# NEIGHBORHOOD ATTENTION: FAST AND FLEXIBLE SPARSE ATTENTION

A Dissertation
Presented to
The Academic Faculty

By

Ali Hassani

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Interactive Computing
College of Computing

Georgia Institute of Technology

October 2025

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

## LIST OF ACRONYMS

| | |
|---|---|
| **NATTEN** | Neighborhood Attention Extension |
| **NATTENSim** | NATTEN Simulator |
| **AI** | Artificial Intelligence |
| **BF16** | 16-bit Brain Floating Point [bfloat16] |
| **CNN** | Convolutional Neural Network |
| **CuTe** | CUDA Tensors |
| **CUTLASS** | CUDA Templates for Linear Algebra Subroutines |
| **DiNA** | Dilated Neighborhood Attention |
| **DiNAT** | Dilated Neighborhood Attention Transformer |
| **DiT** | Diffusion Transformer |
| **FLOP** | Floating Point Operation |
| **FMA** | Fused Multiply and Add |
| **FMHA** | Fused Multi-Headed Attention |
| **FNA** | Fused Neighborhood Attention |
| **FP16** | 16-bit Floating Point [float16] |
| **FP32** | 32-bit Floating Point [float32] |
| **GEMM** | General Matrix-Matrix Multiplication |
| **GEMV** | General Vector-Matrix Multiplication |
| **GETT** | General Tensor-Tensor Contraction |
| **GNA** | Generalized Neighborhood Attention |
| **GPU** | Graphics Processing Unit |
| **IN** | Inverse-Neighborhood Operator |

**KV** key-value

**KV tile** key-value tile

**LLM** Large Language Model

**MMDiT** Multimodal Diffusion Transformer

**NA** Neighborhood Attention

**NAT** Neighborhood Attention Transformer

**NN** Neighborhood-Neighborhood Operator

**PN** Pointwise-Neighborhood Operator

**Q tile** query tile

**SDPA** Scaled Dot Product Attention

**SWA** Sliding Window Attention

**TC** Tensor Core

**TMA** Tensor Memory Accelerator

**TPU** Tensor Processing Unit

**ViT** Vision Transformer

**WFM** World Foundation Model

# SUMMARY

Attention is at the heart of most foundational AI models, across tasks and modalities. In many of those cases, it incurs a significant amount of computation, which is quadratic in complexity, and often cited as one of its greatest limitations. As a result, many sparse approaches have been proposed to alleviate this issue, with one of the most common approaches being masked or reduced attention span.

In this work, we revisit sliding window approaches, which were commonly believed to be inherently inefficient, and we propose a new framework called Neighborhood Attention (NA). Through it, we solve design flaws in the original sliding window attention works, attempt to implement the approach efficiently for modern hardware accelerators, specifically GPUs, and conduct experiments that highlight the strengths and weaknesses of these approaches. At the same time, we bridge the parameterization and properties of Convolution and Attention, by showing that NA exhibits inductive biases and receptive fields similar to that in convolutions, while still capable of capturing inter-dependencies, both short and long range, similar to attention.

We then show the necessity for and challenges that arise from infrastructure, especially in the context of modern implementations such as Flash Attention, and develop even more efficient and performance-optimized implementations for NA, specifically for the most recent and popular AI hardware accelerators, the NVIDIA Hopper and Blackwell GPUs.

We build models based on the NA family, highlighting its superior quality and efficiency compared to existing approaches, and also plug NA into existing foundational models, and showing that it can accelerate those models by up to 1.6× end-to-end and without further training, and up to 2.6× end-to-end with training. We further demonstrate that our methodology can actually create sparse Attention patterns that realize the theoretical limit of their speedups.

This work is open-sourced through the NATTEN project at natten.org.

# CHAPTER 1
# INTRODUCTION

Inarguably among the most highly utilized and influential primitives in modern Artificial Intelligence (AI) and deep learning, **Attention** has long been cited for its complexity and memory footprint, especially in the case of **Self Attention**, where the complexity is quadratic with respect to the number of inputs (tokens, pixels, frames). This process is found in training almost all attention-based models, as well as the prefill stage in Large Language Model (LLM) inference, and the denoising stage in diffusion-based generative models. Many modern generative and foundational models, especially video models [1] and World Foundation Models (WFMs) [2] can spend over 50% of their runtime on self attention.

Attention's wide adoption in deep learning itself is largely owed to the Transformer architecture [3], and its widespread adoption. Transformers have made a significant contribution to AI research, starting with natural language understanding [4, 5] before being applied to other modalities such as speech [6] and vision [7, 8]. This success inspired efforts into attention-based models in vision, from backbone networks [9, 10], to more specific applications including image generation and density modeling [7, 11], object detection [12], image segmentation [13, 14], and more.

Vision Transformer (ViT) [8] was one of the first major demonstrations of transformers as direct alternatives to Convolutional Neural Networks (CNNs) [15, 16, 17], the de facto standard in computer vision. ViT is mostly convolution-free, and is mainly a Transformer encoder network, with the exception of an initial convolution used to embed the RGB space into a latent representation. It demonstrated competitive performance to CNNs on large scale image classification, and was followed with a surge in vision research focused on transformer-based architectures as competitors to CNNs [18, 19]. This architecture has successfully been extended to various other domains and applications [20, 21, 22], including generative models based on diffusion [2, 1].

Shortly after ViTs, hierarchical adaptations of it, such as Swin Transformer [23] became popular in vision for their enhanced applicability to downstream tasks such as object detection, and image segmentation. These architectures are predominantly reliant on sub-quadratic / sparse forms of attention, such as Blocked Attention. This is because the hierarchical structure processes very large feature maps and gradually downsamples them, similar to the CNN design, without actually employing convolutions for anything except embedding and downsampling, like ViT. Handling those very large feature maps with self attention will result in quadratic growth in complexity, which with the infrastructure available at the time also meant a quadratic growth in memory usage, limiting applicability to high-resolution tasks, namely detection and segmentation. Blocked Attention is trivial to implement, linear in complexity, and easy to adopt, but it can also greatly sacrifice inductive biases such as translational equivariance, requires additional tricks to become viable, and has very limited flexibility in terms of parameters. On the other hand, Sliding Window Attention (SWA) [9], offers the full translational equivariance of convolutions, and requires no additional tricks, and is more flexible. However, research into SWA, specifically in the context of vision where tokens assume multi-dimensional layouts (feature maps), winded down in early 2021, when it became a popular belief that SWA is simply too expensive and inefficient [23, 10].

This work is dedicated to investigating sliding window approaches, finds out their exact limitations, sheds more light on their advantages, and addresses many of their shortcomings. To the best of our knowledge, this is the first such work that makes the concept of Sliding Window Attention in multiple dimensions tractable, accessible, understandable, and production-ready. It does so by approaching the problem from multiple points of view in addition to AI, specifically Computer Architecture and High-Performance Computing.

We develop a new family of attention patterns which we call **Neighborhood Attention**

**(NA)**. Instead of explicitly imitating the behavior of discrete convolutions as done in SWA, we redefine the approach in a way that is better-suited for attention. We show that the computational complexity and memory footprint for this approach matches that of SWA and Blocked Attention, which is linear with respect to the number of tokens, instead of quadratic. We also show that other properties such as translational equivariance present in SWA are maintained, while overall model accuracy is improved with respect to both SWA and Blocked Attention. We then explore a weakness that is present in all three approaches: all of them perform locally dense computation, limiting receptive field growth to the same level as CNNs, and limiting the ability to capture global interactions and interdependencies. This observation is in line with findings from subsequent works that show modern CNNs can be competitive with comparable hierarchical vision transformers [24]. To overcome this weakness, we introduce a **dilation** parameter to NA, which behaves similarly to dilation in convolution, but enjoys much better upper bounds, and successfully recovers some of the global context lost compared to self attention baselines. Through extensive experiments, we show that in almost all cases, a vision transformer made of the combination of standard and Dilated Neighborhood Attention (DiNA) can outperform Swin Transformer [23] and modern CNNs such as ConvNeXt [24].

We show that infrastructure is the root of the misconception that sliding window approaches are inefficient, and that NA would suffer from this at least as much as SWA. We therefore developed custom implementations in CUDA C++ for NVIDIA Graphics Processing Units (GPUs) that successfully pushed inference and training speeds for NA-based models to levels comparable with Swin and ConvNeXt. This implementation, while initially successful at enabling research in this direction, is quite limited, especially with the rapid evolution in GPU architectures, and the seismic shift in how attention is implemented. In late 2021, Rabe and Staats [25] proposed an implementation of attention that avoids fully materializing the attention weight matrix, thereby eliminating the $\mathcal{O}(n^2)$ memory requirement. This concept is today commonly referred to as Fused Multi-Headed Attention

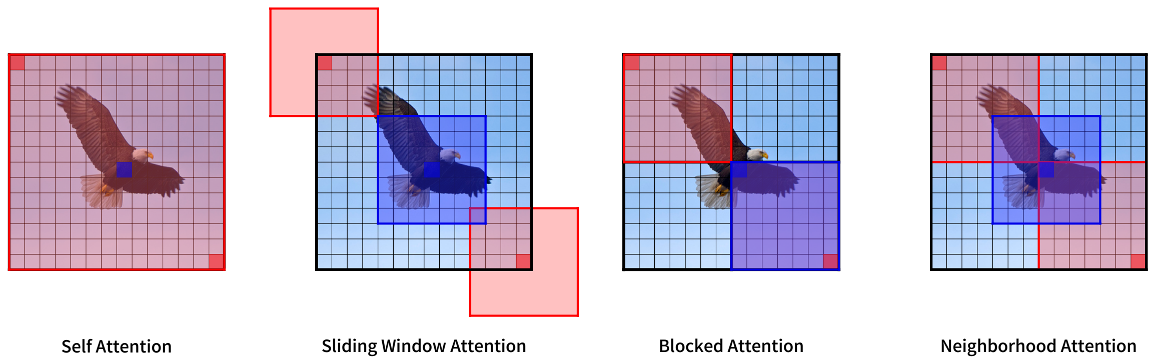


**Figure 1.1. Illustration of different attention patterns compared to Neighborhood Attention (NA).** Each image represents a $14 \times 14$ feature map (layout of tokens), and highlighted windows represent the context window for one specific query token. Self attention allows global interactions between every pair of tokens / pixels. Sliding Window Attention (SWA) restricts interactions for every token to a local window around it, which bears local inductive biases present in CNNs. Blocked attention partitions tokens, and restricts global interactions to each partition, which is easier to implement and parallelize compared to SWA, but eliminates the dynamically shifting receptive field among other properties present in the former two. NA restricts interactions for every token to its nearest neighbors, capturing more context compared to SWA, while preserving its useful properties, and increasing flexibility.

(FMHA), following the name of a different implementation from approximately the same period. In 2022, Dao et al. [26] created the first viable implementation of this concept, dubbed Flash Attention, for NVIDIA GPUs, which not only offered the benefit of reduced memory footprint, but also a significant boost in speed, both in the forward and backward pass. The reason behind these speedups is primarily the fact that the former unfused implementations of attention become bottlenecked by memory operations much more quickly than utilizing the computational power of modern hardware accelerators. In the years that followed, most implementations of attention across various hardware architectures have switched to this implementation as a result. Given the limitations of existing infrastructure for NA, and the improvement in the baseline thanks to fused attention kernels (FMHA), we study new methods of implementing these classes of attention patterns that can be more efficient, considering modern hardware and software.

We show that sliding window attention approaches like NA are vector-matrix multiplication problems, while most of the computations in deep learning, including Attention itself, are dense matrix-matrix multiplications. The massive computational power of modern hardware accelerators is mainly realizable with these dense operations, thanks to specialized hardware units for such operations (specifically matrix multiplication), such as NVIDIA Tensor Cores (TCs). This makes NA limited by memory bandwidth, and not computation, leading to potentially inferior throughput compared to the baseline it attempts to accelerate. However, NA preserves *spatial locality*, meaning tokens that are close to each other in the coordinate space overlap greatly in their attention context window. Based on this observation, we propose a packed implementation of NA that uses matrix multiplications, allowing it to target TCs, offering significant speedups over the former naive implementation. This implementation also numerically matches the former implementation by masking the accumulation of invalid dot products. We further expand this methodology to fused implementations such as FMHA, and develop **Fused Neighborhood Attention (FNA)**, which operates similarly to FMHA, but instead implements exact NA. Through these improved implementations, we can accelerate existing models based on NA in both inference and training time, and enable far greater adoption in state of the art AI models. In addition to accelerating existing models and applications based on NA which are presented in this work, these implementations have also enabled other independent studies use and utilize NA and its superior properties, which would have been intractable with the more naive implementations. Among them are applications such as RGB space generative models [27] and weather prediction and modeling [28].

However, as the demand for scale continues to dominate AI, and each generation of hardware accelerators, specifically NVIDIA GPUs, push the limits of compute further, hardware architecture and as a consequence programming models and software, becomes more complex. The aforementioned FNA implementation would therefore fall short in delivering what is naturally expected from sparse forms of attention: delivering speedups. Speedup in this context is always with respect to the best FMHA implementation available for the hardware of interest, which is naturally compute kernels designed for a specific hardware architecture. While forward-compatibility allows FNA to run on newer generations, it has little hope for utilizing the new hardware enough and to the point of achieving a similar throughput to the state of the art FMHA. We study this, and other weaknesses of NA and FNA, under an extended framework called Generalized Neighborhood Attention (GNA), where we resolve two issues: 1. we adopt a new design for FNA that is more agnostic to architectural differences, different programming models, and FMHA designs, through the use of which we develop very efficient FNA kernels for NVIDIA's most recent datacenter-class architectures: Hopper and Blackwell. 2. we extend the NA family by adding a new parameter, **stride**, that relaxes the sliding window property, and trades off translational equivariance for potentially better speedups through reduced fine-grained masking. We show that through these improvements, we can achieve speedups proportional to savings in compute, which is the maximum theoretically achievable.

We demonstrate GNA's effectiveness by introducing it into visual foundation models, both out-of-the-box and with further fine-tuning, and attempt to maximize sparsity while minimizing quality degradation. Most notably, we show that by using the entire family of Neighborhood Attention (NA) in the recent NVIDIA Cosmos Predict 2 WFM, we can accelerate single-GPU inference workloads by 1.7× to 2.6× across model variants and GPUs.

All of the aforementioned advances in faster, more efficient, and more reliable implementations are open-sourced and available as part of the Neighborhood Attention Extension (NATTEN) project [1]. All models trained, and their training recipes and relevant implementations are also publicly available under the Neighborhood Attention Transformer (NAT) project [2].

[1]https://natten.org

[2]https://github.com/SHI-Labs/Neighborhood-Attention-Transformer

# CHAPTER 2
# NEIGHBORHOOD ATTENTION TRANSFORMER

In this chapter we introduce various sparse forms of Attention, with focus on their differences from a deep learning architecture point of view. We introduce Neighborhood Attention (NA), and use it to build a vision architecture, which we call Neighborhood Attention Transformer (NAT), and investigate its useful properties and shortcomings. We discuss a shortcoming that affects not just NA, but other more prominent sparse attention methods: lack of global context modelling and localized receptive fields. We show that by using Dilated Neighborhood Attention (DiNA), which allows non-contiguous neighborhoods, we can recover some of the global context and grow the receptive field more quickly, which leads to gains in model accuracy, without impacting efficiency. We conduct experiments with NAT and Dilated Neighborhood Attention Transformer (DiNAT) on multiple vision benchmarks, and present their achieved performance levels compared to state-of-the-art architectures such as Swin Transformer [23] and ConvNeXt [24].

Part of the materials in this chapter were published in CVPR 2023, under “Neighborhood Attention Transformer”, and the rest are available as an arXiv preprint titled “Dilated Neighborhood Attention”.

## 2.1 Background and related works

In this section, we review Attention, the Transformer [3], and Vision Transformer (ViT) [8]. We then briefly discuss some of the various sparse forms of attention proposed and adopted in recent years, focusing on those more closely related to the topic, specifically Blocked Attention [10, 23], and Sliding Window Attention (SWA) [9, 29].

### 2.1.1 Self Attention and the Transformer

Attention is defined as an operation between two sets of vectors: a query set and a context set. The interaction of query and context vectors is mapped to a probability distribution, which is then used as weights for accumulating context per each query vector. The most widely used form of attention is **Scaled Dot Product Attention (SDPA)**, in which the query and context sets are transformed using linear projections, with the context projected into two sets, referred to as the key-value pair. Scaled dot products of query and key vectors, $A$, is mapped into a probability distribution through the softmax operator, which produces the final attention weights / probabilities, $P$. The output is a transformation of the query set. Each output vector is derived from the weighted sum of all value vectors according to the probability between the corresponding query and corresponding key. It can be expressed as follows:

$$Attention(Q, K, V) = \overbrace{softmax\left(\underbrace{\frac{QK^T}{\sqrt{d}}}_{A}\right)}^{P} V, \tag{2.1}$$

where $Q$, $K$, and $V$ are matrix representations of query, key, and value vectors respectively, $\sqrt{d}$ is the scale term, and $d$ is the dimensionality of the dot product (size of the query and key vectors, or number of columns in $Q$ and $K$). Self Attention, is a special case of Attention in which the query and context sets are identical. In the case of SDPA, this means for a set of $n$ input vectors, the attention weight / probability matrix, $P$, is $\in \mathbb{R}^{n \times n}$, incurring an $\mathcal{O}(n^2)$ time and space complexity.

This operation bears a number of advantages, including, but not limited to, the ability to model global inter-dependencies, and therefore a global receptive field, as well as equivariance to permutations of the order / arrangement of the input sequence, therefore making it equivariant to translations as well. Translational equivariance is a property also present in CNNs.

ViT [8], one of the earliest works applying a pure Transformer encoder to vision, showed the power of large-scale self attention based models in computer vision. The Transformer Encoder is typically comprised of Self Attention specifically, suffering from the quadratic complexity. Follow up works extended the study by incorporating changes to training techniques [18], architecture [19], applications to tasks such as detection [21] and generation [22], application to videos [30], and applications to small data regimes [31].

Self Attention is notorious for the aforementioned quadratic complexity with respect to the number of input tokens. Its memory footprint can be reduced to linear [25, 26] thanks to kernel fusion and online computation of softmax statistics [32], a technique referred to as Fused Multi-Headed Attention (FMHA). Its time complexity, however, can still limit scaling to long documents and large context windows in LLMs [33, 34], high-resolution images [23], and high-resolution or long videos [2, 1]. In addition, some prior works have shown that modeling global interactions can in some cases be unnecessary, especially in the presence of redundant tokens [35, 36, 37, 38]. Sparsity and restricting interactions in Self Attention is, as a result, a problem that continues to be of interest to the research community.

### 2.1.2 Sparse Attention

SDPA is commonly implemented with two matrix multiplications, and therefore can enjoy many forms of sparsity which exist for matrix multiplications, such as structured sparsity [39], and low-rank approximations [40, 41]. Sparsity can also be introduced into attention by choosing other targets of sparsity, which is sometimes application-specific. For example, some approaches are designed specifically for introducing sparsity into LLM inference workloads, where key-value (KV) caching is a necessity, and therefore calls for some form of sparsity or compression [42, 43, 44]. Approaches such as Token Merging [37] attempt to directly reduce the number of both query and context tokens together, and have been shown to be effective in training-free acceleration for certain vision tasks [38].

Sparse Attention approaches can be divided into static and dynamic approaches. Dynamic approaches [45, 38, 39] can be more effective without fine-tuning, whereas static approaches [7, 41, 9, 11, 29] are more likely to achieve better efficiency gains [33, 34, 23]. Some approaches can be classified as hybrids of the two [38, 37, 46], where statistics gathered at runtime guide rearrangement of computation. Some may even use entirely dense computations [23, 38, 37]. In this work, our focus is primarily on static approaches, which include SWA and Blocked Attention, with the latter being the most widely adopted technique, and the former only recently being adopted by some LLM architectures [33, 34, 46].

### 2.1.3 Sliding Window Attention (SWA)

Ramachandran et al. [9] proposed one of the earliest forms of local attention, specifically designed for applications in computer vision, and therefore defined for 2-dimensional layouts of tokens (2-D feature maps). The operation, later dubbed Sliding Window Attention (SWA), defines a unique key-value pair for every query token according to its spatial coordinates. For every query, its key-value pair is reduced, from the entire set of context tokens, to only those that fall within a local sliding window around the query token in the 2-dimensional coordinate space. The same raster scan pattern which is seen in discrete convolution [1] can be used to extract the tokens to which a query will attend. SWA maintains translational equivariance, and introduces inductive biases such as locality. Ramachandran et al. [9] showed that SWA could replace convolution in existing CNNs, such as ResNets [17], and reduce computational cost. Despite the promise it showed, the authors found that the resulting model suffers from worse end-to-end runtime compared to the baseline, citing the inefficiency of their implementation. Works succeeding it therefore switched to alternative methods that could run more efficiently. An example is HaloNet [10], which proposed trading off the inductive biases and superior modeling quality of sliding window

[1] Implementation is commonly referred to as image-to-column (im2col).

for the increased efficiency. They did so by introducing a delay step into the sliding window. Implementation difficultly and lack of infrastructure is not the only shortcoming of SWA. This operation also eliminates the global receptive field, and the ability to model global inter-dependencies, which is a flaw in all localized attention patterns. In addition to that, the raster scan pattern reduces interactions when sliding windows go out of bounds. This issue becomes more serious as window sizes grow, or when patterns such as dilation are introduced, since the reduced interactions are a direct result of the "zero padding"-style handling of corner cases.

### 2.1.4 Blocked Attention

Blocked Attention [47] partitions input tokens, and applies Self Attention to each partition independently. Unlike SWA, this pattern can be implemented trivially through deep learning frameworks like PyTorch [48]. Tensor partitioning itself is a no-op, and only changes the tensor layout, but most implementations require a permutation to interface with Attention implementations, and this forces a small memory operation. Since partitions are non-overlapping regions, Blocked Attention breaks translational equivariance, which is a useful property present in not only CNNs, but also models using Self Attention or SWA. Another issue with this approach is that implementation can get more complicated if the shape of the input token layout is not evenly divisible by the partition shape. In addition, without any change in the partition shape or origin, tokens in different partitions will never interact. This means that the receptive field will not grow, which could greatly limit achievable performance. Liu et al. [23] proposed "cyclic shifts" as a solution to this issue, shifting pixels and masking invalid interactions, which is functionally the same as moving the partition origin (partition lines). In their proposed Swin Transformer model, they chose to alternate between standard Blocked Attention, and Blocked Attention with cyclic shift throughout the layers. Swin is a hierarchical vision transformer focusing on applications to downstream vision tasks. It was further noted in this work that SWA was not pursued citing the same implementation and infrastructure woes as prior works [9, 10].

### 2.1.5 Non-contiguous Sparse Attention

Some works in language processing, such as Longformer [29], explored introducing dilation into SWA, which leads to a non-contiguous pattern that helps capture more global context without increasing window size. Blocked Attention can also be transformed into a non-contiguous sparse form [47], bearing similar properties as dilated SWA without the sliding windows. Child et al. [11] proposed Sparse Transformers, which included different non-contiguous and local patterns, spread across different attention heads. To the best of our knowledge, no prior works explored multi-dimensional dilated SWA.

In summary, viable Sparse Attention as a direct solution to the complexity of Self Attention has been sought-after for long, but primarily limited to Blocked Attention, especially in computer vision. In this chapter, we aim to revive SWA methods by addressing their most limiting shortcomings, and shed new light on their advantages.

## 2.2 Methodology

In this section, we define Neighborhood Attention (NA), and describe its properties and its extensions, such as dilation, and causal masking. We then introduce a hierarchical vision transformer based on NA, which we dub Neighborhood Attention Transformer (NAT). We conduct various experiments in image classification, object detection, and image segmentation, in which we compare NAT variants to comparable variants of Swin Transformer [23], as well as a strong CNN competitor, ConvNeXt [24]. We also experiment with replacing Self Attention in the original ViT architecture [8] with different forms of NA.

### 2.2.1 Neighborhood Attention (NA)

For simplicity, we assume only a single attention head and batch, and assume a single-dimensional layout of tokens. Extension to multi-dimensional layouts of tokens is a repetition of the same notation over each dimension independently of the rest.

Given an input sequence in the form of a matrix, $X \in \mathbb{R}^{n \times d}$, the rows of which are $d$-dimensional token vectors, we first project $X$ into queries, and key-value pairs, following SDPA (see Eq. (2.1)). We denote those projections with matrices $Q$, $K$, and $V$, all of which have the same number of rows as $X$. We denote the $i$-th row in those matrices, corresponding to the $i$-th input token, with $Q_i$, $K_i$ and $V_i$. We define neighborhood attention dot products for the $i$-th token with window size $k$, $\mathbf{A}_i^k$, as the following vector-matrix multiplication:

$$\mathbf{A}_i^k = Q_i \begin{bmatrix} K_{\rho_1(i)} \\ K_{\rho_2(i)} \\ \vdots \\ K_{\rho_k(i)} \end{bmatrix}^T , \tag{2.2}$$

where $\rho_j(i)$ denotes token $i$'s $j$-th nearest neighbor. We similarly define neighboring values, $\mathbf{V}_i^k$, to which attention weights would be applied, as a sub-matrix of $V$, the rows of which are the $i$-th token's $k$ nearest neighboring value projections:

$$\mathbf{V}_i^k = \begin{bmatrix} V_{\rho_1(i)} \\ V_{\rho_2(i)} \\ \vdots \\ V_{\rho_k(i)} \end{bmatrix} . \tag{2.3}$$

The final output from neighborhood attention for the $i$-th token with window size $k$ is:

$$\mathbf{O}_i^k = softmax\left(\frac{\mathbf{A}_i^k}{\sqrt{d}}\right)\mathbf{V}_i^k, \tag{2.4}$$

where similar to Eq. (2.1), $\sqrt{d}$ is the scaling parameter, and $d$ is the embedding dimension.

We note that $k$ is always assumed to be an odd number, so that the query token is perfectly centered within the sliding window (with the exception of corner cases).

### 2.2.2 Dilated Neighborhood Attention (DiNA)

Given dilation value $\delta$, we define $\rho_j^\delta(i)$ as the $i$-th token's $j$-th nearest neighbor that also satisfies: $j \bmod \delta = i \bmod \delta$. Using it, we define neighborhood attention dot products for the $i$-th token with window size $k$ and dilation $\delta$, $\mathbf{A}_i^{(k,\delta)}$, as follows:

$$\mathbf{A}_i^{(k,\delta)} = Q_i \begin{bmatrix} K_{\rho_1^\delta(i)} \\ K_{\rho_2^\delta(i)} \\ \vdots \\ K_{\rho_k^\delta(i)} \end{bmatrix}^T . \tag{2.5}$$

We similarly define the corresponding value tokens, $\mathbf{V}_i^{(k,\delta)}$, to which attention weights would be applied, as the following sub-matrix of $V$:

$$\mathbf{V}_i^{(k,\delta)} = \begin{bmatrix} V_{\rho_1^\delta(i)} \\ V_{\rho_2^\delta(i)} \\ \vdots \\ V_{\rho_k^\delta(i)} \end{bmatrix} . \tag{2.6}$$

The final output from neighborhood attention for the $i$-th token with window size $k$ and dilation $\delta$ is:

$$\mathbf{O}_i^{(k,\delta)} = softmax\left(\frac{\mathbf{A}_i^{(k,\delta)}}{\sqrt{d_k}}\right)\mathbf{V}_i^{(k,\delta)}. \tag{2.7}$$

The goal of dilation is to help grow the receptive field more quickly, and capture more global context but with the same computation budget. We discuss receptive field growth further in Sec. 2.2.5.

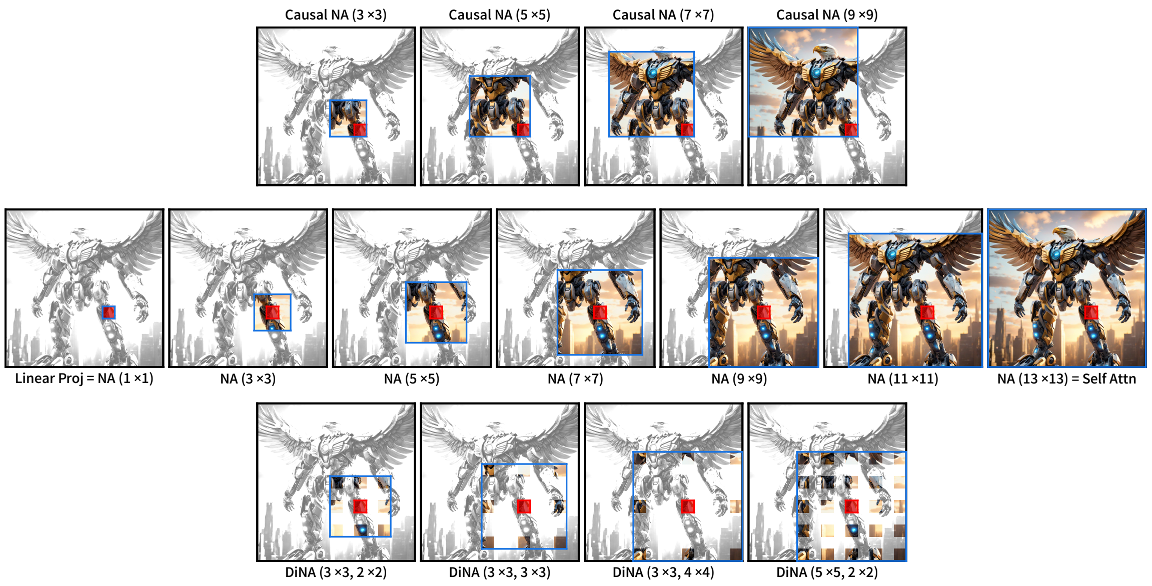


**Figure 2.1. Illustration of the spectrum of various attention patterns implementable with NA.** NA only attempts to center the query token (red) within the context window (blue). NA with window size 1 is equivalent to linear projection ("no attention"). NA approaches Self Attention as window size grows, and matches it when equal to input size. Dilation introduces global context without increasing window size by making the context window non-contiguous, and causal masking treats the input dimension as a temporal dimension, preventing interaction with tokens with larger coordinates. Window size, dilation, and causal masking, can assume different values per dimension.

### 2.2.3 Causal NA

Since NA is a special form of attention, it can also take a causal form. Any dimension can optionally use the causal form, which treats that dimension as temporal, and therefore interactions between any one token and future tokens will be excluded. Causal NA also attempts to shift all $k$ neighbors to the left side of each query, so that wherever possible, each query still attends to $k$ tokens. This definition is useful for any application with a temporal dimension, such as text and videos. In video domains, tokens corresponding to a certain time frame should only attend to tokens in previous and current frames, and not future frames. With causal NA, they can also optionally restrict the number of frames in the past, as well as the number of tokens within each frame. There is no restriction on using causal masking and dilation together, nor in using different window sizes, dilation values,

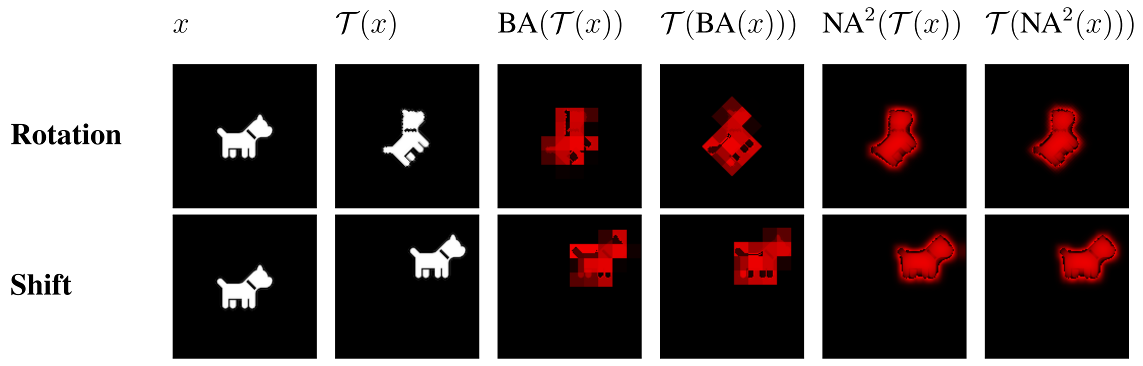

$x$
$\mathcal{T}(x)$
$\text{BA}(\mathcal{T}(x))$
$\mathcal{T}(\text{BA}(x)))$
$\text{NA}^2(\mathcal{T}(x))$
$\mathcal{T}(\text{NA}^2(x)))$
Rotation
Shift

**Figure 2.2. Visualization of translations applied to Blocked Attention and NA.** $\mathcal{T}$ denotes the translation function (top row is rotation, bottom row is shift). "BA" denotes blocked attention with shifts applied to the input per Swin, with a residual connection in between. This pattern breaks translational equivariance. "NA$^2$" denotes two consecutive neighborhood attention operations applied to the input, again with a residual connection in between. NA preserves translational equivariance.

or use of causal masking across different dimensions. In Fig. 2.1 we present a visualization of the spectrum of possible attention patterns when considering all three parameters in Neighborhood Attention: window size, dilation, and causal masking.

### 2.2.4 Translational equivariance

One of the useful properties present in both Self Attention and convolution is translational equivariance [49]. Any function $f$ is equivariant to a translation function $\mathcal{T}$ if $\mathcal{T}(f(x)) = f(\mathcal{T}(x))$. Translations, in the context of vision, refer to transformations such as shift and rotation. Self Attention preserves this property because it is equivariant to permutations in the order of tokens. Convolutions are translationally equivariant [49], since every output pixel is the product of its corresponding input pixel centred in a local window and multiplied by a static kernel, and the local window preserves spatial locality. Note that convolutions are not equivariant to arbitrary permutations like attention. SWA [9], and by extension NA, like convolution, are no longer permutation equivariant. However, translational equivariance is still maintained, because of the local raster scan pattern which closely follows convolutions. Blocked Attention, due to the partitioning, breaks translational equivariance across the entire feature map, while permutation equivariance within partitions is maintained. To better illustrate this, we visualize applying Blocked Attention (per Swin [23]) and NA with translations in Fig. 2.2.

### 2.2.5 Receptive field

Receptive field growth rate is considered as one of the contributing factors to a model's theoretical performance. Herein we present an analysis of NA's receptive field, and the effect of dilation, and compare against that of Self Attention, Blocked Attention, and convolution. For simplicity, we will continue to use 1-dimensional notation. We denote the number of layers with $\ell$, window size with $k$, and number of tokens with $n$. Self Attention has a global receptive field, because every token can contribute to the output of every other token, therefore receptive field size in Self Attention is always the upper bound, $n$. On the other hand, convolutional layers and NA start with a receptive field of size $k$, since $k$ tokens / inputs contribute to the output of any given token / input. They also grow by $k-1$ tokens per layer. As a result, both will end up with a receptive field growth of $k+(k-1)*(\ell-1)$, which can is simplified to $\ell(k-1)+1$. Blocked attention on its own maintains a constant receptive field of the same $k$ tokens per layer. This can be problematic if used throughout the model without any operation introducing interactions between tokens in different partitions, meaning the receptive field will not grow. To introduce cross-partition interactions, Liu et al. [23] proposed shifting inputs in order to create a shifted partitioning effect with minimal overhead, which they dubbed cyclic shift. Alternating between blocked attention with and without cyclic shift resolves this issue, and expands receptive fields by exactly one window per layer, which is an expansion of $k$ per layer. This results in a growth rate of $\ell k$, which is slightly larger than that of convolutions and NA by $\ell-1$.

As for NA's receptive field growth with dilation, it can range anywhere from the original $\ell(k-1)+1$, to an exponentially growing receptive field of $k^{\ell}$. The lower bound is simply the growth rate from NA with no dilation, meaning every layer's dilation value is

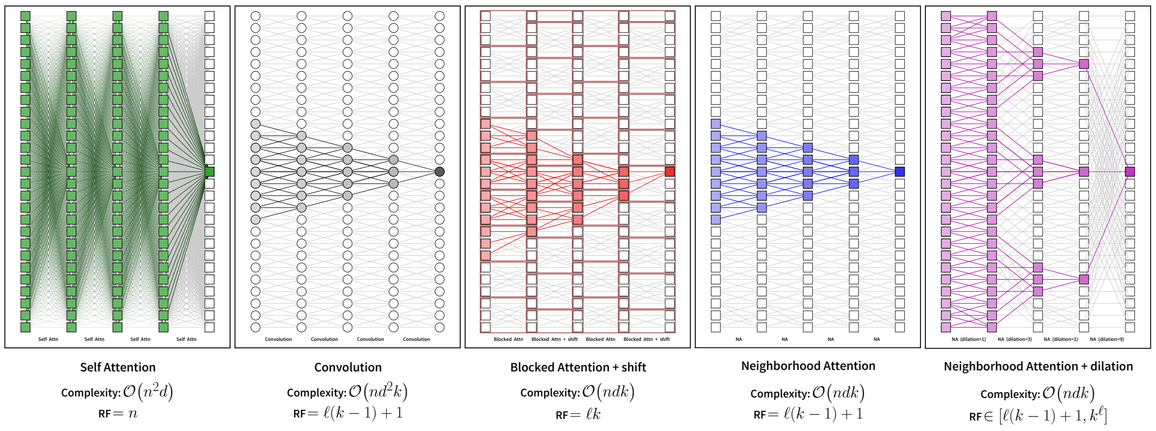


**Figure 2.3. Receptive fields from Self Attention (per ViT), Convolution, Blocked Attention with shifts (per Swin), and NA, both with and without dilation.** $\ell$ denotes number of layers, $n$ denotes number of tokens, $d$ denotes embedding dimension, and $k$ denotes window size. All receptive fields are bounded by input size, $n$. Self Attention's global receptive field yields the maximum receptive field per every layer. Convolution, Blocked Attention, and NA grow their receptive fields linearly with more layers. With dilation, NA's receptive field growth can range from linear, $\ell(k-1)+1$, to exponential, $k^{\ell}$.

set to 1. Regardless of dilation, the first layer always yields a receptive field of size $k$. Each one of the $k$ tokens in the first layer can target a maximum of $k$ new tokens in the following layer, given a dilation value that guarantees there is no overlap (except on the token itself) with the previous layer. As a result of this, the upper bound for growth in the second layer is $k^2$. As long as overlaps are avoided in every layer, this will generalize to $k^{\ell}$ over $\ell$ layers. Therefore, by introducing dilation into NA, one can potentially achieve an exponentially growing receptive field, which is a step in the right direction in terms of reaching a global receptive field. It is important to note that convolution can be dilated as well, but dilation values are much more constrained compared to NA. This difference will be discussed further in Sec. 2.2.8. An illustration of the receptive field growth in the mentioned operators is also presented in Fig. 2.3.

### 2.2.6 Complexity

Herein we present an analysis of the time complexity for NA, and compare against Self Attention, Blocked Attention, and convolution. Computing Self Attention weights is a matrix multiplication of an $n \times d$ matrix by the transpose of a matrix with the same size, which is $\mathcal{O}(n^2d)$. The softmax operation requires computing element-wise exponential values followed by $n$ summations of $d$ elements, and an element-wise division, which is simply $\mathcal{O}(nd)$. Applying attention weights is similar in complexity to computing the weights, but is an element-wise multiplication of $n$ weights by and $n \times d$ matrix ($\mathcal{O}(nd)$) for every one of $n$ tokens, which is again $\mathcal{O}(n^2d)$. NA, Blocked Attention, and SWA, share the same computational complexity given the same window size. For these methods, per-token interactions drop from $n$ tokens to $k$ tokens, reducing the original complexity to $\mathcal{O}(ndk)$. Convolutions on the other hand apply static kernels of size $k$ to the input, which is functionally the same as applying a linear projection, $k \times d \rightarrow d$, to $n$ activations of size $k \times d$ (assuming no padding). This is a matrix multiplication of an $n \times kd$ matrix by a $kd \times d$ matrix, which has a complexity of $\mathcal{O}(nd^2k)$. In comparison to local / sparse Attention, the cost of dense convolution can grow more quickly. It is worth noting, however, that depthwise convolutions have a complexity of $\mathcal{O}(ndk)$, and that SDPA also always requires three linear projections, which would add $\mathcal{O}(nd^2)$ to the overall complexity. Fig. 2.3 includes computational complexity in addition to receptive field growth.

### 2.2.7 Relation to Self Attention and Linear Projection

NA can exhibit certain properties under different window sizes. Given a window size of 1, each token can only attend to itself, resulting in only a single dot product. Since attention weights are derived from mapping said dot products to a probability distribution through softmax, the attention weight for every context token will be exactly 1, and therefore every output token would be equal to 1 times its corresponding value token. In other words, with a window size of 1, the attention operator becomes the identity function over $V$.

On the other hand, given the maximum window size, which is the same as input size, each token will attend to every other token, therefore computing exactly self attention. This can be easily seen by increasing window size $k$ in Eq. (2.4). This property is useful because any self attention operation can be restricted step by step by simply reducing the number of interactions through reducing window size. We illustrate this relationship in Fig. 2.1 as well.

### 2.2.8 Relationship with Convolution

NA's similarity to discrete convolution in the adoption of the sliding window pattern raises the question: *what is the advantage of using NA over convolution?*

This question is best addressed by pointing out both the differences between Sliding Window Attention and convolution in general, and by looking at properties specific to NA that cannot be transferred to convolutions.

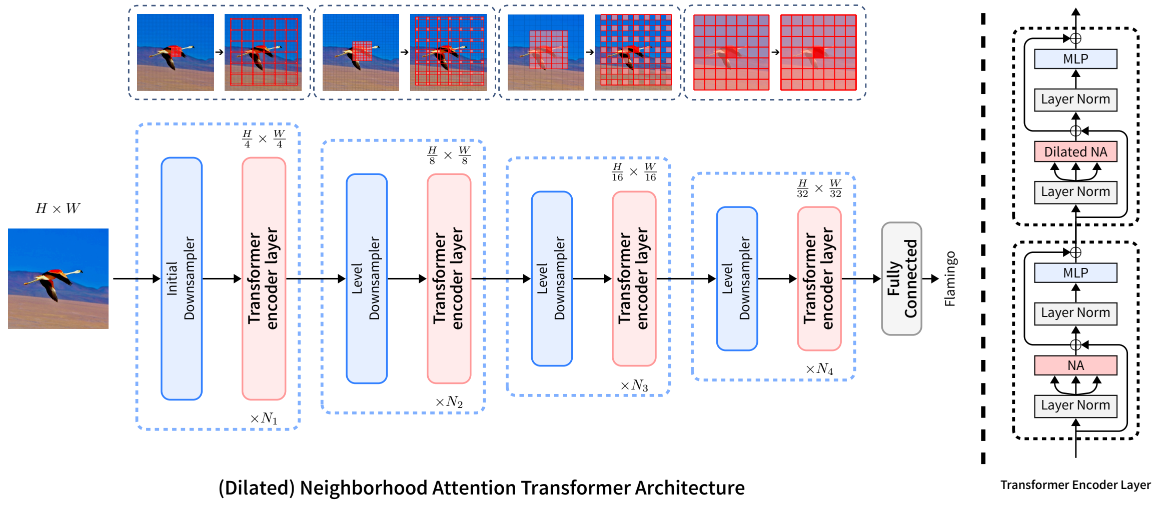
**SWA versus Convolution.** While the method used to extract key-value pairs in SWA is identical to that used in convolution for extracting local patches, the computation that is performed is very different. In convolution, the sliding window operation, sometimes referred to as im2col (image-to-column) is applied to the input feature map, extracting windows over which a static weight matrix is applied. In Attention, as discussed earlier, the two operands are both inputs, and there are no static weights. One input, the queries, are not transformed in any way, and the key-value pair instead goes through im2col. This is the primary difference between convolution and SWA. The former applies static weights to the extracted sliding windows, while the latter uses the extracted sliding windows from the context as weights, using which the queries are transformed.

**Properties specific to NA.** As stated, NA's key difference compared to SWA is in the sliding window pattern itself. NA restricts attention to nearest neighbors, which is not necessarily a window centered around each token. This difference is very important, because explicitly centering attention window around every token results in reduced receptive field for tokens that cannot be centered, which can limit performance. This is because the ratio of such tokens increases linearly with window size and dilation, which would limit the ability to capture meaningful interactions at scale. Defining NA resolves this issue without imposing any additional computation.

This difference is the sole reason why NA windows can be dilated to greater extents compared to SWA and convolution. Given $n$ tokens, window size can be any odd number in the range $[1, n]$, and dilation can be any natural number in the range $[1, \lfloor n/k \rfloor]$. This property cannot be transferred to convolution, as it only results in repeated pixels in the corners, due to the weights (convolution kernel) being static. As a result, convolution is more limited in terms of choices for window size and dilation, as larger window sizes or dilation values require increases padding in order to maintain the spatial size, which in turn only biases the outputs further.

### 2.2.9 Implementation

Implementing sliding window attention patterns at the deep learning framework level, especially in multi-dimensional token layouts, is not trivial. Since most such frameworks mostly offer efficient primitives at the tensor-level, as opposed to fine-grained element-wise primitives, any such implementation would need to perform memory operations (im2col) that expand the tensor by extracting sliding windows. The additional memory overhead from this implementation greatly limits one's ability to even train models with this approach. In our initial experiments, we observed that while training a very small scale Swin model with batch size 1024 on 8 NVIDIA A100 GPUs takes approximately 1.3 days, training the same model but with an im2col-based implementation of NA takes over 9 days. The Swin model trains while utilizing less than 16GB of DRAM on each A100 GPU, while the NA-based model requires cutting down the batch size by half, and still uses around 67GB of DRAM on each GPU.



$H \times W$
Initial Downsampler
Transformer encoder layer
$\frac{H}{4} \times \frac{W}{4}$
$\times N_1$
Level Downsampler
$\frac{H}{8} \times \frac{W}{8}$
$\times N_2$
$\frac{H}{16} \times \frac{W}{16}$
$\times N_3$
$\frac{H}{32} \times \frac{W}{32}$
$\times N_4$
Fully Connected
Flamingo
MLP
Layer Norm
Dilated NA
NA
(Dilated) Neighborhood Attention Transformer Architecture
Transformer Encoder Layer

**Figure 2.4. An illustration of the hierarchical NAT / DiNAT architecture.** Inputs are downsampled to a quarter of their original spatial resolution, and go through 4 levels of Transformer encoders. Feature maps are downsampled to half their spatial size and doubled in channels between levels. In NAT variants, Self Attention is replaced with NA. In DiNAT variants, we alternate between non-dilated and dilated NA throughout the layers.

We therefore created a project dedicated to efficient implementations of NAT, called Neighborhood Attention Extension (NATTEN). These implementations are naive CUDA C++ kernels written from scratch to perform forward pass and backward pass operations for NA, without explicitly performing im2col in global memory. All of these kernels directly load and compute the dot products for queries and their corresponding neighborhoods. By using an early build of NATTEN in the aforementioned mock training setup, the memory usage for the NA-based model drops to that of Swin, and training time drops to 1.8 days. We further optimize some of the kernels by utilizing spatial locality and tiling the computation, which allows using faster on-chip memory to reduce the number of redundant accesses to global memory. The final training time after all of these optimizations is 1.2 days, which is slightly better than that of Swin.

## 2.2.10 Neighborhood Attention Transformer (NAT)

In order to evaluate NA both as a deep learning primitive, and in comparison to full Self Attention, as well as comparable methods such as Blocked Attention, we create vision

**Table 2.1. NAT / DiNAT variants.** Channels (number of attention heads and embedding dimension) double after every level except the last. No dilation is used in NAT variants, whereas in DiNAT variants we alternate between NA with and without dilation.

| **Variant** | **Layers per level** | **Dim × Heads** | **MLP ratio** | **# of Params** | **FLOPs** |
|---|---|---|---|---|---|
| **(Di)NAT-Mini** | 3, 4, 6, 5 | 32 × 2 | 3 | 20 M | 2.7 G |
| **(Di)NAT-Tiny** | 3, 4, 18, 5 | 32 × 2 | 3 | 28 M | 4.3 G |
| **(Di)NAT-Small** | 3, 4, 18, 5 | 32 × 3 | 2 | 51 M | 7.8 G |
| **(Di)NAT-Base** | 3, 4, 18, 5 | 32 × 4 | 2 | 90 M | 13.7 G |
| **(Di)NAT-Large** | 3, 4, 18, 5 | 32 × 6 | 2 | 200 M | 30.6 G |

architectures based on it. These architectures follow the design of hierarchical vision transformers such as Swin [23]. While architecture design is not the focus of this work, we tweaked our architecture to further enhance performance by re-introducing convolutional downsamplers as opposed to non-overlapping, a.k.a patched downsampling, from Swin. This is because we want to preserve translational equivariance as best as possible, especially given that NA also preserves that property. This change adds some additional Floating Point Operations (FLOPs) and parameters, which we undo by reducing the size of inverted bottlenecks, just so the models have similar statistics to competitor variants. We label the new architectural configuration Neighborhood Attention Transformer (NAT), and summarize its variants ranging from 20 million parameters to 200 million parameters in Tab. 2.1, with an illustration of the design in Fig. 2.4. Variants using DiNA are dubbed DiNAT.

## 2.3 Experiments

In this section, we present experiments with multiple vision architectures, including our proposed configuration from Sec. 2.2.10, NAT and DiNAT. We trained these models on image classification, region-based object detection, and image segmentation. Unless otherwise stated, we used NA with window size 7 × 7, following the Blocked Attention window size used in Swin [23], and convolution window size in ConvNeXt [24].

**Table 2.2. ImageNet classification performance with NAT / DiNAT architecture.** Dilation can improve accuracy, with little to no effect on throughput. The exception here are the Mini and Tiny variants. As we will further see, the same variants outperform their non-dilated counterpart in downstream tasks. Throughput and peak memory usage are measured from a forward pass with a batch size of 256 on a single A100 GPU.

| **Model** | **# of Params** | **FLOPs** | **Thru.** (imgs/sec) | **Memory** (GB) | **Top-1** (%) |
|---|---|---|---|---|---|
| **NAT-M** | 20 M | 2.7 G | 2297 | 2.4 | **81.8** |
| **DiNAT-M** | 20 M | 2.7 G | 2245 | 2.4 | **81.8** |
| **NAT-T** | 28 M | 4.3 G | 1664 | 2.5 | **83.2** |
| **DiNAT-T** | 28 M | 4.3 G | 1619 | 2.5 | 82.7 |
| **NAT-S** | 51 M | 7.8 G | 1130 | 3.7 | 83.7 |
| **DiNAT-S** | 51 M | 7.8 G | 1142 | 3.7 | **83.8** |
| **NAT-B** | 90 M | 13.7 G | 856 | 5.0 | 84.3 |
| **DiNAT-B** | 90 M | 13.7 G | 864 | 5.0 | **84.4** |

### 2.3.1 Image classification

Following prior works, we train our image classification models on ImageNet-1K [50] for 300 epochs, the first 20 of which warm up to a learning rate of 1e-3. We used a batch size of 1024, and an additional 10 cooldown epochs in addition to the 300, again following the standard practice [51, 23, 24]. Our large scale variants ($\approx$ 200M parameters) are pre-trained on the ImageNet-22K superset for 90 epochs, and fine-tuned on the original ImageNet-1K for an additional 30 epochs, following the setups from Swin and ConvNeXt.

**NAT and DiNAT.** Tab. 2.2 summarizes the results of our ImageNet-1K experiments with NAT and DiNAT. As expected, dilation has little effect on throughput, but typically improves accuracy. In smaller scale variants, dilated variants either match or fall short of non-dilated variants. However, we found these instances to be the exception, as we will see variants with dilation significantly outperform those without dilation, both in image classification at scale, and downstream tasks. Moreover, we did not observe this exception while experimenting with other architectures, such as Swin, which is presented next.

**Table 2.3. ImageNet classification performance with Swin architecture.** The original Swin architecture alternates between Blocked Attention without and with their proposed cyclic shift. In this table, we present results from replacing both of them with NA, dubbed NAT$_s$. We also present results from replacing the non-shifted variant with NA and shifted with DiNA, dubbed DiNAT$_s$, which creates the same alternating pattern present in DiNAT. We also experiment with replacing Swin's cyclic shifted Blocked Attention with the non-contiguous form [47], which like DiNA can help capture global context, as a somewhat more relevant point of reference for DiNAT$_s$. Throughput and peak memory usage are measured from a forward pass with a batch size of 256 on a single A100 GPU.

| Model | Attn | # of Params | FLOPs | Thru. (imgs/sec) | Memory (GB) | Top-1 (%) |
|---|---|---|---|---|---|---|
| **Swin-T** | BA / shifted BA | 28 M | 4.5 G | 1988 | 4.8 | 81.2 |
| | BA / non-contig BA | 28 M | 4.5 G | 2091 | 4.5 | 81.1 |
| **NAT$_s$-T** | NA / NA | 28 M | 4.5 G | **2146** | **4.0** | **81.8** |
| **DiNAT$_s$-T** | NA / DiNA | 28 M | 4.5 G | 2100 | **4.0** | **81.8** |
| **Swin-S** | BA / shifted BA | 50 M | 8.7 G | 1225 | 5.0 | 83.0 |
| | BA / non-contig BA | 50 M | 8.7 G | 1292 | 4.7 | 82.9 |
| **NAT$_s$-S** | NA / NA | 50 M | 8.7 G | **1323** | **4.1** | 83.2 |
| **DiNAT$_s$-S** | NA / DiNA | 50 M | 8.7 G | 1293 | **4.1** | **83.5** |
| **Swin-B** | BA / shifted BA | 88 M | 15.4 G | 892 | 6.7 | 83.5 |
| | BA / non-contig BA | 88 M | 15.4 G | 934 | 6.3 | 83.1 |
| **NAT$_s$-B** | NA / NA | 88 M | 15.4 G | **979** | **5.5** | 83.5 |
| **DiNAT$_s$-B** | NA / DiNA | 88 M | 15.4 G | 957 | **5.5** | **83.8** |

**Comparison to Swin Transformer.** We replaced Blocked Attention and Blocked Attention with cyclic shift in Swin Transformer variants with NA, and dub the resulting model NAT$_s$. In addition, we created a dilated variant for this new model, where we replace the layers with cyclic shift with DiNA instead, and dub it DiNAT$_s$. Results are presented in Tab. 2.3. In every instance, variants powered by NA can run faster, use less memory (from lack of additional memory operations and lack of pixel shifts, among other reasons), and enjoy improved classification accuracy. We also experimented with replacing only the shifted Blocked Attention with the non-contiguous form of Blocked Attention [47], as this form can capture global context similar to DiNA. While this change slightly reduces memory usage and improves throughput, it suffers from an accuracy drop.

**Large-scale pre-training.** We scaled the aforementioned variants to 200M parameters, which following prior works we pre-trained on the ImageNet-22K superset containing approximately ten times as many training samples, for 90 epochs. We then fine-tuned each

**Table 2.4. Large-scale ImageNet classification performance.** Larger variants ($\approx$ 200M parameter scale) were pre-trained on the ImageNet-22K superset, and fine-tuned on ImageNet-1K. $^{\dagger}$indicates increased window size from $7 \times 7$ to $11 \times 11$ (DiNAT) and $12 \times 12$ (Swin), as a result of the resolution increase from $224 \times 224$ to $384 \times 384$. Throughput and peak memory usage are measured from a forward pass with a batch size of 256 on a single A100 GPU.

| **Model** | **Res.** | **# of Params** | **FLOPs** | **Thru.** (imgs/sec) | **Memory** (GB) | **Top-1** (%) |
|---|---|---|---|---|---|---|
| **Swin-L** | $224^2$ | 197 M | 34.5 G | 540 | 10.4 | 86.3 |
| **DiNAT$_s$-L** | $224^2$ | 197 M | 34.5 G | 585 | 8.6 | 86.5 |
| **DiNAT-L** | $224^2$ | 200 M | 30.6 G | 545 | 7.8 | **86.6** |
| **Swin-L$^{\dagger}$** | $384^2$ | 197 M | 104.0 G | 189 | 32.7 | 87.3 |
| **DiNAT$_s$-L** | $384^2$ | 197 M | 101.5 G | 185 | 22.6 | 87.4 |
| **DiNAT-L** | $384^2$ | 200 M | 89.7 G | 172 | 20.1 | 87.4 |
| **DiNAT-L$^{\dagger}$** | $384^2$ | 200 M | 92.4 G | 135 | 26.9 | **87.5** |

**Table 2.5. ImageNet classification performance with ViT architecture.** NA and DiNA restrict Self Attention, therefore performance drops are expected. However, NA / DiNA require less computation. Throughput and peak memory usage are measured from a forward pass with a batch size of 256 on a single A100 GPU.

| **Model** | **Attn** | **# of Params** | **FLOPs** | **Thru.** (imgs/sec) | **Memory** (GB) | **Top-1** (%) |
|---|---|---|---|---|---|---|
| | **NA/NA** | 22 M | 4.3 G | 3348 | 1.3 | 80.0 |
| | **NA/DiNA** | 22 M | 4.3 G | 3255 | 1.3 | 80.8 |
| **ViT-S** | **SA/SA** | 22 M | 4.6 G | 3070 | 1.9 | **81.2** |
| | **NA/NA** | 86 M | 16.9 G | 1413 | 2.7 | 81.6 |
| | **NA/DiNA** | 86 M | 16.9 G | 1386 | 2.7 | 82.1 |
| **ViT-B** | **SA/SA** | 86 M | 17.5 G | 1288 | 3.7 | **82.5** |

model on ImageNet-1K for 30 epochs, with a batch size of 512, and a linear learning rate schedule with no warmup, and a base learning rate of 5e-5, and weight decay rate of 1e-4, again following prior works. Final ImageNet-1K results are provided in Tab. 2.4. We observe that models based on NA continue to outperform Swin with competitive throughput.

**Comparison to ViT.** We also experimented with restricting Self Attention in the original ViT architecture. We present those experiments in Tab. 2.5, where we replace Self Attention (with relative positional biases) once with NA alone, and once with our standard and dilated interleaving. We keep the $7 \times 7$ window size, similar to previous experiments, but note that

**Table 2.6. ImageNet-1K image classification performance.** †indicates increased window size from 7 × 7 to 11 × 11 (DiNAT) and 12 × 12 (Swin), as a result of the resolution increase from 224 × 224 to 384 × 384. Throughput and peak memory usage are measured from a forward pass with a batch size of 256 on a single A100 GPU.

| **Model** | **Res.** | **# of Params** | **FLOPs** | **Thru.** (imgs/sec) | **Memory** (GB) | **Top-1** (%) |
|---|---|---|---|---|---|---|
| *No pre-training* | | | | | | |
| **DiNAT-M** | $224^2$ | 20 M | 2.7 G | 2245 | 2.4 | **81.8** |
| **Swin-T** | $224^2$ | 28 M | 4.5 G | 1988 | 4.8 | 81.3 |
| **DiNAT$_s$-T** | $224^2$ | 28 M | 4.5 G | 2100 | 4.0 | 81.9 |
| **ConvNeXt-T** | $224^2$ | 28 M | 4.5 G | 1976 | 3.4 | 82.1 |
| **DiNAT-T** | $224^2$ | 28 M | 4.3 G | 1619 | 2.5 | **82.7** |
| **Swin-S** | $224^2$ | 50 M | 8.7 G | 1225 | 5.0 | 83.0 |
| **DiNAT$_s$-S** | $224^2$ | 50 M | 8.7 G | 1293 | 4.1 | 83.5 |
| **ConvNeXt-S** | $224^2$ | 50 M | 8.7 G | 1229 | 3.5 | 83.1 |
| **DiNAT-S** | $224^2$ | 51 M | 7.8 G | 1142 | 3.7 | **83.8** |
| **Swin-B** | $224^2$ | 88 M | 15.4 G | 892 | 6.7 | 83.5 |
| **DiNAT$_s$-B** | $224^2$ | 88 M | 15.4 G | 957 | 5.5 | 83.8 |
| **ConvNeXt-B** | $224^2$ | 89 M | 15.4 G | 887 | 4.8 | 83.8 |
| **DiNAT-B** | $224^2$ | 90 M | 13.7 G | 864 | 5.0 | **84.4** |
| *Pre-trained with ImageNet-22K and fine-tuned* | | | | | | |
| **Swin-L** | $224^2$ | 197 M | 34.5 G | 540 | 10.4 | 86.3 |
| **DiNAT$_s$-L** | $224^2$ | 197 M | 34.5 G | 585 | 8.6 | 86.5 |
| **ConvNeXt-L** | $224^2$ | 198 M | 34.4 G | 531 | 7.5 | **86.6** |
| **DiNAT-L** | $224^2$ | 200 M | 30.6 G | 545 | 7.8 | **86.6** |
| **Swin-L**† | $384^2$ | 197 M | 104.0 G | 189 | 32.7 | 87.3 |
| **DiNAT$_s$-L** | $384^2$ | 197 M | 101.5 G | 185 | 22.6 | 87.4 |
| **ConvNeXt-L** | $384^2$ | 198 M | 101.1 G | 179 | 19.2 | **87.5** |
| **DiNAT-L** | $384^2$ | 200 M | 89.7 G | 172 | 20.1 | 87.4 |
| **DiNAT-L**† | $384^2$ | 200 M | 92.4 G | 135 | 26.9 | **87.5** |

the feature map size in this architecture is 14 × 14. We observe that without dilation, NA alone results in a more significant drop in accuracy compared to the NA and DiNA combinations. This further highlights the importance of using DiNA together with NA to recover global context and accelerate receptive field growth.

**Comparison to ConvNeXt.** ConvNeXt [24] is a modern CNN, which adopts some of the vision transformer design language, specifically hierarchical ones such as Swin [23]. The key difference between ConvNeXt and Swin is that instead of Blocked Attention and

**Table 2.7. MS-COCO object detection and instance segmentation performance using a Mask R-CNN [52] detection head.** Throughput was measured from a single-batch forward pass on a single A100 GPU.

| **Backbone** | **# of Params** | **FLOPs** | **Thru.** (FPS) | **$AP^b$** | **$AP^b_{50}$** | **$AP^b_{75}$** | **$AP^m$** | **$AP^m_{50}$** | **$AP^m_{75}$** |
|---|---|---|---|---|---|---|---|---|---|
| **NAT-M** | 40 M | 225 G | 55.4 | 46.5 | 68.1 | 51.3 | 41.7 | 65.2 | 44.7 |
| **DiNAT-M** | 40 M | 225 G | 54.2 | **47.2** | **69.1** | **51.9** | **42.5** | **66.0** | **45.9** |
| **Swin-T** | 48 M | 267 G | 48.9 | 46.0 | 68.1 | 50.3 | 41.6 | 65.1 | 44.9 |
| **DiNAT$_s$-T** | 48 M | 263 G | 55.0 | 46.6 | 68.8 | 51.3 | 42.1 | 65.7 | 45.4 |
| **ConvNeXt-T** | 48 M | 262 G | 53.1 | 46.2 | 67.0 | 50.8 | 41.7 | 65.0 | 44.9 |
| **NAT-T** | 48 M | 258 G | 44.7 | 47.7 | 69.0 | 52.6 | 42.6 | 66.1 | 45.9 |
| **DiNAT-T** | 48 M | 258 G | 44.8 | **48.6** | **70.2** | **53.4** | **43.5** | **67.3** | **46.8** |
| **Swin-S** | 69 M | 359 G | 34.2 | 48.5 | 70.2 | 53.5 | 43.3 | 67.3 | 46.6 |
| **DiNAT$_s$-S** | 69 M | 350 G | 40.7 | 48.6 | 70.4 | 53.2 | 43.5 | 67.6 | 46.9 |
| **NAT-S** | 70 M | 330 G | 35.7 | 48.4 | 69.8 | 53.2 | 43.2 | 66.9 | 46.5 |
| **DiNAT-S** | 70 M | 330 G | 35.7 | **49.3** | **70.8** | **54.2** | **44.0** | **68.0** | **47.4** |

its linear projections ($Q$, $K$, $V$), they use a depthwise convolution. However, since this design contains far fewer parameters and FLOPs per layer compared to Attention-based models, ConvNeXt variants are also noticeably deeper models. Because of this difference, it is impractical to compare to ConvNeXt by solely replacing the convolution operator with NA. We therefore compare the models already presented, DiNAT$_s$ and DiNAT, to ConvNeXt directly. This comparison is presented in Tab. 2.6. DiNAT$_s$ can reach competitive performance with ConvNeXt, while DiNAT can outperform ConvNeXt without large-scale pre-training. In larger scale models which were pre-trained on ImageNet-22K, DiNAT matches ConvNeXt's performance.

### 2.3.2 Object detection and instance segmentation

We extend the hierarchical classification models to region-based object detection and instance segmentation, which we present in Tabs. 2.7 and 2.8. We conducted these experiments with two the same two objection detection heads as Swin [23] and ConvNeXt [24]: Mask R-CNN [52] and Cascade Mask R-CNN [53]. We observe that DiNAT shows noticeable improvement over NAT, again with minimal impact on throughput. There are even instances where DiNAT even surpasses NAT's throughput, but in our experience it is within

**Table 2.8. MS-COCO object detection and instance segmentation performance using a Cascaded Mask R-CNN [53] detection head.** ‡indicates that the model was pre-trained on ImageNet-22K. ⋆Swin-L performance with Cascade Mask R-CNN was not reported, and we trained it with the relevant checkpoint form their official repository. Throughput was measured from a single-batch forward pass on a single A100 GPU.

| Backbone | # of Params | FLOPs | Thru. (FPS) | $AP^b$ | $AP^b_{50}$ | $AP^b_{75}$ | $AP^m$ | $AP^m_{50}$ | $AP^m_{75}$ |
|---|---|---|---|---|---|---|---|---|---|
| **NAT-M** | 77 M | 704 G | 28.4 | 50.3 | 68.9 | 54.9 | 43.6 | 66.4 | 47.2 |
| **DiNAT-M** | 77 M | 704 G | 28.2 | **51.2** | **69.8** | **55.7** | **44.4** | **67.3** | **47.8** |
| **Swin-T** | 86 M | 745 G | 26.6 | 50.4 | 69.2 | 54.7 | 43.7 | 66.6 | 47.3 |
| **DiNAT$_s$-T** | 86 M | 742 G | 28.0 | 51.0 | 69.9 | 55.4 | 44.1 | 67.3 | 47.6 |
| **ConvNeXt-T** | 86 M | 741 G | 27.7 | 50.4 | 69.1 | 54.8 | 43.7 | 66.5 | 47.3 |
| **NAT-T** | 85 M | 737 G | 24.8 | 51.4 | 70.0 | 55.9 | 44.5 | 67.6 | 47.9 |
| **DiNAT-T** | 85 M | 737 G | 25.3 | **52.2** | **71.0** | **56.8** | **45.1** | **68.3** | **48.8** |
| **Swin-S** | 107 M | 838 G | 21.6 | 51.8 | 70.4 | 56.3 | 44.7 | 67.9 | 48.5 |
| **DiNAT$_s$-S** | 107 M | 829 G | 24.0 | 52.3 | 71.2 | 56.7 | 45.2 | 68.6 | 49.1 |
| **ConvNeXt-S** | 108 M | 827 G | 23.5 | 51.9 | 70.8 | 56.5 | 45.0 | 68.4 | 49.1 |
| **NAT-S** | 108 M | 809 G | 22.2 | 52.0 | 70.4 | 56.3 | 44.9 | 68.1 | 48.6 |
| **DiNAT-S** | 108 M | 809 G | 22.1 | **52.9** | **71.8** | **57.6** | **45.8** | **69.3** | **49.9** |
| **Swin-B** | 145 M | 982 G | 18.6 | 51.9 | 70.9 | 56.5 | 45.0 | 68.4 | 48.7 |
| **DiNAT$_s$-B** | 145 M | 966 G | 20.7 | 52.6 | 71.5 | 57.2 | 45.3 | 68.8 | 49.1 |
| **ConvNeXt-B** | 146 M | 964 G | 19.7 | 52.7 | 71.3 | 57.2 | 45.6 | 68.9 | 49.5 |
| **NAT-B** | 147 M | 931 G | 19.1 | 52.3 | 70.9 | 56.9 | 45.1 | 68.3 | 49.1 |
| **DiNAT-B** | 147 M | 931 G | 18.7 | **53.4** | **72.1** | **58.2** | **46.2** | **69.7** | **50.2** |
| **Swin-L**⋆‡ | 253 M | 1393 G | 14.1 | 53.7 | 72.2 | 58.7 | 46.4 | 69.9 | 50.7 |
| **DiNAT$_s$-L**‡ | 253 M | 1357 G | 15.8 | 54.8 | 74.2 | 59.8 | 47.2 | 71.3 | 51.2 |
| **ConvNeXt-L**‡ | 253 M | 1354 G | 15.1 | 54.8 | 73.8 | 59.8 | 47.6 | 71.3 | 51.7 |
| **DiNAT-L**‡ | 258 M | 1276 G | 14.1 | **55.3** | **74.3** | **60.2** | **47.8** | **71.8** | **52.0** |

the margin of error for this particular task. We also find that similar to classification, DiNAT$_s$ enjoys higher throughput and improved performance compared to the original Swin variants based on Blocked Attention. Additionally, we observe that DiNAT stays ahead of ConvNeXt [24], including in large scale variants, which was not the case in classification.

### 2.3.3 Semantic segmentation

Tab. 2.9 presents results in training a semantic segmentation framework, UPerNet [54], with the aforementioned architectures. We observe again that DiNAT always outperforms NAT with very little drop in throughput, and DiNAT$_s$ improves Swin in both throughput and performance. In addition, we observe DiNAT maintains its place ahead of both models, as well as ConvNeXt, at scale with ImageNet-22K pre-training.

**Table 2.9. ADE20K semantic segmentation performance using UPerNet [54].** ‡indicates that the model was pre-trained on ImageNet-22K. †indicates increased window size from the default 7 × 7 to 12 × 12. Throughput was measured from a single-batch forward pass on a single A100 GPU.

| **Backbone** | **Res.** | **# of Params** | **FLOPs** | **Thru.** (FPS) | **mIoU** single scale | multi scale |
|---|---|---|---|---|---|---|
| **NAT-M** | 2048 × 512 | 50 M | 900 G | 24.9 | 45.1 | 46.4 |
| **DiNAT-M** | 2048 × 512 | 50 M | 900 G | 24.7 | **45.8** | **47.2** |
| **Swin-T** | 2048 × 512 | 60 M | 946 G | 23.0 | 44.5 | 45.8 |
| **DiNAT$_s$-T** | 2048 × 512 | 60 M | 941 G | 24.6 | 46.0 | 47.4 |
| **ConvNeXt-T** | 2048 × 512 | 60 M | 939 G | 23.9 | 46.0 | 46.7 |
| **NAT-T** | 2048 × 512 | 58 M | 934 G | 22.3 | 47.1 | 48.4 |
| **DiNAT-T** | 2048 × 512 | 58 M | 934 G | 22.0 | **47.8** | **48.8** |
| **Swin-S** | 2048 × 512 | 81 M | 1040 G | 18.6 | 47.6 | 49.5 |
| **DiNAT$_s$-S** | 2048 × 512 | 81 M | 1030 G | 20.6 | 48.6 | **49.9** |
| **ConvNeXt-S** | 2048 × 512 | 82 M | 1027 G | 19.6 | 48.7 | 49.6 |
| **NAT-S** | 2048 × 512 | 82 M | 1010 G | 18.6 | 48.0 | 49.5 |
| **DiNAT-S** | 2048 × 512 | 82 M | 1010 G | 18.6 | **48.9** | **49.9** |
| **Swin-B** | 2048 × 512 | 121 M | 1188 G | 16.0 | 48.1 | 49.7 |
| **DiNAT$_s$-B** | 2048 × 512 | 121 M | 1173 G | 17.9 | 49.4 | 50.2 |
| **ConvNeXt-B** | 2048 × 512 | 122 M | 1170 G | 16.9 | 49.1 | 49.9 |
| **NAT-B** | 2048 × 512 | 123 M | 1137 G | 16.1 | 48.5 | 49.7 |
| **DiNAT-B** | 2048 × 512 | 123 M | 1137 G | 15.9 | **49.6** | **50.4** |
| **Swin-L**†‡ | 2560 × 640 | 234 M | 2585 G | 12.0 | - | 53.5 |
| **DiNAT$_s$-L**‡ | 2560 × 640 | 234 M | 2466 G | 12.6 | 53.4 | 54.6 |
| **ConvNeXt-L**‡ | 2560 × 640 | 235 M | 2458 G | 12.4 | 53.2 | 53.7 |
| **DiNAT-L**‡ | 2560 × 640 | 238 M | 2335 G | 11.7 | **54.0** | **54.9** |

### 2.3.4 Image segmentation with Mask2Former

We also experiment with another segmentation framework, Mask2Former [55], which is a Transformer-based segmentation framework capable of targeting instance, semantic, and panoptic segmentation. The original work experimented with Swin-L, and we therefore limited our experiment to our large DiNAT variant. We trained Mask2Former on MS-COCO [56], ADE20K [57], and Cityscapes [58], on all segmentation objectives for which the datasets provided annotations. Results are presented in Tabs. 2.10 to 2.12. Note that DiNAT-L is using an 11 × 11 window size, instead of Swin-L's 12 × 12, since even-sized windows break the symmetry in NA and are therefore not defined. DiNAT-L outperforms Swin-L on all three tasks and datasets in the primary metrics.

**Table 2.10. Instance segmentation performance with Mask2Former.**

| Backbone | Win. Size | # of Params | FLOPs | AP | $AP^{50}$ | $AP^{S}$ | $AP^{M}$ | $AP^{L}$ |
|---|---|---|---|---|---|---|---|---|
| | | | *MS-COCO* | | | | | |
| **Swin-L** | 12 × 12 | 216 M | 641 G | 50.1 | - | 29.9 | 53.9 | **72.1** |
| **DiNAT-L** | 11 × 11 | 220 M | 522 G | **50.8** | **75.0** | **30.9** | **54.7** | **72.1** |
| | | | *ADE20K* | | | | | |
| **Swin-L** | 12 × 12 | 216 M | 654 G | 34.9 | - | **16.3** | **40.0** | 54.7 |
| **DiNAT-L** | 11 × 11 | 220 M | 535 G | **35.4** | - | **16.3** | 39.0 | **55.5** |
| | | | *Cityscapes* | | | | | |
| **Swin-L** | 12 × 12 | 216 M | 641 G | 43.7 | 71.4 | - | - | |
| **DiNAT-L** | 11 × 11 | 220 M | 522 G | **45.1** | **72.6** | - | - | - |

**Table 2.11. Semantic segmentation performance with Mask2Former.**

| Backbone | Win. Size | # of Params | FLOPs | mIoU single scale | mIoU multi scale |
|---|---|---|---|---|---|
| | | | *ADE20K* | | |
| **Swin-L** | 12 × 12 | 215 M | 636 G | 56.1 | 57.3 |
| **DiNAT-L** | 11 × 11 | 220 M | 518 G | **57.3** | **58.1** |
| | | | *Cityscapes* | | |
| **Swin-L** | 12 × 12 | 215 M | 627 G | 83.3 | 84.3 |
| **DiNAT-L** | 11 × 11 | 220 M | 509 G | **83.9** | **84.5** |

### 2.3.5 Window size, dilation size, and order of layers

In this subsection, we present experiments aimed at finding performance differences between different window sizes, dilation values, orderings of NA and DiNA throughout layers, and changes to dilation values during inference.

**Dilation values.** Tab. 2.13 summarizes effects of different dilation values on classification, detection, instance segmentation and semantic segmentation. We denote dilation settings on a per-level basis (all hierarchical vision transformers in this work have 4 levels with 2 × 2 spatial downsampling in between) and as follows: ($x$, $y$, $z$, $w$), where the first level uses a dilation value of $x$, second level uses dilation $y$, third level uses dilation $z$, and the fourth level uses dilation $w$. Since dilation can be extended only to the bounds of the input feature map (window size $\times$ dilation $\leq$ input size must always be satisfied), some dilation settings such as (16, 8, 4, 2) are only applicable to downstream tasks. (8, 4, 2, 1) is the maximum applicable dilation to ImageNet at 224 × 224 resolution.

**Table 2.12. Panoptic segmentation performance with Mask2Former.**

| Backbone | Win. Size | # of Params | FLOPs | PQ | $PQ^{Th}$ | $PQ^{St}$ | $AP^{Th}_{pan}$ | $mIoU_{pan}$ |
|---|---|---|---|---|---|---|---|---|
| | | | | *MS-COCO* | | | | |
| **Swin-L** | 12 × 12 | 216 M | 658 G | 57.8 | 64.2 | 48.1 | 48.6 | 67.4 |
| **DiNAT-L** | 11 × 11 | 220 M | 540 G | **58.5** | **64.9** | **48.8** | **49.2** | **68.3** |
| | | | | *ADE20K* | | | | |
| **Swin-L** | 12 × 12 | 216 M | 660 G | 48.1 | - | - | 34.2 | 54.5 |
| **DiNAT-L** | 11 × 11 | 220 M | 542 G | **49.4** | - | - | **35.0** | **56.3** |
| | | | | *Cityscapes* | | | | |
| **Swin-L** | 12 × 12 | 216 M | 643 G | 66.6 | - | - | 43.6 | 82.9 |
| **DiNAT-L** | 11 × 11 | 220 M | 525 G | **67.2** | - | - | **44.5** | **83.4** |

**Table 2.13. Effect of different dilation values on model accuracy.** Dilation values beyond “8, 4, 2, 1” are only applicable to downstream tasks, as their larger resolution allows for it. Maximum dilation indicates it is set to the maximum possible value based on input size. It is the same as “8, 4, 2, 1” for ImageNet. Gradual dilation indicates that dilation values in DiNA layers increase gradually.

| Model | Dilation per level | ImageNet Top-1 (%) | MS-COCO $AP^b$ | MS-COCO $AP^m$ | ADE20K mIoU |
|---|---|---|---|---|---|
| **Swin-Tiny** | Not Applicable | 81.3 | 46.0 | 41.6 | 45.8 |
| **NAT$_s$-Tiny** | 1, 1, 1, 1 | 81.8 | 46.1 | 41.5 | 46.2 |
| **DiNAT$_s$-Tiny** | 8, 4, 2, 1 | 81.8 | 46.3 | 41.6 | 46.7 |
| **DiNAT$_s$-Tiny** | 16, 8, 4, 2 | - | **46.4** | **41.8** | 47.1 |
| **DiNAT$_s$-Tiny** | Maximum | 81.8 | **46.4** | **41.9** | 47.0 |
| **DiNAT$_s$-Tiny** | Gradual | - | **46.6** | **42.1** | **47.4** |
| **NAT-Tiny** | 1, 1, 1, 1 | 83.2 | 47.7 | 42.6 | 48.4 |
| **DiNAT-Tiny** | 8, 4, 2, 1 | 82.7 | 48.0 | 42.9 | 48.5 |
| **DiNAT-Tiny** | 16, 8, 4, 2 | - | 48.3 | 43.4 | 48.5 |
| **DiNAT-Tiny** | Maximum | 82.7 | **48.6** | **43.5** | 48.7 |
| **DiNAT-Tiny** | Gradual | - | **48.6** | **43.5** | **48.8** |

We also explored input-dependent dilation values, where DiNA layers apply the maximum possible dilation, which is $\lfloor$ input size / window size $\rfloor$ (“Maximum” in Tab. 2.13). Our final setting is “Gradual”, in which we gradually increase dilation to a pre-defined maximum throughout the layers in the level. More details are presented in Sec. 2.3.6.

**Different NA / DiNA interleavings.** We also experimented with models with DiNA layers before NA layers, as opposed to our final choice in which we begin with NA layers. While the local-global order was our initial choice, we’ve also found it to be the more per-

**Table 2.14. Effect of different layer-wise interleaving of NA and DiNA on model accuracy.** Our final model uses the local-global (NA-DiNA) order.

| Variant | Layer structure | ImageNet Top-1 (%) | MS-COCO $AP^b$ | $AP^m$ | ADE20K mIoU |
|---|---|---|---|---|---|
| **Swin-Tiny** | BA / shifted BA | 81.3 | 46.0 | 41.6 | 45.8 |
| **NAT$_s$-Tiny** | NA / NA | **81.8** | 46.1 | 41.5 | 46.2 |
| **DiNAT$_s$-Tiny** | NA / DiNA | **81.8** | 46.4 | **41.8** | **47.1** |
| | DiNA / NA | 81.5 | **46.5** | **41.8** | 46.9 |
| | DiNA / DiNA | 79.7 | 39.8 | 36.8 | 40.7 |
| **NAT-Tiny** | NA / NA | **83.2** | 47.7 | 42.6 | 48.4 |
| **DiNAT-Tiny** | NA / DiNA | 82.7 | 48.3 | 43.4 | **48.5** |
| | DiNA / NA | 82.6 | **48.5** | **43.5** | 47.9 |
| | DiNA / DiNA | 82.2 | 44.9 | 40.5 | 45.8 |

formant choice. We also tried a model with only DiNA modules, and found that it performs significantly worse than other combinations. This highlights the importance of preserving locality, and only relying on non-contiguous patterns as an additional component. Results are summarized in Tab. 2.14.

**Window size.** We summarize experiments with different window sizes in Tab. 2.15. We observed that a DiNAT-Tiny sees a significant decay in performance with a smaller window size across all three tasks. However, we find increasing window size beyond the default 7 × 7 does not result in a significant increase in performance. This aligns with the common notion that Self Attention can sometime be redundant, because models can converge to local masks.

**Test-time dilation changes.** We present an analysis of test-time sensitivity to changing dilation values, in which we attempt different dilation values on already trained models, and evaluate their performance. This can be particularly important to cases with varying resolutions (i.e. multi-scale testing). Results are presented in Tab. 2.16.

**Table 2.15. Effect of window size on model accuracy.** Note that we set dilation to the maximum values possible in each block based on the default resolutions. Therefore, the variant with a 5 × 5 window size has larger dilation values compared to the one with a 7 × 7 window size.

| Model | Win. | ImageNet | | MS-COCO | | | ADE20K | |
|---|---|---|---|---|---|---|---|---|
| | size | Top-1 | Thru. | AP$^b$ | AP$^m$ | Thru. | mIoU | Thru. |
| **NAT-T** | $5^2$ | **81.6** | 1828 imgs/sec | 46.8 | 42.0 | 48.4 fps | 46.3 | 23.4 fps |
| **DiNAT-T** | $5^2$ | 81.3 | 1788 imgs/sec | **47.6** | **42.7** | 48.1 fps | **46.4** | 23.2 fps |
| **NAT-T** | $7^2$ | **83.2** | 1664 imgs/sec | 47.7 | 42.6 | 44.7 fps | 48.4 | 22.3 fps |
| **DiNAT-T** | $7^2$ | 82.7 | 1619 imgs/sec | **48.3** | **43.4** | 44.8 fps | **48.5** | 22.0 fps |
| **NAT-T** | $9^2$ | **83.1** | 1263 imgs/sec | 48.5 | 43.3 | 40.7 fps | 48.1 | 20.6 fps |
| **DiNAT-T** | $9^2$ | **83.1** | 1245 imgs/sec | **48.8** | **43.5** | 40.2 fps | **48.4** | 20.4 fps |

**Table 2.16. Effect of changing dilation values during inference on model accuracy.** Dilation values larger than 8, 4, 2, 1 are inapplicable to ImageNet at 224 × 224.

| Model | Dilation | | ImageNet | MS-COCO | | ADE20K |
|---|---|---|---|---|---|---|
| | Train | Test | Top-1 (%) | AP$^b$ | AP$^m$ | mIoU |
| **NAT-T** | 1, 1, 1, 1 | 1, 1, 1, 1 | 83.2 | 47.7 | 42.6 | 48.4 |
| | 1, 1, 1, 1 | 8, 4, 2, 1 | 81.0 | 42.6 | 39.5 | 46.3 |
| | 1, 1, 1, 1 | 16, 8, 4, 2 | - | 36.0 | 34.4 | 40.2 |
| | 1, 1, 1, 1 | Maximum | - | 31.7 | 30.7 | 38.2 |
| | 8, 4, 2, 1 | 1, 1, 1, 1 | 78.2 | 43.0 | 38.6 | 41.5 |
| **DiNAT-T** | 8, 4, 2, 1 | 8, 4, 2, 1 | 82.7 | 48.0 | 42.9 | 48.5 |
| | 8, 4, 2, 1 | 16, 8, 4, 2 | - | 45.6 | 41.3 | 47.1 |
| | 8, 4, 2, 1 | Maximum | - | 40.2 | 37.3 | 45.8 |
| | 16, 8, 4, 2 | 1, 1, 1, 1 | - | 29.0 | 26.7 | 26.2 |
| | 16, 8, 4, 2 | 8, 4, 2, 1 | - | 42.6 | 38.6 | 43.3 |
| **DiNAT-T** | 16, 8, 4, 2 | 16, 8, 4, 2 | - | 48.3 | 43.4 | 48.5 |
| | 16, 8, 4, 2 | Maximum | - | 47.4 | 42.5 | 48.6 |

### 2.3.6 Training settings

We followed Swin Transformer [23] and ConvNeXt [24] in choosing all hyperparameters and training settings. All image classification models, with the exception of "Large" variants, are trained on ImageNet-1K [50] using the training recipe from `timm` [51] (Apache License v2). Every classification training job, including those for "Large" variants, follow Swin and ConvNeXt in choice of regularization techniques, augmentations (CutMix [59], Mixup [60], RandAugment [61], Random Erasing [62]) and other all other hyperparame-

**Table 2.17. Dilation patterns used for different resolutions.** Due to ImageNet's relatively small input resolution, level 4 layers cannot go beyond a dilation value of 1. At ImageNet's 224×224 resolution, level 4 feature maps will be exactly 7×7, therefore NA will be equivalent to Self Attention. This is not true in downstream tasks where resolutions are noticeably higher and levels 2 and 3 have *gradually* increasing dilation values, which are repeated in deeper models. This corresponds to the highlighted rows in Tab. 2.13 labeled "Gradual". These configurations apply to all downstream experiments (excluding those in Sec. 2.3.5).

| **Resolution** | **Dilation pattern** | | | |
|---|---|---|---|---|
| | **Level 1** | **Level 2** | **Level 3** | **Level 4** |
| *ImageNet classification.* | | | | |
| 224 × 224 | 1, 8 | 1, 4 | 1, 2 | 1, 1 |
| 384 × 384 | 1, 13 | 1, 6 | 1, 3 | 1, 1 |
| *MS-COCO detection and instance segmentation.* | | | | |
| 800 × 800 | 1, 28 | 1, 14 | 1, 3, 1, 5, 1, 7 | 1, 3 |
| 800 × 800 | 1, 28 | 1, 14 | 1, 3, 1, 5, 1, 7 | 1, 3 |
| *ADE20K semantic segmentation.* | | | | |
| 512 × 512 | 1, 16 | 1, 8 | 1, 2, 1, 3, 1, 4 | 1, 2 |
| 640 × 640 | 1, 20 | 1, 10 | 1, 2, 1, 3, 1, 4, 1, 5 | 1, 2 |

ters (300 epochs with a batch size of 1024, iteration-wise cosine learning rate schedule, 20 epoch warmup, 1e-3 learning rate, 5e-2 weight decay, extra 10 cooldown epochs.)

As previously mentioned, all models, unless explicitly stated, use a 7 × 7 window size NA, and dilation values are selected with respect to input resolution. For example, in ImageNet classification at 224 × 224 resolution, inputs are downsampled with a 4 × 4 ratio via Patch Embed / initial convolution, and subsequent levels, except the final level, downsample by another 2 × 2. Therefore, Level 1 layers take as input 56 × 56 feature maps. With a window size of 7 × 7, the maximum possible dilation value is 8 × 8 since $\lfloor 56/7 \rfloor = 8$. Level 2 layers will take as input 28 × 28 as input, which means the maximum dilation is 4 × 4. Level 3 will be limited to dilation 2 × 2 and the last level cannot dilate. Because of this dependency between the upper bound dilation and input resolution, we set dilation values depending on the task and resolution. We present the final dilation values we used in classification, detection, and segmentation in Tab. 2.17. Note that we only change dilation values for DiNA layers, since we found that fine-tuning NA layers to DiNA layers may result in a slight decrease in initial performance (see Sec. 2.3.5, Tab. 2.16).

Object detection and instance segmentation models were trained on the MS-COCO dataset [56], again following Swin [23] and ConvNeXt [24]. We also followed those works in training settings for `mmdetection` [63] (Apache License v2), and trained with the same accelerated $3\times$ learning rate schedule. The same goes for our semantic segmentation experiments on ADE20K [57], which were done using `mmsegmentation` [64] (Apache License v2).

## 2.4 Conclusion

In this chapter, we introduced Neighborhood Attention (NA), which restricts Self Attention to nearest neighboring tokens, resulting in an explicit sliding window behavior. We discussed prior Sliding Window Attention (SWA) patterns, and discussed a key issue in their formulation, but more importantly the challenges of implementation and infrastructure. NA's definition resolves the aforementioned issue in formulation, and opens up a spectrum of possible attention patterns between no attention (linear projection) and Self Attention (see Fig. 2.1). Applying NA to any problem where Self Attention has already been found useful is itself a new research direction, as NA gives the researchers control over precisely how much static sparsity is introduced, and very fine control over the pattern (local, dilated, causal). In theory, any Self Attention operation can be replaced with NA, without affecting the rest of the architecture, and potentially even without the necessity for further fine-tuning. This, however, is not the only application of NA. Another avenue is applying NA with less aggressive downsampling, in order to enhance performance, while still maintaining a tractable computational cost.

Through our implementation, which we package as a PyTorch extension, NATTEN, we scaled models based on NA up to 200 million parameters, and achieved competitive performance compared to former and current state-of-the-art architectures. This was illustrated by our experiments on image classification, object detection, and image segmentation. In addition, other independent works have extended NA to style-based image generation [65], RGB-space image generation with diffusion [27], motion forecasting [66], and weather prediction [28].

However, infrastructure still remains a challenge, as our naive kernels introduced in this chapter only support very specific use cases, and will suffer greatly in newer GPU architectures. In addition, the infrastructure for Attention itself has shifted greatly thanks to FMHA methods [26, 67, 68], which changes the baselines entirely. We present our extended work on infrastructure in the next chapter.

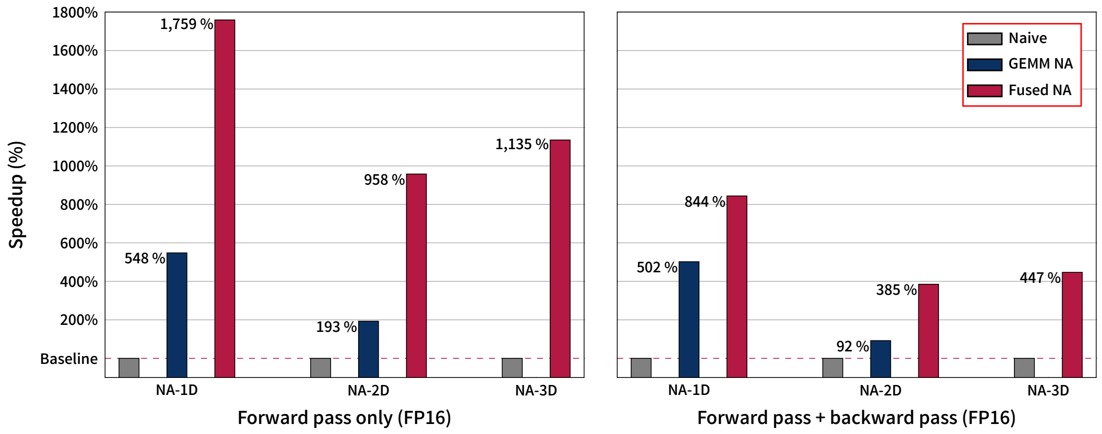


**Figure 3.1. Overview of average speedup from GEMM-based and fused implementations of NA.** Baseline is the implementation from Sec. 2.2.9. GEMM-based NA improves 1-D problems by an average of 548% (forward pass) and 502% (forward + backward), and 2-D problems by an average of 193% (forward pass) and 92% (forward + backward). 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).

# CHAPTER 3
# FASTER NEIGHBORHOOD ATTENTION

Sliding Window Attention (SWA) is a powerful deep learning primitive, and an effective technique for introducing sparsity into Attention-based models [9, 10, 7, 29, 11, 69, 33, 34, 28]. Neighborhood Attention (NA), which we introduced in the last chapter, is an extension of SWA into a family of attention patterns designed for multi-dimensional inputs, predominantly seen in visual AI models and applications. These methods have long been limited by infrastructure [9, 10, 23], which is why we developed custom CUDA C++ kernels designed specifically for use cases of interest in our experiments with the Neighborhood Attention Transformer (NAT) architecture in the last chapter. These implementations were packaged as a deep learning extension called NATTEN, allowing other researchers to use this family of patterns in their work. This, however, is not enough, as they are only proof of concept implementations, unable to deliver reasonable computational performance for many applications that go beyond the scope of the last chapter. In addition, Fused Multi-Headed Attention (FMHA) approaches significantly improve the baseline, Self Attention, in terms of computational performance, leading to an even larger gap to fill in order to fulfill the promise of speedups from saved computation.

In this chapter, we introduce new implementations of NA that pack the existing General Vector-Matrix Multiplication (GEMV) problems (see Sec. 2.2.1) into General Matrix-Matrix Multiplication (GEMM) problems. This unlocks higher levels of computational performance, as we can now target specialized dense linear algebra accelerators, specifically Tensor Cores (TCs), in addition to getting closer to conventional implementations of Self Attention. We then extend this methodology by creating FMHA-inspired kernels, which we dub Fused Neighborhood Attention (FNA). This new implementation boosts computational performance even further, resulting in an average 2.6× speedup over the unfused GEMM-based implementation. These improvements translate into end-to-end speedups in existing models and applications, while also unlocking new applications.

Most of the materials in this chapter are from our NeurIPS 2023 publication titled "Faster Neighborhood Attention: Reducing the $\mathcal{O}(n^2)$ Cost of Self Attention at the Threadblock Level".

## 3.1 Background and related works

Herein we discuss the evolution of Attention implementations and state-of-the-art FMHA methods. Attention was considered quadratic in both space and time complexity for years, until FMHA approaches [25, 26], which evade the full materialization of attention weights in memory, became the standard. We then discuss the shortcomings of the naive implementations of NA in CUDA C++ introduced in the last chapter, which in addition to the rise of FMHA methods, serve as our motivation behind the work presented in this chapter.

### 3.1.1 Fused Multi-Headed Attention (FMHA)

Attention, specifically Scaled Dot Product Attention (SDPA) (see Eq. (2.1)), has historically been implemented as two GEMMs, separated by the softmax operator. Softmax creates a challenge for distributed implementations of Attention, since it involves a reduction operation (the sum of exponents). In the case of SDPA, that reduction is over all the dot products corresponding to a certain query. Given that the number of those dot products can easily grow to thousands, and today even hundreds of thousands, this quickly becomes a bottleneck in an otherwise highly parallelizable computation.

**Online Softmax.** Milakov and Gimelshein [32] presented a technique for computing partial softmax statistics, over which we can perform a final reduction step and derive exact softmax results. This simple approach unlocks distributed forms of softmax, and led to the development of more memory-efficient implementations of Attention.

**The Original FMHA.** To our knowledge, the first open-source implementation of an Attention kernel that avoided fully materializing the dot products / attention weights in memory was was contributed to the NVIDIA Apex project [1] by Young-Jun Ko. This implementation however did not adopt Online Softmax, and was therefore limited to very specific problem sizes. Its primary application was accelerated inference for Transformer-based language models.

**The Rabe-Staats Approach.** Rabe and Staats [25] used Online Softmax to create an implementation of Attention that like the aforementioned approach avoided materializing the full dot products in memory. Also referred to as *memory-efficient attention*, this work illustrated the approach in JAX [70] and for Tensor Processing Units (TPUs). The implementation supported backpropagation, but relied on automatic differentiation, rather than explicitly creating a backward pass operator. This approach successfully scaled up to a 1

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

million token sequence without running out of memory, unlike the baseline which ran out of memory at around 65 thousand tokens, both reported from running on a TPU v3 chip with 16 gigabytes of memory. It also reportedly ran with similar runtime compared to the baseline.

**Flash Attention.** Dao et al. [26] presented and open-sourced Flash Attention, which also utilizes Online Softmax, in order to create a performant and generic fused attention implementation. Unlike Rabe-Staats, Flash Attention was implemented directly in CUDA C++, which in addition to avoiding the materialization of attention weights in global memory also attempts to minimize global memory accesses, and keeping dot products, softmax statistics, and accumulated partial results (when possible) in local (on-chip) memory. Outperforming unfused implementations available in both training and inference, Flash Attention was quickly adopted by many frameworks such as PyTorch [48]. It was also further improved for the NVIDIA Ampere architecture in Flash Attention 2 [67], which relied on the NVIDIA CUDA Templates for Linear Algebra Subroutines (CUTLASS) framework [71] and the tensor layout algebra and primitives form CUDA Tensors (CuTe). Flash Attention 3 [68] introduced kernels specialized for the NVIDIA Hopper architecture, further relying on abstractions from CUTLASS and CuTe in adopting the changes in the CUDA programming model, and the new hardware components. Other implementations of this concept have been created in various frameworks. The xFormers [72] FMHA kernels in CUTLASS support multiple architectures older than Ampere, and Ampere itself, as well as full precision or 32-bit Floating Point [float32] (FP32), while most other implementations tend to only support half precision: 16-bit Floating Point [float16] (FP16) and 16-bit Brain Floating Point [bfloat16] (BF16). Implementations in various other frameworks such as Triton [73] have also been created, in addition to other platform-specific implementations for AMD hardware, Apple Silicon, and even CPU-based implementations.

### 3.1.2 Limitations of existing NA implementations

Our custom CUDA C++ kernels described in Sec. 2.2.9 enabled our experiments, and many other studies [65, 27, 74]. However, those implementations do not achieve reasonable hardware utility, nor are designed for it. Some specific use cases of interest were further performance-optimized, but generalizing those optimizations is very difficult. SDPA is also a computationally intensive operation, and with more efficient implementations, such as FMHA methods, said implementations of NA are unlikely to deliver any meaningful speedup, despite the only functional difference between NA and SDPA being fewer FLOPs. The most important limitation of the existing implementation is that they are vector-matrix multiplications (GEMVs), which are fundamentally bound by memory bandwidth instead of computation, and cannot target hardware units such as TCs. Targeting TCs is essential for utility, as they can achieve up to 29 times the throughput of the alternative: Fused Multiply and Add (FMA) math via CUDA cores [2]. A naive targeting of TCs to perform vector-matrix multiplication can likewise suffer in terms of throughput if too many of the individual FLOPs within each TC operation are wasted (from accumulating zeros). We therefore design an NA implementation that packs the individual vector-matrix multiplies (GEMVs) into matrix-matrix multiplies (GEMMs). This implementation preserves spatial locality in multi-dimensional layouts of tokens by performing the tiling / blocking of the operands in multiple dimensions. This multi-dimensional tiling increases the intersection of neighbors corresponding to the queries within a tile, reducing "wasted" FLOPs. We then directly extend this concept to FMHA methods, and create Fused Neighborhood Attention (FNA), which again is based on preserving spatial locality by tiling in multiple dimensions. FNA is our final implementation, massively outperforming the GEMM-based implementation, and our previous naive implementation. Both implementations are based on CUTLASS [71], and rely on its design hierarchy and GEMM abstractions.

[2]Ampere FP32 math peak is 19.5 TFLOPs/sec, TC math is 156 TFLOPs/sec. Blackwell FP32 math peak is 75 TFLOPs/sec, TC math is 2.2 PFLOPs/sec.

## 3.2 Methodology

Herein we describe three primary operations are required to implement unfused Neighborhood Attention forward and backward passes. All three are fundamentally GEMV problems, but we demonstrate how we can represent them with GEMMs, with modifications to the scheduling logic, and the fusion of a scatter / gather operator. We then discuss our key finding, that the scatter / gather is a bottleneck for these implementations which greatly limits their low-precision performance. We then introduce Fused Neighborhood Attention (FNA), which builds on our GEMM-based implementation. This approach escapes the aforementioned bottleneck and successfully boosts lower-precision performance, as attention weights are not materialized in global memory in FMHA methods by definition.

### 3.2.1 Operators

A standard unfused attention forward pass (excluding softmax) is comprised of two operations: $QK^T$, which produces the dot products ($A$), and $PV$, which applies attention weights ($P = softmax(A)$) to values ($V$). The first operation reduces the embedding / head dimension, and maps each query to dot products corresponding to each context token. The second operation reduces the dimension corresponding to context tokens, by pairing each one with a context value, mapping the dot products for each query to a new output vector the size of each value vector. Both of these operations are matrix-matrix multiplications (GEMMs), but with different operand layouts ($K$ is transposed). For backpropagation we can similarly use GEMMs to compute gradients for $Q$, $K$, $V$ and $P$. The gradient for $A$ is computed using the softmax backward operator.

In the case of NA, the first operation creates a new dimension that is not the size of the entire context set, and instead the size of the NA window. For each query (vector), we need to identify the subset of key tokens (matrix) in its neighborhood, and compute the dot products between the vector and matrix (see Sec. 2.2.1). In the second operation, we must load the previous dot products, but pair them with the correct subset of value tokens, before performing the vector-matrix multiplication. This creates a divergence in the implementations of the two, even though the underlying GEMV logic is identical.

We found that these operations, as with standard Attention, can also be reused to compute the backward pass for some of the tensors. The first operation, which we will henceforth refer to as Pointwise-Neighborhood Operator (PN), can also be used to compute the gradient for attention weights, $P$ ($\nabla P$). The second operation, henceforth referred to as Neighborhood-Neighborhood Operator (NN), can compute the gradient for $Q$ ($\nabla Q$).

There is a third operation, which is similar to NN, but inverses the neighborhood mapping, which normally maps query coordinate to context coordinates. The operation computing gradients for $K$ and $V$ ($\nabla K$ and $\nabla V$) works similarly to NN, but with the key difference that we map key/value coordinates back to all the queries attending to them, instead of mapping query coordinates to key/value coordinates to which the query will attend. The inverse of this mapping does not equal itself for NA: given any NA mask, query token $i$ attending to context token $j$ does not imply that query token $j$ attends to context token $i$. Because of this, the neighborhood mapping logic in this operation, and with it the workload, is different form that in NN. We dub this final operation Inverse-Neighborhood Operator (IN).

With these three operations, which are again GEMV operations, one can implement exact NA for both forward and backward propagation, as opposed to two forward pass, and another four backward pass operations.

### 3.2.2 GEMM-based NA

We transform the aforementioned GEMV problems into GEMMs, via multi-dimensional tiling and the fusion of additional scatter or gather operations, all of which we outline in this subsection. All implementations were done on top of CUTLASS 2.X TC GEMMs for the Ampere architecture. We present an illustration of this concept for PN in Fig. 3.2.

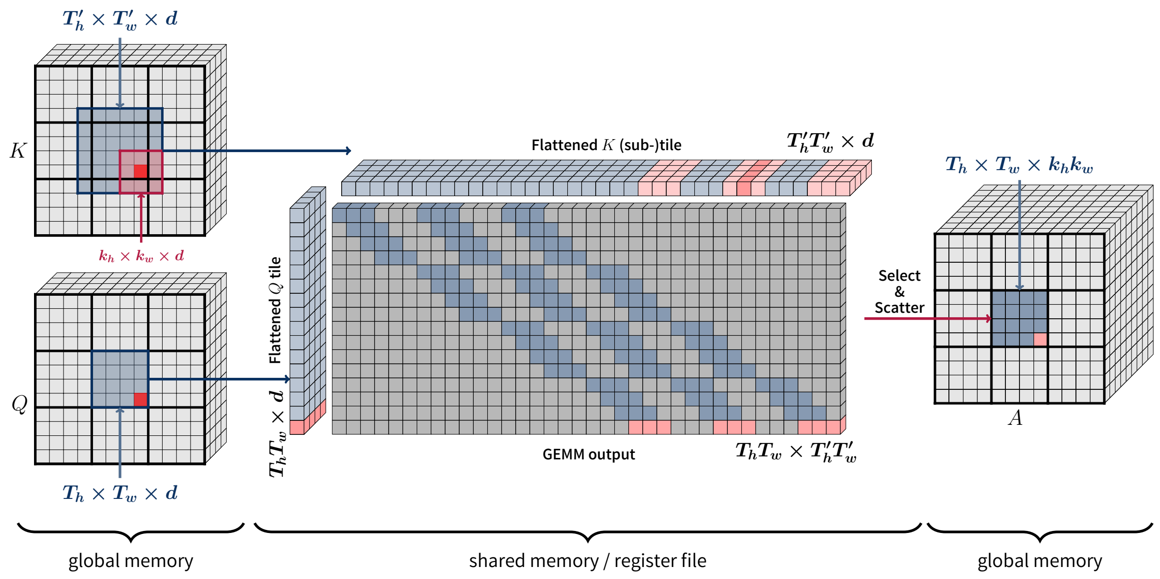


**Figure 3.2. Illustration of GEMM-based NA: 2-D Pointwise-Neighborhood Operator (PN).** Input tensors are tiled with respect to their 2-D token layout (multi-dimensional tiling). Query is tiled with a static tile shape of $T_h \times T_w$. For each query tile (Q tile), we determine the smallest region in $\mathbf{K}$ containing all corresponding neighbors based on the definition of NA and parameters such as window size, $k_h \times k_w$. That region, $T'_h \times T'_w$, is sliced out and if necessary further tiled into sub-tiles. Once the relevant tiles are loaded from global into local memory, they are viewed with matrix layouts, over which a GEMM is performed. Resulting dot products are a matrix of shape $T_hT_w \times T'_hT'_w$. Dot products valid according to the NA mask ($T_h \times T_w \times k_hk_w$) are scattered back into global memory.

The scatter / gather fusion is a modification of global memory store and load operations respectively. In a GEMM-based PN, we end up with all the dot products between a query tile (Q tile) and the relevant key tiles, which is a superset of valid dot products according to the NA mask. We therefore need additional logic to visit the individual dot products and map their offsets back into their coordinates in the token layout. If the interaction is valid, we proceed with computing the correct offset in global memory to store the dot product, and if it is invalid, it is simply ignored. In GEMM-based NN and PN, this turns into a gather operation, where for each query and key-value pair with a valid interaction, we find the correct offset in global memory to load the corresponding dot products, and otherwise zero it out to prevent accumulating the corresponding value for that query.

Neighborhood Attention (NA) is specifically designed for multi-dimensional layouts of tokens (i.e. 2-D and 3-D), as 1-D operations are typically trivial to implement. While tokens live in a multi-dimensional space, tiling in GEMM kernels is done contiguously along rows, which in this case correspond to the 1-D layout of tokens in memory. For PN, this means query tokens within the current tile may strongly disagree on their neighborhoods, which means more key tiles have to be visited to compute all the dot products for the current Q tile. This means more FLOPs and therefore more time spent to perform this operation, while the analytical FLOPs are unchanged. This increases runtime, and since throughput in this context is expressed as FLOPs per unit of time, this behavior hurts throughput. To circumvent this issue, we perform **multi-dimensional** tiling, which effectively turns the GEMM problem into a special case of General Tensor-Tensor Contractions (GETTs). This is done by re-interpreting the GEMM tile size as a multi-dimensional tile shape. For example, for a 2-D NA implementation, if the original kernel has a Q tile size of 128, it can be re-interpreted as $8 \times 16$, or $16 \times 8$, or any two natural numbers the product of which is the original 128. We implement this concept by altering the workload scheduler and load and predication units in the kernel, while keeping the rest of the underlying logic, specifically the GEMM subroutine, unchanged.

Multi-dimensional tiling preserves the original layout of tokens, thereby preserving spatial locality, a concept well known to exist in NA and SWA: Query tokens in a local region overlap greatly in the context tokens to which they attend. However, both multi-dimensional tiling and scatter / gather fusion as proposed here face some performance limitations. Multi-dimensional tiling greatly alters the scheduling logic, as well as predication and load logic. Predication for GEMMs can be done very efficiently, but in the case of multiple dimensions, it is quite difficult to avoid additional indexing and integer operations, which can increase register pressure in addition to the overhead from the operations themselves. Scatter / gather fusion on the other hand suffers from the fact that loads and stores need to be done on an element-wise basis, instead of the typical vectorized loads and stores. This especially hurts NN and IN in half precision, where pipelining relies on

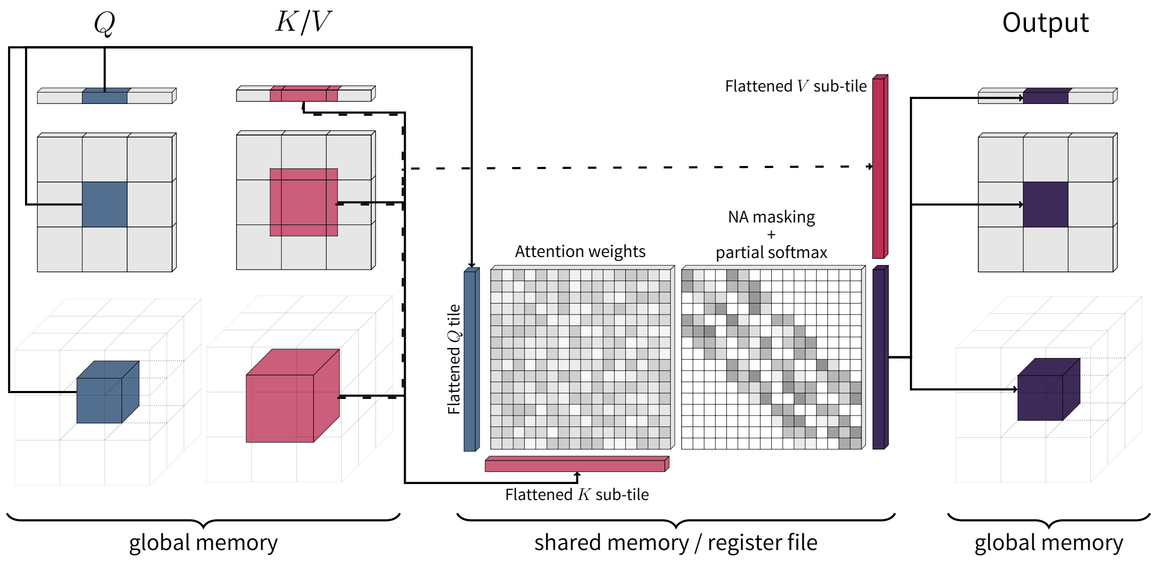


**Figure 3.3. An illustration of Fused Neighborhood Attention (FNA) forward pass.** All tensors are tiled according to their token layout (1-D, 2-D, 3-D). For each query tile (Q tile), we identify the smallest region in the key and value tensors containing all corresponding neighbors based on the NA mask. That region is then tiled to produce the FMHA inner loop over key-value tiles (KV tiles). During each iteration, resulting attention weights from the first GEMM are masked according to NA parameters, before undergoing online softmax scaling, and going through the second GEMM with the corresponding value tile.

the vectorized load instructions for input operands, as they are non-blocking (`LDGSTS`). Forced to fall back to the blocking operation (`LDS`), this breaks pipelining in the kernel, leading to a considerable slowdown.

Given the relative level of complexity in this implementation, its limitations, and the general limitations of unfused attention, we did not build a 3-D version of this approach, and instead incorporated the ideas and lessons learned here in our fused kernel, FNA.

### 3.2.3 Fused Neighborhood Attention (FNA)

We extend our methodology for GEMM-based implementations of NA to Fused Multi-Headed Attention (FMHA) methods. This is not only motivated by the potential to reduce runtime and memory footprint, and potentially making NA actually bound by compute, but also to circumvent the bottleneck in the prior unfused implementations: scatter / gathering attention weights to / from global memory. Since attention weights are never materialized in global memory in fused kernels, this particular limitation will simply cease to exist. We start off with the xFormers FMHA [72], which is based on the CUTLASS 2.X API, and is set up to support architectures even older than Ampere (Maxwell , SM50; Volta, SM70; and Turing, SM75.) By carefully applying our multi-dimensional tiling, NA masking, and changes to software predication for multi-dimensional tensor layouts, we can successfully implement NA for 1-D, 2-D, and 3-D problems. This approach again dynamically slices out the smallest region in the key and value tensors that contains all neighbors for the Q tile being processed, in an attempt to minimize the number of key-value tiles (KV tiles) visited, and therefore the number of GEMMs performed, or the number of FLOPs done. We dub this approach **dynamic KV tiling**, as different workers assigned to different Q tiles will tile key and value tensors with different offsets. Due to this dynamism, this concept may not be easily implementable with hardware predication engines such as the NVIDIA Tensor Memory Accelerator (TMA), but we will discuss this further in the next chapter. We also note that the overhead of software predication as a result of multi-dimensional tiling, and that of fine-grained masking, can still limit this new approach, just as they did the prior GEMM-based approach. However, fine-grained masking itself is a far smaller bottleneck here, as we avoid the scatter / gathering of attention weights to global memory entirely.

Fig. 3.3 presents an overview of the forward pass in FNA.

### 3.2.4 Dilation and causal masking

Our methodology allows for dilation support trivially, through simple partitioning and slicing ahead of time. A DiNA problem can be mapped to a set of non-dilated NA problems over non-overlapping and non-contiguous regions of the input. This transformation can be done during the initialization of the kernel scheduler, with little to no overhead. The rest of the kernel logic will not need to consider dilation at all. Causal masking along any dimension simply changes the definition of the NA mask to the causal variant. This only changes the softmax stage where fine-grained masking is performed.

**Table 3.1. Overview of GEMM-based NA and FNA performance in FP16 forward pass.** We benchmark naive NA kernels against our new GEMM-based and fused kernels in half precision, over a large set of problem sizes varying in batch size, token layout shape, 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 base implementation) and Flash Attention 2 (FAv2).

| **Implementation** | **% of problems matched or outperformed** | | | | |
|---|---|---|---|---|---|
| | **NA** | | | **SA** | |
| | **Naive** | **GEMM-based NA** | **FNA** | **xFormers** | **FAv2** |
| *1-D Neighborhood Attention* | | | | | |
| **Naive** | - | 1.7 % | 0.0 % | 21.8 % | 8.8 % |
| **GEMM-based NA** | 98.7 % | - | 0.0 % | 72.0 % | 54.2 % |
| **FNA** | 100.0 % | 100.0 % | - | 100.0 % | 98.2 % |
| *2-D Neighborhood Attention* | | | | | |
| **Naive** | - | 16.4 % | 0.0 % | 32.9 % | 15.8 % |
| **GEMM-based NA** | 84.0 % | - | 0.0 % | 59.3 % | 29.8 % |
| **FNA** | 100.0 % | 100.0 % | - | 98.6 % | 92.4 % |
| *3-D Neighborhood Attention* | | | | | |
| **Naive** | - | - | 0.0 % | 43.5 % | 20.2 % |
| **FNA** | 100.0 % | - | - | 97.3 % | 87.0 % |

### 3.2.5 Auto-tuner

GEMM kernels are, among other settings, parameterized by their tiling sizes. Multi-dimensional FNA kernels are likewise parameterized by their tile **shapes**. Different tile shapes can have very different performance levels, especially in the case of FNA where tile shapes determine how many FLOPs are done, and how many of those are wasted due to the fine-grained NA mask.

We therefore implement a very simple auto-tuning method as a proof of concept. This auto-tuner 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 the full set of tiling configurations, and the best configuration is selected and cached. This can sometimes lead to significant speedups, but can also significantly slow down the first run of any unique operation, as the number of valid configurations for FNA is in the hundreds.

**Table 3.2. Overview of GEMM-based NA and FNA performance in FP32 forward pass.** We benchmark naive NA kernels against our new GEMM-based and fused kernels in full precision, over a large set of problem sizes varying in batch size, token layout shape, 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 base implementation).

| **Implementation** | **% of problems matched or outperformed** | | | |
|---|---|---|---|---|
| | | **NA** | | **SA** |
| | **Naive** | **GEMM-based NA** | **FNA** | **xFormers** |
| *1-D Neighborhood Attention* | | | | |
| **Naive** | - | 0.0 % | 0.0 % | 34.6 % |
| **GEMM-based NA** | 99.9 % | - | 37.7 % | 98.4 % |
| **FNA** | 100.0 % | 64.8 % | - | 99.9 % |
| *2-D Neighborhood Attention* | | | | |
| **Naive** | - | 11.7 % | 5.4 % | 52.0 % |
| **GEMM-based NA** | 89.5 % | - | 28.1 % | 92.4 % |
| **FNA** | 96.0 % | 74.0 % | - | 99.3 % |
| *3-D Neighborhood Attention* | | | | |
| **Naive** | - | - | 0.0 % | 61.1 % |
| **FNA** | 100.0 % | - | - | 98.6 % |

## 3.3 Experiments

We evaluate the performance of our proposed methods by measuring their runtime against existing kernels in NATTEN. Most use cases in NATTEN target naive CUDA C++ kernels, with only one exception: 2-D NA cases with 32-dimensional attention heads, square windows up to and including 13 × 13, and no causal masking, can target a special PN kernel that includes additional performance optimizations, as mentioned in Sec. 2.2.9. All prior implementations, including the special PN variants, are considered our baseline, and will henceforth be referred to as naive kernels.

We created a set of problem sizes that vary in batch size, token layout shape, number of attention heads, and dimensions per attention head. There are 6150 cases of 1-D problems, 5676 cases of 2-D problems, and 2448 cases of 3-D problems. Approximately 40% of the 2-D problems are ones that are implementable with the specialized variants of the

**Table 3.3. Breakdown of GEMM-based NA and FNA forward pass performance.** Both GEMM-based and fused NA improve naive kernels. However, there exist cases in which naive kernels may be preferable to GEMM-based, but those are typically very small problem sizes. GEMM-based and fused are also more or less comparable in many FP32 cases, but unfused kernels are generally not a good choice for most FP16 cases. FP16 is also the more used precision among the two, across many applications.

| Dim | GEMM-based NA over naive | | | FNA over naive | | | FNA over GEMM-based NA | | |
|---|---|---|---|---|---|---|---|---|---|
| | 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 % | - | - | - |

**Table 3.4. Breakdown of GEMM-based NA and FNA forward and backward pass performance.** Improvements of theGEMM-based and FNA kernels over naive, while not as significant as in the forward pass, are still considerable. In some very small sized cases, naive can still outperform the new kernels, but those are rare exceptions. This benchmark is limited to half precision only, because most training is done in half precision.

| Dim | GEMM-based NA over naive | | | FNA over naive | | | FNA over GEMM-based NA | | |
|---|---|---|---|---|---|---|---|---|---|
| | 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 % | - | - | - |

naive kernels, and could potentially achieve better performance with respect to the new approaches, specifically the unfused GEMM-based one. We ran these problem sizes through every implementation on an NVIDIA A100 GPU and measure their runtime using CUDA events. We iterate through multiple NA window sizes and dilation values for every problem size. A summary of these benchmarks is presented in Tab. 3.1 (FP16) and Tab. 3.2 (FP32).

We find that our GEMM-based kernels can improve or match naive in approximately 99% of 1-D problems and 84% of 2-D problems using half precision, and approximately 100% of 1-D problems and 89% of 2-D problems using full precision. As noted earlier, the half-precision implementation of the GEMM-based NN operation suffers from blocking loads which break the pipelining structure (see Sec. 3.2.2), therefore, inferior half-precision performance is to be expected. On the other hand, our fused kernels improve or match the

**Table 3.5. NAT / DiNAT classification inference speedups using GEMM-based and fused NA (FP16).** FNA can provide up to a two-fold speedup over naive, while the GEMM-based implementation suffers significantly from the broken pipelining in NN, leading to minor slowdowns compared to the performance-optimized naive kernels, which were designed specifically for the ImageNet use cases seen here.

| **Model** | **# of Params** (M) | **FLOPs** (G) | **Throughput** **Naive** | **GEMM-based NA** (imgs/sec) | **FNA** | **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 |

naive runtime in approximately 100% of both 1-D and 3-D problems in both half precision and full precision, and 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. In both Tab. 3.1 and Tab. 3.2 we also inspect the percentage of problem sizes in which using FNA we can outperformed the original xFormers FMHA kernel. This is a useful comparison as our implementation is done on top of the xFormers kernel. Through this comparison, we can clearly see a relationship between the dimensionality of token layout, and percentage of cases in which FNA successfully accelerates FMHA. 3-D is the worst-performing case and 1-D is the best performing case. This is in line with our observation that the overhead of software predication and fine-grained masking is unavoidable in multiple dimensions, and the effects are worsened with more dimensions.

**Table 3.6. NAT / DiNAT classification inference speedups using GEMM-based and fused NA (FP32).** While fused attention kernels are not expected to have a large edge over unfused attention kernels in FP32, FNA still noticeably speeds up naive kernels in this setting. The same does not hold for GEMM-based NA, as improvements over the performance-optimized naive kernels, which were designed for these use cases specifically, are small and within the margin of error.

| **Model** | **# of Params** (M) | **FLOPs** (G) | **Throughput** **Naive** (imgs/sec) | **GEMM-based NA** | **FNA** | **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 |

We further present a breakdown of the above benchmarks in Tab. 3.3, where we report the average, minimum, and maximum improvement over different use cases, comparing FNA, GEMM-based NA, and naive. GEMM-based kernels exhibit strong performance improvement compared to naive kernels, specifically in full precision. Additionally, GEMM-based kernels can also match the performance of FNA kernels in full precision. Both new approaches also outperform naive kernels even in many cases that can use the more optimized naive kernels. While the further optimized variants of the naive kernels can sometimes exhibit competitive or slightly improved performance over the new approaches, we note that they cannot generalize to all problem sizes as our GEMM-based kernels can, nor are they easily extensible. We also present the same benchmark, but with both forward and backward pass numbers included, in Tab. 3.4.

In addition to operation-level benchmarks, we also evaluate the effect of our proposed methodology on models that use NA as a primitive, namely NAT and DiNAT from Chap. 2,

**Table 3.7. NAT / DiNAT classification training time improvement estimates using GEMM-based and fused NA.** Each variant was trained for one epoch, and extrapolated to 300 training epochs, with an initial warmup epoch that is not measured into the estimate. All measurements are according to half precision, following the typical training scenario for these models.

| **Model** | **# of Params** (M) | **FLOPs** (G) | **Training time estimate (hours)** | | |
|---|---|---|---|---|---|
| | | | **Naive** | **GEMM-based NA** (hours) | **FNA** |
| **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 %** ) |

and StyleNAT [65], a style-based generative adversarial (GAN) model based on NA. We benchmark throughput for all NAT and DiNAT variants according to the same setup as in Sec. 2.3, and report both FP16 and FP32 measurements in Tab. 3.5 and Tab. 3.6 respectively. We benchmarked StyleNAT [65] throughput according to the specifications from the original work, and present our measurements in Tab. 3.8, but only in FP32, which is the recommend precision for this application.

We find that the problem sizes used by the ImageNet classification variants of NAT and DiNAT, which are typically smaller in token count 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. FNA kernels on the other hand never fail to improve upon the baseline, but improvement is strongest in half precision, and cases that use dilation frequently (DiNAT variants). In StyleNAT [65], we similarly observe that both our GEMM-based and FNA kernels can improve inference time compared to the existing naive kernels, with FNA having a much more noticeable edge.

**Table 3.8. Image generation end-to-end throughput improvement when using GEMM-based and fused NA.** We benchmark the throughput of StyleNAT [65], a style-based generative adversarial model based on NA under different approaches. We experimented with different batch sizes in order to achieve the best performance, and settled for 64 for the $256 \times 256$ variant, and 8 for the $1024 \times 1024$ variant. All measurements are done in FP32, which is the recommended precision level according to the original work.

| **Dataset** | **# of Params** | **Throughput** | | | **FID** |
|---|---|---|---|---|---|
| | | **Naive** | **GEMM-based NA** (imgs/sec) | **FNA** | |
| **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 |

Finally, we also attempted to estimate improvements in training time compared to our baseline. As suggested by our earlier findings regarding the limits of our GEMM-based implementation in the backward pass (specifically NN and IN), we do not see any improvement in training time compared to the naive baseline. Again as in inference, FNA kernels do improve half precision training time, by up to about 40%. We present our estimates in Tab. 3.7, which are based on measurements from training NAT and DiNAT variants according to the same setup in Sec. 2.3. 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.

## 3.4 Conclusion

In this chapter, we introduced a methodology for packing the computation of Neighborhood Attention, a vector-matrix multiplication problem (GEMV), into matrix-matrix multiplications (GEMMs). We provide two such implementations, one in unfused form, dubbed GEMM-based NA, and a fused (FMHA) implementation, FNA. These implementations can significantly speed up existing implementations in NATTEN introduced in the last chapter, and come with far fewer restrictions in terms of problem sizes that are supported. We find that, as hypothesized, FNA is the clear winning choice, as it is not bottlenecked by scatter / gather operations (due to fine-grained mask), and memory bandwidth (as typical unfused forms of attention are). These new implementations can accelerate nearly all existing applications based on NA, with little effort on the part of end users. They can additionally enable new applications for more complex use cases of NA, specifically those with large-scale 2-D and 3-D token layouts, spatio-temporal token layouts, and non-uniform parameters. Those applications can now finally achieve decent speedups compared to their modern Self Attention baselines, and evade the limits of global memory capacity and global memory bandwidth.

However, FNA itself can suffer greatly from four remaining bottlenecks: 1. **Software predication** overhead from multi-dimensional tiling grows with dimensionality. 1-D problems are safe from this issue, but 2-D and 3-D problems may suffer from additional performance gaps with the original xFormers FMHA kernel. 2. **Fine-grained masking** hurts throughput, and comes with additional overhead of its own. The former is due to the sliding window nature of NA, while the latter can potentially be further performance-optimized, but never completely eliminated. 3. **Performance gap with the state of the art** not just as a result of the implementation overhead, but also stemming from our base implementation, xFormers FMHA, being roughly half the speed of the best available implementation (for Ampere), Flash Attention 2 [67]. 4. **Auto-tuning**, while helpful, is mostly feasible when done ahead of time. Since there are hundreds of valid configurations for the current FNA implementation, profiling all of them can significantly slow down applications, specifically distributed training jobs.

In the next chapter, we attempt to address all 4 of the above, by extending the current FNA methodology to state-of-the-art FMHA implementations for recent architectures, introducing a new design for FNA kernels, and extending the definition of NA to support sliding windows with delayed steps, which can potentially improve throughput and minimize the issues with fine-grained masking.

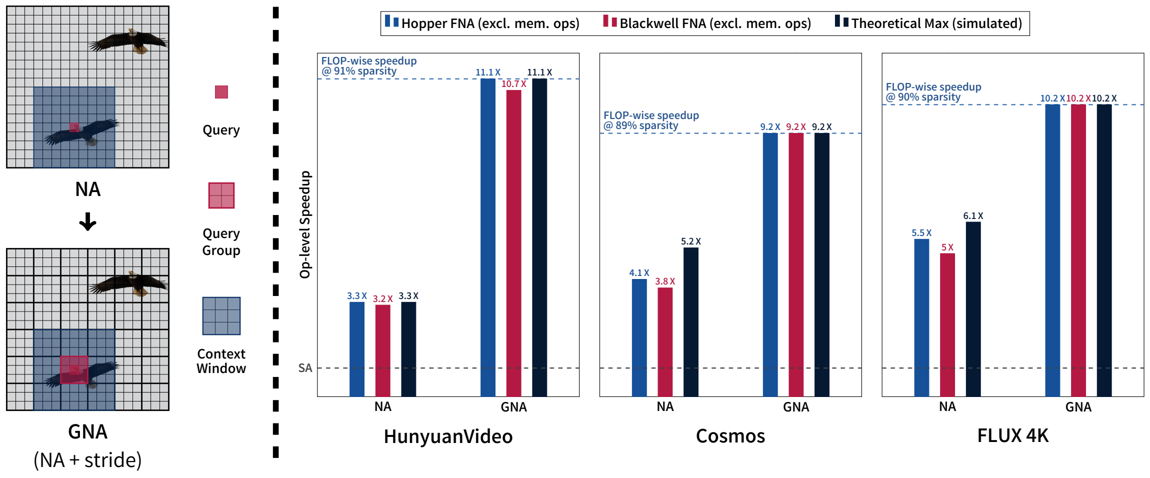


**Figure 4.1. Overview of Generalized Neighborhood Attention (GNA).** GNA expands the definition of NA by adding a new "stride" parameter, which can coarsen the resulting attention map, and even create fully block-sparse forms of NA. This, together with a new FNA design, can result in speedups proportional to savings in FLOPs, which is the maximum speedup theoretically possible.

# CHAPTER 4
# GENERALIZED NEIGHBORHOOD ATTENTION

Neighborhood Attention (NA), and many other sparse attention methods often do not provide a consistent and reliable speed improvement over standard attention. In the case of NA, and specifically the Fused Neighborhood Attention (FNA) implementation introduced in Chap. 3, the most significant causes are inferior base implementations, implementation overhead, and wasted FLOPs. In this chapter, we aim to study and resolve all three problems in the NA family.

Wasted FLOPs cannot generally be avoided in NA, due to the sliding window nature of the approach. This is consequence of the packed GEMM-based computation, virtually all performant implementations of which utilize tiling of the computation. To mitigate this issue, we introduce an optional delay step parameter into NA called **stride**, which coarsens the attention mask, eventually leading to those that waste no FLOPs. Implementation overheads in FNA, as discussed in the previous chapter, primarily stem from the cost of complex software predication logic, and the fine-grained masking logic. The latter can be completely avoided in certain strided NA cases, specifically those that do not waste FLOPs. The former can be alleviated with hardware predication, but it would be invasive to the design of the rest of the kernel with potential performance implications of its own, in addition to being limited to architectures supporting hardware predication (NVIDIA Hopper and Blackwell today). We therefore take a different approach, and instead of insisting on kernel fusion attempt to separate out logic from the FNA kernel (**kernel decomposition**), so that a small and fixed-cost memory operation handles the complexity of multi-dimensional tiling. The key advantage of this decomposition is, however, the simplification of the FNA design itself, which allows us to extend existing state-of-the-art FMHA kernels to FNA kernels, thereby resolving the last remaining issue.

Through this framework, which we dub Generalized Neighborhood Attention (GNA), we achieve unprecedented performance levels for existing forms of NA on the Hopper and Blackwell architectures. In addition, in some of the new "strided" forms of NA, we can achieve speedups proportional to reduction in FLOPs, which is the **theoretically maximum achievable**. We demonstrate the effectiveness of this new framework on generative AI models, specifically Image, Video, and World Foundation Models.

Most of the materials in this chapter are available separately as an arXiv preprint, titled "Generalized Neighborhood Attention: Multi-dimensional Sparse Attention at the Speed of Light", and pending peer review. The Cosmos Predict 2 experiments in Sec. 4.3.2 were done in collaboration with NVIDIA Research.

## 4.1 Background and related works

In this section, we first review the limitations of FNA as introduced in Chap. 3, and then discuss the connection between FNA and a very common approach in implementing efficient sparse Attention masks, called block-sparsity. We also discuss an important problem that affects only multi-dimensional layouts of tokens and their coordinate-based masks, which includes NA in addition to SWA, Blocked Attention, and many other similar approaches. This problem, which we dub the **curse of multi-dimensionality**, although addressed directly by FNA's multi-dimensional tiling, is also its most significant bottleneck. Our solution to escaping these opposing goals is a kernel decomposition, which resolves the two most significant limitations. We finally discuss prior works inspiring our solution to FLOPs wasted as a result of the sliding window pattern. This solution involves trading off the sliding window nature and with it properties such as translational equivariance (see Sec. 2.2.4) for fewer wasted FLOPs. This creates a new spectrum of possible attention patterns between sliding window / Neighborhood Attention, and Blocked Attention.

### 4.1.1 Limitations of the original FNA

FNA suffers from a relatively large performance gap with respect to the state of the art for its main architecture target, the NVIDIA Ampere architecture. The solution to this is simply implementing FNA on top of a state-of-the-art FMHA kernel, but also ensuring the design is not invasive to the base kernel to the extent that it hurts its performance. Setting aside the gap between its base kernel, xFormers FMHA, and the state of the art, Flash Attention 2 [67], we can see another gap even between the base kernel and FNA when moving to multi-dimensional layouts of tokens, which is our main focus. Most of this gap is due to the overhead of **software predication**. In 1-D cases, we can expect at most only 1 partial tile, making predication relatively efficient. In multi-dimensional cases, no such assumption can be made, and dynamic KV tiling exacerbates this, leading to an implementation that requires predicating token transfers on a token-by-token basis. Eliminating multi-dimensional tiling is not a good solution, as it breaks the spatial structure, potentially introducing more wasted FLOPs, and further complicates scheduling, which relies on spatial locality in the multi-dimensional case. We dub this problem the curse of **multi-dimensionality**, and illustrate an example in Fig. 4.2. Maintaining multi-dimensional tiling

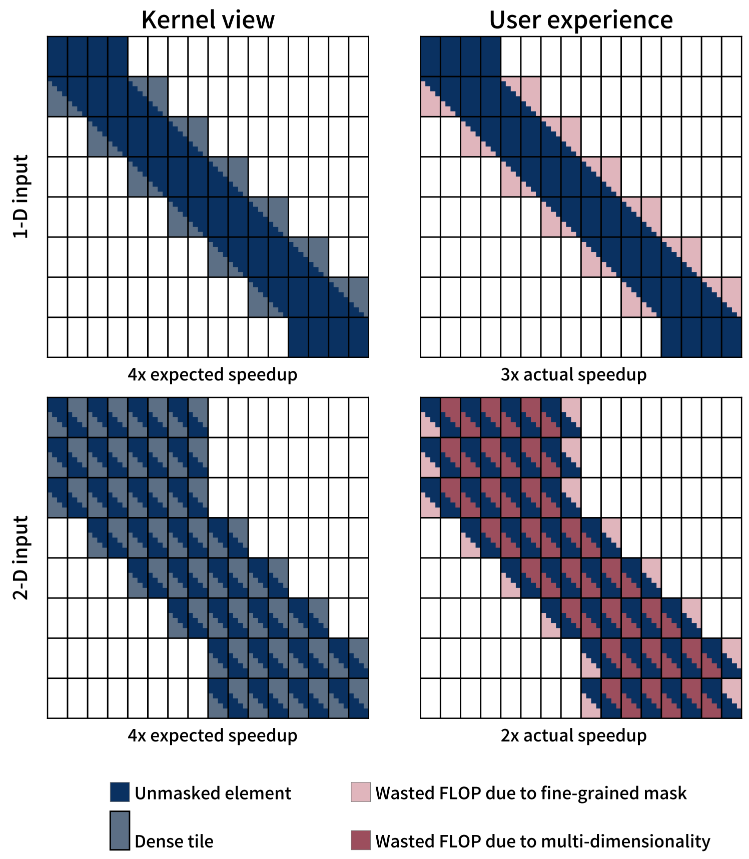


**Figure 4.2. Curse of Multi-Dimensionality.** Attention masks such as NA, when implemented with packed GEMMs, waste FLOPs due to fine-grained masking. In multi-dimensional cases, more FLOPs are wasted if tiling is still done along a single dimension. Multi-dimensional tiling in FNA avoids this, but wasted FLOPs due to fine-grained masking cannot be avoided.

without the overhead of software prediction can be done by using hardware predication engines such as the Tensor Memory Accelerator (TMA) [1], but this will be limited only to architectures supporting it, and is quite invasive to the design of performant FMHA kernels. For example, FMHA kernels for the Blackwell architecture split the softmax workload between two warps, either along the dimension corresponding to query tokens or KV tokens. When operands are multi-dimensional, this split will require further adaptation. Implementing Dilated Neighborhood Attention when using the TMA will also create additional complications, as it requires an additional non-contiguous tiling and predication itself.

[1]https://docs.nvidia.com/cuda/hopper-tuning-guide/index.html#tensor-memory-accelerator

We therefore introduce a new FNA design, based on *kernel decomposition* instead of fusion, separating out multi-dimensional tiling and implementing it as a memory operation that rearranges tokens. This greatly simplifies the design of the FNA compute kernel, allowing non-invasive implementation on top of state-of-the-art FMHA kernels for the Hopper and Blackwell architectures, with very limited overhead. We note, however, that this only addresses the first two limitations. Fine-grained masking, and the auto-tuning requirement can still greatly impede performance gains proportional to saved compute.

**Fine-grained masking** is mainly an artifact of packing patterns that are inherently GEMV problems into GEMM problems, as a consequence of which tiling and tiled matrix multiplications will produce dot products that are not needed. To ensure numerical correctness, FMHA kernels implement fine-grained masking, which is applied prior to softmax. This holds true even for cases that are not necessarily considered sparse, such as causally masked attention. Fine-grained masking has two implications: the overhead of masking, and wasted FLOPs. The former can vary in its severity, with cases like 1-D causal mask being a single index comparison, and cases like 3-D NA potentially including up to 6 integer comparisons. The latter hurts the overall throughput and scaling capabilities, as it introduces a discrepancy between FLOPs actually performed (grain size is tiled matrix multiplication size), and analytical FLOPs (grain size is FMA operations). Fig. 4.2 illustrates the effect of wasted FLOPs due to the fine-grained masking.

**Auto-tuning** is another limitation of FNA, as the default tiling configurations can sometimes waste significantly more FLOPs and result in inferior performance. With so many such configurations possible, and the vast parameter space of NA, auto-tuning even ahead of time can be a great burden. To address this, especially considering that this work further expands the NA parameter space, we propose an analytical tool that simulates FNA and computes the number of saved tiles of work, which typically correlates with runtime. This approach can largely replace the auto-tuner and select the best configuration for any given use case.

### 4.1.2 FNA and Block-Sparsity

In the context of this work, we define block-sparsity for attention as a tile scheduler capable of predicting tiles of work which will be masked out entirely, and skipping them, saving computation. Block-sparse scheduling should only affect the runtime of the implementation, and not functionality. Most FMHA kernels come with block-sparse scheduling for causally masked attention. Any form of attention mask (including 1-D causal mask) will require the implementation of a fine-grained mask, unless it is guaranteed that the mask is fully block-sparse with respect to the block (tile) size of the block-sparse implementation. A fine-grained mask is fully block-sparse with respect to an implementation if every tile of work is either fully masked, or fully unmasked. The advantage of this is that a block-sparse scheduler can skip the masked tiles, saving compute, and the visited tiles do not require fine-grained masking, reducing the number of wasted FLOPs to zero, and potentially eliminating the overhead of fine-grained masking.

In the previous chapter, both the GEMM-based unfused NA operators, and FNA can be considered instances of block-sparse scheduling. They skip blocks that are fully masked in order to save computation and accelerate with respect to the Self Attention baseline. One key difference compared to many common block-sparse schedulers is *dynamic KV tiling*: GEMM-based NA and FNA first carve out the region within the key-value pair containing all unmasked elements for the current Q tile (all key-values attended to by at least one query within the current tile), and then perform tiling over the sliced region. Dynamic KV tiling can potentially result in fewer KV tiles visited for each Q tile, therefore saving more compute. A downside to dynamic tiling is incompatibility with certain FMHA kernel designs, specifically those statically tiling all operands. This is especially the case in the most performant implementations for the NVIDIA Hopper and Blackwell architectures, which use the TMA for global memory transfers.

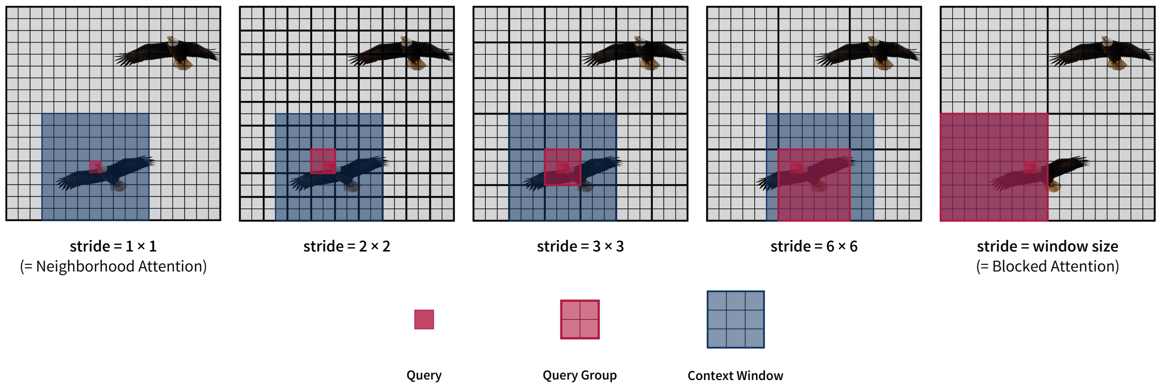


**Figure 4.3. Visualization of the effect of stride in GNA.** GNA allows customizable delay steps in the sliding window pattern through the new **stride** parameter. Stride groups queries together and forces them all to attend to the same neighborhood. Stride can take any positive integer value smaller than or equal to window size. Stride of 1 is equivalent to standard NA. When stride is equal to window size, it is equivalent to Blocked Attention.

### 4.1.3 Trading off inductive biases for efficiency

Strided Sliding Window Attention was proposed by Vaswani et al. [10] in a work following the original Sliding Window Attention (SWA) work [9]. Citing implementation woes, specifically those related to the memory overhead from materializing the extracted sliding windows in global memory, this new approach introduced a delay step parameter which relaxes translational equivariance, but aims to improve efficiency, thus creating a trade-off space between the two. The effect of this is multiple queries being packed together to share the same local context window, reducing the number of extracted windows. Although modern implementations such as FNA completely avoid materializing sliding windows and therefore do not face this memory issue, this concept can serve a different purpose today: reducing the number of wasted FLOPs by approaching more block-sparse masks. We therefore transfer the same concept the NA family by introducing a new "stride" parameter. This new parameter, visualized in Fig. 4.3 and described in the next section, can coarsen the fine-grained attention mask to the point of full block-sparsity, meaning the elimination of wasted FLOPs, the last remaining limitation.

## 4.2 Methodology

In this section, we describe our methodology for addressing the remaining major limitations of NA and FNA. We first discuss our new FNA design, which is a decomposition of the existing approach into two new operations: 1. A memory operation responsible for the rearrangement of tokens, simulating the behavior of multi-dimensional tiling, and 2. Simplified FNA compute kernel with static block-sparse scheduling. This design allowed us to implement FNA on top of state-of-the-art FMHA kernels from CUTLASS [71] for the NVIDIA Hopper and Blackwell (datacenter-class) architectures, with very little overhead. We then describe the new **stride** parameter and its implication on efficiency and inductive biases. We specifically discuss how stride is not always guaranteed to improve efficiency, and that in certain cases, it may exhibit worse efficiency compared to standard NA, and how this behavior can differ across different tiling configurations, and as a result, different hardware. In addition, the already vast parameter space of NA is now even larger. We therefore create a simulation tool, dubbed NATTEN Simulator (NATTENSim). We describe how it functions, and how it can be used to guide the process of selecting parameters based on information about the use case, design choices, and even hardware-specific details.

### 4.2.1 Decomposed FNA implementation

FNA's biggest performance limiter is caused by what is supposed to make it efficient. Multi-dimensional tiling breaks the curse of multi-dimensionality (see Fig. 4.2), but it can introduce a new overhead: software predication. Multi-dimensional tiling is also what is responsible for most of the complexity in the kernel, compared to a standard FMHA kernel. Our approach for circumventing this issue involves decomposing the FNA kernel into memory operations and a single compute operation. Memory operations are responsible for rearranging tokens (and rearranging them back) in such a way that the compute kernel focuses on delivering the best throughput, unencumbered by additional predication and in-

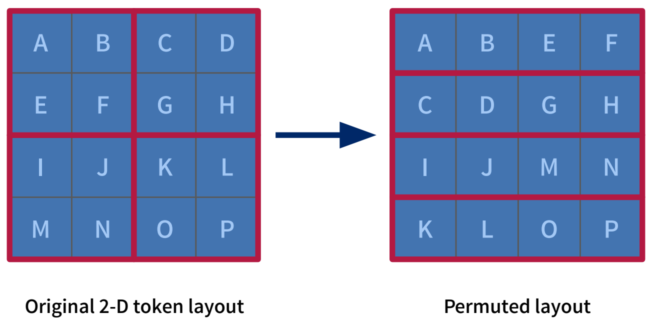


**Figure 4.4. Visualization of 2-D Token Permutation.** This visualization assumes the input is a $4 \times 4$ layout of tokens, and the target kernel uses tile size 4, which we re-interpret as a $2 \times 2$ multi-dimensional tile shape per FNA. Token permutation applies the multi-dimensional tiling, and then stores individual tiles contiguously in memory. The result is that each tile loaded by the kernel will have the spatial structure preserved, simulating the same behavior as the original FNA, but without the added cost of software predication.

teger operations. This also further simplifies the design of the kernel, taking away most of the complexity from both the forward and backward pass kernels, which reduces room for newly introduced overhead.

The memory operation, which we dub **Token Permutation**, is a layout transformation that creates a new 1-D layout of multi-dimensional tiles, visualized in Fig. 4.4. The kernel, which will now have to statically tile KV, instead of dynamically in the original approach, will load the aforementioned multi-dimensional tiles. The scheduler and fine-grained mask can easily map their 1-D coordinates back to the multi-dimensional coordinates. The switch to static tiling can potentially introduce more wasted FLOPs, but in turn will have a much better throughput than dynamic tiling in the original kernel ever could. In addition, since the more recent architectures tend to use the TMA, dynamic tiling is not even an option unless bulk memory operations are avoided in favor of the older `LDGSTS`, with potentially inferior performance. Token permutation is responsible for also padding the operands as necessary, and handling the dilation partitioning. For now, token permutation is implemented through PyTorch [48] directly. In future versions, we hope to develop specialized kernels for it, as the current solution typically utilizes only a fraction of the memory bandwidth.

Aside from specialized kernels, the overhead from token permutation can be minimized in other ways. When dealing with an isotropic architecture such as the original Transformer [3], ViT [8], and Diffusion Transformer (DiT) [75], and as long as dilation values do not vary across layers, and tile shapes are likewise kept consistent, token permutation can be done only once prior to the first NA operator, and once to undo the permutation after the last NA operator. In other use cases, if tokens are distributed among multiple processors, and gathered using All2All, token permutation can potentially be handled through the network primitive. We leave these as potential future directions, as our current implementation exhibits decent end-to-end performance even with the naive implementation of token permutation, since the cost, especially in larger problems, is often negligible compared to the compute kernel.

As for the kernel, we start off with the CUTLASS [71] FMHA kernels for the Hopper and Blackwell architectures. The Hopper kernel achieves competitive forward pass performance with Flash Attention 3 [68], the state of the art, and the Blackwell kernel is competitive with the cuDNN kernel, which is again the state of the art. It's important to note that this design can likewise extend to most FMHA methods, including the various versions of Flash Attention [26, 67, 68], but this is left as future work and only as necessary. Each kernel is modified to implement static FNA-style block-sparse scheduling, the NA fine-grained mask, and additional logic for supporting dilation and stride. To further minimize the only remaining overhead in the kernel, which is from fine-grained masking, we skip fine-grained masking for problems that are predicted to be fully block-sparse. Standard NA can sometimes qualify for this: when window size matches the input size (Self Attention), or it is a dilated case where dilation and window size cover the entire input. Strided NA, which we will introduce in the next subsection, can under certain settings be fully block-sparse, for which we develop heuristics. This ensures that when the new FNA kernel is working on fully block-sparse problems, we can avoid almost all overhead within the compute kernel, leaving token permutation as the only exposed overhead.

### 4.2.2 Generalized Neighborhood Attention (GNA)

We start with relaxing the definition of standard NA (Sec. 2.2.1) to allow for even values for window size. Our motivation for this is simple: window sizes that are multiples of the KV tile shape are the only ones that can potentially reach full block-sparsity. Since tile sizes are typically powers of two, or multiples of certain powers of two, odd-valued window sizes will never be fully block-sparse, and this can greatly limit NA. In Sec. 2.2 NA was only defined for odd-valued window sizes so that the query could be perfectly centered in its neighborhood (except corner cases). Since each query must attend to itself (key-value token with the same coordinate), an odd-valued window size can ensure the same number of neighbors on both sides. For example, with a window size of 7, query attends to itself (1), 3 tokens on the right, and 3 tokens on the left side. We relax this definition by splitting window size into two parameters: window size left, and window size right, and as follows:

$$\text{window size left} = \left\lfloor \frac{\text{window size}}{2} \right\rfloor, \tag{4.1}$$

$$\text{window size right} = \left\lfloor \frac{\text{window size}}{2} \right\rfloor + (\text{window size} \,\%\, 2 - 1). \tag{4.2}$$

These two window sizes can be set arbitrarily, as long as they are non-negative integers. However, we choose not to expose them directly at the moment as to not further expand the already large NA parameter space.

We further add a fourth parameter to NA, which we call "**stride**". Stride can take any positive integer value less than or equal to window size. Stride groups queries together, forcing them to attend to the same neighborhood. Queries in each group elect a leader query, and attend to its neighborhood, as defined by standard NA. Using the same 1-D notation in Sec. 2.2.1, query groups are denoted with $Q_{S_j}$, where $S_j$ is a set of query indices (coordinates) corresponding to the $j$-th query group, defined as follows:

$$S_j = \left\{ i \mid 1 \leq i \leq n,\ \left\lfloor \frac{i}{\text{stride}} \right\rfloor = j \right\} \tag{4.3}$$

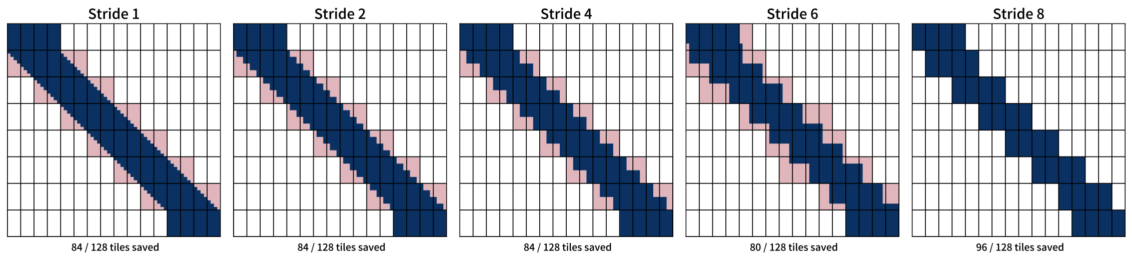


**Figure 4.5. Tiles of work saved under different stride values in a 1-D use case.** The use case is a 64-token input, with window size 16, and the kernel is assumed to use a Q tile size of 8, and KV tile size of 4. Stride 8 is the smallest stride that saves more work tiles. It is also the smallest stride that is fully block-sparse.

where $n$ denotes the total number of tokens (along the dimension). The leader query, which we will denote with $e_j$ can also be arbitrary, but again to further avoid unnecessary expansion of parameters, we statically set the leader to the center-most query in the group. If stride is an even number, we pick the one on the right side, which offsets the bias of the left-side window being potentially larger (see Eq. (4.2)):

$$e_j = \min\left(n,\ j * \text{stride} + \left\lfloor \frac{\text{stride}}{2} \right\rfloor + 1\right) \tag{4.4}$$

Finally, we define **Strided Neighborhood Attention** with window size $k$, for query group $j$, as follows:

$$\mathbf{O}^k_{S_j} = softmax\left( Q_{S_j} \begin{bmatrix} K_{\rho_1(e_j)} \\ K_{\rho_2(e_j)} \\ \vdots \\ K_{\rho_k(e_j)} \end{bmatrix}^T * \text{scale} \right) \begin{bmatrix} V_{\rho_1(e_j)} \\ V_{\rho_2(e_j)} \\ \vdots \\ V_{\rho_k(e_j)} \end{bmatrix}, \tag{4.5}$$

where scale is the standard SDPA scale, the default value for which is $\sqrt{d}$, $d$ being the head dimension. This is repeated for every stride group ($S_j$). Extension of this to multiple dimensions, like standard and dilated NA, is the repetition of the same logic over multiple dimensions.

We further note that unlike the definition for standard NA in Sec. 2.2.1, as long as stride is greater than 1, this is no longer a vector-matrix multiplication, and rather a matrix-matrix multiplication. Stride effectively *coarsens* the fine-grained attention mask, as it ensures consistency between the masks of queries within the same group, which increases local similarity between attention masks of different queries. As long as certain assumptions can be made about the values for window size, stride, and the design of the kernel (specifically dynamic vs static KV tiling), this can extend to every tile of work, which can reach full block-sparsity. It is noteworthy, however, that stride is not guaranteed to improve work tile statistics. In Fig. 4.5, we visualize a static block-sparse mask for a simple 1-D use case, in which strides 2 through 7 either maintain or decrease the number of saved tiles, and stride 8 is the first to decrease the number of tiles, and is fully block-sparse. Visualizations are not enough, nor capture the full range of design choices for FNA kernels. We therefore develop a simulator program that can quickly compute scheduling statistics for various FNA implementations, given a problem size. We discuss this further in the next section.

Strided NA exhibits similar behavior as strided SWA in HaloNet [10], with their "block size" parameter introducing the same delay step effect into sliding window attention [7, 9] as stride does in NA. Both approaches are used to trade off translational equivariance for efficiency, but in completely different scenarios. As mentioned in Sec. 4.1.3, in this work we employ stride specifically to improve the throughput by reducing the number of wasted FLOPs, while HaloNet was concerned with the memory footprint of their implementation. In our case, stride has no effect on global memory footprint. On the other hand, both have the same effect on translational equivariance, which we visualize in Fig. 4.6. Larger strides break translational equivariance by introducing larger "patch" effects, eventually approaching Blocked Attention, which is the extreme case of strided NA: when stride is equal to window size.

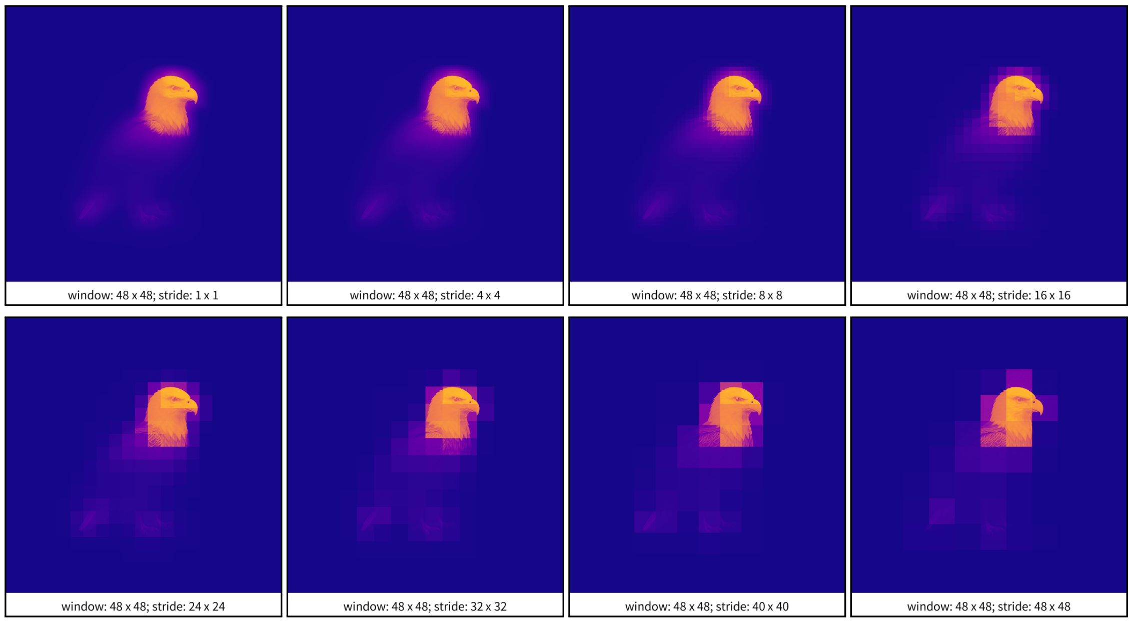

window: 48 x 48; stride: 1 x 1
window: 48 x 48; stride: 4 x 4
window: 48 x 48; stride: 8 x 8
window: 48 x 48; stride: 16 x 16
window: 48 x 48; stride: 24 x 24
window: 48 x 48; stride: 32 x 32
window: 48 x 48; stride: 40 x 40
window: 48 x 48; stride: 48 x 48

**Figure 4.6. Effect of stride on translational equivariance.** Similar to the visualizations in Fig. 2.2, we apply NA on an input image, but here we focus on the effect of increasing stride. Larger strides introduce a "patch"-like effect, approaching Blocked Attention.

### 4.2.3 NATTEN Simulator

Stride extends an already vast parameter space. This, along with the different design choices and configurations in implementation, specifically that of FNA kernels, makes it difficult to find useful parameters for end-users that offer the best quality-efficiency trade-off. We therefore created an analytical tool called NATTENSim, which can shed light on exactly that. NATTENSim computes the number of KV tiles visited by an implementation of FNA, taking into account design choices and the problem size. The process is comprised of transforming the shapes of the problem according to the tiling and scheduling logic, and counting the resulting number of work tiles. Design choices that affect this are:

**Single vs multi-dimensional tiling.** Though not recommended, FNA can still be implemented with single-dimensional tiling. Some implementations may choose this over multi-dimensional tiling for ease of implementation. NATTENSim can simulate this behavior, as well as the multi-dimensional tiling case.

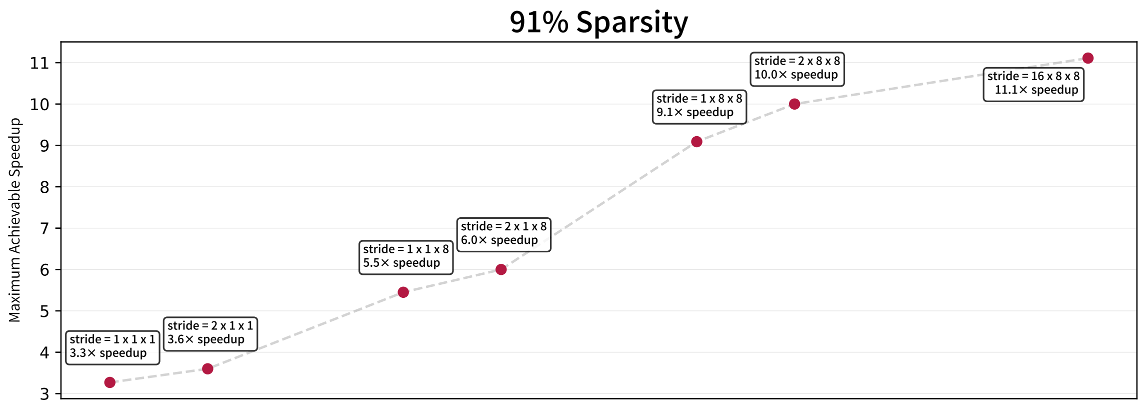


**Figure 4.7. Sweep of different stride values and their analytical speedups according to NATTENSim.** We assume a Q tile shape of 4 × 8 × 8, and KV tile shape of 2 × 8 × 8, a combination supported by our new kernels. The use case is from a video generation model, with a 30 × 48 × 80 token layout and a window size of 18 × 24 × 24 (≈ 91% sparsity). Standard NA (stride 1) will be limited by a 3.3× speedup, while some larger strides can improve that, and eventually cross 9× speedup with a stride of 8 × 8 across the spatial dimensions. With some larger strides along the temporal dimension, it can reach full block-sparsity, and yield a speedup of 11.1×, which is proportional to reduction in FLOPs.

**Static vs dynamic KV tiling.** The original FNA dynamically tiles KV, which can result in fewer work tiles, while our new approach must always use static KV tiling.

**Tile sizes / shapes.** FMHA kernels, especially the ones specialized for a specific architecture, very rarely have the same freedom over choosing tile sizes as in GEMMs. For example, the CUTLASS Blackwell FMHA kernel [71], which we use in this work, only supports Q tile size 256 and KV tile size 128 in the forward pass, while the xFormers FMHA supported Q tile sizes 32, 64, and 128. In addition, different tile sizes do not typically yield the same performance for a given problem, which means we may prefer one tiling configuration over others. When performing multi-dimensional tiling, these tile sizes must be reinterpreted as multi-dimensional shapes. These tile sizes / shapes directly affect the block sparsity of the attention map, which affects how many blocks (tiles) are computed for each Q tile, which correlates with performance.

There are many ways to utilize NATTENSim. It can be used to pick the best tiling configuration and other design choices for a specific use case of interest. This can be useful since different design choices, unless already implemented, will have their own challenges and therefore implementation timelines. Predicting the expected return from a specific implementation can be used to justify it, or rule it out.

The most important use of NATTENSim in this work is finding the optimal stride values, and even other NA parameters, based on our new implementation. We illustrate this in Fig. 4.7, where we fix the design choices according to compatibility with our new implementations, run NATTENSim over a video generation use case which we later experiment with, and with a window size that introduces approximately 90% sparsity, sweeping over all possible stride values. This points us to the most efficient, and in this case block-sparse, use case, and the intermediate points can additionally be used in ablation studies for finding the best trade-off between quality and efficiency. NATTENSim can also replace the former FNA auto-tuner, as it can predict performance levels over different tiling configurations without having to profile or benchmark their actual runtimes.

## 4.3 Experiments

Given that the improvements over existing use cases, namely NAT and DiNAT, have already saturated (see Sec. 3.3), we study the power and efficiency of GNA and our new implementations in a new set of applications. We specifically chose models limited by Self Attention: workloads in which Self Attention accounts for 50% or more of the end-to-end runtime. We find that diffusion-based generative models, specifically Video and World Foundation Models (WFMs), easily meet this criterion on the Hopper and Blackwell architectures, provided that the total token count is approximately around 50,000 or more. Our experiments are divided into two categories: 1. Replacing Self Attention with GNA without further fine-tuning, and 2. Replacing Self Attention with GNA with some fine-tuning on the original set of data on which the model was trained.

**Table 4.1. Workload distribution with respect to Self Attention in off-the-shelf generative models.** Measurements were done on a single NVIDIA B200 GPU and without any additional performance optimizations, and using the original FP16 / BF16 precision.

| Use case | % Self Attn. in diffusion | % diffusion in E2E workload | % Self Attn. in E2E workload |
|---|---|---|---|
| **Cosmos Predict 1 7-B, 720p, 5s** | 58.7% | $> 99\%$ | 58.7% |
| **Hunyuan Video 14-B, 720p, 5s** | 65.4% | 92.8% | 60.7% |
| **FLUX 1.dev, 4K** | 56.8% | 91.2% | 51.8% |

### 4.3.1 GNA without fine-tuning

Our final candidates for this category are: Cosmos Predict 1 [2] (WFM), Hunyuan Video [1] (video generation), and FLUX 1.dev [76] (image generation). We experiment with two different sparsity levels, a medium sparsity setting (between 50-60% attention sparsity) and a high sparsity setting (between 85-95% attention sparsity). For each sparsity setting, we search various GNA window sizes that introduce sparsity levels within the expected ranges, and are not grossly imbalanced across dimensions (high sparsity along one dimension and low sparsity along another). We also constrain window sizes to be multiples of valid KV tile shapes in our new FNA implementations, as we're trying to reach full block-sparsity. We also experiment with two schedules: 1. Applying sparsity in all diffusion steps for exploring the limits of speedups, and 2. Retaining Self Attention in the first few steps for better quality. The number of diffusion steps retaining Self Attention are adjusted experimentally based on observing quality degradation over a small set of outputs.

Due to hardware constraints, this study is limited to the NVIDIA Blackwell architecture, which is the most recent architecture. All performance measurements were done on an NVIDIA B200 GPU. However, as we will show in the next study (Sec. 4.3.2) which investigates performance on both Hopper and Blackwell architectures, performance measurements of our new FNA implementation and end-to-end speedups are even more favorable on the Hopper architecture. We present the workload distribution for each model in Tab. 4.1.

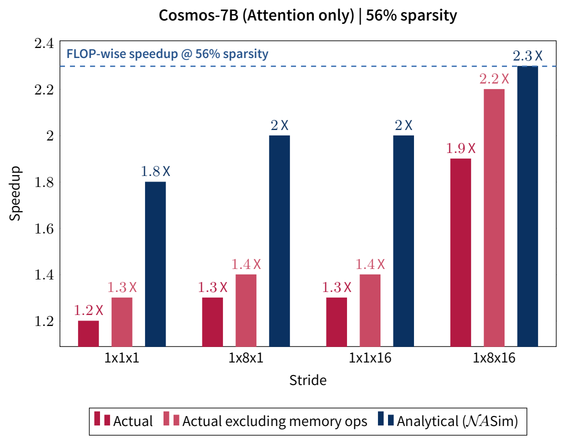
For each model, we present three sets of results:

**1. Operation-level speedup measurements over different strides.** Strides are filtered by NATTENSim to exclude those that despite an increase in stride do not improve performance. The largest stride in each group is fully-block sparse. For each model, we present three speedup measurements: an analytical measure based on NATTENSim, which is the theoretically maximum speedup possible, an actual measurement including the cost of token permutation, and an actual measurement excluding the cost of token permutation. The last measurement highlights implementation overheads in the compute kernel, and performance expected when optimizations to token permutation, as discussed in Sec. 4.2.1, are implemented. We note that no such optimizations have been implemented for the experiments in this section, as we find the cost of token permutation is typically within the margin of error in end-to-end measurements. These operation-level measurements are presented in Figs. 4.8 and 4.9.

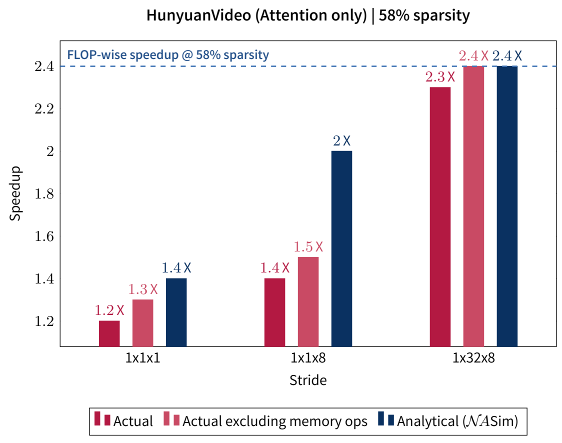
**2. End-to-end speedup measurements over different settings.** These measurements, presented in Tabs. 4.2 to 4.4, cover all sparsity settings and diffusion schedules, and present three three measurements: analytical speedup based on FLOPs (identical for any sparse approach with the same window size / level of attention sparsity), analytical speedup based on NATTENSim, and actual speedup with our Blackwell FNA implementation.

**3. Qualitative results.** We present samples from each model (still-frames from the video models) when running without any sparsity, then with our best non-strided NA configuration, and finally with our best strided configuration (GNA). We again note that the final strided configurations are all fully block-sparse. These samples are presented in Figs. 4.10 to 4.12.

Details regarding integration, workload, our choices of window and tile shapes, are as follows:



**(a)** Cosmos use case with 56% sparsity.



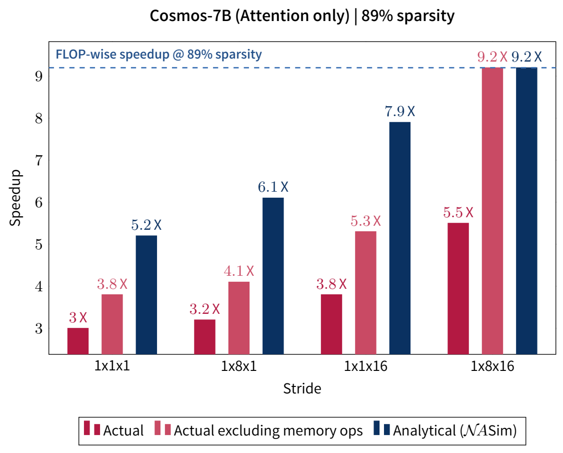
**(b)** Hunyuan use case with 58% sparsity.

**Figure 4.8. Operation-level speedups from GNA on off-the-shelf generative models with medium-level sparsity.** Analytical speedup is according to NATTENSim, and actual speedups are measured by running our Blackwell FNA kernel on an NVIDIA B200 GPU.

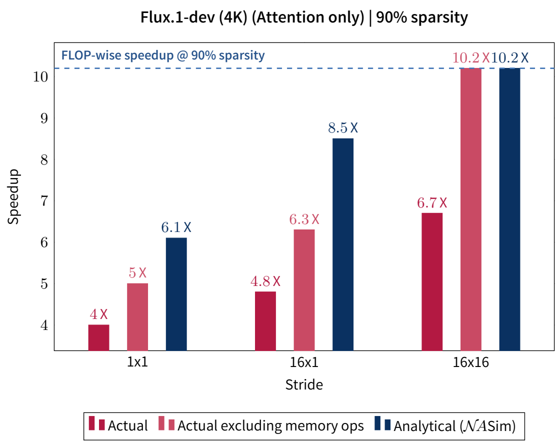
**Cosmos Predict 1.** This WFM uses the vanilla DiT architecture [75]. We specifically use the 7 billion parameter (7-B) variant, and our chosen workload produces 5 seconds of 720p resolution outputs. This workload involves a token layout shape of $16 \times 44 \times 80$ (56,320 tokens). The default diffusion scheduler uses 35 diffusion steps.

For our Blackwell FNA kernel, we set Q tile shape to $8 \times 4 \times 8$ and KV tile shape to $4 \times 4 \times 8$, which are compatible with the underlying FMHA tile sizes of 256 and 128, and evenly divide the token layout shape. This particular model is very sensitive to sparsity along the temporal dimension, as a result of which we only introduce sparsity along the spatial dimensions. For the medium sparsity setting, we chose window size $16 \times 32 \times 48$, which is approximately 56% sparsity. For the high sparsity setting, we chose window size $16 \times 24 \times 16$, which is approximately 89% sparsity. For non-sparse diffusion steps, we find that retaining the first 12 steps out of 35 provides the best trade-off.

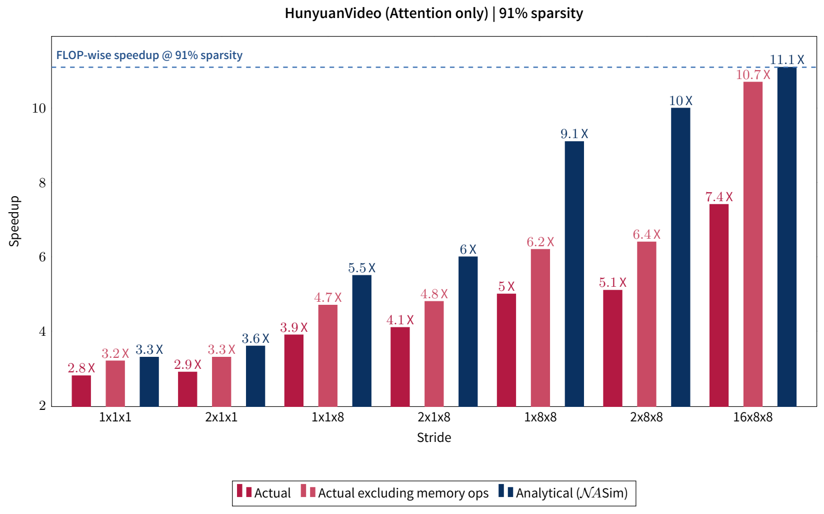
We present operation-level speedups for this model in Figs. 4.8a and 4.9a, end-to-end speedups in Tab. 4.2, and still-frame samples in Fig. 4.10. With this model, we find that across settings, GNA can achieve 97% or higher of the maximum speedup theoretically possible with our Blackwell FNA and naive token permutation implementation. The best



**(a)** Cosmos use case with 89% sparsity.



**(b)** FLUX use case with 90% sparsity.



**(c)** Hunyuan use case with 91% sparsity.

**Figure 4.9. Operation-level speedups from GNA on off-the-shelf generative models with high-level sparsity.** Analytical speedup is according to NATTENSim, and actual speedups are measured by running our Blackwell FNA kernel on an NVIDIA B200 GPU.

setting for speedups is naturally the high sparsity setting with all sparse diffusion steps, in which we achieve a 2.05× speedup, with the maximum possible speedup of 2.10×. However, the only setting in which we can preserve reasonable generation quality is the medium sparsity setting, with 12 non-sparse diffusion steps. In this setting, we achieve a 1.26× speedup (maximum possible is 1.28×). In the next study (Sec. 4.3.2), we experiment with fine-tuning the successor to this model, Cosmos Predict 2, which can achieve up to 2.3× speedup and maintain quality with limited fine-tuning.

**Table 4.2. Cosmos end-to-end speedups under different sparsity levels and strides.** We report both analytical speedups based on NATTENSim, as well as actual speedups on B200. We also report two settings: one with all diffusion steps done with GNA, and the other with the first 12 steps done with Self Attention. The former can be more representative of cases where the model can be fine-tuned with GNA, while the latter is more appropriate for off-the-shelf application.

| Window Size | Stride | # SA Steps | E2E Speedup ↑ | | |
|---|---|---|---|---|---|
| | | | Analytical | | Actual |
| | | | FLOP-wise | NASim | |
| **56% sparsity** | | | | | |
| $16 \times 32 \times 48$ | $1 \times 1 \times 1$ | 0 | 1.50× | 1.35× | 1.18× |
| $16 \times 32 \times 48$ | $1 \times 8 \times 1$ | 0 | 1.50× | 1.42× | 1.20× |
| $16 \times 32 \times 48$ | $1 \times 1 \times 16$ | 0 | 1.50× | 1.42× | 1.21× |
| $16 \times 32 \times 48$ | $1 \times 8 \times 16$ | 0 | 1.50× | 1.50× | 1.46× |
| $16 \times 32 \times 48$ | $1 \times 1 \times 1$ | 12 | 1.28× | 1.21× | 1.11× |
| $16 \times 32 \times 48$ | $1 \times 8 \times 1$ | 12 | 1.28× | 1.24× | 1.12× |
| $16 \times 32 \times 48$ | $1 \times 1 \times 16$ | 12 | 1.28× | 1.24× | 1.13× |
| $16 \times 32 \times 48$ | $1 \times 8 \times 16$ | 12 | 1.28× | 1.28× | 1.26× |
| **89% sparsity** | | | | | |
| $16 \times 24 \times 16$ | $1 \times 1 \times 1$ | 0 | 2.10× | 1.90× | 1.76× |
| $16 \times 24 \times 16$ | $1 \times 8 \times 1$ | 0 | 2.10× | 1.96× | 1.79× |
| $16 \times 24 \times 16$ | $1 \times 1 \times 16$ | 0 | 2.10× | 2.05× | 1.88× |
| $16 \times 24 \times 16$ | $1 \times 8 \times 16$ | 0 | 2.10× | 2.10× | 2.05× |
| $16 \times 24 \times 16$ | $1 \times 1 \times 1$ | 12 | 1.52× | 1.45× | 1.40× |
| $16 \times 24 \times 16$ | $1 \times 8 \times 1$ | 12 | 1.52× | 1.48× | 1.40× |
| $16 \times 24 \times 16$ | $1 \times 1 \times 16$ | 12 | 1.52× | 1.51× | 1.44× |
| $16 \times 24 \times 16$ | $1 \times 8 \times 16$ | 12 | 1.52× | 1.52× | 1.50× |

**Hunyuan Video.** This video generation model uses the Multimodal Diffusion Transformer (MMDiT) architecture [77]. The key difference concerning our experiments is that the Self Attention operation over video tokens and Cross Attention between video and text prompt tokens are combined into one Self Attention operator. To apply GNA, we separate the interaction between the different modalities (video and text) into 3 Attention operations. Text tokens cross-attend to both text and video tokens, without any sparsity introduced. Video tokens attend to all text tokens. The Self Attention over video tokens is replaced by GNA. We use attention merging, a feature implemented in NATTEN, to handle the video attention branch and combine the outputs from GNA and cross attention between video and text tokens. Similar to Cosmos Predict 1, we chose the 5-second 720p video generation use case, which has a token layout shape of $30 \times 48 \times 80$ (115,200 tokens).

**Table 4.3. Hunyuan end-to-end speedups different GNA parameters and settings.** We report both analytical speedups based on NATTENSim, as well as actual speedups on B200. We also report two settings: one with all diffusion steps done with GNA, and the other with the first 15 steps done with Self Attention. The former can be more representative of cases where the model can be fine-tuned with GNA, while the latter is more appropriate for off-the-shelf application.

| **Window Size** | **Stride** | **# SA Steps** | **E2E Speedup ↑** | | |
|---|---|---|---|---|---|
| | | | **Analytical** | | **Actual** |
| | | | **FLOP-wise** | **NASim** | |
| **58% sparsity** | | | | | |
| 30 × 40 × 40 | 1 × 1 × 1 | 0 | 1.55× | 1.21× | 1.15× |
| 30 × 40 × 40 | 1 × 1 × 8 | 0 | 1.55× | 1.44× | 1.26× |
| 30 × 40 × 40 | 1 × 32 × 8 | 0 | 1.55× | 1.55× | 1.53× |
| 30 × 40 × 40 | 1 × 1 × 1 | 15 | 1.33× | 1.14× | 1.08× |
| 30 × 40 × 40 | 1 × 1 × 8 | 15 | 1.33× | 1.27× | 1.15× |
| 30 × 40 × 40 | 1 × 32 × 8 | 15 | 1.33× | 1.33× | 1.30× |
| **91% sparsity** | | | | | |
| 18 × 24 × 24 | 1 × 1 × 1 | 0 | 2.23× | 1.73× | **1.73×** |
| 18 × 24 × 24 | 2 × 1 × 1 | 0 | 2.23× | 1.78× | 1.77× |
| 18 × 24 × 24 | 1 × 1 × 8 | 0 | 2.23× | 1.99× | 1.95× |
| 18 × 24 × 24 | 2 × 1 × 8 | 0 | 2.23× | 2.02× | 1.98× |
| 18 × 24 × 24 | 1 × 8 × 8 | 0 | 2.23× | 2.17× | 2.09× |
| 18 × 24 × 24 | 2 × 8 × 8 | 0 | 2.23× | 2.20× | 2.11× |
| 18 × 24 × 24 | 16 × 8 × 8 | 0 | 2.23× | 2.23× | **2.23×** |
| 18 × 24 × 24 | 1 × 1 × 1 | 15 | 1.63× | 1.42× | **1.42×** |
| 18 × 24 × 24 | 2 × 1 × 1 | 15 | 1.63× | 1.44× | **1.44×** |
| 18 × 24 × 24 | 1 × 1 × 8 | 15 | 1.63× | 1.53× | 1.51× |
| 18 × 24 × 24 | 2 × 1 × 8 | 15 | 1.63× | 1.55× | 1.54× |
| 18 × 24 × 24 | 1 × 8 × 8 | 15 | 1.63× | 1.61× | 1.58× |
| 18 × 24 × 24 | 2 × 8 × 8 | 15 | 1.63× | 1.62× | 1.59× |
| 18 × 24 × 24 | 16 × 8 × 8 | 15 | 1.63× | 1.63× | **1.63×** |

Unlike Cosmos, this 14-B parameter model runs for 50 diffusion steps.

In the Blackwell FNA kernel, we set Q tile shape to 2 × 16 × 8 and KV tile shape to 2 × 8 × 8, which are compatible with the underlying FMHA tile sizes of 256 and 128, and evenly divide the token layout shape. For the medium sparsity setting, we chose window size 30 × 40 × 40, which is approximately 58% sparsity. For the high sparsity setting, we chose window size 18 × 24 × 24, which is approximately 91% sparsity, but we use Q tile shape 4 × 8 × 8, as NATTENSim finds more balanced stride values for this configuration. For non-sparse diffusion steps, we find that retaining the first 15 steps out of 50 provides the best trade-off.

**Table 4.4. FLUX end-to-end speedups different GNA parameters and settings.** We report both analytical speedups based on NATTENSim, as well as actual speedups on B200. We also report two settings: one with all diffusion steps done with GNA, and the other with the first 9 steps done with self attention. The former can be more representative of cases where the model can be fine-tuned with GNA, while the latter is more applicable for off-the-shelf applications.

| **Window Size** | **Stride** | **# SA Steps** | **E2E Speedup ↑** | | |
|---|---|---|---|---|---|
| | | | **Analytical** | | **Actual** |
| | | | **FLOP-wise** | **NASim** | |
| 80 × 80 | 1 × 1 | 0 | 1.88× | 1.76× | 1.65× |
| 80 × 80 | 16 × 1 | 0 | 1.88× | 1.84× | 1.72× |
| 80 × 80 | 16 × 16 | 0 | 1.88× | 1.88× | 1.82× |
| 80 × 80 | 1 × 1 | 9 | 1.46× | 1.42× | 1.37× |
| 80 × 80 | 16 × 1 | 9 | 1.46× | 1.45× | 1.40× |
| 80 × 80 | 16 × 16 | 9 | 1.46× | 1.46× | 1.45× |

We present operation-level speedups for this model in Figs. 4.8b and 4.9c, end-to-end speedups in Tab. 4.3, and still-frame samples in Fig. 4.11. With this model, we find that across settings, GNA can achieve 97% or higher of the maximum speedup theoretically possible with our Blackwell FNA and naive token permutation implementation. In some settings, GNA, and even NA can realize the maximum possible end-to-end speedup. The best setting for speedups is naturally the high sparsity setting with all sparse diffusion steps, in which we realize the full 2.23× speedup. However, to maintain quality across samples our final chosen setting retains 15 non-sparse diffusion steps, which a realizes its theoretical maximum speedup of 1.63×.

**FLUX 1.dev.** FLUX, like Hunyuan, also uses the MMDiT architecture [77], for which we use the same approach to introduce GNA into the multi-modal Self Attention operators. FLUX only meets our constraint of being attention-bound when generating 4K images, for which we had to use extra adapters from Yu et al. [78], as FLUX 1.dev does not natively support 4K image generation. In this workload, we end up with token layouts of shape 256 × 256 (65,536 tokens). The default number of diffusion steps is 28.

**Table 4.5. GNA configurations used in Cosmos Predict 2.** The 2-B and 14-B variants have 28 and 36 DiT layers respectively. We do not introduce any sparsity along the temporal dimension (T), and only introduce sparsity along the spatial dimensions (H and W). All configurations are fully block-sparse. We selected these configurations by measuring the mean squared error compared to Self Attention for each layer, and selecting the highest level of sparsity the error from which was smaller than a fixed threshold. The threshold itself was adjusted by conducting small ablations.

| **Configuration** | **GNA type** | **2-B Layers** | **14-B Layers** |
|---|---|---|---|
| 98% sparsity | dilated along H & W | 1, 2 | 1 - 11 |
| 95% sparsity | local along H, dilated along W | 3, 6, 11 | 12, 13 |
| 92% sparsity | local along H & W | 4, 5, 7, 8, 9, 10, 28 | 14 - 23 |
| 77% sparsity | local along H & W | 12 - 23, 25 - 27 | 24, 26, 30, 31, 34, 36 |
| 55% sparsity | local along H & W | 24 | 25, 27, 29, 32, 33, 35 |
| No sparsity | - | - | 28 |

In our Blackwell FNA kernel, we set Q tile shape to $16 \times 16$ and KV tile shape to $16 \times 8$, which are compatible with the underlying FMHA tile sizes of 256 and 128, and evenly divide the token layout shape. For this model, we only evaluate the high sparsity setting, as we find the quality to be sufficiently similar to the baseline. For this setting, we chose window size $80 \times 80$, which is approximately 90% sparsity. For non-sparse diffusion steps, we find that retaining the first 9 steps out of 28 provides the best trade-off.

We present operation-level speedups for this model in Fig. 4.9b, end-to-end speedups in Tab. 4.4, and samples in Fig. 4.12. With this model, we find that across settings, GNA can again achieve 97% or higher of the maximum speedup theoretically possible with our Blackwell FNA and naive token permutation implementation. The most sparse setting achieves a 1.82× end-to-end speedup (maximum possible is 1.88×). Our final setting, which maintains quality across samples, retains 9 non-sparse diffusion steps, and achieves an end-to-end speedup of 1.45×, which is approximately 99% of the maximum speedup theoretically possible in this setting, 1.46×.

**Table 4.6. Cosmos Predict 2 inference runtimes on various NVIDIA GPUs, with and without GNA.** Self Attention accounts for a larger portion of the workload in the 2-B variant compared to the 14-B variant, which is part of the reason why it can achieve higher speedups.

| GPU | 2-B Video2World | | 14-B Video2World | |
|---|---|---|---|---|
| | Base | + GNA | Base | + GNA |
| **NVIDIA B200** | 123.9 s | 54.0 s (2.3×) | 439.4 s | 223.1 s (2.0×) |
| **NVIDIA H200 SXM** | 221.7 s | 89.4 s (2.5×) | 836.9 s | 412.9 s (2.0×) |
| **NVIDIA H200 NVL** | 267.2 s | 104.3 s (2.6×) | 1006.7 s | 489.5 s (2.1×) |
| **NVIDIA H100 PCIe** | 378.5 s | 149.6 s (2.5×) | 1425.4 s | 706.9 s (2.0×) |
| **NVIDIA H100 NVL** | 355.7 s | 138.7 s (2.6×) | 1348.6 s | 677.0 s (2.0×) |
| **NVIDIA H100 SXM** | 228.8 s | 94.2 s (2.4×) | 856.9 s | 426.0 s (2.0×) |

### 4.3.2 GNA with fine-tuning

Our experiments in this section are limited to the Cosmos Predict 2 WFM, specifically the Video2World branch. The successor to the Cosmos Predict 1 model [2] in the previous set of experiments, this model has a slightly different architecture, but noticeably different workload statistics and token layout shapes. Unlike its predecessor, this model no longer has a mid-range 7-B variant, and instead is shipped with only two variants: a 2-B parameter, and a larger 14-B parameter model, both of which are included in this study.

**GNA integration.** For our experiments with this model, we again followed our prior findings and did not attempt to introduce sparsity along the temporal dimension, and only apply sparsity along the spatial dimensions. We first curated a list of useful sparsity configurations, which include both strided and dilated forms of NA. While dilated NA has limited applicability in off-the-shelf-deployment, prior experiments in Sec. 2.3 suggest that training with a mixture of local and dilated patterns can best recover the loss of accuracy / quality compared to Self Attention. We limited the configurations to only those that are fully block-sparse in order to maximize returns. Sparsity levels themselves range from 55% to 98%. Instead of training all possible configurations, we then inspected the mean squared error introduced by each sparsity configuration over each Self Attention layer. For

**Table 4.7. Physical AI Benchmark (PBench) scores on Cosmos Predict 2, with and without GNA.** We additionally include PBench scores from the previous generation, Cosmos Predict 1, as a point of reference. While the Cosmos Predict 2 family consistently outperform their predecessor for reasons unrelated to GNA, it is still noteworthy that the impact of high levels of attention sparsity using GNA is very insignificant comparatively.

| Variant | Domain score | Quality score | Overall score |
|---|---|---|---|
| **Cosmos Predict 1, 14-B** | 77.6 | 69.0 | 73.3 |
| **Cosmos Predict 2, 2-B** | 84.8 | 69.6 | 77.2 |
| **Cosmos Predict 2, 2-B + GNA** | 84.5 | 69.5 | 77.0 |
| **Cosmos Predict 2, 14-B** | 84.9 | 69.9 | 77.4 |
| **Cosmos Predict 2, 14-B + GNA** | 84.2 | 69.9 | 77.0 |

each layer, we select the highest level of sparsity with its error within a certain threshold, which we set by conducting small experiments. Our final configurations are summarized in Tab. 4.5. End-to-end speedups achievable with these variants are reported in Tab. 4.6. We find that across both Hopper and Blackwell GPUs, we can achieve end-to-end speedups of approximately 2.3× - 2.6× in the 2-B variant, and 2.0× - 2.1× in the 14-B variant.

**Training setup and infrastructure.** We trained the GNA variants using identical hyperparameters and setup as the original models, resuming from checkpoints of the originals. The hardware setup was a cluster of H100 DGX boxes, each box containing 8 NVIDIA H100 GPUs, with a high-bandwidth all-to-all NVLink network used for model parallelism. This setup therefore utilizes model parallelism among the 8 GPUs in each group. The Attention operator uses head-parallelism via the All2All collective, since there are enough attention heads in each variant, all of which are divisible by 8. This also eases our implementation, as each GPU gathers all tokens corresponding to their slice of heads, requiring minimal implementation for handling token parallelism. Fine-tuning was done only for approximately 50,000 training iterations for each variant, which in both cases took only a few days, a fraction of the total training time for the original models. In addition, with our Hopper FNA kernels, training time was improved noticeably compared to the original models as well, but less significantly than our reported inference times, since model forward and backward passes have different workload statistics.

**Quality evaluation.** We conducted a blind qualitative comparison of over 100 samples from each of the 2-B and 14-B variants, generated with and without GNA, and found that in the majority of cases, they are tied, and in approximately half of the remaining cases, the GNA variants exhibited lower quality, while in the other half they exhibited better quality. Even in the cases where there was a quality difference, those differences can easily be classified as artifacts of different training checkpoints. From this, we conclude that these fine-tuned GNA variants have limited impact on output quality. We present still-frame comparisons from a few samples in Fig. 4.13. We also present quantitative results over Physical AI Benchmark (PBench), which is a benchmark designed for evaluating Physical AI models like Cosmos, in Tab. 4.7.

The Cosmos Predict 2 variants with GNA are publicly available as part of the Cosmos Predict 2 project[2].

## 4.4 Conclusion

In this final chapter, we resolved the 4 major remaining bottlenecks of Neighborhood Attention (NA) and its most efficient implementation, Fused Neighborhood Attention (FNA).

We addressed the shortcomings of the original FNA by decomposing the kernel into a small memory operation and the core compute kernel, eliminating much of the implementation overhead observed in the kernel, while simplifying its design. This simplification allowed us to implement FNA for the Hopper and Blackwell architectures, on top of state-of-the-art FMHA kernels for the two. These implementations can finally allow NA to deliver speedups compared to Self Attention, and come with a powerful simulation program, NATTENSim, that can quickly compute the upper bounds of speedup for any given configuration and use case.

In addition, we introduced Generalized Neighborhood Attention (GNA), which adds a new parameter to NA that allows it to save more compute, potentially reach full block-

[2]https://github.com/nvidia-cosmos/cosmos-predict2

sparsity (wasting 0 FLOPs), and creates a new spectrum of attention patterns between NA and Blocked Attention (considered one of the most efficient forms of sparse attention). This unifies many attention patterns under one family, all implementable with our new FNA kernels, and analyzable with NATTENSim. NATTENSim further eliminates the need for time-consuming profiling or auto-tuning of NA, the last remaining bottleneck.

To evaluate this new methodology, we plugged GNA into three attention-bound foundation models without any further fine-tuning, as well as another foundation model which we were able to fine-tune for a short period of time. GNA can accelerate the former three by up to 1.63× end-to-end, matching the maximum speedup theoretically possible. In the latter model, GNA can accelerate some variants by up to 2.6× end-to-end, a figure typically unattainable with one single optimization.

All of our new implementations and tools are open-sourced and shipped as part of the NATTEN project, allowing researchers to directly use NA and GNA in their work and enjoying the speedups.

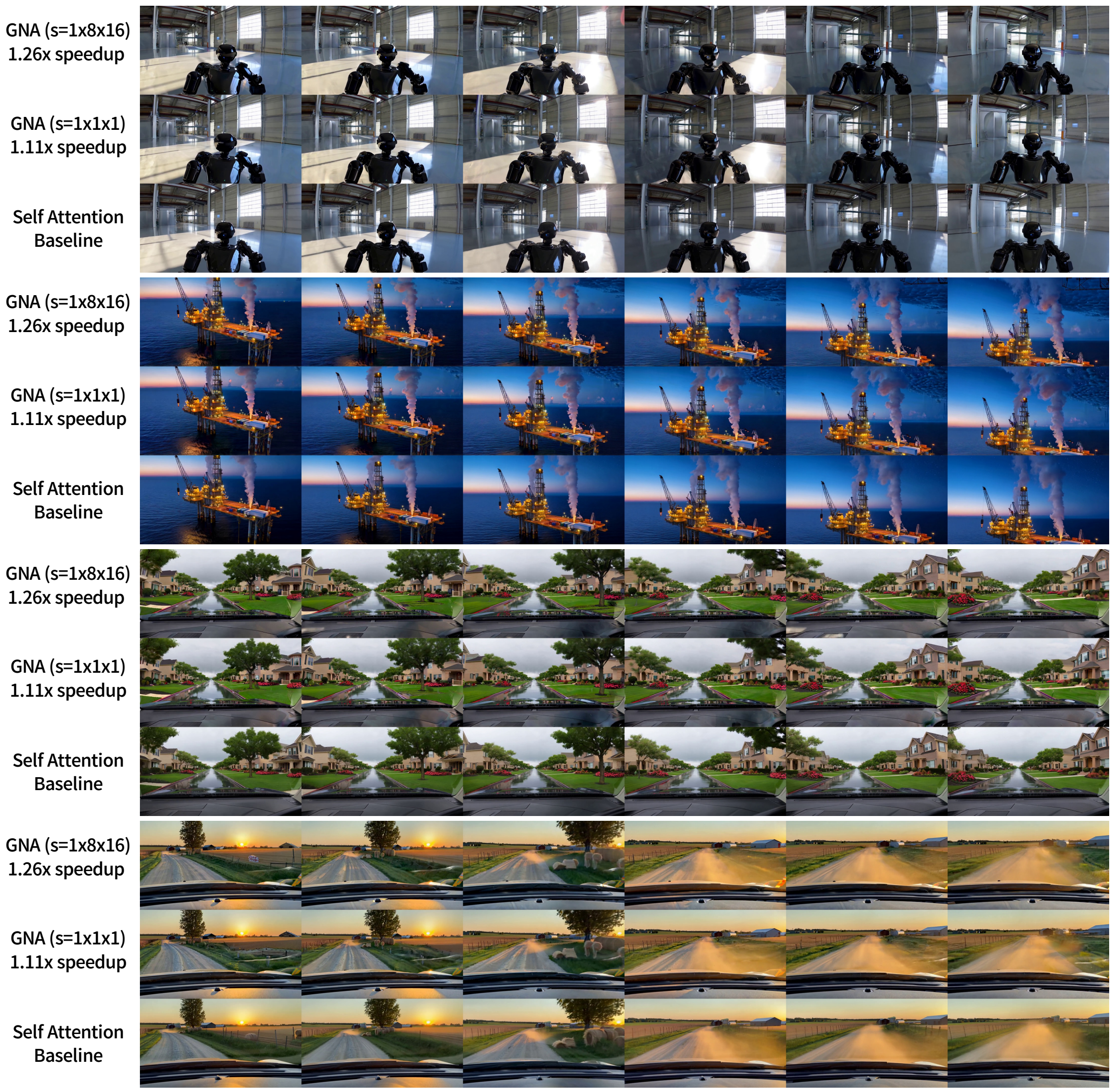


**Figure 4.10. Sample frames from videos generated by Cosmos Predict 1, with GNA introduced into the last 23 of the 35 diffusion steps.** Window size is $16 \times 32 \times 48$ ($\approx$ 56% sparsity). Speedup limit under this setting, with the same level of sparsity, is 1.28×.

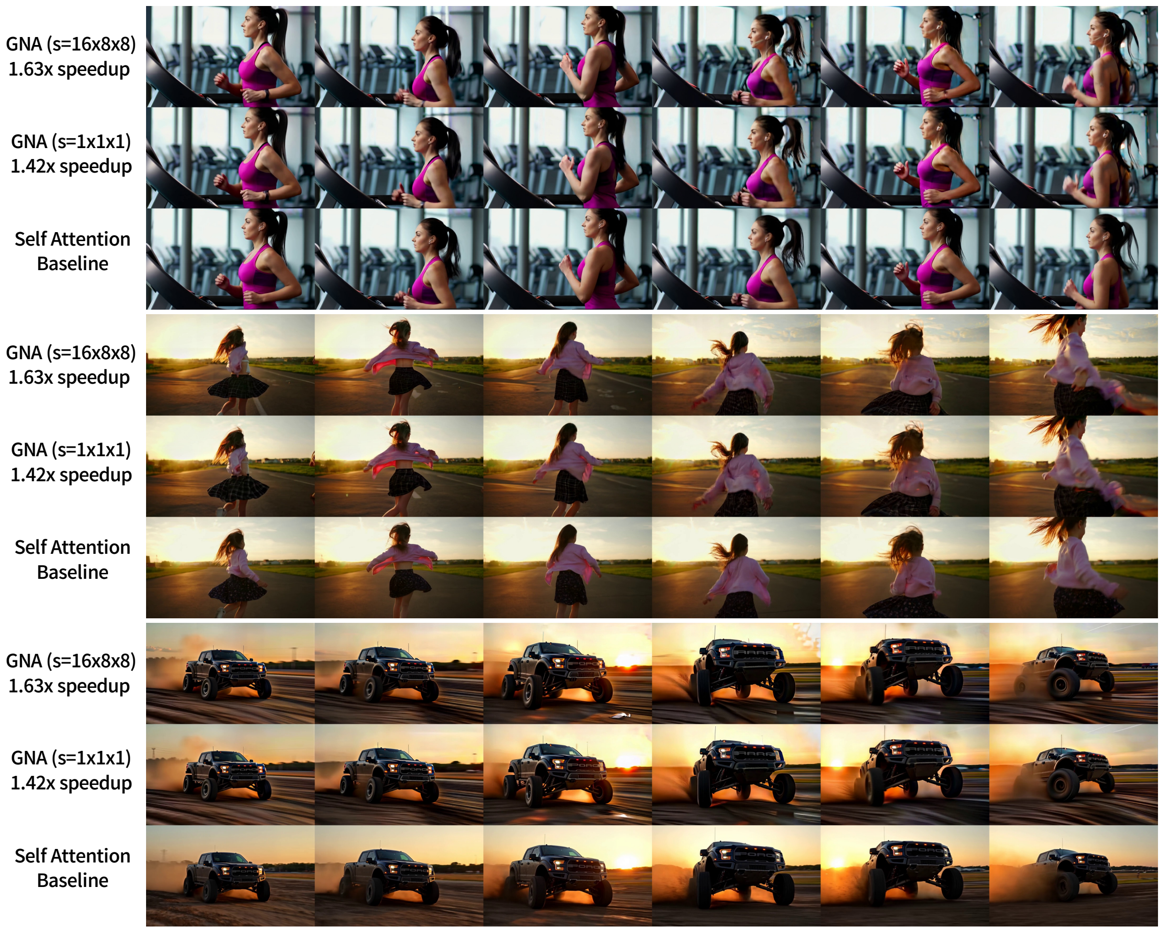


**Figure 4.11. Sample frames from videos generated by Hunyuan Video, with GNA introduced into the last 35 of the 50 diffusion steps.** Window size is $18 \times 24 \times 24$ ($\approx$ 91% sparsity). Speedup limit under this setting, with the same level of sparsity, is 1.63×.

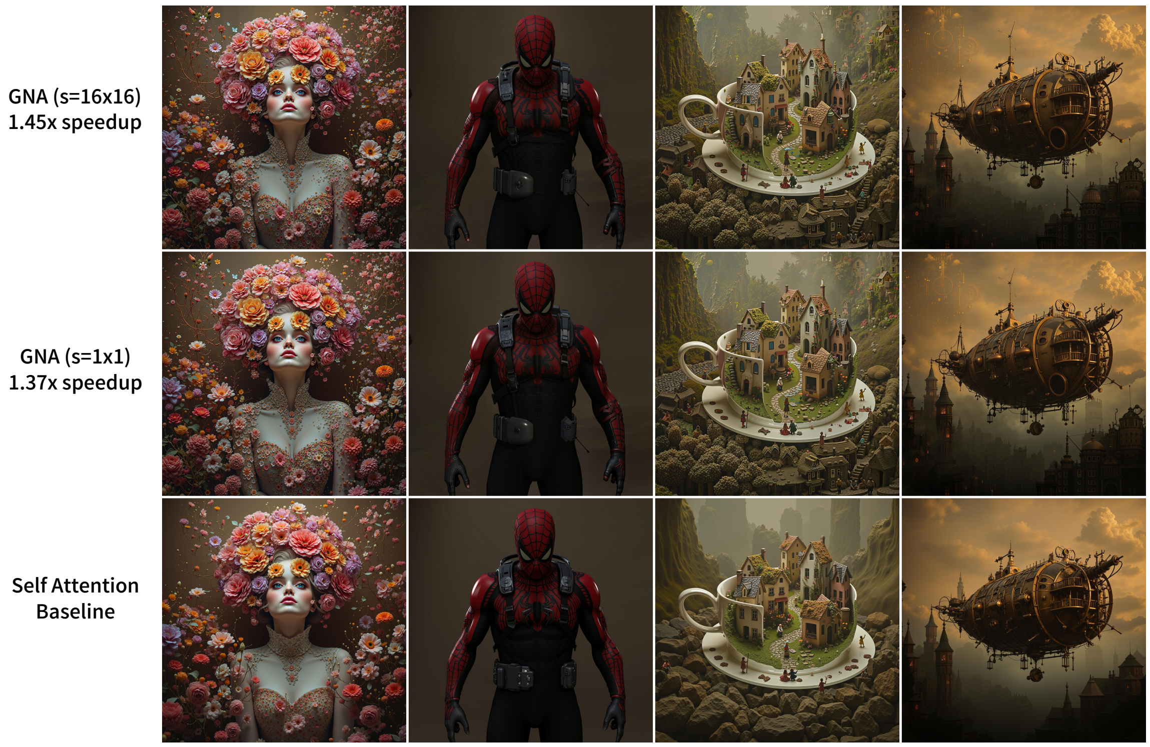


**Figure 4.12. Images generated by ultra-resolution FLUX, with GNA introduced into the last 19 of the 28 diffusion steps.** Window size is $80 \times 80$ ($\approx 90\%$ sparsity). Speedup limit under this setting, with the same level of sparsity, is $1.46\times$.

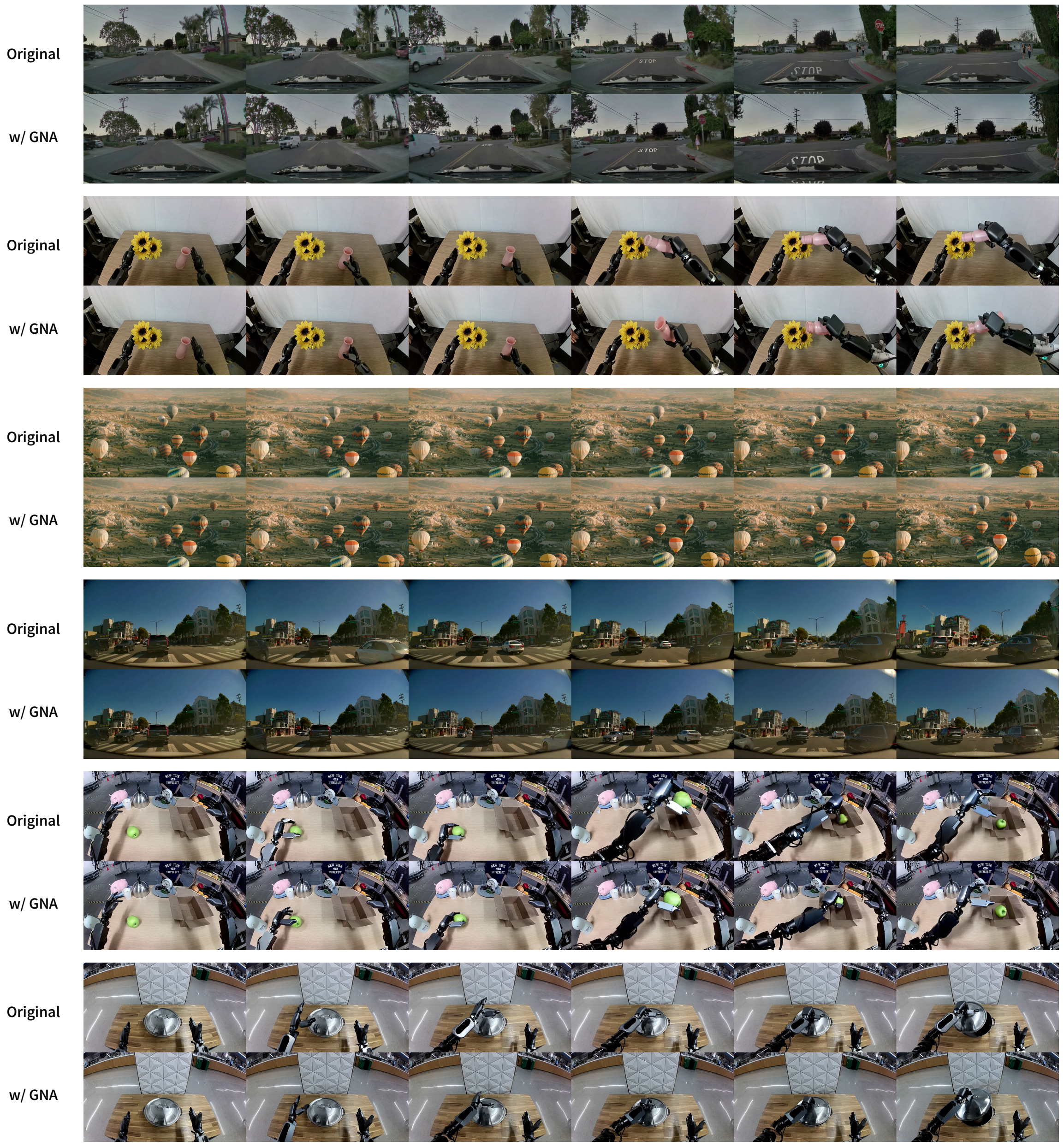


**Figure 4.13. Sample frames from videos generated by the base Cosmos Predict 2 model, and the GNA variant.** The GNA variants offer up to 2.6× end-to-end speedup in the 2-B variant, which is more attention-bound, and up to 2.1× end-to-end speedup in the 14-B variant.

# CHAPTER 5
# CONCLUSION

Attention, whether all we need or not, is what still powers most AI foundation models. Visual foundation models, specifically Video and World Foundation Models, can spend more than 80% of their runtime on Attention. Only recently have we seen a surge in works that attempt to study or address Attention's quadratic complexity, but even now, the focus is largely on Large Language Models (LLMs), where tokens are laid out in a single-dimensional sequence, while visual models tend to have multi-dimensional token layouts. In this work, we studied Sliding Window Attention methods for multi-dimensional layouts of tokens, from multiple points of view.

From a deep learning architecture point of view, we first propose a new pattern that fixes a critical issue in the original methodology proposed by Ramachandran et al. [9], called Neighborhood Attention (NA). We discuss its useful properties such as translational equivariance, local receptive fields, and sub-quadratic complexity. We introduce Dilated Neighborhood Attention (DiNA), which maintains the complexity, and thus, efficiency of NA, but accelerates its receptive field growth, leading to significantly improved performance across multiple computer vision tasks. We investigate the infrastructure and implementation woes that prior to this work had misled the research community into thinking these approaches are inefficient, and show that with even naive but lower-level implementations of the approach, we can achieve competitive throughput with respect to approaches commonly hailed as highly efficient.

From a high-performance computing and architecture point of view, we investigate sound methodologies for accelerating these approaches, taking into account the advances in implementations for Attention (FMHA), and the fact that dense linear algebra is the key to utilizing the ever-growing computational power of modern GPUs. We build implementations of NA based on dense linear algebra primitives, which can now target hardware units such as Tensor Cores. These new implementations, dubbed Fused Neighborhood Attention (FNA), while significantly improve the existing NA infrastructure, can still be limited by a few factors, the most important of which is the rapid evolution of modern GPU architectures. Until recently, architectural changes did not significantly alter programming models, and were forward-compatible with future architectures. Aside from that, it is not a reasonable expectation for highly performant software designed for one architecture to exhibit the same high performance on later architectures.

We therefore re-design our FNA approach to be less invasive of the structure of the core compute kernel, and therefore less specific to architecture, and instead decompose the approach into distinct memory and compute operations. In addition, we introduce Generalized Neighborhood Attention (GNA), which gives even more fine-grained control over the NA pattern, and allows one to trade off useful inductive biases such as translational equivariance, for even better efficiency. Through these efforts, we successfully demonstrate that the Neighborhood Attention family of attention patterns can accelerate large-scale foundation models. These improvements can be as large as 1.6× end-to-end with limited sparsity and no fine-tuning, and 2.6× end-to-end with high levels of sparsity and limited fine-tuning.

# REFERENCES

[1] W. Kong *et al.*, "Hunyuanvideo: A systematic framework for large video generative models," *arXiv preprint arXiv:2412.03603*, 2024.

[2] N. Agarwal *et al.*, "Cosmos world foundation model platform for physical ai," *arXiv preprint arXiv:2501.03575*, 2025.

[3] A. Vaswani *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[4] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, *Improving language understanding by generative pre-training*, 2018.

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019.

[6] A. Gulati *et al.*, "Conformer: Convolution-augmented transformer for speech recognition," *Interspeech*, 2020.

[7] N. Parmar *et al.*, "Image transformer," in *International Conference on Machine Learning (ICML)*, 2018.

[8] A. Dosovitskiy *et al.*, "An image is worth 16 x 16 words: Transformers for image recognition at scale," in *International Conference on Learning Representations (ICLR)*, 2020.

[9] P. Ramachandran, N. Parmar, A. Vaswani, I. Bello, A. Levskaya, and J. Shlens, "Stand alone self-attention in vision models," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[10] A. Vaswani, P. Ramachandran, A. Srinivas, N. Parmar, B. Hechtman, and J. 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.

[11] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," *arXiv preprint arXiv:1904.10509*, 2019.

[12] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," in *European Conference on Computer Vision (ECCV)*, 2020.

[13] Z. Huang *et al.*, “Ccnet: Criss-cross attention for semantic segmentation,” in *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2020.

[14] H. Wang, Y. Zhu, B. Green, H. Adam, A. Yuille, and L.-C. Chen, “Axial-deeplab: Stand-alone axial-attention for panoptic segmentation,” in *European Conference on Computer Vision (ECCV)*, 2020.

[15] Y. LeCun *et al.*, “Backpropagation applied to handwritten zip code recognition,” *Neural computation*, 1989.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

[17] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[18] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, “Training data-efficient image transformers & distillation through attention,” in *International Conference on Machine Learning (ICML)*, 2020.

[19] H. Touvron, M. Cord, A. Sablayrolles, G. Synnaeve, and H. Jégou, “Going deeper with image transformers,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[20] Y. Li, H. Mao, R. Girshick, and K. He, “Exploring plain vision transformer backbones for object detection,” in *European Conference on Computer Vision (ECCV)*, 2022.

[21] K. He, X. Chen, S. Xie, Y. Li, P. Dollár, and R. Girshick, “Masked autoencoders are scalable vision learners,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[22] W. Peebles and S. Xie, “Scalable diffusion models with transformers,” *arXiv preprint arXiv:2212.09748*, 2022.

[23] Z. Liu *et al.*, “Swin transformer: Hierarchical vision transformer using shifted windows,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[24] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, “A convnet for the 2020s,” *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[25] M. N. Rabe and C. Staats, “Self-attention does not need $O(n^2)$ memory,” *arXiv preprint arXiv:2112.05682*, 2021.

[26] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, “Flashattention: Fast and memory-efficient exact attention with io-awareness,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[27] K. Crowson, S. A. Baumann, A. Birch, T. M. Abraham, D. Z. Kaplan, and E. Shippole, “Scalable high-resolution pixel-space image synthesis with hourglass diffusion transformers,” in *International Conference on Machine Learning (ICML)*, 2024.

[28] H. Du *et al.*, “Weathermesh-3: Fast and accurate operational global weather forecasting,” *arXiv preprint arXiv:2503.22235*, 2025.

[29] I. Beltagy, M. E. Peters, and A. Cohan, “Longformer: The long-document transformer,” *arXiv preprint arXiv:2004.05150*, 2020.

[30] A. Arnab, M. Dehghani, G. Heigold, C. Sun, M. Lučić, and C. Schmid, “Vivit: A video vision transformer,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[31] A. Hassani, S. Walton, N. Shah, A. Abuduweili, J. Li, and H. Shi, “Escaping the big data paradigm with compact transformers,” *arXiv:2104.05704*, 2021.

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

[33] A. Q. Jiang *et al.*, “Mistral 7b,” *arXiv preprint arXiv:2310.06825*, 2023.

[34] T. Cohere *et al.*, “Command a: An enterprise-ready large language model,” *arXiv preprint arXiv:2504.00698*, 2025.

[35] Y. Rao, W. Zhao, B. Liu, J. Lu, J. Zhou, and C.-J. Hsieh, “Dynamicvit: Efficient vision transformers with dynamic token sparsification,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[36] Y. Liang, C. Ge, Z. Tong, Y. Song, J. Wang, and P. Xie, “Not all patches are what you need: Expediting vision transformers via token reorganizations,” in *International Conference on Learning Representations (ICLR)*, 2022.

[37] D. Bolya, C.-Y. Fu, X. Dai, P. Zhang, C. Feichtenhofer, and J. Hoffman, “Token merging: Your ViT but faster,” in *International Conference on Learning Representations (ICLR)*, 2023.

[38] D. Bolya and J. Hoffman, “Token merging for fast stable diffusion,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2023.

[39] Z. Chen, Z. Qu, Y. Quan, L. Liu, Y. Ding, and Y. Xie, “Dynamic n: M fine-grained structured sparse attention mechanism,” in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023.

[40] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, “Linformer: Self-attention with linear complexity,” *arXiv preprint arXiv:2006.04768*, 2020.

[41] B. Chen, T. Dao, E. Winsor, Z. Song, A. Rudra, and C. Ré, “Scatterbrain: Unifying sparse and low-rank attention,” *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[42] N. Shazeer, “Fast transformer decoding: One write-head is all you need,” *arXiv preprint arXiv:1911.02150*, 2019.

[43] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebron, and S. Sanghai, “Gqa: Training generalized multi-query transformer models from multi-head checkpoints,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[44] A. Liu *et al.*, “Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model,” *arXiv preprint arXiv:2405.04434*, 2024.

[45] A. Roy, M. Saffar, A. Vaswani, and D. Grangier, “Efficient content-based sparse attention with routing transformers,” *Transactions of the Association for Computational Linguistics (TACL)*, vol. 9, 2021.

[46] J. Yuan *et al.*, “Native sparse attention: Hardware-aligned and natively trainable sparse attention,” *arXiv preprint arXiv:2502.11089*, 2025.

[47] L. Huang, Y. Yuan, J. Guo, C. Zhang, X. Chen, and J. Wang, “Interlaced sparse self-attention for semantic segmentation,” *arXiv preprint arXiv:1907.12273*, 2019.

[48] A. Paszke *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[49] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[50] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[51] R. Wightman, *Pytorch image models*, https://github.com/rwightman/pytorch-image-models, 2019.

[52] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2017.

[53] Z. Cai and N. Vasconcelos, "Cascade r-cnn: Delving into high quality object detection," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[54] T. Xiao, Y. Liu, B. Zhou, Y. Jiang, and J. Sun, "Unified perceptual parsing for scene understanding," in *European Conference on Computer Vision (ECCV)*, 2018.

[55] B. Cheng, I. Misra, A. G. Schwing, A. Kirillov, and R. Girdhar, "Masked attention mask transformer for universal image segmentation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[56] T.-Y. Lin *et al.*, "Microsoft coco: Common objects in context," in *European Conference on Computer Vision (ECCV)*, 2014.

[57] B. Zhou, H. Zhao, X. Puig, S. Fidler, A. Barriuso, and A. Torralba, "Scene parsing through ade20k dataset," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[58] M. Cordts *et al.*, "The cityscapes dataset for semantic urban scene understanding," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[59] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo, "Cutmix: Regularization strategy to train strong classifiers with localizable features," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[60] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, "Mixup: Beyond empirical risk minimization," in *International Conference on Learning Representations (ICLR)*, 2018.

[61] E. D. Cubuk, B. Zoph, J. Shlens, and Q. V. Le, "Randaugment: Practical automated data augmentation with a reduced search space," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2020.

[62] Z. Zhong, L. Zheng, G. Kang, S. Li, and Y. Yang, "Random erasing data augmentation," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[63] K. Chen *et al.*, "Mmdetection: Open mmlab detection toolbox and benchmark," *arXiv:1906.07155*, 2019.

[64] M. Contributors, *MMSegmentation: Openmmlab semantic segmentation toolbox and benchmark*, https://github.com/open-mmlab/mmsegmentation, 2020.

[65] S. Walton, A. Hassani, X. Xu, Z. Wang, and H. Shi, "Efficient image generation with variadic attention heads," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2025.

[66] J. Cheng, X. Mei, and M. Liu, "Forecast-mae: Self-supervised pre-training for motion forecasting with masked autoencoders," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023.

[67] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," in *International Conference on Learning Representations (ICLR)*, 2023.

[68] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "Flashattention-3: Fast and accurate attention with asynchrony and low-precision," *arXiv preprint arXiv:2407.08608*, 2024.

[69] M. Zaheer *et al.*, "Big bird: Transformers for longer sequences," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[70] J. Bradbury *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.3.13, 2018.

[71] V. Thakkar *et al.*, *Cutlass*, 2023.

[72] B. Lefaudeux *et al.*, *Xformers: A modular and hackable transformer modelling library*, https://github.com/facebookresearch/xformers, 2022.

[73] P. Tillet, H.-T. Kung, and D. Cox, "Triton: An intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019.

[74] T. Kim and J. Nam, "All-in-one metrical and functional structure analysis with neighborhood attentions on demixed audio," in *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, 2023.

[75] W. Peebles and S. Xie, "Scalable diffusion models with transformers," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023.

[76] B. F. Labs, *Flux*, https://github.com/black-forest-labs/flux, 2024.

[77] P. Esser *et al.*, "Scaling rectified flow transformers for high-resolution image synthesis," in *International Conference on Machine Learning (ICML)*, 2024.

[78] R. Yu, S. Liu, Z. Tan, and X. Wang, “Ultra-resolution adaptation with ease,” *arXiv preprint arXiv:2503.16322*, 2025.