From TeAAL to FuseMax: Separation of Concerns for Attention Accelerator Design

Nandeeka Nayak, *University of California, Berkeley, CA, 94720, USA*
Toluwanimi O. Odemuyiwa, *University of California, Davis, CA, 95616, USA*
Xinrui Wu, *Tsinghua University, Beijing 100190, China*
Michael Pellauer, *Nvidia, Westford, MA, 01886, USA*
Joel S. Emer, *Massachusetts Institute of Technology, Cambridge, MA, 02139, USA*
Christopher W. Fletcher, *University of California, Berkeley, CA, 94720, USA*

Attention for transformers has recently received significant "attention" as a target for custom acceleration. Our prior work, TeAAL, proposes a new accelerator design methodology that allows architects to reason about and optimize their designs iteratively. With a focus on attention, this work makes contributions to both the theory and practice of TeAAL's methodology. On the theory side, we propose a set of analyses that can be applied using only the algorithm specification—the cascade of Einsums summations. On the practice side, we use the new analyses to understand and taxonomize the space of attention algorithms and to iteratively build up an efficient, high-utilization accelerator. Our resulting design, FuseMax, achieves an average 6.7× speedup on attention and 5.3× speedup on end-to-end transformer inference over the prior state of the art, FLAT, while using 79% and 83% of the energy, respectively.

Over the past few years, transformers[7] have emerged as the model architecture of choice for a wide range of machine learning applications, from natural language processing to computer vision to speech recognition. This rise has been accompanied by a corresponding wave of proposals for accelerating transformers in both software and hardware.

Although these accelerators have the potential to provide dramatic speedup over the best CPU and GPU algorithms, they take significant effort and space to describe, model, evaluate, compare, and extend. Indeed, accelerators regularly employ specialized algorithms and mechanisms to improve compute utilization and performance while reducing memory traffic and energy, all tailored to the specific algorithms they aim to optimize. As a result, accelerators are typically described with either Register Transfer Language (RTL) or a block diagram and an accompanying natural language description. The former is verbose and often makes it difficult to identify key conceptual features, while the latter is imprecise and often incomplete. Neither makes it easy to model and evaluate the impact of proposed design changes.

Our prior work, TeAAL,[4] improves accelerator design methodology in a way that enables architects to qualitatively and quantitatively compare and optimize accelerators. More specifically, it provides a separation of concerns between the different features that compose the implementation of a specific kernel on an accelerator by proposing a hierarchy of specifications from the most high level and concise—the algorithm—to the most detailed and verbose—the binding of

Digital Object Identifier 10.1109/MM.2025.3589955
Date of publication 17 July 2025; date of current version 12 September 2025.

compute and data movement in time and space. The architecture of the accelerator exists outside the hierarchy, placing constraints on the choices that can be made at each level.

At the top of this hierarchy, TeAAL proposes a novel algorithm specification: a directed, acyclic graph of tensor algebra operations called the *cascade of Einsums* (*Einstein summations*). In a nutshell, an Einsum describes a multidimensional space called the *iteration space*, and, for each point in that space, it defines a projection into the tensors and a compute specification. A cascade of Einsums is a sequence of dependent Einsums that can be used to describe and specify a larger kernel. Although we hypothesized that cascades of Einsums were general enough to describe algorithms across a variety of domains, TeAAL focused on a single domain-sparse tensor algebra.

In this work, we go beyond sparse tensor algebra and propose a novel accelerator for attention, called *FuseMax*. Beyond the FuseMax accelerator itself, the work makes contributions to both the theory and practice of TeAAL's separation of concerns-based design methodology.

On the theory side, we propose a set of Einsum-level analyses that provide insight into accelerator efficiency regardless of later choices made in other parts of the TeAAL hierarchy (e.g., the mapping). Importantly, in TeAAL's separation of concerns, *the precise amount of compute and the precise dependencies between those computes are fully specified in the cascade of Einsums*, enabling designers to reason about accelerator efficiency using *only* the cascade of Einsums. In this work, we show how Einsum-explicit compute dependencies can be used to make nontrivial deductions about a kernel's allowed fusion granularity and algorithmic minimum per-tensor live footprint. The relationship between the live footprint and the buffer capacity, in turn, has implications for the required data movement.

On the practice side, we apply TeAAL's separation of concerns-based methodology to the design of FuseMax. We formalize attention algorithms by expressing them as cascades of Einsums and taxonomize the space of attention algorithms using our newly proposed analysis. Based on our taxonomy, we select an efficient variant to implement for our accelerator. We then lower this cascade onto a spatial-array accelerator, modifying the architecture to remove constraints and enable the desired choices at each level of the hierarchy.

FuseMax is compute bound and provides ~100% utilization of both its 2-D and 1-D arrays throughout the attention layer. Additionally, FuseMax's on-chip memory requirements are invariant to sequence length and require no extra spills to memory regardless of sequence length. With this design, FuseMax achieves 6.7× speedup on attention and a 5.3× speedup on full end-to-end inference with 79% and 83% of the energy, respectively, relative to the prior state-of-the-art accelerator FLAT.[3]

CASCADES OF EINSUMS

We describe the algorithms implemented by the accelerator using computations expressed as Einsums, where an Einsum expresses computation on tensors.

An *N-tensor* is a multidimensional array with *N* dimensions (called *ranks*). For example, a 0-tensor is a scalar, a 1-tensor is a vector, and a 2-tensor is a matrix. A *point* is a logical location within a tensor that contains a scalar *value*. A point is identified by an *N*-tuple of *coordinates* with one coordinate for each rank in the tensor. For example, we can define a matrix A with a value at coordinates (m,k) as $A_{m,k}$. A *fiber* is a set of points in a tensor with the same coordinates in all ranks except one. Following the example, we can talk about *M*-fibers of *A*—points in *A* with the same *k* coordinate and different *m* coordinates—and *K*-fibers of *A*—points in *A* with the same *m* coordinate and different *k* coordinates. We use the same symbol (e.g., *M*) to describe both the name and shape of a rank.

In Einsum notation, we can express matrix multiply as

$$Z_{m,n} = A_{k,m} \times B_{k,n}. \tag{1}$$

An Einsum describes a multidimensional space $K \times M \times N$ called the *iteration space*. For each point (k,m,n) in that space, it defines a projection into the tensors—$A_{k,m}$, $B_{k,n}$, and $Z_{m,n}$—and a compute specification $(\times)$ between the inputs and an (implied) sum of partial products into the output.

Extended General Einsums (EDGE)[6] allow us to express kernels beyond standard tensor algebra by supporting user-defined compute operators. For example, the following is the full specification for matrix multiplication:

$$Z_{m,n} = A_{k,m} \cdot B_{k,n} :: \bigvee_{k} \times (\cap) \bigwedge_{k} + (\cup). \tag{2}$$

EDGE allows the user to specify two actions: the map action $(\wedge)$ and the reduce action $(\vee)$, specified after the "::" delineator. Each map and reduce action contains two operations: merge and compute. Compute defines the operation to apply between two data values and can be *any* user-defined function. In this work, we use $\times$, $\div$, max (maximum), $+$, exp (exponentiation) and $-$ (see Cascades 2 and 3). Merge specifies which regions of the iteration space to touch; execution will not need to access the data space corresponding to

culled points. Common merge operations include intersection ($\cap$), which touches points with nonzero values in *both* operands, and union ($\cup$), which touches points where at least one of the operands is nonzero. Together, merge and compute precisely define the computations in an Einsum.

In (2), $(\wedge_k)$ specifies a map action between A and B on the k rank, and the intersection merge operator $(\cap)$ culls k points where at least one operand is zero. The compute operator $(\times)$ multiplies the data values of the coordinates surviving intersection. The reduce action $(\vee_k)$ on the k rank gathers all non-empty points in the k rank and reduces them using addition $(+)$. As a shorthand, we allow the compute operator of the map action to be placed between the inputs and use addition as the default compute for reduction [e.g., (1)].

Additionally, EDGE expresses recursion and iteration through generational/iterative ranks. We can express the iterative prefix sum as follows:

$$S_{i+1} = S_i + A_i \tag{3}$$

$$\diamond : i \geq I. \tag{4}$$

Here, S is a tensor with the *iterative* rank (as opposed to a *standard* rank), I, ranging from zero to K (inclusive). Statement 4 indicates the stopping condition for the iterative expression (when i is greater than or equal to I).

A *cascade of Einsums*[4] is a sequence of dependent Einsums that can be used to describe and specify a larger kernel. The cascade provides a precise specification of the dependencies between tensors in the algorithm. Additionally, depending on how the cascade is interpreted, it can provide an upper bound,[4] lower bound,[2] or, like in this work, an exact specification for the compute performed. Importantly, all other details of the kernel implementation are described in lower-level specifications, as discussed in the next section.

SEPARATION OF CONCERNS

Beyond the cascade of Einsums, this work applies TeAAL's separation of concerns-based hierarchy, visualized in Figure 1, to accelerator design. Although the cascade of Einsums specifies what computation is required, the *mapping*, *format*, and *binding* describe how it should occur.

The mapping describes a *mapping graph*, a directed, acyclic graph whose nodes are points in the iteration spaces of the Einsums, and whose edges are ordering constraints between the points. Mapping specifications typically include aspects such as loop order (dataflow), partitioning, and work scheduling (sequential versus parallel operations). Thus, the edges in the mapping graph can be true dependencies (enforced by the cascade) or additional constraints imposed by the mapping specification. We note that the mapping graph is never realized explicitly; it is simply an abstraction used to explain what we mean by mapping.

The format describes the tensors' representations. Because all the tensors in the target workload (attention) are dense, the only rational representation is that of an uncompressed, multidimensional array. Thus, we do not discuss the format specification further in this work.

The binding describes how the nodes in the mapping graph are bound to the architecture, including which compute unit each node is associated with, when that node will be executed, and where the inputs and outputs are stored in the memory hierarchy. This binding must obey the ordering enforced by the mapping graph and the physical limitations of the architecture but is otherwise unconstrained.

The architecture (not pictured in Figure 1) constrains the allowed choices at each level of the hierarchy. Based on the included functional units, data paths, buffers, and so on, certain Einsums, mappings, formats, and bindings may or may not be efficient or even legal. Designing a new accelerator requires enabling sufficient choices in the cascade, mapping, format, and binding to support efficient execution while, at the same time, making it cost-effective to implement.

PASSES PERFORMED BY A CASCADE

Before describing the FuseMax architecture, we demonstrate a novel analysis on kernels expressed in the cascade of Einsums abstraction. Although prior work[4] provides a precise definition of Einsums, a major contribution in our work is to show how this definition

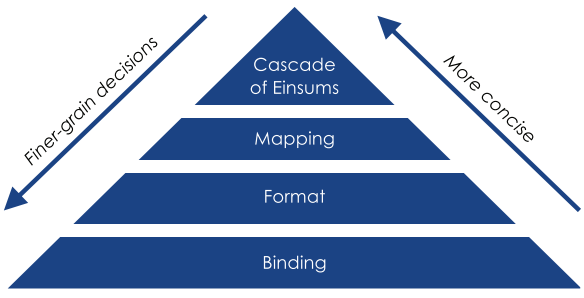

FIGURE 1. Separation of the concerns-based hierarchy first proposed in TeAAL.[4]

can be leveraged to inform accelerator design. Specifically, we recognize that the cascade makes explicit *precisely* what the dependencies are between Einsums. We show how this can be used to make nontrivial deductions about a kernel's allowed fusion granularity and algorithmic minimum per-tensor live footprint. The relationship between the live footprint and the buffer capacity, in turn, has implications for the required data movement.

To illustrate this idea, we present a simple pedagogical example, shown in Cascade 1.

Cascade 1: An example two-pass cascade.

$$Y = A_k \times B_k \tag{5}$$

$$Z = Y \times A_k. \tag{6}$$

The Einsum in (5) performs a dot product between A_k and B_k, and the Einsum in (6) multiplies the first Einsum's result Y by A_k again to produce Z. If we want to minimize the data traffic of A_k, we need to choose a dataflow for each Einsum that keeps A_k stationary and fuses the two Einsums together. In other words, the dataflow must finish using the first element of A_k before moving on to the next. However, such a dataflow does not exist for this cascade. Any implementation must visit *every* element of A_k to compute Y before it can revisit *any* element of A_k to compute Z.

Based on this pattern, we define a *pass* that a cascade performs over a particular fiber of a particular rank and tensor to be a traversal of every element of that fiber. Each time that an element *must* be revisited *after* visiting every other element of that fiber, there is an additional pass. For example, Cascade 1 performs two passes over the K rank of A_k.

The number of passes that a cascade performs is relevant because it restricts the possible fusion schedules. Einsums within a pass can be fused at will, producing and consuming a tile of the intermediate at a time. Einsums in different passes can only be fused if fibers of the reread rank are small enough to fit in local storage, which is often not the case. Revisiting Cascade 1, for large K, Einsums 5 and 6 cannot be fused on the K rank. Any implementation must visit all elements of the K-fiber of A to produce Y before it can visit any of the elements of that fiber to produce Z.

This analysis also provides a nontrivial lower bound on the tensors' live footprints. For example, the algorithmic minimum live footprint for tensor A is a fiber of shape K. In other words, an architecture must either have enough buffer space to hold an entire K-fiber of A or spill and reload that fiber, incurring memory traffic proportional to the shape of K. We note that this analysis is mapping independent. There is no dataflow for this cascade that enables a smaller live footprint. Different algorithms for the same problem may require different numbers of passes. A full discussion of how to manipulate a cascade to reduce the number of passes required can be found in the original article.[5]

TAXONOMIZING ATTENTION

Next, we apply the passes analysis to the family of numerically stable attention algorithms. To do so, we first need to express each algorithm as a cascade of Einsums.

In the original transformer article,[7] the kernel was described with the following equation:

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{7}$$

However, this equation says almost nothing about what the inputs Q, K, and V look like, what iteration space needs to be traversed, or even what compute is required for *softmax*. We clarify these points by rewriting (7) as a cascade of Einsums (Cascade 2). We name the ranks as follows: M and P are the sequence lengths for Q and K/V, respectively, and E and F are the embeddings for Q/K and V, respectively.

Cascade 2: A three-pass attention cascade.

$$QK_{m,p} = Q_{e,p} \times K_{e,m} \ /* \ \text{Pass 1} \ */ \tag{8}$$

$$GM_p = QK_{m,p} :: \bigvee_m \max \ (\cup) \tag{9}$$

$$SN_{m,p} = e^{QK_{m,p} - GM_p} \ /* \ \text{Pass 2} \ */ \tag{10}$$

$$SD_p = SN_{m,p} \tag{11}$$

$$A_{m,p} = SN_{m,p}/SD_p \ /* \ \text{Pass 3} \ */ \tag{12}$$

$$AV_{f,p} = A_{m,p} \times V_{f,m}. \tag{13}$$

We now summarize this cascade; more detail is available in the original article.[5] Einsum 8[a,b] multiplies the query $(Q_{e,p})$ and the key $(K_{e,m})$. Einsum 9 then computes the maximum value in each M-fiber, improving numerical stability by preventing overflow during exponentiation. Einsum 10 computes the softmax numerator, Einsum 11 computes the softmax denominator, and Einsum 12 computes the softmax result. Finally, Einsum 13 multiplies this result with the value

[a]Einsums do not require the transpose as this information is implicit in the indices.

[b]In Einsum 8, we also substitute E for d_k following the notation defined earlier, where the name of a rank is also its rank shape.

TABLE 1. Classifying prior attention algorithms based on the number of passes required by each. The reference citations correspond to those in the conference article.[5]

Three Pass	Two Pass	One Pass
PyTorch[42]	TileFlow[62]	FlashAttention[15]
TensorFlow[2]	Choi et al.[12]	FlashAttention-2[14]
FLAT[28]	< >	Rabe and Staats[47]
E.T.[6]	< >	< >

$(V_{f,m})$. Einsums 8–13 can be modified to refer to the full batched, multihead self-attention[7] by adding the batch (B) and head (H) ranks to all tensors. To simplify the notation, we assume the presence of the B and H ranks but omit writing them throughout the rest of this work.

We find that existing approaches to attention can be classified as three-, two-, or one-pass cascades, where an N-pass cascade performs N passes of a given M-fiber. Under this taxonomy, Cascade 2 is a three-pass cascade. In pass 1, we compute $QK_{m,p}$ and GM_p; in pass 2, we compute $SN_{m,p}$ and SD_p; and in pass 3, we compute $A_{m,p}$ and $AV_{f,p}$. Notice that we must finish an entire M-fiber of Einsum 9 (reading an entire M-fiber of $QK_{m,p}$) before GM_p is ready to start Einsum 10 (where we must read the same M-fiber of $QK_{m,p}$ again). Similarly, we must finish an entire M-fiber of Einsum 11 (reading an entire M-fiber of $SN_{m,p}$) before SD_p is ready to start Einsum 12 (where we must read the same M-fiber of $SN_{m,p}$ again). To summarize, as a consequence of the *dependencies* between Einsums, this cascade must perform three passes over each M-fiber. This holds for any choice of mapping (including ones that perform fusion).

Using the notion of passes, we taxonomize the space of attention algorithms, dividing them into three-, two-, and one-pass algorithms (see Table 1). We select the FlashAttention-2 cascade (reproduced in Cascade 3) to implement for FuseMax because it is the one-pass cascade that minimizes the required compute. A detailed walk-through of this cascade is available in the original article.[5]

Cascade 3: A one-pass attention cascade. Note that $M1$ is used as a standard rank (e.g., to access $BQK_{m1,m0,p}$) and as an iterative rank (e.g., to access $RM_{m1,p}$). The stopping condition for all iterative ranks is $m1 \geq M1$ (Statement 31).

Initialization:

$$BK_{e,m1,m0} = K_{e,m1\times M0+m0} \tag{14}$$

$$BV_{f,m1,m0} = V_{f,m1\times M0+m0} \tag{15}$$

$$RM_{m1:m1=0,p} = -\infty \tag{16}$$

$$RD_{m1:m1=0,p} = 0 \tag{17}$$

$$RNV_{m1:m1=0,p} = 0. \tag{18}$$

Extended Einsums:

$$BQK_{m1,m0,p} = Q_{e,p} \times BK_{e,m1,m0} \tag{19}$$

$$LM_{m1,p} = BQK_{m1,m0,p} :: \bigvee_{m0} \max(\cup) \tag{20}$$

$$RM_{m1+1,p} = \max(RM_{m1,p}, LM_{m1,p}) \tag{21}$$

$$SLN_{m1,m0,p} = e^{BQK_{m1,m0,p} - RM_{m1+1,p}} \tag{22}$$

$$SLD_{m1,p} = SLN_{m1,m0,p} \tag{23}$$

$$SLNV_{f,m1,p} = SLN_{m1,m0,p} \times BV_{f,m1,m0} \tag{24}$$

$$PRM_{m1,p} = e^{RM_{m1,p} - RM_{m1+1,p}} \tag{25}$$

$$SPD_{m1,p} = RD_{m1,p} \times PRM_{m1,p} \tag{26}$$

$$RD_{m1+1,p} = SLD_{m1,p} + SPD_{m1,p} \tag{27}$$

$$SPNV_{f,m1,p} = RNV_{f,m1,p} \times PRM_{m1,p} \tag{28}$$

$$RNV_{f,m1+1,p} = SLNV_{f,m1,p} + SPNV_{f,m1,p} \tag{29}$$

$$AV_{f,p} = RNV_{f,M1,p} \,/\, RD_{M1,p} \tag{30}$$

$$\diamond : m1 \geq M1. \tag{31}$$

FuseMax Accelerator Design

We now describe FuseMax, an efficient mapping and binding of an attention algorithm (specifically the one-pass cascade in Cascade 3) to a spatial-array-style architecture. We start with an architecture based on the prior state-of-the-art accelerator, FLAT,[3] shown in Figure 2(a). This architecture features three main components: a 2-D processing element (PE) array that performs matrix multiplication via a skewed dataflow, a 1-D array that contains many more functional units, and a global buffer between the arrays and the off-chip memory. We now describe the modifications that we make to the architecture to enable an efficient cascade, mapping, and binding.

The goal when selecting the mapping, binding, and architecture is to fully utilize all the available compute units. In our evaluation of prior work [Figure 5(b) and (c)], we observe that at short sequence lengths, the 2-D PE array is underutilized because it must wait for the 1-D PE array to compute the softmax. At longer sequence

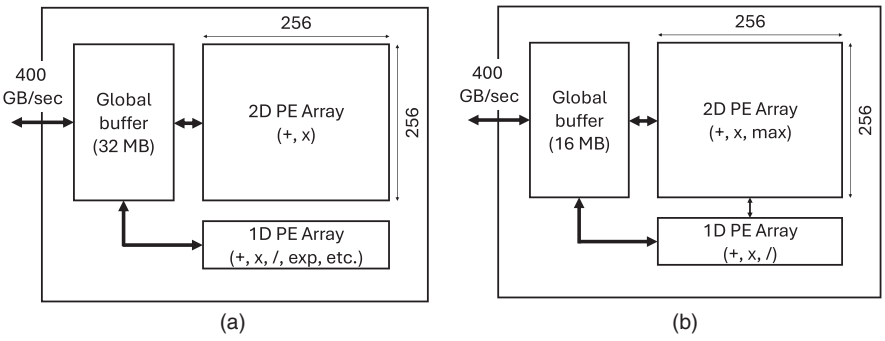

FIGURE 2. Spatial-array architecture evolution. (a) Spatial-array architecture used by FLAT.[3] (b) Spatial-array architecture used by FuseMax. max: maximum; exp.: exponentiation.

lengths, both arrays are underutilized because the workload becomes memory-bandwidth limited.

Cascade of Einsums

Looking only at the cascade of Einsums, we already see that this architecture will be unable to achieve high utilization of the 2-D array. Only (19) and (24) can be evaluated by the 2-D array, and all other Einsums must be evaluated on the 1-D array. In fact, assuming that all floating-point operations (including exponentiation and division) take one (pipelined) cycle, when computing attention for BERT, the FLAT architecture requires the 1-D array to be active, greater than 9× longer than the 2-D array.

To alleviate the 1-D array compute bottleneck, we add logic to support exponentiation via the Taylor series expansion and a max functional unit, to allow Einsums 20, 22, and 23 to also be computed on the 2-D array. Figure 3 shows these changes. At long sequence lengths, the ratio of 1-D to 2-D array compute is nearly one to one.

Mapping

FuseMax leverages fusion in conjunction with the one-pass cascade to eliminate the memory traffic of these tensors, regardless of the sequence length. Specifically, we partition on both *M* and *P* (forming *M*1, *M*0 and *P*2, *P*1, and *P*0) and maximally fuse all levels in the attention loop nest. The full loop nest is available in the original article,[5] but for this discussion, let us focus on Einsums 20–22. The loop nest for these Einsums is shown in Mapping 1.

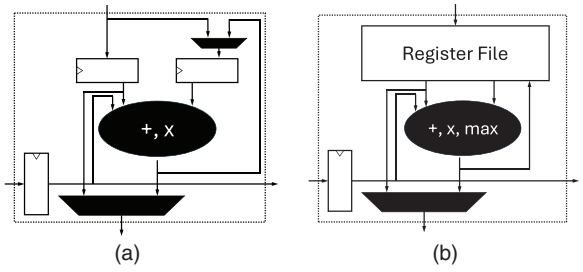

FIGURE 3. 2-D array PE architecture evolution. (a) FLAT.[3] PE. (b) FuseMax PE.

Mapping 1: The fused loop nest for Einsums 20–22.

```
for p2 in range(P2):
  for m1 in range(M1):
    for p1 in range(P1):
      parallel_for p0 in range(P0):
        parallel_for m0 in range(M0):
          # Computed on the 2-D array
          LM[p2, m1, p1, p0, m0 + 1] =
            max(LM[p2, m1, p1, p0, m0],
              BQK[p2, m1, p1, p0, m0])
        # Computed on the 1-D array
        RM[p2, m1 + 1, p1, p0] =
            max(RM[p2, m1, p1, p0],
              LM[p2, m1, p1, p0, M0])
        parallel_for m0 in range(M0):
          # Computed on the 2-D array
          SLN[p2, m1, p1, p0, m0] =
            exp(BQK[p2, m1, p1, p0, m0]
              - RM[p2, m1 + 1, p1, p0])
```

To enable this mapping, we favor a very tight coupling between the 2-D and 1-D arrays. As soon as the 2-D array finishes producing an element of *LM*, it should be available for *RM* on the 1-D array, and, similarly, as soon as the 1-D array produces the new element of *RM*, it should be available for the 2-D array. Unfortunately, in the FLAT architecture [see Figure 2(a)], the two arrays communicate via the global buffer, requiring extra cycles to translate between the skewed

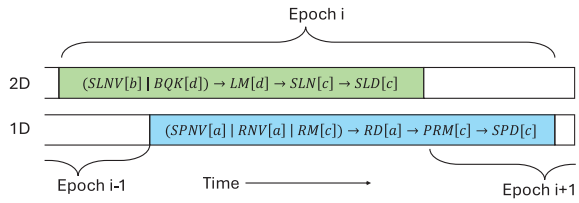

FIGURE 4. FuseMax pipelining at a glance. Each tensor name (e.g., SLNV) corresponds to the Einsum used to compute that tensor (see Cascade 3). *a*, *b*, *c*, and *d* denote tile-relative coordinates where $a < b < c < d$. If epoch i produces tiles with coordinates *a*, *b*, *c*, and *d*, Epoch $i + 1$ produces tiles with identifiers $a + 1$, $b + 1$, $c + 1$, $d + 1$ and so on. "*A*|*B*" denotes "computing tile *A* is interleaved with computing tile *B*." "$A \rightarrow B$" denotes "computing tile *A* is done before computing tile *B*." Computing $AV_{f,p}$ is not shown. The green and blue time periods that make up an epoch take almost the same number of cycles.

(2-D array) and unskewed (1-D array) dataflows and extra energy to read and write operands to the buffer. We alleviate these issues by skewing the dataflow of the 1-D array and supporting direct connectivity between the two arrays [see Figure 2(b)].

Binding

The dependencies between different Einsums in our cascade necessitate a binding that implements fine-grain pipeline parallelism to achieve high utilization of both the 1-D and 2-D spatial arrays. Figure 4 shows the waterfall diagram for FuseMax in the steady state. Time is broken into epochs. Each epoch performs the same set of tile-granular operations at specific tile-relative coordinates (given by (a)–(d) in the figure). Across all epochs, the kernel evaluates all tiles, and each Einsum in Cascade 3 is mapped to either the 2-D or 1-D array for all epochs (as shown in Figure 4). A more detailed description of the interleaving is available in the original article.[5] The only change we need to make to enable this interleaved binding is to expand the register files of the 1-D and 2-D arrays so that they can hold intermediates for multiple tiles.

EVALUATION

We now summarize our experimental setup and evaluation results for FuseMax. We evaluate all accelerators and configurations using the same transformer models used by FLAT:[3] BERT-Base (BERT), TrXL-wt103 (TrXL), T5-small (T5), and XLM, focusing on sequence lengths between 1K and 1M. We perform our evaluation using two tools for tensor algebra accelerator modeling and design space exploration: Timeloop and Accelergy. We use these tools to build models of the accelerator architectures and evaluate each Einsum individually. The results from individual Einsums are combined using the heuristics presented in prior work for evaluating full cascades.[4]

We evaluate FuseMax against two baselines: an unfused baseline and FLAT.[3] We build the unfused baseline that implements the three-pass cascade (Cascade 2) by combining the costs of three phases: *QK* (Einsum 8), the three-pass softmax (Einsums 9–12), and *AV* (Einsum 13). For the FLAT baseline, we created and validated a Timeloop model that reproduces the FLAT authors' (corrected) code to within a <1% error.

To demonstrate the sources of the improvements achieved by FuseMax, we present three configurations, building up the full design: +Cascade uses the one-pass cascade on the FLAT architecture, +Architecture adds all FuseMax architecture changes but implements a binding that fully produces and consumes one $M0 \times P0$ tile of *BQK* before starting the next, and +Binding adds FuseMax's pipelined/interleaved binding. The FuseMax architecture uses the hardware parameters presented in Figure 2(b).

Figure 5(a) shows the speedup of the various configurations of FuseMax over an unfused baseline and FLAT. Across all models and sequence lengths, FuseMax achieves an average speedup of 10× over the unfused baseline and 6.7× over FLAT on attention. We explore the source of these improvements by looking specifically at the utilization of the two PE arrays.

Figure 5(b) shows the utilization of the 1-D PE array when performing attention. FLAT's utilization drops for sequence lengths greater than 256K—it becomes memory-bandwidth limited because it must spill the *QK* and *A* tensors to memory. By using a one-pass cascade (+Cascade), FuseMax is never memory bound, and its utilization becomes independent of sequence length. We also note that without the FuseMax binding (+Architecture), the 1-D array is forced to stall and utilization drops. Adding in this binding (+Binding) enables FuseMax to fully utilize the 1-D array again.

Similarly, Figure 5(c) shows the utilization of the 2-D array. Because of the large amount of compute required for the softmax, most configurations achieve poor utilization of this array. In fact, because the one-pass cascade increases the compute required, +Cascade's 2-D array utilization is lower than FLAT's at short sequence lengths. On the other hand, FuseMax (+Binding) achieves high utilization across the board and, at long sequence lengths, reaches almost 100% utilization, making it 2-D-array compute bound. Both baselines achieve slightly higher utilization on XLM,

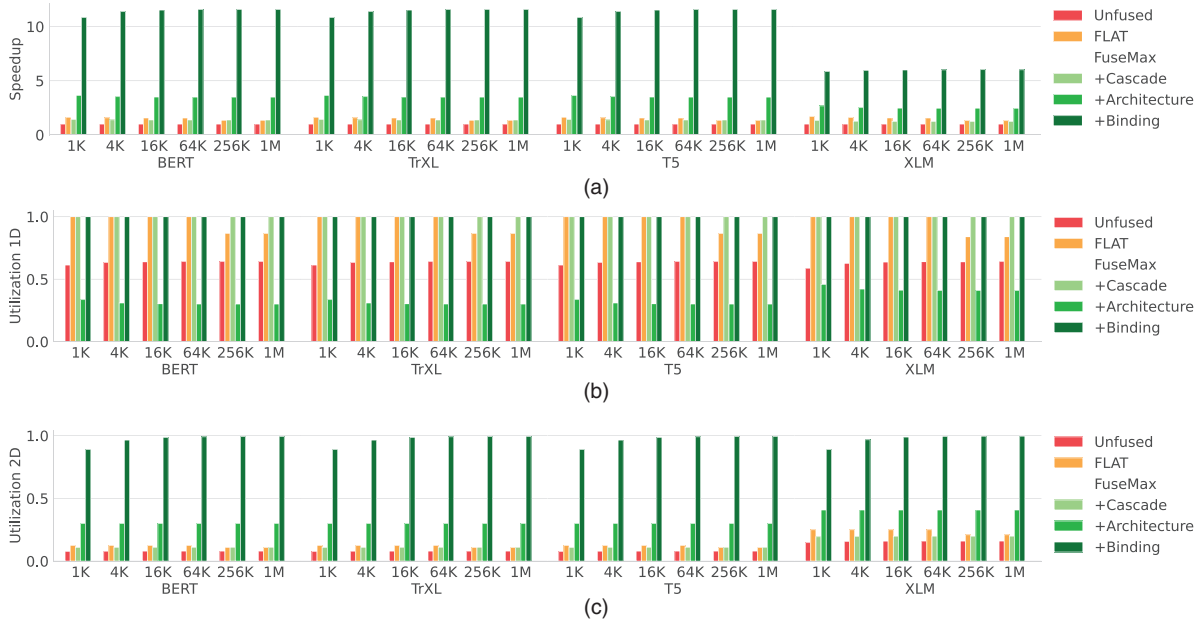

FIGURE 5. Utilization and speedup for the different PE arrays on the unfused baseline, FLAT, and three configurations building up FuseMax. (a) Speedup on attention. (b) 1-D PE array utilization. (c) 2-D PE array utilization.

which can be attributed to the higher intensity caused by a larger embedding dimension (*E*/*F*).

FuseMax's improved utilization and significantly decreased memory traffic translate not only to speedups but also to energy savings across the board. Across all models and sequence lengths, FuseMax uses an average of 77% of the unfused baseline's energy and 79% of FLAT's energy. These results indicate that, even though FuseMax requires more compute than the unfused baseline or FLAT, the cost of the additional compute is less than the savings achieved by lower memory and global buffer traffic.

As sequence length increases, a larger fraction of the total end-to-end transformer latency and energy is spent on attention. On end-to-end transformer inference, FuseMax achieves an average speedup of 7.6× over the unfused baseline and 5.3× over FLAT, using only 82% of the unfused baseline's energy and 83% of FLAT's energy.

We further observe the efficiency of the FuseMax design across a range of accelerator sizes. By varying the size of the PE array (between 16 × 16 and 512 × 512) and setting the global and per-PE buffers to accommodate the resulting pipelined/interleaved binding, we can generate a family of designs for efficient transformer inference (see Figure 6). These designs scale only the compute available, not the memory bandwidth, and yet continue to see improvements,

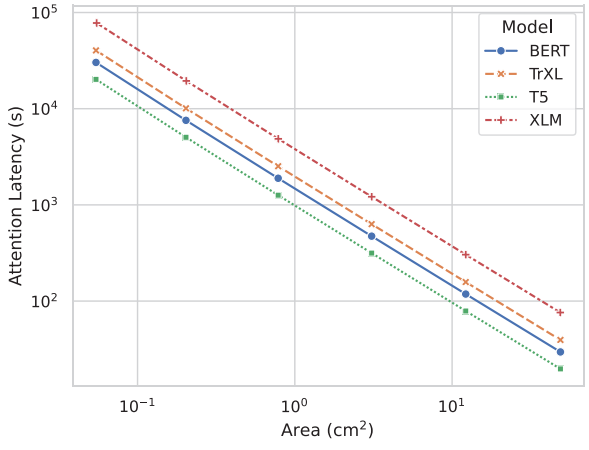

FIGURE 6. Pareto-optimal curves for FuseMax at sequence length 256 K.

indicating that even for very large accelerators, the FuseMax kernel is not memory bound.

CONCLUSION

This work extends and implements the TeAAL[4] accelerator design methodology to propose a high-utilization spatial-array architecture for attention. To design FuseMax, we first started by expressing the space of numerically stable exact attention variants as cascades of Einsums. Studying these cascades revealed

a novel analysis, applicable to any cascade, that can be used to infer nontrivial lower bounds on either the required on-chip buffer capacity or memory traffic. We then explored the architecture requirements at each level of the TeAAL hierarchy and minimally, iteratively improved our accelerator design to remove constraints and enable a high-utilization and efficient implementation.

Beyond this work, we are actively exploring how to apply this methodology to a wider range of domains. Cascades are a powerful tool for describing algorithm proposals and differentiating between competing proposals. For example, leading up to the publication of this work, we used cascades to describe and differentiate between sparse attention kernels. Similar to the attention kernels that we described in this article, the sparse attention kernels we explored were describable in ~10 Einsums each, and one could clearly articulate the core differences between them as point differences in the cascades. We are currently exploring the expression of algorithms across domains—including fully homomorphic encryption, robotics, RTL simulation, tensor decomposition, and more—using cascades of Einsums.

Similarly, we have found the iterative approach enabled by the TeAAL methodology to be effective both for the design of new accelerators (like this work) and the implementation of software kernels on existing architectures. For example, as part of an ongoing collaboration with Amazon Web Services (AWS), we have successfully applied the TeAAL methodology to the design of kernels for the AWS Trainium accelerator. We hope that this line of work will promote a more effective and more efficient implementation and optimization process for software and hardware designers and across both academia and industry.

ACKNOWLEDGMENTS

We thank the anonymous *IEEE Micro* and MLArchSys reviewers for their feedback on submitted versions of this work; Abhimanyu Bambhaniya, Sheng-Chun Kao, and Tushar Krishna for discussions on FLAT; and finally, Tanner Andrulis and Angshuman Parashar for help with Timeloop. We also thank Nafea Bshara, Ron Diamant, Serina Tan, Stephen Neuendorffer, Yakun Sophia Shao, Hongbin Zheng, and others at Amazon, AMD, and Nvidia for helpful discussions about the work as it matured.

Finally, we thank our collaborators in FPSG and the TeAAL-UGrads (and affiliate researchers) group, including Timor Averbuch, Arz Bshara, Severin Bochem, Kris Dong, Jaewon Hur, Yuxin Jin, Ronit Nagarapu, Jules Peyrat, Chenxi Wan, Yingchen Wang, Frederic Wu, Sabrina Yarzada, and Yan Zhu.

REFERENCES

1. T. Dao, "FlashAttention-2: Faster attention with better parallelism and work partitioning," 2023, *arXiv:2307.08691*.
2. M. Gilbert, Y. N. Wu, A. Parashar, V. Sze, and J. S. Emer, "Looptree: Exploring the fused-layer dataflow accelerator design space," 2024, *arXiv:2409.13625*.
3. S.-C. Kao, S. Subramanian, G. Agrawal, A. Yazdanbakhsh, and T. Krishna, "FLAT: An optimized dataflow for mitigating attention bottlenecks," in *Proc. 28th ACM Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, 2023, pp. 295–310.
4. N. Nayak, T. Odemuyiwa, S. Ugare, C. W. Fletcher, M. Pellauer, and J. S. Emer, "TeAAL: A declarative framework for modeling sparse tensor accelerators," in *Proc. 56th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2023, pp. 1255–1270.
5. N. Nayak, X. Wu, T. Odemuyiwa, M. Pellauer, J. S. Emer, and C. W. Fletcher, "FuseMax: Leveraging extended Einsums to optimize attention accelerator design," in *Proc. 57th IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2024, pp. 1458–1473, doi: 10.1109/MICRO61859.2024.00107.
6. T. O. Odemuyiwa, J. S. Emer, and J. D. Owens, "The EDGE language: Extended general einsums for graph algorithms," 2024, *arXiv:2404.11591*.
7. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst. (NIPS)*, 2017, pp. 6000–6010.

NANDEEKA NAYAK is a computer science Ph.D. student at University of California, Berkeley, CA, 94720, USA, where she is advised by Christopher W. Fletcher. Her research interests include improving the design and implementation of efficient kernels for tensor algebra and related domains, spanning both theory—through methodologies for designing efficient kernels—and practice, by applying these methodologies to propose efficient implementations for specific kernels. Nayak received her B.S. degree in computer science from Harvey Mudd College. Contact her at nandeeka@berkeley.edu.

TOLUWANIMI O. ODEMUYIWA is a computer science Ph.D. student at the University of California, Davis, CA, 95616, USA. Her research interests include parallel programming and computer architecture, general purpose computing on GPUs, and sparse tensor computations. Odemuyiwa received her

M.Sc. degree in electrical engineering from San Jose State University. She is a Graduate Student Member of IEEE. Contact her at todemuyiwa@ucdavis.edu.

XINRUI WU is a computer science Ph.D. student in the Department of Computer Science at the University of California, Los Angeles, CA, 90095, USA. Her research interests include microarchitectural design for sparse tensor algebra and hardware–software co-optimization. Wu received her B.Eng. degree in computer science and technology from Tsinghua University. Contact her at alicewu@cs.ucla.edu.

MICHAEL PELLAUER is a principal research scientist in the Architecture Research Group, Nvidia, Westford, MA, 01886, USA, and is also with the Massachusetts Institute of Technology. His research interest is building domain-specific accelerators, with a special emphasis on deep learning and sparse tensor algebra. Pellauer received his Ph.D. degree in computer science from the Massachusetts Institute of Technology. Contact him at mpellauer@nvidia.com.

JOEL S. EMER is a professor of the practice at the Massachusetts Institute of Technology, Cambridge, MA, 02139, USA, and a senior distinguished research scientist at Nvidia. His research interests include accelerators, parallel and secure architectures, and reliability analysis. Emer received his Ph.D. degree in electrical engineering from the University of Illinois Urbana-Champaign. He is a Fellow of IEEE and the Association for Computing Machinery and a member of the National Academy of Engineering. Contact him at emer@csail.mit.edu.

CHRISTOPHER W. FLETCHER is an associate professor of electrical engineering and computer science at the University of California, Berkeley, CA, 94720, USA. His research interests include secure architecture, application-specific acceleration, and applied cryptography. Fletcher received his Ph.D. degree in electrical engineering and computer science from the Massachusetts Institute of Technology. Contact him at cwfletcher@berkeley.edu.