Fast Vision Mamba: Pooling Spatial Dimensions for Accelerated Processing

Supplementary Material

Table 9. Classification benchmarks on **ImageNet-1k** [14] dataset. All models are trained from scratch on Image size of 224 × 224. † denotes we extend the training of Vim to base-size model.

Model	#Params (M)	Top-1 (%)
Conv-Based		
ConvNeXt-T [37]	29M	82.1
ConvNeXt-S [37]	50M	83.1
ConvNeXt-B [37]	89M	83.8
Transformer-Based		
DeiT-T [56]	6M	72.2
DeiT-S [56]	22M	79.8
DeiT-B [56]	86M	81.8
Swin-T [36]	28M	81.3
Swin-S [36]	50M	83.2
Swin-B [36]	88M	83.5
Hybrid (Mamba + {2D convolution, Attention module})		
VMamba-T [35]	30M	82.6
VMamba-S [35]	50M	83.6
VMamba-B [35]	89M	83.9
GroupMamba-T [51]	23M	83.3
GroupMamba-S [51]	34M	83.9
GroupMamba-B [51]	57M	84.5
Eff.VMamba-T [44]	6M	76.5
Eff.VMamba-S [44]	11M	78.7
Eff.VMamba-B [44]	33M	81.8
MambaVision-T [20]	32M	82.3
MambaVision-S [20]	50M	83.3
MambaVision-B [20]	98M	84.2
Pure Mamba architecture		
Vim-T [67]	7M	76.1
Vim-S [67]	26M	80.5
Vim-B [67]	98M	81.9
PlainMamba-L1 [61]	7M	77.9
PlainMamba-L2 [61]	25M	81.6
PlainMamba-L3 [61]	50M	82.3
Mamba®-T [58]	9M	77.4
Mamba®-S [58]	28M	81.1
Mamba®-B [58]	99M	82.9
Vim-T-prune [63]	7M	75.1
Vim-S-prune [63]	26M	78.8
Vim-T (R-MeeTo) [53]	7M	75.3
Vim-S (R-MeeTo) [53]	26M	79.9
Vim-B (R-MeeTo) [53]	98M	81.3
FastVim-T	7M	75.4
FastVim-S	26M	81.1
FastVim-B	98M	82.6

In this supplementary material, details are provided on the following:

- Token Activation Map visualization (7)
- Effectiveness of Pooling under high-resolution scenarios (8)
- Additional ablations (9)
- Efficiency Analysis (additional) (10)
- Self-Supervised Learning: MAE (additional) (11)
- JUMP-CP (additional) (12)
- Additional Throughput analysis (13)
- Semantic Segmentation implementation details (14)
- Object Detection and Instance Segmentation implementation details (15)
- Kernel details (16)
- Model configurations (17)

7. Token Activation Map visualization

Visualization-based explaination of effectiveness of skip-connection. In Fig. 5, we visualize tokens activation before the 'z' gating (see Fig. 2, main paper), averaged over the last 10 layers. As discussed in Sec. 3.1 and Sec. 4.7, our proposed skip connection a) 'before pool – after repeat' preserves finer spatial details better than naive pooling of tokens b) 'after pool – before repeat' ($\mathbf{D}x_t$). Consequently, FastVim trained with naive pooling strategy (Fig. 5b) shows **repetitive artifacts** due to lack of skip-connection post repeat operation.

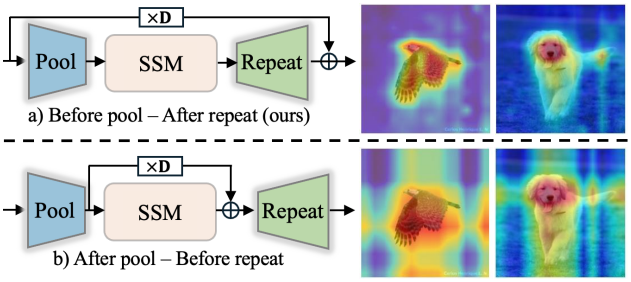


Figure 5. Tokens activation map in FastVim.

8. Effectiveness of Pooling under high-resolution scenarios

Here we provide further empirical validation that FastVim retains Vim's accuracy while significantly enhancing efficiency at high resolutions. Table 10 compares Vim and FastVim on two high-resolution datasets: BRIGHT ROI images (BRIGHT Challenge, IBM Research) (3-class) resized to 1536×1536, and UniToPatho '800' subset images [4] (6-class) resized from 1812×1812 to 1792×1792. Both Vim and FastVim were trained from scratch for 50 epochs using a batch size of 32 across four A6000 GPUs, employing

standard ImageNet settings. FastVim consistently matches Vim's performance with a 2.3× training speedup, demonstrating both (1) the effectiveness of our pooling method and (2) democratization of large-scale vision models.

Table 10. Classification benchmarks on high resolution datasets.

Dataset (resolution)	Vim-T (training time)	FastVim-T (training time)
BRIGHT (1536 × 1536)	61.4% (3h 44m)	61.2% (1h 35m)
UniToPatho (1792 × 1792)	47.5% (9h 24m)	47.2% (3h 54m)

9. Additional ablations

1. **Effect of using class token in FastVim.** In Table 11, we compare the performance of FastVim-S with a class token versus without a class token (default). We observe that having a class token improves performance but leads to slower reshape-transpose, pooling, and repeat operations to handle the middle class token. Since the goal of this study is to improve throughput while maintaining performance relative to the Vim baseline, we proceeded with all experiments without a class token. However, it is worth noting that even with a class token, our method is faster than Vim, according to our preliminary analysis. One future direction could be to mean pool only middle rows/columns for image-level representation instead of current mean pooling of all tokens for image representation.

Table 11. Effect of using class token in FastVim on ImageNet-1K.

Model	FastVim-S	w/ Class token
Top-1 (%)	81.1	81.3

2. **The performance impact of different input norm and post-ssm norm combinations.** In Table 12, we empirically demonstrate the performance of FastVim trained with different combinations of input normalization and post-SSM normalization. We found that using default RMS normalization as the input norm and LayerNorm as the post-SSM norm yields the best performance.

3. **Exploring pooling in ViT.** We now examine how our pooling method performs with the contextualization module in Transformers, namely Self-Attention. As seen in Table 13, our method, which applies alternating Pool_{col} and Pool_{row} pooling across blocks, performs significantly worse compared to the baseline ViT-S, which was trained using the default settings from DeiT[55]. We believe that pooling in Vim might be working better than in ViT due to: 1) local token interactions via Conv1D in vision mamba, 2) our design choice of token grid information preservation through skip connection ($\mathbf{D}x_t$) of SSM module, and 3) Vim scanpath's inherent inductive bias. To mimic 1) in ViT, we train a ViT with our pooling along with added conv1D module simiar to conv1D in Vim. Next we further added skip connection between pre-pool - self-attention - post-repeat operation akin to FastVim to mimic 2) in ViT. As observed in the Table 13, while conv1d doesn't benefit ViT variant much, the simple skip connection leads to significant improvement of 2%, aligning with our observation in FastVim in ablation Table 7. Probably, since the self-attention lacks inductive bias unlike the scanpath in SSM, there is still a performance drop of 4% between ViT and our best variant of ViT + pool + conv1d + skip connection. We note that while the failure of pooling approach in retaining performance in ViT as compared to Vim is interesting, it has opportunities for further exploration, and is out of scope of this work.

Table 12. Effect of using different normalization combination in FastVim-S on ImageNet-1K.

Model	RMS-LN	RMS-RMS	LN-LN
Top-1 (%)	81.1	80.7	80.9

Table 13. Effect of pooling in ViT on ImageNet-1k.

Model	ViT-S	+ Pool	+ conv1d	+ $\mathbf{D}x_t$
Top-1 (%)	80.1	73.9	74.0	76.1

10. Efficiency Analysis (additional)

Here, we demonstrate the reduction in FLOPs and the increase in throughput achieved by our FastVim compared to Vim. In Fig. 6, we compare the FLOPs requirements of FastVim, Vim, and ViT. At a lower resolution of 224, Vim demands the most operations, whereas ViT and FastVim have similar computational needs. As the resolution increases, ViT's computational requirements grow quadratically, while both Vim and FastVim scale linearly, with FastVim using up to 38% fewer FLOPs. Notably, within a Mamba block, all components scale linearly in terms of FLOPs with the number of tokens, leading to a quadratic increase with respect to resolution for vision tasks. FastVim optimizes computations exclusively in the SSM, reducing its scaling to linear with respect to resolution. As a result, the other layers remain unchanged and maintain the

same quadratic scaling as in Vim. Consequently, the overall FLOPs reduction in FastVim compared to Vim does not widen significantly with increasing resolution. The computational savings become more apparent at the SSM level, but this widening effect is muted at the block level, with FastVim-T using 35% fewer FLOPs at 224 resolution and 38.5% fewer FLOPs at 2048 resolution compared to Vim-T.

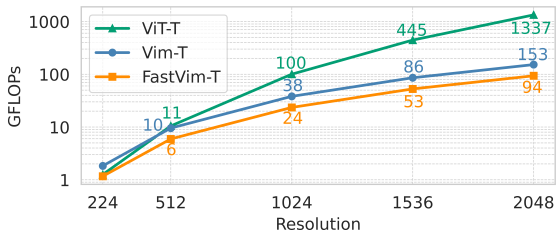


Figure 6. Comparison of FLOPs (G) for FastVim, Vim, and ViT across different resolutions.

11. Self-Supervised Learning: MAE (additional)

Implementation Details. We closely followed the pretraining (Table 14), fine-tuning (Table 15), and linear-probing (Table 16) settings from the Masked Autoencoders [22] codebase. All MAE pretraining is done for 1600 epochs in this study. A few key changes, particularly for fine-tuning and linear probing, are discussed below.

Table 14. MAE: Pre-training setting.

config	value
optimizer	AdamW [38]
base learning rate	1.5e-4
weight decay	0.05
optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.95$
batch size	4096
learning rate schedule	cosine decay [39]
training epochs	1600
warmup epochs	40
augmentation	RandomResizedCrop

Key Recipe Details.

1) Since Vim contains twice the number of layers compared to ViTs, we decreased the layer-wise learning rate decay every two blocks, instead of every block as in the MAE codebase for ViT fine-tuning, to ensure adequate fine-tuning of the initial layers. 2) We applied a scaling factor of 1 − mask ratio (75% masking by default) during fine-tuning and linear probing when pooling tokens before the SSM block. During pretraining, each row averaged

Table 15. MAE: End-to-end fine-tuning setting. Note that layer-wise lr decay is applied after every two blocks instead of one.

config	value
optimizer	AdamW
base learning rate	5e-4 (B), 1e-3 (L/H)
weight decay	0.05
optimizer momentum	β_1, β_2=0.9, 0.999
layer-wise lr decay [2]	0.65 (B), 0.75 (L/H)
batch size	1024
learning rate schedule	cosine decay
warmup epochs	5
training epochs	100 (B), 50 (L/H)
augmentation	RandAug (9, 0.5) [11]
label smoothing	0.1
mixup [65]	0.8
cutmix [62]	1.0
drop path [24]	**0.3**

Table 16. MAE: Linear probing setting.

config	value
optimizer	SGD
base learning rate	0.1
weight decay	0
optimizer momentum	0.9
batch size	4096
learning rate schedule	cosine decay
warmup epochs	10
training epochs	90
augmentation	RandomResizedCrop

25% of the tokens. In FastMaskVim, we sum the unmasked tokens and then divide by the number of columns for pooling instead of using mean pooling. To align this in fine-tuning and linear probing tasks, a scaling factor of 0.25 was necessary to achieve better performance.

Ablations.

1. **Divide by number of columns vs. mean pool in FastMaskVim.** In Table 17, we compare the performance of pre-training FastMaskVim using the default setting, where the sum of tokens in a row is divided by the number of columns, against mean pooling, where each row's sum is divided by the number of unmasked tokens present in the row. We observe in MAE pre-training that mean pooling performs slightly worse compared to the constant divide technique in corresponding downstream fine-tuning. However, exploring the comparison between mean pooling and constant divide in the context

of supervised training is left for future research.

Table 17. Comparison of constant divide vs. mean pool in pre-training FastMaskVim

FastMaskVim-B	Constant divide (default)	Mean Pool
Top-1 (%)	83.0	82.8

2. **Finetuning with alternate layer lr decay.** In Table 18, we compare the performance of fine-tuning pre-trained FastMaskVim using alternate layer learning rate decay instead of per-layer decay as in the MAE codebase. We observe that, since Vim typically contains twice the number of layers compared to ViTs with a similar parameter count, adjusting the decay logic to apply the learning rate decay every 2 blocks was necessary to ensure adequate fine-tuning of the early layers. With this simple adjustment, we were able to improve performance by a significant margin of 1%. This analysis motivates us to believe that with further recipe improvements, FastVim can match the performance of ViTs with MAE pre-training, where we currently observe a lag of 0.6-1% across Base to Huge model sizes (see Table 2).

Table 18. Comparison of alternate layer lr decay vs. per layer lr decay in finetuning

FastMaskVim-L	Alternate lr decay (default)	All layer lr decay
Top-1 (%)	84.9	83.9

3. **Finetuning with scaling factor.** In Table 19, we demonstrate the effect of using a scaling factor (0.25) in the fine-tune transfer of pre-trained FastMaskVim. Applying the scaling factor results in an improvement of 0.3% compared to the default mean pooling in fine-tuning without multiplying by the scaling factor. As shown in Fig. 7, when scaling is not used, the initial performance is much lower, although it catches up closely by the end of the training schedule.

Table 19. Effect of scaling factor in finetuning

FastMaskVim-L	w/ scaling (default)	w/o scaling
Top-1 (%)	84.9	84.6

4. **Linear probing with scaling factor.** In Table 20, we compare the linear probing performance of FastMaskVim with and without the scaling factor (0.25).

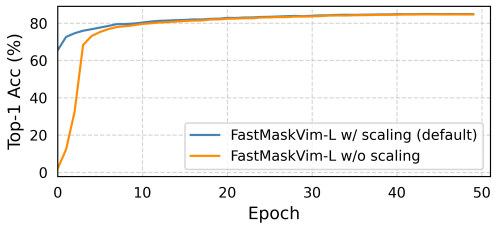


Figure 7. Effect of scaling factor in finetuning performance from MAE pretraining FastMaskVim-L on ImageNet-1k.

We observe a drastic difference in performance and note that without the scaling factor, the model was unable to train due to the significant difference between the pre-training and linear probing distributions of number of unmasked tokens. During pre-training, on average, each row had 25% of the number of columns (or number of rows when transposed in alternate layers) as unmasked tokens. Since we divided by the number of columns following the sum operation, the signal magnitude was in a lower range. In contrast, during linear probing, because all tokens are unmasked, we add the number of column tokens and divide by the number of columns, resulting in a very different signal range. We further compared the performance with pre-trained Vim's performance in linear probing and found that it performs considerably worse than FastVim. In MAE pre-training, random masking disrupts the sequential token positions, as demonstrated by Vim-prune [63]. In contrast, during linear probing, this issue does not exist, causing a shift in the neighborhood distribution and resulting in low linear probe performance. This issue does not occur in our FastVim since, in both pre-training and linear probing, the number of rows/columns remains the same, ensuring that the neighborhood remains consistent.

Table 20. Effect of scaling factor in linear probing

Method	Vim-L	FastMaskVim-L w/ scaling	w/o scaling
Top-1 (%)	45.6	60.2	0.02

12. JUMP-CP (additional)

Implementation details.

We followed the implementation details primarily from ChannelViT [3]. Specifically, we used a learning rate of 1×10^{-3}, a batch size of 256, and trained the model for 100 epochs, including 10 warmup epochs. We set the drop path rate to 0.05 and did not use EMA. All details and configuration files will be made available in the code.

Ablations.

1. **ChannelVim-S: Effect of Spatial-First vs. Channel-First with and without sorted HCS.** In Table 21, we demonstrate the key configurations required to extend ChannelViT [3] to the Mamba-based encoder, termed ChannelVim. As explained in detail in Sec. 3.3, due to the sequential processing in Mamba, the order of tokens matters. We found that the Channel-First method performs significantly better than Spatial-First. Whereas, the effect of sorting the output of hierarchical channel sampling (HCS) is opposite: it might be acting as an augmentation in the Spatial-First approach due to the order covering channel-by-channel, while it might be causing disruption in the neighborhood in the Channel-First approach since every next token in the sequence is another channel. Randomly shuffling the channel order (no sort) makes it difficult for learning.

Table 21. ChannelVim-S: Effect of Spatial-First vs. Channel-First with and without sorted HCS on 160-way perturbed gene prediction on JUMP-CP dataset. All methods use hierarchical channel sampling [3] for training, and testing is done using all 8 channels. Each cell image is of resolution $224 \times 224 \times 8$.

Method	HCS	Top-1
Channel-First	Sorted	73.5
	Unsorted	69.4
Spatial-First	Sorted	65.9
	Unsorted	67.9

2. **FastChannelVim-S: Effect of different pooling methods (mean, max, and attention pooling):** In this study, we use average pooling of tokens to compress the tokens before the SSM scan. We then explore the effect of different pooling methods, such as max pooling [46] and attention pooling [26], as detailed in Table 22 on the JUMP-CP dataset. For attention pooling, we added a simple linear layer before each pooling layer to project each token to a size of one. This is followed by a SoftMax operation across tokens in the row, which is multiplied by a learned attention value and then summed across the row. We found that at a patch size of 16, all methods perform comparably. However, at a patch size of 8, max pooling and attention pooling methods start to perform better, likely due to the increased number of tokens in a row, allowing them to capture the most discriminative signals more effectively than mean pooling. Based on the accuracy-throughput trade-off, max pooling emerges as the best choice on the JUMP-CP dataset, as it is as fast as mean pooling while performing very close to attention pooling. Exploring effect of these pooling operation in natural imaging is left for future studies.

Table 22. FastChannelVim-S: Effect of different pooling methods (mean, max, and attention pooling) on 160-way perturbed gene prediction on JUMP-CP dataset. All methods use hierarchical channel sampling [3] for training, and testing is done using all 8 channels. Each cell image is of resolution $224 \times 224 \times 8$.

Pooling	patch-size	Top-1
Mean	16	73.6
Max	16	72.9
Att	16	73.1
Mean	8	83.1
Max	8	85.0
Att	8	85.8

3. **FastChannelVim-S: Effect of Pooling across 2 dimensions:** So far, we have explored pooling along only one spatial dimension, either across rows or columns. Now, we preliminarily explore pooling along two dimensions, which is particularly applicable in 3-dimensional datasets. When performing channel-wise tokenization, we obtain a 3D token grid. We experiment with the following pooling combinations in sequence every three blocks: column-channel pooling - row-channel pooling - row-column pooling - repeat. This approach provides much stronger compression, reducing the 3D token grid to a 1D token grid for the SSM scan.
In Table 23, we demonstrate that at a patch size of 16 (token grid 14x14x8), both mean pooling and max pooling with 2D pooling work well and are on par with 1D pooling. In contrast, at a patch size of 8 (token grid 28x28x8), given the significantly larger number of tokens to pool (28x28 in row-column, 28x8 in row-channel, 28x8 in column-channel pooling blocks), mean pooling does not perform well. However, when we use max pooling, it performs much better, achieving results on par with ChannelVim with a patch size of 8 (see Table 3). Thus, even after pooling a much larger number of tokens, our method, FastChannelVim, still performs well with max pooling. This has implications in making the video models even faster [27].

13. Additional Throughput analysis

1. **Throughput across model sizes.** In Fig. 9, we display the throughput of Vim and FastVim across Tiny, Small,

Table 23. FastChannelVim-S: Effect of Pooling across 2 dimensions on 160-way perturbed gene prediction on JUMP-CP dataset. All methods use hierarchical channel sampling [3] for training, and testing is done using all 8 channels. Each cell image is of resolution $224 \times 224 \times 8$.

Pooling	patch-size	Top-1
Mean - 1D	16	73.6
Max - 1D	16	72.9
Mean - 2D	16	74.3
Max - 2D	16	73.5
Mean - 1D	8	83.1
Max - 1D	8	85.0
Mean - 2D	8	78.4
Max - 2D	8	84.0

and Base-sized models with a batch size of 16. Across all model sizes, our method consistently provides a speedup in throughput compared to the Vim baseline.

2. **Throughput on per-channel modeling tasks.** In Table 24, we demonstrate the throughput improvement in FastChannelVim compared to ChannelVim. With a longer token sequence (patch size 8), FastChannelVim delivers a speedup of 62.3% over ChannelVim without any drop in accuracy (see Table 3).

Table 24. Comparison of inference throughput analysis between ChannelVim and FastChannelVim across patch sizes 16 and 8. Each cell image has a resolution of $224 \times 224 \times 8$, and the batch size is set to 8.

Method	Token-grid	Throughput (it/s)
ChannelVim-S/16	$14^2 \times 8$	234
FastChannelVim-S/16	$14^2 \times 8$	318
ChannelVim-S/8	$28^2 \times 8$	61
FastChannelVim-S/8	$28^2 \times 8$	99

3. **Dissecting SSM processing time.** Here, we calculate the processing time for Forward SSM + Backward SSM in only one block (see Fig. 2) for Vim-T versus FastVim-T. The SSM time include the parameter projection $(\mathbf{B}, \mathbf{C}, \mathbf{\Delta})$ for selective scan, the SSM **scan** time, and the skip connection $(\mathbf{D}x_t)$. Note that since the Mamba implementation enables the skip connection inside the CUDA kernel for faster processing, for Vim, we put the skip connection inside the kernel. However, for FastVim, we can't input the skip connection matrix $(\mathbf{D})$ to the kernel since we need to first perform the repeat operation and then add it with the skip connection, which takes place outside the CUDA kernel in FastVim. This results in significant overhead for FastVim, but since our SSM scan and parameter projection is lot more computationally cheaper due to compressed input after pool, FastVim still gives significant speedup over Vim for the Forward SSM + Backward SSM in a block.

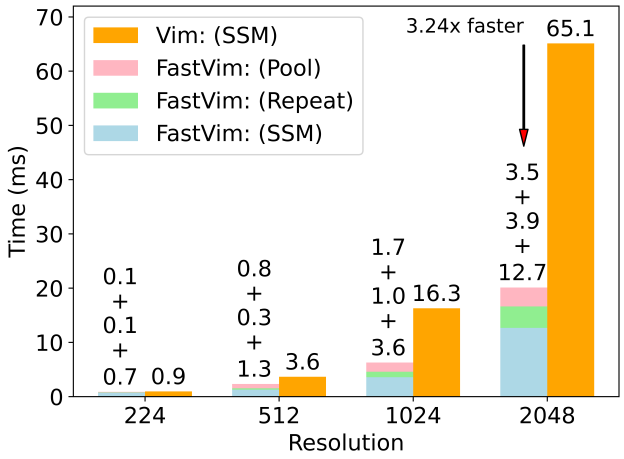


Figure 8. Comparison of inference processing time for only the Forward and Backward SSM layer in one block of Vim-T versus FastVim-T. For FastVim, we calculate Pool + Repeat + SSM time. The annotations indicate SSM time for Vim; for FastVim, the upper value indicates the time for Pool, the middle value indicates the time for Repeat, and the lower value indicates the time for SSM.

In Fig. 8, for Vim-T, we calculate the time for SSM, whereas for FastVim-T, since we perform pooling and repeat operations, we measure the time for pooling and SSM and repeat. We can observe that the time taken in Vim scales quadratically with increasing resolution (approximately more than $4\times$ increase in time for a $2\times$ resolution increase), whereas ours scales sub-quadratically (approximately less than $3\times$ increase in time for a $2\times$ resolution increase). Thus at higher resolution (2048), FastVim (Pool + SSM + Repeat) is $3.24\times$ faster than Vim (SSM). It can also be observed that as resolution increases, the repeat operation becomes increasingly expensive, taking almost 25% of the time for FastVim (Pool + SSM + Repeat) compared to only about 8% at resolution 224. Preliminary optimizations for this **redundant** repeat operation are discussed in Sec. 16. Note that the time is recorded in milliseconds, and is for 1 batch with a batch size of 128. We use enough warmup runs to make sure the reported times are correct in practice. Residual (not the skip connection one) is omitted in these calculations for Vim

Table 25. Dissecting SSM processing time (in milliseconds) at inference for only the Forward and Backward SSM layer in one block of Vim-T versus FastVim-T. For Vim, SSM includes parameter projection + SSM scan (with skip connection in CUDA kernel); for FastVim, pool + parameter projection + SSM scan + repeat + skip connection. Note that since skip connection is added in CUDA kernel for SSM scan for Vim, we don't report the time for skip-conn. separately for Vim.

Operations	Vim (224)	FastVim (224)	Vim (512)	FastVim (512)	Vim (1024)	FastVim (1024)	Vim (2048)	FastVim (2048)
SSM scan	0.79	0.44	3.20	0.41	14.52	0.42	58.20	0.30
Parameter proj.	0.17	0.07	0.44	0.08	1.80	0.11	6.90	0.20
Pool	-	0.10	-	0.80	-	1.70	-	3.46
Repeat	-	0.06	-	0.26	-	1.00	-	3.90
Skip conn.	-	0.17	-	0.78	-	3.10	-	12.2
Total	0.96	0.84	3.64	2.33	16.32	6.33	65.10	20.06

and FastVim, and transpose operation is omitted as well in FastVim.

In Table 25, we mention the time taken by each component in more details for Forward SSM + Backward SSM for Vim and FastVim. For Vim, we report the parameter projection time and the SSM scan time. Note that since the Mamba kernel enables the skip connection in its CUDA kernel, for Vim, we do not separately report the skip connection time as it already becomes negligible in the Mamba kernel implementation. However, we observe that when the skip connection is not included inside the kernel, it takes significantly more time, similar to the skip connection time values mentioned for FastVim (in Table 25). For FastVim, we measure the time for pooling, parameter projection, SSM scan, repeat operation, and skip connection, since it can't be added in the CUDA kernel due to the required repeat operation beforehand. It can be observed that even with a large image size of 2048×2048, FastVim's SSM scan time and parameter projection time are lower than Vim's SSM scan time and parameter projection time at a much smaller 224 resolution. This is because, following tokenization and pooling, we have just **128** tokens for a 2048 resolution image, whereas Vim has 196 tokens for a 224 resolution image during the SSM scan and parameter projection. We would like to note that at higher resolutions, for FastVim, the pooling, repeat, and skip-connection operations take the majority of the time, whereas the SSM scan and parameter projection take significantly less time. These operations can be fused in CUDA kernel in future studies to achieve even more speedup.

4. **Throughput analysis in VMamba.** In Fig. 10, we compare the throughput of VMamba and FastVMamba for the Tiny, Small, and Base-sized models with a batch size of 16. Across all model sizes, our method consistently provides a significant speedup in throughput compared to the VMamba baseline at higher resolutions. For the Tiny-sized model, we observe an improvement of up to 50% at higher resolutions. Note that VMamba [35] uses only one state in the SSM scan for VMamba-T, and therefore the percentage of time spent on the SSM scan relative to other layers in VMamba-T is lower. In contrast, the Small and Base-sized models use two states for the SSM scan, and since our method makes the SSM scan an order of magnitude faster at higher resolutions (see Table 25), this translates to an overall throughput improvement of up to 140% for the Small and Base-sized VMamba variants.

14. Semantic Segmentation implementation details

In line with Vim [67] and LocalVim [25], we used a batch size of 16 and an input size of 512x512. We employed the AdamW optimizer with a weight decay of 0.01. A Poly learning rate schedule was used, decaying over 160K iterations, with an initial learning rate of 6×10^{-5}. For Tiny and Small models, we used drop path rate of 0.05, and for Base, we used 0.4. For evaluation, we used sliding window prediction with crop size of 512 and stride of 341. We utilized the code provided by LocalVim [25], which is based on MMSegmentation [10]. In Table 26, we compare FastVim with other baselines, including hierarchical architectures such as Swin [36] and VMamba [35]. Notably, FastVim is most appropriately compared with its base architecture, Vim, since we use the same architecture and accelerate it with our pooling.

15. Object Detection and Instance Segmentation implementation details

Following the code from LocalVim [25], we utilize the neck architecture from ViTDet and train Cascade Mask R-CNN as the detector. We employed the AdamW optimizer with a weight decay of 0.05, with a total batch size of 64. Initial learning rate is set to 1×10^{-4} and incorporates lin-

Table 26. Semantic segmentation benchmarks on **ADE20K** [66] dataset. UperNet [59] framework is used for all comparison backbones, with a crop size of 512 × 512. MLN: multi-level neck.

Backbone	mIoU
DeiT-T	39.2
DeiT-S + MLN	43.8
DeiT-B + MLN	45.5
Vim-T	41.0
Vim-S	44.9
FastVim-T	41.8
FastVim-S	44.9
FastVim-B	47.8
LocalVim-T	43.4
LocalVim-S	46.4
PlainMamba-L1	44.1
PlainMamba-L2	46.8
PlainMamba-L3	49.1
Swin-T	44.4
Swin-S	47.6
VMamba-T	47.3
VMamba-S	49.5

ear decay in the learning rate. We used drop path rate of 0.1 for Tiny and Small sized models, and 0.4 for Base sized model. In Table 27, we compare FastVim with other baselines. We only consider non-hierarchical architectures to maintain consistency with the ViTDet neck architecture and the Cascade Mask-RCNN detector for detection task. FastVim is most appropriately compared with its base architecture, Vim, as we use the same architecture and accelerate it with our pooling.

Table 27. Object detection and instance segmentation benchmarks on COCO dataset using Cascaded Mask R-CNN [21] framework. *detection transfer conducted using provided Vim-S (GitHub).

Backbone	AP^{box}	AP^{box}_{50}	AP^{box}_{75}	AP^{mask}	AP^{mask}_{50}	AP^{mask}_{75}
DeiT-T	44.4	63.0	47.8	38.1	59.9	40.5
Vim-T	45.7	63.9	49.6	39.2	60.9	41.7
Vim-S*	47.1	65.8	50.7	40.6	62.9	43.5
FastVim-T	45.1	63.7	48.5	39.0	60.8	41.6
FastVim-S	48.4	67.2	52.2	41.8	64.3	44.7
FastVim-B	50.0	68.7	54.2	43.2	66.0	46.6
LocalVim-T	45.3	66.2	49.1	39.9	63.0	42.5
PlainMamba-Adapter-L1	44.1	64.8	47.9	39.1	61.6	41.9
PlainMamba-Adapter-L2	46.0	66.9	50.1	40.6	63.8	43.96
PlainMamba-Adapter-L3	46.8	68.0	51.1	41.2	64.7	43.9

16. Kernel details

In FastVim (refer to Fig. 2), we apply mean pooling to the tokens before performing the SSM scan. Consequently, this operation must be repeated before integrating with the skip connection ($\mathbf{D}$ in Eq. 3). When implementing this in PyTorch, we utilize the `repeat_interleave` function to duplicate the output of the SSM scan prior to adding it with $\mathbf{D}x_t$. However this operation becomes computationally expensive and redundant as demonstrated in Table 25. Instead, we preliminarily explored modifying this repetition and moving the skip connection in the new CUDA kernel.

Table 28. Comparison of inference throughput analysis with our kernel versus default Mamba kernel on a H100 gpu. Image resolution is 224, and batch size is set to 128.

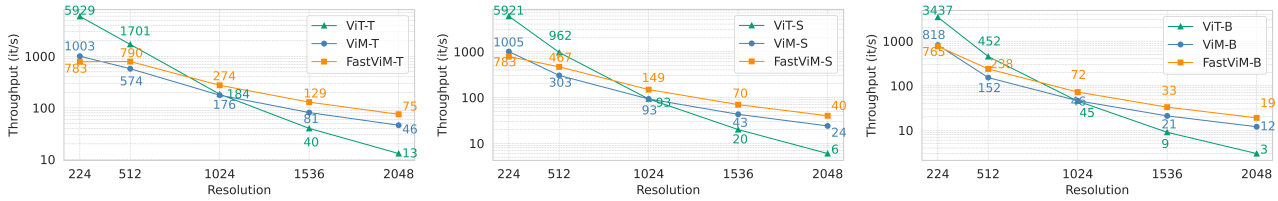
FastVim-T	Throughput (it/s)
with Mamba kernel	3680
with our kernel	4009

Specifically, given a input flattened token sequence (x_t) with a length $L = h \times w$, the compressed output (pool across column) will have a length of h. Our objective is to have the i^{th} element of this compressed output directly added to the token sequence spanning $i \cdot w$ to $(i+1) \cdot w$ within the skip connection $\mathbf{D}x_t$. This technique can reduce the time spent on redundant repetition in a naive PyTorch implementation, translating to practical speedup. In Table 28, we demonstrate the increased throughput for FastVim at a resolution of 224 with our kernel compared to default Mamba kernel that we used in this study. With the current optimizations, we observe an improvement in speed for 224-sized images; however, there is a decrease in speed at higher resolutions. This indicates the need for further refinement to optimize our kernel. Therefore, we also plan to release our kernel implementation to the open-source community so that others can build upon it.

17. Model configurations

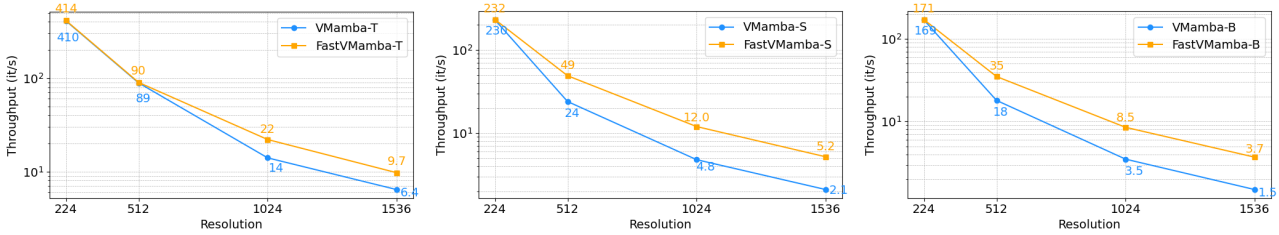
Table 29. Model configurations for FastVim

Model	Layers	Embedding dim.
Tiny	24	192
Small	24	384
Base	24	768
Large	48	1024
Huge	64	1280



(a) Tiny (b) Small (c) Base

Figure 9. Comparison of Inference Throughput (it/s) for FastVim, Vim, and ViT across different resolutions and model sizes. Tested on H100 GPU with batch size of 16.



(a) Tiny (b) Small (c) Base

Figure 10. Comparison of Inference Throughput (it/s) for FastVMamba and VMamba across different resolutions and model sizes. Tested on RTX 8000 GPU with batch size of 16.

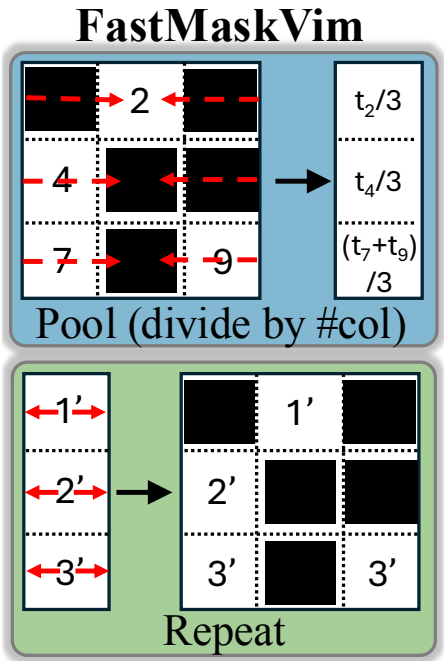


Figure 11. Illustration of pooling and repeat operations in FastMaskVim. Instead of naive mean pooling of tokens in a row, we add the tokens and then divide it by number of columns in the token grid. Similarly when alternatively pooling tokens in a column.

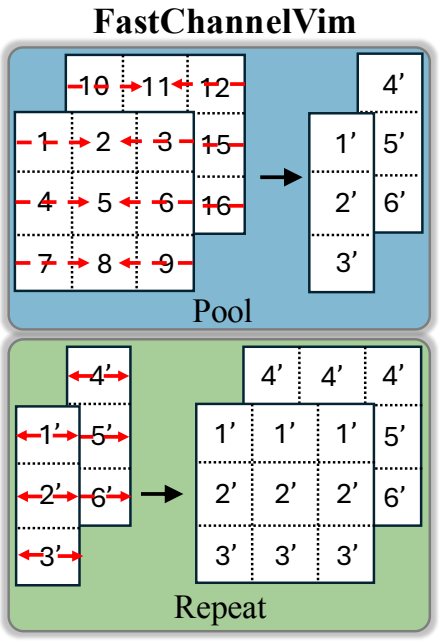


Figure 12. Illustration of pooling and repeat operations in FastChannelVim. These two operations are performed independently for each channel in per-channel tokenization paradigm.

References

[1] Kumar Ashutosh, Rohit Girdhar, Lorenzo Torresani, and Kristen Grauman. Hiervl: Learning hierarchical video-

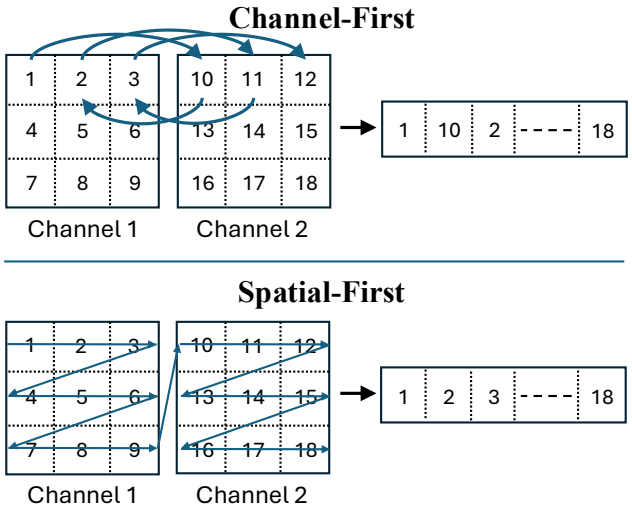


Figure 13. Illustration of flattened scanpath options available following per-channel tokenization in ChannelVim due to sequential nature of Mamba.

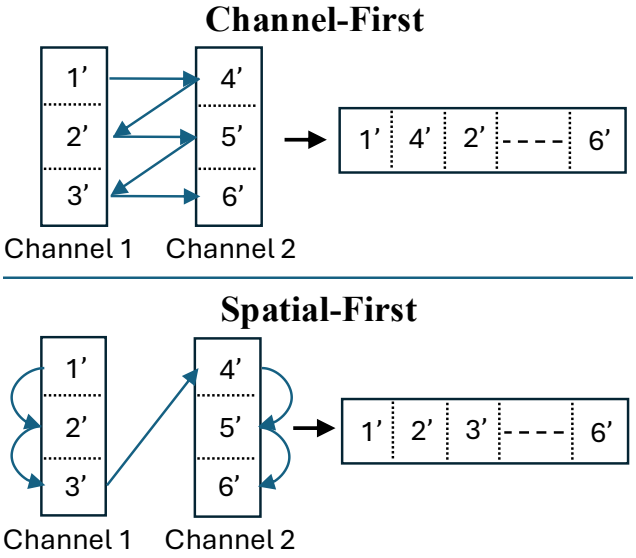


Figure 14. Illustration of flattened scanpath options available following per-channel tokenization in FastChannelVim due to sequential nature of Mamba.

language embeddings. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 23066–23078, 2023. 8

[2] Hangbo Bao, Li Dong, Songhao Piao, and Furu Wei. Beit: Bert pre-training of image transformers. *arXiv preprint arXiv:2106.08254*, 2021. 11

[3] Yujia Bao, Srinivasan Sivanandan, and Theofanis Karaletsos. Channel vision transformers: An image is worth c x 16 x 16 words. *arXiv preprint arXiv:2309.16108*, 2023. 2, 3, 4, 6, 7, 12, 13, 14

[4] Carlo Alberto Barbano, Daniele Perlo, Enzo Tartaglione, Attilio Fiandrotti, Luca Bertero, Paola Cassoni, and Marco Grangetto. Unitopatho, a labeled histopathological dataset for colorectal polyps classification and adenoma dysplasia grading. In *2021 IEEE International Conference on Image Processing (ICIP)*, pages 76–80. IEEE, 2021. 9

[5] Varun Belagali, Lei Zhou, Xiang Li, and Dimitris Samaras. Hypermae: Modulating implicit neural representations for mae training. 2023. 4

[6] Guy E Blelloch. Prefix sums and their applications. 1990. 1

[7] Daniel Bolya, Cheng-Yang Fu, Xiaoliang Dai, Peizhao Zhang, Christoph Feichtenhofer, and Judy Hoffman. Token merging: Your vit but faster. *arXiv preprint arXiv:2210.09461*, 2022. 8

[8] Srinivas Niranj Chandrasekaran, Jeanelle Ackerman, Eric Alix, D Michael Ando, John Arevalo, Melissa Bennion, Nicolas Boisseau, Adriana Borowa, Justin D Boyd, Laurent Brino, et al. Jump cell painting dataset: morphological impact of 136,000 chemical and genetic perturbations. *BioRxiv*, pages 2023–03, 2023. 2, 6

[9] Richard J Chen, Tong Ding, Ming Y Lu, Drew FK Williamson, Guillaume Jaume, Andrew H Song, Bowen Chen, Andrew Zhang, Daniel Shao, Muhammad Shaban, et al. Towards a general-purpose foundation model for computational pathology. *Nature Medicine*, 30(3):850–862, 2024. 2

[10] MMSegmentation Contributors. Openmmlab semantic segmentation toolbox and benchmark, 2020. 15

[11] Ekin D Cubuk, Barret Zoph, Jonathon Shlens, and Quoc 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 workshops*, pages 702–703, 2020. 11

[12] Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060*, 2024. 8

[13] Srijan Das, Rui Dai, Di Yang, and Francois Bremond. Vpn++: Rethinking video-pose embeddings for understanding activities of daily living. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(12):9703–9717, 2021. 8

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. 2, 5, 9

[15] Alexey Dosovitskiy. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 1

[16] Alexandros Graikos, Srikar Yellapragada, Minh-Quan Le, Saarthak Kapse, Prateek Prasanna, Joel Saltz, and Dimitris Samaras. Learned representation-guided diffusion models for large-image generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8532–8542, 2024. 8

[17] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023. 1, 2

[18] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021. 1, 2

[19] Albert Gu, Isys Johnson, Karan Goel, Khaled Saab, Tri Dao, Atri Rudra, and Christopher Ré. Combining recurrent, convolutional, and continuous-time models with linear state space layers. *Advances in neural information processing systems*, 34:572–585, 2021. 2

[20] Ali Hatamizadeh and Jan Kautz. Mambavision: A hybrid mamba-transformer vision backbone. *arXiv preprint arXiv:2407.08083*, 2024. 5, 8, 9

[21] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017. 7, 16

[22] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 16000–16009, 2022. 2, 4, 6, 11

[23] Le Hou, Richard Yuanzhe Pang, Tianyi Zhou, Yuexin Wu, Xinying Song, Xiaodan Song, and Denny Zhou. Token dropping for efficient bert pretraining. *arXiv preprint arXiv:2203.13240*, 2022. 8

[24] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*, pages 646–661. Springer, 2016. 11

[25] Tao Huang, Xiaohuan Pei, Shan You, Fei Wang, Chen Qian, and Chang Xu. Localmamba: Visual state space model with windowed selective scan. *arXiv preprint arXiv:2403.09338*, 2024. 15

[26] Maximilian Ilse, Jakub Tomczak, and Max Welling. Attention-based deep multiple instance learning. In *International conference on machine learning*, pages 2127–2136. PMLR, 2018. 4, 13

[27] Kumara Kahatapitiya, Adil Karjauv, Davide Abati, Fatih Porikli, Yuki M Asano, and Amirhossein Habibian. Object-centric diffusion for efficient video editing. In *European Conference on Computer Vision*, pages 91–108. Springer, 2025. 13

[28] Saarthak Kapse, Pushpak Pati, Srijan Das, Jingwei Zhang, Chao Chen, Maria Vakalopoulou, Joel Saltz, Dimitris Samaras, Rajarsi R Gupta, and Prateek Prasanna. Si-mil: Taming deep mil for self-interpretability in gigapixel histopathology. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11226–11237, 2024. 4

[29] Kian Kenyon-Dean, Zitong Jerry Wang, John Urbanik, Konstantin Donhauser, Jason Hartford, Saber Saberian, Nil Sahin, Ihab Bendidi, Safiye Celik, Marta Fay, et al. Vitally consistent: Scaling biological representation learning for cell microscopy. *arXiv preprint arXiv:2411.02572*, 2024. 7

[30] Kunchang Li, Xinhao Li, Yi Wang, Yinan He, Yali Wang, Limin Wang, and Yu Qiao. Videomamba: State space model for efficient video understanding. In *European Conference on Computer Vision*, pages 237–255. Springer, 2025. 8

[31] Yanghao Li, Hanzi Mao, Ross Girshick, and Kaiming He. Exploring plain vision transformer backbones for object detection. In *European conference on computer vision*, pages 280–296. Springer, 2022. 7

[32] Youwei Liang, Chongjian Ge, Zhan Tong, Yibing Song, Jue Wang, and Pengtao Xie. Not all patches are what you need: Expediting vision transformers via token reorganizations. *arXiv preprint arXiv:2202.07800*, 2022. 8

[33] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 740–755. Springer, 2014. 7

[34] Yunze Liu and Li Yi. Map: Unleashing hybrid mamba-transformer vision backbone's potential with masked autoregressive pretraining. *arXiv preprint arXiv:2410.00871*, 2024. 6

[35] Yue Liu, Yunjie Tian, Yuzhong Zhao, Hongtian Yu, Lingxi Xie, Yaowei Wang, Qixiang Ye, and Yunfan Liu. Vmamba: Visual state space model, 2024. 1, 5, 8, 9, 15

[36] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 10012–10022, 2021. 1, 8, 9, 15

[37] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11976–11986, 2022. 9

[38] I Loshchilov. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017. 11

[39] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016. 11

[40] Ali Nasiri-Sarvi, Vincent Quoc-Huy Trinh, Hassan Rivaz, and Mahdi S Hosseini. Vim4path: Self-supervised vision mamba for histopathology images. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6894–6903, 2024. 8

[41] Duy-Kien Nguyen, Mahmoud Assran, Unnat Jain, Martin R Oswald, Cees GM Snoek, and Xinlei Chen. An image is worth more than 16x16 patches: Exploring transformers on individual pixels. *arXiv preprint arXiv:2406.09415*, 2024. 7

[42] Maxime Oquab, Timothée Darcet, Théo Moutakanni, Huy Vo, Marc Szafraniec, Vasil Khalidov, Pierre Fernandez, Daniel Haziza, Francisco Massa, Alaaeldin El-Nouby, et al. Dinov2: Learning robust visual features without supervision. *arXiv preprint arXiv:2304.07193*, 2023. 2, 4

[43] Badri Narayana Patro and Vijay Srinivas Agneeswaran. Mamba-360: Survey of state space models as transformer alternative for long sequence modelling: Methods, applications, and challenges. *arXiv preprint arXiv:2404.16112*, 2024. 8

[44] Xiaohuan Pei, Tao Huang, and Chang Xu. Efficientvmamba: Atrous selective scan for light weight visual mamba. *arXiv preprint arXiv:2403.09977*, 2024. 8, 9

[45] Chau Pham and Bryan A Plummer. Enhancing feature diversity boosts channel-adaptive vision transformers. *arXiv preprint arXiv:2405.16419*, 2024. 7

[46] Kanchana Ranasinghe, Brandon McKinzie, Sachin Ravi, Yinfei Yang, Alexander Toshev, and Jonathon Shlens. Perceptual grouping in contrastive vision-language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5571–5584, 2023. 13

[47] Yongming Rao, Wenliang Zhao, Benlin Liu, Jiwen Lu, Jie Zhou, and Cho-Jui Hsieh. Dynamicvit: Efficient vision transformers with dynamic token sparsification. *Advances in neural information processing systems*, 34:13937–13949, 2021. 8

[48] Sucheng Ren, Xianhang Li, Haoqin Tu, Feng Wang, Fangxun Shu, Lei Zhang, Jieru Mei, Linjie Yang, Peng Wang, Heng Wang, et al. Autoregressive pretraining with mamba in vision. *arXiv preprint arXiv:2406.07537*, 2024. 6

[49] Cedric Renggli, André Susano Pinto, Neil Houlsby, Basil Mustafa, Joan Puigcerver, and Carlos Riquelme. Learning to merge tokens in vision transformers. *arXiv preprint arXiv:2202.12015*, 2022. 8

[50] Michael S Ryoo, AJ Piergiovanni, Anurag Arnab, Mostafa Dehghani, and Anelia Angelova. Tokenlearner: What can 8 learned tokens do for images and videos? *arXiv preprint arXiv:2106.11297*, 2021. 8

[51] Abdelrahman Shaker, Syed Talal Wasim, Salman Khan, Juergen Gall, and Fahad Shahbaz Khan. Groupmamba: Parameter-efficient and accurate group visual state space model. *arXiv preprint arXiv:2407.13772*, 2024. 5, 8, 9

[52] Hui Shen, Zhongwei Wan, Xin Wang, and Mi Zhang. Famba-v: Fast vision mamba with cross-layer token fusion. *arXiv preprint arXiv:2409.09808*, 2024. 8

[53] Mingjia Shi, Yuhao Zhou, Ruiji Yu, Zekai Li, Zhiyuan Liang, Xuanlei Zhao, Xiaojiang Peng, Tanmay Rajpurohit, Shanmukha Ramakrishna Vedantam, Wangbo Zhao, et al. Faster vision mamba is rebuilt in minutes via merged token retraining. *arXiv preprint arXiv:2412.12496*, 2024. 5, 9

[54] Jimmy TH Smith, Andrew Warrington, and Scott W Linderman. Simplified state space layers for sequence modeling. *arXiv preprint arXiv:2208.04933*, 2022. 1, 3

[55] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *International conference on machine learning*, pages 10347–10357. PMLR, 2021. 10

[56] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *International conference on machine learning*, pages 10347–10357. PMLR, 2021. 9

[57] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017. 1

[58] Feng Wang, Jiahao Wang, Sucheng Ren, Guoyizhe Wei, Jieru Mei, Wei Shao, Yuyin Zhou, Alan Yuille, and Cihang Xie. Mamba-r: Vision mamba also needs registers. *arXiv preprint arXiv:2405.14858*, 2024. 9

[59] Tete Xiao, Yingcheng Liu, Bolei Zhou, Yuning Jiang, and Jian Sun. Unified perceptual parsing for scene understanding. In *Proceedings of the European conference on computer vision (ECCV)*, pages 418–434, 2018. 7, 16

[60] Hanwen Xu, Naoto Usuyama, Jaspreet Bagga, Sheng Zhang, Rajesh Rao, Tristan Naumann, Cliff Wong, Zelalem Gero, Javier González, Yu Gu, et al. A whole-slide foundation model for digital pathology from real-world data. *Nature*, pages 1–8, 2024. 8

[61] Chenhongyi Yang, Zehui Chen, Miguel Espinosa, Linus Ericsson, Zhenyu Wang, Jiaming Liu, and Elliot J Crowley. Plainmamba: Improving non-hierarchical mamba in visual recognition. *arXiv preprint arXiv:2403.17695*, 2024. 9

[62] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6023–6032, 2019. 11

[63] Zheng Zhan, Zhenglun Kong, Yifan Gong, Yushu Wu, Zichong Meng, Hangyu Zheng, Xuan Shen, Stratis Ioannidis, Wei Niu, Pu Zhao, et al. Exploring token pruning in vision state space models. *arXiv preprint arXiv:2409.18962*, 2024. 5, 8, 9, 12

[64] Zheng Zhan, Yushu Wu, Zhenglun Kong, Changdi Yang, Yifan Gong, Xuan Shen, Xue Lin, Pu Zhao, and Yanzhi Wang. Rethinking token reduction for state space models. *arXiv preprint arXiv:2410.14725*, 2024. 8

[65] Hongyi Zhang. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017. 11

[66] Bolei Zhou, Hang Zhao, Xavier Puig, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Scene parsing through ade20k dataset. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 633–641, 2017. 7, 16

[67] Lianghui Zhu, Bencheng Liao, Qian Zhang, Xinlong Wang, Wenyu Liu, and Xinggang Wang. Vision mamba: Efficient visual representation learning with bidirectional state space model. *arXiv preprint arXiv:2401.09417*, 2024. 1, 3, 5, 7, 9, 15