TransFusion: End-to-End Transformer Acceleration via Graph Fusion and Pipelining

Linxuan Zhang
Department of Electrical and Computer Engineering
University of Alberta
Edmonton, Alberta, Canada
linxuan2@ualberta.ca

José Nelson Amaral
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
jamaral@ualberta.ca

Di Niu
Department of Electrical and Computer Engineering
University of Alberta
Edmonton, Alberta, Canada
dniu@ualberta.ca

Abstract

Transformer acceleration has increasingly emphasized local fusion within isolated submodules, such as multi-head attention (MHA) and softmax. However, as Transformer models continue to scale in both depth and context length, such fragmented optimizations fail to address end-to-end inefficiencies across the full encoder/decoder stack. This paper presents TransFusion, a comprehensive framework for end-to-end Transformer layers, including QKV projections, MHA, LayerNorm, and FFN, as structured Einsum Cascades, enabling precise modelling of data dependencies and execution order. TransFusion introduces DPipe, a unified graph-based scheduler that partitions the Einsum-centric directed acyclic graph (DAG) and applies latency-aware pipelining across hardware hierarchies using dynamic programming (DP). To enable scalable execution under strict memory budgets, TransFusion integrates TileSeek, a Monte Carlo Tree Search (MCTS)-based tiling search algorithm that balances buffer reuse and system constraints. Evaluated across both cloud and edge architecture, TransFusion achieves up to an average of 1.6× speedup on cloud and 2.2× on edge over the prior state-of-the-art, FuseMax, by jointly optimizing inter-layer data reuse, intra-layer pipelining, and operator scheduling.

Keywords

Transformer, Accelerator, Operator Fusion, Tensor Tiling, Pipelining Scheduler, Tiling Search

1 Introduction

Transformer [45] has emerged as the foundational architecture behind a wide array of state-of-the-art models in natural-language understanding [8] [38] [19] [22] and generation [4] [39] [56] [44]. The scaling of Transformers — with more layers — to handle longer contexts shifts bottlenecks from computation to memory bandwidth, data movement, and scheduling, has led to intensive research into software-hardware co-designs for efficient deployment both in cloud infrastructure and edge devices [3] [40] [18] [58].

A substantial body of prior work has concentrated on optimizing the attention mechanism to reduce the data transfer between high-latency off-chip memory and on-chip buffer. FLAT [18] applies multi-level granularity tiling and sub-operator fusion to linearize memory growth. Frameworks such as xFormers [20], NVIDIA TensorRT [33], and Apple CoreML [1] follow similar principles to optimize Transformer subgraphs across diverse platforms. Moving beyond DRAM-SRAM interactions, FuseMax [31] further minimizes data movement between SRAM and register files by expanding PE register capacity (10 entries per PE), allowing full in-register retention of intermediate results and deeper operator fusion.

Further work emphasizes pipeline-level optimization under hardware constraints. FlashAttention [6] [5] [40] overlaps General Matrix Multiplication (GEMM) and softmax via warp-level tiling for NVIDIA A100/H100 GPUs, while MAS-Attention [41] adopts a similar pipeline with row-wise chunking to improve data reuse in the edge accelerators. FuseMax [31], focusing on cloud-centric pipelines, employs Extended Einsum representations to guide operator fusions and pipelines partial softmax over 2D PE arrays, pushing PE utilization to its architectural limits.

However, the potential of end-to-end Transformer fusion is yet to be fully explored. Fully fusing the QKV projection, multi-head attention (MHA), feed-forward network (FFN), and Layer Normalization (LayerNorm) remains a significant challenge due to several factors. First, the algorithm complexity and intermediate data movement increase with context length and model hierarchy depth, exacerbating memory bottlenecks. Second, scheduling and fusion strategies must be aware of and able to adapt to performance characteristics and constraints of diverse hardware architectures on cloud and edge. Third, end-to-end fusion necessitates joint tiling across multiple fused modules; as the fusion scope and operator complexity increase, the difficulty of searching for global tiling factors escalates dramatically, posing a major barrier to scalable optimization.

To address these challenges, this paper proposes TransFusion, a comprehensive and architecture-aware framework for end-to-end Transformer acceleration, that extends beyond attention calculation [18] [6] [5] and does not depend on hardware-specific assumptions [31] [41] [40]. TransFusion performs full-stack fusion by leveraging Einsum Cascades to capture the fine-grained computational patterns of QKV, MHA, FFN, and LayerNorm in Transformers. Guided by Einsum formulations, we introduce DPipe, a directed acyclic graph (DAG) pipeline scheduler as the backbone of on-chip execution strategy, which models the compute cost of each Einsum while performing execution stage scheduling. Additionally, a full-stack tiling search algorithm that spans the entire encoder-decoder stack efficiently explores the expanded search space with fusion-aware and hardware-specific considerations. TransFusion is open-source on Github.[1] In summary, this paper makes the following contributions:

- **End-to-End Fusion:** A two-level fusion strategy that spans both inter-layer and intra-layer optimization. TransFusion

MICRO 2025, Seoul, Korea
2025. ACM ISBN 978-X-XXXX-XXXX-X/XX/XX
https://doi.org/XXXXXXX.XXXXXXX

[1]https://github.com/FusedMindLab/TransFusion

expresses QKV, MHA, Add & LayerNorm, and FFN as structured Einsum Cascades, enabling fine-grained scheduling across the entire encoder-decoder stack. At the inter-layer, TransFusion performs sequence-wise outer tiling, enabling on-chip propagation of intermediate results across layers. At the intra-layer, TransFusion schedules fine-grained pipelinable inner tiles onto 1D/2D PE arrays to enable overlapped execution across epochs and improve parallelism.[2]

- **DPipe:** A graph-based pipeline scheduling framework that constructs operation-level DAGs from Einsum-based representations of Transformer layers, enabling precise modelling of computation dependencies. DPipe partitions the DAG into pipelinable subgraphs, taking into account hardware-specific characteristics such as memory hierarchy, compute parallelism, and PE granularity to generate optimized execution plans under hardware constraints using dynamic programming (DP).
- **TileSeek:** A full-stack tiling search algorithm leveraging Monte Carlo Tree Search (MCTS) to explore the joint design space of tiling factors and mapping parameters to maximize cache reuse and data locality, enabling effective adaptation to hardware-specific memory hierarchies. TileSeek also analyzes tile feasibility and embedding vector integrity as critical requirements for module-level fusion, ensuring both the implementability and correctness of end-to-end fusion.

2 Background

This section reviews a general architecture for cloud and edge devices, the standard Transformer design, recent acceleration techniques, and the Einsum-based abstractions that enable efficient execution on modern hardware.

2.1 Architecture

Prior research primarily targets specific hardware: FlashAttention-1/2/3 are optimized for NVIDIA GPUs (A100/H100), while Google's TPU v2/v3[32] underpins several spatial and dataflow accelerators[18, 31]. For edge scenarios, MAS-Attention builds on a modified spatial accelerator from TileFlow[58].

TransFusion generalizes across both cloud (e.g., TPU v2/v3[32]) and edge-oriented neural processing units (NPUs)[18, 31, 58]. Our evaluation adopts the architecture in Figure 1, featuring off-chip memory, a shared on-chip buffer, and two compute arrays: a 2D PE array for matrix-dense operations and a 1D PE array for streaming/vector workloads.

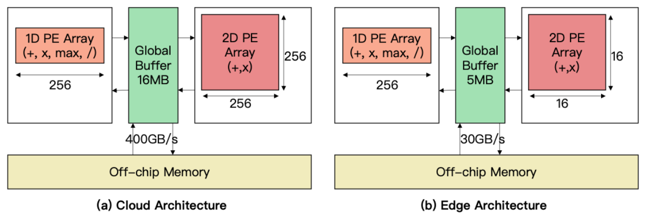


Figure 1: Simulated Cloud and Edge and Architecture Design.

[2]The outer tile represents off-chip memory to on-chip buffer and the inner tile represents on-chip buffer to PE.

For hardware simulation, we adopt Timeloop and Accelergy, developed by NVIDIA and MIT[34, 51]. Timeloop supports loop-level dataflow and mapping analysis for deep neural networks (DNNs) on spatial accelerators, while Accelergy offers cycle-level energy estimation across compute and memory hierarchies. We integrate both to model Einsum performance under cloud and edge architectures, and to analyze energy breakdowns across compute arrays and memory components.

2.2 Transformer

The Transformer [45] is a fully attention-based architecture for sequence modelling, eliminating recurrence and convolutions. It adopts a modular encoder-decoder design built from stacked multi-head self-attention (MHA) and feed-forward networks (FFN), each wrapped with residual connections and layer normalization for training stability. The encoder transforms token embeddings into contextual representations via repeated MHA-FFN layers, while the decoder employs masked self-attention for auto-regressive generation and encoder-decoder attention to integrate source context. Self-attention enables the model to capture global dependencies across the sequence in parallel.

At the core of the architecture is MHA, which projects input sequences into multiple sets of queries Q, keys K, and values V via learned linear transformations. Each head computes scaled dot-product attention in parallel to capture diverse contextual dependencies.

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

These heads operate in distinct representation subspaces, enabling the model to capture diverse relational patterns. Their outputs are concatenated and linearly projected:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{2}$$

Each sub-layer is surrounded by a residual connection, followed by layer normalization, which stabilizes training by re-centering and re-scaling the inputs:

$$\text{LayerNorm}(x) = \gamma \odot \frac{x - \mu}{\sigma} + \beta \tag{3}$$

where μ and σ are the mean and standard deviation of the input vector, and γ, β are learnable affine parameters.

The FFN in each layer is applied identically and independently to each position, transforming representations through two linear layers separated by a non-linear activation:

$$\text{FFN}(x) = \text{Linear}_2(\phi(\text{Linear}_1(x))) = \phi(xW_1 + b_1)W_2 + b_2 \tag{4}$$

Common choices for ϕ include Rectified Linear Unit (ReLU), Gaussian Error Linear Unit (GeLU), and Sigmoid Linear Unit (SiLU), enabling the network to learn rich, token-specific transformations.

2.3 Transformer Acceleration

Recent acceleration efforts for Transformers primarily target MHA, aiming to reduce intermediate memory traffic and enhance compute efficiency. These approaches commonly employ tile-and-fuse strategies to avoid off-chip writes of large intermediate results such as QK^T and softmax outputs.

For instance, FlashAttention [6] and FlashAttention-2 [5] tile the Q, K, and V matrices, stream data from GPU High-Bandwidth Memory (HBM) to on-chip SRAM, compute attention outputs block-wise, and fuse multiple operators to minimize intermediate writes. This leads to both lower memory overhead and better compute throughput on architectures such as NVIDIA A100.

FLAT, targeting TPUs and general spatial accelerators, applies row-wise fusion across the entire attention computation. It processes one row of Q at a time, computes the corresponding QK_T, softmax, and final output entirely on-chip, reducing buffer demands while maintaining full operator fusion.

To improve compute utilization further, FlashAttention-3 introduces pipelining across warp groups using ping-pong scheduling, overlapping MatMul and softmax stages to keep GPU cores busy. Similarly, FuseMax maps fused attention onto spatial arrays using Einsum-based designs, tailored for high-throughput cloud deployments.

2.4 Einsum

Recent advances in tensor algebra introduce Einsum, a formal abstraction that models sequences of dependent tensor operations. Traditionally, Einsum denotes tensor contractions using compact index notation, defining computations over shared indices—for example, matrix multiplication as:

$$Z_{m,n} = \sum_k A_{m,k} \cdot B_{k,n} \tag{5}$$

Traditional Einsums are limited to basic operations. The Extended Einsum abstraction allows user-defined map-and-reduce operations. For instance, the softmax computation over a vector I_m can be expressed as:

$$G = \max_m I_m \tag{6}$$

$$S_m = \exp(I_m - G) \tag{7}$$

$$A_m = \frac{S_m}{\sum_k S_k} \tag{8}$$

where S_m is the exponentiated and shifted score for element m, and A_m is the final normalized attention weight or softmax output for element m.

For sequences of dependent tensor computations, the Cascade of Einsums abstraction represents a series of Einsums where intermediate results feed into subsequent computations. For instance,

$$Y_k = A_k \cdot B_k \tag{9}$$

$$X = A_k \tag{10}$$

$$Z = Y \cdot X \tag{11}$$

represent a typical cascaded pattern from [31], which appear throughout Transformer layers.

To implement such a cascade, FuseMax [31] adopts the 1-pass attention dataflow from FlashAttention-2 [5] shown in Figure 2 and referred to as Einsum Cascade 1. This dataflow has the following steps: (1) partition the key (K) and value (V) tensors into blocks (BK and BV) indexed by m_1 and m_0 where m_1 represents the outer sequence tile and m_0 is the inner sequence tile size; the attention computation is carried out block by block. (2) perform the following computations for each tile m_1: (a) compute the block dot product BQK; (b) compute the local maximum (LM) across the m_0 dimension; (c) update the running max RM as the maximum between the current RM and the new local max LM; (d) subtract the current RM from BQK and exponentiates the result to form the local softmax numerator (SLN); (e) compute the local softmax denominator as the sum of SLN (SLD) along the m_0 dimension; (f) compute a matrix multiplication with BV, producing the result $SLNV$. (3) scale past values RD and RNV using a correction factor PRM to accumulate results across tiles, generating SPD and $SPNV$ respectively to align them with the current numerical base; (4) combine these scaled past values with the local results to update the running sums (RD and RNV). (5) normalize the accumulated numerator-times-V product (RNV) by the accumulated denominator (RD) to produce the final attention output (AV).

$$BQK_{h,m1,m0,p} = Q_{h,e,p} \times BK_{h,e,m1,m0} \tag{12}$$

$$LM_{h,m1,p} = BQK_{h,m1,m0,p} :: \bigvee_{m_0} \max(\cup) \tag{13}$$

$$RM_{h,m1+1,p} = \max(RM_{h,m1,p}, LM_{h,m1,p}) \tag{14}$$

$$SLN_{h,m1,m0,p} = e^{BQK_{h,m1,m0,p} - RM_{h,m1+1,p}} \tag{15}$$

$$SLD_{h,m1,p} = SLN_{h,m1,m0,p} \tag{16}$$

$$SLNV_{h,f,m1,p} = SLN_{h,m1,m0,p} \times BV_{h,f,m1,m0} \tag{17}$$

$$PRM_{h,m1,p} = e^{RM_{h,m1,p} - RM_{h,m1+1,p}} \tag{18}$$

$$SPD_{h,m1,p} = RD_{h,m1,p} \times PRM_{h,m1,p} \tag{19}$$

$$RD_{h,m1+1,p} = SLD_{h,m1,p} + SPD_{h,m1,p} \tag{20}$$

$$SPNV_{h,f,m1,p} = RNV_{h,f,m1,p} \times PRM_{h,m1,p} \tag{21}$$

$$RNV_{h,f,m1+1,p} = SLNV_{h,f,m1,p} + SPNV_{h,f,m1,p} \tag{22}$$

$$AV_{h,f,p} = \frac{RNV_{h,f,M1,p}}{RD_{h,M1,p}} \tag{23}$$

$$\diamond : \quad m1 \equiv M1 + 1 \tag{24}$$

Figure 2: Einsum Cascade 1: 1-pass attention cascade used in FuseMax (from [31]).

The core idea is to stream attention computation in a tile-wise and stateful manner, maintaining a running max and scaled running denominator across m_1 chunks, to compute numerically stable softmax without materializing large intermediates like QK or softmax outputs. Represented as a cascade of Einsums, this formulation maps naturally onto spatial architectures, enabling FuseMax to express attention as a fused, pipelined stream that eliminates intermediate memory writes and supports efficient end-to-end execution on specialized hardware.

3 TransFusion

TransFusion implements end-to-end fusion across full-stack encoder/decoder sub-modules (QKV projection, MHA, Add & LayerNorm, and FFN). As shown in Figure 3, starting with the QKV projection, TransFusion first partitions the input tensor into the outer tiles, which are loaded into on-chip buffers and processed by three separate linear layers to generate the corresponding Q, K and

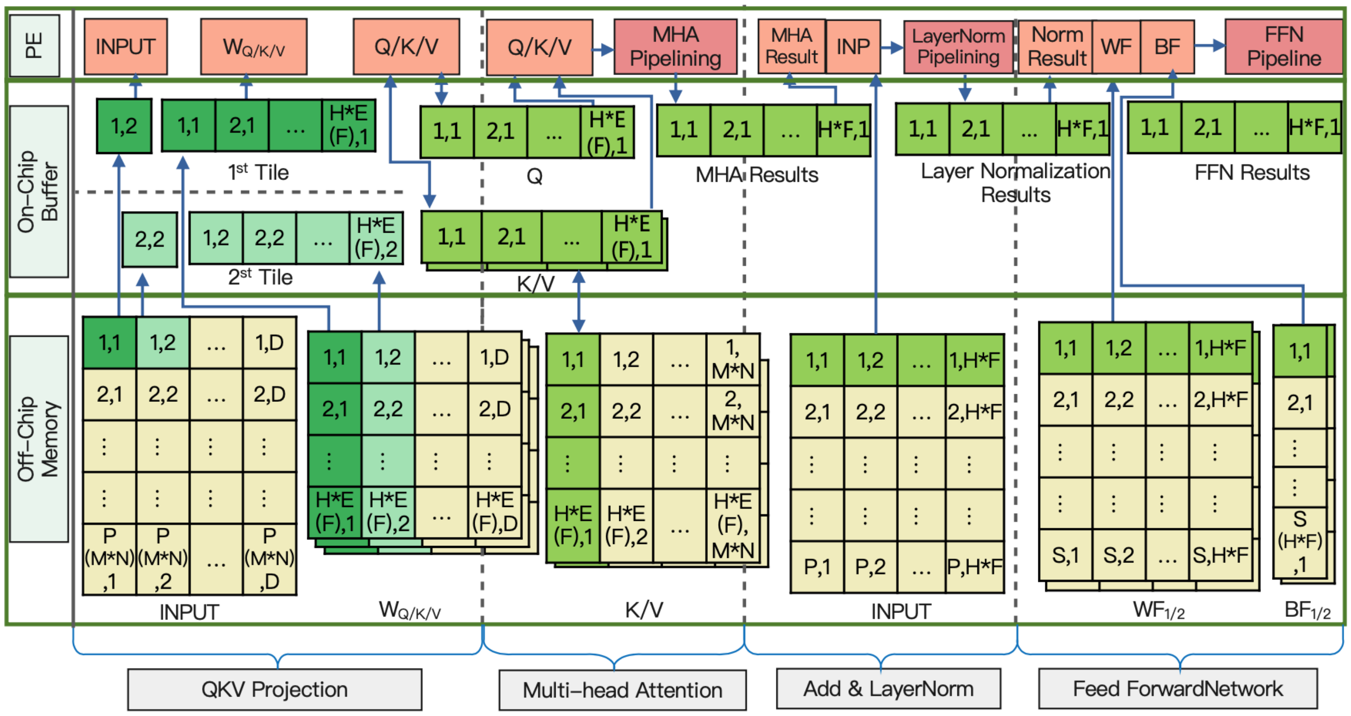


Figure 3: TransFusion Dataflow for a single-token tile ($p = 1$) showing the flow of data across off-chip memory, on-chip buffer, and PE arrays across the QKV projection, MHA, Add & LayerNorm, and FFN. Each layer operates on a tile that preserves the full head and embedding dimensions (H,F). Intermediate results are retained on-chip and forwarded between layers, except for K and V, which are stored in off-chip memory to enable reuse across Q tiles.

$$Q_{h,e,p} = INPUT_{d,p} \times WQ_{d,h,e} \quad (25)$$
$$BK_{h,e,m1,m0} = INPUT_{d,m1,m0} \times WK_{d,h,e} \quad (26)$$
$$BV_{h,f,m1,m0} = INPUT_{d,m1,m0} \times WV_{d,h,f} \quad (27)$$

Figure 4: Einsum Cascade 2: Tiled QKV Projectoions with Shared Input.

V tiles. The K and V tiles are written back to off-chip memory, and cached for reuse by all Q tiles during full-stack computation. TransFusion then propagates each Q tile through the MHA layer using intra-layer pipelining and passes its output to the subsequent layers (Add & Layer and FFN). TransFusion continues the process as intermediate results are forwarded on-chip through each Transformer layer, until producing the final output at the topmost layer.

This section first defines (1) the Einsum Cascades for each layer (QKV, MHA, Add & LayerNorm, FFN), followed by the details of (2) the inter-layer mechanism which enables direct on-chip propagation of intermediate results across each layer, and (3) the intra-layer strategies that achieve efficient computation within each layer through pipelining execution.

3.1 Transformer as Einsum Cascades

The definition of the Einsums cascades for each Transformer layer begins with the QKV projection shown in Einsum Cascade 2. This projection captures the QKV projection phase, where the input activations are linearly projected into Q (Equation 25), K (Equation 26), V (Equation 27) tensor using shared or separate inputs and weights. Here, h denotes the number of attention heads, and e is the per-head embedding dimension. The K and V tensors are partitioned into multi-sequence tiles (BK and BV), following the same layout used in Einsum Cascade 1.

TransFusion then feeds the Q, BK, and BV into the MHA module using the 1-pass MHA execution pattern introduced in Einsum Cascade 1, where TransFusion propagates each Q tile through the whole layer, but with a key difference: while FuseMax loads only a single head per tile into the on-chip buffer, TransFusion fuses and retains the full head dimension to ensure correctness and completeness of the subsequent FFN computation.

Thus, the Add & LayerNorm Einsum Cascade is in Einsum Cascade 3. TranFusion performs element-wise addition with the residual input (Equation 28), followed by normalization (Equation 29-Equation 36) across the head (h) and embedding (f) dimensions at each sequence position (p). TransFusion computes the mean (Equation 30) and variance (Equation 34) per token, and then normalizes the activation (Equation 36). The scaling (γ) and shifting (β) follow the design of Li et al. [23] by deferring and fusing them into the subsequent layer.

Finally, We define the FFN Einsum Cascade in the Einsum Cascade 4. TransFusion computes a complete sub-block of FFN1 for each tile (Equation 37), which is immediately passed through the activation function in a pipeline manner (Equation 38). TransFusion

$$IAV_{h,f,p} = INP_{h,f,p} + AV_{h,f,p} \tag{28}$$
$$SAV_p = IAV_{h,f,p} \tag{29}$$
$$MAV_p = \frac{1}{H \times F} \times SAV_p \tag{30}$$
$$DAV_{h,f,p} = IAV_{h,f,p} - MAV_{h,f,p} \tag{31}$$
$$QAV_{h,f,p} = DAV_{h,f,p} \times DAV_{h,f,p} \tag{32}$$
$$SQAV_p = QAV_{h,f,p} \tag{33}$$
$$MQAV_p = \frac{1}{H \times F} \times SQAV_p \tag{34}$$
$$SR_p = \frac{1}{\sqrt{MQAV_p}} \tag{35}$$
$$NR_{h,f,p} = DAV_{h,f,p} \times SR_p \tag{36}$$

Figure 5: Einsum Cascade 3: Add & LayerNorm Layer.

$$FFN1_{s,p} = NR_{h,f,p} \times WF1_{h,f,s} + BF1_s \tag{37}$$
$$AR_{s,p} = Actication(FFN1_{s,p}) \tag{38}$$
$$FFN2_{h,f,p} = FFN1_{s,p} \times WF2_{h,f,s} + BF2_{h,f} \tag{39}$$

Figure 6: Einsum Cascade 4: Feed Forward Network Layer.

then consumes the resulting activated tile to compute an incomplete fragment of FFN2 via the second matrix multiplication (Equation 39), with buffering the partial results on-chip and awaiting accumulation with subsequent tiles.

For clarity, this presentation omits the batch dimension (b) in all Einsum cascades because it does not affect the core computation patterns. Section 5 will revisit batch size (b) tiling and its impact on performance.

3.2 Inter-layer Fusion: On-chip Intermediate Propagation

TransFusion implements inter-layer fusion by retaining intermediate activations on-chip and directly forwarding them between layers. This section describes how TransFusion enables end-to-end propagation across the full encoder/decoder stack, including QKV projection, MHA, Add $ LayerNorm and FFN as follows:

QKV Projection. TransFusion partitions both the input sequences ($INPUT$) and projection weights (WQ, WK, WV) along sequence dimension (p) and hidden dimension (d), generating the outer tiles processing for each input segment. For each outer tile, TransFusion iterates over the full hidden dimension (d) to compute the Q, BK, and BV via three separate linear transformations. The resulting Q, K, and V tensors serve as inputs to the MHA module. To preserve correctness in downstream computation (particularly in the FFN), each outer tile fully retains the head and embedding (H, E, F) on-chip. Among the projected outputs, BK and BV tiles are written back to off-chip memory for reuse across all Q tiles during attention computation in the full-stack encoder/decoder pipelining.

MHA. TransFusion operates MHA and the subsequent layers in a Q-tile-wise execution pattern: TransFusion propagates each Q tile through each Transformer layer to generate its final output before processing the next Q tile. Within MHA, TransFusion feeds the current Q tile over the entire set of K and V tiles (from off-chip memory as needed). Thus, TransFusion must iterate over the m_1 dimension to accumulate the complete attention result. Therefore, it performs a localized inter-layer fusion between QKV and MHA layers: along the m_1 axis, TransFusion jointly computes the accumulated numerator-times-V product (RNV) and accumulated denominator (RD), allowing the attention result (AR, tensor shape $[B, H, F, P]$) to be computed on-chip and forwarded immediately to the next layer.

Add & LayerNorm and FFN. TransFusion adopts a shape-consistent fusion pattern to connect Add & LayerNorm and FFN with MHA, leveraging their identical input/output tensor shape $[B, H, F, P]$ to enable seamless data forwarding. TransFusion composes and reorders Add & LayerNorm, FFN and MHA by their uniform input/output tensor shape, supporting different model structures such as encoders, decoders, or hybrid configurations.

3.3 Intra-layer Fusion: On-chip Tile Execution within Layers

TransFusion implements intra-layer fusion by partitioning the on-chip buffer into smaller inner tiles that match the size of the underlying 2D PE array. TransFusion executes each inner tile as a self-contained unit, running through the entire Cascades of Einsum for each layer. TransFusion schedules the inner tiles in a pipelining manner across the PE array, allowing overlapping execution across multiple inner tiles. While the pipelining scheduling strategy is detailed in Section 4, this section focuses on how the inner tiles are formed based on the Einsum structure of each layer.

TransFusion determines tiling boundaries by mapping shared Einsum dimensions, typically the sequence length (p, m_0), head count (h), and embedding size (e, f), onto the 2D PE array, as shown in Table 1. When operating on a 1D PE array, TransFusion retains the row-based mapping along the sequence dimension (p, m_0), and when additional compute resources are available, further unfolds computation along dimensions originally assigned to 2D PE columns.

Table 1: Dimension mapping of each Transformer layer onto the 2D PE array.

Layer	2D PE Row	2D PE Column
QKV	p/m_0	h, e
MHA	p	m_0
LayerNorm	p	h, f
FFN	p	s

The tiling strategy used in each layer works as follows:

QKV. TransFusion treats the Einsums for Q (Equation 25), BK (Equation 26) and BV (Equation 27) as independent, dependency-free computations. Thus, TransFusion maps each Einsum to different dimension assignments over the 2D PE array. For Q, TransFusion maps the sequence dimension (p) to PE rows, and (h,e) to columns. For BK and BV, TransFusion maps the sequence's inner

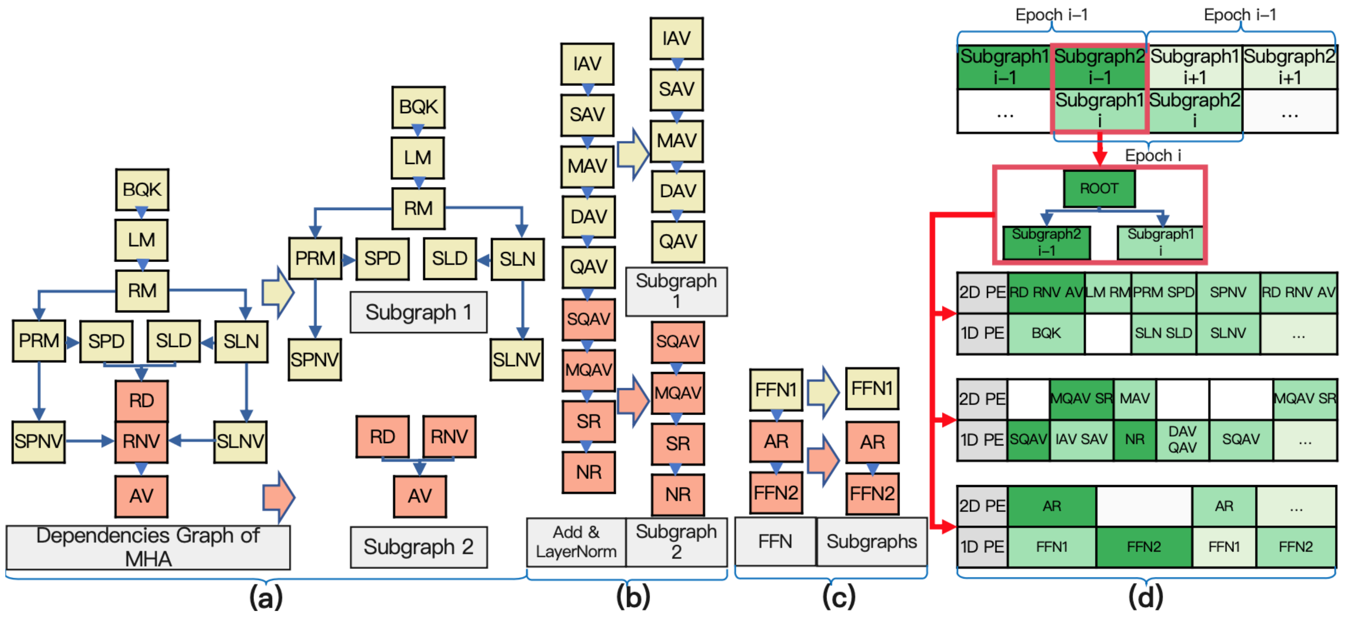


Figure 7: DPipe Pipelining Scheduling overview showing the computation DAGs and one valid bipartition for Transformer sublayers (a) MHA, (b) Add & LayerNorm, and (c) FFN. (d) illustrates how DPipe constructs a pipelined execution from a given bipartition. It first overlaps the execution of Einsums across epochs by interleaving subgraph schedules. DPipe then adds a virtual root node (*ROOT*) to connect the overlapping subgraphs and form a new DAG. Finally, DPipe generates latency-aware pipelining schedules using DP strategies.

sequence dimension (m_0) to rows, and (h, e) and (h, f) respectively to columns.

MHA. TransFusion maps the sequence (p) to PE rows and the inner sequence dimension (m_0) to PE columns. During intra-layer pipelining, TransFusion partitions the attention computation into multiple head-specific tiles. When the resulting 2D tile from PE mapping does not fully utilize the available 1D PE array, TransFusion packs multiple head tiles into a single pipeline pass to increase computational efficiency across the PE array.

Add & LayerNorm. TransFusion distributes the sequence dimension p across 2D PE rows, assigning each row to one token. Then, TransFusion flattens the head (h) and embedding (f), distributing them across 2D PE columns with each inner tile processing an entire feature vector for the given token. TransFusion computes the full normalization over ($H \times F$), within each tile, for one token position p.

FFN. TransFusion maps the sequence length (p) across PE rows, while the hidden (s) dimension is mapped across PE columns. During execution, TransFusion defines a inner tile by a slice over the s and p based on 2D PE array size. TransFusion uses each inner tile to compute a sub-block of *FFN1* (Equation 37), and immediately passes it through the activation function (Equation 38) in a pipelining manner. Then, TransFusion consumes the resulting activated inner tile to compute an incomplete fragment of FFN2 via Equation 39, with the partial results being buffered on-chip and awaiting accumulation with subsequent tiles.

4 DPipe: an Einsum Pipelining Scheduler via DAG Traversal and Dynamic Programming

This section describes DPipe, a DAG-based, Einsum-centric pipelining scheduler that uses DP to optimize intra-layer inner tile execution. Instead of relying on heuristic or static scheduling [40] [41] [31], DPipe introduces flexible adaptation to diverse hardware and model configurations, forming latency-aware and overlapped pipelining strategies for optimal execution.

DPipe, depicted in Figure 7, models fused Transformer computations as a computation DAG where nodes represent an inner tile of Einsum operations and edges encode data dependencies. By enumerating valid DAG partitions and their corresponding execution orders, DPipe identifies pipelined schedules that enable overlapped execution across sub-graphs. DPipe applies a DP-based cost model to estimate the latency of each schedule, taking into account operation dependencies, tiling granularity, and PE-level parallelism. By combining this graphical representation with performance analysis, DPipe provides a principled solution for scheduling fused Einsum layers onto parallel hardware architectures.

This section first introduces (1) the DAG partitioning and pipelining execution strategy, next describes (2) a latency estimation method for each Einsum operation, and finally presents our (3) DP-based cost model that schedules the pipelining with awareness of dependencies and resource utilization.

4.1 DAG Partitioning and Pipelining Execution Strategy

DPipe first constructs the computation DAG from the Einsum Cascades introduced in Section 3.1, capturing operation-level dependencies. And then, DPipe partitions the DAG into two weakly connected subgraphs, subject to the following constraints:

(1) **Source-Sink Alignment:** The source nodes (i.e., nodes with zero in degree) must belong to the first subgraph, and the sink nodes (i.e., with zero out-degree) must belong to the second subgraph.
(2) **Weak Connectivity:** The partitioned subgraphs must remain weakly connected within the original DAG structure.
(3) **Dependency Completeness:** The first subgraph must be dependency-complete: all its input dependencies must be contained within the subgraph.
(4) **Reachability:** All nodes in the first sub-graph must remain reachable from the DAG's source nodes after partitioning.

Next, DPipe enumerates all valid bipartitions satisfying these constraints. For each partition, DPipe constructs a pipelined execution model by applying intra-layer tiling to subdivide the workload of each subgraph into multiple computational blocks. Each block processes the full sequence of Einsum operations across the two subgraphs, effectively creating an overlapped execution pipeline across epochs. For each valid bipartition, DPipe introduces a virtual root node to connect the source nodes of the two subgraphs. DPipe then enumerates all valid topological orderings that respect the new dependencies. Each ordering defines a schedule interleaving Einsum operations from both subgraphs. DPipe evaluates each candidate schedule by constructing a pipelined execution model that applies intra-layer tiling — based on Table 1 — to partition the workload into inner tiles along the sequence (p, m_0), head (h) and embedding (e, f) dimensions mapped onto the PE array. These inner tiles form the pipeline's execution units (epochs), with each epoch executing the full sequence of Einsum operations in the prescribed topological order. As the inner tiles traverse the fused computation graph, intermediate results between subgraphs mapped to different PE arrays (2D↔1D) are staged in the on-chip buffer. This strategy decouples producer-consumer timing, allowing the next tiles to begin subgraph-1 while the previous tile proceeds in subgraph-2, forming a temporally overlapped pipeline across PEs. The on-chip buffer enables smooth handoff and sustained parallelism.

4.2 Latency Estimation for Einsum Operations

This section describes how DPipe computes latency estimation for each Einsum, which serves as input to the DP-based scheduling cost model. An Einsum map operation is expressed in the form: *einsum*(*InputIndices* $\rightarrow$ *OutputIndices*), where *InputIndices* denotes a comma-separated list of index labels corresponding to each input tensor, and *OutputIndices* specifies the index labels of the resulting output tensor. For example, matrix multiplication $A \in \mathbb{R}^{m\times k}, B \in \mathbb{R}^{k\times n} \rightarrow C \in \mathbb{R}^{m\times n}$ corresponds to $einsum(mk, kn \rightarrow mn)$.

The reduction dimensions (k) are the set of index labels present in multiple inputs but not in the output, and the output dimensions (m, n) are those appearing in the output index. The estimated compute latency of scalar arithmetic operations required is:

$$\text{ComputeLoad}_{op} = \left(\prod_{d\in\text{OutputDims}} d\right)\cdot\left(\prod_{d\in\text{ReductionDims}} d\right) \tag{40}$$

$$\text{ComputeCycles}_{op} = \frac{\text{ComputeLoad}_{op}}{\text{NumPEs}_{op}} \tag{41}$$

$$\text{Latency}_{op} = \frac{\text{ComputeCycles}_{op}}{f_{\text{clk}}} \tag{42}$$

where d denotes the extent of the dimension, *OutputDims* is the output dimensions, *ReductionDims* refers to the reduction dimensions, $NumPEs_{op}$ is the number of PEs assigned to the operation, and f_{clk} is the clock frequency of the processing elements. The metric captures the full computation complexity and the compute latency of the Einsum.

DPipe estimates latency by modelling Einsum compute complexity as the product of output dimensions and reduction dimensions (Equation 40), scaled by PE count and clock frequency (Equation 41,Equation 42). This provides an accurate latency prediction based on arithmetic intensity and parallelism, under compute-bound conditions.

4.3 Latency-Aware Scheduling via DP

Next, DPipe applies a DP strategy to generate the optimal pipeline schedule for each candidate's topological ordering. This algorithm computes the earliest feasible start time for each Einsum under resource and dependency constraints, aiming to minimize total completion time. The scheduler follows the update rules below:

$$\text{StartT}[op_i][pe_j] = \max\left(\text{Time}[pe_j], \max_{op_k\rightarrow op_i}\text{EndT}[op_k]\right) \tag{43}$$

$$\text{EndT}^{PE}[op_i][pe_j] = \text{StartT}[op_i][pe_j] + \text{Latency}[op_i][pe_j] \tag{44}$$

$$\text{EndT}[op_i] = \min_{pe_j\in[1d,2d]}\left(\text{EndT}^{PE}[op_i][pe_j]\right) \tag{45}$$

$$\text{Time}\left[\arg\min_{pe_j}\left(\text{EndT}^{PE}[op_i][pe_j]\right)\right] = \text{EndT}[op_i] \tag{46}$$

Here op_i denotes the i-th Einsum operation in the topologically sorted computation graph, and pe_j is the j-th processing element in a 1D and 2D PE array. $Time[pe_j]$ tracks the total elapsed time that pe_j has already been occupied by previously assigned einsums.

Equation 43 computes the start time of op_i on pe_j by taking the maximum of (a) the current cumulative workload on pe_j, and (b) the latest completion time among all its direct dependencies op_k. Equation 44 computes the completion time of op_j by adding the known latency. Equation 45 determines the best PE assignment by selecting the one yielding the earliest completion. Finally, Equation 46 updates the selected PE's timeline to reflect the scheduled operation.

The DP formulation ensures that each operation respects both dependency constraints and hardware-level parallelism. It aims to minimize the critical path while distributing the workload evenly across available compute units.

In summary, our DAG-based pipelining scheduler is tailored for Einsum-centric Transformer layers, and systematically explores the space of valid subgraph partitions and their corresponding topological schedules, leveraging DP to compute latency-aware, resource-constrained execution plans. This approach enables fine-grained intra-module parallelism and balanced PE utilization, forming the foundation for efficient hardware mapping of Einsum-based Transformer layers.

5 TileSeek: an Outer Tiling Search Algorithm

This section introduces TileSeek, an outer tiling search algorithm used to optimize data movement from off-chip memory to on-chip buffer, to reduce energy consumption and off-chip memory traffic. TileSeek focuses on determining tiling factors that ensure each outer tile can support the complete computation of a Transformer layer during end-to-end fusion. This process does not cover the on-chip buffer to PE/register file level because the corresponding inner tiling and pipelining strategies have already been discussed in Section 3.3 and Section 4. To accommodate the full Transformer stack, TileSeek applies fine-grained outer tiling over the dimensions $[B, D, M_1, P, S]$, where each dimension is mapped according to the on-chip constraints to balance the workload and reduce traffic.

This section first introduces (1) the detailed implementation of the TileSeek based on MCTS, and then analyzes (2) the on-chip buffer requirements for executing each fused layer tile.

5.1 MCTS-based Exploration Framework

TileSeek adopts a search strategy leveraging MCTS for outer tiling exploration. TileSeek defines each node in the search tree corresponds to a decision along a specific outer tiling factor. TileSeek maps each complete traversal from the root to a leaf node to a full outer tiling configuration, specifying how data blocks are partitioned and transferred from off-chip memory to on-chip buffer. The MCTS framework in TileSeek has the following key components:

- **Node:** Each Node encodes a partial tiling decision. Collectively, nodes along a path represent an outer tiling strategy applied to the Einsum.
- **Selection:** Select child nodes based on the Upper Confidence Bound (UCB) criterion during traversal, balancing exploration of less-visited nodes and exploitation of high-performing subtrees.
- **Constraint Validation:** TileSeek validates the tiling factors against the hardware constraints of the target accelerator, including memory capacity and bandwidth limitations. Section 5.2 will discuss the buffer constraints analysis.
- **Simulation (Evaluation):** Evaluate the leaf tilling factors using Timeloop and Accelergy [34] [51], which estimates the energy consumption and latency for executing the tiled computation. The resulting energy or latency can serve as the reward signal for MCTS.
- **Backpropagation:** TileSeek propagates the estimated energy score back through the nodes along the selected path, updating their statistics to inform future UCB-based selection decisions.

5.2 On-chip Buffer Requirements

This section analyzes the on-chip buffer requirements associated with each intra-layer computation. Since our fusion strategy executes a complete tile per layer, the on-chip buffer must be provisioned to hold not only the layer input and output activations, but also any intermediate state required for pipelined execution with the layer.

Table 2 summarizes the buffer requirements of key Transformer components (QKV, MHA, Add&LayerNorm, and FFN).

Table 2: Buffer requirements per tile for different intra-layer modules.

Layer	Buffer Req.
QKV Projection	$BD(4P + 3M_1M_0) + 3DHE + 2BHP$
MHA	$BHE(P + 2M_1M_0) + BHP(2 + 2F)$ $+4M_0P' + 18P'$
Add & LayerNorm	$3BHFP + 4HFP'$
FFN	$HF(2BP + S) + S(P + 2) + 2SP'$

In the Table 2, let B denote the batch size per tile, D the model dimension, P the sequence length, and M_1, M_0 the hierarchical splits of the sequence introduced in the MHA computation. H is the number of attention heads, while E and F represent the key/query and value embedding dimensions, where $E = F$ and $D = H \times E = H \times F$. S denotes the hidden size in the FFN, and the P' corresponds to the intra-tile sequence length processed per PE row.

A layer's input and output activations must be fully buffered on-chip across all modules. In the QKV projection, the inputs include:

- $INP^{[B,D,P]}$ and $INP^{[B,D,M_1,M_0]}$,
- Weight metrices $W_Q^{[D,H,E]}$, $W_K^{[D,H,E]}$, $W_V^{[D,H,F]}$,

and the outputs are:

- $Q^{[B,H,E,P]}$, $BK^{[B,H,E,M_1,M_0]}$, $BV^{[B,H,F,M_1,M_0]}$,

MHA and QKV are localized and fused, TransFusion requires additional buffers to store MHA intermediate states required across M_1-loop iterations:

- $RM^{[B,H,P]}$, $RNV^{[B,H,F,P]}$, $RD^{[B,H,P]}$, $AV^{[B,H,F,P]}$,

The MHA module takes:

- $Q^{[B,H,E,P]}$, $BK^{[B,H,E,M_1,M_0]}$, $BV^{[B,H,F,M_1,M_0]}$,

as inputs and outputs:

- $AV^{[B,H,F,P]}$,

while also maintaining the recurrent state:

- $RM^{[B,H,P]}$, $RNV^{[B,H,F,P]}$, and $RD^{[B,H,P]}$

To support intra-layer pipelineing, each Einsum kernel requires dedicated staging buffers. For large Einsums such as BQK and SLN, a full $M_0 \times P'$ tile must be buffered; for others, a single P'-length slice is sufficient.

For the Add & LayerNorm layer, both inputs $AV^{[B,H,F,P]}$ and $INP^{[B,H,F,P]}$ must reside in buffer, along side the output $NR^{[B,H,F,P]}$. Additionally, only intermediate results like IAV and DAV are reused across non-consecutive Einsums, requiring a small number (typically two) of block-level cache buffers sized $H \times F \times P'$, maintained via double-buffering.

In the FFN layer, the inputs are $NR^{[B,H,F,P]}$, weights $WF^{[F,H,S]}$, and bias$BF^{[S]}$, producing the output $FFN2^{[B,H,F,P]}$. For pipelining, each stage in the FFN requires buffer space of size $S \times P'$, also maintained in double-buffered form to support overlapped execution.

This modelling of buffer-sensitive requirements for all intra-layer components enables the incorporation of hardware constraints into the outer tiling feasibility evaluation. Each candidate tiling configuration must satisfy on-chip buffer capacity constraints, accounting for input/output activations, intermediate recurrent states, and pipeline staging buffers. These constraints directly prune the search space and ensure that only implementable configurations are passed to the performance evaluation stage.

TileSeek is a unified tiling search framework that jointly explores buffer-sensitive fusion opportunities and hardware-constrained tensor partitioning. By integrating fine-grained buffer modelling, outer tiling strategies, and joint exploration via MCTS, TileSeek enables high-throughput Transformer execution under strict memory budgets.

6 Performance Impact of TransFusion

This section evaluates TransFusion's ability to deliver improved Transformer fusion across full-stack encoder-decoder workloads. This experimental evaluation aims to answer the following key questions: (1) Does TransFusion improve latency and energy efficiency compared to state-of-the-art baselines such as FLAT and FuseMax? (2) Does TransFusion preserve its performance advantage under varying compute resources (e.g., different PE sizes)? (3) What are the primary contributors to speedup in TransFusion across different sequence lengths? (4) How does DPipe impact entire Transformer modules, including QKV, MHA, FFN, and LayerNorm? (5) What are the underlying factors influencing energy efficiency at different hardware and sequence lengths ?

6.1 Architectures, Modeling Tools, and Workloads

This performance study evaluates TransFusion across two architectural models representing cloud and edge environments. The cloud-setting evaluation adopts the TPU v2/v3 [32] accelerator model used by Kao et al. and Nayak et al [18] [31]. This model maps attention computation onto a spatial architecture consisting of 2D and 1D PE arrays. Specifically, it features a 256 × 256 2D spatial array, a 256-element 1D PE array, 16MB of on-chip buffer, and a DRAM bandwidth of 400 GB/s. The edge architecture is based on the edge DNN accelerator [58], representing a resource-constrained design. It includes a 16 × 16 2D PE array, the same 256-element 1D PE array, 5MB of on-chip buffer, and 30 GB/s of DRAM bandwidth. Figure 1 and Table 3 summarize these architectural parameters.

Table 3: Architecture Specification in Evaluation.

Name	2D PE size	1D PE size	On-chip Mem. Size	DRAM BW.
Cloud	256 × 256	256	16MB	400GB/s
Edge	16 × 16	256	5MB	30GB/s

Simulation and Modeling Tools: The performance estimation of TransFusion uses the Timeloop [34] and Accelergy [51] simulation frameworks for latency and energy prediction. We construct architectural models of the accelerator at the 45nm technology node to enable the evaluation of each Einsum operation in isolation. This study integrates the individual Einsum results using heuristic methods introduced by Nayak et al. [30] [31] to model full Transformer execution. These heuristics overlap Einsum executions following the DPipe strategy to compute end-to-end latency. Accelergy estimates overall energy consumption by aggregating compute and memory access statistics across all Einsums.

Workloads: This evaluation covers a diverse set of Transformer models, including BERT-Base [8] (BERT), TrXL-wt103 [4] (TrXL), T5-small [39] (T5), XLM [19], adopted from the benchmarks used in FLAT and FuseMax, along with Llama3-8B [11] (Llama3). Following the setup in FLAT and FuseMax, all experiments use a fixed batch size of $B = 64$.

Unfused: The unfused baseline is modelled by sequentially executing QKV projections, MHA, Add & LayerNorm, and FFN, with intermediate results written to off-chip memory between phases. QKV projections are computed on the 2D PE array, followed by QK^T on the 2D array and full softmax on the 1D array. The resulting attention weights are multiplied with V using the 2D array, and Add & LayerNorm are performed on the 1D array. In the FFN, linear layers run on the 2D, while activations are handled by the 1D.

FLAT: FLAT applies local fusion to the attention layer in a tiled, sequential fashion. For each Q tile, QK^T, softmax normalization, and the weighted sum with V are computed on-chip with outputs written back to off-chip memory. Other layers (e.g., QKV projection, Add & LayerNorm, FFN) remain unfused and execute sequentially with standard memory access.

FuseMax: The main baseline of my paper is FuseMax. FuseMax adopts a fully fused design for MHA, structured as a sequence of 12 primitive Einsum operators (detailed in Einsum Cascade 1). Attention scores are computed via QK^T and normalized using a multi-stage softmax, where 2D and 1D PE arrays operate in a pipelined and partially parallel fashion. The softmax and the weighted sum with V are fused into a single pass pipeline, eliminating intermediate memory writes. The rest of the end-to-end execution, including QKV projection, Add & LayerNorm, and FFN, follows the same unfused flow as FLAT.

FuseMax+LayerFuse: As an ablation study, we extend FuseMax by applying inter-layer fusion across QKV projection, MHA, Add & LayerNorm, and FFN, forming an end-to-end fuse design. We executed all layers within the same on-chip computation flow, following the method in Section 3.2. This variant, however, does not incorporate DPipe and executes layers sequentially without pipeline-level overlap, except for the original intra-attention pipeline in FuseMax.

Speedup Contribution: We decompose the speedup gain using a weighted attribution scheme to analyze which components contribute most to the overall speedup. For each layer i (e.g., QKV, MHA, Add & LayerNorm, FFN), we first define the speedup:

$$\mathrm{S}_i = \frac{T^i_{\text{baseline}}}{T^i_{\text{TransFusion}}} \tag{47}$$

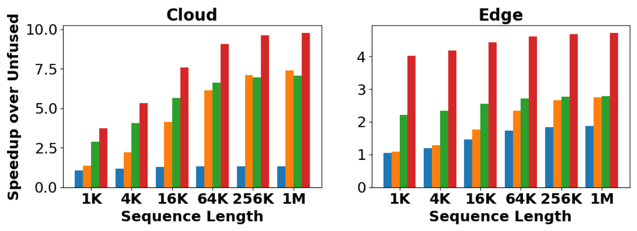
where T^i_{baseline} and $T^i_{\text{TransFusion}}$ denote the execution time of component layer i under the baseline and TransFusion respectively.

The normalized proportional speedup (speedup contribution) of each component i is given by:

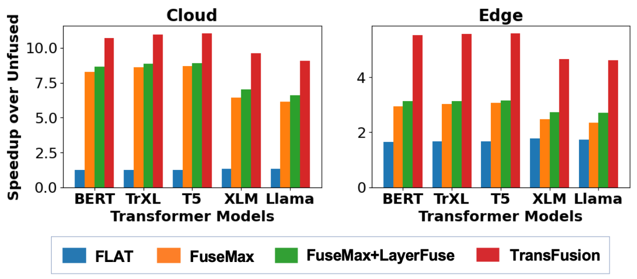
$$\text{SpeedupContribution}_i = \frac{S_i \cdot T^i_{baseline}}{\sum_j S_j \cdot T^j_{baseline}} \tag{48}$$

6.2 Evaluating TransFusion

The experimental results indicate that all the models studied exhibit similar trends across sequence lengths. Thus, for conciseness, this section presents (1) the scaling results for Llama3 in Figure 8a, Figure 9a, Figure 10a, Figure 11, Figure 12a and Figure 13; and (2) a cross-model comparison for 64K sequences in Figure 8b, Figure 9b, Figure 10b, and Figure 12b highlighting the robustness of TransFusion.



(a) Llama3: Speedup over Unfused across sequence lengths (1K-1M) on cloud and edge architecture.

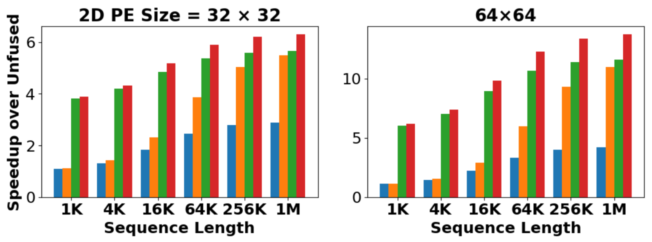


(b) Model-wise speedup comparision (BERT, TrXL, T5, XLM, Llama3) at *64K* sequence length under the same hardware.

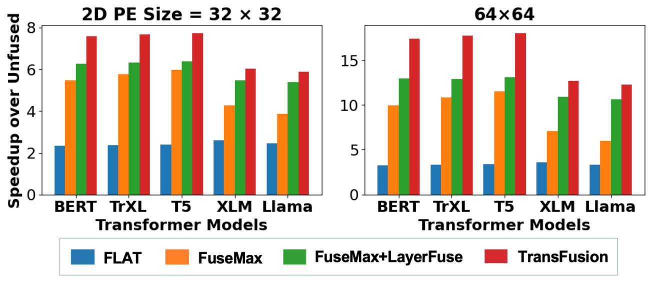
Figure 8: Speedup over Unfused of end-to-end Transformer acceleration across sequence and models. (a) shows the scalability on Llama3, while (b) benchmarks multiple models at a 64K sequence length.

Speedup. Answering (1), TransFusion achieves a geometric mean speedup of 1.3× over FuseMax with layer fusion, 1.6× over FuseMax, and 7.0× over FLAT (Figure 8). A similar trend is observed in the edge architecture, where TransFusion achieves a geometric mean speedup of 1.8× over FuseMax with layer fusion, 2.2× over FuseMax, and 3.2× over FLAT.

Addressing (3), adding layer fusion to FuseMax offers the most benefit over FuseMax at 1K (up to 2.1× on both edge and cloud) — green bars in Figure 8a. However, its benefit diminishes as sequence length increases, resulting in negligible gains for large sequences. For short sequences, the processing is memory-bound, and reducing on-chip memory accesses via fusion effectively improves performance. As sequence length grows, the processing is dominated by computation, limiting the impact of layer fusion.



(a) Llama3: Speedup over Unfused across sequence lengths (1K-1M) on edge architecture with 2D PE sizes of 32×32 and 64×64.



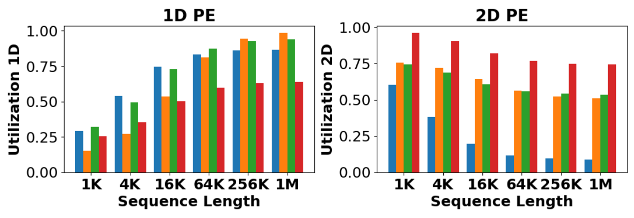
(b) Model-wise speedup comparision (BERT, TrXL, T5, XLM, Llama3) at *64K* sequence length under 2D PE sizes of 32×32 and 64×64.

Figure 9: Impact of 2D PE size on end-to-end Transformer acceleration. (a) reports Llama3 scalability from 1K-1M sequences under 32×32 and 64×64 PEs, while (b) compares multiple models at 64K sequence length across the same PE configurations.

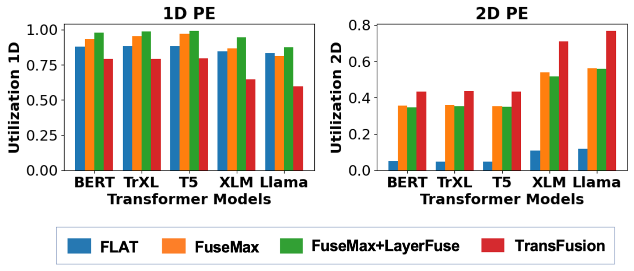
In summary, layer fusion reduces data movement in memory-bound scenarios (e.g., short sequences), and pipelining (DPipe) improves PE utilization in compute-bound cases (e.g., long sequences).

Generalization across Computational Capability. An evaluation of TransFusion on the edge architecture under two additional different 2D PE sizes addresses (2). For the 32×32 configuration, TransFusion achieves up to 1.1× speedup over FuseMax with layer fusion, 1.8× over FuseMax, and 3.0× over FLAT. For the larger 64×64 configuration, where the on-chip buffer size increases to 8MB, TransFusion still delivers strong performance gains, achieving up to 1.2×, 2.4×, 4.8× speedup over the same baseline. These performance gains across different compute capacities and memory budgets indicate that Transfusion is robust and adaptable to variations in hardware resource configurations.

Utilization. Under the cloud architecture, DPipe achieves relatively higher 2D PE utilization (averaging 58%, 1.3× more than FuseMax with layer fusion, 1.2× over FuseMax, 5.7× over FLAT), with an acceptable trade off 1D utilization (Figure 10). This improvement stems from DPipe's ability to offload a portion of the 1D operations (such as LayerNorm and FFN activation operators) onto the 2D, alleviating the bottleneck on 1D and balancing the overall load. In non-pipelined baselines, 2D PEs often remain idle while waiting for dependent 1D to finish, leading to substantial resource underutilization. A mirrored pattern occurs on the edge, where DPipe prioritizes 1D PE utilization (averaging 82%) by shifting more workload to 1D arrays to match the resource balance on edge devices.



(a) Llama3: PE array utilization across sequence lenghts (1K-1M) on cloud architecture



(b) Utilization comparison (BERT, TrXL, T5, XLM, Llama3) at 64K sequence length on cloud architecture.

Figure 10: Utilization of 1D and 2D PE arrays in end-to-end Transformer execution across sequence lengths and model types.

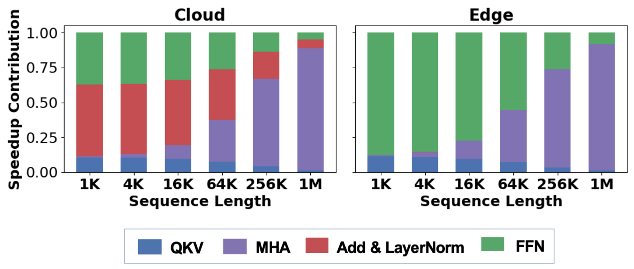
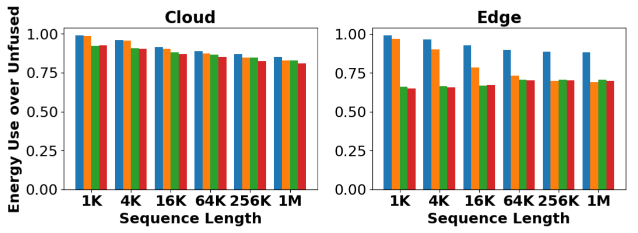


Figure 11: Speedup contribution breakdown for each layers (QKV, MHA, Add & LayerNorm, FFN) of TransFusion over FuseMax on Llama3 across sequecen lengths (1K-1M) under both cloud and edge architecture.

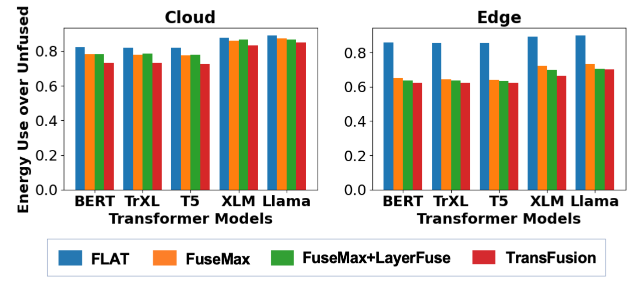
Layer-wise Speedup Contribution. Addressing $\textcircled{4}$, using the speed contribution method described in Section 6.1, the results in Figure 11 confirm the earlier observation that for short sequences (below 256K, memory-bound), TransFusion primarily accelerates LayerNorm and FFN by efficiently fusing full stack and reducing data movement on both cloud and edge architecture. When the bottleneck shifts to the quadratic complexity of the MHA layer for longer sequences, the performance improvement is primarily driven by DPipe's optimized pipeline schedules.

Energy Breakdown. The results in Figure 12 indicate that the faster TransFusion also consumes less energy. To answer $\textcircled{5}$, Figure 13 breaks down energy consumption by component (off-chip memory, global buffer, register file, and PE computation). In the cloud architecture, computation in the PE arrays consumes most of the energy because, given the large on-chip buffer (16MB) and high bandwidth (400 GB/s), the architecture enables more aggressive tiling and high data reuse, leaving fewer opportunities for further energy reduction through fusion.

In contrast, the edge architecture's limited on-chip buffer capacity and lower bandwidth result in smaller tile sizes and more frequent off-chip accesses, amplifying the energy cost of data movement. For short sequences in the edge architecture, up to 25% of the

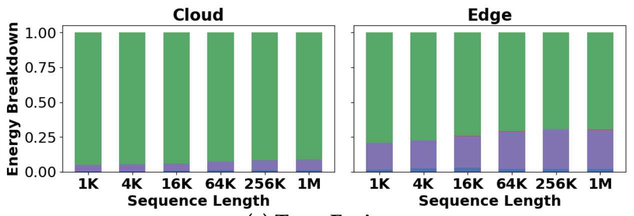


(a) Llama3: Energy consumption over Unfused across sequence lengths (1K-1M) on cloud and edge architecture.

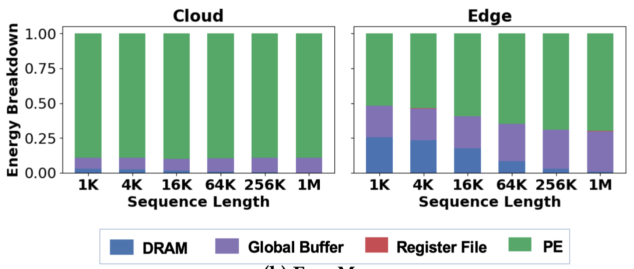


(b) Model-wise energy consumption (BERT, TrXL, T5, XLM, Llama3) at 64K sequence length under the same hardware.

Figure 12: Energy consumption over Unfused of end-to-end Transformer acceleration across sequence and models.



(a) TransFusion



(b) FuseMax

Figure 13: Energy breakdown across memory hierarchy (DRAM: off-chip memory, Global Buffer: on-chip buffer, Register File, PE arrays) for end-to-end Transformer using TransFusion and FuseMax on the Llama3 model under both cloud and edge architecture.

energy used by FuseMax is spent in DRAM (see Figure 13b), indicating opportunities for further optimization. For shorter sequence lengths (below 64K), TransFusion significantly reduces energy consumption by improving data reuse via fusion and tiling strategies, reducing redundant memory accesses. However, as sequence length increases (above 64K), the workload becomes compute-bound, reducing the effectiveness of these memory-centric optimizations.

7 Related Work

Approximate Acceleration Algorithm. Palletization [2, 43, 48], quantization [7, 9, 12, 24–28, 36, 37, 47, 52, 53, 57], pruning [15, 29, 35, 46, 54, 55], and knowledge distillation [10, 13, 17, 42, 49, 50] compress model size by reducing weight precision, removing redundant parameters, or transferring knowledge from larger models, thereby improving deployment efficiency, inference speed, and energy efficiency on resource-constrained hardware. A^3 [14] proposes approximate attention by a sparse content-based search, selecting likely relevant keys through preprocessing and computing only a subset of scores during inference. FalshDecoding++ [16] eliminates softmax synchronization via unified max value approximation. However, these methods often come at the cost of degraded model quality due to reduced numerical precision or structural simplification. TransFusion preserves original computation semantics through our fusion strategies, making it a high-quality and deployment-friendly optimization solution without compromising model accuracy.

Exact Attention Acceleration Methods. Exact accelerator on Transformer primarily focus on operation fusion [5, 6, 18, 21] and pipelining [31, 40, 41, 58] to reduce memory traffic and improve compute utilization. Early works like FLAT, and FlashAttention-1/2 adopt fine-grained tiling to enable efficient on-chip fusion and reduce data movement. Later methods, such as FlashAttention-3 and MAS-Attention, improve utilization by overlapping GEMM and softmax on specific NVIDIA A100 GPU and edge architectures. TileFlow generalizes pipelined execution via producer-consumer tiling with shared resources. However, these approaches are limited by the lack of end-to-end Transformer fusion and rely on static pipelining strategies. TransFusion enables end-to-end Transformer fusion with a flexible tile-level pipelining strategy that adapts to computation patterns and hardware constraints.

8 Conclusion

This paper proposes TransFusion, a Transformer fusion framework that performs full-stack operator fusion and pipelining scheduling across the transformer encoder-decoder pipeline. By leveraging Einsum-based abstractions, TransFusion models Transformer layers as Einsum Cascades, enabling fine-grained intra-layer pipelining and direct inter-layer propagation of activation. The paper describes and evaluates DPipe, a DAG-based DP scheduler that maps fused computation graphs onto spatial accelerators through latency-aware pipelining. In addition, TileSeek explores the expanded tiling space introduced by full-stack fusion using Monte Carlo Tree Search to jointly optimize tiling factors and memory locality under strict buffer constraints. Evaluated across both cloud and edge architecture, TransFusion delivers up to an average of 1.6× speedup on cloud architecture and 2.2× on edge architecture over the prior state-of-the-art, FuseMax, by jointly optimizing inter-layer data reuse, intra-layer pipelining, and operator scheduling.

Acknowledgement

HPCA anonymous reviewers provided valuable feedback and constructive comments that helped improve the quality of the paper. In particular, one reviewer's suggestions led us to include experiments under varying computational capabilities, which made our evaluation more comprehensive.

References

[1] Apple. 2024. Core ML Tools. https://apple.github.io/coremltools/docs-guides/source/opt-palettization-overview.html.

[2] Minsik Cho, Keivan Alizadeh-Vahid, Saurabh N. Adya, and Mohammad Rastegari. 2021. DKM: Differentiable K-Means Clustering Layer for Neural Network Compression. *ArXiv* abs/2108.12659 (2021). https://api.semanticscholar.org/CorpusID:237353080

[3] Jaewan Choi, Hailong Li, Byeongho Kim, Seunghwan Hwang, and Jung Ho Ahn. 2022. Accelerating Transformer Networks through Recomposing Softmax Layers. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 92–103.

[4] Alexis CONNEAU and Guillaume Lample. 2019. Cross-lingual Language Model Pretraining. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/c04c19c2c2474dbf5f7ac4372c5b9af1-Paper.pdf

[5] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations (ICLR)*.

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[7] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 2198, 15 pages.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[9] 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).

[10] Prakhar Ganesh, Yao Chen, Xin Lou, Mohammad Ali Khan, Yin Yang, Hassan Sajjad, Preslav Nakov, Deming Chen, and Marianne Winslett. 2021. Compressing Large-Scale Transformer-Based Models: A Case Study on BERT. *Transactions of the Association for Computational Linguistics* 9 (2021), 1061–1080. https://doi.org/10.1162/tacl_a_00413

[11] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024).

[12] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. 2023. OliVe: Accelerating Large Language Models via Hardware-friendly Outlier-Victim Pair Quantization. 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 3, 15 pages. https://doi.org/10.1145/3579371.3589038

[13] Yatharth Gupta, Vishnu V. Jaddipal, Harish Prabhala, Sayak Paul, and Patrick von Platen. 2024. Progressive Knowledge Distillation Of Stable Diffusion XL Using Layer Level Loss. *ArXiv* abs/2401.02677 (2024). https://api.semanticscholar.org/CorpusID:266818179

[14] Tae Jun Ham, Sungjun 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. A3: Accelerating Attention Mechanisms in Neural Networks with Approximation. *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), 328–341. https://api.semanticscholar.org/CorpusID:211296403

[15] 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

[16] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. 2024. FlashDecoding++: Faster Large Language Model Inference with Asynchronization, Flat GEMM Optimization, and Heuristics. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 148–161. https://proceedings.mlsys.org/paper_files/paper/2024/file/5321b1dabcd2be188d796c21b733e8c7-Paper-Conference.pdf

[17] Tao Huang, Yuan Zhang, Mingkai Zheng, Shan You, Fei Wang, Chen Qian, and Chang Xu. 2023. Knowledge Diffusion for Distillation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 2849, 18 pages.

[18] 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

[19] Guillaume Lample and Alexis Conneau. 2019. Cross-lingual Language Model Pretraining. *ArXiv* abs/1901.07291 (2019). https://api.semanticscholar.org/CorpusID:58981712

[20] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. 2022. xFormers: A modular and hackable Transformer modelling library. https://github.com/facebookresearch/xformers.

[21] Jianhui Li, Zhennan Qin, Yijie Mei, Jingze Cui, Yunfei Song, Ciyong Chen, Yifei Zhang, Longsheng Du, Xianhang Cheng, Baihui Jin, Yan Zhang, Jason Ye, Eric Lin, and Dan Lavery. 2024. oneDNN Graph Compiler: A Hybrid Approach for High-Performance Deep Learning Compilation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 460–470. https://doi.org/10.1109/CGO57630.2024.10444871

[22] Junlong Li, Yiheng Xu, Tengchao Lv, Lei Cui, Cha Zhang, and Furu Wei. 2022. DiT: Self-supervised Pre-training for Document Image Transformer. In *Proceedings of the 30th ACM International Conference on Multimedia* (Lisboa, Portugal) *(MM '22)*. Association for Computing Machinery, New York, NY, USA, 3530–3539. https://doi.org/10.1145/3503161.3547911

[23] Wenjie Li, Dongxu Lyu, Gang Wang, Aokun Hu, Ningyi Xu, and Guanghui He. 2024. Hardware-oriented algorithms for softmax and layer normalization of large language models. *Science China Information Sciences* 67, 10 (2024), 200404.

[24] Yanjing Li, Sheng Xu, Baochang Zhang, Xianbin Cao, Peng Gao, and Guodong Guo. 2022. Q-ViT: Accurate and Fully Quantized Low-bit Vision Transformer. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 2496, 13 pages.

[25] Zhikai Li and Qingyi Gu. 2023. I-ViT: Integer-only Quantization for Efficient Vision Transformer Inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 17065–17075.

[26] Yang Lin, Tianyu Zhang, Peiqin Sun, Zheng Li, and Shuchang Zhou. 2022. FQ-ViT: Post-Training Quantization for Fully Quantized Vision Transformer. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, Luc De Raedt (Ed.). ijcai.org, 1173–1179. https://doi.org/10.24963/IJCAI.2022/164

[27] Zhenhua Liu, Yunhe Wang, Kai Han, Wei Zhang, Siwei Ma, and Wen Gao. 2021. Post-Training Quantization for Vision Transformer. In *Proceedings of the 35th International Conference on Neural Information Processing Systems (NIPS '21)*. Curran Associates Inc., Red Hook, NY, USA, Article 2152, 12 pages.

[28] 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

[29] Jiachen Mao, Huanrui Yang, Ang Li, Hai Li, and Yiran Chen. 2021. TPrune: Efficient Transformer Pruning for Mobile Devices. *ACM Trans. Cyber-Phys. Syst.* 5, 3, Article 26 (April 2021), 22 pages. https://doi.org/10.1145/3446640

[30] Nandeeka Nayak, Toluwanimi O. Odemuyiwa, Shubham Ugare, Christopher Fletcher, Michael Pellauer, and Joel Emer. 2023. TeAAL: A Declarative Framework for Modeling Sparse Tensor Accelerators. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 1255–1270. https://doi.org/10.1145/3613424.3623791

[31] Nandeeka Nayak, Xinrui Wu, Toluwanimi O. Odemuyiwa, Michael Pellauer, Joel S. Emer, and Christopher W. Fletcher. 2024. FuseMax: Leveraging Extended Einsums to Optimize Attention Accelerator Design. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1458–1473. https://doi.org/10.1109/MICRO61859.2024.00107

[32] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. 2021. The Design Process for Google's Training Chips: TPUv2 and TPUv3. *IEEE Micro* 41, 2 (2021), 56–63. https://doi.org/10.1109/MM.2021.3058217

[33] Nvidia. 2024. TensorRT. https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-803/best-practices/index.html.

[34] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315. https://doi.org/10.1109/ISPASS.2019.00042

[35] Hongwu Peng, Shaoyi Huang, Tong Geng, Ang Li, Weiwen Jiang, Hang Liu, Shusen Wang, and Caiwen Ding. 2021. Accelerating Transformer-based Deep Learning Models on FPGAs using Column Balanced Block Pruning. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. 142–148. https://doi.org/10.1109/ISQED51717.2021.9424344

[36] Tairen Piao, Ikhyun Cho, and U. Kang. 2022. SensiMix: Sensitivity-Aware 8-bit index 1-bit value mixed precision quantization for BERT compression. *PLOS ONE* 17, 4 (04 2022), 1–22. https://doi.org/10.1371/journal.pone.0265621

[37] Zheng Qu, Liu Liu, Fengbin Tu, Zhaodong Chen, Yufei Ding, and Yuan Xie. 2022. DOTA: Detect and Omit Weak Attentions for Scalable Transformer Acceleration. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 14–26. https://doi.org/10.1145/3503222.3507738

[38] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving Language Understanding by Generative Pre-Training. (2018).

[39] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[40] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 68658–68685. https://proceedings.neurips.cc/paper_files/paper/2024/file/7ede97c3e082c6df10a8d6103a2eebd2-Paper-Conference.pdf

[41] Mohammadali Shakerdargah, Shan Lu, Chao Gao, and Di Niu. 2024. MAS-Attention: Memory-Aware Stream Processing for Attention Acceleration on Resource-Constrained Edge Devices. *arXiv preprint arXiv:2411.17720* (2024).

[42] S. Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. 2019. Patient Knowledge Distillation for BERT Model Compression. In *Conference on Empirical Methods in Natural Language Processing*. https://api.semanticscholar.org/CorpusID:201670719

[43] Hamid Tabani, Ajay Balasubramaniam, Shabbir Marzban, Elahe Arani, and Bahram Zonooz. 2021. Improving the Efficiency of Transformers for Resource-Constrained Devices . In *2021 24th Euromicro Conference on Digital System Design (DSD)*. IEEE Computer Society, Los Alamitos, CA, USA, 449–456. https://doi.org/10.1109/DSD53832.2021.00074

[44] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[46] 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)*. 97–110. https://doi.org/10.1109/HPCA51647.2021.00018

[47] Naigang Wang, Chi-Chun Liu, Swagath Venkataramani, Sanchari Sen, Chia-Yu Chen, Kaoutar El Maghraoui, Vijayalakshmi Srinivasan, and Leland Chang. 2022. Deep Compression of Pre-trained Transformer Models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1028, 15 pages.

[48] Shuohang Wang, Luowei Zhou, Zhe Gan, Yen-Chun Chen, Yuwei Fang, Siqi Sun, Yu Cheng, and Jingjing Liu. 2020. Cluster-Former: Clustering-based Sparse Transformer for Long-Range Dependency Encoding. *ArXiv* abs/2009.06097 (2020). https://api.semanticscholar.org/CorpusID:260424300

[49] Wenhui Wang, Hangbo Bao, Shaohan Huang, Li Dong, and Furu Wei. 2021. MiniLMv2: Multi-Head Self-Attention Relation Distillation for Compressing

Pretrained Transformers. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, Online, 2140–2151. https://doi.org/10.18653/v1/2021.findings-acl.188

[50] Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020. MiniLM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 485, 13 pages.

[51] Yannan N. Wu, Joel S. Emer, and Vivienne Sze. 2019. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*.

[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. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1970, 16 pages.

[53] Chong Yu, Tao Chen, Zhongxue Gan, and Jiayuan Fan. 2023. Boost Vision Transformer with GPU-Friendly Sparsity and Quantization. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 22658–22668. https://doi.org/10.1109/CVPR52729.2023.02170

[54] Fang Yu, Kun Huang, Meng Wang, Yuan Cheng, Wei Chu, and Li Cui. 2022. Width & Depth Pruning for Vision Transformers. *Proceedings of the AAAI Conference on Artificial Intelligence* 36, 3 (Jun. 2022), 3143–3151. https://doi.org/10.1609/aaai.v36i3.20222

[55] Shixing Yu, Tianlong Chen, Jiayi Shen, Huan Yuan, Jianchao Tan, Sen Yang, Ji Liu, and Zhangyang Wang. 2022. Unified Visual Transformer Compression. In *ICLR*.

[56] Li Yuan, Yunpeng Chen, Tao Wang, Weihao Yu, Yujun Shi, Zi-Hang Jiang, Francis E.H. Tay, Jiashi Feng, and Shuicheng Yan. 2021. Tokens-to-Token ViT: Training Vision Transformers From Scratch on ImageNet. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 558–567.

[57] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. 2020. GOBO: Quantizing Attention-Based NLP Models for Low Latency and Energy Efficient Inference . In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 811–824. https://doi.org/10.1109/MICRO50266.2020.00071

[58] Size Zheng, Siyuan Chen, Siyuan Gao, Liancheng Jia, Guangyu Sun, Runsheng Wang, and Yun Liang. 2023. TileFlow: A Framework for Modeling Fusion Dataflow via Tree-based Analysis. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 1271–1288. https://doi.org/10.1145/3613424.3623792