477002

[100] 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 *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD'20)*. ACM, New York, NY, Article 61, 9 pages. https://doi.org/10.1145/3400302.3415609

[101] Yichi Zhang, Ankush Garg, Yuan Cao, Łukasz Lew, Behrooz Ghorbani, Zhiru Zhang, and Orhan Firat. 2023. Binarized neural machine translation. Retrieved from https://arXiv:2302.04907

[102] Yichi Zhang, Junhao Pan, Xinheng Liu, Hongzheng Chen, Deming Chen, and Zhiru Zhang. 2021. FracBNN: Accurate and FPGA-efficient binary neural networks with fractional activations. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[103] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. 2023. Atom: Low-bit quantization for efficient and accurate LLM serving. Retrieved from https://arXiv:2310.19102

[104] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. Retrieved from https://arxiv:2306.05685

[105] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. USENIX Association, 559–578. Retrieved from https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

Received 21 December 2023; revised 22 February 2024; accepted 19 March 2024