SAS: Structured Activation Sparsification
Supplemental Material

A Detail in Trajectory Length Analysis

Network design for the *Trajectory Length* analysis (section 3) is illustrated infig. A1. We evaluate the length by randomly initializing the weight 100 times and reporting their average. We also compose the SWS network having the same sparsity, which consumes approximately the same `mult` count. The SWS network uses `ReLU` activation.

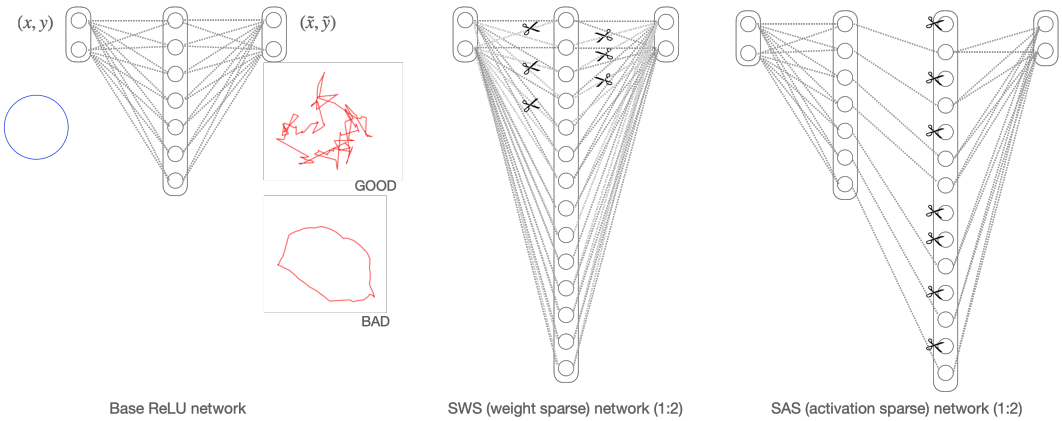

Figure A1: **Network design used for the *Trajectory Length Analysis*** (section 3). Relative output length with respect to the input circle length is an indicator of the network's expressive power; a longer length (complex trajectory) indicates more expressive power.

B Detail in SWS Network

In this paper, we propose SAS to improve the network capacity or accuracy without increasing the actual `mult` count. Therefore, we compare the SWS in the same scenario (keeping the `mult` count constant while increasing the sparsity). Specifically, we consider a base layer consisting of `matmul` between activation $\mathbf{X}$ and weight $\mathbf{W}$, where $\mathbf{X} \in \mathbb{R}^{\bar{C}_i \times HW}$ and $\mathbf{W} \in \mathbb{R}^{C_o \times \bar{C}_i}$ (eq. (1)). Using the proposed SAS, one could increase the network width for M times while maintaining the same `mult` count as the base layer by utilizing the 1:M sparsity pattern. The sparsified matrix shape is, $\mathbf{X} \in \mathbb{R}^{M\bar{C}_i \times HW}$ and $\mathbf{W} \in \mathbb{R}^{C_o \times M\bar{C}_i}$ (eq. (3)). Note that the SAS does not change the output channel dimension C_o.

On the other hand, in the case of SWS, if one increases the network width for the M times and uses the 1:M sparse pattern on weight, the `mult` count of the resultant network increases by about $M \times M/M = M$ because both input and output channel is M times wider. In the case of SWS, by using $\sqrt{M}$ times wider network for the 1:M sparsity pattern, we can construct the SWS network, which has roughly the same `mult` count as the base dense network and SAS. More specifically, we adopt the following configuration: We use $\sqrt{M}$ times (instead of M times) wider input/output channel. In this configuration, we'll have approximately the same `mult` count as the base network. The network width needs to be an integer value, and it also needs to be a multiple of M. Hence, we use the weight having the shape of $\lfloor\sqrt{M}\bar{C}_i^{(l)}\rceil \times \lfloor\sqrt{M}C_o^{(l)}\rceil$ for the 1:M SWS network, where $\lfloor\cdot\rceil$ is a rounding operator, $\bar{C}_i^{(l)}$, $C_o^{(l)}$ is the input/output channel dimension of l-th layer of the base dense network. For the last chunk, when it does not equal to M, we use $(\lfloor\sqrt{M}\bar{C}_i^{(l)}\rceil\lfloor\sqrt{M}C_o^{(l)}\rceil$

mod M):1 sparse pattern. This way, we construct the SWS network having approximately the same `mult` count as the base dense network.

For example, consider the l-th layer of the base network consisting of convolution with $\bar{C}_i$=288 (C_i=32, kernel size k=3) and C_o=128. When M=8, then we have $\lfloor\sqrt{M}\bar{C}_i^{(l)}\rceil$=815 and $\lfloor\sqrt{M}C_o^{(l)}\rceil$=181. Then `mult` count of the original dense layer (for single pixel) is 288×64=18432, the `mult` count of the weight sparse layer (SWS) is 18440 (815×181/8=18439 with modulo 3, we use M=3 for the last chunk).

C Detail in Speed Benchmarking

On the wall-clock speed benchmarking reported in section 2.3, we adopt the opposite configuration as the neural network experiment. We evaluate the speed by changing the sparsity pattern while keeping the matrix dimension unchanged to report more concrete comparisons.

Specifically, in the wall-clock speed benchmarking in fig. 3, we consider the `matmul` **WX** where $\mathbf{X} \in \mathbb{R}^{\gamma\times\alpha}$ and $\mathbf{W} \in \mathbb{R}^{\beta\times\gamma}$ **which is same for dense `matmul`, SAS `matmul`, and SWS `matmul`**. The `mult` count of the three variants is the same ($\gamma\times\alpha\times\beta$) when one does not take the sparsity into account. By utilizing the 1:2 fine-grained (or semi-structured) sparsity on *Sparse Tensor Core*, the `mult` count becomes half for SAS and SWS, i.e., $(\gamma \times \alpha \times \beta)/2$. As shown in fig. 3, we use fixed γ=10240, changing $\alpha = \beta$ from 10240 to 20480, which is the same for dense `matmul` SAS `matmul`, and SWS `matmul`.

We want to emphasize that the scenario in this speed benchmarking (keep the same matrix dimension) is different from the scenario for neural networks (keep the (almost) same `mult` count). We adopt the different configurations to evaluate the speed in a fair setting between SWS and SAS. In the neural network setting, one could construct the SWS network having approximately the same `mult` count with the base one by using $\sqrt{M}$ wider input/output channel (supp. B); however, it is hard to align the `mult` count precisely, and the shape of the matrices is different, which makes it hard to evaluate the overhead specific to SAS which we are interested in (computation of index, reorder, and memory transfer for the index). On the other hand, we can clearly evaluate this by measuring the time of the `matmul` of the same-sized matrix (section 2.3).

D Detail in Main Experiment

The primary experimental setup is summarized in table A1. The FLOPS and memory footprint of the network used in the experiments are summarized in table A2.

For training the SWS network, we employed the method of (Zhou et al., 2021) instead of APEX's Automatic SParsity[1] because ①code base of (Zhou et al., 2021)[2] supports arbitrarily $1:M$ sparsity and ②it allows training from scratch which enables fair comparison with SAS.

Table A1: **Experimental setup**

	CIFAR-10	CIFAR-100	ImageNet
Network	ResNet18		ConvNeXt-B
Batch size	512		4096
Training epochs	$16/\alpha\times1000$		600
Optimizer	ERAdam (section 2.4)		AdamW
Scheduler	Two cycle cosine with kDecay=2.0 Zhang & Li (2020)		Cosine
Initialization	Kaiming-uniform He et al. (2015)		Truncated Gaussian
Base width α	4/8/16		2
Sparsity M	`ReLU`/2/4/8/16		`ReLU`/2

CIFAR-10/CIFAR-100

The code for CIFAR-10 and CIFAR-100 is based on Pytorch lightning CIFAR10 tutorial code[3]. We use a single A6000 GPU for CIFAR10 and CIFAR100 experiments. It takes a day to train the single model.

ImageNet

The code for ImageNet is based on ConvNeXt's (Liu et al., 2022) official code base[4]. We use four A100 GPUs (each holding 256 batches) with an update frequency of four to virtually construct the batch size of 4096. It takes about two weeks to train a single SAS model for 600 epochs (double the original 300 epochs to compensate for sparse gradient); we use the default setup of ConvNeXt's (Liu et al., 2022) official repository for training ImageNet-1K (without pre-training using ImageNet-22K), only changing the original dense `matmul` to our proposed SAS `matmul`(conv2D $\rightarrow$ SASconv2D, and linear $\rightarrow$ SASlinear) and training epochs ($300\rightarrow600$). We use the original AdamW optimizer to keep the original ConvNeXt's highly optimized settings intact as much as possible (furthermore, in this moderate sparsity of M=2, proposed ERAdam behaves almost identical to AdamW with warm-up). Refer to their paper for a more detailed setup.

Table A2: **FLOPS and memory footprints**. Note that FLOPS and the number of parameters and FLOPS of SWS are not precisely the same as the base dense network as discussed in supp. B, but the difference is less than 1% for all the configurations.

Network	FLOPS (all)	Params (dense, SWS)	Params (SAS)
ResNet18 (α=4)	114K	731K	731K$\times M$
ResNet18 (α=8)	28K	182K	182K$\times M$
ResNet18 (α=16)	7K	46K	46K$\times M$
ConvNeXt-B (α=2)	3850M	22M	22M$\times M$

[1] `http://github.com/NVIDIA/apex`
[2] `http://github.com/NM-sparsity/NM-sparsity`
[3] `http://lightning.ai/docs/pytorch/stable/notebooks/lightning_examples`
[4] `https://github.com/facebookresearch/ConvNeXt`

Training

During the training, SAS `matmul` is computed as follows: it first projects the dense/narrow activation map into a structurally sparse/wide space by $\mathcal{S}$, constructing the sparse activation explicitly; then, it performs the conventional dense `matmul`. Note that it is equivalent to the sparse `matmul` for efficient inference when hardware support is available, and we do not construct the sparse matrix explicitly during inference. Still, the hardware directly processes the narrow dense feature along with the index computed online (fig. 2). Refer to the pseudo-code in listing 2 during training. When implemented this way, the gradient for the wide weight $\tilde{\boldsymbol{W}}$ could be computed using *autograd* mechanism.

```
1 def SAS_proj(x, m): # m corresponds to log2(M) in the main text
2     B, C, H, W = x.shape
3     xa = [torch.roll(x[:, None], i, dims=1) for i in range(m)]
4     ind = torch.cat([2**i*(torch.signbit(x_)) for i, x_ in enumerate(xa)], dim=2).sum(dim=2, keepdim=True)
5     x_sparse = torch.zeros([B, 2^m, C, H, W]).scatter_(1, ind, x[:, None]).view([B, (2^m)*C, H, W])
6     return x_sparse
7 def forward(self, x): # x: input activation, m: sparsity factor (actual sparsity is 100(1−1/2^m)[%])
8   return F.conv2d(SAS_proj(x, m), self.weight, bias=self.bias, stride=....)
```

Listing 2: Code of SAS `conv2d` layer for training (PyTorch). Note: During inference, one does not need to construct the sparse activation explicitly (L5); refer to fig. 2 for efficient infrence mechanism.

E Memory Arrangement for `cusparseLtMatmul`

The *Sparse Tensor Core* and the `cuSPARSELt` library were originally developed to speed up the DNN having structured weight sparsity (SWS). Our SAS improved the accuracy/computation tradeoff by using **the same hardware and software library**. In the case of SWS, the index could be precomputed after training using `cusparseLtSpMMACompress` function of `cuSPARSELt` library. Figure A2 illustrates the memory arrangement of value and index for `cusparseLtMatmul` operation for NVIDIA's *Sparse Tensor Core*. The value matrix follows the index matrix. In the case of a 1:2 sparse pattern (TF32), the index is stored using 4-bit (although it can be represented in 1-bit; *Sparse Tensor Core*use 4-bit to index a 1:2 pattern), and '0x4' and '0xe' is assigned for indexing the first and second element, respectively.

The important note for *Sparse Tensor Core* is that the order of the index is not aligned with its corresponding activation value; it needs to be reordered (L8 in listing 1), and the reordered index $\tilde{\mathcal{I}}$ needs to be supplied to the core to get the correct result. The arrangement depends on the size of activation matrices X (The memory arrangement in fig. A2 illustrates the specific case when the input matrix is 64×32).

How to use the `cusparseLtMatmul` for SAS?

One can also use the `cusparseLtSpMMACompress` function for SWS to executed the SAS `matmul`, ①compute the index $\mathcal{I}$ using equation 3, then ②explicitly computing the sparse activation $\tilde{\boldsymbol{X}}$ and finally ③compute the reordered index $\tilde{\mathcal{I}}$ using `cusparseLtSpMMACompress`. However, it is redundant and inefficient. We already have compressed activation $\boldsymbol{X}$ as an output from the previous layer; the index $\mathcal{I}$ is computed cheaply. We want to reorder the index $\mathcal{I}$ to get $\tilde{\mathcal{I}}$ without explicit construction of sparse activation $\tilde{\boldsymbol{X}}$ as we discussed in the main text (section 2.3). The problem is that NVIDIA does not provide information about the rendering mechanism of `cuSPARSELt`.

Tools for digging up the reordering matrix

We developed a helper tool of `cuSAS`[5] for finding out the rendering matrix $\boldsymbol{O}$ for arbitrary size matrix to realize a general `matmul` for SAS activation. The core idea for realizing the elucidation is *impulse response* of `cusparseLtSpMMACompress` function. Specifically, we input the sparse activation $\tilde{\boldsymbol{X}}$; all the odd element has a non-zero value except the (i, j) element. Then, looking at the reordered index computed by `cusparseLtSpMMACompress`, we can find the destination index $(\tilde{i}, \tilde{j})$ where (i, j) element in the original index should be warped. Repeating this process for all the row-column pairs, we get the reordering matrix $\boldsymbol{O}$ such that $\tilde{\mathcal{I}} = \mathcal{I}[\boldsymbol{O}]$. The reordering matrix $\boldsymbol{O}$ depends only on the size of $\boldsymbol{X}$; therefore could be precomputed. We'll also open-source this tool.

It is possible to implement the CUDA kernel, which runs the following operation at once for more efficient SAS `matmul`; ①checks the sign bit, ②assign either '0x4' or '0xe', and ③reorder.

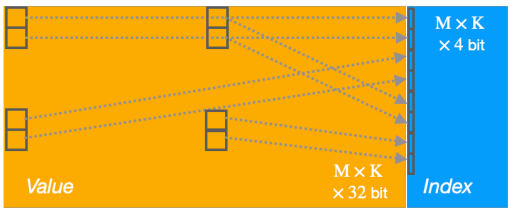

Figure A2: **Memory arrangement for `cusparseLtMatmul`.** The index is located just after the value. The index needs to be arranged for the execution of sparse `matmul` using `cusparseLtMatmul`. The formatting of the index depends on the size of the matrix X in listing 1. This figure illustrate the case for the 1:2 sparse pattern (TF32) and the input matrix size is 32×64.

[5]The current version supports sparse activation with a 1:2 structured pattern.

F Fine-grained (semi) Structured Weight Sparsity.

NVIDIA's SWS (NVIDIA, 2020) could speed up the `matmul` with weight having moderate rate sparsity (e.g., 50%) on GPU, which is almost impossible for the unstructured sparse pattern (section 5). They realized actual speed up by utilizing a specific pattern in their sparsity, namely $N : M$ structured sparsity. Suppose a typical matrix multiplication between activation $\boldsymbol{X} \in \mathbb{R}^{16\times 32}$ and weight $\boldsymbol{W} \in \mathbb{R}^{32\times 8}$. The *Dense Tensor Cores* implement this `matmul` by two cycles. In contrast, the Sparse Tensor Core only needs one cycle if the weight tensor $\boldsymbol{W}$ satisfies the structured sparse pattern (fig. A3).

Our SAS could utilize the same hardware by the novel structured sparse projection mechanism. With the same computational budget and on the same hardware, SAS realizes better accuracy if one can use extra memory for storing the weight.

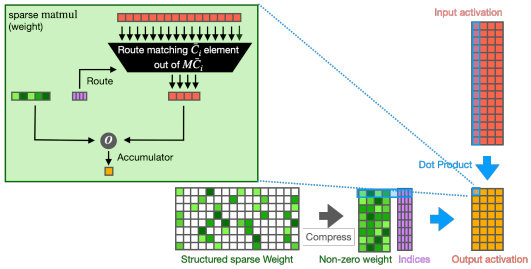

Figure A3: **SWS `matmul` on Sparse Tensor Core** NVIDIA (2020). Compare with our SAS `matmul` mechanism in fig. 2.