# Faster Neighborhood Attention: Reducing the $\mathcal{O}(n^2)$ Cost of Self Attention at the Threadblock Level

**Ali Hassani**[1], **Wen-mei Hwu**[2,3], **Humphrey Shi**[1,3]
[1]SHI Labs @ Georgia Tech, [2]NVIDIA, [3]UIUC

## Abstract

Neighborhood attention reduces the cost of self attention by restricting each token's attention span to its nearest neighbors. This restriction, parameterized by a window size and dilation factor, draws a spectrum of possible attention patterns between linear projection and self attention. Neighborhood attention, and more generally sliding window attention patterns, have long been bounded by infrastructure, particularly in higher-rank spaces (2-D and 3-D), calling for the development of custom kernels, which have been limited in either functionality, or performance, if not both. In this work, we aim to massively improve upon existing infrastructure by providing two new methods for implementing neighborhood attention. We first show that neighborhood attention can be represented as a batched GEMM problem, similar to standard attention, and implement it for 1-D and 2-D neighborhood attention. These kernels on average provide 895% and 272% improvement in full precision runtime compared to existing naive CUDA kernels for 1-D and 2-D neighborhood attention respectively. We find that aside from being heavily bound by memory bandwidth, certain inherent inefficiencies exist in all unfused implementations of neighborhood attention, which in most cases undo their theoretical efficiency gain. Motivated by the progress made into fused dot-product attention kernels, we developed fused neighborhood attention; an adaptation of fused dot-product attention kernels that allow fine-grained control over attention across different spatial axes. Known for reducing the quadratic time complexity of self attention to a linear complexity, neighborhood attention can now enjoy a reduced and constant memory footprint, and record-breaking half precision runtime. We observe that our fused implementation successfully circumvents some of the unavoidable inefficiencies in unfused implementations. While our unfused GEMM-based kernels only improve half precision performance compared to naive kernels by an average of 548% and 193% in 1-D and 2-D problems respectively, our fused kernels improve naive kernels by an average of 1759% and 958% in 1-D and 2-D problems respectively. These improvements translate into up to 104% improvement in inference and 39% improvement in training existing models based on neighborhood attention, and additionally extend its applicability to image and video perception, as well as other modalities. Our work is open-sourced at `https://github.com/SHI-Labs/NATTEN/`.

## 1 Introduction

Inarguably among the most highly utilized and influential primitives in modern deep learning, attention has long been cited for its complexity and memory footprint, especially when the query and context sets are identical (self attention). For years since its adoption in deep learning [23], the most common implementation of attention was through two batched GEMM (General Matrix-Matrix Multiplication) operations, sometimes referred to as "BMM-style" attention. This implementation stores attention weights to global memory, which can become a bottleneck in both speed and memory

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

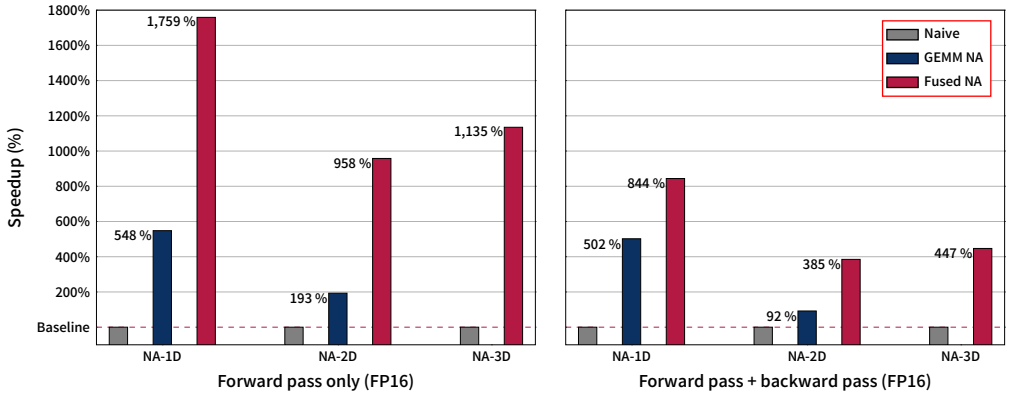


Figure 1: **Overview of average improvement in speed on A100 from our proposed implementation.** Baseline is the set of naive CUDA kernels introduced in Neighborhood Attention Transformer [9]. GEMM-based NA improves 1-D problems by an average of 548% (forward pass) and 502% (forward + backward), and 2-D problems by an average of 193% (forward pass) and 92% (forward + backward). GEMM-based NA does not implement 3-D problems yet. Fused NA boosts performance further and improves 1-D problems by an average of 1759% (forward pass) and 844% (forward + backward), and 2-D problems by an average of 958% (forward pass) and 385% (forward + backward), and 3-D problems by an average of 1135% (forward pass) and 447% (forward + backward).

footprint. As the number of tokens grow, the number of attention weights grow as well, and the problem gets bounded by global memory bandwidth and capacity.

Over the past few years, some works proposed attention implementations in which attention weights are kept in on-chip memory (shared memory or register file) instead, until the second matrix multiplication is performed and the resulting attention outputs are written directly to global memory [18, 6]. These implementations, known as fused or memory-efficient attention, reduce the number of global memory accesses in addition to global memory usage, and successfuly turn dot product attention into a compute-bound problem at scale. Thanks to the first open-source implementation, Flash Attention [6], these fused attention kernels have started replacing the standard BMM-style implementations in many deep learning frameworks and inference engines such as PyTorch [16].

Orthogonal to these efforts, many have sought to address the quadratic complexity of self attention, which can become a significant bottleneck in vision models more quickly. Neighborhood attention [9] is one such method in which each query token is restricted to only interact with its nearest neighboring context tokens. In most cases, this pattern creates a sliding window pattern, like that of the discrete convolution operator heavily employed in vision models. This restriction can similarly be parameterized by a window size and dilation factor, and reduces the quadratic complexity of self attention down to a linear complexity. This approach is, however, very difficult to implement at the tensor library or deep learning framework level. Tensor views can represent sliding window attention [19], but not the neighborhood attention pattern. In addition, standard GEMM implementations typically do not support such tensor views in higher-rank/multi-dimensional spaces (2-D and 3-D) without explicit copying into contiguous tensors, which in practice undoes the theoretical efficiency gain from the reduced attention complexity. As a result, neighborhood attention was proposed along with an extension carrying naive CUDA kernels [9] implementing the operation. While those kernels provided competitive FP32 performance in eager mode inference, and in some cases even FP16/BF16 performance, they fall short of general adoption in larger scale experiments. In addition, fused attention implementations, such as Flash Attention, effectively eliminate the $\mathcal{O}(n^2)$ memory footprint in self attention, while also reducing runtime significantly [6], making subquadratic attention patterns that are only possible to implement "BMM-style" less practical.

In this work, we present two new classes of neighborhood attention kernels: GEMM-based BMM-style kernels (GEMM NA), and fused kernels (Fused NA), which are aimed at providing significantly improved infrastructure for neighborhood attention. We first show that neighborhood attention, and by

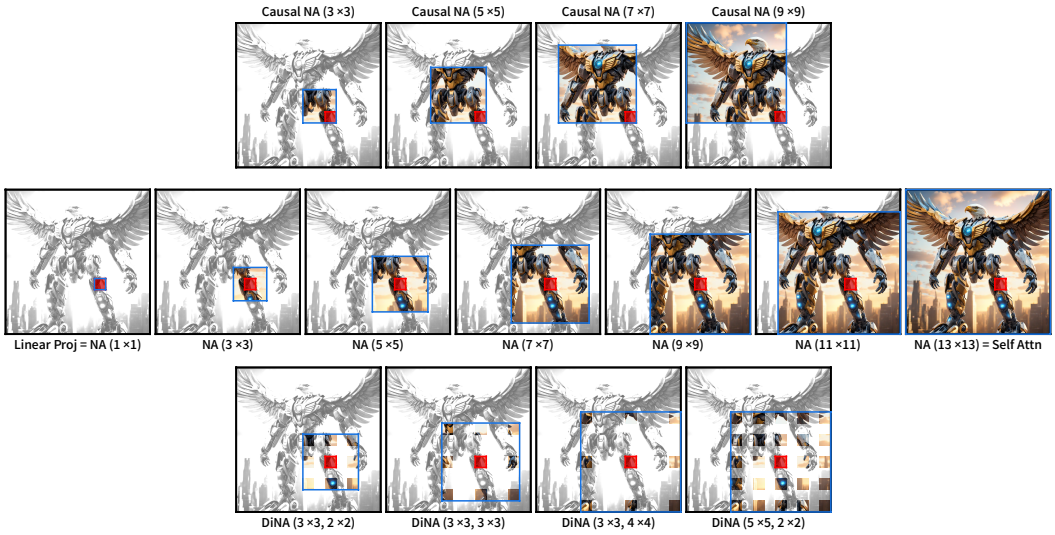


Figure 2: **Illustration of the spectrum of possible attention patterns provided by neighborhood attention.** Neighborhood attention only attempts to center the query token (red) within the context window (blue), unlike sliding window attention [19] which forces it. Neighborhood attention with window size 1 is equivalent to linear projection ("no attention"). Neighborhood attention approaches self attention as window size grows, and matches it when equal to input size. Dilation introduces sparse global context, and causal masking prevents interaction between query tokens that have a smaller coordinate than neighboring context tokens along the corresponding mode. Window size, dilation, and whether or not causally masked, can be defined per mode/axis.

extension sliding window attention, both of which are GEMV (General Matrix-Vector Multiplication) problems, can be expressed as GEMM problems with space-aware tiling and gather/scatter fusion. This would allow implementing such attention patterns with performance-optimized GEMM primitives, which can also utilize specialized hardware components such as NVIDIA's Tensor Cores. We then extend the same logic to fused attention kernels by removing all assumptions that the token mode ("space") is rank-1 (single-axis). We write specializations that support higher-rank/multi-dimensional spaces, such as 2-D and 3-D. This, in theory, allows any fused attention kernel to be modified to accommodate token spaces of any rank. In addition, part of the logic is evaluated at compile time, resulting in less overhead. Finally, the structural simplicity of the resulting fused neighborhood attention kernels allows for easily adding features such as varying window sizes / dilation values across ranks/axes, causal masking, and more.

## 2 Related works

Attention being adopted as a deep learning primitive is largely owed to the Transformer architecture [23], which despite its original application in machine translation rose to the position of being the predominant deep learning architecture. Its design and use of the attention operator have been extended to many other applications and modalities [15, 7, 1, 17]. Attention is defined as an operation between a two sets of vectors: a query set and a context set. The two undergo linear projections, with the latter projected into a set of key and value pairs. Scaled dot product of query and key vectors, $A$, is mapped into a probability distribution through the softmax operator, which produces the final attention weights, $P$. The output is a set of vectors, each derived from the weighted sum of all value vectors according to the query vector's attention weights. It can be expressed as follows:

$$Attention(Q,K,V) = \overbrace{softmax\left(\underbrace{\frac{QK^T}{\sqrt{d}}}_{A}\right)}^{P} V, \tag{1}$$

where $Q$, $K$, and $V$ are matrices of query, key, and value vectors as rows respectively, $\sqrt{d}$ is the scale term, and $d$ is the number of dimensions for which the dot product is computed (number of columns in $Q$ and $K$). Dot product self attention, or simply, self attention, is a special case in which the query and context sets are identical. This means for a set of $n$ input vectors, the attention weights matrix, $P$, is $\in \mathbb{R}^{n \times n}$, incurring an $\mathcal{O}(n^2)$ time and space complexity. In addition, the softmax term requires column-wise reduction over the attention weight matrix, making kernel fusion more challenging.

Nevertheless, fused attention kernels successfully eliminate the $\mathcal{O}(n^2)$ global memory footprint, which makes self attention finally bound by compute and not memory bandwidth. These two achievements paved the way for the scaling and application of attention across modalities. To our knowledge, the first open-source implementation of a fused multi-headed attention (FMHA) kernel was contributed to the NVIDIA Apex [1] project by Young-Jun Ko, which was primarily used for accelerating inference of Transformer-based language models. As a result of that, it was heavily limited in terms of supported models and problem sizes, as it was performing a full softmax reduction step within the kernel. On the other hand, Milakov and Gimelshein [14] presented a technique for computing partial softmax statistics, over which we can perform a final reduction step and derive exact softmax results. This method makes the fusion of attention kernels more practical, because they would no longer be required to compute a full row of attention weights before proceeding to perform the second matrix multiplication. Dao et al. [6] presented and open-sourced Flash Attention, which utilizes online softmax in order to create a performant and generic fused attention implementation. Outperforming BMM-style implementations available in both training and inference, Flash Attention was quickly adopted by many frameworks such as PyTorch [16], and further improved for the NVIDIA Ampere [5] and Hopper architectures [20].

Parallel to these efforts, many proposed restricted self attention patterns, in which context is restricted to a subset in order to generate fewer attention weights, which in turn reduces the $\mathcal{O}(n^2)$ time and space complexity. Stand-alone self attention (SASA) [19] is a simple 2-dimensional sliding window attention pattern, which was shown to effectively replace convolution operations in ResNet [10] variants. Noting challenges in implementing such patterns without incurring additional overhead from tensor copies and expansion, the authors later moved away from explicit sliding window attention patterns to alternatives that relaxed the sliding window movement in HaloNet [22]. In addition to these works, sliding window attention patterns in 1-dimensional spaces has been explored in language, in works such as Sparse Transformers [4], Longformer [2], BigBird [26], and more recently, Mistral [11]. Neighborhood attention [9, 8] is the practice of restricting the context of each token to its nearest neighbors, which in many cases behaves like a sliding window pattern, with the exception of corner cases in which the query cannot be centered in a sliding window. Per definitions from SASA [19] and Longformer [2], the sliding context window can go out of bounds, in which case the attention weights corresponding to out-of-bounds tokens are masked. This means tokens close to spatial bounds interact with fewer context tokens. This difference allows neighborhood attention to approach self attention as window size grows. In addition, neighborhood attention defines a dilation factor [8], where the number of such corner cases only increase. Fig. 2 depicts possible attention patterns for a single token under different neighborhood attention parameters. Facing similar implementation challenges as previous works [19], neighborhood attention was implemented with naive CUDA kernels packaged as a PyTorch extension, named $\mathcal{N}ATTEN$. While those kernels have accelerated research in this direction, they were simply not intended to fully utilize the underlying hardware. The only exception is the tiled kernels, which are somewhat better optimized, but only apply to a fraction of common use cases, and are not extensible. In addition, with the rise of fused attention kernels such as Flash Attention [6], such implementations which are not performance-optimized and heavily memory-bandwidth-bound, can hardly compete in terms of performance and memory footprint.

To address these challenges, we present two new implementations and integrate them into $\mathcal{N}ATTEN$, aiming to accelerate all neighborhood attention applications, reduce their existing memory overhead, and extend existing functionality. We first simplify the operations that implement neighborhood attention's forward and backward pass into 3 primary operators, and show each can be implemented as batched GEMM kernels with a fused gather/scatter operation. We then point out key limitations in unfused neighborhood attention implementations that would prevent them from achieving competitive performance compared to standard BMM-style attention implementations (in more memory-bandwidth-bound cases.) Motivated by this, and the progress made in fused attention kernels, we propose fused neighborhood attention, which directly extends our batched GEMM

[1] https://github.com/NVIDIA/apex

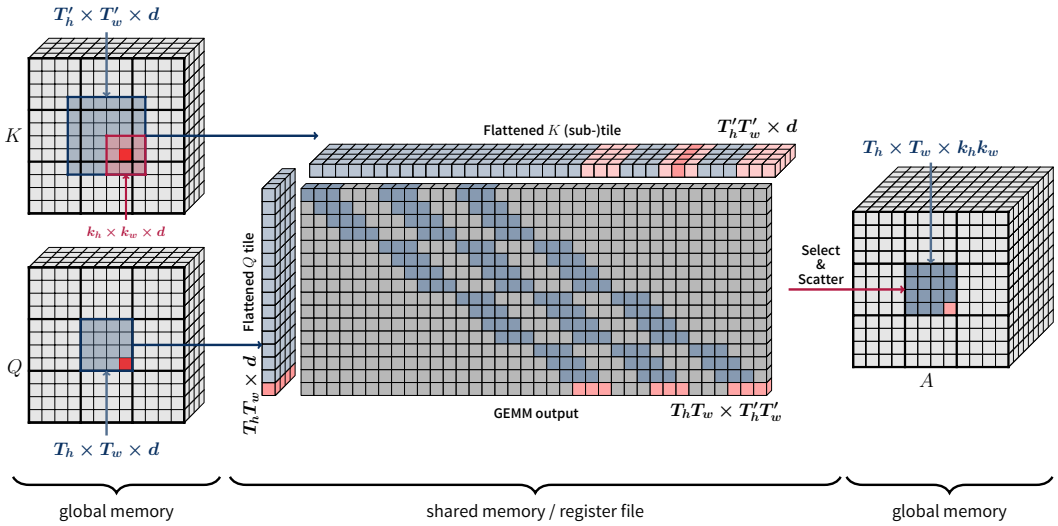


Figure 3: **Illustration of our GEMM-based implementation of the 2-D PN operation.** Input tensors $Q$ and $K$ are tiled according to their 2-D spatial layout. $Q$ is tiled with a static tile shape, $T_h \times T_w$. $K$ is tiled with a haloed shape of the $Q$ tile, $T'_h \times T'_w$, which is a function of the attention window size ($k_h \times k_w$) and the $Q$ tile coordinates. Once tiles are moved into local memory, they are viewed in matrix layout, and a $T_h T_w \times T'_h T'_w \times d$ shaped GEMM is computed ($d$ is embedding dim). Once done, the tile of dot products with shape $T_h T_w \times T'_h T'_w$ is scattered into valid attention weights of shape $T_h \times T_w \times k_h k_w$.

methodology. Since our main objectives are efficiency, Tensor Core utilization, and performance optimization, and our methodology requires significant flexibility in the programming model, we implement both approaches in CUDA C++ using NVIDIA's CUTLASS [21] framework. We show that the batched GEMM kernels can successfully outperform most existing $\mathcal{N}ATTEN$ kernels in performance, and that our fused kernels can outperform our batched GEMM kernels while reducing the memory footprint.

## 3 Methodology

Herein we describe three primary operations (excluding softmax) that are required to implement a full neighborhood attention forward and backward pass. We then show that each operation can be expressed as a batched GEMM problem, as long as tiling is done according to the underlying spatial rank, and attention weights are scatter/gathered. However, we find that scatter/gather is a major bottleneck for all unfused implementations of neighborhood attention, limiting their low-precision performance specifically on more recent architectures (Ampere and later.) We then introduce our fused neighborhood attention (FNA) formulation, which builds on our batched GEMM formulation and tiles according to the underlying spatial rank. This approach no longer requires scatter/gathering of attention weights to/from global memory by definition, and thereby circumvents the aforementioned bottleneck and successfully boosts lower-precision performance on modern architectures.

### 3.1 Operators

A standard BMM-style attention forward pass (excluding softmax) is comprised of two operations: $QK^T$, which produces pre-softmax attention weights ($A$), and $PV$, which applies post-softmax attention weights ($P$) to values ($V$). These operations are different due to layout differences in the matrix multiplications (note that $K$ is transposed, $V$ is not). [2]

In the case of neighborhood attention, and sliding window attention in general, these will become General Matrix-Vector Multiplication (GEMV) problems. In $QK^T$, each query token (vector) is multiplied by its neighboring or surrounding key tokens (matrix), and in $PV$, the set of attention

[2] $QK^T$ is a TN-layout and $PV$ is a TT-layout GEMM in BLAS.

weights corresponding to each query token (vector) is multiplied by corresponding value tokens (matrix). Given that some of these operations can be reused in the backward pass, we dub the $QK^T$ operation "Pointwise-Neighborhood" (PN) and the $PV$ operation "Neighborhood-Neighborhood" (NN). PN can compute the gradient for post-softmax attention weights ($\nabla P$) when operating on the output gradient instead of $Q$, and $V$ instead of $K$. Similarly, NN can compute the gradient for $Q$ ($\nabla Q$) when operating on the pre-softmax attention gradient ($\nabla A$) instead of $A$ and $K$ instead of $V$. We define a third operator, which can compute gradients for both $K$ and $V$: Inverse-Neighborhood (IN). This operation is very similar to NN, but differs in gather pattern, as well as the number of attention weights. IN may require loading more attention weights for every token, because unlike in self attention, the relationship between query and context tokens in neighborhood attention is not commutative. In other words, query token at coordinate $i$ attending to context token at coordinate $j$ does not imply that query token at coordinate $j$ attends to context token at coordinate $i$.

BMM-style implementations of standard self attention have a clear edge over neighborhood and sliding window attention implementations, because they are GEMM problems and by extension not as bound by memory bandwidth as the latter, all of which are GEMV problems. In addition, GEMV problems cannot effectively utilize matrix multiply and accumulate (MMA) accelerators, such as Tensor Cores. We aim to minimize this issue by formulating all three operators as batched GEMM problems with scatter/gather fusion, in order to better utilize modern hardware accelerators.

### 3.2 Batched GEMM NA

We transform the aforementioned GEMV problems into batched GEMMs with scatter/gather fusion. At an abstract level, implementations of GEMM-based neighborhood attention predicate the execution of tiled MMAs on whether any of the rows in the query tile interact with at least one of the rows in the context tile, given the context window size, dilation, and other masking-related parameters. We propose modifying a CUTLASS GEMM as follows in order to implement PN, NN, and IN:

1. GEMM tiling is done according to the original multi-dimensional layout of the token mode in QKV. For example, if the attention problem is 1-D, query and context tensors are tiled along the sequence into tiles of size 64, for a 2-D problem, the token mode, which is comprised of height and width, are tiled by a 2-D tiler of the same size, like $8 \times 8$.
2. Predication logic, and global pointer iterators and accessors are modified to iterate according to the original layout in global memory instead of assuming a standard rank-2 matrix layout.
3. Attention weights are required to be scattered to and gathered from global memory, which in 16-bit or lower precision cannot be copied asynchronously (with `LDGSTS`), which breaks pipelining in those kernels on modern architectures. This is because the minimum transaction size for `LDGSTS` is 32 bits.

We implemented these concepts by extending implicit GEMM (convolution) in CUTLASS (2.X API) into kernels that compute the three neighborhood attention operators in 1-D and 2-D. Fig. 3 shows an illustration of the 2-D GEMM-based PN kernel. The first change is relatively inexpensive, but the second change incurs additional predication and indexing logic that can result in additional overhead and register pressure. The final change is a major bottleneck, and leads to lower-precision kernels (FP16/BF16) providing little to no improvement compared to their full precision (FP32/TF32) counterparts. NN and IN suffer from this issue more significantly, because gathering attention weights (`LDG`) breaks pipelined kernels on Ampere, since they load GEMM operands asynchronously (`LDGSTS`), which has a minimum transaction size of 32 bits. This forces our FP16/BF16 GEMM-based kernels to fall back to global loads (`LDG`), which significantly impacts achievable runtime. To our knowledge, this issue is unavoidable in most cases, and will continue to be a bottleneck as long as attention weights are stored in global memory.

### 3.3 Fused NA

We extend our methodology for implementing neighborhood attention operators using batched GEMM kernels to fused attention kernels like Flash Attention [6]. This is not only motivated by the potential to reduce runtime and memory footprint, and potentially making neighborhood attention actually bound by compute, but also to circumvent the bottleneck in the batched GEMM and naive kernels: scatter/gathering attention weights to/from global memory. Since attention weights are only computed

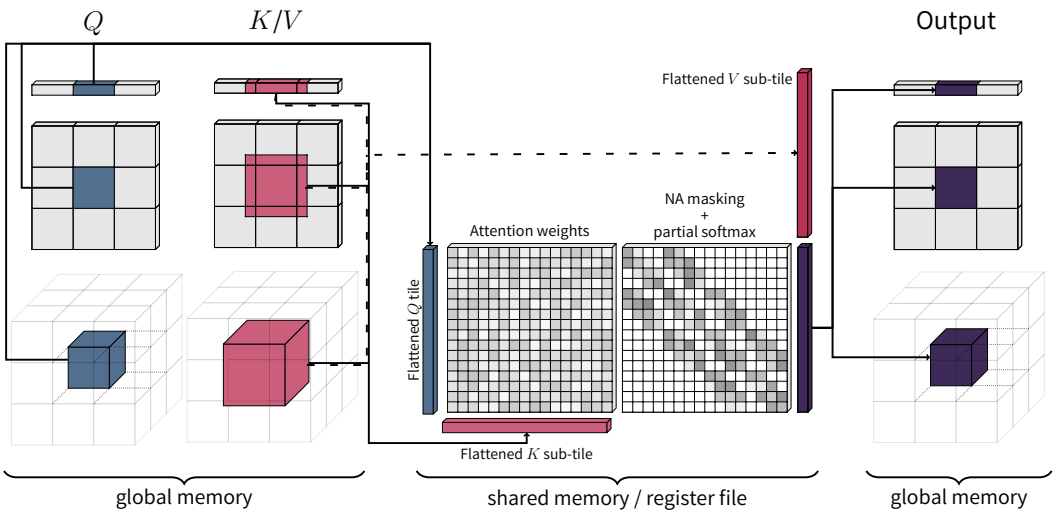


Figure 4: **A simplified illustration of fused neighborhood attention.** $Q$ and $KV$ tensors are tiled according to their spatial layout (1-D, 2-D, 3-D), with the latter haloed to include the entire neighborhood for all corresponding queries in the query tile. Resulting attention weights from the first GEMM are masked according to neighborhood attention parameters, before undergoing online softmax scaling, and going through the second GEMM with the corresponding value sub-tile.

at the threadblock level and never fully stored in global memory in fused kernels, the bottleneck will simply cease to exist. We started off with xFormers FMHA [13], a fused multi-headed attention kernel based on the CUTLASS 2.X API, which can target architectures even older than Ampere (Maxwell, SM50; Volta, SM70; and Turing, SM75.) By carefully applying our methodology for space-aware tiling, neighborhood attention masking, and software predication for multi-dimensional tensor layouts, we successfully implemented neighborhood attention for 1-D, 2-D, and 3-D problems. Fig. 4 presents an overview of how our fused kernels function when dealing with multi-dimensional (multi-axis) data.

### 3.4 Dilation and causal masking

Our methodology allows for dilation support trivially, through simple partitioning and slicing ahead of time. A dilated neighborhood attention problem can be mapped to a set of non-dilated neighborhood attention problems over non-overlapping tiles of the input. All sub-problems can be computed within the same kernel call, simply by issuing more CTAs in the grid. We additionally define and implement causal neighborhood attention into our fused kernel, which can be crucial to certain applications where only one spatial dimension requires causal masking (i.e. video embeddings may benefit from causally masked attention across the time axis and standard attention across height and width axes, which would be an exact 3-D spatio-temporal attention module.)

### 3.5 Notes on arithmetic intensity

Arithmetic intensity is the ratio of floating point operations over bytes of memory transactions, as defined by the Roofline model [25]:

$$\text{Arithmetic Intensity} = \frac{N_{ops}}{N_{bytes}} \tag{2}$$

Arithmetic intensity is typically used to determine whether an implementation/algorithm is bound by memory bandwidth or computational capacity, on a given problem size and hardware. Let's consider a simplified representation of self attention, where we only look at pure matrix multiplication FLOPs and bytes. Self attention is comprised of two back-to-back BMMs, which would be $2bhn^2d$ FLOPs for each of the BMMs, where $b$, $h$, $n$, and $d$ denote batch size, number of attention heads, sequence

length, and per-head dimension respectively. In total, that would be $4bhn^2d$ FLOPs. In the unfused implementation, 4 tensors with size $bhnd$ ($Q$, $K$, $V$ and output) are accessed in global memory, along with one intermediary tensor with size $bhn^2$ (attention weights or $P$), which is accessed twice. In total, that is $(4 \times bhnd + 2 \times bhn^2) \times s_{dtype}$ bytes, where $s_{dtype}$ is the byte size of the tensor element type. When implemented with fused attention, however, the number of bytes accessed for matrix multiplication from global memory is reduced to only reads and writes for $Q$, $K$, $V$, and attention outputs, or $4 \times bhnd \times s_{dtype}$.

Unfused implementations of attention are typically memory-bandwidth-bound at scale, given that their arithmetic intensity approaches a constant value as sequence length grows. If we take the limit of their intensity according to the aforementioned approximation of FLOPs and transaction bytes, as $n \to \infty$ with everything else as constants, we see that:

$$\lim_{n\to\infty} \frac{4bhn^2d}{(4bhnd + 2bhn^2)s_{dtype}} = \lim_{n\to\infty} \frac{2nd}{(2d + n)s_{dtype}} = \frac{2d}{s_{dtype}} \tag{3}$$

Fused attention therefore solves a key problem here, by reducing the number of memory transactions from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$, which means as sequence length grows, the limit of arithmetic intensity in fused attention does not converge, and it will therefore only become bound by computational capacity. This means that optimal fused attention kernels can almost fully utilize the underlying computational power of modern GPUs, and is the reason behind FP8 attention kernels for the Hopper architecture exceeding the 1 petaFLOP/s threshold [3, 20].

A natural question to ask is what happens to local attention patterns such as neighborhood attention, which promised to deliver more efficiency. In fused implementations of neighborhood attention (i.e. our proposed FNA), we can look at the growth of arithmetic intensity similar to self attention. If we consider the FLOPs for neighborhood attention to be $4bhn\ell d$, where $\ell$ is the size of the attention window, and that the number of global memory transaction bytes is the same as fused self attention (worst case), $4 \times bhnd \times s_{dtype}$, then we see that as $n \to \infty$, we converge towards a constant again, therefore making neighborhood attention more memory-bandwidth-bound as $n$ alone scales:

$$\lim_{n\to\infty} \frac{4bhn\ell d}{4bhnds_{dtype}} = \frac{\ell}{s_{dtype}} \tag{4}$$

However, the constant here is a function of $\ell$, the size of our attention window, which means that as we scale the sequence length or feature map size, attention window size will determine whether or not the problem is bound by memory bandwidth or computational power. Since smaller window sizes are closer to linear projections, and larger window sizes are closer to self attention, the fact that neighborhood attention can be bound by compute or memory bandwidth depending on window size is not a surprise. Therefore, it is highly recommended to choose neighborhood attention window sizes according to the input size and even hardware to maximize efficiency gain.

### 3.6 Limitations

Our formulation of GEMM-based and fused neighborhood attention kernels poses a critical question: *how much overhead can one expect from switching from a standard self attention kernel to neighborhood attention?* As pointed out in Sec. 3.2, our GEMM-based kernels suffer from a major bottleneck, especially in lower-precision, which stems from scatter/gathering of attention weights. We consider this to be an unavoidable issue in unfused implementations of neighborhood and sliding window attention. Unsurprisingly, our proposed changes to fused implementations are also not free. Changes that we find unavoidable, which in some cases can cause our fused kernels to incur higher runtime than the self attention baseline (xFormers FMHA) are the following (ordered by most significant to least significant):

1. Kernels specialized for 2-D and 3-D problems are no longer GEMMs, they are General Tensor-Tensor contractions (GETTs)! Similar to convolution, if the input layout is multi-dimensional, then the GEMM is converted to a special case of GETT. On older GPU architectures, this requires more complicated software predication, which will incur more instructions and heavier register usage, whereas on modern architectures like Hopper, the Tensor Memory Accelerator (TMA) can easily provide hardware predication. Our software predication logic is similar to standard practice for such cases in CUTLASS 2.X GEMMs, and similarly less performant than predication for contiguous matrix layouts. We find this to

Table 1: **FP16 forward pass benchmark overview.** We benchmark naive neighborhood attention kernels against our proposed GEMM and fused kernels in half precision, over a large set of problem sizes varying in batch size, spatial size, number of attention heads, and dimensions per head, and over different window sizes and dilation values. For every problem size, we also benchmarked self attention running with the xFormers FMHA (our baseline) and Flash Attention V2.

| NA Kernel | % of problems matched or outperformed | | | | |
|---|---|---|---|---|---|
| | neighborhood attn | | | self attn | |
| | Naive | GEMM | Fused | FMHA | FAv2 |
| *1-dimensional neighborhood attention* | | | | | |
| **Naive** | - | 1.7 % | 0.0 % | 21.8 % | 8.8 % |
| **GEMM** | 98.7 % | - | 0.0 % | 72.0 % | 54.2 % |
| **Fused** | 100.0 % | 100.0 % | - | 100.0 % | 98.2 % |
| *2-dimensional neighborhood attention* | | | | | |
| **Naive** | - | 16.4 % | 0.0 % | 32.9 % | 15.8 % |
| **GEMM** | 84.0 % | - | 0.0 % | 59.3 % | 29.8 % |
| **Fused** | 100.0 % | 100.0 % | - | 98.6 % | 92.4 % |
| *3-dimensional neighborhood attention* | | | | | |
| **Naive** | - | - | 0.0 % | 43.5 % | 20.2 % |
| **Fused** | 100.0 % | - | - | 97.3 % | 87.0 % |

Table 2: **FP32 forward pass benchmark overview.** We benchmark naive neighborhood attention kernels against our proposed GEMM and fused kernels in full precision, over a large set of problem sizes varying in batch size, spatial size, number of attention heads, and dimensions per head, and over different window sizes and dilation values. For every problem size, we also benchmarked self attention running with the xFormers FMHA (our baseline).

| NA Kernel | % of problems matched or outperformed | | | |
|---|---|---|---|---|
| | neighborhood attn | | | self attn |
| | Naive | GEMM | Fused | FMHA |
| *1-dimensional neighborhood attention* | | | | |
| **Naive** | - | 0.0 % | 0.0 % | 34.6 % |
| **GEMM** | 99.9 % | - | 37.7 % | 98.4 % |
| **Fused** | 100.0 % | 64.8 % | - | 99.9 % |
| *2-dimensional neighborhood attention* | | | | |
| **Naive** | - | 11.7 % | 5.4 % | 52.0 % |
| **GEMM** | 89.5 % | - | 28.1 % | 92.4 % |
| **Fused** | 96.0 % | 74.0 % | - | 99.3 % |
| *3-dimensional neighborhood attention* | | | | |
| **Naive** | - | - | 0.0 % | 61.1 % |
| **Fused** | 100.0 % | - | - | 98.6 % |

be the most significant contributor to additional runtime in our fused kernels, when compared to the baseline fused self attention kernel, FMHA. However, FNA is perfectly capable of hiding this additional overhead in many cases, and only falls behind in cases close to self attention (window size is approximately the same as input size.)

2. The attention masking logic, which depends on corresponding query and context token coordinates, original layout, and window size, introduces additional indexing logic in order to map linear indices to coordinates (unlike in 1-D problems where the mapping is the identity function), and it gets more complicated with more dimensions. This, along with additional statements in the masking condition, contributes to runtime, and is expected to worsen with more dimensions. Together, these contribute to more serious register spilling than the original 1-D kernel.

Despite these issues, we find that our fused kernels can still match or outperform our self attention baseline in approximately 100% of 1-D, 98.6% of 2-D, and 97.3% of 3-D problem sizes that we benchmarked.

## 4 Experiments

We evaluate the performance of our proposed methods by measuring their runtime against existing kernels in $\mathcal{N}ATTEN$. Most use cases in $\mathcal{N}ATTEN$ target naive CUDA kernels, with the exception of 2-D neighborhood attention with 32-dimensional attention heads. $\mathcal{N}ATTEN$ implements tiled kernels for those cases for up to and including window size 13 × 13, and only for the $QK$ operation. However, we treat all kernels in $\mathcal{N}ATTEN$ as our baseline, and will refer to them as naive kernels. We use a fixed set of problem sizes that vary in batch size, spatial size, number of attention heads, and dimensions per attention head, and run them through every implementation on an NVIDIA A100 GPU and measure their runtime using CUDA events. We iterate through multiple neighborhood attention window sizes and dilation values for every problem size. A summary of these benchmarks is presented in Tab. 1 (FP16) and Tab. 2 (FP32). We find that our GEMM-based kernels can improve or match the naive runtime in approximately 99% of 1-D problems (of 6150), and 84% of 2-D problems (of 5676) in half precision, and approximately 100% of the 1-D problems and 96% of the 2-D problems in full precision. Note that over 40% of the 2-D problems target tiled kernels in $\mathcal{N}ATTEN$, which we find can sometimes outperform our GEMM-based kernels. Another point of disadvantage in the FP16/BF16 variants of our GEMM-based kernels is using `LDG`s in pipelined

kernels, noted in Sec. 3.2. On the other hand, our fused kernels improve or match the naive runtime in approximately 100% of both 1-D (of 6150) and 3-D problems (of 2448) in both half precision and full precision, an 100% of 2-D problems in half precision, while only improving approximately 96% of 2-D problems in full precision. We also find that our fused kernels match or outperform our GEMM kernels in 100% of both 1-D and 2-D problems in half precision, while only doing so in approximately 65% of 1-D problems and 74% of 2-D problems in full precision, which is not very surprising given that full precision is typically more memory-bandwidth-bound. In both Tab. 1 and Tab. 2 we also inspect the percentage of problem sizes in which using our fused neighborhood attention kernel is outperformed by the FMHA kernel. This is only to inspect additional overhead caused by our implementation, which we expect to be more noticeable in 2-D and 3-D problems. Some of the overhead may be avoidable, but our takeaway is that it is unlikely to be fully avoidable, as pointed out in Sec. 3.6.

We further present a breakdown of our benchmarks in Tab. 3, where we report the average, minimum, and maximum improvement observed from switching from naive to GEMM-based, naive to fused, and GEMM-based to fused kernels. GEMM-based kernels exhibit strong performance compared to both naive and fused kernels in full precision, where fused kernels only have a very minimal edge over unfused. GEMM-based kernels also outperform naive kernels in half precision, especially in cases where tiled kernels are not available. While the tiled kernels are sometimes the better choice, we note that they simply cannot generalize to all problem sizes as our GEMM-based kernels can, nor are they easily extensible.

Table 3: **Forward pass benchmark breakdown.** Both GEMM-based and fused NA improve the baseline naive kernels on average. However, there exist cases in which naive kernels may be preferable to GEMM-based in both FP16 and FP32, but naive is rarely a good choice in half precision where both naive and GEMM are more memory bandwidth bound than fused.

| Dim | GEMM over naive | | | Fused over naive | | | Fused over GEMM | | |
|---|---|---|---|---|---|---|---|---|---|
| | Average | Min | Max | Average | Min | Max | Average | Min | Max |
| | | | | *FP16* | | | | | |
| **1-D** | ↑ **548 %** | ↓ **-53 %** | ↑ **3025 %** | ↑ **1759 %** | ↑ **60 %** | ↑ **11885 %** | ↑ **180 %** | ↑ **71 %** | ↑ **466 %** |
| **2-D** | ↑ **193 %** | ↓ **-57 %** | ↑ **862 %** | ↑ **958 %** | **0 %** | ↑ **7169 %** | ↑ **257 %** | ↑ **38 %** | ↑ **1199 %** |
| **3-D** | - | - | - | ↑ **1135 %** | ↑ **118 %** | ↑ **5497 %** | - | - | - |
| | | | | *FP32* | | | | | |
| **1-D** | ↑ **874 %** | ↓ **-31 %** | ↑ **3565 %** | ↑ **978 %** | ↑ **13 %** | ↑ **4419 %** | ↑ **17 %** | ↓ **-54 %** | ↑ **136 %** |
| **2-D** | ↑ **386 %** | ↓ **-43 %** | ↑ **1933 %** | ↑ **564 %** | ↓ **-30 %** | ↑ **4043 %** | ↑ **43 %** | ↓ **-53 %** | ↑ **451 %** |
| **3-D** | - | - | - | ↑ **712 %** | ↑ **25 %** | ↑ **3029 %** | - | - | - |

## 5 Future work & Conclusion

In this work, we formulated the neighborhood attention problem, and by extension multi-dimensional sliding window attention, which are inherently GEMV problems, as GEMM/GETT problems. Through this finding, we implemented extensible GEMM-based and fused CUDA kernels that implement neighborhood attention, which can significantly improve upon existing kernels in the $\mathcal{N}ATTEN$ project. These kernels will not only speed up previously-proposed models based on neighborhood attention, but can also significantly enhance ongoing research efforts in this direction. In addition, our fused kernels are the most flexible in terms of parameterization, by supporting varying window sizes, dilation factors, and causal masking across different axes, which enable unique applications such as 3-D spatio-temporal attention with causal masking across time. They also enjoy a reduced memory footprint, and can avoid being bound by memory bandwidth at scale.

Future directions in this area include but are not limited to: support for Context Parallelism (CP), implementations using more efficient predication (i.e. with the Hopper TMA), extension to more modern architectures (warp-specialized kernels in Hopper and Blackwell), extension to other AI accelerators, and better auto-tuning (or alternatives involving graph compilation).

We've shown that multi-dimensional local attention can indeed serve as solutions for scaling future large-scale long-context architectures, when provided with suitable software infrastructure. We hope that this inspires more research into multi-dimensional attention, as deep learning systems continue to grow larger in both model and input size.

**Acknowledgements.** We would like to thank NVIDIA and members of the CUTLASS project, in particular Haicheng Wu, for his valuable feedback and comments which led to the creation of GEMM-based NA. We also thank Meta xFormers team for developing FMHA, which is what our fused neighborhood attention kernels are based on. A. H. thanks Michael Isaev, Aditya Kane, and Kai Wang for their feedback on the paper. A. H. also thanks Bing Xu, Hao Lu, Michael Iovine, and Terry Chen for the invaluable learning experience while interning at HippoML, which helped accelerate the timeline of this project. This research was supported in part by National Science Foundation under Award #2427478 - CAREER Program, and by National Science Foundation and the Institute of Education Sciences, U.S. Department of Education under Award #2229873 - National AI Institute for Exceptional Education. This project was also partially supported by cyberinfrastructure resources and services provided by the Partnership for an Advanced Computing Environment (PACE) at the Georgia Institute of Technology, Atlanta, Georgia, USA.

## References

[1] Anurag Arnab, Mostafa Dehghani, Georg Heigold, Chen Sun, Mario Lučić, and Cordelia Schmid. Vivit: A video vision transformer. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

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

[3] HippoML Blog. Petaflops inference era: 1 pflops attention, and preliminary end-to-end results. `https://medium.com/p/21f682cf2ed1`, 2024.

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

[5] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2023.

[6] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[7] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations (ICLR)*, 2020.

[8] Ali Hassani and Humphrey Shi. Dilated neighborhood attention transformer. *arXiv preprint arXiv:2209.15001*, 2022.

[9] Ali Hassani, Steven Walton, Jiachen Li, Shen Li, and Humphrey Shi. Neighborhood attention transformer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[11] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

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

[14] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

[15] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International Conference on Machine Learning (ICML)*, 2018.

[16] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[17] William Peebles and Saining Xie. Scalable diffusion models with transformers. *arXiv preprint arXiv:2212.09748*, 2022.

[18] Markus N Rabe and Charles Staats. Self-attention does not need $O(n^2)$ memory. *arXiv preprint arXiv:2112.05682*, 2021.

[19] Prajit Ramachandran, Niki Parmar, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jon Shlens. Stand-alone self-attention in vision models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[20] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

[21] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, et al. Cutlass, 2023.

[22] Ashish Vaswani, Prajit Ramachandran, Aravind Srinivas, Niki Parmar, Blake Hechtman, and Jonathon Shlens. Scaling local self-attention for parameter efficient visual backbones. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[24] Steven Walton, Ali Hassani, Xingqian Xu, Zhangyang Wang, and Humphrey Shi. Stylenat: Giving each head a new perspective. *arXiv preprint arXiv:2211.05770*, 2022.

[25] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[26] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

# A Auto-tuner

GEMM kernels are, among other settings, parameterized by their tiling shape. Multi-dimensional variants (2-D and 3-D neighborhood attention) can also be parameterized by their fine-grained tile sizes, introduced by our formulation. As mentioned earlier, a GEMM with row tile size 64 can be reinterpreted as a number of 2-D and 3-D tiles (i.e. $x \times y$ for all positive integers $x$ and $y$ where $xy = 64$, and $x \times y \times z$ for all positive integers $x$, $y$, and $z$ where $xyz = 64$.) As a result, selecting tiling sizes based on factors such as problem size, hardware, and environment can further decrease achievable runtime. We therefore implement a very simple auto-tuning method as a proof of concept. Auto-tuning creates and maintains a cache for the lifetime of the application, which maps problems (defined by problem size, data type, and other such factors) to a tiling configuration. On a cache miss, the problem is benchmarked over a set of tiling configurations, and the best configuration gets cached.

While the auto-tuner can noticeably improve performance even further, we note that it is presently limited in the following:

1. **Distributed training.** auto-tuner context is limited to a single process, meaning jobs involving distributed training or inference will run the auto-tuner separately in each individual process. Aside from the possibility of different processes choosing different settings, which can slightly impact numerical determinism, this behavior is counter-intuitive. A more advanced auto-tuner would distribute possible settings over available processes and reduce auto-tuning time in the process, and guarantee the same settings across processes.
2. **Vast search space.** there exist in the order of thousands of valid settings for any given problem size, and searching over all of them is intractable. Our solution so far has been to generate far fewer possible settings, and even reduce the number of settings further by introducing a "thorough mode", which is disabled by default, but when enabled, will allow users to search over more settings and potentially gain more in speed. This issue is a common problem in modern computational packages, and we hope to alleviate it by common practices such as reducing benchmark time, distributing the process, caching to disk, lazy benchmarking, and approximate performance models.

# B Additional experiments

Herein we present some additional performance metrics from our GEMM-based and fused kernels compared against the baseline.

In Tab. 4, we break down expected performance improvements at the operation level from a single forward and backward pass. Both our GEMM-based and fused kernels provide significant improvement on average over the baseline, while there still exist cases where naive could potentially perform better, especially compared to our GEMM-based kernels. As pointed out in Sec. 3.2, the scatter and gather operation in our GEMM kernels are a significant bottleneck, especially in lower-precision and in NN and IN operations. In lower precision, NN and IN, which account for 75% of the backward pass operations (excluding softmax) will fail to hide their prefetch runtime from global reads, which are not asynchronous, and this will essentially impact the backward pass more than it does the forward pass. This issue, however, is limited to our unfused variant, and our fused kernels maintain their superior performance levels, offering up to an order of magnitude improvement in all variants (1-D, 2-D, and 3-D).

Table 4: **Forward + backward pass benchmark breakdown.** Improvements over naive, while not as significant as in the forward pass, are still significant. We report benchmark the full forward and backward pass in half precision only, because most training is done in lower precision.

| Dim | GEMM over naive | | | Fused over naive | | | Fused over GEMM | | |
|---|---|---|---|---|---|---|---|---|---|
| | Average | Min | Max | Average | Min | Max | Average | Min | Max |
| 1-D | ↑ 502 % | ↓ -48 % | ↑ 3017 % | ↑ 844 % | ↓ -20 % | ↑ 7605 % | ↑ 57 % | ↓ -50 % | ↑ 229 % |
| 2-D | ↑ 92 % | ↓ -70 % | ↑ 474 % | ↑ 385 % | ↓ -61 % | ↑ 3723 % | ↑ 150 % | ↓ -49 % | ↑ 855 % |
| 3-D | - | - | - | ↑ 447 % | ↓ -45 % | ↑ 2824 % | - | - | - |

In addition to our operation-level benchmarks, we also evaluate the effect of our proposed methodology on existing models that use neighborhood attention as a primitive, NAT [9] and DiNAT [8]. We

benchmark the throughput from all variants according to ImageNet-1K [12] specifications, and report FP16 and FP32 measurements in Tab. 5 and Tab. 6 respectively. We also benchmark a style-based generative adversarial (GAN) model based on neighborhood attention, StyleNAT [24] and report performance improvements in Tab. 7. We find that at least in problem sizes that the ImageNet classification models NAT and DiNAT typically require, which are typically smaller in spatial size and window size, and larger in batch size, our GEMM-based approach fails to improve the baseline in half precision, and only minimally improves it in full precision. Our fused kernels on the other hand never fail to improve upon the baseline, but they only provide significant improvement in half precision, and cases that use dilation frequently (DiNAT [8] variants). Improvements in the generative model, StyleGAN [24], are only observed in full precision (half precision is not recommended in this application), where again we find that both our GEMM-based and fused kernels can improve inference speed compared to existing naive kernels, with our fused kernels having a much more noticeable edge.

Table 5: **Model-level throughput changes when using our proposed GEMM-based and fused kernels in ImageNet classification.** Hierarchical vision transformers NAT and DiNAT can see between 26% to 104% improvement in FP16 throughput on an A100 (batch size 128) with our proposed fused kernel. Suffering from the memory alignment issue, our half precision GEMM kernels usually result in a much smaller improvement over naive kernels, particularly the tiled variants. The same measurements with FP32 precision are presented in Tab. 6.

| **Model** | **# of Params** (M) | **FLOPs** (G) | **Throughput** (imgs/sec) **Naive** | **GEMM** | **Fused** | **Top-1 Accuracy** (%) |
|---|---|---|---|---|---|---|
| **NAT-M** | 20 | 2.7 | 2975 | 2660 ( ↓ **-11 %** ) | 3742 ( ↑ **26 %** ) | 81.8 |
| **DiNAT-M** | 20 | 2.7 | 2672 | 2548 ( ↓ **-5 %** ) | 3930 ( ↑ **47 %** ) | 81.8 |
| **DiNAT$_s$-T** | 28 | 4.5 | 2850 | 2504 ( ↓ **-12 %** ) | 3847 ( ↑ **35 %** ) | 81.8 |
| **NAT-T** | 28 | 4.3 | 2167 | 1939 ( ↓ **-11 %** ) | 2772 ( ↑ **28 %** ) | 83.2 |
| **DiNAT-T** | 28 | 4.3 | 1910 | 1845 ( ↓ **-3 %** ) | 2909 ( ↑ **52 %** ) | 82.7 |
| **DiNAT$_s$-S** | 50 | 8.7 | 1800 | 1571 ( ↓ **-13 %** ) | 2445 ( ↑ **36 %** ) | 83.5 |
| **NAT-S** | 51 | 7.8 | 1457 | 1309 ( ↓ **-10 %** ) | 1879 ( ↑ **29 %** ) | 83.7 |
| **DiNAT-S** | 51 | 7.8 | 1360 | 1313 ( ↓ **-3 %** ) | 2145 ( ↑ **58 %** ) | 83.8 |
| **DiNAT$_s$-B** | 88 | 15.4 | 1351 | 1178 ( ↓ **-13 %** ) | 1837 ( ↑ **36 %** ) | 83.8 |
| **NAT-B** | 90 | 13.7 | 1110 | 997 ( ↓ **-10 %** ) | 1448 ( ↑ **30 %** ) | 84.3 |
| **DiNAT-B** | 90 | 13.7 | 982 | 950 ( ↓ **-3 %** ) | 1517 ( ↑ **54 %** ) | 84.4 |
| **DiNAT$_s$-L** | 197 | 34.5 | 846 | 744 ( ↓ **-12 %** ) | 1119 ( ↑ **32 %** ) | 86.5 |
| **DiNAT-L** | 200 | 30.6 | 669 | 647 ( ↓ **-3 %** ) | 1042 ( ↑ **56 %** ) | 86.6 |
| **DiNAT$_s$-L$^{(384\times384)}$** | 197 | 101.5 | 295 | 239 ( ↓ **-19 %** ) | 391 ( ↑ **33 %** ) | 87.4 |
| **DiNAT-L$^{(384\times384)}$** | 200 | 92.4 | 153 | 134 ( ↓ **-12 %** ) | 312 ( ↑ **104 %** ) | 87.5 |

Finally, we also attempted to estimate improvements in training time compared to our baseline. As suggested by our earlier findings regarding the limit of our GEMM-based implementation in the backward pass, we do not see any improvement in training time compared to the naive baseline. However, we find that our fused kernels deliver on the promise of improved half precision training time. We present our estimates in Tab. 8, which are based on measurements from training NAT [9] and DiNAT [8] variants according to their original specifications. We ran each model for 1 warmup epoch, and 1 benchmark epoch, the average throughput of which is used to estimate training time for 300 epochs.

Table 6: **Model-level throughput changes when using our proposed GEMM-based and fused kernels in ImageNet classification (full precision).** While fused attention kernels are not expected to have as large of an edge over BMM-style attention kernels in FP32, our fused kernels still happen to outperform naive kernels in full precision. It is also visible that our GEMM kernels can outperform naive kernels when we eliminate the memory alignment issue. That said, our FP32 GEMM kernels still impose a maximum alignment of 1 element on the attention weights tensor, which limits its ability to compete with other BMM-style attention kernels.

| **Model** | **# of Params** (M) | **FLOPs** (G) | **Throughput** (imgs/sec) **Naive** | **GEMM** | **Fused** | **Top-1 Accuracy** (%) |
|---|---|---|---|---|---|---|
| **NAT-M** | 20 | 2.7 | 2416 | 2481 ( ↑ **3 %** ) | 2658 ( ↑ **10 %** ) | 81.8 |
| **DiNAT-M** | 20 | 2.7 | 2217 | 2364 ( ↑ **7 %** ) | 2905 ( ↑ **31 %** ) | 81.8 |
| **DiNAT$_s$-T** | 28 | 4.5 | 2270 | 2255 ( ↓ **-1 %** ) | 2771 ( ↑ **22 %** ) | 81.8 |
| **NAT-T** | 28 | 4.3 | 1739 | 1802 ( ↑ **4 %** ) | 1942 ( ↑ **12 %** ) | 83.2 |
| **DiNAT-T** | 28 | 4.3 | 1591 | 1706 ( ↑ **7 %** ) | 2123 ( ↑ **33 %** ) | 82.7 |
| **DiNAT$_s$-S** | 50 | 8.7 | 1403 | 1393 ( ↓ **-1 %** ) | 1717 ( ↑ **22 %** ) | 83.5 |
| **NAT-S** | 51 | 7.8 | 1160 | 1199 ( ↑ **3 %** ) | 1293 ( ↑ **11 %** ) | 83.7 |
| **DiNAT-S** | 51 | 7.8 | 1102 | 1183 ( ↑ **7 %** ) | 1490 ( ↑ **35 %** ) | 83.8 |
| **DiNAT$_s$-B** | 88 | 15.4 | 1020 | 1009 ( ↓ **-1 %** ) | 1240 ( ↑ **22 %** ) | 83.8 |
| **NAT-B** | 90 | 13.7 | 867 | 897 ( ↑ **3 %** ) | 966 ( ↑ **11 %** ) | 84.3 |
| **DiNAT-B** | 90 | 13.7 | 795 | 851 ( ↑ **7 %** ) | 1059 ( ↑ **33 %** ) | 84.4 |
| **DiNAT$_s$-L** | 197 | 34.5 | 609 | 601 ( ↓ **-1 %** ) | 721 ( ↑ **18 %** ) | 86.5 |
| **DiNAT-L** | 200 | 30.6 | 506 | 540 ( ↑ **7 %** ) | 669 ( ↑ **32 %** ) | 86.6 |
| **DiNAT$_s$-L$^{(384 \times 384)}$** | 197 | 101.5 | 211 | 193 ( ↓ **-9 %** ) | 245 ( ↑ **16 %** ) | 87.4 |
| **DiNAT-L$^{(384 \times 384)}$** | 200 | 92.4 | 116 | 115 ( ↓ **-1 %** ) | 179 ( ↑ **54 %** ) | 87.5 |

Table 7: **Model-level throughput changes when using our proposed GEMM-based and fused kernels in style-based image generation.** We benchmark StyleNAT [24], a style-based generative adversarial model based on neighborhood attention under different kernels. We experimented with different batch sizes in order to achieve peak performance, and settled for 64 for the 256 × 256 variant, and 8 for the 1024 × 1024. StyleNAT does not recommend lower-precision, therefore these measurements are only done in FP32.

| **Dataset** | **# of Params** | **Throughput** (imgs/sec) **Naive** | **GEMM** | **Fused** | **FID** |
|---|---|---|---|---|---|
| **FFHQ (256 × 256)** | 48.9 M | 36.7 | 40.6 ( ↑ **11 %** ) | 45.5 ( ↑ **24 %** ) | 2.05 |
| **FFHQ (1024 × 1024)** | 49.4 M | 8.2 | 8.5 ( ↑ **3 %** ) | 11.5 ( ↑ **40 %** ) | 4.17 |

Table 8: **Training time improvement when using fused neighborhood attention kernels.** We ran each of the classification models based on neighborhood attention for one warmup epoch and one benchmark epoch, all with half precision (the typical training scenario), and report the estimated training time. Note that these numbers exclude positional biases, as our fused backward kernel does not support it.

| **Model** | **# of Params** (M) | **FLOPs** (G) | **Training time estimate** (hours) | | |
|---|---|---|---|---|---|
| | | | **Naive** | **GEMM** | **Fused** |
| **NAT-M** | 20 | 2.7 | 19.4 | 20.4 ( ↓ **-5 %** ) | 16.6 ( ↑ **17 %** ) |
| **DiNAT-M** | 20 | 2.7 | 20.4 | 21.2 ( ↓ **-4 %** ) | 17.4 ( ↑ **17 %** ) |
| **DiNAT$_s$-T** | 28 | 4.5 | 21.1 | 22.0 ( ↓ **-4 %** ) | 17.4 ( ↑ **21 %** ) |
| **NAT-T** | 28 | 4.3 | 26.5 | 28.2 ( ↓ **-6 %** ) | 24.0 ( ↑ **10 %** ) |
| **DiNAT-T** | 28 | 4.3 | 27.4 | 28.5 ( ↓ **-4 %** ) | 21.9 ( ↑ **25 %** ) |
| **DiNAT$_s$-S** | 50 | 8.7 | 33.3 | 33.2 ( **0 %** ) | 25.1 ( ↑ **33 %** ) |
| **NAT-S** | 51 | 7.8 | 39.2 | 41.8 ( ↓ **-6 %** ) | 33.7 ( ↑ **16 %** ) |
| **DiNAT-S** | 51 | 7.8 | 38.0 | 40.1 ( ↓ **-5 %** ) | 30.8 ( ↑ **23 %** ) |
| **DiNAT$_s$-B** | 88 | 15.4 | 45.4 | 46.1 ( ↓ **-2 %** ) | 32.6 ( ↑ **39 %** ) |
| **NAT-B** | 90 | 13.7 | 51.1 | 54.6 ( ↓ **-6 %** ) | 47.7 ( ↑ **7 %** ) |
| **DiNAT-B** | 90 | 13.7 | 54.4 | 56.0 ( ↓ **-3 %** ) | 41.0 ( ↑ **33 %** ) |

## NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes] .

   Justification: All claims made are with regard to earlier implementations of neighborhood attention, and how our approach aims to accelerate it. Claims are also based on experiments presented in the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes] .

   Justification: Limitations in terms of performance upper bounds and utilization of newer hardware accelerators are discussed in full.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: [NA]

4. **Experimental Result Reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: any reported performance numbers can be reproduced on the same hardware and with the specified versions of software, alongside our our released software distribution.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: All of our implementations are open sourced. Release of anonymized code is not possible.

6. **Experimental Setting/Details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [NA]

   Justification: [NA]

7. **Experiment Statistical Significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [NA]

   Justification: [NA]

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [NA]

Justification: [NA]

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes] .

   Justification: -.

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: The work introduces a set of implementations, and does not of itself pose any societal impacts positive or negative.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: [NA]

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes] .

    Justification: all prior arts have been cited, and all packages / open source projects and software used are credited and if possible cited.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes] .

    Justification: The open source project associated is well documented for all uses.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: [NA]

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: [NA]