FlightLLM: Efficient Large Language Model Inference with a Complete Mapping Flow on FPGAs

Shulin Zeng[*]
Tsinghua University
Infinigence-AI

Jun Liu[*]
Shanghai Jiao Tong University
Infinigence-AI

Guohao Dai[†]
Shanghai Jiao Tong University
Infinigence-AI

Xinhao Yang
Tsinghua University
Infinigence-AI

Tianyu Fu
Tsinghua University
Infinigence-AI

Hongyi Wang
Tsinghua University
Infinigence-AI

Wenheng Ma
Tsinghua University

Hanbo Sun
Tsinghua University

Shiyao Li
Tsinghua University
Infinigence-AI

Zixiao Huang
Tsinghua University

Yadong Dai
Infinigence-AI

Jintao Li
Infinigence-AI

Zehao Wang
Infinigence-AI

Ruoyu Zhang
Infinigence-AI

Kairui Wen
Infinigence-AI

Xuefei Ning
Tsinghua University

Yu Wang[†]
Tsinghua University

ABSTRACT

Transformer-based Large Language Models (LLMs) have made a significant impact on various domains. However, LLMs' efficiency suffers from both heavy computation and memory overheads. Compression techniques like sparsification and quantization are commonly used to mitigate the gap between LLM's computation/memory overheads and hardware capacity. However, existing GPU and transformer-based accelerators cannot efficiently process compressed LLMs, due to the following unresolved challenges: low computational efficiency, underutilized memory bandwidth, and large compilation overheads.

This paper proposes **FlightLLM**, enabling efficient LLMs inference with a complete mapping flow on FPGAs. In FlightLLM, we highlight an innovative solution that the computation and memory overhead of LLMs can be solved by utilizing FPGA-specific resources (*e.g.*, DSP48 and heterogeneous memory hierarchy). We propose a configurable sparse DSP chain to support different sparsity patterns with high computation efficiency. Second, we propose an always-on-chip decode scheme to boost memory bandwidth with mixed-precision support. Finally, to make FlightLLM available for real-world LLMs, we propose a length adaptive compilation method to reduce the compilation overhead. Implemented on the Xilinx Alveo U280 FPGA, FlightLLM achieves 6.0× higher energy efficiency and 1.8× better cost efficiency against commercial GPUs (*e.g.*, NVIDIA V100S) on modern LLMs (*e.g.*, LLaMA2-7B) using vLLM and SmoothQuant under the batch size of one. FlightLLM beats NVIDIA A100 GPU with 1.2× higher throughput using the latest Versal VHK158 FPGA.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; • **Computer systems organization → Reconfigurable computing**.

KEYWORDS

Large Language Model, Inference, FPGA, Mapping Flow

ACM Reference Format:
Shulin Zeng, Jun Liu, Guohao Dai, Xinhao Yang, Tianyu Fu, Hongyi Wang, Wenheng Ma, Hanbo Sun, Shiyao Li, Zixiao Huang, Yadong Dai, Jintao Li, Zehao Wang, Ruoyu Zhang, Kairui Wen, Xuefei Ning, and Yu Wang. 2024. FlightLLM: Efficient Large Language Model Inference with a Complete Mapping Flow on FPGAs. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24), March 3–5, 2024, Monterey, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3626202.3637562

1 INTRODUCTION

Recently, we have witnessed the rapid development and significant impact of Large Language Models (LLMs) [5, 48]. LLMs demonstrate amazing power to understand all the users' input requests (*prefill* stage) and generate accurate responses token-by-token (*decode*

[*] Both authors contributed equally to this research.
[†] Corresponding authors. Email: daiguohao@sjtu.edu.cn, yu-wang@tsinghua.edu.cn

FPGA '24, March 3–5, 2024, Monterey, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0418-5/24/03
https://doi.org/10.1145/3626202.3637562

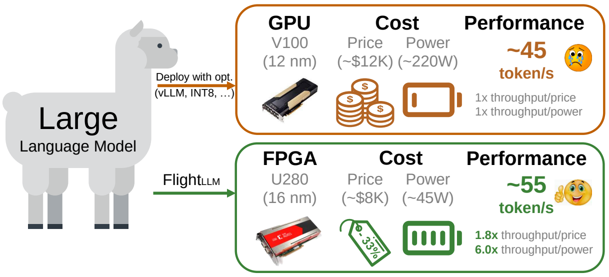


Figure 1: FlightLLM on Alveo U280 FPGA outperforms NVIDIA V100S GPU (using vLLM [31] and SmoothQuant [49]) with better performance and cost efficiency.

stage). LLMs are being widely used in latency-sensitive scenarios [36], such as code completion [42], real-time chatbots [7, 40], customer support [26], online legal advice [12], and beyond. The latency is critical for a good user experience, and the batch size is usually set as 1 to meet the real-time requirement. However, current LLMs suffer from both heavy computation and memory overheads because of the explosive growth model size of LLMs. Taking GPT-3 [6] as an example, it has 175 billion parameters (*i.e.*, 350GB in FP16), requiring about 660TOPS of computation amount to complete a single inference.

Model compression methods [13] (*e.g.*, sparsification, quantization, etc.) are commonly applied to address the above issues. However, the unique computation schemes of these methods are not efficiently supported by current hardware platforms, like GPUs, for LLMs. From the computation perspective, current GPUs only support structured sparsity (*e.g.,* 2:4 sparsity), leading to significant algorithm accuracy loss of LLMs [19]. In contrast, the unstructured sparsity ensuring algorithm accuracy cannot bring end-to-end acceleration for LLMs. For example, the 75% unstructured sparsity only leads to negligible end-to-end speedup [18]. From the memory perspective, quantization and large on-chip memory can reduce data access. Recent algorithm studies [14, 28] are pushing the limit of bit-width with mixed-precision quantization. However, the alignment feature of GPU's cache and SIMD architecture requires homogeneous bit-widths of LLM parameters for weight access reduction [49]. Compounding the issue, GPU's KB-scaled share memory of SMs cannot hold all the activations for LLM text generation.

FPGAs are potential solutions to accelerate LLM inference and explore the benefits brought by model compression, which has been proven in previous deep learning models [21, 22, 39, 43, 55]. However, efficient LLM inference on FPGAs needs to solve the following challenges (Fig. 2):

- **Low computation efficiency.** Flexible sparsity patterns (*e.g.,* block sparsity [53], N:M sparsity [8], etc.) in LLM leads to low computation efficiency.
- **Underutilized memory bandwidth.** The *decode* stage of LLM repetitively accesses fine-grained data from off-chip memory, leading to underutilized bandwidth (29-43%).
- **Large compilation overheads.** The dynamic sparsity patterns and input lengths of LLMs constitute a large design space. For example, generating instructions for 2048 input token length results in ~TB storage overhead on FPGAs.

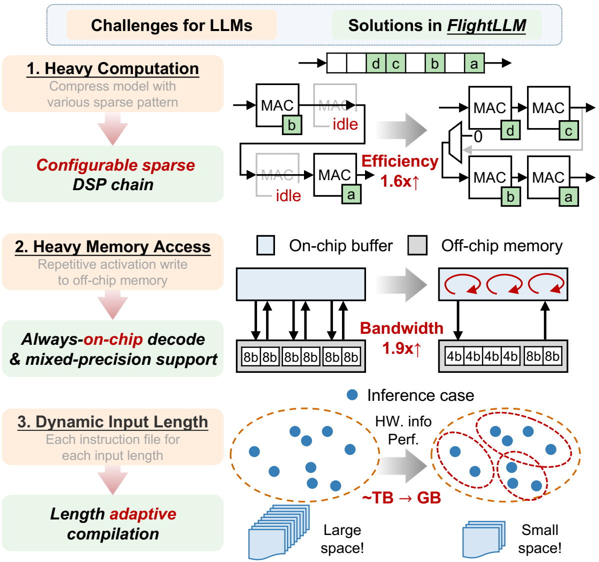


Figure 2: Three challenges of LLM inference on FPGAs, and the corresponding solutions in FlightLLM.

In this paper, we propose **FlightLLM**, enabling efficient LLMs inference with a complete mapping flow on FPGAs (Fig. 1). FlightLLM innovatively points out that the computation and memory overhead of LLMs can be solved by utilizing FPGA-specific resources (*e.g.*, DSP48 and heterogeneous memory hierarchy). To address the challenges of low computation efficiency, FlightLLM exploits a configurable sparse DSP chain. We introduce a flexible cascaded DSP48 architecture to support different sparsity patterns with high computation efficiency (*i.e.*, runtime DSP utilization). To tackle the underutilized memory bandwidth, FlightLLM proposes an always-on-chip decode scheme. Activations reside in the on-chip memory during the *decode* stage with the support of mixed-precision quantization. To reduce the compilation overhead, FlightLLM proposes a length adaptive compilation method. Instructions for consecutive input token length are grouped, and the total storage overhead for instructions can be reduced.

The main contributions of this paper are as follows.

- We propose a configurable sparse DSP chain to support different sparsity patterns. FlightLLM improves the computation efficiency by 1.6× with block-wise and N:M sparsity.
- We propose an always-on-chip decode scheme with mixed-precision support. FlightLLM boosts the memory bandwidth from 35.6% to 65.9%.
- We propose a length adaptive compilation method to reduce the instruction storage overhead by 500× (~GB), enabling deploying real-world LLMs onto FPGAs.

We implement FlightLLM on the Xilinx Alveo U280 FPGA [1]. Evaluated on the OPT-6.7B and LLaMA2-7B, FlightLLM achieves better end-to-end latency than NVIDIA V100S GPU using vLLM [31] and SmoothQuant [49] under the batch size of one. Besides, FlightLLM outperforms NVIDIA V100S and A100 GPU with 6.0× and 4.2× higher energy efficiency, and 1.8× and 1.4× better cost efficiency on average, respectively. When evaluated on the latest Versal VHK158 FPGA, FlightLLM beats NVIDIA A100 with 1.2× higher throughput.

[1] Artifact is available at: https://zenodo.org/doi/10.5281/zenodo.10422477

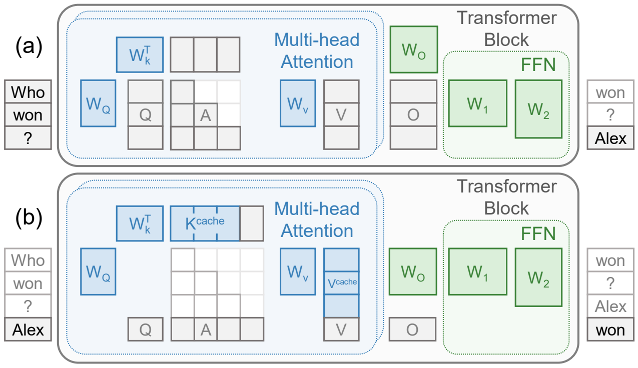


Figure 3: The (a) *prefill* and (b) *decode* stage of LLMs. Colored squares are weights or cached data. Gray denotes activations.

2 BACKGROUND AND RELATED WORK

2.1 Background

Transformer-based LLMs [41, 54] achieve state-of-the-art (SOTA) performance across all kinds of Natural Language Processing (NLP) tasks. The transformer model architecture consists of many cascaded transformer blocks and each transformer block generally includes two types of networks: the Multi-Head Attention (MHA) and the Feed Forward Network (FFN).

Given N input tokens embedded in D dimensional space $X \in \mathbb{R}^{N\times D}$, the MHA projects the token embedding as h heads' query Q, key K and value V matrices in $\mathbb{R}^{h\times N\times D}$ space and performs attention operation for each head as shown in equation 1.

$$Q = XW_Q, K = XW_K, V = XW_V; O = \text{softmax}(QK^T)V \quad (1)$$

where W_Q, W_K, W_V represent the weight matrix in MHA.

The FFN further transforms each token embedding. Given the MHA output matrix $O \in \mathbb{R}^{h\times N\times D}$, FFN passes it through two fully connected layers with a non-linear activation function g and generates the token embedding for the next transformer block.

$$X = g(OW_1)W_2 \quad (2)$$

where, W_1, W_2 represent the two weight matrix in FFN.

As shown in Fig. 3, the workflow of LLMs can be divided into two main stages: the prefill stage and the decode stage. In the prefill stage, LLM takes a prompt from the user which is a sequence of tokens as the input (e.g. the "*Who won ?*" in Figure.3 (a)). Then, LLM will understand the context of the prompt and generates the first response token (e.g. the "*Alex*" in Figure.3 (a)). All the N input tokens are processed simultaneously with high throughput. In the decode stage, LLM treats the newly generated token as length $N = 1$ input and generates the next token (e.g. the "*won*" in Figure.3 (b)). Since LLM only processes one new token at a time in the decode stage, the matrix-matrix multiplications in equation 1 and 2 become matrix-vector multiplications. The decode stage is called iteratively to generate the response token by token.

2.2 Related Work

Efficient Transformer. To tackle the extreme overhead of LLMs, various compression techniques are commonly used. Quantization [15, 20, 33, 49, 52] approaches use low-bit integers to substitute the 16-bit floating point parameters and activations for inference. Sparse attention [4, 8, 9, 45, 53] and weight pruning [19, 30] approaches skip part of the attention matrix or weight matrix computing according to the defined sparse pattern. Various sparse patterns are proposed for different tasks and transformers, including local diagonal pattern [4], block sparse [9, 29, 30, 53], N:M sparse pattern [8, 19], row-column skipping pattern [45], unstructured pattern [19] and so on.

LLM-related Accelerators. Previous work [17, 23–25, 27, 32, 34, 37, 45, 47] propose customized architecture design for transformer models. Some work [17, 24, 32, 34, 44, 45] lay more emphasis on accelerating sparse attention. They design specialized architectures to fully utilize the pre-defined static attention pattern [17, 32] or dynamically generated attention pattern [34, 37, 44, 45]. Recently, FACT [37] points out the importance of compressing linear layers with mixed-precision quantization to help reduce latency. However, these methods cannot accelerate the decode stage of LLMs since they mainly focus on the *prefill* stage for discriminative models, like medium-sized BERT [16] model. DFX [25] emphasizes the acceleration of the decode stage of LLMs. However, it lacks hardware support for model compression of LLMs, making it hard to further expand model size or maximum token size.

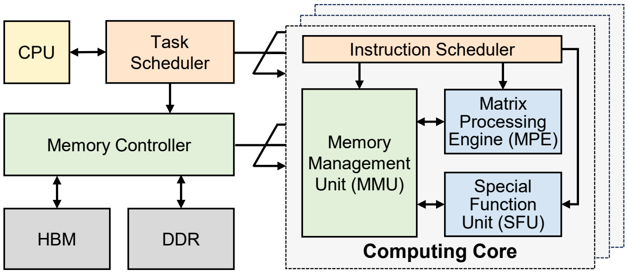

CPU
Task Scheduler
Memory Controller
HBM
DDR
Instruction Scheduler
Memory Management Unit (MMU)
Matrix Processing Engine (MPE)
Special Function Unit (SFU)
Computing Core

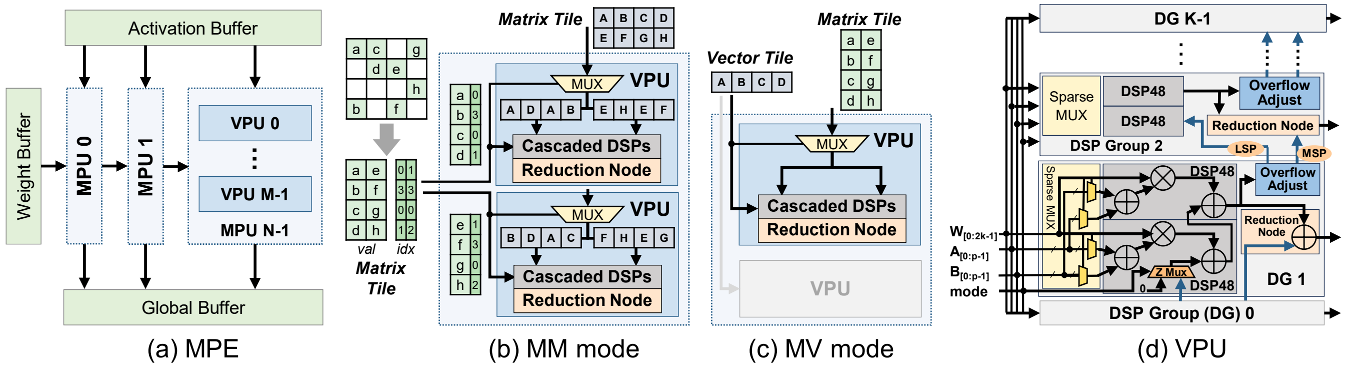
Figure 4: The overall architecture of FlightLLM, including task scheduler, memory controller and computing cores.

3 COMPUTING ARCHITECTURE

3.1 Overall Architecture

We design a high-performance FPGA-based accelerator for generative LLMs by making full use of FPGA resources. Combined with compression techniques like sparsification and quantization, FlightLLM can effectively accelerate the generative LLMs and reduce the inference overhead. As shown in Fig. 4, the overall hardware architecture of FlightLLM mainly includes a task scheduler, memory controller, and multiple computing cores (short as cores). The accelerator uses model parallelism on multiple cores to complete the LLM inference task. The task scheduler assigns tasks to different cores and controls data synchronization.

The components of each core include the unified Matrix Processing Engine (MPE), Memory Management Unit (MMU), Special Function Unit (SFU), and Instruction Scheduler. The instruction scheduler decodes the input instructions and schedules different hardware units to perform computations. The main functions of the remaining hardware units are as follows: **MPE** handles all matrix (*i.e.*, dense and sparse) operations in LLMs. MPE utilizes the configurable sparse DSP chain to reduce the hardware overhead on FPGA. **MMU** reduces memory access overheads by designing customized



(a) MPE (b) MM mode (c) MV mode (d) VPU

Figure 5: The unified Matrix Processing Engine (MPE) can perform multiple types of matrix multiplications. (a) MPE includes multiple Matrix Processing Units (MPUs), which are composed of multiple Vector Processing Units (VPUs). By configuring the MPU, the MPE can support both (b) matrix-matrix multiplication (MM) mode and (c) matrix-vector multiplication (MV) mode. (d) We utilize DSP resources on the FPGA to implement the VPU.

quantization units for low-bit mixed-precision and optimizing data placement for off-chip memory. **SFU** handles miscellaneous operations (*e.g.*, Softmax, etc.) besides matrix processing operations. It also provides an additional data path to share data with other SFUs in different cores, accelerating the MV operation.

3.2 Unified Matrix Processing Engine

Although sparsification can bring huge theoretical benefits to LLM inference, they cannot directly achieve these benefits on existing architectures. To maximize the benefits of sparsification, we design the unified Matrix Processing Engine (MPE) to handle all operations related to matrix computation, including General Matrix Multiplication (GEMM), Sparse Matrix-Matrix multiplication (SpMM), General Matrix-Vector multiplication (GEMV), Sparse Matrix-Vector multiplication (SpMV), and Sampled-Dense-Dense Matrix Multiplication (SDDMM). As shown in Fig. 5(a), the MPE includes multiple Matrix Processing Units (MPUs), which transfer weights from the weight buffer using the streaming approach. The activation buffer and the global buffer store the input and output activations of the MPE, respectively. By configuring the MPU, the MPE can support both matrix-matrix multiplication (MM) (Fig. 5(b)) and matrix-vector multiplication (MV) mode (Fig. 5(c)). The MPU is composed of multiple vector processing units (VPUs). The VPU is the basic component in the MPE, which performs the dot product of two vectors.

We build the unified MPE to support all the five operator on the same hardware achitecture. FlightLLM overcomes the challenge of low computational efficiency through hardware/software co-design. To do this, we first introduce the MPU, which exploits configurable sparse DSP chain to reduce hardware overhead while supporting sparse reduction. We use the MM mode as an example to illustrate the main idea and the implementation of MPU. Then, we re-design MPE's parallel scheme to maximize the performance in the MV mode. Finally, we introduce the SDDMM support through simple instruction scheduling.

3.2.1 MPU Design. In transformer-based LLMs, sparsification methods including sparse attention and weight pruning are widely used to accelerate the LLM inference. The sparse pruning generates sparse matrix, whose densities and sparse patterns are uncertain. It brings great challenges to hardware design, especially for FPGA-based architectures that use the fixed DSP48 as the multiplication unit. Existing work introduces large additional hardware architectures to support sparse computations, which leads to a significant increase in hardware resources (about 5× [38]). Without proper architectural design, the benefit of sparsification is weakened.

We utilize the DSP48 engine on FPGA to support sparse operations. In order to reduce the hardware overhead, previous work cascades the DSPs to take full advantage of the hardware resources in DSP48. DSP cascading makes the most use of the accumulator, the result carry-out port, and the result carry-in port, improving the hard-core utilization. However, the fully cascaded DSP architecture are not friendly to sparse computation since the cascaded chain is a fixed path. In FlightLLM, we propose a **configurable sparse DSP chain (CSD-Chain)** to supplement the fixed DSP chain. In the CSD-Chain, a long DSP chain is divided into several DSP groups. A DSP group (DG) has several DSP48 cores, that are cascaded in a fixed manner. We pack two INT8 MACs on DSP48 [1]. Different DGs are cascaded with a configurable path. A VPU is made up of a CSD-Chain and a MPU consists of several VPUs. Fig. 5(d) shows the architecture of the CSD-Chain based VPU. Each DG has two DSP48 cores. We use configurable cascading to support sparse matrix operations by adding three units to the fixed DSP chain.

Sparse Mux. As shown in Fig. 5(d), two activations (A and B) are delivered to one DSP48 core simultaneously for weight reuse. Before the delivery, they are sent to a sparse MUX unit. In the sparse MUX unit, each activation is selected from multiple inputs according to the sparse index (shown in Fig. 5(b)). With this sparse-based multiplexer, only nonzero inputs are sent to the DSP48 core.

Reduction Node (RN). Compared to GEMM operations, calculating SpMM may produce more outputs, as demonstrated in Fig. 6(b). Thus, DSPs are grouped in our design to implement non-breaking MAC and a reduction node are inserted at the end of a DG. When a SpMM operation wants to generate multiple outputs, the RN will break the configurable cascade path between DGs, and calculate the output. Other DGs on the CSD-Chain will start a new SpMM computation by selecting zero in the Z-MUX.

Overflow Adjust Unit (OAU). When a DSP48 shares a weight with two activations, only 18 bits can be accessed for each activation.

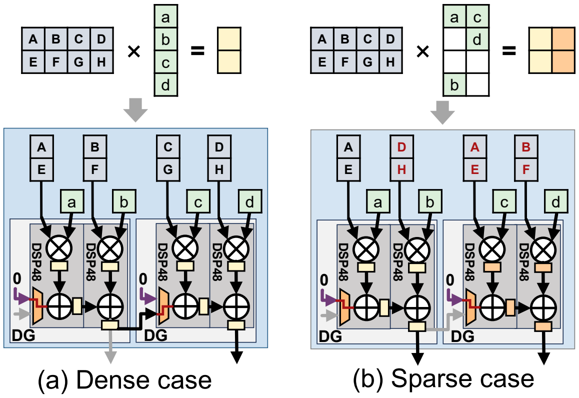


Figure 6: By configuring the VPU, the MPE can support both (a) dense and (b) sparse cases.

As a result, a long cascade accumulation may overflow. Therefore, we adjust the output data before sending it to the next DG. In the overflow adjust unit, the result is split into a most significant part (MSP) and a least significant part (LSP). The LSP cascades to the next DSP48 with limited bits to avoid accumulation overflow. The MSP are delivered to the RN in the next DG to calibrate the output result. In this way, all accumulators in DSP48 are fully utilized. Since a 18-bit integer will never overflow if no more than eight 16-bit integer are accumulated, the OAU is skipped with no more than eight DSP48 cores.

Due to the configurable cascade path, VPU with the CSD-Chain can efficiently work on both dense and sparse multiplications. As shown in Fig. 6, all DSP48 cores are fully used in both cases. The only difference is that the RN in sparse case will break the CSD-Chain into two individual DSP chains to execute two different MACs and produce two outputs.

Supporting sparse matrix multiplication could improve the computation efficiency. But arbitrary sparsity may cause data mismatch between different DGs, leading to unexpected efficiency decrease. Existing work shows that N:M sparse pattern is a promising sparsification method. It maintains the same sparsity ratio within each matrix block, and allocates different sparsity ratios among different matrix blocks. Where M is an integer power of 2, and N is the partial factor of M. For example, M=16, N=0, 2, 4, 8, 16. The N:M sparse method restricts the number and position of nonzero elements while maintaining flexible sparsity. It can be easily mapped to a CSD-Chain. For a N:M sparse architecture, a CSD-Chain can be splited into N groups. Each DSP will select one input from M inputs. In each cycle, the entire CSD-Chain can produce one MAC output in dense case and N MAC outputs in N:M sparse case. Fig. 6 shows the case of a VPU supporting 2:4 sparse pattern.

3.2.2 Matrix-Vector Multiplication Analysis. We explore the hyperparameter space of compute tiling to fully utilize the off-chip memory bandwidth. We model the memory access time T_{mem} and computing time T_{cmp} of general MM in equation 3. $M \times K$, $K \times N$, and $M \times N$ denote the shapes of two input matrices and one output matrix, respectively. p_M, p_K, and p_N denote the three dimensions of computational parallelism in matrix computation. BW denotes the off-chip bandwidth.

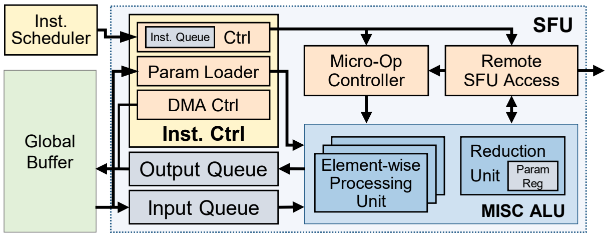


Figure 7: SFU contains a MISC ALU, an instruction controller, a micro-operator controller, and a remote SFU access engine.

$$
\begin{aligned}
T_{mem} &= \frac{M \cdot K + K \cdot N + M \cdot N}{BW} \\
T_{cmp} &= \frac{M \cdot K \cdot N}{p_M \cdot p_K \cdot p_N}
\end{aligned}
\tag{3}
$$

To overlap the computation and memory access with double-buffer, we need to make sure that $T_{mem} < T_{cmp}$. For MV operations, $M = 1$ and $p_M = 1$ are set. For the MV mode, we can iterate through the space to obtain a set of $[p'_K, p'_N]$ to guarantee the bound for double-buffer. In other words, under the configuration of $[p'_K, p'_N]$, we can realize that the MPE can still fully utilize the off-chip memory bandwidth in MV mode, although the computing resources in the MPE are partially idle at this time (Fig. 5(c)). By redesigning the computational parallelism, MPUs can maximize the execution performance of executing GEMV and SpMV on FPGAs.

3.2.3 SDDMM Support. SDDMM is the key operator of the sparse self-attention layer. The block-wise sparsity of SDDMM can be used to reduce the amount of computation and improve the hardware energy efficiency. Therefore, we can treat SDDMM as multiple GEMMs in a block-wise manner. We only need to do some processing on the SDDMM operator with the instruction scheduler to efficiently complete the SDDMM computation on the MPE.

3.3 Special Function Unit

Besides MM and MV computations, there are many other operations in LLMs, including softmax, layer normalization, etc. These miscellaneous (MISC) operations can be classified into two types: (a) Element-wise operation, which generates the result element by element (such as element-add and concat); (b) Two-phase operation, which will perform a reduction operation to get one or more parameters before the element-wise operation (such as softmax and layer normalization). Unlike MM and MV operations, these operations are not compute-intensive. Thus, we design a Special Function Unit (SFU) to handle all MISC operators, as shown in Fig. 7. The SFU splits a MISC instruction into micro-operations, and delivers each micro-op to the ALU. The ALU will calculate the output according to the micro-op. All the input data is fetched from the MMU. For two-phase operations, the SFU will read an entire vector data from MMU to generate necessary parameters and read the same data again calculating the final output. For accuracy consideration, softmax and layer normalization operations are calculated in fp16 since the hardware cost of SFU is acceptable.

Hiding the computation latency of MISC operations is important to improve the end-to-end latency in LLMs. For MM and multi-head MV operations, MISC calculations can be hidden between different vectors. For MISC operations after single-head MV calculations, the SFU breaks the entire vector into several sub-vectors and performs

MISC operations in fine granularity to hide the computation latency. In consideration of the scalability, multiple SFUs in different PEs may work together by accessing remote SFUs. A SFU can share parameters and calculation results with other SFUs. Thus, although a vector may be generated by different PEs simultaneously, the result could be sent to all other PEs without writing back to HBM. It reduces the end-to-end latency and the wire overhead on FPGA.

4 ALWAYS-ON-CHIP DECODE

4.1 On-chip Decode Dataflow

In the decode stage, the main efficiency constraint arises from the frequent access of off-chip memory for small data-volume activation vectors. To reduce off-chip memory access of activation vectors, we employ the concept of operator fusion and fuse the computation within each inference of decode stage. Consequently, we can significantly increase the off-chip HBM (High Bandwidth Memory) bandwidth utilization from about 35.6% to 65.9%.

Since the activations in the decode stage are small data-volume vectors instead of matrices, they can be fully accommodated by the on-chip buffer of the FPGA. Therefore, to reduce the frequent read and write operations of the activation vectors, we fuse the computations of all layers during each inference of the decode stage. Finally, the computation result is written back to the off-chip memory at the end of each inference of the decode stage.

As depicted in Fig. 8(b), given that we can directly use the output activation of the current layer as the input activation for the subsequent layer without writing the activation to off-chip memory, we retain the output activations from linear or attention operations within the on-chip buffer. Through appropriate scheduling, the activation vector can consistently be stored in the on-chip buffer and then processed by either the MPE or SFU.

Furthermore, as illustrated in Fig. 8(a), since there is no hardware resource conflict between SFU and MPE computations, we fuse the computations of these two different units to reduce the off-chip memory access of the intermediate results. Specifically, for the Softmax and LayerNorm operations, since they require a complete activation vector, it requires the `MV` operator in MPE to compute the activation vector before the `MISC` operator in SFU. For the activation functions (*e.g.*, SiLU) and Element-wise addition/multiplication (Eltwise), they are `MISC` operators in SFU that can be computed immediately after the `MV` operators in MPE.

4.2 Discussion: Fusion in the Prefill

The MISC fusion for the prefill stage is similar to that of the decode stage. However, the decode stage uses `MV` (see Table 1 in Sec. 5.1), while the prefill stage uses `MM` for matrix multiplication. Thus, in the prefill stage, Softmax and LayerNorm can start after the `MM` has processed an activation vector, while Eltwise and SiLU can start the computation after each `MM`.

In prefill attention computation, sparse attention can be applied. The sparse attention computation can be divided into 3 steps, computing QK^T, Softmax, and SV, as shown in Fig. 8(c). When computing QK^T, the computation result is sparse according to the attention mask. When the zero attention mask completely covers the computation results, the corresponding `LD` and `MM` can be skipped. If the zero attention mask covers a part of the computation results, the `MM` instruction writes only the required part of the computation results back into the global buffer and then performs Softmax. When computing SV, the S matrix is sparse, where the matrix S stands for the attention matrix after the Softmax operation. Thus, in the proposed fused attention dataflow, Softmax and SV are only computed for the parts that are not covered by the zero attention mask. The idea of acceleration by fusion is similar to the decode stage.

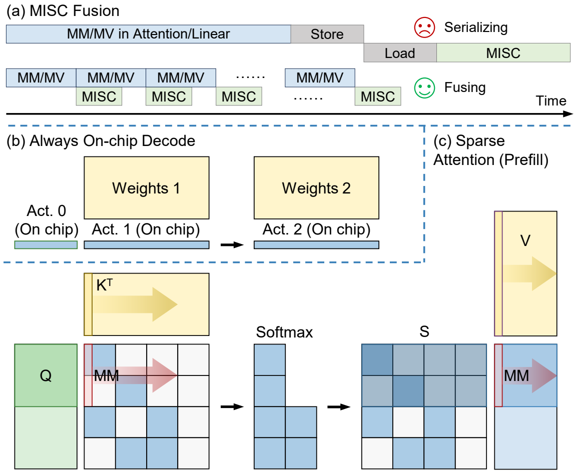


Figure 8: (a) MISC fusion with attention or linear operation. And an example of always on-chip decode approach in the (b) decode and (c) prefill stage.

4.3 Mixed-precision Support

The low-bit mixed-precision strategy can significantly reduce the parameters and the off-chip memory access for LLMs. However, GPU with the low-bit mixed-precision strategy (2/3/4/8-bit) makes it difficult to reduce memory access overhead. GPU uses SIMT-based computing architecture, which cannot efficiently process irregular and different bit-width data in memory. Therefore, we design a dedicated mixed-precision dequantization unit on the FPGA, which can efficiently process the compactly stored mixed-precision data in the buffer and convert it into a unified INT8 format to the MPE. Specifically, we transform mixed-precision multiplication (2/3/4-bit) into INT8 multiplication, to avoid excessive LUT overhead. The dequantization unit consists of a set of parallel bit-width expansion units, which automatically expand the input data to 8 bits according to the control signal, scale factor, and sign bit.

4.4 Memory Latency Optimization

We point out that the unique HBM+DDR hybrid memory system on FPGA has advantages over both HBM-only and DDR-only in reducing memory access overhead for generative LLMs. As we discussed in the previous section, the memory access patterns of SFU and MPE are significantly different. MPE needs to handle large-scale matrix multiplications, so single memory access is very large (~M Bytes). Using HBM can take full advantage of its high bandwidth. However, the operations processed by SFU include Eltwise, Softmax, etc., which are characterized by small single memory access data (~100 Bytes). Because the memory access latency of HBM is higher than DDR, and the refresh cycle is less than DDR [46, 56]. As a result, the overhead of HBM exceeds that of DDR when accessing a small amount of data. Therefore, we utilize the unique

Table 1: ISA design of FlightLLM.

Inst.	Discriptions
LD	Load data from HBM or DDR to on-chip buffer.
ST	Store data from on-chip buffer to HBM or DDR.
MM	Calculate matrix-matrix multiplication $\mathbf{C} = \mathbf{X}\mathbf{W}^T + \mathbf{b}$.
MV	Calculate matrix-vector multiplication $\mathbf{c} = \mathbf{x}\mathbf{W}^T + \mathbf{b}$.
MISC	Calculate LayerNorm, SiLU, Softmax and Eltwise.
SYS	Synchronize between multiple SLRs or with host CPU.

HBM+DDR hybrid memory system on the U280 FPGA to optimize the inference of the generative LLM. Specifically, we store small single-access data (*e.g.*, Softmax, Silu, and Gelu lookup tables) on DDR to take advantage of the low memory access latency of DDR. We store large single-access data (*e.g.*, KV cache, weights) on HBM to take advantage of the high memory bandwidth of HBM.

5 SOFTWARE DESIGN

5.1 Instruction Set Architecture

Instruction Set Architecture (ISA) acts as a connection between the LLM and hardware accelerator, consisting of six instructions listed in Table 1. LD and ST stand for transmission between off-chip (HBM or DDR) and on-chip memory. MM and MV represent for matrix-matrix and matrix-vector multiplication respectively. MISC controls other computations, including layer normalization (LayerNorm), SiLU, Softmax, and Eltwise. SYS is responsible for synchronization between multiple Super Logic Regions (SLRs) after each layer or with the host CPU after each inference is completed.

5.2 Length Adaptive Compilation

5.2.1 Challenges. The instructions of FPGA accelerators are usually generated using static compilation, leading to a large instruction volume for different input shapes. Due to the fact that generative LLMs generate one token at a time, the token length increases by 1 with each inference. This means that each inference process of the LLM requires different instructions. However, due to the large computational and storage requirements of the LLM, even with coarse-grained instructions, the number of instructions is still enormous. Taking the example of deploying the LLaMA2-7B model on the U280 FPGA, the average volume of instructions required for the decode stage of each inference on each SLR is approximately 2.9 MB. For the prefill stage, the average volume of instructions required for each inference on each SLR is about 282.1 MB. Suppose we need to store instructions for all 3 SLRs, covering all token scenarios for prefill and decode 1-2048, in order to handle random input and output token quantities. In this case, the instructions would require approximately 1.67 TB. This size already far exceeds the capacity of U280 DDR. Unfortunately, due to sparse attention and N:M sparse pattern, each layer and each head in the LLM has a different sparse pattern, resulting in different instructions. Thus, it is not possible to reuse one set of instructions for multiple layers and attention heads. Therefore, we urgently need a method to reduce the size of the instruction sequence, allowing for the realization of inputs and outputs with arbitrary token lengths within limited storage.

5.2.2 Solutions. The fundamental reason for the large size of the instruction file is to adapt arbitrary lengths of prefill and decode stages, which requires storing instructions for all possible scenarios in memory. To address this issue, we can reuse the same instructions by allowing different lengths of prefill or decode. Specifically, by setting a threshold range, token lengths within this range can share the same instructions. For example, when the input token length is between 1 and 16, we can reuse the instructions for 16 tokens. Additionally, considering our N:M sparse block size (16×16) and the size of the sparse attention block (64×64), reusing instructions in this manner would not have a significant impact on performance.

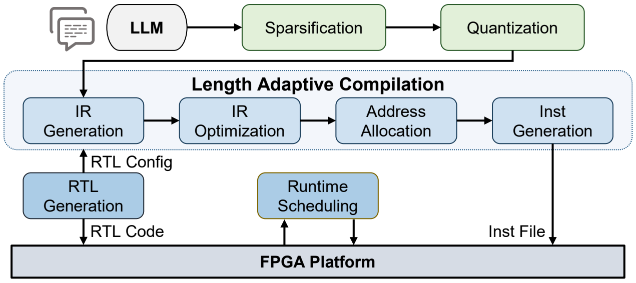


Figure 9: The overall mapping flow of FlightLLM.

We find that instructions are executed more frequently in the decode stage than in the prefill stage. The bottleneck of the decode stage lies in memory access, which is directly proportional to the token length. Therefore, we use more refined thresholds in the decode stage to avoid too much redundant computation. Moreover, we can reuse the same instruction file by configuring different base memory addresses of PEs of different SLRs through registers. The instruction size can be reduced to 4.77 GB with these optimizations, which the DDR of U280 can already store.

We optimize the instructions for multiple HBM channels memory access to reduce the instruction size further. For example, in each PE, the A buffer and the global buffer are connected to 8 HBM channels, and each HBM channel requires an LD or ST instruction each time the data is moved between the buffer and the HBM. We combine these similar instructions into one instruction, and the hardware decoder decodes the single instruction into eight hardware instructions. The eight hardware instructions will be launched to eight HBM channels simultaneously, enabling the concurrent read/write of multiple HBM channels to utilize the HBM bandwidth fully. Through these optimizations, the instruction size is reduced to 3.25 GB and stored in the DDR memory with negligible performance loss.

5.3 Analytical Model for RTL Generation

In this section, we analyze the relationship between the theoretical hardware resource utilization of FlightLLM on FPGA platforms and hardware implementation. For the main computation resource on FPGA, the usage of DSP is determined by the computing parallelism ($p_M * p_K * p_N$) of MPU and its amount. The theoretical usage of DSP can be estimated as follows: $DSP = (p_M * p_K * p_N * MPU) * MPE$. As for the on-chip buffers, their data width is designed based on the parallelism of the computation units. We implement activation buffer with URAM for its larger capacity, while the remaining buffers are implemented with BRAM36. The theoretical buffer usage are as follows: $URAM = (p_M * p_K * Activation\_width/URAM\_width) * MPU * MPE$, $BRAM = (Weight\_buffer + Global\_buffer + Index\_buffer) * MPE$. For memory bandwidth, the theoretical peak

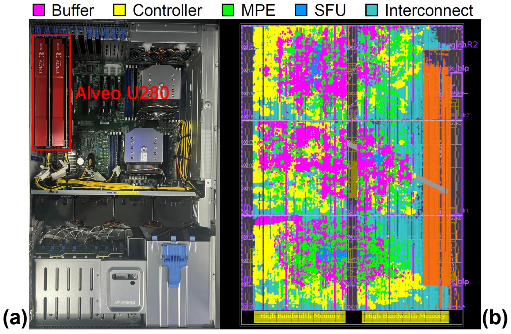


Figure 10: (a) U280 FPGAs on the server (one card is used only). (b) FlightLLM implementation layout on U280 FPGA.

bandwidth is calculated as $Bandwidth = (MPU/8 + 2) * MPE * 14.4GB/s$. The RTL generator determines the deployment of hardware modules of FlightLLM on a specific FPGA platform based on these theoretical models. It aims to fully utilize on-chip resources to improve the performance of the accelerator.

5.4 Mapping Flow

Fig. 9 shows our entire deployment flow. FlightLLM takes the PyTorch-based LLM as input and converts it to ISA according to the customized intermediate representation (IR). First, the original LLM undergoes sparsification and quantization to create a compressed LLM. Subsequently, the IR is exported, encompassing the model's structure, weights, sparse indexes, and attention masks, which is achieved through automated parsing of the model's structure. Following that, the generated IR undergoes optimization, which involves operations like removing the `view()` layers that do not impact the data arrangement and performing layer fusion. More specifically, the attention layer will be fused with the softmax layer, and the linear layer will be fused with ReLU, SiLU, and element-wise layers. Subsequently, all the data in the optimized IR will be assigned HBM or DDR storage addresses. Additionally, the data stored in the HBM will be partitioned into appropriate channels to prevent inefficient access across different channels, thus leveraging the FPGA's high bandwidth effectively. Lastly, the compiler will generate instructions using the optimized IR and schedule the on-chip buffer according to manually defined templates.

In addition, we support generating corresponding RTL for different FPGA platforms. Specifically, the RTL Generator takes parameters of different FPGA platforms (including the amount of DSP, the capacity and bandwidth of HBM/DDR and on-chip RAM resources) to dynamically adjust the computing parallelism and buffer size. This is done to generate corresponding RTL code for implementation and configurations for compilation, in order to maximize the optimal performance on different platforms.

6 EVALUATION

6.1 Evaluation Setup

Models and Datasets. We evaluate the effectiveness of FlightLLM with state-of-the-art large language models LLaMA2-7B [41] and OPT-6.7B [54]. We finetune the compressed model with a small sampled subset of RedPajama dataset [11] consisting 8192 rows with 56M tokens. The accuracy evaluation is performed on the commonly used WikiText-103 and WikiText-2 [35] datasets.

Table 2: Hardware parameters of GPU and FPGA platforms.

	GPU	GPU	FPGA	FPGA
Platform	NVIDIA V100S(12nm)	NVIDIA A100(7nm)	Xilinx Alveo U280(16nm)	Xilinx Versal VHK158(7nm)
Frequency	1245 MHz	1065 MHz	225 MHz	225 MHz
Computing Units	640 Tensor Cores	432 Tensor Cores	9024 DSPs	7392 DSPs
Memory	32 GB	80 GB	8 & 32 GB	32 & 32 GB
Bandwidth	1134 GB/s	1935 GB/s	460 & 38 GB/s	819 & 51 GB/s

Table 3: Hardware utilization of FlightLLM on Alveo U280.

Component	LUT	FF	BRAM	URAM	DSP
Buffer	42k(3.2%)	75k(2.9%)	816(40.5%)	792(82.5%)	0
Controller	162k(12.4%)	156k(6.0%)	408(20.2%)	0	0
MPE	190k(14.6%)	360k(13.8%)	0	0	6144(68.1%)
SFU	30k(2.3%)	36k(1.4%)	24(1.2%)	0	201(2.1%)
Interconnect	150k(11.5%)	316k(12.1%)	4(0.2%)	0	0
Total	574k(44.0%)	943k(36.2%)	1252(62.1%)	792(82.5%)	6345(70.2%)

Metrics. We leverage latency and throughput to evaluate the difference between FlightLLM and other baselines comprehensively. Latency is used to measure the end-to-end time cost of the entire inference. Throughput is used to measure the speed of the decode stage by dividing the number of output tokens by the time of the decode stage. Unless otherwise noted, all results are evaluated under the batch size of 1 to accommodate latency-sensitive scenarios.

FPGA Platforms. We use two FPGA platforms for evaluation, including Xilinx Alveo U280 [50] and Versal VHK158 [51]. Table 2 lists the hardware parameters of FPGA platforms. Alveo U280 FPGA is equipped with two kinds of memory: 8GB HBM with 460GB/s bandwidth and 32GB DDR with 38GB/s bandwidth. Versal VHK158 FPGA's HBM capacity and bandwidth are significantly improved compared to U280, reaching 32GB and 819GB/s. We implement FlightLLM on the real system with U280 FPGAs (Fig. 10(a)). For VHK158 evaluation, we implement a cycle-accurate simulator, which has been verified with RTL emulation using Vitis 2023.1.

FPGA Implementation. Fig. 10 shows the layout of our implementation on U280 FPGA. Since cross-die connections are more likely to become the critical paths for timing closure, we instantiate multiple computing cores and place them on different SLRs. For the memory controller, we place it on SLR0 closest to the HBM for easy reading. The implementation report shows that our kernel runs at 225 MHz, while detailed hardware utilization is listed in Table 3. We also measure the power of FlightLLM through the vendor-provided Xilinx Board Utility tool xbutil [3].

GPU Baselines. We choose NVIDIA A100 and V100S as our GPU baselines, and their specifications are also listed in Table 2. We conduct evaluations on the selected model with huggingface PyTorch as the GPU-*naive* design, vLLM [31], and SmoothQuant [49] as the GPU-*opt* design. vLLM is the commonly used LLM framework with KV cache memory optimization, and SmoothQuant is the SOTA LLM quantization framework with INT8 CUDA kernel for both activations and weights. We use nvprof [2] to profile the GPU power consumption at runtime.

SOTA Accelerator Baselines. We also selected three domain-specific accelerators targeting at accelerating attention mechanism:

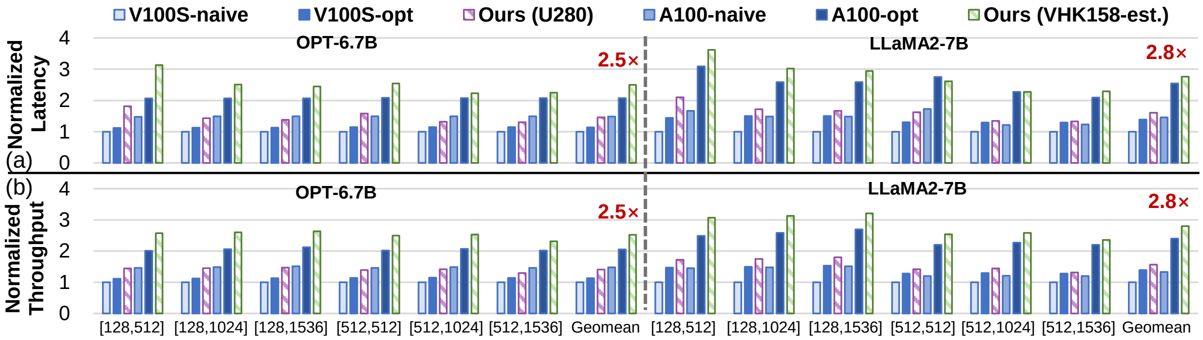


Figure 11: Latency and throughput of FlightLLM and V100S/A100 GPU. The horizontal axis represents [prefill size, decode size].

Table 4: Perplexity of LLMs under different optimization configuration on wikitext-103 and wikitext-2 datasets.

LLM	Compression	wikitext-103	wikitext-2
LLaMA2-7B	None	8.7	21.2
	Sparse Attention	8.1	19.0
	Weight Pruning	8.3	17.8
	Quantization	9.9	20.6
	All	10.2	21.9
OPT-6.7B	None	11.0	10.0
	Sparse Attention	11.1	10.5
	Weight Pruning	11.8	11.1
	Quantization	10.8	10.3
	All	13.0	12.5

DFX [25], FACT [37], and CTA [44]. It has to be mentioned that DFX is a multi-FPGA acceleration work, and we only evaluate its hardware performance of a single card. Since there are no open-source codes for these accelerators, and they have not supported recent LLMs such as LLaMA2. We build C++ simulators based on corresponding hardware designs to evaluate their performance, achieving less than 5% deviation using their original data. For fairness, we align the hardware parameters (clock frequency, peak performance, bandwidth) for these baselines.

6.2 Evaluation Results

6.2.1 Accuracy of Compressed LLMs. FlightLLM harnesses the power of state-of-the-art model compression methods by optimizing them to fit the distinct features of FPGA. As depicted in Table 4, we conduct experiments around different optimization configurations on LLMs. For sparsification, FlightLLM builds upon previous work to use block sparse for sparse attention [53] and N:M sparse for weight pruning [57]. FlightLLM uses gradient-based analysis [15] to quantify the importance of each weight and attention value and remove the unimportant values. For quantization, FlightLLM builds upon previous work [49] and extends its single-precision scheme to mix-precision. FlightLLM follows the same idea as sparsification and use the gradiant-based analysis to quantify weight importance and assign three, four or five bit width accordingly. With this scheme, FlightLLM achieves average 3.5-bit for weights and 8-bit for activations. Note the compression methods are also compatible with GPU, but they need the customized units of FPGA to yield real wall-clock speedup. By using the block sparse attention, N:M weight pruning and mixed-precision quantization all together, FlightLLM successfully compresses the original LLM with minimum perplexity influence.

Table 5: The bandwidth utilization of different platform.

Platform	V100S GPU		A100 GPU		U280	VHK158
Solution	None	Opt.	None	Opt.	Ours	Ours
BW Util.	42.5%	65.5%	28.6%	57.4%	**65.9%**	**64.8%**

6.2.2 Comparison with GPUs. We compare the end-to-end latency of GPUs and FlightLLM. Fig. 11 shows that FlightLLM on VHK158 outperforms V100S and A100 GPU in latency on both models with different combinations of input token size and output token size. For OPT-6.7B/LLaMA2-7B model, FlightLLM on U280 improves the end-to-end latency by 1.5/1.6× and 1.3/1.2× compared to V100S-*naive* and V100S-*opt*, respectively. Table 5 shows the bandwidth utilization of FlightLLM on FPGAs is better than A100 GPUs. This is because FlightLLM can be customized to design hardware units, which can fully exploit the sparsity of LLM and the memory access optimization in the decode stage.

6.2.3 Comparison with SOTA accelerators. Fig. 12(a) shows the latency of different architectures running on OPT-6.7B and LLaMA2-7B model. It can be seen that FlightLLM achieves a general speed-up in end-to-end latency compared to DFX, CTA and FACT. The geomean latency speedups of FlightLLM on U280 and on VHK158 are 2.7× and 4.6× compared to DFX for OPT-6.7B, respectively. And the geomean throughput speedups of FlightLLM on U280 and on VHK158 are 2.6× and 4.6× compared to DFX. Compared to DFX, the acceleration effects of sparse attention in CTA and FACT are not significant, mainly because the attention computation does not account for a high proportion under small prefill size. However, our work adopts lower bit-width quantization scheme, which effectively alleviates the memory bottleneck in the decode stage. Fig. 12(b) shows the geomean throughput of different architectures, with FlightLLM achieving the highest performance. Under the same hardware parameters, FlightLLM achieves better utilization of computing resources as well as bandwidth.

6.2.4 Energy and Cost Efficiency. We consider energy efficiency as fair metrics to compare GPU and our FPGA-based FlightLLM. Fig. 13 shows the results of energy efficiency (Token/J). FlightLLM on U280 consistently outperforms GPUs and achieves 6.7×, 4.6×, 6.0× and 4.2× energy efficiency compared to V100S-*naive*, A100-*naive*, V100S-*opt* and A100-*opt* respectively for OPT-6.7B. For LLaMA2-7B, FlightLLM on U280 achieves achieves 6.0×, 4.4×, 5.5× and 3.8× energy efficiency. For cost efficiency (Token/s/dollor), GPU generally has higher product price compared to FPGA. The price of V100S, A100 and Aleveo U280 are approximately 12000$, 17000$ and 8000$ respectively. Therefore, FlightLLM on U280 achieves

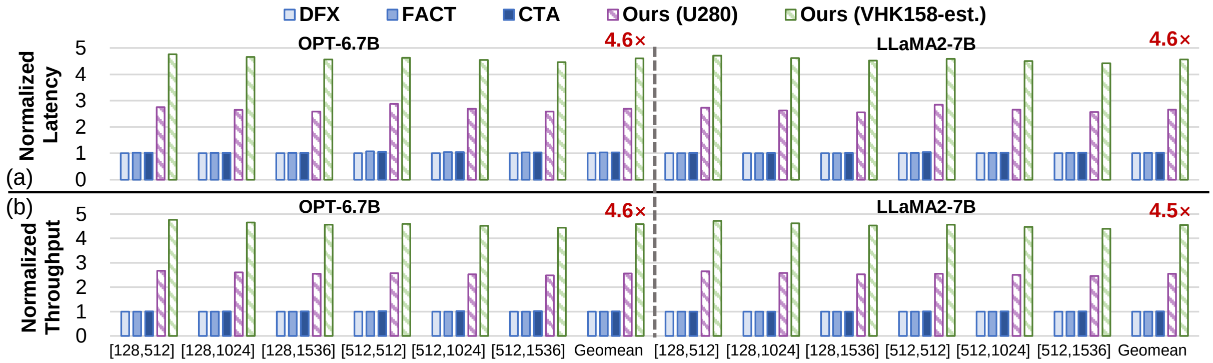


Figure 12: Performance of FlightLLM, DFX, CTA, and FACT. The horizontal axis represents [prefill size, decode size].

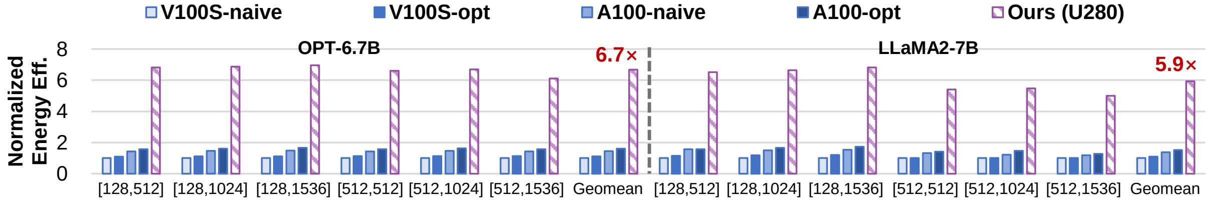


Figure 13: Energy efficiency of FlightLLM, NVIDIA V100S/A100 GPU. The horizontal axis represents [prefill size, decode size].

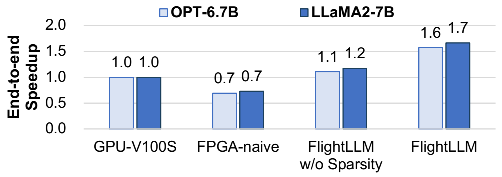


Figure 14: The latency breakdown of FlightLLM.

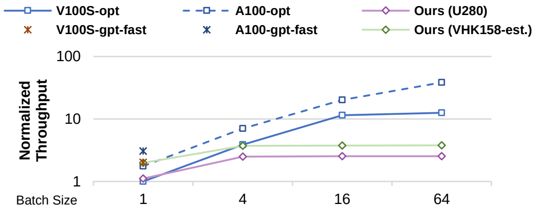


Figure 15: The multi-batch performance on LLaMA2-7B.

1.9× and 1.5× higher geomean cost efficiency over V100S-*opt* and A100-*opt* for OPT-6.7B, and achieves 2.3× and 1.4× higher geomean cost efficiency over V100S-*opt* and A100-*opt* for LLaMA2-7B.

6.2.5 *Performance Breakdown.*

Fig. 14 shows the latency breakdown of FlightLLM. We normalize the latency of the LLaMA-2 and OPT model running on a V100S GPU. We naively implement the LLaMA-2 model on the U280 FPGA, which has only 70% of the performance of the V100S GPU. The gap is due to the larger peak memory bandwidth (1134GB/s vs. 460GB/s) and higher peak performance (130TOPS vs. 25TOPS) of the V100S GPU [10] compared to the U280 FPGA. After using the flexible sparse method and the configurable sparse DSP chain, the performance of FlightLLM is improved by 1.1-1.2×, because we reduce the inference computation and make full use of DSP resources. After further using the always on-chip decoder, the performance improvement of FlightLLM achieves 1.6-1.7×, because we effectively reduce the overhead of off-chip memory access.

6.2.6 *Discussion.*

gpt-fast[2] is a new SOTA Pytorch-native codebase optimized for LLM inference, achieving 196.8 tokens/s with INT4 quantization on the NVIDIA A100 GPU. However, the current version of gpt-fast has no support for OPT models and multi-batch processing. Evaluated on LLaMA2-7B, FlightLLM on the VHK158 FPGA achieves 92.5 tokens/s, and provides 2.9× better energy efficiency and higher bandwidth utilization (64.8% vs. 44.6%) than gpt-fast on the A100 GPU.

As for the multi-batch performance, gpt-fast has no support of multi-batch processing, so only the results of GPU-*opt* (*i.e.*, vllm and SmoothQuant) are reported. In Fig. 15, as the batch size increases, the performance advantage of FlightLLM over GPU will gradually decrease. The main reason is that GPUs have more hardware resources (*i.e.*, memory bandwidth and computing units with higher frequency) than FPGAs.

7 CONCLUSION

This paper proposes FlightLLM, enabling efficient LLMs inference with a complete mapping flow on FPGAs. In FlightLLM, we innovatively point out that the computation and memory overhead of LLMs can be solved by utilizing FPGA-specific resources. FlightLLM demonstrates that FPGAs are promising candidates for efficient LLM inference. FlightLLM achieves 6.0× higher energy efficiency and 1.8× better cost efficiency against commercial GPUs (*e.g.*, NVIDIA V100S) on modern LLMs (*e.g.*, LLaMA2).

ACKNOWLEDGMENTS

This work was supported by National Natural Science Foundation of China (No. 62325405, 62104128, U19B2019, U21B2031, 61832007, 62204164), and Beijing National Research Center for Information Science and Technology (BNRist). We thank for the insightful discussion with Shengen Yan and colleagues at Infinigence-AI. This work is supported by dgSPARSE project[3].

[2] https://github.com/pytorch-labs/gpt-fast, released in 2023.11.30.
[3] The open source dgSPARSE project: https://dgsparse.github.io/

REFERENCES

[1] 2017. Deep Learning with INT8 Optimization on Xilinx Devices. [Online]. https://docs.xilinx.com/v/u/en-US/wp486-deep-learning-int8.

[2] 2022. nvprof. [Online]. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[3] 2022. Xilinx Board Utility Tool. [Online]. https://xilinx.github.io/XRT/2021.1/html/xbutil2.html.

[4] Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* (2020).

[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. *arXiv preprint arXiv:2108.07258* (2021).

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[7] Siyuan Chen, Mengyue Wu, Kenny Q Zhu, Kunyao Lan, Zhiling Zhang, and Lyuchun Cui. 2023. LLM-empowered Chatbots for Psychiatrist and Patient Simulation: Application and Evaluation. *arXiv preprint arXiv:2305.13614* (2023).

[8] Zhaodong Chen, Zheng Qu, Yuying Quan, Liu Liu, Yufei Ding, and Yuan Xie. 2023. Dynamic n: M fine-grained structured sparse attention mechanism. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 369–379.

[9] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509* (2019).

[10] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro* 41, 2 (2021), 29–35.

[11] Together Computer. 2023. *RedPajama: An Open Source Recipe to Reproduce LLaMA training dataset*. https://github.com/togethercomputer/RedPajama-Data

[12] Jiaxi Cui, Zongjian Li, Yang Yan, Bohua Chen, and Li Yuan. 2023. ChatLaw: Open-Source Legal Large Language Model with Integrated External Knowledge Bases. *arXiv:2306.16092* (2023).

[13] Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. 2020. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proc. IEEE* 108, 4 (2020), 485–532.

[14] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339* (2022).

[15] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339* (2022).

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[17] Hongxiang Fan, Thomas Chau, Stylianos I. Venieris, Royson Lee, Alexandros Kouris, Wayne Luk, Nicholas D. Lane, and Mohamed S. Abdelfattah. 2022. Adaptable Butterfly Accelerator for Attention-based NNs via Hardware and Algorithm Co-design. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 599–615. https://doi.org/10.1109/MICRO56248.2022.00050

[18] Aosong Feng, Irene Li, Yuang Jiang, and Rex Ying. 2023. Diffuser: efficient transformers with multi-hop attention diffusion for long sequences. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 12772–12780.

[19] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot. (2023).

[20] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323* (2022).

[21] Yu Gong, Zhihan Xu, Zhezhi He, Weifeng Zhang, Xiaobing Tu, Xiaoyao Liang, and Li Jiang. 2022. N3H-core: Neuron-designed neural network accelerator via FPGA-based heterogeneous computing cores. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 112–122.

[22] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2017. Angel-eye: A complete design flow for mapping CNN onto embedded FPGA. *IEEE transactions on computer-aided design of integrated circuits and systems* 37, 1 (2017), 35–47.

[23] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H. Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W. Lee, and Deog-Kyoon Jeong. 2020. A^3: Accelerating Attention Mechanisms in Neural Networks with Approximation. arXiv:2002.10941 [cs.DC]

[24] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W. Lee. 2021. ELSA: Hardware-Software Co-design for Efficient, Lightweight Self-Attention Mechanism in Neural Networks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 692–705. https://doi.org/10.1109/ISCA52012.2021.00060

[25] 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 *2022 IEEE Hot Chips 34 Symposium (HCS)*. 1–17. https://doi.org/10.1109/HCS55958.2022.9895626

[26] Cheonsu Jeong. 2023. A Study on the Implementation of Generative AI Services Using an Enterprise Data-Based LLM Application Architecture. *arXiv preprint arXiv:2309.01105* (2023).

[27] Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, Amir Yazdanbakhsh, and Tushar Krishna. 2023. FLAT: An Optimized Dataflow for Mitigating Attention Bottlenecks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 295–310. https://doi.org/10.1145/3575693.3575747

[28] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W Mahoney, and Kurt Keutzer. 2023. SqueezeLLM: Dense-and-Sparse Quantization. *arXiv preprint arXiv:2306.07629* (2023).

[29] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451* (2020).

[30] Woosuk Kwon, Sehoon Kim, Michael W Mahoney, Joseph Hassoun, Kurt Keutzer, and Amir Gholami. 2022. A fast post-training pruning framework for transformers. *Advances in Neural Information Processing Systems* 35 (2022), 24101–24116.

[31] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[32] 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. arXiv:2007.08563 [cs.DC]

[33] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. 2023. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. *arXiv preprint arXiv:2306.00978* (2023).

[34] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A Co-Design Framework for Enabling Sparse Attention Using Reconfigurable Architecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 977–991. https://doi.org/10.1145/3466752.3480125

[35] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).

[36] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Nick Barnes, and Ajmal S. Mian. 2023. A Comprehensive Overview of Large Language Models. *ArXiv* abs/2307.06435 (2023).

[37] Yubin Qin, Yang Wang, Dazheng Deng, Zhiren Zhao, Xiaolong Yang, Leibo Liu, Shaojun Wei, Yang Hu, and Shouyi Yin. 2023. FACT: FFN-Attention Co-Optimized Transformer Architecture with Eager Correlation Prediction. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 22, 14 pages. https://doi.org/10.1145/3579371.3589057

[38] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[39] Mengshu Sun, Zhengang Li, Alec Lu, Yanyu Li, Sung-En Chang, Xiaolong Ma, Xue Lin, and Zhenman Fang. 2022. Film-qnn: Efficient fpga acceleration of deep neural networks with intra-layer, mixed-precision quantization. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 134–145.

[40] Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. 2023. Large language models in medicine. *Nature medicine* 29, 8 (2023), 1930–1940.

[41] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[42] Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023. Practitioners' Expectations on Code Completion. *ArXiv* abs/2301.03846 (2023).

[43] Erwei Wang, James J Davis, Georgios-Ilias Stavrou, Peter YK Cheung, George A Constantinides, and Mohamed Abdelfattah. 2022. Logic shrinkage: Learned FPGA netlist sparsity for efficient neural network inference. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 101–111.

[44] Haoran Wang, Haobo Xu, Ying Wang, and Yinhe Han. 2023. CTA: Hardware-Software Co-design for Compressed Token Attention Mechanism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 429–441. https://doi.org/10.1109/HPCA56546.2023.10070997

[45] Hanrui Wang, Zhekai Zhang, and Song Han. 2021. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 97–110.

[46] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. 2020. Shuhai: Benchmarking high bandwidth memory on fpgas. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 111–119.

[47] Zhican Wang, Gang Wang, Honglan Jiang, Ningyi Xu, and Guanghui He. 2023. COSA: Co-Operative Systolic Arrays for Multi-head Attention Mechanism in Neural Network using Hybrid Data Reuse and Fusion Methodologies. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[48] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).

[49] 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 *International Conference on Machine Learning*. PMLR, 38087–38099.

[50] Xilinx. 2021. Alveo U280 Data Center Accelerator Card Data Sheet. https://docs.xilinx.com/v/u/en-US/ds963-u280.

[51] Xilinx. 2023. Versal™ Architecture and Product Data Sheet. https://docs.xilinx.com/v/u/en-US/ds950-versal-overview.

[52] 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. *Advances in Neural Information Processing Systems* 35 (2022), 27168–27183.

[53] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big bird: Transformers for longer sequences. *Advances in neural information processing systems* 33 (2020), 17283–17297.

[54] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. [n. d.]. Opt: Open pre-trained transformer language models, 2022. *URL https://arxiv.org/abs/2205.01068* ([n. d.]).

[55] 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 *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 171–182.

[56] Han Zhao, Quan Chen, Yuxian Qiu, Ming Wu, Yao Shen, Jingwen Leng, Chao Li, and Minyi Guo. 2018. Bandwidth and locality aware task-stealing for manycore architectures with bandwidth-asymmetric memory. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 4 (2018), 1–26.

[57] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. 2021. Learning n: m fine-grained structured sparse neural networks from scratch. *arXiv preprint arXiv:2102.04010* (2021).