Structure-Aware Operator Fusion and Dynamic Materialization for Efficient Transformer Inference

Ruiting Sun, Honglu He, Guanwen Zhang*, and Wei Zhou
School of Electronics and Information, Northwestern Polytechnical University
Xi'an, Shaanxi, 710129, China
{sunrt, hehl}@mail.nwpu.edu.cn, {*guanwen.zh, zhouwei}@nwpu.edu.cn

***Abstract*—Deploying Transformer models in latency-critical applications is frequently hindered by memory bandwidth bottlenecks caused by fragmented operators and redundant global memory access. Conventional deep learning compilers often fail to optimize these heavy workloads effectively because their rule-based fusion heuristics remain overly conservative regarding complex graph dependencies. In this paper, we propose a novel optimization framework that combines Structure-Aware Operator Fusion with a Dynamic Materialization strategy. We introduce specialized kernels for Multi-Head Attention (MHA) and Feed-Forward Networks (FFN) that leverage shared memory tiling and streaming Softmax to rigorously minimize off-chip data traffic. Additionally, our approach resolves fusion conflicts at residual boundaries by dynamically searching for optimal materialization configurations instead of enforcing static safety barriers. Extensive evaluations on benchmarks including Vision Transformers (ViT) and Swin Transformer demonstrate significant improvements over industry standards. Our method achieves a 1.52× speedup over PyTorch on ViT-Base and consistently reduces inference latency compared to an auto-tuned baseline, while producing smaller binary artifacts than established frameworks in cross-compiler comparisons.**

***Index Terms*—Transformer Inference, Deep Learning Compiler, Operator Fusion, Dynamic Materialization, Memory Optimization**

I. Introduction

Transformer architectures have fundamentally revolutionized the landscape of deep learning, establishing state-of-the-art performance across a diverse spectrum of tasks ranging from natural language processing [1] to computer vision [2]. However, deploying these models in latency-critical applications remains a formidable challenge. The primary obstacle is the memory wall on modern hardware. The inference process is hindered by a massive number of fragmented operators and redundant global memory accesses. These issues are particularly acute within Multi-Head Attention (MHA) and Feed-Forward networks (FFN) where they saturate memory bandwidth and prevent compute units from reaching peak utilization.

While existing deep learning compilers attempt to mitigate these inefficiencies through automated operator fusion, general-purpose strategies often fall short when applied to the complex subgraphs of Transformers. Standard compilers struggle to fully exploit the specific geometric properties of attention mechanisms. This limitation leaves a gap between generic heuristic optimization and domain-specific kernel efficiency. They affect not only vision backbones but also massive generative models like GPT and LLaMA. Since these architectures share identical foundational blocks, optimizations targeting Attention and FFN layers offer universal benefits.

In this paper, we propose a novel optimization framework to bridge this gap. First, we introduce specialized fusion operators, `Fused_Attention` and `Fused_FFN`. These operators are designed to aggressively utilize on-chip shared memory. We leverage streaming softmax strategies to rigorously minimize off-chip data traffic. Second, we propose a Dynamic Materialization strategy to address fusion competition at residual boundaries. Unlike traditional compilers that guarantee safety through static barriers, our method treats materialization as a search problem. We dynamically evaluate potential subgraphs to identify the configuration that maximizes fusion benefits without violating data dependencies.

The overall framework of our approach is illustrated in Fig. 1. The main contributions are summarized as follows:

- We propose the `Fused_Attention` operator, a domain-specific fusion operator for the self-attention mechanism that utilizes shared memory tiling to fuse Softmax and masking operations. This effectively eliminates the bottleneck of intermediate memory I/O.
- We develop the `Fused_FFN` operator, a fused operator tailored for the FFN that integrates linear projections with activations. It effectively reuses registers and significantly improves the throughput of the Multi-Layer Perceptron (MLP) layers in Transformer blocks.
- We introduce a *Dynamic Materialization* search algorithm that replaces static heuristic boundaries. This method dynamically explores the trade-off space for materialization points at complex graph intersections and resolves fusion conflicts between residual connections and normalization layers.

We conduct extensive experiments on standard benchmarks including ViT [2], Swin Transformer [3], and Data-efficient Image Transformers (DeiT) [4]. Our evaluation compares the proposed method against PyTorch [5] and Ansor [6]. The results confirm that our method effectively mitigates memory bottlenecks and achieves superior inference performance.

*Corresponding Author: Guanwen Zhang
This work was supported by the National Major Science and Technology Projects of China under Grant 2025ZD1401102.

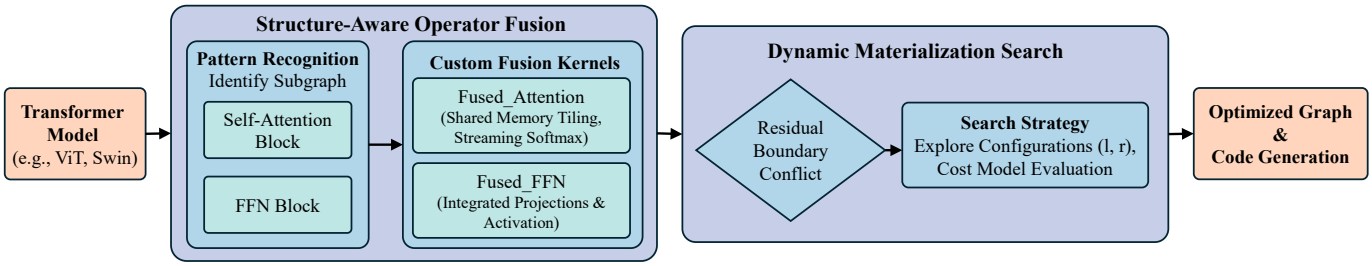

Fig. 1. The overall framework of Structure-Aware Operator Fusion and Dynamic Materialization. The system identifies Transformer-specific patterns, applies custom fusion kernels, and resolves boundary conflicts via a dynamic search strategy before code generation.

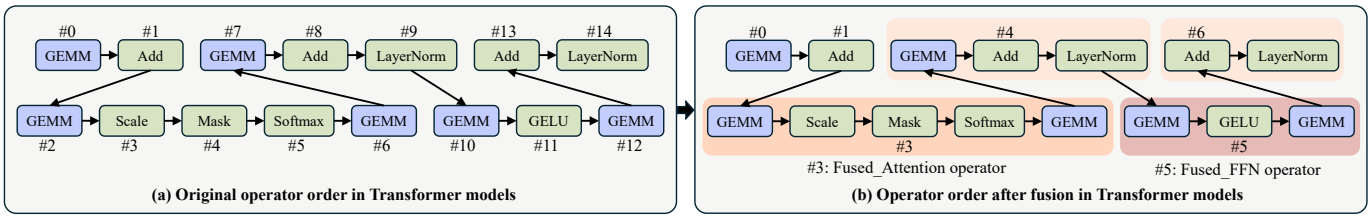

Fig. 2. Operator execution flow before and after fusion: (a) the original fragmented operator sequence in a standard Transformer layer; (b) the fused execution where operations are consolidated into `Fused_Attention` and `Fused_FFN` to reduce global-memory traffic and kernel-launch overhead.

II. Related Work

A. Operator Fusion in Deep Learning Compilers

Deep learning compilers function as the bridge between model definitions and hardware intrinsics, yet early frameworks like TVM [7] rely on rigid rule-based strategies that often falter under complex Transformer dependencies. While these systems attempt to merge operators via pattern matching, the intricate triangular structures common in attention mechanisms frequently trigger conservative fallback rules. To overcome this rigidity, search-based frameworks such as Ansor [6] automate tensor program generation using learned cost models, although they typically remain bound by the partitioning limitations of upstream compilers. Recent research has shifted toward holistic optimization. Welder [8] employs tile-graph abstractions to unify memory access, while Hidet [9] allows flexible fusion through a task-mapping paradigm. Further advancements include Ladder [10], which optimizes custom data types via hardware-aware transformations. Most recently, SpaceFusion [11] and Neptune [12] have introduced unified space mappings to improve locality, yet our work uniquely addresses the dynamic materialization needs at residual boundaries that these static analyses often overlook.

B. Efficient Attention and FFN Kernels

Standard self-attention implementations suffer from quadratic memory complexity. This $O(N^2)$ footprint forces the materialization of large attention matrices to High Bandwidth Memory multiple times. Milakov and Gimelshein [13] introduced the *Online Softmax* algorithm. This technique enables the calculation of Softmax statistics in a single streaming pass without materializing the full matrix. Building on this, FlashAttention [14] leverages tiling and recomputation to fuse the entire attention mechanism into a single CUDA kernel. By keeping the attention matrix in on-chip shared memory (SRAM), these methods achieve linear memory complexity. Ye et al. introduced FlashInfer [15] to generate customizable attention kernels for Large Language Models. Their approach employs block-sparse formats and just-in-time compilation to handle heterogeneous requests efficiently. Parallel efforts such as FlashDecoding++ [16] and ByteTransformer [17] have optimized the decoding phase of Large Language Models, specifically addressing the bottlenecks of flat General Matrix Multiply (GEMM) operations. Optimization efforts also extend to the feed-forward networks. TARDIS [18] accelerates these layers by treating activation functions as partially linear. This method reduces parameter counts through constant folding without sacrificing model accuracy. However, seamless integration of these specialized kernels into automated compilation pipelines remains difficult, as it requires bridging the semantic gap between high-level graph definitions and hardware-specific optimizations without manual intervention.

C. Materialization and Correctness in Graph Optimization

A critical challenge in graph optimization is determining where to materialize tensors. This decision dictates precisely where to break fusion and write data to global memory. In residual blocks, the input feeds both the FFN branch and the residual addition, creating a competition for fusion direction. Standard compilers usually resolve this by enforcing static fusion barriers at nodes with multiple consumers to prevent dependency cycles. While this approach guarantees numerical correctness, it often precludes opportunities for vertical fusion. Recent systems have begun to address this rigidity through more dynamic management strategies. For instance, vLLM [19] introduced PagedAttention to manage

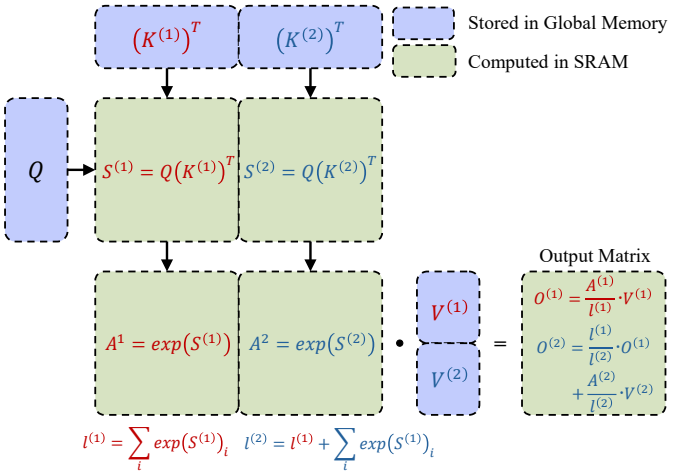

Fig. 3. Schematic of the matrix blocking calculation within the Fused Attention operator. Intermediate results are accumulated in on-chip shared memory (SRAM) using the Online Softmax update rules to minimize global memory access.

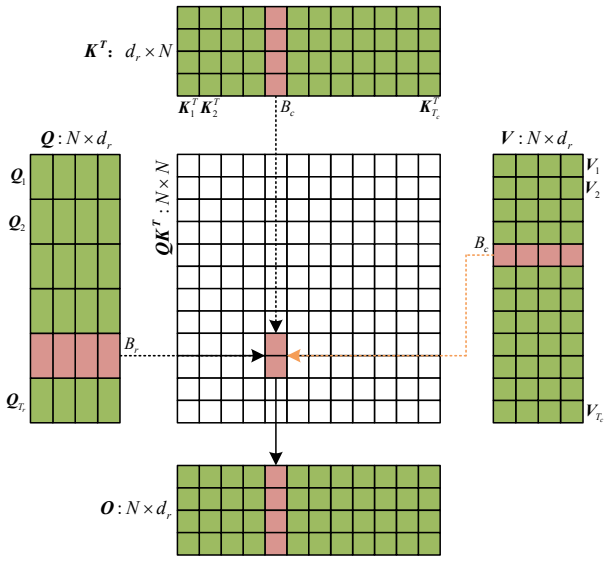

Fig. 4. Schematic of the Fused Attention algorithm and tiling strategy. The Query matrix is tiled by rows, while Key and Value matrices are tiled by columns to fit within shared memory.

memory dynamically, which reduces fragmentation and enables higher throughput. Similarly, SpotServe [20] tackles dynamic availability in distributed settings, demonstrating that flexible context management can significantly reduce latency. Despite these advances, standard inference compilers continue to rely heavily on static analysis for materialization decisions. Consequently, they frequently fail to optimize the complex trade-offs at residual boundaries, missing scenarios where recomputation is more efficient than memory access.

III. Methodology

To address the efficiency bottlenecks caused by memory fragmentation, we propose a structure-aware optimization framework. As illustrated in Fig. 2, the standard Transformer execution (a) involves numerous small operators that saturate memory bandwidth. Our approach (b) consolidates these operations into coarse-grained kernels. Specifically, we introduce the `Fused_Attention` operator to merge the scale, mask, and softmax steps with matrix multiplications, and the `Fused_FFN` operator to integrate projections with activation functions.

A. Fused Attention Operator

We introduce the `Fused_Attention` operator to streamline the self-attention mechanism. Our approach adapts the Online Softmax algorithm [13] to perform computation within shared memory. We employ a streaming formulation. Let $S \in \mathbb{R}^{N\times N}$ denote the attention score matrix computed as QK^T. We decompose the computation into blocks to fit within the shared memory. We partition the Query matrix Q into row blocks and the Key K and Value V matrices into column blocks. During the computation of a row block of the output, we iterate over the column blocks of K and V. For each block, we maintain running statistics: the current local maximum m and the current unnormalized sum ℓ. When a new block of scores S_{new} is computed, we update the global maximum

Algorithm 1 Fused Attention with Online Softmax

Input: Matrices $Q, K, V \in \mathbb{R}^{N\times d}$ residing in global memory
Output: Result Matrix $O \in \mathbb{R}^{N\times d}$
Parameters: Tile sizes B_r, B_c for row and column blocking
Partition Q into $T_r = N/B_r$ blocks $\{Q_i\}$
Partition K, V into $T_c = N/B_c$ blocks $\{K_j, V_j\}$
for $i = 1$ to T_r **do**
  Load block Q_i from global memory to Shared Memory
  Initialize accumulator $O_i^{(0)} = \mathbf{0} \in \mathbb{R}^{B_r\times d}$
  Initialize statistics $\ell_i^{(0)} = \mathbf{0}, m_i^{(0)} = -\infty \in \mathbb{R}^{B_r}$
  for $j = 1$ to T_c **do**
    Load blocks K_j and V_j into Shared Memory
    Compute scores $S_{ij} = Q_i K_j^T$ stored in registers
    Compute block max $m_{ij} = \text{rowmax}(S_{ij})$
    $m_{new} = \max(m_i^{(j-1)}, m_{ij})$
    Compute correction factor $\alpha = e^{m_i^{(j-1)} - m_{new}}$
    Compute current unnormalized prob $P_{ij} = e^{S_{ij} - m_{new}}$
    Update sum $\ell_i^{(j)} = \ell_i^{(j-1)} \cdot \alpha + \text{rowsum}(P_{ij})$
    Update accumulator $O_i^{(j)} = O_i^{(j-1)} \cdot \alpha + P_{ij} V_j$
    Update max state $m_i^{(j)} = m_{new}$
  end for
  Finalize output $O_i = O_i^{(T_c)} / \ell_i^{(T_c)}$
  Write block O_i to global memory
end for

$m_{new} = \max(m_{prev}, \max(S_{new}))$ and adjust the accumulated sum and partial output accordingly. The update rule ensures that we can effectively perform the Softmax normalization and the subsequent weighted summation in a single pass without storing the full $N \times N$ matrix.

The mathematical derivation for the update of the partial

Algorithm 2 Fused Feed-Forward Network

Input: Input tensor $X \in \mathbb{R}^{B\times N\times d}$
Weights: $W_1 \in \mathbb{R}^{d\times 4d}, b_1 \in \mathbb{R}^{4d}, W_2 \in \mathbb{R}^{4d\times d}, b_2 \in \mathbb{R}^{d}$
Output: Output tensor $Y \in \mathbb{R}^{B\times N\times d}$
Define tile sizes T_b, T_n fitting Shared Memory capacity
for block coordinates (bb, nn) in (B, N) with stride (T_b, T_n) **do**
  Load input tile X_{tile} to Shared Memory
  Initialize intermediate buffer H_{local} in Shared Memory
  {First Linear Layer + Activation}
  for each thread in thread block **do**
    Compute row $h = X_{tile}[local\_idx] \cdot W_1 + b_1$
    Apply activation $h = \text{GeLU}(h)$
    Store h to H_{local}
  end for
  Synchronize Threads
  {Second Linear Layer}
  for each thread in thread block **do**
    Compute output row $y = H_{local}[local\_idx] \cdot W_2 + b_2$
    Store y to global output Y
  end for
end for

output O and the normalization factor ℓ is as follows. As shown in Fig. 3, red terms correspond to the intermediate state after the first K/V tile and blue terms show the updated state after the next tile is accumulated under Online Softmax. Given a previous maximum m_{prev} and a previous sum ℓ_{prev}, upon computing a new score block with local maximum m_{local}, the new global maximum is $m_{new} = \max(m_{prev}, m_{local})$. The normalization factor is updated as:

$$\ell_{new} = \ell_{prev} \cdot e^{m_{prev} - m_{new}} + \sum e^{S_{new} - m_{new}}. \quad (1)$$

Simultaneously, the accumulated output buffer O is rescaled to account for the shift in the maximum value:

$$O_{new} = O_{prev} \cdot e^{m_{prev} - m_{new}} + P_{new} \cdot V_{block}, \quad (2)$$

where P_{new} represents the unnormalized probabilities of the current block. This recursive formulation allows the entire attention operation to be executed within the registers and shared memory of the GPU.

The execution logic of the `Fused_Attention` operator is visualized in Fig. 4 and formalized in Algorithm 1. The algorithm utilizes a double-loop structure. The outer loop tiles the Query matrix, loading a block Q_i into shared memory. The inner loop iterates through the blocks of Key and Value matrices (K_j, V_j). By keeping the accumulator in registers, we reduce the global memory accesses to merely reading the inputs Q, K, V and writing the final output O, thereby reducing the global memory traffic to $O(N \cdot d)$ and avoiding the $O(N^2)$ materialization overhead.

B. Fused FFN Operator

The FFN layer typically follows the attention block and consists of two linear transformations separated by a non-linear activation function, such as Gaussian Error Linear Unit (GeLU). The standard mathematical definition is:

$$FFN(x) = W_2(\text{GeLU}(W_1 x + b_1)) + b_2. \quad (3)$$

A critical characteristic is the expansion of the hidden dimension to $\mathbb{R}^{4d}$. We propose the `Fused_FFN` operator to manage this expansion efficiently. This operator fuses the linear projections, bias addition, and GeLU activation into a single kernel. The key strategy involves utilizing the shared memory of the GPU as a scratchpad. This allows us to store the expanded intermediate results locally. We employ a tiling strategy that partitions the input tensor along the batch and sequence dimensions. For each tile, the data is loaded into shared memory, and the first matrix multiplication is performed. The results are kept in registers or shared memory, where the GeLU activation is immediately applied. Crucially, instead of writing this expanded $4d$ result to global memory, we directly consume it for the second matrix multiplication.

Algorithm 2 details the implementation, and the structural comparison is depicted in Fig. 5. By processing the data in small tiles $(tile\_b, tile\_n)$, we ensure that the intermediate expanded rows fit within the limited shared memory capacity. This approach effectively cuts the global memory traffic by half, as the large intermediate tensor is transient and never leaves the on-chip memory. Furthermore, this fusion enables the compiler to perform aggressive register reuse optimization, keeping the accumulation variables for the second matrix multiplication in registers throughout the computation.

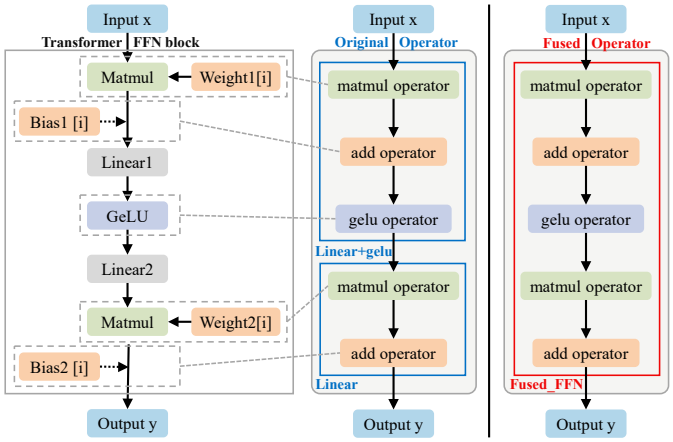

Fig. 5. Structure of the Fused_FFN operator. The hidden dimension is expanded to $4\times d$ on-chip in shared memory, preventing the large intermediate activation tensor from consuming global memory bandwidth.

C. Dynamic Materialization

To resolve the fusion ambiguity inherent in diamond dependency topologies, such as the Residual Additions identified in Section II, we formulate the placement of materialization points as a search optimization problem rather than relying on static barriers.

We focus on fusion competition boundaries, where a subgraph is flanked by nodes that could potentially be fused in

Algorithm 3 Dynamic Materialization Search Strategy

Input: Computational Graph G, Set of Competitive Subgraphs $\mathcal{C}$
Output: Optimized Graph G'
for each subgraph $G_{sub} \in \mathcal{C}$ (e.g., FFN-Residual boundary) **do**
  Identify Predecessor P and Successor N (e.g., LayerNorm)
  Initialize strategy set $\mathcal{S} = \emptyset$
  for left_mat $l \in \{True, False\}$ **do**
    for right_mat $r \in \{True, False\}$ **do**
      if IsCompatible(P, G_{sub}, N, l, r) **then**
        Construct candidate graph G_{cand} with materialization points determined by l, r
        Apply fusion rules to G_{cand}
        Estimate latency $t = \text{CostModel}(G_{cand})$
        $\mathcal{S} \leftarrow \mathcal{S} \cup \{(l, r, t)\}$
      end if
    end for
  end for
  Select optimal strategy $(l^*, r^*) = \arg\min_t \mathcal{S}$
  Apply (l^*, r^*) to update G
end for
return Updated Graph G

multiple directions. We introduce binary decision variables corresponding to the boundaries of these competitive subgraphs. Let l be a boolean variable representing the materialization decision at the input of the subgraph, and r be the decision at the output. The search algorithm enumerates the possible configurations of (l, r). For each configuration, we construct a candidate computational graph. A topological validity check is performed to ensure that the fusion configuration does not introduce circular dependencies or violate hardware resource constraints such as shared memory limits. For all valid configurations, we employ a cost model to estimate the execution latency. This cost model considers the global memory traffic and the kernel launch overhead.

As outlined in Algorithm 3, the system iterates through critical subgraphs such as the FFN-Residual-LayerNorm junction. By evaluating the trade-offs dynamically, the system can identify scenarios where recomputing small portions of the graph is cheaper than accessing global memory, or conversely, where materialization is necessary to prevent register spilling. This approach ensures that the compiler selects the minimum-latency valid configuration under our cost model, tailored to the specific model architecture and hardware constraints.

IV. Experiments

A. Setup

Hardware and Software Environment. We conduct all experiments on a workstation equipped with an Intel Xeon Silver 4214R CPU, and an NVIDIA GeForce RTX 3090 GPU. The software stack builds upon TVM v0.19.dev0 and ingests models via the Open Neural Network Exchange (ONNX) 1.17.

TABLE I
Hyperparameter Configurations for Micro-benchmarks

Parameter	Configuration
Batch Size (B)	32, 64
Sequence Length (N)	128, 256, 512, 768, 1024, 2048
Data Precision	float32, int8
Head Dimension (d_k)	64
FFN Expansion Ratio	4

TABLE II
Cross-compiler comparison on ViT-Base: compiled artifact size (MB) and end-to-end inference latency (ms). Speedup = t_PyTorch/t_Framework

Framework	Size (MB)	Time (ms)	Speedup
PyTorch	344	24.22	1.00×
ONNX-MLIR	292	19.62	1.23×
Triton	239	16.13	1.50×
Ours	**231**	**15.89**	**1.52×**

We strictly define our baseline as the TVM framework augmented with Ansor (10000 iterations per operator), operating under default operator fusion heuristics and standard scheduling strategies. This rigorous process ensures we compare our method against highly optimized tensor programs rather than naive implementations.

Benchmarks. We evaluate diverse Vision Transformer architectures from the `timm` library on the ImageNet-1K classification task. The model selection includes ViT, DeiT, and Swin Transformer variants. These architectures range from canonical global attention structures to complex shifted window topologies, effectively stress-testing our Dynamic Materialization strategy against non-sequential dependencies.

Micro-benchmarks. We also construct isolated micro-benchmarks for Self-Attention and FFN modules. This approach allows for controlled hyperparameter sweeps across batch sizes, sequence lengths, and data precisions. Table I details the specific configurations used for these kernel-level assessments.

B. Main Results

We first present end-to-end performance comparisons against industry standards, followed by micro-benchmark analyses of the individual `Fused_Attention` and `Fused_FFN` operators.

End-to-End Performance. We position our framework against established deep learning systems including PyTorch, ONNX-MLIR, and OpenAI Triton using the ViT-Base benchmark. Our method achieves the lowest inference latency of 15.89 ms as summarized in Table II. This performance represents a 1.52× speedup over PyTorch, which typically suffers from dispatcher overhead and kernel fragmentation. While Triton delivers comparable kernel efficiency, our approach edges it out by 1.5% through graph-level optimizations enabled by Dynamic Materialization. These strategies effectively minimize data movement between kernels. Furthermore, the system produces a compact binary size of 231 MB. This marks

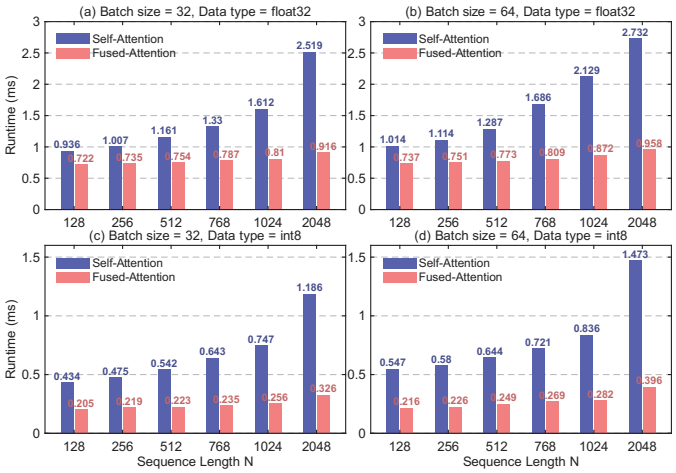

Fig. 6. Runtime of Self-Attention vs. Fused_Attention across sequence lengths (subfigures follow the batch size and precision settings as labeled).

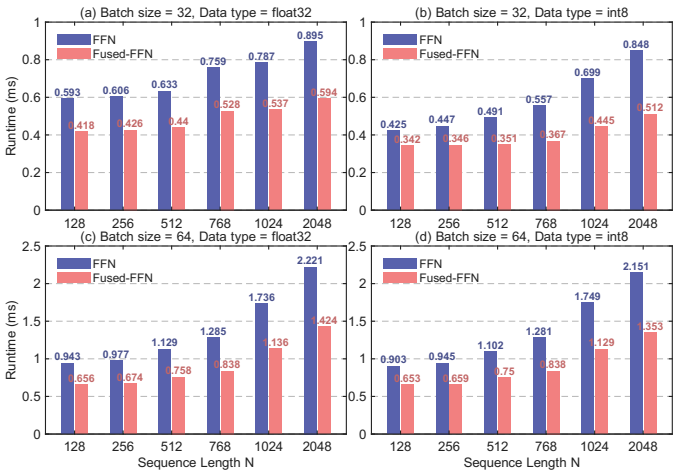

Fig. 7. Runtime of FFN vs. Fused_FFN across sequence lengths (subfigures follow the batch size and precision settings as labeled).

TABLE III
FUSED_ATTENTION SPEEDUP OVER THE BASELINE IMPLEMENTATION (SPEEDUP = T_BASELINE/T_OURS) UNDER THE MICRO-BENCHMARK SETTINGS IN TABLE I.

Batch	Precision	Seq 128	Seq 1024	Seq 2048
32	float32	1.29×	1.99×	2.75×
32	int8	2.12×	2.92×	3.64×
64	float32	1.37×	2.44×	2.86×
64	int8	2.53×	2.96×	3.72×

TABLE IV
FUSED_FFN SPEEDUP OVER THE BASELINE IMPLEMENTATION (SPEEDUP = T_BASELINE/T_OURS) UNDER THE MICRO-BENCHMARK SETTINGS IN TABLE I.

Batch	Precision	Seq 128	Seq 1024	Seq 2048
32	float32	1.41×	1.47×	1.51×
32	int8	1.24×	1.57×	1.58×
64	float32	1.43×	1.54×	1.56×
64	int8	1.38×	1.55×	1.59×

a 33% reduction compared to PyTorch and indicates highly efficient code generation with aggressive graph pruning.

Fused Attention Performance. The empirical results for the `Fused_Attention` operator appear in Fig. 6 and Table III. A clear trend emerges where computational speedup correlates positively with increasing sequence length. Standard baseline implementations suffer from quadratic memory complexity which causes severe latency spikes as the input grows. In contrast, our fused kernel maintains near-linear scaling behavior even as the context window expands. The performance gap becomes most apparent at longer sequences. For instance, at a sequence length of 2048 with a batch size of 32 using float32 precision, the method delivers a substantial 2.75× speedup.

This advantage is further amplified in quantized environments. The int8 implementation achieves a remarkable 3.72× speedup at a batch size of 64. These metrics indicate that our tiling strategy effectively relieves memory bandwidth pressure. By keeping the attention scores in on-chip SRAM, the system prevents high-throughput compute units from stalling while waiting for global memory fetches.

Fused FFN Performance. The `Fused_FFN` operator exhibits a different performance profile. Its efficiency gains remain remarkably stable regardless of sequence length variations. Fig. 7 and Table IV illustrate its consistency with speedup factors generally ranging between 1.4× and 1.6×. The primary driver for this improvement is the efficient management of dimensional expansion. The FFN typically expands hidden dimension by a factor of four. Instead of materializing this large intermediate tensor to global memory, our kernel processes these values entirely within shared memory.

This approach transforms the operation from memory-bound to compute-bound. The stability of the results suggests that we have successfully removed a constant factor of overhead related to memory I/O. We observe the highest efficiency in the quantized setting, where the int8 implementation at batch size 64 reaches a 1.59× speedup. This confirms that fusing element-wise activation functions directly into the matrix multiplication pipeline successfully amortizes kernel launch overheads and maximizes data reuse. Crucially, we verified numerical stability. The maximum relative error remains within 10^{-5}, ensuring no accuracy degradation.

C. Ablation Studies

To assess how our methods affect end-to-end inference, we conducted an ablation study on four system configurations: a TVM baseline, versions enabling only `Fused_Attention` or `Fused_FFN`, and the full system with both operators plus Dynamic Materialization. We tested six Transformer models; Table V reports their inference latency under each configuration.

With only `Fused_Attention`, we observed the largest isolated gains, especially on Swin Transformer models. For example, Swin-Tiny latency fell from 10.37 ms to 9.39 ms, yielding a 1.10× speedup. This architecture benefits strongly because our fusion operator handles windowing internally,

TABLE V
ABLATION STUDIES: END-TO-END INFERENCE LATENCY (MS). "+ATTN" ENABLES ONLY FUSED_ATTENTION; "+FFN" ENABLES ONLY FUSED_FFN; "+FULL" ENABLES BOTH FUSED KERNELS PLUS DYNAMIC MATERIALIZATION.

Model	Baseline	+Attn	+FFN	+Full
ViT-Small	8.49	7.84	8.26	**7.69**
ViT-Base	17.38	16.47	17.24	**15.89**
DeiT-Small	7.86	7.68	7.74	**7.53**
DeiT-Base	15.43	15.13	15.34	**14.92**
Swin-Tiny	10.37	9.39	10.13	**9.17**
Swin-Small	18.41	16.82	17.93	**16.65**

avoiding explicit mask generation and complex tensor reshaping usually required by standard compilers.

In comparison, the `Fused_FFN` operator provides the system with stable additive performance improvements. Although its single-point acceleration on some models is not as drastic as the attention operator, it effectively reduces memory bandwidth pressure by processing dimension expansion in shared memory and fusing activation functions. Experimental data shows that this operator contributed to latency reduction across all tested models, verifying the effectiveness of the strategy of fusing linear projection with activation functions when processing MLP layers in Transformer blocks.

Finally, the Full System showed synergy beyond the simple sum of its parts. Here, Dynamic Materialization was crucial for resolving diamond dependencies at residual connections. By dynamically finding safe configurations that fuse the FFN tail with Residual Add and LayerNorm, the system reduced ViT-Base latency to 15.89 ms, for a 1.09× overall improvement over the baseline.

V. CONCLUSION

In this paper, we presented a comprehensive optimization framework to address the memory bandwidth bottlenecks in Transformer inference. By integrating Structure-Aware Operator Fusion with Dynamic Materialization, we bridge the gap between generic compiler heuristics and domain-specific kernel efficiency. Our approach utilizes specialized operators that leverage shared memory tiling to minimize redundant global memory I/O. Simultaneously, the proposed dynamic search algorithm resolves scheduling ambiguities at residual connections to optimize graph partitioning. Extensive evaluations confirm that this methodology yields consistent speedups and reduced latency across diverse architectures, outperforming established industry frameworks. Future research will focus on extending these strategies to Large Language Models and validating portability across diverse hardware backends.

REFERENCES

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[2] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," in *International Conference on Learning Representations*, 2020.

[3] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 10 012–10 022.

[4] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, "Training data-efficient image transformers & distillation through attention," in *International conference on machine learning*. PMLR, 2021, pp. 10 347–10 357.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[6] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating high-performance tensor programs for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 863–879.

[7] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[8] Y. Shi, Z. Yang, J. Xue, L. Ma, Y. Xia, Z. Miao, Y. Guo, F. Yang, and L. Zhou, "Welder: Scheduling deep learning memory access via tile-graph," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 701–718.

[9] Y. Ding, C. H. Yu, B. Zheng, Y. Liu, Y. Wang, and G. Pekhimenko, "Hidet: Task-mapping programming paradigm for deep learning tensor programs," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 370–384.

[10] L. Wang, L. Ma, S. Cao, Q. Zhang, J. Xue, Y. Shi, N. Zheng, Z. Miao, F. Yang, T. Cao *et al.*, "Ladder: Enabling efficient low-precision deep learning computing through hardware-aware tensor transformation," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 307–323.

[11] L. Zhu, J. Yao, and H. Guan, "Spacefusion: Advanced deep learning operator fusion via space-mapping graph," in *Proceedings of the Twentieth European Conference on Computer Systems*, 2025, pp. 787–802.

[12] Y. Zhao, E. Johnson, P. Chatarasi, V. Adve, and S. Misailovic, "Neptune: Advanced ml operator fusion for locality and parallelism on gpus," *arXiv preprint arXiv:2510.08726*, 2025.

[13] M. Milakov and N. Gimelshein, "Online normalizer calculation for softmax," *arXiv preprint arXiv:1805.02867*, 2018.

[14] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.

[15] Z. Ye, L. Chen, R. Lai, W. Lin, Y. Zhang, S. Wang, T. Chen, B. Kasikci, V. Grover, A. Krishnamurthy *et al.*, "Flashinfer: Efficient and customizable attention engine for llm inference serving," *arXiv preprint arXiv:2501.01005*, 2025.

[16] K. Hong, G. Dai, J. Xu, Q. Mao, X. Li, J. Liu, K. Chen, Y. Dong, and Y. Wang, "Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 148–161, 2024.

[17] Y. Zhai, C. Jiang, L. Wang, X. Jia, S. Zhang, Z. Chen, X. Liu, and Y. Zhu, "Bytetransformer: A high-performance transformer boosted for variable-length inputs," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 344–355.

[18] G. Hu, Z. Wang, J. Wei, W. Huang, and H. Chen, "Accelerating large language models through partially linear feed-forward network," *arXiv preprint arXiv:2501.10054*, 2025.

[19] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th symposium on operating systems principles*, 2023, pp. 611–626.

[20] X. Miao, C. Shi, J. Duan, X. Xi, D. Lin, B. Cui, and Z. Jia, "Spotserve: Serving generative large language models on preemptible instances," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 1112–1127.