2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)

FuseMax: Leveraging Extended Einsums to Optimize Attention Accelerator Design

Nandeeka Nayak
University of California, Berkeley
Berkeley, CA, USA
nandeeka@berkeley.edu

Xinrui Wu
Tsinghua University
Beijing, China
xr-wu20@mails.tsinghua.edu.cn

Toluwanimi O. Odemuyiwa
University of California, Davis
Davis, CA, USA
todemuyiwa@ucdavis.edu

Michael Pellauer
NVIDIA
Westford, MA, USA
mpellaer@nvidia.com

Joel S. Emer
Massachusetts Institute of Technology / NVIDIA
Cambridge, MA, USA
jsemer@mit.edu

Christopher W. Fletcher
University of California, Berkeley
Berkeley, CA, USA
cwfletcher@berkeley.edu

***Abstract*—Attention for transformers is a critical workload that has recently received significant 'attention' as a target for custom acceleration. Yet, while prior work succeeds in reducing attention's memory-bandwidth requirements, it creates load imbalance between operators that comprise the attention computation (resulting in severe compute under-utilization) and requires on-chip memory that scales with sequence length (which is expected to grow over time).**

This paper ameliorates these issues, enabling attention with nearly 100% compute utilization, no off-chip memory traffic bottlenecks, and on-chip buffer size requirements that are independent of sequence length. The main conceptual contribution is to use a recently proposed abstraction—the *cascade of Einsums*—to describe, formalize, and taxonomize the space of attention algorithms that appear in the literature. In particular, we show how Einsum cascades can be used to infer non-trivial lower bounds on the number of *passes* a kernel must take through its input data, which has implications for either required on-chip buffer capacity or memory traffic. We show how this notion can be used to meaningfully divide the space of attention algorithms into several categories and use these categories to inform our design process.

Based on the above characterization, we propose FuseMax—a novel mapping and binding of attention onto a spatial array-style architecture. On attention, in an iso-area comparison, FuseMax achieves an average $6.7\times$ speedup over the prior state-of-the-art, FLAT, while using 79% of the energy. Similarly, on full end-to-end transformer inference, FuseMax achieves an average $5.3\times$ speedup over FLAT using 83% of the energy.

***Index Terms*—Tensor algebra, Extended Einsums, Spatial architectures, Attention**

I. Introduction

Over the past few years, transformers [52] have emerged as the model architecture of choice for a wide range of machine learning applications, from natural language processing [13], [17], [48], [49] to computer vision [18], [33] to speech recognition [4], [26]. This rise has been accompanied by a corresponding wave of proposals for accelerating transformers in both software [12], [14], [15] and hardware [28], [62].

This work was partially funded by NSF grants CNS-1954521, CNS-1942888, CNS-2154183, CCF-8191902, and CCF-2217099; as well as by an Intel gift and a Microsoft Research PhD fellowship.

Fortunately, many of the layers (projections, fully connected layers, etc.) used by transformers look very similar to prior generations of machine learning models. Their resource-intensive tensor products can be described and evaluated with existing tensor algebra accelerator modeling tools [29], [35], [41], and many of the other layers (e.g., layer normalization) have negligible impact on performance and can be safely ignored.

However, the attention layer [52]—usually described as a matrix multiplication, a softmax, and then another matrix multiplication—does not fit into either of these boxes. For example, the softmax is both memory intensive (featuring low algorithmic reuse) *and* compute intensive (featuring exponentiation and division). Furthermore, attention's characteristics preclude many "free lunches" often used to improve efficiency in other DNN models. For example, because all tensors are a function of the model inputs, there is no opportunity to amortize memory access costs with an increased batch size. Additionally, since none of the operands can be computed before the inputs are given, compression/strength reduction techniques (e.g., quantization [22], [60], sparsity [34], [46], [53], etc.) must be applied dynamically, leading to more complicated algorithms and hardware designs.

To illustrate the difficulty in accelerating attention, consider the state-of-the-art accelerator for attention: FLAT [28]. FLAT uses fusion to reduce attention memory bandwidth bottlenecks on a spatial architecture (e.g., a TPU [27]). Specifically, FLAT maps attention's matrix multiplications to the 2D spatial array and softmax operations to a separate 1D array. While FLAT's design does make attention compute bound, it becomes compute bottlenecked in the 1D array (the softmax), causing severe under utilization of the 2D array. While one could add additional PEs to the 1D array, this results in corresponding area costs.

Making matters worse, FLAT requires that the entire vector over which the softmax is performed be buffered on chip. This vector is proportional to the sequence length, which is growing rapidly with time (e.g., Google reports 10 million

2379-315/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00107

length sequences in research, which would require 100s of MegaBytes to buffer [44]). When the vector/sequence length grows beyond allowable buffer capacity, FLAT is forced to spill, which contributes significantly to attention energy consumption and can even make attention memory-bandwidth bound.

This paper. We address the above challenges by proposing a novel spatial architecture – *FuseMax* – to accelerate attention, with particular emphasis on removing bottlenecks imposed by the softmax. Our architecture addresses all of the aforementioned issues associated with FLAT. Namely:

- FuseMax is compute bound, but provides almost 100% utilization of both the 2D and 1D arrays throughout the attention layer, without adding additional PEs to the 1D array.
- FuseMax's on-chip memory requirements are invariant to sequence length and require no extra spills to memory regardless of sequence length.

The paper's technical core is split into three parts.

First, Section III demonstrates a novel analysis on kernels that uses the recently proposed *cascade of Einsums* abstraction [35]. In a nutshell, an Einsum defines an iteration space over tensors and what computation is done on and between tensors at each point in the iteration space. A cascade of Einsums is a sequence of dependent Einsums that can be used to describe and specify a larger kernel.

While prior work [35], [39] provides a precise definition for Einsums, a major contribution in our work is to show how this definition can be leveraged to inform accelerator design. Specifically, we recognize that the cascade makes explicit *precisely* what dependencies there are between Einsums. We show how this can be used to make non-trivial 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.

In more detail, this analysis provides insight into the number of *passes* an algorithm performs, i.e., the number of times a given element of an input must be revisited after visiting every other element of the input. Normally, one strives to choose a dataflow that exploits maximal reuse in a given element (or tile of elements) to avoid having frequently reload it. However, some algorithms preclude this strategy. In this work, we describe how to count the number of passes a cascade requires and present two methods for reducing the number of passes. In general, fewer passes is preferable; although, interestingly, we find that decreasing the number of passes can increase the required compute. Given that an Einsum cascade is mapping/scheduling agnostic, this analysis provides insight given any possible scheduling of the cascade onto hardware.

Next, Section IV applies the cascade of Einsums abstraction to describe and formalize the attention kernel. Using the notion of passes introduced in Section III, we taxonomize the space of numerically stable attention proposals that appear in the literature. For example, in a naïve implementation of attention, one must traverse the entire softmax input to build the softmax denominator *and only after that* can one revisit and scale each input (softmax numerator) by the denominator. Because this analysis is performed on the cascade of Einsums, our lower bounds on passes hold for any choice of mapping, including applications of fusion. For example, despite using fusion, FLAT employs a 3-pass cascade and its reliance on large on-chip buffering is a symptom of trying to avoid three passes-worth of DRAM traffic. We, then, show how transforming the attention cascade reduces the number of passes required.

Additionally, we find that expressing attention as a cascade of Einsums reveals that optimizations that were previously conflated can actually be applied separately. We specifically call out one that is used by 1-pass algorithms to eliminate the need for a second pass after the final softmax denominator has been calculated. We recognize that this optimization has the added benefit of decreasing the required divisions, which is not only useful for but can be applied to 2- and 3-pass cascades as well.

Finally, Section V uses the insights from Section IV as a starting point to develop a novel mapping and binding for attention that can be lowered to a spatial architecture. We call our architecture FuseMax. FuseMax adopts the 1-pass attention cascade used in FlashAttention-2 [14]. However, despite using the cascade from FlashAttention-2, binding this cascade to a spatial architecture is non-trivial. In particular, FlashAttention-2 binds the cascade onto a GPU, an architecture that features homogeneous PEs, each with relatively large per-PE storage, and expensive inter-PE communication. Spatial architectures feature opposite characteristics: heterogeneous PEs, each with smaller per-PE storage, and cheap (but restricted) inter-PE communication. Specifically, the networks that connect the PEs within the 2D array allow efficient communication primarily between neighbors. We overcome these differences and demonstrate a novel mapping and binding for the 1-pass cascade that achieves high utilization for entire transformer layers. Our architecture requires only minimal changes to a standard spatial architecture and is performance/energy robust to long sequence lengths (e.g., 1M tokens and beyond).

To summarize, we make the following contributions:

- We show how cascades of Einsums can be used to inform accelerator design, both in terms of reasoning about compute requirements and per-tensor live footprints. We formalize lower bounds on the number of passes a cascade imposes given any possible mapping of the cascade onto hardware.
- We use cascades of Einsums, and the observation about pass lower bounds, to provide a taxonomy and precise specification of numerically stable attention algorithms in the literature. Orthogonally, we show how previously entangled attention optimizations can be applied across attention algorithms.
- We propose a novel mapping and binding for attention for a spatial architecture—which we call FuseMax—that achieves high utilization for both 2D and 1D array PEs,

and has memory traffic requirements that are independent of sequence length.
- We evaluate FuseMax on BERT [17], TrXL [13], T5 [49], and XLM [13] and demonstrate a $6.7\times$ speedup on attention with 79% of the energy and a $5.3\times$ speedup on full end-to-end inference with 83% of the energy relative to FLAT.

II. Background

In this section, we describe the concepts and terminology used in the remainder of the paper.

A. Tensors

This paper focuses on algebraic computations on tensors, where a tensor is a multidimensional array. A tensor's *rank* refers to a specific dimension of the tensor, while the tensor's *shape* is the set of valid coordinates for each of the tensor's ranks. We use the notation N-tensor to denote a tensor with N ranks, where a 0-tensor is a scalar, a 1-tensor is a vector, a 2-tensor is a matrix, etc.

We adopt the format-agnostic *fibertree* abstraction of tensors, where a tensor is represented as a tree of fibers, as detailed in prior work [25], [35], [38], [43], [51], [55], [57], [58], using the specific version described in TeAAL [35, Section 2.1]. In this abstraction, a *fiber* consists of the set of coordinates for a given rank with common coordinates for all higher-level ranks. Each coordinate is coupled with a *payload*. The payload may contain a reference to a fiber in the next lower rank, or to a leaf data *value*.

B. Traditional Einsums

An Einsum expression defines a computation on a set of tensor operands using an iteration space that specifies the set of points where the computations are performed [35], [39]. For example, we describe matrix-matrix multiplication (GEMM) with the following Einsum:

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

where A and B are input 2-tensors of shape $K \times M$ and $K \times N$, respectively. Z is an output 2-tensor with shape $M \times N$. Throughout this paper, we use the same symbol for both the shape and *name* of a rank (e.g., rank K in A has a shape of K).

The *iteration space* of this Einsum is $[0, K) \times [0, M) \times [0, N)$. An evaluation of this Einsum must: (1) walk every (k, m, n) point in the iteration space; and, at each point (2) project into the *data space* of all input tensors, (3) multiply the corresponding data values, and (4) place the result at the corresponding data point in Z. If a value already exists at an (m, n) point in Z (due to computation at the same (m, n) point for a different k in the iteration space), reduce the two values together using addition. Note that the Einsum specifies *what* to compute; it does not indicate the order in which one walks the iteration space. These aspects are left to the *mapping* [9], [35], [41].

We also note that we can view the iteration space itself as a tensor. In the example above, this tensor has shape $K \times M \times N$. Therefore, we define a special fibertree—called the iteration space fibertree or *is-fibertree*—that is the fibertree representation of this iteration space tensor.

C. Extended Einsums

Traditional Einsums sufficiently express standard tensor algebra, including those supported in Basic Linear Algebra Subprograms (BLAS) [19], [30] and tensor network contractions [1]. However, they cannot handle more complex computations. The recently proposed Extended General Einsums notation (EDGE) [39], extends Einsums to handle graph algorithm computations. We find this abstraction useful for also expressing complex tensor algebra computations and use its notation throughout the paper. We now briefly summarize the portions of EDGE that we leverage.

1) User-Defined Computations: EDGE separates computations into three "actions": map ($\bigwedge$), reduce ($\bigvee$), and populate ($=$) [39]. Map specifies the pair-wise computation between the shared ranks of two tensors, reduce specifies the computation for the reduction step of an Einsum, and default populate ($=$) places a computed value from the right-hand side (RHS) of the Einsum to its location on the left-hand side (LHS).

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. Merge specifies which regions of the iteration space to touch; execution will not need to access the data space corresponding to culled points. Together, merge and compute precisely define the computations in an Einsum. Common merge operations include intersection ($\cap$), which touches points with non-zero values in *both* operands; and union ($\cup$), which touches points where at least one of the operands is non-zero.

The full EDGE specification for GEMM is then:

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

where $\bigwedge_k$ 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 coordinates surviving intersection. The reduce action ($\bigvee_k$) on the k rank gathers all non-empty points in the k rank and reduces them using addition ($+$).

In this work, we use three user-defined computations:

1) Maximum ($\max(\cup)$) takes the maximum of two values. Suppose we have the following expression: $Z_m = A_m \cdot B_m :: \bigwedge_m \max(\cup)$. The union merge operator ($\cup$) filters out any m coordinates where both operands contain 0 (and places 0 in the output). The $\max$ compute operator then returns the maximum of the two operands.
2) Divide ($\div(\leftarrow)$) divides two data values. Given the following expression, $Z_m = A_m \cdot B_m :: \bigwedge_m \div(\leftarrow)$, the merge operator ($\leftarrow$) only touches m points where there is a non-zero value in the B operand (see [39,

Appendix]), and the compute operator divides the data value in A with the data value in B.

3) Subtraction and Exponentiation: To apply the exponential to an expression that subtracts two tensors, we use the following expression: $Z_m = A_m \cdot B_m :: \bigwedge_m$ sub-then-exp($\mathbb{1}$). The user-defined operator (sub-then-exp) performs A_m minus B_m then applies the exponential to the result. The merge operator, $\mathbb{1}$, is EDGE's "pass-through" operator, which touches all m points in the iteration space.

In addition to map and reduce, EDGE enables the expression of user-defined *unary* operations on tensors. For example, we can express the application of the non-linear, sigmoid function (σ) on each element of a tensor A as $Z_m = \sigma(A_m)$.

2) Shorthand Notation: Throughout this paper, we take advantage of EDGE's shorthand notation [39] in the following ways:

- We drop all reduce actions that consist of add and union in the compute and merge operator, respectively ($\bigvee +(\cup)$). Thus, $Z_m = A_{k,m} :: \bigvee_k +(\cup)$ becomes $Z_m = A_{k,m}$.
- We express all map actions using infix notation; that is, $A_{k,m} \cdot B_{k,n} :: \bigwedge_k \times(\cap)$ becomes $A_{k,m} \times B_{k,n}$.
- When max is part of a map action ($A_m \cdot B_m :: \bigwedge_m \max(\cup)$), we replace it with the following shorthand: $\max(A_m, B_m)$.
- When $\div$ is part of a map action ($A_m \cdot B_m :: \bigwedge_m \div(\leftarrow)$), we replace it with the following: A_m / B_m.
- When sub-then-exp is part of the map action ($A_m \cdot B_m :: \bigwedge_m$ sub-then-exp($\mathbb{1}$)), we replace it with the shorthand $e^{A_m - B_m}$.
- We can express rank variable expressions with only one valid coordinate (e.g., $S_{i:i=2}$) using just the coordinate (in this case, S_2).

3) Filtering Rank Expressions: EDGE also enables expressing Einsums that touch only a subset of the data space of their constituent tensors. For example, we may express the prefix sum of a tensor A_k with the following Einsum:

$$S_{i+1} = A_{k:k \leq i}$$

For each coordinate i, S_{i+1} is built by reducing together the subset of A whose coordinates are $\leq i$. Note that this definition of prefix sum computes the entire sum for a given i without iteratively reusing the previous sum.

4) Expressing Iterative Computations: EDGE expresses recursion and iteration through generative/iterative ranks. We use the term *standard* ranks to differentiate non-iterative ranks from iterative ranks. We can express the iterative prefix sum as follows:

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

$$\diamond : i \geq K \tag{4}$$

Here, S is a tensor with the iterative rank, I, ranging from 0 to K (inclusive). Statement 4 indicates the stopping condition for the iterative expression (when i is greater than or equal to K).

5) Cascades of Einsums: TeAAL [35] introduces the concept of *cascades* of Einsums, which expresses directed acyclic graphs (DAGs) of Einsum expressions as a sequence of sub-Einsums. One can view the unrolled iterative expression in Einsum 3 as a cascade:

$$S_1 = S_0 + A_0$$
$$S_2 = S_1 + A_1$$
$$\ldots$$
$$S_K = S_{K-1} + A_K$$

Finally, we use the EDGE *Initialization* label to specify computations that initialize tensors, which occur once. We use the EDGE *Extended Einsum(s)* label to specify the computation that occurs on each iteration of a cascade of Einsums [39]. For example, see (Einsum) Cascade 5.

D. Mapping and Binding

While the cascade of Einsums specifies what computation is required, the *mapping* and *binding* describe how it should occur [9], [35], [41], [51]. We use the concept of *logical tasks* to define these terms. A logical task is a grouping of points in the iteration spaces of all Einsums. Tasks are defined such that each point in the iteration spaces is assigned to exactly one task. Logical tasks can be as small as a single point or as large as an entire iteration space. In the final schedule, each logical task must be assigned to exactly one compute unit that finishes the given task before moving onto the next task.

The mapping, therefore, describes a *task graph*, a directed, acyclic graph whose nodes are logical tasks and edges are dependencies between the tasks. Mapping specifications typically include aspects such as loop order, partitioning, and work scheduling (sequential vs. parallel operations) [35]. Thus, the dependencies in the task graph can be true dependencies (enforced by the cascade) or additional ordering constraints imposed by the mapping specification.

The binding describes how the tasks are bound to the actual hardware, including which compute unit each task is associated with, when that task will be executed, and where the inputs and outputs are stored in the memory hierarchy. This binding must obey the dependencies present in the task graph and the physical limitations of the architecture but is otherwise unconstrained.

E. Tensor Algebra Accelerators

In recent years, the popularity of domain-specific tensor algebra accelerators has increased. A typical accelerator based on a spatial architecture consists of off-chip main memory, an on-chip shared global buffer, various scratchpads, and a 1D and/or 2D processing engine (PE) array where each PE contains compute units [9], [27], [28], [38], [62]. This design minimizes memory transfer latency while maximizing compute utilization [7]–[9], [11], [27]. Various tools enable the quick modeling and design space exploration of tensor algebra accelerators, including Timeloop [41] and Accelergy [56], GAMMA [61], and DOSA [23].

III. Passes Performed by a Cascade of Einsums

Our first contribution is to demonstrate a novel analysis that can be applied to a cascade of Einsums. The key insight is that cascades of Einsums provide a precise description of the iteration space for each Einsum and the data space for each constituent tensor, enabling us to derive the algorithmic minimum live footprint for each tensor, with implications for the allowed fusion schedules and required buffer capacity/memory traffic. Because this analysis relies only on the cascade of Einsums, it holds for any choice of mapping and binding.

A. Calculating the Number of Passes

We will apply our analysis to attention in Section IV. To illustrate ideas, we first start with a simple pedagogical example, shown in Cascade 1.

$$Y = A_k \times B_k \tag{5}$$
$$Z = Y \times A_k \tag{6}$$

Cascade 1: An example 2-pass cascade.

Einsum 5 performs a dot product between A_k and B_k, and Einsum 6 multiplies the first Einsum's result Y by A_k again to produce Z. If we want to minimize 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 onto 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.

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 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.

Since an Einsum's iteration space can also be represented as a fibertree (i.e., an *is-fibertree* – see Section II-B), we extend our definition of an iteration space for a cascade of Einsums by considering its iteration space to be the sequence of the is-fibertrees for each Einsum. Now, in a scenario where fibers for a particular rank exist in multiple is-fibertrees; in each, they project to the same tensor; and there is a dependency such that all of the elements of the earlier is-fibertree's fiber must be read before any element can be read again by the later is-fibertree (for all mappings of the cascade), we refer to that read-read sequence as creating an additional *pass*. When there is a sequence of N such read-read dependencies, we say the cascade is an $(N+1)$-pass cascade. For our example, Cascade 1 requires two passes of the K rank.

B. Implications of the Number of Passes

The number of passes a cascade performs is relevant because it restricts 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 cannot be fused. Revisiting Cascade 1, 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 non-trivial 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.

C. Reducing the Number of Passes via Reassociation

Given the restrictions that multi-pass cascades place on the allowed dataflows and tensor live footprints, it can be beneficial to manipulate the cascade to reduce the number of passes required. Crucially, these manipulations are functionally equivalent and only change how Z is computed. In this section, we will present two methods for doing so, though we leave a full analysis of the space of pass-reduction approaches to future work.

1) Deferring the Multiplication by Y: First, we recognize that, by the distributive property, Einsum 6 can be factored to perform the reduction of A_k first, before multiplying the result by Y. Doing so, we get the following cascade:

$$Y = A_k \times B_k \tag{7}$$
$$X = A_k \tag{8}$$
$$Z = Y \times X \tag{9}$$

Cascade 2: A reassociation of Cascade 1 that defers the $Y\times$ to compute Z with 1-pass of the K rank.

Now, because there is no read-after-write dependency between Einsums 7 and 8, both Einsums can be included in the same pass. In fact, because Einsum 8 reduces away the K rank, Cascade 2 is a 1-pass cascade with respect to this rank. This reassociation actually provides a second benefit over Cascade 1: Einsum 9 now only requires one multiplication (as opposed to K multiplications in Einsum 6).

2) Iteratively Constructing Y and Z: Alternatively, we can iteratively construct Y and Z as we perform the pass through A_k. To do so, we will take a similar approach to the prefix-sum (see Sections II-C3-II-C4) and build intermediate Ys and Zs.

$$RY_{i+1} = A_{k:k\leq i} \times B_{k:k\leq i} \tag{16}$$
$$RZ_{i+1} = RY_{i+1} \times A_{k:k\leq i} \tag{17}$$

Just like with the prefix sum, this version requires a lot of extra compute, but, because $Y = RY_K$ and therefore $Z = RZ_K$, the final result is the same.

Initialization:

$$RY_{i:i=0} = 0 \tag{10}$$
$$RZ_{i:i=0} = 0 \tag{11}$$

Extended Einsums:

$$RY_{i+1} = RY_i + A_i \times B_i \tag{12}$$
$$RZ_{i+1} = RZ_i \times \frac{RY_{i+1}}{RY_i} + RY_{i+1} \times A_i \tag{13}$$
$$Z = RZ_K \tag{14}$$
$$\diamond : i \geq K \tag{15}$$

Cascade 3: A reassociation of Cascade 1 that iteratively constructs Y and Z with 1-pass of the K rank.

We remove this extra work by making the I ranks of RY_{i+1} and RZ_{i+1} iterative. This is shown in Cascade 3. Iterative RY_{i+1} (Einsum 12) looks very similar to the iterative prefix-sum. However, computing RZ_{i+1} is a little more complicated.

To derive the expression for RZ_{i+1}, we start by introducing one more intermediate S_i, which is the prefix sum for A_k:

$$S_i = A_{k:k\leq i-1} \tag{18}$$

Now, we can combine Einsums 17 and 18 to write RZ_i in terms of this prefix-sum:

$$RZ_i = RY_i \times S_i \tag{19}$$

Dividing both sides by RY_i, we derive an alternate definition for S_i:

$$S_i = \frac{RZ_i}{RY_i}$$

S_{i+1} can also be written using this alternative definition:

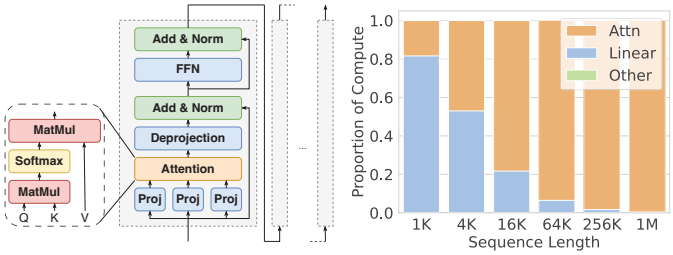
$$S_{i+1} = \frac{RZ_i}{RY_i} + A_i \tag{20}$$

We can combine Einsums 19 and 20 to compute RZ_{i+1} in terms of RZ_i (i.e., iteratively):

$$RZ_{i+1} = RY_{i+1} \times \left(\frac{RZ_i}{RY_i} + A_i\right)$$

Distributing RY_{i+1} and performing some reassociation, we get Einsum 13.

Cascade 3 is also a 1-pass cascade, performing one pass of the K rank of A_k (indexed with the variable i) and iteratively building RY_{i+1} and RZ_{i+1}. Unfortunately, unlike Cascade 2, Cascade 3 does require extra compute over the original Cascade 1. However, memory bandwidth-limited workloads can afford to trade off extra compute for reduced memory traffic, and Cascade 3 may still provide benefit.



(a) Encoder architecture (b) Required compute

Fig. 1: Overview of transformer encoder inference.

IV. Taxonomizing Attention as Einsum Cascades

Our second contribution is to apply the cascade of Einsums abstraction and the notion of passes to transformer models to describe, taxonomize, and highlight trade-offs in the space of attention implementations. This section first looks at the transformer model as a whole, identifying attention as an important kernel (Section IV-A). We then give an overview of attention and a "straightforward" (but inefficient) algorithm for softmax by writing them as cascades of Einsums (Sections IV-B-IV-C). Finally, we show how optimizations to softmax can be described by modifying the cascades and provide a taxonomy of the space using the number of passes required by each cascade (Sections IV-D-IV-E).

A. Transformers

Transformer models generally follow the architecture defined in [52]. Our work, which addresses the impact of long sequence lengths during self-attention, focuses on the encoder architecture.[1] Figure 1a gives an overview. The transformer first projects the input (by multiplying it by weight tensors) to form a *query*, *key*, and *value*. Self-attention is made up of three operations: a matrix multiplication of the query and key, a softmax on the result, and another matrix multiplication, which combines the softmax output with the value. The attention output is then deprojected (again, multiplying by a weight tensor), normalized, passed through a two-layer feed-forward neural network (FFN), and normalized once more.

As the sequence length grows, the relative importance of the different operations changes. Figure 1b shows that at shorter sequence lengths, the *weight-times-activation* "linear" layers are a larger fraction of the total required compute, while at long sequence lengths, the attention operation dominates. In all cases, the additional non-linearities (e.g., the normalization, the ReLU between the FFN layers, etc.) have negligible impact. In the next section, we focus on describing attention more precisely, and use our analysis to understand prior work on efficient implementations.

[1]During the decoder phase, inference is severely bottlenecked on the memory traffic required to read the KV cache [24], and therefore the on-chip accelerator design has less impact on performance.

B. Redefining Attention's "Matrix Multiplications"

In the original transformer paper [52], the kernel was described with the following equation:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{21}$$

However, this equation says almost nothing about what the inputs Q, K, and V look like or what iteration space needs to be traversed. We clarify these points by rewriting the above as a cascade of Einsums, with the exception of the softmax, whose cascade we will explore in Section IV-C. The first step is to give each of the ranks names: 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.

$$QK_{m,p} = \frac{1}{\sqrt{E}} \times Q_{e,p} \times K_{e,m} \tag{22}$$
$$A_{m,p} = softmax(QK_{m,p}) \tag{23}$$
$$AV_{f,p} = A_{m,p} \times V_{f,m} \tag{24}$$

Here, Einsums 22[2,3] and 24 look like matrix multiplications. Taking Einsum 24 as an example, for each point in the iteration space $F \times M \times P$, we perform a multiplication using elements from two 2-tensors ($A_{m,p}$ and $V_{f,m}$) to produce a 2-tensor output ($AV_{f,p}$), which requires reducing across the inputs' shared rank M. Einsums 22-24 can be modified to refer to the full batched, multi-head self attention [52] by adding the batch (B) and head (H) ranks to all tensors. This changes the characteristics of the kernel. Adding the B and H ranks means that Einsums 22 and 24 behave like many independent matrix multiplications instead of one monolithic matrix multiplication. The challenges with attention, described in Section I, still follow clearly from this modification. Because *all* tensors contain a B rank, the matrix multiplications are all unique to the specific batch's inputs. Therefore, none of these tensors can be computed before the inputs are given, and there is no data sharing between the different elements in the batch. Hence, to simplify notation, we assume the presence of the B and H ranks but omit writing them throughout the rest of paper.

C. Softmax as a Cascade of Einsums

We now apply the same precise notation to the softmax. A softmax [5] over a 1-tensor is traditionally expressed with the following equation:

$$A_m = \frac{e^{I_m}}{\sum_k e^{I_k}} \tag{25}$$

[2]Einsums do not require the transpose, since this information is implicit in the indices.

[3]In Einsum 22, we also substitute E for d_k following the notation defined in Section II-B, where the shape of a rank is also its rank name.

In the context of attention, this operation becomes two dimensional and can be expressed using the following cascade with input $QK_{m,p}$:

$$SN_{m,p} = e^{QK_{m,p}} \tag{26}$$
$$SD_p = SN_{m,p} \tag{27}$$
$$A_{m,p} = SN_{m,p}/SD_p \tag{28}$$

For each point in the iteration space (m, p), we exponentiate $QK_{m,p}$ to generate the softmax numerator ($SN_{m,p}$ in Einsum 26), reduce $SN_{m,p}$ with addition to produce the softmax denominator (SD_p in Einsum 27), and finally, divide the numerator and denominator to produce the final result ($A_{m,p}$ in Einsum 28).

1) Improving Numerical Stability: Because $e^{QK_{m,p}}$ can easily become extremely large, the above formulation suffers from overflow. Therefore, practical implementations [2], [42] often prefer the numerically stable variant that replaces Einsum 26 with:

$$GM_p = QK_{m,p} :: \bigvee_m \max(\cup) \tag{29}$$
$$SN_{m,p} = e^{QK_{m,p} - GM_p} \tag{30}$$

and drop the $\frac{1}{\sqrt{E}}$ term when computing $QK_{m,p}$.[4] To compute the *global maximum*[5] GM_p, we reduce $QK_{m,p}$ with the operator max (instead of $+$). Notice that subtracting GM_p from $QK_{m,p}$ in the exponent is equivalent to dividing by e^{GM_p}, and because the $\frac{1}{e^{GM_p}}$ term appears in both the numerator ($SN_{m,p}$ via Einsum 30) and denominator (SD_p via Einsum 27), the result ($A_{m,p}$) stays the same. This construction improves numerical stability by bounding the values of the softmax numerator $SN_{m,p}$ to the range $(0, 1]$.

D. Optimizing Softmax Compute

We now describe an optimization to attention that reduces compute requirements, specifically division. This optimization was used in FlashAttention-2 [14]. We point out that it can be applied more broadly, i.e., to any cascade we discuss in Section IV-E. Einsum 28 requires $M \times P$ divisions. While this is the best we can do for an independent softmax, we note that attention does not use the softmax in isolation [52]. Instead, it subsequently multiplies the result, $A_{m,p}$, and another tensor, $V_{f,m}$, per Einsum 24, reproduced here:

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

To optimize the full attention cascade, we can refactor Einsums 28 and 24 by, instead, first combining $SN_{m,p}$ and $V_{f,m}$ (Einsum 31) and reducing across the M rank and then performing the division (Einsum 32), as follows:

$$SNV_{f,p} = SN_{m,p} \times V_{f,m} \tag{31}$$
$$AV_{f,p} = SNV_{f,p}/SD_p \tag{32}$$

[4]The $\frac{1}{\sqrt{E}}$ term was introduced to bound the magnitude of $SN_{m,p}$ [52]. Because the numerically stable softmax variant already accomplishes this, the scaling is often omitted [12], [14], [15].

[5]"Global" here refers to over the entire M fiber.

3-pass	2-pass	1-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]		

TABLE I: Classifying prior attention algorithms.

This reassociation does $F \times P$ divisions instead of $M \times P$ divisions. Since M is the sequence length and F is an embedding dimension (i.e., $M \gg F$), this reassociation *reduces* the required divisions (by a factor of $\frac{M}{F}$).

E. Optimizing Softmax Live Footprint and Memory Traffic

We now apply the analysis described in Section III to analyze attention's live footprint and memory traffic. We consider the *exact attention* literature, omitting works that either do not model/evaluate the softmax or include approximation strategies that improve performance at the cost of reduced accuracy (increased perplexity). We discuss the latter in Section VII.

We find that existing approaches to attention can be classified as either 3-pass, 2-pass, or 1-pass cascades, where an N-pass cascade performs N passes of a given M fiber. See Table I. Next, we describe the key ideas of each.

1) 3-Pass Attention Cascades: The 3-pass cascade is the straightforward, numerically stable cascade that we already discussed in Section IV-C1, namely Einsums 29-30 followed by Einsums 27-28, reproduced in Cascade 4 for clarity.

$$QK_{m,p} = Q_{e,p} \times K_{e,m} \quad \text{/* Pass 1 */} \tag{33}$$
$$GM_p = QK_{m,p} :: \bigvee_m \max(\cup) \tag{34}$$
$$SN_{m,p} = e^{QK_{m,p} - GM_p} \quad \text{/* Pass 2 */} \tag{35}$$
$$SD_p = SN_{m,p} \tag{36}$$
$$A_{m,p} = SN_{m,p}/SD_p \quad \text{/* Pass 3 */} \tag{37}$$
$$AV_{f,p} = A_{m,p} \times V_{f,m} \tag{38}$$

Cascade 4: The 3-pass attention 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 34 (reading an entire M fiber of $QK_{m,p}$) before GM_p is ready to start Einsum 35 (where we must read the same M fiber of $QK_{m,p}$ again). Similarly, we must finish an entire M fiber of Einsum 36 (reading an entire M fiber of $SN_{m,p}$) before SD_p is ready to start Einsum 37 (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).

2) 2-Pass Attention Cascades: We now briefly summarize the 2-pass cascade, deferring details due to space. Rather than computing the global max and then starting the softmax (as in the 3-pass cascade), the 2-pass cascade first partitions the input, computes a per-partition *local max* and applies it to form a variant of $SN_{m,p}$ whose elements are likewise partitioned and adjusted by the local max. Analogously, each partition gets a local denominator (also adjusted by the same local max). While this is occurring, it builds the global max from the local max values. Next, in a second pass, it uses the global max to correct the per-partition numerators and denominators and compute the softmax output.

3) 1-Pass Attention Cascades: While prior work proposes multiple different 1-pass cascades [14], [15], [47], the main ideas are the same in each. Rather than using the per-partition local max to compute the local numerator and denominator, instead keep a *running max* that represents the max value seen so far. Each time a new running max is computed, also adjust previous results (e.g., numerator-times-V, denominator, etc.) with this max.

This transformation can be described more precisely using the reassociations presented in Section III-C. First, we modify Cascade 4 to multiply the softmax numerator-times-V and then compute the division (as described in Section IV-D). This reassociation combines the second and third passes of Cascade 4 (see Section III-C1). To ensure numerical stability, we cannot use the same strategy to combine the first and second passes. So we instead use the iterative approach (see Section III-C2).

We are now ready to describe FlashAttention-2's 1-pass cascade (shown as Cascade 5). We later use it to build FuseMax. Note the evidently increased compute relative to the 3-pass cascade. We will carefully design the binding in Section V to hide these overheads on a spatial architecture.

We will start by expressing the partitioning of both of the inputs $K_{e,m}$ and $V_{f,m}$ into M1 chunks of M0 elements each (Einsums 39-40). After computing $BQK_{m1,m0,p}$, this allows us to perform operations like maximum on individual $M0$ fibers, rather than on the whole tensor (Einsum 45). The problem is, of course, that the local maximum is not necessarily the same for all $M0$ fibers and so will not just cancel nicely like the global maximum.

We resolve this by instead using the running maximum ($RM_{m1,p}$)—the global maximum of all inputs seen so far—instead of the local maximum. We recognize that $M1$ can also serve as an iterative rank, and iteratively build up $RM_{m1,p}$. After initializing $RM_{m1:m1=0,p}$ to $-\infty$ (Einsum 41), we compute a new running maximum $RM_{m1+1,p}$ using the running maximum computed in the previous iteration $RM_{m1,p}$ and the new local maximum $LM_{m1,p}$ (Einsum 46).

We can now use the running maximum to compute a local numerator $SLN_{m1,m0,p}$ (Einsum 47), a local denominator $SLD_{m1,p}$ (Einsum 48), and even the softmax numerator-times-V $SLNV_{f,m1,p}$ (Einsum 49) using the partitioned $BV_{f,m1,m0}$ (Einsum 40).

Now consider the softmax denominator. Eventually, we would like to reduce $SLD_{m1,p}$ into a 1-tensor, but because its values may have been computed with different maximums, we cannot simply use addition. Instead, by introducing a

Initialization:

$$BK_{e,m1,m0} = K_{e,m1\times M0+m0} \tag{39}$$
$$BV_{f,m1,m0} = V_{f,m1\times M0+m0} \tag{40}$$
$$RM_{m1:m1=0,p} = -\infty \tag{41}$$
$$RD_{m1:m1=0,p} = 0 \tag{42}$$
$$RNV_{m1:m1=0,p} = 0 \tag{43}$$

Extended Einsums:

$$BQK_{m1,m0,p} = Q_{e,p} \times BK_{e,m1,m0} \tag{44}$$
$$LM_{m1,p} = BQK_{m1,m0,p} :: \bigvee_{m0} \max(\cup) \tag{45}$$
$$RM_{m1+1,p} = max(RM_{m1,p}, LM_{m1,p}) \tag{46}$$
$$SLN_{m1,m0,p} = e^{BQK_{m1,m0,p} - RM_{m1+1,p}} \tag{47}$$
$$SLD_{m1,p} = SLN_{m1,m0,p} \tag{48}$$
$$SLNV_{f,m1,p} = SLN_{m1,m0,p} \times BV_{f,m1,m0} \tag{49}$$
$$PRM_{m1,p} = e^{RM_{m1,p} - RM_{m1+1,p}} \tag{50}$$
$$SPD_{m1,p} = RD_{m1,p} \times PRM_{m1,p} \tag{51}$$
$$RD_{m1+1,p} = SLD_{m1,p} + SPD_{m1,p} \tag{52}$$
$$SPNV_{f,m1,p} = RNV_{f,m1,p} \times PRM_{m1,p} \tag{53}$$
$$RNV_{f,m1+1,p} = SLNV_{f,m1,p} + SPNV_{f,m1,p} \tag{54}$$
$$AV_{f,p} = RNV_{f,M1,p}/RD_{M1,p} \tag{55}$$
$$\diamond : m1 \geq M1 \tag{56}$$

Cascade 5: A 1-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 56).

new running denominator $RD_{m1,p}$ with iterative rank $M1$, we can correct the old denominator $RD_{m1,p}$ to the new running maximum $RM_{m1+1,p}$ and then perform the addition. We start by initializing the running denominator at the start of the computation to 0 (Einsum 42). Then, at each point $m1$, the correction factor $PRM_{m1,p}$ allows us to correct the previous running denominator $RD_{m1,p}$ with the new maximum (Einsum 51). In other words, $RD_{m1,p}$ is downscaled by $e^{RM_{m1,p}}$. $SPD_{m1,p}$ "switches" the downscaling factor on $RD_{m1,p}$ to $e^{RM_{m1+1,p}}$ by multiplying $RD_{m1,p}$ by $\frac{e^{RM_{m1,p}}}{e^{RM_{m1+1,p}}}$ ($PRM_{m1,p}$). Once $SLD_{m1,p}$ and $SPD_{m1,p}$ have the same maximum, they can be combined to produce the new running denominator $RD_{m1+1,p}$ (Einsum 52). We can do the same to compute the running numerator-times-V (Einsums 43, 53-54).

Finally, $AV_{f,p}$ can be computed by dividing the final numerator-times-V by the final denominator. By construction, at this point, $RNV_{f,M1,p}$ and $RD_{M1,p}$ are both downscaled by the same maximum $RM_{M1,p}$ (conveniently, also the global maximum) and can be correctly combined.

V. Mapping and Binding Attention

Based on the framework from Section IV, we now describe FuseMax, an efficient mapping and binding of an attention algorithm (specifically the 1-pass cascade in Cascade 5) to a spatial array-style architecture. To enable maximum flexibility while binding, FuseMax's mapping places each iteration space point in its own logical task.

The goal when binding a cascade onto hardware is to fully utilize all available compute units. In our evaluation of prior work (Figure 6 and Section VI-B), we observe that at short sequence lengths, the 2D PE array is under-utilized because it must wait for the 1D PE array to compute the softmax. At longer sequence lengths, both arrays are under-utilized since the workload becomes memory-bandwidth limited.

FuseMax's binding addresses these issues to achieve full utilization on both the 1D and 2D PE arrays. First, we decrease the compute performed by the 1D array by (1) applying the division reduction optimization (Section IV-D) and (2) sharing the other operations (sum/max/exp) between the 1D and 2D arrays. Similarly, we ensure that the workload is never memory-bandwidth limited by deeply fusing all Einsums in the cascade to restrict the live footprint to only what can be buffered on-chip. No matter the sequence length, our dataflow is never forced to spill any of its intermediates off-chip.

Architecture. FuseMax is a spatial array architecture based on the TPUv2/TPUv3 [37, Figure 1(e)]. The off-chip DRAM and a large global buffer both feed data to connected 2D and 1D arrays (see Figure 2). We set parameters to match the cloud configuration in prior work [28].

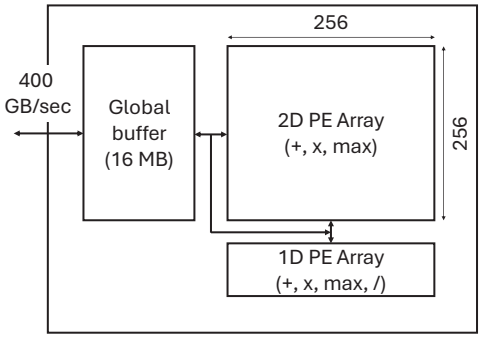
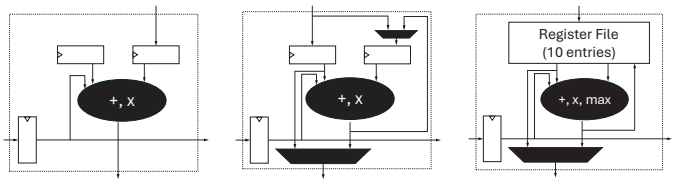


Fig. 2: Spatial array architecture assumed for FuseMax.

Figure 3 shows the evolution of the 2D PE array architecture, from a fixed-dataflow multiply-accumulate TPU PE (Figure 3a) to a flexible-dataflow multiply-accumulate PE (Figure 3b) to a FuseMax PE (Figure 3c). Note, although both the 1D and 2D PE arrays in FuseMax perform exponentiation, we implement exponentiation with 6 sequential multiply-accumulate operations [36], [53] and therefore do not require a dedicated exponentiation unit.

Mapping. Prior attention accelerators [28], [62] explore fusing many of attention's loop nests together. However, because these accelerators all use multi-pass cascades, the algorithmic minimum live footprint of some tensors (e.g.,



(a) TPU [27] PE (b) FLAT [28] PE (c) FuseMax PE

Fig. 3: 2D PE architecture evolution.

$QK_{m,p}$) is $O(M)$, meaning that for long sequence lengths, intermediates cannot be buffered on chip.

FuseMax leverages fusion in conjunction with the 1-pass cascade to eliminate the memory traffic of these tensors, regardless of the sequence length. Specifically, we partition on both M and P (forming $M1, M0$ and $P2, P1, P0$), and maximally fuse all levels in the attention loopnest as shown in Mapping 1. That is, all Einsums in Cascade 5 are fused except for the last (which is fused to the rest only on $P2$).

```
for p2 ...:
  for m1 ...:
    for p1 ...:
      parallel_for p0 ...:
        parallel_for m0 ...:
          (RNV[:, m1 + 1, p2, p1, p0],
           RD[m1 + 1, p2, p1, p0]) =
              ComputeRNVTile(
                Q[:, p2, p1, p0],
                K[:, m1, m0], V[:, m1, m0])
  for p1 ...:
    parallel_for p0 ...:
      AV[:, p2, p1, p0] =
        ComputeAVTile(
          RNV[:, m1 + 1, p2, p1, p0],
          RD[m1 + 1, p2, p1, p0])
```

Mapping 1: The FuseMax mapping as a loopnest. We partition on both M and P and map the innermost ranks $M0$ and $P0$ to the spatial array PEs. `ComputeRNVTile` performs Einsums 44-54 from Cascade 5. `ComputeAVTile` performs Einsum 55. Note that each Einsum represents a loopnest: by writing all Einsums in `ComputeRNVTile` under a single loopnest, we mean that we are maximally fusing those loopnests. Outer loops over B and H (if performing batched multihead attention) are not shown.

While prior work implementing attention in hardware [28], [62] does utilize the 2D spatial array for the tensor products, it fails to do so for the softmax, choosing instead to use the 1D array. Because there are far fewer total PEs in the 1D array than the 2D array, the softmax becomes a bottleneck. FuseMax improves utilization of the 2D spatial array by using it for both the tensor products and the exponentiation operator in the softmax. FuseMax parallelizes across the $M0$ and $P0$ ranks throughout the attention kernel (see Mapping 1). We set $M0 \times P0 = \#$ 2D Array PEs. The large spatial reductions required when parallelizing across the $M0$ rank are easily handled by the low-cost inter-PE communication network.

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 1D and 2D 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, b, c, d in the figure). Across all epochs, the kernel evaluates all tiles and each Einsum in Cascade 5 is mapped to either the 2D or 1D array for all epochs (as shown in the figure).

A major design consideration when binding the attention kernel is how to overcome the latency of fills and drains to/from the spatial array. Consider a tile of $QK_{m,p}$ of shape $M0 \times P0$. Per Einsum 22, the iteration space to evaluate this tile is $E \times M0 \times P0$ which becomes E cycles on the spatial array. For the networks we evaluate, $E = 64$ or 128. Assume $E = 64$. Using an output stationary dataflow, while each PE performs 64 MACCs, it takes ~ 256 cycles to both fill and drain the spatial array. Without careful interleaving, this combination of parameters causes low utilization because, for example, the running max $RM_{m1+1,p1,:}$ cannot be computed until a tile of $QK_{m1,:,p1,:}$ is completed and spatially reduced (drained) to form the local max $LM_{m1,p1,:}$ (Einsums 45-46).

Our binding address the above issues with two levels of interleaving. First, we interleave the construction of dependent tiles across epochs. This is reminiscent of software pipelining. For example, in Figure 4 the d-th tile of BQK and LM are completed in Epoch i (as they correspond to a fill followed by a drain and can be easily pipelined). The RM (which has to wait for the drain) for tile d takes place *in a later epoch*. Instead, Epoch i computes an earlier tile's running maximum $RM[c]$.

Second, we interleave the construction of certain tiles within an epoch at a fine (e.g., cycle-by-cycle) granularity. See the notation '$A|B$' in Figure 4. This is to ensure high utilization of both the 2D and 1D PE arrays at all times. To make this more clear, Figure 5 shows the start up and steady-state interleaving of $SLNV$ and BQK in the 2D array and $SPNV$ and RNV in the 1D array. In each cycle, a given PE in the 2D array computes a value for either BQK or $SLNV$ and this alternates cycle by cycle. Each neighbor-neighbor link in the array is active in every cycle—carrying data for one of the two operation types. By interleaving $SLNV$ with BQK, the 1D PEs can concurrently compute $SPNV$ and RNV.

Putting everything together, as Section VI-B will show, the above enables high utilization of all 2D and 1D array PEs.

FuseMax on GPUs. FuseMax's mapping and binding cannot be directly applied to GPUs. FuseMax's architecture features heterogeneous PEs, each with smaller per-PE storage, and cheap (but restricted) inter-PE communication. Specifically, the networks that connect the PEs within the 2D array allow efficient, fixed-latency communication primarily between neighbors, including between the bottom of the 2D array and the 1D array. However, the GPU architecture features opposite characteristics: homogeneous PEs, each with relatively large per-PE storage, and expensive, loosely coupled inter-PE communication. While concurrent work [50] has explored using the GPU's Tensor Cores to compute BQK and

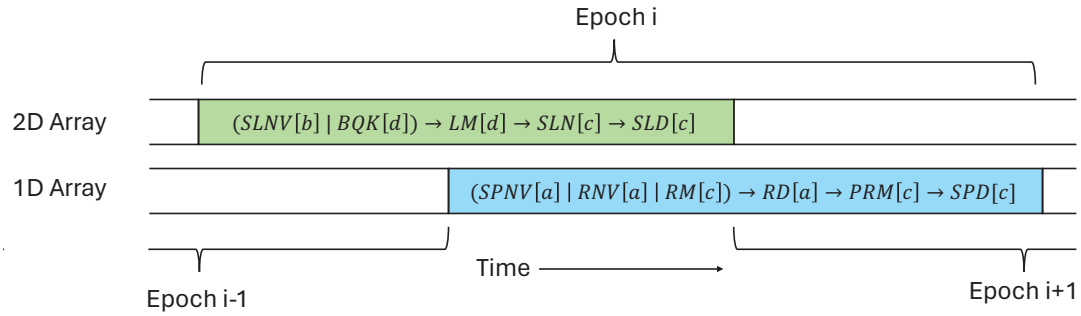


Fig. 4: FuseMax pipelining at a glance. Each tensor name (e.g., $SLNV$) corresponds to the Einsum used to compute that tensor (see Cascade 5). a, b, c and d denote tile-relative coordinates where $a < b < c < d$. If Epoch i produces tiles with coordinates a, b, c, d, Epoch $i+1$ produces tiles with identifiers $a+1, b+1, c+1, d+1$. 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 making up an epoch take almost the same number of cycles.

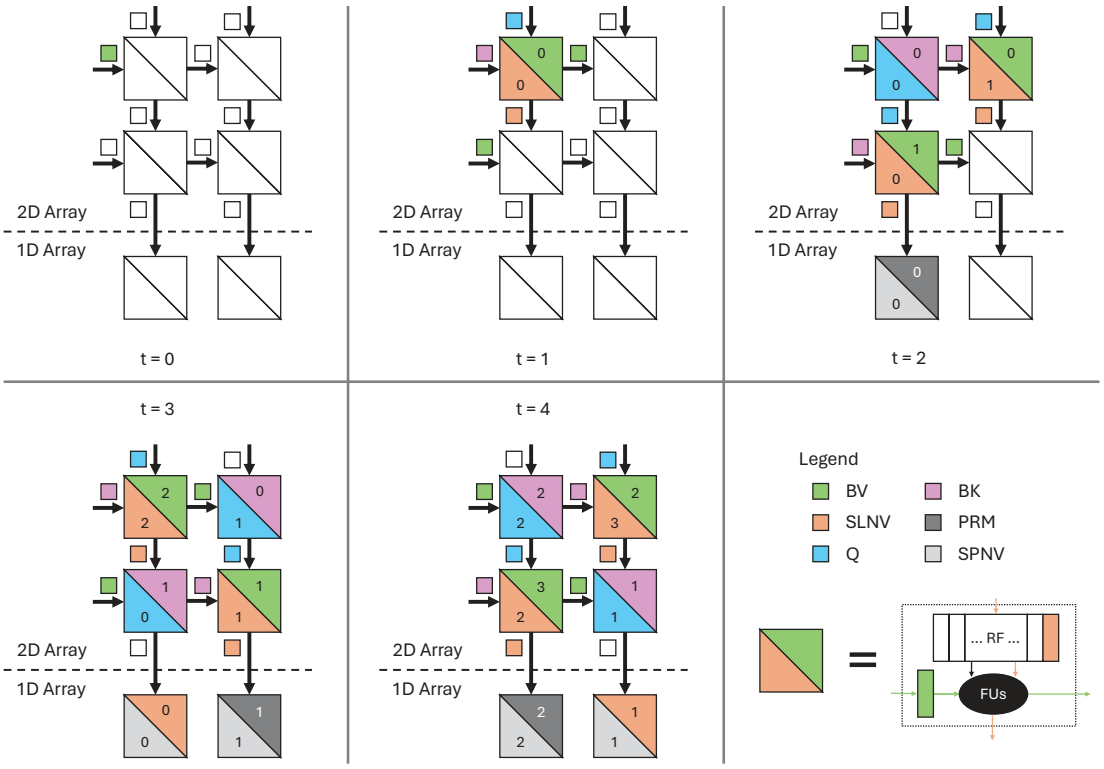


Fig. 5: Intial pipeline fill ($t = 0$ to $t = 2$) and steady-state ($t = 3$ and $t = 4$) for the intra-epoch interleaving of $SLNV|BQK$ and $SPNV|RNV$ to maximize 2D and 1D PE utilization, respectively, on a toy 2x2 array. Each color indicates a tensor and each number indicates a point in that tensor (e.g., the point BV_0 moves from the top left PE at $t = 1$ to the top right PE at $t = 2$). To reason about signal timing, we use input (but not output) latches for data in each PE, so moving data appears on output wires. Some stationary tensors (e.g., BQK) and Einsums (e.g., SLD) are omitted for clarity.

$SLNV$ and using software pipelining to hide the latency of the other compute, the GPU's loosely coupled threads require frequent synchronization to maintain correctness. FuseMax takes advantage of the tight coupling between the 2D and 1D arrays to statically schedule compute between the arrays, enabling high utilization across the board without sychronization.

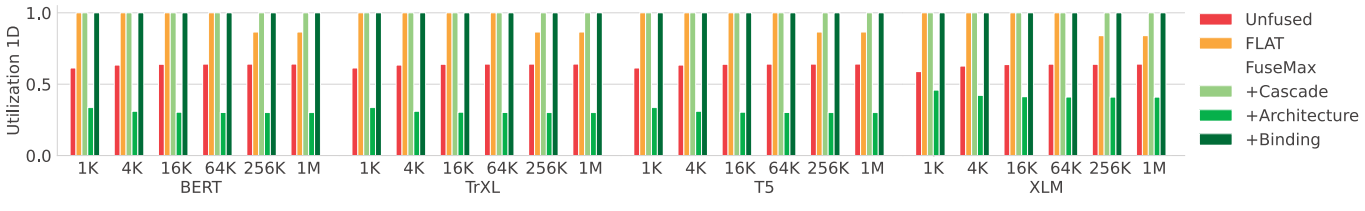
VI. Evaluation

In this section, we demonstrate how FuseMax's cascade, architecture, and binding work together to achieve improvements in both performance and energy relative to the state of the art, for both attention and end-to-end transformer inference.

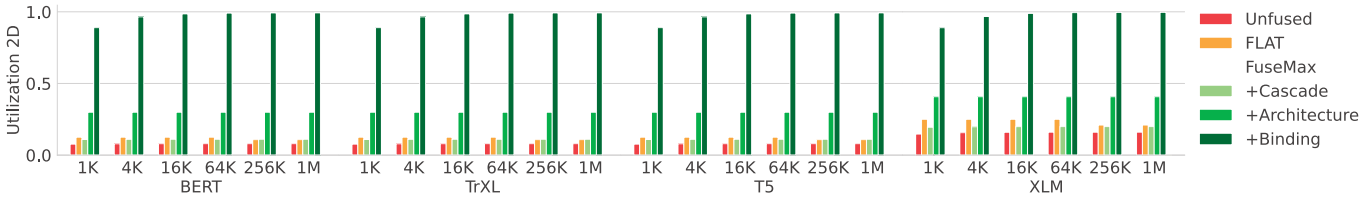
A. Experimental Set-Up

First, we present the experimental setup details common to all following subsections.

Workloads. We evaluate all accelerators and configurations using the same transformer models used by FLAT [28]: BERT-Base [17] (BERT), TrXL-wt103 [13] (TrXL), T5-small [49] (T5), and XLM [13]. We omit FlauBERT [31] because it uses the same hyperparameters as TrXL. We also note that though T5 is an encoder-decoder model, we only evaluate the encoder in this work. Following FLAT, we use a batch size $B = 64$ for all evaluations.



(a) 1D PE array utilization



(b) 2D PE array utilization

Fig. 6: Utilization of the different PE arrays on the unfused baseline, FLAT, and three configurations building up FuseMax.

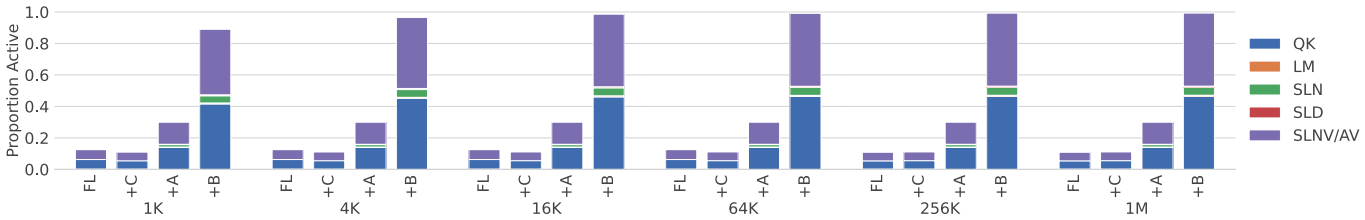


Fig. 7: 2D array utilization by Einsum across different configurations—FLAT (FL), +Cascade (+C), +Architecture (+A), and +Binding (+B)—and sequence lengths on BERT.

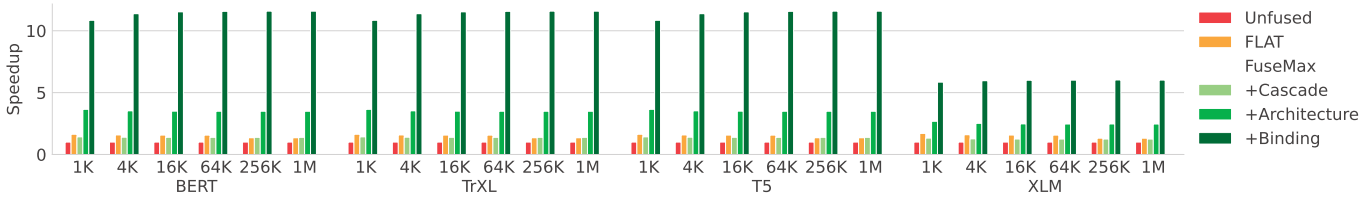


Fig. 8: Speedup of attention for FLAT and three configurations building up FuseMax over an unfused baseline.

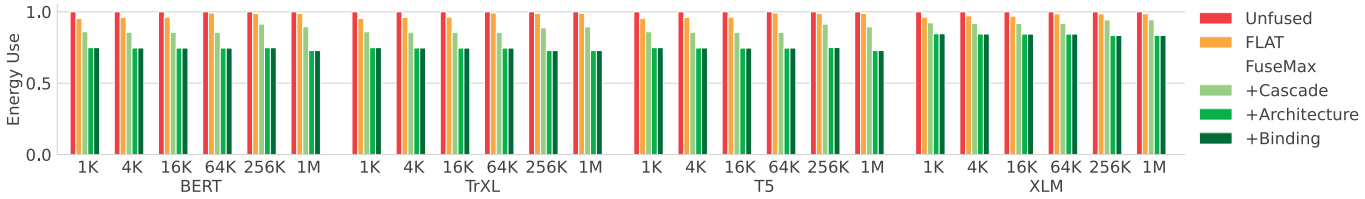


Fig. 9: Energy consumption of attention for FLAT and three configurations building up FuseMax over an unfused baseline.

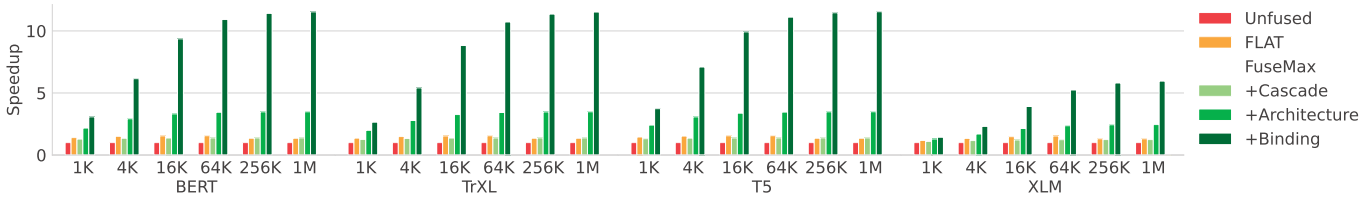


Fig. 10: Speedup of transformer inference on FLAT and three configurations building up FuseMax over an unfused baseline.

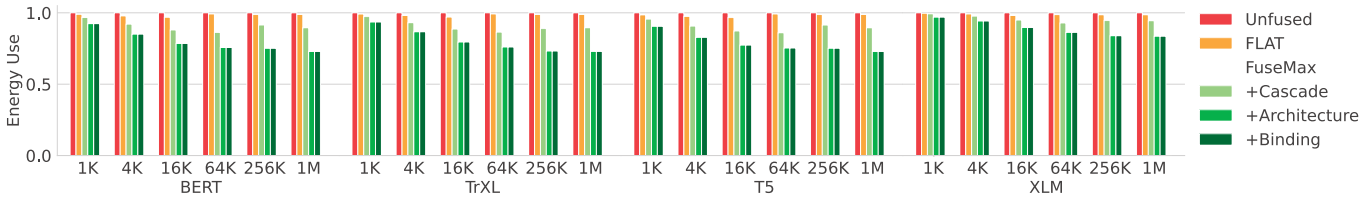


Fig. 11: Energy consumption of transformer inference on FLAT and three configurations building up FuseMax over an unfused baseline.

Modeling with Timeloop and Accelergy. We perform our evaluation using two tools for tensor algebra accelerator modeling and design space exploration: Timeloop [41] and Accelergy [56]. We use these tools to build models of the accelerator architectures at a 45nm technology node and evaluate each Einsum individually. Results from individual Einsums are combined using heuristics presented in prior work for evaluating full cascades [35]. Together, these tools allow us to evaluate execution time, energy, and area for all our designs. We perform floating-point division using the design in Xia et al. [59], scaled down to a 45nm technology node [56].

Unfused Baseline. We build the unfused baseline by combining the costs of three phases: QK (Einsum 22), the 3-pass softmax (Cascade 4), and AV (Einsum 24). Because this baseline is unfused, each phase can be scheduled independently, but proceed sequentially and require outputs to be written to memory between phases. We use Timeloop to search for efficient mappings to perform QK and AV. Additionally, we model the softmax for the unfused baseline by allowing the accelerator to load the M fibers of the input on-chip one-by-one (spilling if there is not enough space) before performing the compute. We model the memory traffic, compute, and energy required to perform all Einsums required for attention.

FLAT Baseline. Our main baseline is the state-of-the-art attention accelerator FLAT [28]. Though we started with the FLAT authors' original code, we found and corrected a number of bugs. Through private correspondence with the FLAT authors, we verified the bugs were indeed bugs. We also discovered a couple of larger conceptual errors, which the authors told us to avoid by restricting FLAT to only search through configurations without these issues.

Beyond correcting the FLAT codebase, we created and validated a Timeloop model that reproduces the FLAT authors' (corrected) code to within $< 1\%$ error. However, the FLAT codebase does not model the cost to perform the softmax. Specifically, their model ignores the cost of the data transfers required for the softmax (between any levels of the memory hierarchy) and uses 2^{30} 1D PEs for compute. When comparing FuseMax and FLAT in this work, we augment our Timeloop model to model softmax correctly per the 3-pass cascade implicitly assumed by FLAT using only 256 1D PEs.

FuseMax Configurations. To demonstrate the sources of the improvements achieved by FuseMax, we present three configurations, one associated with each of the major changes we propose: +Cascade uses the 1-pass cascade on the FLAT architecture, +Architecture adds the FuseMax architecture 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.

Hardware parameters. Figure 2 shows the selected hardware parameters. We chose the PE array dimension to match FLAT's cloud accelerator and then set the global buffer capacity so that the overall chip area was as close to FLAT's as possible. Also following FLAT, we use a 940 MHz frequency. We use Accelergy to model the area of both designs and find that FuseMax is 6.4% smaller.

B. *Evaluating Attention*

We now evaluate FuseMax to demonstrate the benefits it provides on the attention kernel by comparing it to the two baselines.

Utilization. Figure 6a shows the utilization of the 1D PE array when performing attention. FLAT's utilization drops for sequence lengths $\geq$ 256K—it becomes memory bandwidth limited because it must spill the QK and A tensors to memory. By using a 1-pass cascade (+Cascade), FuseMax's utilization becomes independent of sequence length. We also note that without the FuseMax binding (+Architecture), the 1D array is forced to stall and utilization drops. Adding in this binding (+Binding) enables FuseMax to fully utilize the 1D array again.

Similarly, Figure 6b shows the utilization of the 2D array. Because of the large amount of compute required for the softmax, most configurations achieve poor utilization of this array. In fact, because the 1-pass cascade increases the compute required, +Cascade's 2D 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. Both baselines achieve slightly higher utilization on XLM, which can be attributed to the higher intensity caused by a larger embedding dimension (E/F).

Figure 7 explores this phenomenon in more detail, breaking down the utilization by Einsum. FuseMax effectively hides both the costs of the memory traffic and softmax compute, allowing it to achieve high 2D array utilization while spending most of the cycles on the tensor products.

Speedup. Figure 8 shows that FuseMax achieves an average speedup of $10\times$ over the unfused baseline and $6.7\times$ over FLAT. We note FuseMax achieves lower speedup on XLM only because the baselines are able to achieve higher utilization of the 2D array on this transformer (Figure 6b).

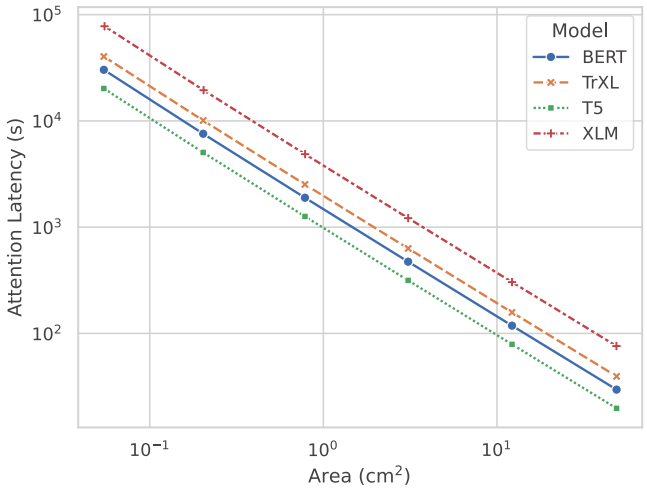


Fig. 12: Pareto-optimal curves for FuseMax at sequence length 256K.

Energy. Figure 9 shows that FuseMax uses 77% the energy of the unfused baseline and 79% the energy of FLAT.[6] The energy use of the unfused baseline and FLAT are dominated by the DRAM access energy, the global buffer access energy, and the QK and AV (Einsums 22 and 24) compute energy. FuseMax achieves its energy savings by significantly reducing the DRAM and global buffer access energies. In fact, $\geq 95\%$ of the energy used by FuseMax across all models and sequence lengths goes to the compute performed by the MACC functional units in the 2D array.

C. Evaluating Transformer Inference

To evaluate the benefits of FuseMax on end-to-end transformer inference, we include the other required linear layers (Section IV-A). We use Timeloop to search for optimal mappings for these linear layers and use the same mappings for all three accelerator configurations. The attention modeling remains the same as Section VI-B.

Speedup. Figure 10 shows the performance improvement achieved by FuseMax. Across the sequence lengths tested, FuseMax achieves an average speedup of $7.6\times$ over the unfused baseline and $5.3\times$ over FLAT. As discussed in Section IV-A, as sequence length grows, attention becomes a larger fraction of the total required compute. Therefore, at 1M tokens, FuseMax achieves an average $10\times$ speedup over the unfused baseline and $7.5\times$ speedup over FLAT.

Energy. Figure 11 shows the energy reduction achieved by FuseMax. Here, we see similar results: as attention becomes a larger fraction of the kernel, the energy reduction increases. FuseMax uses 82% of the unfused baseline and 83% of FLAT's energy during end-to-end inference.

D. Pareto-Optimality of FuseMax

We further observe that 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 generate a family of designs for efficient transformer inference.

[6]FLAT reports larger energy savings over the unfused baseline because it only reports energy associated with DRAM traffic during the tensor products.

VII. Related Work

Spatial architectures have been applied successfully to a variety of domains in academia [9], [10], [40], [45] and industry [3], [27]. Beyond FLAT [28] (discussed in the main body of the paper), TileFlow [62] is a framework for modeling and searching for efficient fused dataflows (including for attention) on spatial architectures. Though TileFlow does explore a broader space of dataflows than FLAT, even implementing the 2-pass softmax cascade (Section IV-E2), its dataflows remain softmax-compute limited. Recent work has explored the scheduling/compilation of a multi-Einsum kernels [21], [54], [62]. However, these works explore a limited set of transformations, making FuseMax's inter-Einsum interleaving not discoverable.

Quantization and sparsity have also been successfully applied to reduce the transformer inference compute and live footprint. We view these schemes as complementary to our work. GPTQ [20], AWQ [32], and LLM.int8() [16] quantize model weights to 4 or 8 bits without significant accuracy degradation. Outlier-aware quantization schemes like GOBO [60] and OliVe [22] quantize both weights and activations to a low-bit precision on specific hardware designs. SpAtten [53] prunes entire tokens and heads, while Sanger [34] and DOTA [46] use quantized or low-rank projected Q and K tensors to estimate which values of QK and A can be safely pruned. All of these algorithms are expressible as cascades of Einsums, and therefore, may be combined with FuseMax to improve performance and energy efficiency, though we leave their specification and implementation to future work.

VIII. Conclusion

This paper advanced the state of the art in spatial accelerator design for transformer inference. To do so, we expressed attention and its variants as cascades of Einsums. We used these cascades to reason about attention's characteristics, independent of its mapping/scheduling. Using these principles, we proposed FuseMax—an accelerator that uses deep fusion and fine-grain pipelining to map attention onto a spatial architecture. FuseMax achieves $\sim$ 100% utilization of both PE arrays, demonstrating $6.7\times$ speedup over the prior state-of-the-art (FLAT) using 79% of the energy on attention and $5.3\times$ speedup over FLAT using 83% of the energy on end-to-end inference.

Our work shows that cascades of Einsums provide a powerful abstraction for representing and analyzing domain-specific kernels. Future work may explore their application to other attention variants (e.g., those exploiting quantization and sparsity) or even other domains (e.g., fully homomorphic encryption, scientific computing, relational algebra, etc.). Doing so enables mapping-agnostic analysis and may elucidate previously undiscovered cascades and schedules for these algorithms.

ACKNOWLEDGMENT

We thank the anonymous MICRO and MLArchSys reviewers for their feedback on submitted versions of the 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 would also like to 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.

APPENDIX

A. Abstract

In this artifact, we provide Timeloop and Accelergy models of the accelerator FuseMax, an accelerator for encoder-style transformer inference. For ease-of-use, we provide a Docker container and a set of Jupyter notebooks through which to run the experiments. This artifact can be evaluated on an x86-84 machine with 5 GB of disk space.

B. Artifact check-list (meta-information)

- **Algorithm:** Timeloop/Accelergy model of the FuseMax accelerator and the baselines it was evaluated against
- **Program:** Python, Timeloop, Accelergy
- **Run-time environment:** Docker, Jupyter
- **Hardware:** x86-64 machine
- **Output:** Plots generated from scripts
- **Experiments:** Modeling of the five different accelerator design points via Timeloop and Accelergy models
- **How much disk space required (approximately)?:** 5GB
- **How much time is needed to prepare workflow (approximately)?:** 20 minutes
- **How much time is needed to complete experiments (approximately)?:** 9 hours
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** Provided after evaluation

C. Description - How to access

The artifact is hosted on Github at https://github.com/FPSG-UIUC/micro24-fusemax-artifact. Following the instructions in this repository will allow you to install the relevant dependences, run the experiments, and display the graphs. System requirements can be found at https://github.com/FPSG-UIUC/micro24-fusemax-artifact/blob/main/README.md#system-requirements.

D. Installation

Installation instructions can be found at https://github.com/FPSG-UIUC/micro24-fusemax-artifact/blob/main/README.md#installation.

E. Evaluation

Evaluation instructions can be found at https://github.com/FPSG-UIUC/micro24-fusemax-artifact/blob/main/README.md#run-experiments.

F. Expected Results

Graphs will be displayed within the Jupyter notebook and/or found in `workspace/outputs/generated/<timestamp or default>/figs/`. They can be compared with Figures 6-12 in the paper or the corresponding figures in `workspace/outputs/pregenerated/figs/`.

G. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae

REFERENCES

[1] "Tensor network contractions," ser. Lecture Notes in Physics, vol. 964. Springer Cham, 2020.
[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning," in *OSDI'16*.
[3] AWS. (2024) Trainium architecture. [Online]. Available: https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/arch/neuron-hardware/trainium.html
[4] A. Baevski, H. Zhou, A. Mohamed, and M. Auli, "wav2vec 2.0: a framework for self-supervised learning of speech representations," in *NIPS '20*.
[5] J. S. Bridle, "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition," in *NATO Neurocomputing*, 1989.
[6] S. Chen, S. Huang, S. Pandey, B. Li, G. R. Gao, L. Zheng, C. Ding, and H. Liu, "E.t.: Re-thinking self-attention for transformer models on gpus," in *SC'21*.
[7] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM Sigplan Notices*, 2014.
[8] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A survey of accelerator architectures for deep neural networks," *Engineering*, 2020.
[9] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA'16*.
[10] ——, "Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks," in *ArXiv*, 2018.
[11] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *MICRO'14*.
[12] J. Choi, H. Li, B. Kim, S. Hwang, and J. H. Ahn, "Accelerating transformer networks through recomposing softmax layers," in *IISWC'22*.
[13] A. Conneau and G. Lample, "Cross-lingual language model pretraining," in *NIPS'19*.
[14] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," in *ArXiv*, 2023.
[15] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," in *ArXiv*, 2022.
[16] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Llm.int8(): 8-bit matrix multiplication for transformers at scale," in *ArXiv*, 2022.
[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *NAACL'19*.
[18] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *ICLR'21*.
[19] I. S. Duff, M. A. Heroux, and R. Pozo, "An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum," in *TOMS'02*.
[20] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate post-training compression for generative pretrained transformers," in *ArXiv*, 2022.
[21] M. Gilbert, Y. N. Wu, A. Parashar, V. Sze, and J. S. Emer, "Looptree: Enabling exploration of fused-layer dataflow accelerators," in *ISPASS'23*.

[22] C. Guo, J. Tang, W. Hu, J. Leng, C. Zhang, F. Yang, Y. Liu, M. Guo, and Y. Zhu, "Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization," in *ISCA'23*.

[23] C. Hong, Q. Huang, G. Dinh, M. Subedar, and Y. S. Shao, "DOSA: Differentiable model-based one-loop search for DNN accelerators," in *MICRO'23*.

[24] C. Hooper, S. Kim, H. Mohammadzadeh, M. W. Mahoney, Y. S. Shao, K. Keutzer, and A. Gholami, "Kvquant: Towards 10 million context length llm inference with kv cache quantization," in *ArXiv*, 2024.

[25] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjølstad, "The sparse abstract machine," in *ASPLOS'23*.

[26] W.-N. Hsu, B. Bolte, Y.-H. H. Tsai, K. Lakhotia, R. Salakhutdinov, and A. Mohamed, "Hubert: Self-supervised speech representation learning by masked prediction of hidden units," *TASLP'21*.

[27] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *ISCA'17*.

[28] S.-C. Kao, S. Subramanian, G. Agrawal, A. Yazdanbakhsh, and T. Krishna, "Flat: An optimized dataflow for mitigating attention bottlenecks," in *ASPLOS'23*.

[29] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach," in *MICRO'19*.

[30] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," in *TOMS'79*.

[31] H. Le, L. Vial, J. Frej, V. Segonne, M. Coavoux, B. Lecouteux, A. Allauzen, B. Crabbé, L. Besacier, and D. Schwab, "Flaubert: Unsupervised language model pre-training for french," in *ArXiv*, 2019.

[32] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, "Awq: Activation-aware weight quantization for llm compression and acceleration," in *MLSys'24*.

[33] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *ICCV'21*.

[34] L. Lu, Y. Jin, H. Bi, Z. Luo, P. Li, T. Wang, and Y. Liang, "Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture," in *MICRO'21*.

[35] N. Nayak, T. Odemuyiwa, S. Ugare, C. W. Fletcher, M. Pellauer, and J. Emer, "Teaal: A declarative framework for modeling sparse tensor accelerators," in *MICRO'23*.

[36] P. Nilsson, A. U. R. Shaik, R. Gangarajaiah, and E. Hertz, "Hardware implementation of the exponential function using taylor series," in *NORCHIP'14*.

[37] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, "The design process for google's training chips: Tpuv2 and tpuv3," *IEEE Micro*, 2021.

[38] T. O. Odemuyiwa, H. Asghari-Moghaddam, M. Pellauer, K. Hegde, P.-A. Tsai, N. Crago, A. Jaleel, J. D. Owens, E. Solomonik, J. Emer, and C. Fletcher, "Accelerating sparse data orchestration via dynamic reflexive tiling," in *ASPLOS'23*.

[39] T. O. Odemuyiwa, J. S. Emer, and J. D. Owens, "The EDGE language: Extended general einsums for graph algorithms," in *ArXiv*, 2024.

[40] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, "Efficient spatial processing element control via triggered instructions," *IEEE Micro*, 2014.

[41] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *ISPASS'19*.

[42] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: an imperative style, high-performance deep learning library," in *NIPS'19*.

[43] M. Pellauer, J. Clemons, V. Balaji, N. C. Crago, A. Jaleel, D. Lee, M. O'Connor, A. Parashar, S. Treichler, P. Tsai, S. W. Keckler, and J. S. Emer, "Symphony: Orchestrating sparse and dense tensors with hierarchical heterogeneous processing," in *TOCS'23*.

[44] S. Pichai and D. Hassabis. (2024) Our next-generation model: Gemini 1.5. [Online]. Available: https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/#context-window

[45] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," in *SIGARCH Computer Architecture News*, 2017.

[46] Z. Qu, L. Liu, F. Tu, Z. Chen, Y. Ding, and Y. Xie, "Dota: detect and omit weak attentions for scalable transformer acceleration," in *ASPLOS '22*.

[47] M. N. Rabe and C. Staats, "Self-attention does not need $o(n^2)$ memory," in *ArXiv*, 2022.

[48] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:49313245

[49] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," in *JMLR'20*.

[50] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "Flashattention-3: Fast and accurate attention with asynchrony and low-precision," in *ArXiv*, 2024.

[51] V. Sze, Y. Chen, T. Yang, and J. S. Emer, *Efficient Processing of Deep Neural Networks*, ser. Synthesis Lectures on Computer Architecture. Springer, 2020.

[52] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS'17*.

[53] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *HPCA'21*.

[54] J. Wang, L. Guo, and J. Cong, "Autosa: A polyhedral compiler for high-performance systolic arrays on fpga," in *FPGA'21*.

[55] J. Won, C. Hong, C. Mendis, J. Emer, and S. Amarasinghe, "Unified convolution framework: A compiler-based approach to support sparse convolutions," in *MLSys'23*.

[56] Y. N. Wu, J. S. Emer, and V. Sze, "Accelergy: An architecture-level energy estimation methodology for accelerator designs," in *ICCAD'19*.

[57] Y. N. Wu, P. Tsai, S. Muralidharan, A. Parashar, V. Sze, and J. S. Emer, "HighLight: Efficient and flexible DNN acceleration with hierarchical structured sparsity," in *MICRO'23*.

[58] Y. N. Wu, P.-A. Tsai, A. Parashar, V. Sze, and J. S. Emer, "Sparseloop: An analytical approach to sparse tensor accelerator modeling," in *MICRO'22*.

[59] J. Xia, W. Fu, M. Liu, and M. Wang, "Low-latency bit-accurate architecture for configurable precision floating-point division," in *Applied Sciences*, 2021.

[60] A. H. Zadeh, I. Edo, O. M. Awad, and A. Moshovos, "Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference," in *MICRO'20*.

[61] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication," in *ASPLOS'21*.

[62] S. Zheng, S. Chen, S. Gao, L. Jia, G. Sun, R. Wang, and Y. Liang, "Tileflow: A framework for modeling fusion dataflow via tree-based analysis," in *MICRO'23*.